Date of acceptance Grade

Instructor

Geometric Approaches to Big Data Modeling and Performance Prediction

Peter Goetsch

Helsinki June 6, 2018

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Laitos — Institution — Department Department of Computer Science

Tekijä — Författare — Author
Peter Goetsch

Työn nimi — Arbetets titel — Title
Geometric Approaches to Big Data Modeling and Performance Prediction

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year June 6, 2018	Sivumäärä — Sidoantal — Number of pages 52 pages + 0 appendices

Tiivistelmä — Referat — Abstract

Big Data frameworks (e.g., Spark) have many configuration parameters, such as memory size, CPU allocation, and the number of nodes (parallelism). Regular users and even expert administrators struggle to understand the relationship between different parameter configurations and the overall performance of the system. In this work, we address this challenge by proposing a performance prediction framework to build performance models with varied configurable parameters on Spark. We take inspiration from the field of Computational Geometry to construct a *d*-dimensional mesh using Delaunay Triangulation over a selected set of features. From this mesh, we predict execution time for unknown feature configurations. To minimize the time and resources spent in building a model, we propose an adaptive sampling technique to allow us to collect as few training points as required. Our evaluation on a cluster of computers using several workloads shows that our prediction error is lower than the state-of-art methods while having fewer samples to train.

Avainsanat — Nyckelord — Keywords
Big Data, Spark, Performance Modeling, Computational Geometry

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Today Big Data is ubiquitous in our daily lives. Big Data influences everything from the daily weather forecast and our route to work, to the food in the grocery store and our favorite sports game. We constantly interact with Big Data, whether we notice it or not. Driving this growth of Big Data over time has been the rise of Cloud computing which has enabled low-cost computation, storage, and network for the masses. Many frameworks have been introduced to organize fault-tolerant Big Data computation in a distributed, fault-prone environment like the Cloud. Frameworks such as Spark [ZCF+10], MapReduce [DG04] and Hadoop [Whi09] are among the leaders of general-purpose Big Data frameworks which are capable of handling many Big Data workloads such as Machine Learning, Streaming, Graphs, SQL, etc. These frameworks have gained popularity because they have been designed from the ground-up for execution in clustered, fault-tolerant environments like the Cloud.

In Cloud environments where resources are billed down to the second, it is crucial that applications run efficiently. This is particularly true in the case of Big Data frameworks, which are by their nature resource intensive. However, this is easier said than done. Big Data frameworks are complicated pieces of software, requiring deep knowledge of distributed systems, databases, hardware and programming to understand fully. For the end-users submitting the jobs, which are most-often not systems engineers, the task of tuning a Big Data job can be daunting. Big Data frameworks have hundreds of configuration parameters which are often left to the default setting, leading to wasted time and resources. In Chapter 3 we run a series of benchmarks against various Spark workloads to demonstrate the complex relationships between some key Spark parameters, namely parallelism, memory allocation, and CPU core allocation. For just these three tuning parameters, we demonstrate the dynamic relationships present between them. In some newer Big Data systems like Spark, building an accurate model that takes into account all the tuning parameters is extremely challenging. Our benchmarking results reinforce the complexity of Big Data systems, reiterating the need of performance tuning in these systems. The process of determining which parameters are relevant for a particular job is very crucial, yet very difficult for non-technical users. Compounding things is the fact that parameters which are good for one job may be awful for another seemingly similar job.

To address this challenge, many tuning frameworks have been proposed for Big Data systems. To create a tuning framework, one first must create a model of the underlying system that can predict performance under a wide range of hypothetical workloads. Models of systems can be classified either as *white-box* or *black-box.* In white-box models, the inputs of the system are related to the output(s) by trying to understand how the individual pieces in a system impact its input and output. On the contrary, black-box models do not try to understand the internals of a system, but rather search for a blind mathematical relation between input and output. Black-box models often employ Machine Learning techniques, but it is not a rule. In practice however, models are not purely white-box or black-box. On top of a model, a tuning framework can be built which selects runtime configuration parameters to optimize a user-defined metric. A fast runtime is a potential goal for some tuning situations, while others might want to optimize data locality or minimize total resource usage.

For MapReduce, we have seen Starfish [HLL+11] and MRTuner [SZL+14], which offer unique approaches to optimizing the MapReduce jobs. For the newer Spark, we have seen the black-box Ernest [VYF+16] model and some other white-box approaches (e.g. [WK15, SS17]). In general, we observe that MapReduce-type frameworks have more white-box models because MapReduce has a simple internal structure that lends itself well to white-box approximations. For systems like Spark, which are more complicated internally, many black-box models have been proposed. These are all novel approaches, however, they are either (1) constrained by the number of features available in the model, (2) inaccurate for common use cases, or (3) requiring high amounts of training data to achieve good results.

In this Thesis, we provide an answer to the research problem of predicting runtime for Big Data jobs at various runtime configurations, given some samples of past system behavior. To evaluate the model in our experiments, we introduce a metric called the Mean Absolute Percentage Error (MAPE), which is the average of the Percent Errors for a set of estimated runtimes ($\hat{T}$) and actual runtimes (T):

$$MAPE = \frac{100\%}{l} \sum_{i=1}^{l} \left|\frac{T_i - \hat{T}_i}{T_i}\right| \tag{1}$$

where l is the number of specific feature configurations to be tested.

We search for a prediction model with a minimized Mean Absolute Percentage Error for the space parameterized by the model. From an accurate prediction model, accurate tuning suggestions can be developed. We model job features against runtime, forming a mesh over them, from which we may interpolate runtime predictions for unknown feature configurations. We introduce the Delaunay Triangulation [Del34] for creating the mesh, and an iterative sampling technique, Adaptive Sampling, for optimally picking samples for the model. Our model works with an arbitrary amount of features, forming a mesh in any number of dimensions. The Delaunay Triangulation is one of many possibilities borrowed from the field of Computational Geometry for making the mesh covering of the feature space. Paired with the Delaunay Triangulation, we propose Adaptive Sampling, which works to iteratively improve the model by optimally selecting samples based upon their estimated *utility* to the model. Our model works well over other traditional black-box techniques such as Linear Regression [Nas07] and the Gaussian Process [Ras04] because it requires fewer samples to achieve a high prediction accuracy, and also works in an arbitrary number of dimensions. Furthermore, our Delaunay Triangulation model is generic enough that it may be applied to not only other Big Data frameworks (MapReduce, Hadoop, etc.) but also general black-box systems with large inputs and complex internals. In summary, the main contributions of this Thesis are as follows:

1. We propose a framework to model the performance of Big Data applications using Delaunay Triangulation by constructing a mesh over a set of selected features, which enables the prediction of the whole configuration space.

2. We propose an adaptive sampling technique to discover the areas of abrupt change in the performance space, which consequently leads to a small prediction error and minimizes the time and resources cost.

3. The Delaunay Triangulation model and the adaptive sampling algorithms are implemented and comprehensively evaluated on the Spark platform across multiple workloads. Our evaluation result shows that the Delaunay Triangulation model outperforms the state-of-the-art methods in terms of prediction accuracy and sample data size.

The layout of this Thesis is as follows. In Chapter 2 we introduce Cloud computing and its relevance to today's Big Data frameworks, as well as an overview of the current Big Data frameworks, with an emphasis on Spark. In Chapter 3, we deep-dive into Spark and benchmark its performance characteristics against various workloads. In Chapter 4 we perform a survey of the current literature on performance tuning and modeling for Big Data frameworks like Spark and MapReduce. In Chapter 5 we present the main design and theory behind our Delaunay Triangulation model supported by adaptive sampling. In Chapter 6 we thoroughly evaluate the presented model, followed by a discussion and finally, concluding remarks in Chapter 7.

2 The Big Data Landscape

The term "Big Data" has been around since at least the 1990s, when it was first used in [Mas99] and [WI98] (among others). In the early 2000s, it became clear that "Big Data" was an emerging field, and efforts were made to define it formally. It has been widely accepted across academia and industry that Big Data is data which can be characterized by three Vs: volume, velocity, and variety [Lan01]. Volume refers to the sheer volume of data coming into systems today, which is ever-growing in size. Velocity refers to the speed at which the data is traveling through today's systems, whereby it increasingly comes from streaming sources where it can only be read once. Finally, variety refers to the many types of data that systems today are having to process, for example not only highly-structured relational data but also unstructured data sources like images, audio, and biometric data. This "3V" approach to Big Data first originated in [Lan01], and has been extended upon over the years to include a fourth "V" of "Value," "Veracity" and others. Value in this context indicates the value potential that can be unlocked from Big Data [HYA+15], whereas "Veracity" indicates the uncertainty of incoming data which must be taken into account [WB13]. Still others have avoided the nV approaches to Big Data and instead offered more general definitions. In a cornerstone 2011 report, the McKinsey Global Institute defined Big Data as "datasets whose size is beyond the ability of typical database software tools to capture, store, manage, and analyze" [MCB+11]. Other similar definitions exist. While there are many definitions to Big Data, even more than 20 years after its birth, they all approach on the same idea. It is important to note that due to its nature, Big Data has primarily been an industry-driven research field, swapping advancements with traditional academia over the years. This tight collaboration has lead to many innovations but has also blurred the lines in this field between traditional academics and industrial applications.

2.1 Cloud Computing

The rise of Big Data can be directly attributed to the rise of Cloud computing in the last two decades. Cloud computing is a huge topic itself, but for our needs, it is sufficient to define it as "the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services" [AFG+10]. The rise of Cloud computing has transformed computation (and the associated network and storage components) into a utility service model in which (1) users have access to seemingly infinite resources, (2) users require no upfront commitment to access these resources, and (3) users only pay for the resources that they use. Cloud computing allows individuals and organizations to access practically infinite compute resources at economical costs. Because of these benefits, Cloud computing usage has experienced high growth rates. This adoption has created a feedback loop: users come to the Cloud because it is cheap, and when more users come, Cloud vendors are able to use the economies of scale to make things cheaper. This feedback loop continues to drive down costs. Inexpensive computation, storage, and network resources have been the driver of Big Data growth. In past times, an organization would have to buy and provision hardware themselves, then supply it with enough workloads to achieve a high rate of utilization. Today the Cloud distributes the up-front and operational costs, leaving the user only to pay for what they use. This fundamental transformation in the way users perform computation has lead to huge growth in Big Data.

2.2 MapReduce

The rise of Cloud computing allows end-users to focus on extracting value from their data, rather than managing infrastructure. From this starting point, the field of Big Data has been able to grow exponentially. It can be argued that Big Data began to gain widespread popularity in industry and academia around the time that Google released papers in the early 2000s on their distributed file system (GFS) and MapReduce systems [GGL03, DG04]. These papers spurred the creation of the Hadoop framework, which includes open-source versions of a distributed file system, and MapReduce. MapReduce's simple programming model can be used to implement many applications. The simple flow of `Map` → `Shuffle` → `Reduce` is easy to understand and makes building systems based upon it almost trivial. Because of this, MapReduce gained widespread adoption in practice throughout the mid-2000s and early 2010s.

The rise of MapReduce brought with it much interest and funding for its research during this same time [HAG$^+$16]. One important takeaway from all this research is the limitations of MapReduce. One of the most significant weaknesses of MapReduce is its poor performance with algorithms that run in iterations, whose results build upon previous results [DN14]. Algorithms like Pagerank, K-means, and simple regressions have poor performance in MapReduce because they require multiple iterations to converge, with each iteration being a complete MapReduce process with disk reads and writes on each end. At the end of each MapReduce iteration, results are written back to disk, adding significant overhead over the course of many iterations. Other well-documented weaknesses of MapReduce are (1) its lack of support for stream, interactive and real-time processing, (2) its high bisectional traffic generation within a cluster and (3) its inefficiencies in load balancing [DN14, HAG$^+$16]. Naturally, systems have been developed to patch these weaknesses of MapReduce, however, there is only so much that can be done to fix a flawed system. MapReduce is an excellent tool for what it was designed for. Its simple programming interface and ability to handle most problems across an enterprise have helped it to become famous. MapReduce is not the fastest, but it is easy to understand and much can be solved with it, so its popularity has remained.

For the general enterprise that simply needs a data processing engine, MapReduce is a great solution. For other environments that aren't the general use case, another solution is needed. As Cloud computing has become more ubiquitous in organizations, the costs of MapReduce's inefficiencies have become more visible due to the pay-per-use nature of Cloud computing. Furthermore, in the time since MapReduce was introduced in circa 2000, organizations have come to expect more from their data processing frameworks that were not even on the horizon during MapReduce's development. Enterprises today put more emphasis on real-time (stream) and interactive processing of their data, rather than the offline approach of MapReduce. Enterprises also are handling more graph-based data, which is often iterative in nature and suffers from the drawbacks mentioned above. Above all, enterprises need a fast and efficient data processing framework. In earlier days, enterprises with their own hardware could run MapReduce, and it did not matter if the job took longer because they had already bought-in. Today when Cloud resources are billed to the second, the inefficiencies in frameworks like MapReduce show. For these reasons, a new generation of Big Data processing frameworks has been introduced. Some frameworks attempt to solve one problem well, for example Apache Kafka[1] and Flink[2] for handling streams and Apache Giraph[3] for graph processing. While others have tried to solve the general problem of creating a Big Data processing framework which addresses the inefficiency of MapReduce and also handles graph processing, streams processing and more. One such framework that is leading in this area is Spark [ZCF$^+$10], and is what forms the backbone of this thesis.

2.3 Spark

Apache Spark is a general-purpose Big Data cluster computing engine that was first introduced around 2010 by researchers from UC-Berkeley [ZCF$^+$10, ZCD$^+$12]. Spark was created to make iterative data processing algorithms run faster, but it performs well in most other cases too. It is built around a distributed memory abstraction called Resilient Distributed Datasets (RDDs) [ZCD$^+$12], which will shortly be described in detail. Spark outperforms MapReduce in almost all benchmarks [SQM$^+$15]. All programming problems that can be solved with MapReduce are able to be solved with Spark. Spark's advantage is that it can solve much more; the Map-Reduce pattern is just one of many patterns supported in Spark. Spark accomplishes all of this in a distributed and fault-tolerant fashion.

Resilient Distributed Datasets (RDDs) are the central idea to which Spark is built upon, and their understanding is crucial. RDDs represent large data sets in-memory in a fault-tolerant fashion [ZCD$^+$12]. RDDs are read-only data abstractions: once a RDD has been created, users can transform it ("write") into another RDD by performing transformations on it like `filter`, `join` and `reduceByKey`. Furthermore, once created, users can themselves decide upon how to handle caching of RDDs (called *persistence*) into memory (on-heap or off-heap) and disk with various combinations and serialization options.

The key idea to understand about RDDs is that they store the steps for how to compute a dataset, not necessarily the data set itself. This "lineage" is represented internally as a directed acyclic graph (DAG) [ZCD$^+$12]. Data sets are only constructed fully in memory when they are absolutely needed in the program, otherwise,

[1]`https://kafka.apache.org/`
[2]`https://flink.apache.org/`
[3]`https://giraph.apache.org/`

they are kept in the lineage state. Representing RDDs in this fashion makes them resilient to faults, and can also preserve memory in the system because they only consume memory when absolutely required by the application. RDDs are a similar concept to distributed shared memory (DSM) [NL91], however RDDs use coarse-grained updates (e.g. "apply that function to each item in this dataset"), whereas DSM operates at a much finer-grained level (e.g. "update value at `0xc81cf6b2`"). It may not even be fair to compare RDDs and DSM directly because they are memory abstractions created for different purposes, at different times. RDDs are read-only and used in applications which apply transformations in bulk, whereas DSM is both read and write and is used in applications with fine-grained, random memory access patterns. Since DSM is the closest comparison we have today to RDDs, making the comparison is hard to avoid.

The primary benefit that RDDs have over DSM is the high fault tolerance, and consequently, by utilizing RDDs, Spark achieves a high level of fault-tolerance. One significant reason why is because RDDs do not have to duplicate or checkpoint data because any lost partition of an RDD can be recomputed on-the-fly from its lineage. In addition, the lazy-evaluation of RDDs means that if a node fails, it has a smaller probability of having the RDD data actually evaluated and written into memory. Finally, because of the immutable nature of RDDs, lost partitions can quickly be recovered more quickly than if there was some state in them to preserve.

By representing data sets in-memory in an efficient and fault-tolerant manner, Spark programs achieve high performance across many different kinds of workloads. In particular, the usage of RDDs has enabled Spark to handle iterative workloads very well. For iterative algorithms, iterations are implemented as transformations of existing RDDs. In contrast to MapReduce, another iteration of the algorithm does not necessitate a read-write cycle from the disk. Another aspect of Spark that makes it fast is its fine-grain control given to developers of data partitioning. Application developers can use context about the application to distribute RDD partitions across nodes in a cluster, thereby reducing overall cross-traffic between nodes as well as speeding up RDD recovery in the case of a node failure. A final aspect of Spark which makes it fast is the immutable nature of RDDs. With immutability, there is much less state to keep track of in each partition, which often enables partitions to be computed in parallel.

Resilient Distributed Datasets form the foundation of Spark, upon which all the other tooling is written. From an architectural point of view, a Spark application

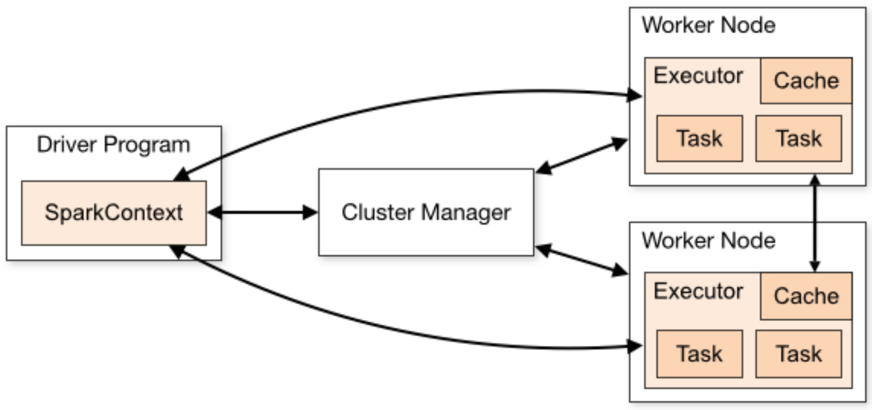

Figure 1: Coordination of job tasks in Spark [4]

consists of a driver program and any number of worker nodes. The driver program runs the main application code and manages workers in a cluster, who actually perform the computation tasks. Each worker contains one or more executors which further divide-up the computation tasks within each worker. Refer to Table 1 for summary of these components and Figure 1 for a visual reference of the overall Spark architecture.

At the center of each Spark program is a cluster resource manager such as Apache Mesos or Hadoop's YARN [HKZ+11, VMD+13]. Spark also offers options to run in "standalone" mode, but for this work, we will only focus on cluster mode because the standalone mode is not a realistic runtime for production workloads.

Spark exposes an application library in a number of conventional programming languages and runtimes, forming the core platform of Spark. On top of this core platform sit official modules that provide extra functionality. The first module is Spark SQL and DataFrames, which is an extension to Spark which exposes a SQL-like query interface for programs to run against RDDs. For data that is more structured in nature, Spark SQL is used to interactively query against it, as if it were in a vast SQL database. On top of Spark SQL is Spark Streaming, which provides a way to run computation on streaming data. Also on top of Spark SQL is MLlib,

[4]`https://spark.apache.org/docs/latest/img/cluster-overview.png`

Table 1: Spark Terminology

Term	Definition
Executor	A process that manages the execution of tasks. Remains alive for the duration of the job.
Task	A self-contained unit of computation taken from the overall job.
Worker	Any node capable of hosting ≥ 1 executor processes.
Driver	Main controller for the job; coordinates with executors to manage job execution.

a machine learning toolbox containing many common machine learning algorithms, as well as some common patterns used in machine learning scripts. Finally, there is GraphX, an extension to handle graph-based data more efficiently. As we have explored earlier, there presently exists frameworks dedicated to handling each of these components separately. The benefit of using Spark here is that once the data has been expressed as an RDD, it can be passed around from Spark SQL to MLlib to GraphX without unloading to disk. The drawback, of course, is that the solutions designed from the ground-up to solve a specific problem (e.g., Apache Kafka for streaming workloads or Giraph for graphs) often perform better than a framework like Spark which was designed to solve a general problem.

In comparison to Hadoop's MapReduce, Spark outperforms it in almost all benchmarks. Spark in particular shows dominance in iterative algorithms (K-Means, regressions, etc., exactly what it was designed for), with up to 20x speedups over MapReduce [ZCF+10, SQM+15]. The speedups can be attributed to (1) the earlier discussed RDD design model which reduces overall I/O in the system and (2) Spark's smaller initialization and termination time, which has a considerable influence on "small" workloads. It has however been shown that for some large shuffle-heavy workloads (such as Sort), MapReduce outperforms Spark due to its superior reduce-side shuffler design [SQM+15], it can be argued that comparing Spark to MapReduce directly is not a fair comparison because the systems were created years apart to address different requirements. MapReduce was primarily designed for single-pass analytics algorithms, whereas Spark was designed for iterative and interactive algorithms.

As we have seen, Spark performs very well and can solve more problems compared

Table 2: Key Spark Configuration Parameters

Executor Parameter	Configuration Parameter
Count	`spark.executor.instances`
Executor Memory	`spark.executor.memory`
Executor Threads	`spark.executor.cores`
Our Parameter	**Configuration Parameter**
Memory	`spark.executor.instances` × `spark.executor.memory`
CPU Cores	`spark.executor.instances` × `spark.executor.cores`

to its predecessors. This speedup can primarily be attributed to the use of Resilient Distributed Datasets as the intermediate storage used during computation.

3 Spark Benchmarking in a Clustered Environment

Up to this point, Spark has been discussed at a theoretical level. Because the goal of this thesis is to build a system which is practical in nature, some work should be done to understand the practical side of Spark. To dig deeper into the internals of Spark, some benchmarks have been performed, with each benchmark trying to expose a different characteristic in Spark. Because there is presently little literature on this topic, the benchmarks here were run ourselves. From these benchmarks, we can gain some key insights which will be applied later for the main contribution of this work.

Spark is released with a few hundred configuration parameters to each job that can be tuned by the user[5], many of which have an impact on the job's performance. Recall from earlier that the Spark driver program initializes executors, and each executor can run multiple tasks, each task being responsible for computing some portion of the overall job. In Spark, the method we use to vary the amount of memory and CPU cores given to each job is to tune (1) the number of executors using `spark.executor.instances`, (2) the number of parallel tasks per executor using `spark.executor.cores`, and (3) the amount of memory per executor using `spark.executor.memory`. From these parameters, the total amount of resources

[5]`https://spark.apache.org/docs/latest/configuration.html#spark-properties`

(memory and CPU cores) assigned to a single job[6] can be easily calculated. Both the Spark configurations and the total resource calculations have been summarized in Table 2.

Jobs were run in the University of Helsinki's Ukko high performance computing cluster[7] on 1-3 nodes, each node exposing 16 virtual-cores (vcores) and 64 GB of virtual memory (vmem). Spark 2.1.0 was run on top of HDFS (utilizing an NFS underneath) and along with YARN as the container manager. For each round of tests, we ran a collection of different workloads provided by the HiBench Big Data benchmarking library, which provides a collection of standardized workload implementations [HHD+10]. In particular, we ran the following workloads: Terasort, WordCount, K-Means Clustering, Bayesian Classification, Pagerank, and SQL-Join. These specific workloads were picked because they display a wide variety of resource usage (e.g. memory, CPU, disk I/O, etc.). More space is devoted later to describe the testing environment and workload characteristics in detail.

3.1 Executor Memory

The first benchmark performed was to see how memory impacts runtime. The results can be found in Figure 2. We observe a binary behavior among the algorithms. Firstly, we observe that memory has no influence on the runtime for algorithms that make a single pass at the data like WordCount, Terasort, and SQL join. These algorithms require a base-level of memory to operate, and adding more memory has no impact on the overall performance. It is interesting also to note that the WordCount and Terasort algorithms perform at the same speed, even with less memory than the size of the data set. This indicates that for this class of algorithms, Spark operates on them without pulling them entirely into memory as an RDD at once. Secondly, we observe for the iterative algorithms (PageRank, K-Means, Bayes), that there is an exponential relationship between runtime and memory allocation. As more memory is given to each executor, both algorithms' runtimes drop exponentially, until they reach a point (around 17GB/executor for PageRank and 7GB/executor for K-Means) where adding more memory has no impact. What this indicates is

[6] The total job resources are often written using the notation $nvmg$ (for example 10v40g) where n and m are integers denoting the total cores and memory given to the job, respectively.

[7] `https://www.cs.helsinki.fi/en/compfac/high-performance-cluster-ukko`

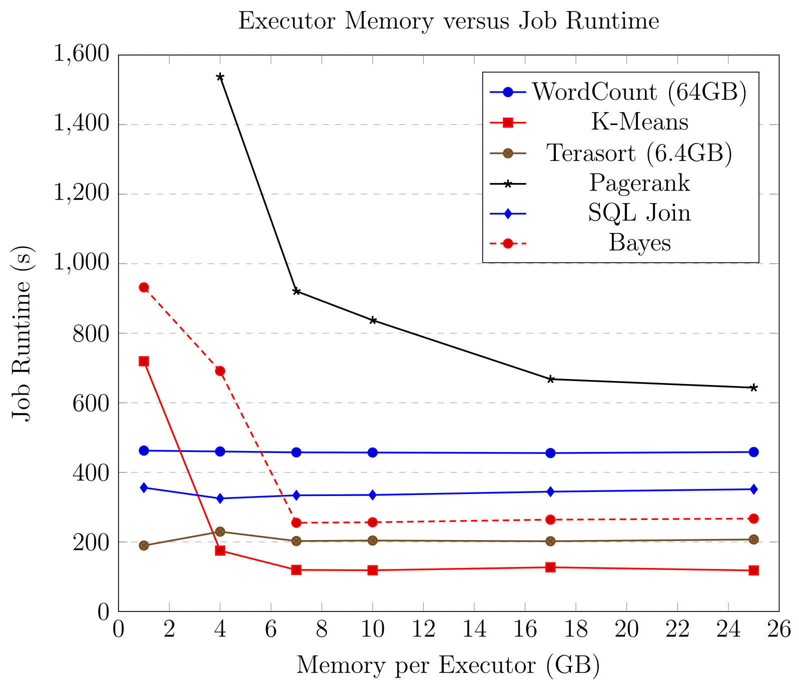

Figure 2: Benchmarking impact of allocated memory on job performance against common Big Data workloads on Spark.

that for iterative algorithms, their runtime is relatively high until they have been given enough memory to store the intermediate results completely in memory. Even if some fraction of the results do not fit into memory, the results indicate that having to perform I/O to disk for this portion with each iteration is very costly for the job's performance.

3.2 Executor Cores

The second benchmark performed was to see how the number of cores assigned to each executor impacts runtime. The results can be found in Figure 3. What we observe for all algorithms is that there is also an exponential relationship between the number of cores given to a job and its runtime. The results suggest that there is a baseline amount of cores required by each Spark job, and once that threshold has been passed, the benefits of adding more cores are reduced. Running a Spark job on only one core is slowest, whereas adding just two or three more cores can significantly improve the performance.

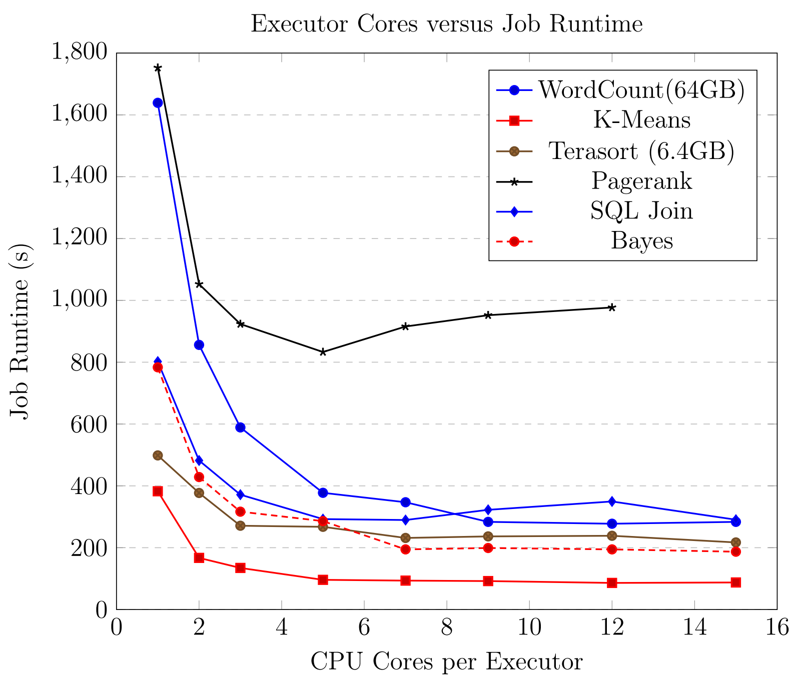

Figure 3: Benchmarking impact of CPU cores on job performance against common Big Data workloads on Spark.

3.3 Executor Parallelism

The third benchmark performed was to see how the worker (container) size impacts runtime. In this benchmark, we fixed the total resources given to a job at 12 vcores and 24GB memory but varied how many executors were created. The results can be found in Figure 4. What we observe is that for all algorithms, the container size has no impact on the job's performance. The benchmark shows that no matter how the total resources are divided into pieces, the job is still getting the same amount in total. We would expect to have perhaps slightly higher runtimes as the job becomes more parallelized because of the extra overhead of coordinating all the executors. This was not observed except for the Pagerank benchmark. This indicates either that most jobs require >6 parallel executors before they face this issue, or that it is not a valid concern.

3.4 Complete Topography

Our final benchmark is not a benchmark among workloads, but rather a survey of a complete performance surface for the K-Means workload in three dimensions (CPU

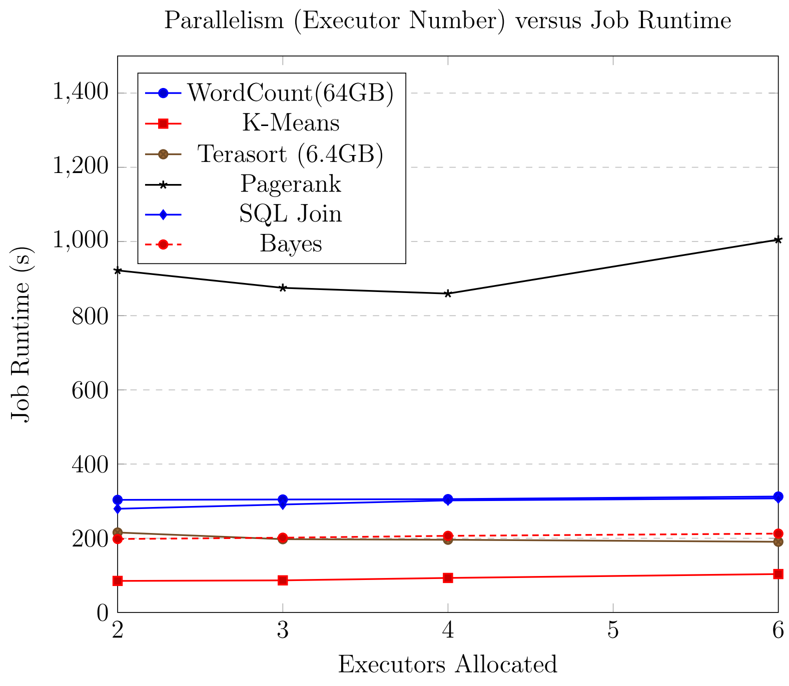

Figure 4: Benchmarking parallelism (number of executors) on job performance against common Big Data workloads on Spark.

Cores, Memory, and Runtime). Refer to Figure 5. The surface is a combination of the Executor Core and Executor Memory benchmarks for only the K-Means workload. When combined, we see how two features can join to form a surface. We display the results in three dimensions because it is easy to illustrate, but it is important to remember that a surface like this can be formed in the same way in even higher dimensions. For the K-Means surface, we observe a maximum in runtimes when approaching $\langle 0, 0 \rangle$, and an exponential decrease in runtimes when radiating away from $\langle 0, 0 \rangle$. Note that the runtime drops faster for the axis of Memory than the axis of CPU Cores. It is typical for other Spark workloads to display this asymmetry, though we have not included other results here.

3.5 Conclusions

The benchmarks that we have run highlight a few fundamental properties about the performance of Spark jobs. Firstly, for the executor memory and executor core benchmarks, we can identify points in the graph at which the benefit of adding more resources is diminished. We call these points *turning points*. They are essential in Spark applications because they separate the graph into two regions: one region

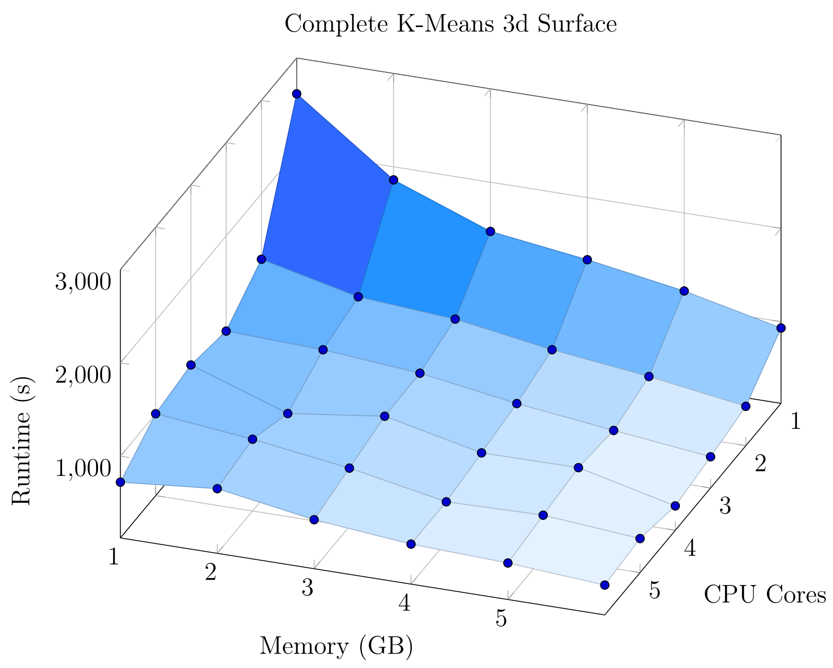

Figure 5: K-Means surface in three dimensions (Memory, CPU cores and runtime).

where adding more resources dramatically reduces the runtime, and another where adding further resources reduces the runtime only barely.

The second thing to take away from these benchmarks is the complex relationship that exists in Spark between memory, CPU and other resources. We observe a push-pull relationship between resources. For example, after a point, giving more memory to a job does not matter because then CPU or I/O becomes the bottleneck for performance. This idea is not trivial. Naively, one would think the way to speed up a memory-bound job is to simply give it more memory. Giving more memory will speed it up, but only along with additional amounts of CPU and other resources. By varying a single resource and holding the others constant, we see from the benchmarks that runtime performance does not improve. From this, we see just how interconnected resources are in Spark jobs.

A final takeaway from these benchmarks is the difficulties of doing experiments in a non-virtualized, fault-prone, multi-tenant environment like Ukko[8]. It was discovered that if there are idle resources, a YARN container will try to use them. This is particularly true for CPUs and can skew experiment results. Furthermore,

[8]`https://www.cs.helsinki.fi/en/compfac/high-performance-cluster-ukko`

in practice, it is quite difficult to say how much resources a job used accurately. With CPUs for example, there is hyperthreading happening at the processor level, followed by an abstraction of actual cores to virtual-cores at the resource manager (YARN) level, and then at runtime, there are potentially multiple users competing for these resources. When this happens, there is no way for Spark or any other layer to definitively know how much resources they have used or had access to during some period. Each layer has an idea, but with so many layers of resource abstractions, this idea becomes meaningless at some point. One way to address this problem is to run many trials of each test, and then it becomes easy to pick out the trials in which another user was draining resources on the node. Another way to address this might be to implement virtualization on Ukko for all users, which would have much stronger resource control and monitoring. A final solution might be to enable Control Groups[9] (`cgroups`) across all Ukko nodes, which is a kernel-level resource manager.

These baselines have focused on three trivial parameters to Spark jobs: container size, virtual memory, and virtual-cores. Future work might look at other aspects such as I/O tuning and serialization to see how they impact the job's performance. From these benchmarks, we have gained critical insights into Spark that will allow us to look seriously at the question of modeling and performance tuning Spark.

4 Related Work

When examining literature on Spark performance prediction and tuning, we can first look back to works on MapReduce/Hadoop because they are *similar enough* to Spark, where some techniques are still relevant. In terms of research, Spark is a new system built upon the same primitives as MapReduce/Hadoop. Therefore, when performing research on a system like Spark, we should not ignore the previous works in MapReduce/Hadoop. There are two previous works from the MapReduce/Hadoop literature which we will consider here: Starfish and MRTuner. Both approach the problem of job optimization from different directions. Starfish is an optimizer for MapReduce built to integrate tightly into the existing Hadoop ecosystem [HLL$^+$11]. Starfish takes both a macro and a micro approach to tuning Hadoop. At

[9]`https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`

the micro level, Starfish optimizes tuning parameters on the job-level. It uses a technique called Dynamic Instrumentation [CSL+04] to analyze the individual Map and Reduce programs submitted by the user. By using Dynamic Instrumentation, the instrumentation does not slow the original program. From this analysis, Starfish's just-in-time optimizer can build a job profile capturing (1) the job's timing of each sub-stage, (2) an idea of the data flow, and (3) resource usage throughout the job's lifetime. From this job profile, Starfish has a good idea of the overall job behavior and can suggest optimal parameters to the job.

Starfish not only considers performance at the job and sub-job level but also at the higher *workflow* and *workload* levels. The main idea is to not only tune single jobs but rather collections of jobs. In large organizations with many jobs executing daily, there are significant gains to be had by looking at the overall picture. Starfish can, for example, reorganize the job order and data placement to achieve higher data locality over the course of many jobs. By examining both micro and macro optimizations for MapReduce, Starfish achieves significant performance gains than the default settings for MapReduce/Hadoop.

In contrast to Starfish, MRTuner only focuses on optimizations at the job level [SZL+14]. Even though Starfish has a job-level component, the approach of MRTuner is different. To optimize job runtimes, MRTuner attempts to optimize parallel execution between map and reduce tasks. By achieving maximum parallel execution between tasks, MRTuner can better utilize system resources. The critical insight of MRTuner is its Producer-Transporter-Consumer (PTC) cost model, which is one of the few cost models for MapReduce jobs that takes into account parallel execution and the overlapping of the map, shuffle, and reduce tasks. Another significant contribution of MRTuner is their reduction of system parameters, which can significantly decrease the optimal configuration search space.

Starfish and MRTuner approach the job of MapReduce tuning from different sides. There is not one correct approach; both have good and bad aspects. Starfish's use of dynamic instrumentation is clever because it enables a lightweight execution model to be built without impacting the performance of the application in the process. Likewise, MRTuner's PTC cost-based model is a theoretical approach to the same goal. One area MRTuner excels over Starfish is in the reduction of configuration parameters, which enable faster and more-optimal tuning results. Similarly, it was clever of Starfish to also include optimizations at the macro or workload level, though perhaps it would be better to fork those into a separate project because they are not mainly related to optimizations at the job level. Both approaches from the MapReduce world have provided some valuable insights which we can apply to Spark and perhaps other Big Data frameworks in the future.

Because of the complexity of Spark over MapReduce and other predecessors, research into modeling, predicting and performance tuning Spark applications is still in relative infancy. It is a hard problem. Nonetheless, a few have tried to tackle it from different directions. Similar to what we have seen in MapReduce, approaches can take a white-box analytical (cost-based) approach or a black-box machine learning approach. The first serious system to consider for Spark performance prediction is Ernest [VYF$^+$16]. The goals of Ernest are (1) to predict runtimes of large Spark jobs efficiently and (2) suggest optimal runtime resource configurations. It uses a black-box approach combined with machine learning. Spark jobs are run on small input sizes, then a linear model is built, from which optimal coefficients can be selected using the Non-Negative Least Squares regression variant. The authors claim that a linear model works well enough because of the tendency of Big Data analytics jobs to be linear in nature themselves. Once the prediction model has been created, Ernest uses it to select the optimal number of machines and machine types. To shorten the overhead from the system, Ernest also uses optimal experiment design [Puk93] to select optimal training data sets. Ernest can suggest parameters based upon shortened runtime and cost minimization, although it appears flexible enough to be able to optimize parameters based upon some other metric.

Other cost-based models for Spark have been developed in recent times, e.g. [SS17] and [WK15]. Both approaches are very similar in nature. Recall that in Spark, jobs are divided up into stages, with each stage containing one or more tasks. The number of stages is dependant upon the program structure, and tasks dependant upon the data size and layout (partitioning). The above models work first by running the entire job on small input sizes to capture resource usage (e.g., I/O, memory, cache usage, serialization time, etc.) at the task-level. From there, they can estimate the number of tasks and their runtimes that would be used for larger datasets. Implicit in these models is the assumption that resource usage scales somewhat linearly as the data input size grows (this is the same assumption made by Ernest). The two cost-based models can predict performance characteristics relatively well for the selected workloads and data sizes.

In the presented black-box and white-box approaches to performance prediction, one thing that stands out is the difficulties in predicting the I/O demands of Spark

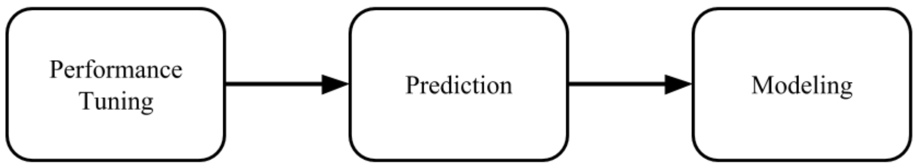

Figure 6: Dependencies between Performance Tuning, Predicting and Modeling

applications. The predominant method today of building models based upon small input samples cannot accurately model the I/O consumption at larger data sizes. For some shuffle-heavy jobs like Sort and WordCount where I/O is a factor, this can make predictions less accurate, as has been found in [WK15]. In Spark-like environments, resources such as virtual memory and virtual-cores scale well with the input size, but I/O it does not. Models which are not based upon sampling and extrapolation may address this limitation, however, no serious proposals have yet been proposed to our knowledge.

By looking at previous research in MapReduce with systems like Starfish and MR-Tuner, we have been able to see the motivation behind newer systems for Spark such as Ernest. In MapReduce, it has been shown that black-box and white-box solutions can work well. Likewise in Spark, the current research indicates that the same division is also present. Because of MapReduce's simplified Map-Shuffle-Reduce architecture, it was easier to develop white-box models for its behavior. For Spark and its parallel nature, it is harder to build accurate models that do not over-fit. Despite this, we have covered two published attempts, and there are undoubtedly more in the works along this path.

5 Delaunay Triangulation Framework for Spark Performance Modeling

5.1 Spark Modeling and Performance Tuning

The previous benchmarks have shown the dynamic performance space that exists for Spark applications. Adding or subtracting just one CPU core or 2GB of memory can have an enormous impact on the job's performance. Modeling and understanding this performance space is essential because, at the Big Data scale, these costs add up. Inefficient jobs translate into a direct loss of time and resources for an organization. To have a job run efficiently, it means that we have searched all the possible execution configurations and selected an optimal one.

So far we have avoided defining what an "optimal" Spark job is. Optimal for whom? A business leader might want all cluster computing jobs to be tuned so that the least overall amount of resources are used. Likewise, a Data Scientist probably wants their Spark job to run as fast as possible so that they can iterate on the results. Furthermore, a cluster administrator might tune the jobs by some other metric, such as grouping them onto as few machines as possible. The key idea here is that "performance tuning" is an ambiguous term because it means different things to different stakeholders. This thesis began with the thought that "performance tuning" Spark was a binary subject – there is a right way and a wrong way to do it. However, it was quickly discovered that there is no single correct answer to this question.

Given this paradox of performance tuning that exists in Spark applications, it was decided to focus on modeling Spark applications. Performance tuning depends on the ability to predict system performance under many situations, which itself requires a flexible and realistic model of the system (refer to Figure 6). An accurate model of an application's performance space can be used to tune applications towards many different performance goals and also predict future performance under different runtime configurations.

5.2 Primitives

Our method borrows the algorithm of Delaunay Triangulation from Computational Geometry. As such, some basic terms and concepts used in our work are defined following.

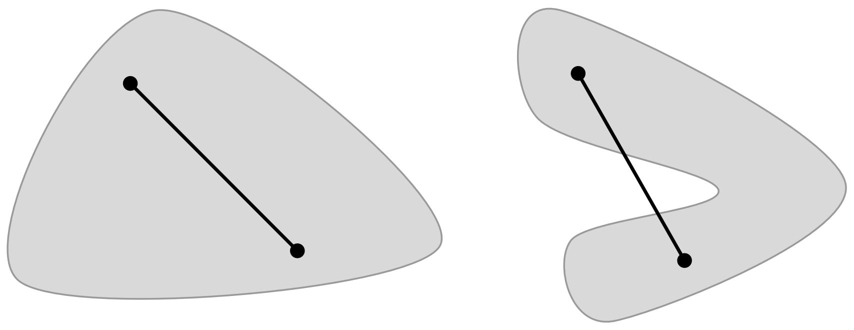

Figure 7: Convex hull bounding a convex region (left) and a non-convex hull bounding a non-convex region (right).

Convex Region A convex region [MS15] has the property that any line segment joining any two points lies completely within the region. We show an example in Figure 7, illustrating convex region and non-convex region.

Convex Set A convex set [MS15] represents the points inside a convex region.

Convex Hull It is the fundamental construction of Computational Geometry. The convex hull [BDH96] of a set of points is the smallest convex set that contains the points. Refer to Figure 7.

Hyperplane It is the generalization of a plane in three dimensions to higher dimensions. That is, any (d-1) subspace in $\mathbb{R}^d$. In n dimensions, a hyperplane is defined by the equation $b = a_1n_1 + a_2n_2 + ... + a_nn_n$ where each a_i and b are constants.

Simplex It is the generalization of a triangle to different dimensions. In d dimensions, the concept of a triangle becomes a d-simplex. For example in Figure 8, a $2d$ triangle is a 2-simplex, a $3d$ tetrahedron is a 3-simplex, and so on. Furthermore, each simplex is constructed of facets, which form the boundary (i.e., min and max values) of the surface. The number of facets in a simplex is a function of the number of edges.

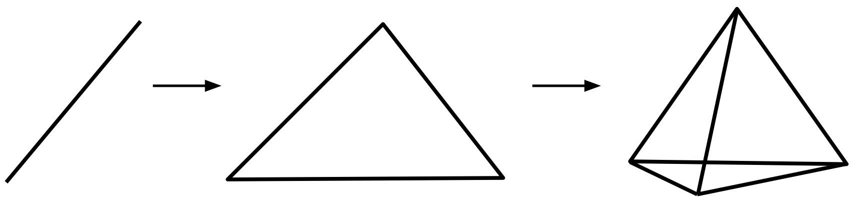

Figure 8: A 1-simplex (line), 2-simplex (triangle) and 3-simplex (tetrahedron). A Delaunay Triangulation in $\mathbb{R}^d$ constructs d-simplexes.

Delaunay Triangulation "The unique triangulation of a set of sites so that the circumsphere of each triangle has no sites in its interior. The dual graph of the Voronoi Diagram." [For92]

Voronoi Diagram "The set of all Voronoi faces. The dual graph of the Delaunay Triangulation." [For92]

Voronoi Face "The set of points for which a single site is closest (or more generally a set of sites is closest)." [For92]

Circumsphere / Circumcircle A circle or sphere formed around a polygon that touches all of the points of the polygon.

As is hopefully apparent, most items here are just generalizations of 2d concepts to arbitrary dimensions.

5.3 Delaunay Triangulation Framework

As we have now seen, there are many important aspects to consider when performance tuning Big Data frameworks like Spark. Proper performance tuning is built doubly upon (1) a good model of the underlying system, and (2) predictions for system performance at various runtime configurations. In this work, we contribute a model for predicting job runtimes on Spark which applies to other Big Data frameworks. Future works may expand upon this framework to add the final piece of performance tuning.

Many models place constraints on the features that may be included. For example, Ernest [VYF+16] constrains their model to the dataset (input size) and machine type, and the models presented in [SS17] are constrained to executor number, executor cores, and executor memory. Some models can support more features, but it requires significant work to recalculate them, especially those which are white-box in nature. The model we propose in this chapter is not tied to any particular set of features and can be applied with equal utility to an arbitrary amount of features. Furthermore, by generalizing the features available to the model, we open the model up to broader applications and adoptions. In this chapter, we will first formally discuss our approach to the modeling of Spark applications. Secondly, a novel geometric method for predicting Spark runtimes will be introduced and analyzed. Thirdly, we will tie everything together into an end-to-end algorithm for predicting Spark runtimes. This chapter lays the theoretical foundation required to understand our approach and the proceeding experiments and discussions.

To model Spark's runtime and make predictions, we have created a black-box model using the history of the job at various feature configurations as our primary guide. A black-box approach was taken because of the aforementioned complexities of the Spark framework. The Ernest framework [VYF+16] also makes use of a black-box model and achieves satisfactory results, so it has already been proven to work for Spark. There are of course white-box models for Spark (we have analyzed a few here), however, as these systems become more complex, it can be argued that black-box models will carry more of a significance. White-box models can only handle increasing complexity in a system to a threshold, before a black-box approach *must* be taken.

In this section, we present our framework for making runtime predictions under multi-dimensional feature configuration scenarios. Supporting the framework is the Delaunay Triangulation and Adaptive Sampling techniques. We begin by sampling the feature space using Latin Hypercube Sampling (LHS), a space-filling sampling technique well-suited for higher-dimensional spaces. With these samples, we combine them with first *boundary samples*, thereby forming enough *seed samples* to construct the initial model. The model itself is constructed by computing the Delaunay Triangulation over the seed samples, which is a technique to split the data into a set of interconnected simplexes (triangles), forming a mesh over the data set. From these simplexes, we lift them into a higher dimension by including their runtime, and then we calculate the hyperplane passing through the vertices of each simplex. From this hyperplane, predictions for unknown runtime configurations may be made by linear interpolation. To improve the model, we propose a sampling technique, Adaptive Sampling, which uses LHS sampling combined with a *utility* function to

Table 3: Variable Reference

Abbrev.	Description
n	Number of (current) samples
m	Number of LHS samples
d	Number of features
F	Configuration space (available runtime configurations)
F_i	Specific runtime configuration
$\hat{T}$	Estimated runtime
$T(F_i)$	Runtime at a specific F_i runtime configuration

strategically pick the next optimal sample to add to the model. The details of each step are given in the proceeding sections.

5.4 Problem Statement

At a high level, we want to use historical runtime data to predict the runtime of future jobs under different parameter configurations for arbitrary Big Data jobs like those running on the Spark framework. More formally, given n samples $\mathcal{S} = \{\langle F_i, T(F_i)\rangle | 1 \leq i \leq n\}$, a configuration space F, a prediction model $PM(\cdot)$ returns the estimated running time $\hat{T}$ for F:

$$\hat{T} = PM(\mathcal{S}, F) \tag{2}$$

where each configuration $F_i = \{f_1, f_2, ..., f_d\}$ $(1 \leq i \leq n)$ includes d features (parameters) and the corresponding runtime is $T(F_i)$. To evaluate the model in our experiments, we introduce a metric called the Mean Absolute Percentage Error (MAPE), which is the average of the Percent Errors for a set of estimated runtimes ($\hat{T}$) and actual runtimes (T):

$$MAPE = \frac{100\%}{l} \sum_{i=1}^{l} |\frac{T_i - \hat{T}_i}{T_i}| \tag{3}$$

where l is the number of specific feature configurations to be tested. For reference, we have included a table (refer to Table 3 summarizing the important variables used in our model and the proceeding sections.

In general, this problem can be converted to a general prediction problem with historical data (i.e., training data), which can be solved by a statistics machine learning method such as Multivariate Linear Regression (LR) [Nas07] or Gaussian Process (GP) [Ras04]. However, as we will show in our evaluation section, these existing methods require massive training data points for better prediction, and they are very costly to construct an accurate performance model for Big Data analytics platform.

The goal of this Thesis is to develop a novel method for accurate runtime prediction for Big Data analytics jobs. Since acquiring training data samples can be expensive, our approach emphasizes efficient system model building with as few training samples as possible.

5.5 Modeling

Our model of the underlying Big Data system is based upon a d-dimensional Delaunay Triangulation and a set of overlaid $(d + 1)$-dimensional hyperplanes. To create the model, we must first select features; then we must collect runtime data, construct the Delaunay Triangulation, and finally, calculate the hyperplanes. A high-level summary is given below:

1. Select a set of d features (parameters) to build the model around, $\{f_1, f_2, ..., f_d\}$, e.g., $\{memory, vcores\}$.
2. Gather a set of previous job runtimes at specific feature configurations, e.g., {16 GB, 4 VCores} $\rightarrow$ 455 seconds. Runtimes may be collected from previous historical data, or strategically sampled as needed.
3. Generate Delaunay Triangulation in $\mathbb{R}^d$ space using the set of d features.
4. For each d-simplex returned from the Delaunay Triangulation containing $(d+1)$ vertices, add the actual runtimes to each of the $(d+1)$ points and then calculate the hyperplane passing through the vertices of the d-simplex.
5. Solve the equation for the hyperplane in terms of runtime and use this to make runtime predictions.

5.5.1 Feature Selection

The first step in building our model is to select a set of d features, $\{f_1, f_2, ..., f_d\}$, to contain in the model. For most Spark workloads, as our benchmarking data (refer to

Chapter 3) has shown, the amount of memory and CPU cores (vcores) allocated has one of the largest impacts on a job's performance. For an initial model of a Spark system, we suggest starting with those features. Other influential features might include input size, data locality, bisection bandwidth, and serialization techniques. Our model is unique in that it does not constrain to fixed features, but rather allows the user to apply any collection of features. Instead of using conventional wisdom, a user may also use automated techniques (e.g., [APGZ17]) to select features.

5.5.2 Runtime Data Collection

Since the performance metric is the runtime, the next step is to generate input data for different feature configurations to form the basis of our model. Intuitively, as more data is sampled, a more accurate model can be constructed. It is, however, a research challenge to build an accurate model using as few sample data as required. To this end, we develop a feedback-driven sampling solution to select points which have a high potential for improving the accuracy of the model. The detailed sampling method will be described in Section 5.7. Here, it is sufficient to understand that a subset of the available input data is strategically selected to build the model upon.

5.5.3 Delaunay Triangulation

After runtime data has been collected (refer previous section), the next step is to plot it into a d-dimensional space. To enable runtime predictions, the objective is to fit a hypersurface to the data so that predictions can be made for unknown $\langle f_1, f_2, ..., f_d \rangle$ points. The hypersurface should:

1. Assume that each $\langle f_1, f_2, ..., f_d \rangle$ point cannot have multiple runtime values, i.e. it resembles a proper function that passes the vertical-line test.
2. Pass through each historical $\langle f_1, f_2, ..., f_d \rangle \rightarrow runtime$ point
3. Be quick to compute and avoid major recomputation for the addition or removal of a single point
4. Minimize prediction error for unknown $\langle f_1, f_2, ..., f_d \rangle$ values

The naive approach to this problem is to take a black-box machine learning (ML) approach to it. There are two problems with a machine learning approach: training

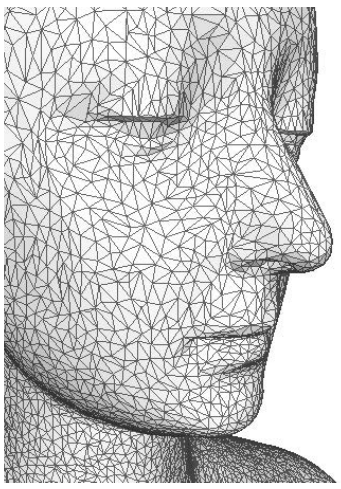

Figure 9: Example of a polygon mesh used in computer graphics [10]

cost and model accuracy. The machine learning approach works from the assumption that there is a pool of data to train the model with initially. This assumption is not always optimal because in a potentially costly environment like the public Cloud, it is better to have a usable model as fast as possible, and then iterate from there. Machine learning models take many data to train and test before they are accurate enough for production use.

The second problem with a machine learning approach is the accuracy uncertainties of the model. The problem with higher dimensional regressions is that for a given Spark workload, multiple regressions need to be performed before one is possibly found which meets the criteria outlined above. Some workloads might exhibit topography that lends itself well to a higher dimensional quadratic regression, while others might be better suited for a quartic or linear regression. The shape of the surface can be highly dynamic (also consider the possibility of outlier points), and because of this, many regressions of different degrees may have to be performed before an adequate one is potentially found.

It is indeed possible to apply machine learning techniques such as a d-dimensional linear regression to this problem, and future works may do so, but there exists

[10] `https://www.sci.utah.edu/the-institute/highlights/24-research-highlights/cibc-highlights/439-cleaver.html`

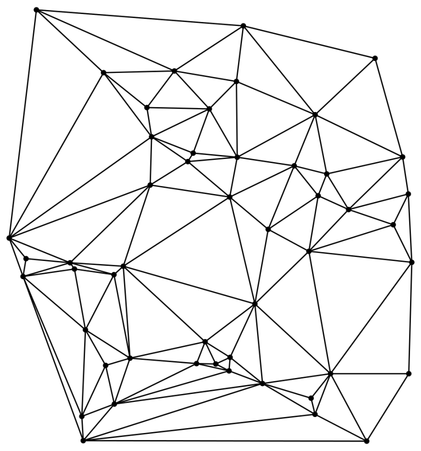

Figure 10: Delaunay Triangulation over a set of random points [11]

a simpler solution. Instead of defining the surface as one smooth function, it is more comfortable and better in many aspects to break the surface into a piecewise collection of hyperplanes. The inspiration for this solution comes from the computer graphics world. In computer graphics, the requirements are similar, i.e., cost-effectively approximating an object's shape with high accuracy. In computer graphics, the typical solution is to construct a mesh of polygons that wrap the surface, giving an approximation of it. We will borrow this idea and apply it to our problem of fitting the performance space of Spark.

A Delaunay Triangulation [Del34] is used to divide an n-dimensional surface into a collection of conjoined simplexes. Formally, a Delaunay Triangulation is "the unique triangulation of a set of sites so that the circumsphere of each triangle has no sites in its interior" [For92]. Figure 10 shows the Delaunay Triangulation for a set of 50 random points. Over a set of points, the Delaunay Triangulation will partition them into a set of triangles in a fashion such that each point P of a given triangle is not inside the circumcircle of any other triangle.

A Delaunay Triangulation can be constructed from the dual graph of a Voronoi diagram. Together, Delaunay Triangulations and Voronoi diagrams form the building

[11] `https://people.eecs.ku.edu/~jrmiller/Courses/775/InClass/TessellationExamples/TessellationExamples.html`

blocks of the field of Computational Geometry [Zem10]. Voronoi diagrams are used in many related applications, but not this work. Despite this, Voronoi diagrams are so closely linked that many works consider both as one. In two dimensions, the Delaunay Triangulation partitions the feature space into a set of 2-simplexes (triangles). In general, in *d* dimensions, the Delaunay Triangulation forms *d*-simplexes. It is manageable to display in two or three dimensions, but it gets quickly complex after that. We have chosen the Delaunay Triangulation in particular over other techniques because:

1. It prefers to form equilateral simplexes as much as possible, leading to less "sliver" simplexes, which are problematic for interpolation of the values within.
2. It is easily extensible into higher dimensions – the math remains unchanged.
3. Many efficient algorithms have been developed to produce and traverse Delaunay Triangulations.

Other polygon-mesh generation methods exist, but few are as simple or have been studied as extensively as the Delaunay Triangulation.

To generate a Delaunay Triangulation for a set of points in $\mathbb{R}^d$ space, we utilize the relationship[12] between Delaunay Triangulations in $\mathbb{R}^d$ and a parabola in $\mathbb{R}^{d+1}$ [BDH96]. At a high level, we first compute the convex hull of the point set in $\mathbb{R}^{d+1}$ space (by lifting to a parabola), then project the convex hull back into $\mathbb{R}^d$ space, leaving the Delaunay Triangulation in $\mathbb{R}^d$. The convex hull is a fundamental construction in Computational Geometry and has wide applications in many areas. Because of this, many efficient algorithms have been developed to generate convex hulls. For this reason, we will use a convex hull in $\mathbb{R}^{d+1}$ as the starting point for our Delaunay Triangulation. Convex hull algorithms are extremely well-researched due to their applications in computer graphics, and hence, all can be used since *the Delaunay Triangulation is itself a convex hull problem.* Other methods exist to generate Delaunay triangulations and Voronoi diagrams, however, implementation is more difficult in higher dimensions. As experience has proven, it is easier to form our Delaunay Triangulation from the well-known convex hull base.

[12]This relationship between the Delaunay Triangulation and parabola projection is explored further in [BDH96]

In our model, we use the Quickhull algorithm [BDH96] to form a convex hull of the points in $\mathbb{R}^{d+1}$. Pseudocode for the algorithm can be found in Algorithm 1. There are many variations to the Quickhull algorithm, and the one presented is a simplified version (it quickly becomes complex in higher dimensions). Quickhull runs in $\Theta(n \log n)$ with a worst-case complexity of $O(n^2)$. The worst-case complexity occurs when points have unfavorable (highly symmetric) distributions. In our system, we do not consider this to be a valid cause for concern because our sampling method described in Section 5.7 ensures that we asymmetrically pick points, thereby avoiding the pitfalls of high-symmetry. The algorithm is very similar to Quicksort because it uses a divide-and-conquer recursive approach to solving the problem. There exist many variations to Quickhull, and the one listed here is the easiest from an understanding perspective.

Algorithm 1: Quickhull [BDH96]

```
Result: Subset of input points that form the convex hull
Input  : Set of points $S$ in any $\mathbb{R}^d$ where $d > 1$
Create $d$-simplex with $d + 1$ points;
foreach facet $F$ do
    foreach unassigned point $p$ do
        if if $p$ is above $F$ then
            assign $p$ to $F$'s outside set;
        end
    end
end
foreach facet $F$ with a non-empty outside set do
    select the furthest point $p$ of $F$'s outside set;
    initialize the visible set $V$ to $F$;
    forall unvisited neighbors $N$ of facets in $V$ do
        if $p$ is above $N$ then
            add $N$ to $V$;
        end
    end
    the set of horizon ridges $H$ is the boundary of $V$;
    foreach ridge $R$ in $H$ do
        create a new facet from $R$ and $p$;
        link the new facet to its neighbors;
    end
    foreach new facet $F'$ do
        foreach for each unassigned point $q$ in an outside set of a facet in $V$ do
            if $q$ is above $F'$ then
                assign $q$ to $F'$'s outside set;
            end
        end
    end
    delete the facets in $V$;
end
```

After the convex hull has been created in $\mathbb{R}^{d+1}$, we project it downward back to $\mathbb{R}^d$, thereby giving the Delaunay Triangulation.

With Delaunay Triangulations, we have what we need to create a polygon mesh over a generic $\langle f_1, f_2, ..., f_d \rangle \rightarrow runtime$ Spark feature space. Delaunay Triangulations provide a reasonable approximation of the underlying surface topography and offer a simple alternative to d-dimensional regressions. There are many algorithms available for creating the Delaunay Triangulations efficiently ($O(n * log(n))$ time or better), and furthermore, they have been extensively studied by computer scientists.

5.5.4 Calculating Prediction Hyperplane

The Delaunay Triangulation partitions the feature space into a series of simplexes, and we construct a hyperplane by bringing in the runtime dimension for each simplex. This process is best explained by stepping through in two and three dimensions, then extrapolating into higher dimensions. In two dimensions, e.g., $\langle f_1, f_2 \rangle$, the Delaunay Triangulation generates 2-simplexes (triangles), each naturally containing three points. For each point $\langle f_1, f_2 \rangle$, we add the runtime as the third dimension (e.g. $\langle f_1, f_2, \mathit{runtime}(f_1, f_2) \rangle$), then compute the hyperplane containing these three points. In three dimensions, e.g. $\langle f_1, f_2, f_3 \rangle$, the Delaunay Triangulation creates 3-simplexes (tetrahedrons), each containing four points. From that, we add runtime in and then compute the hyperplane containing each of the tetrahedron's points. As we travel into higher dimensions, the process remains the same. In other words, in a model constructed with d features, we must lift it into $\mathbb{R}^{d+1}$ by including runtime in a separate step.

Once we have the equation for each hyperplane, we can solve it in terms of *runtime*, thereby giving us a function to predict runtime based upon the features in the model. The generic scalar form of a hyperplane of d dimensions (e.g. $\beta_0 = \sum_{i=1}^{d} \beta_i x_i$) solved in terms of runtime, t, is given in Equation 4, with the assumption that *runtime* is always the last term of the equation, i.e. x_n.

$$t = x_d = \frac{\beta_0 - \sum_{i=1}^{d-1} \beta_i x_i}{\beta_d} \tag{4}$$

5.6 Prediction

Once hyperplanes have been computed for each simplex, the final step is to determine which simplex a given point belongs to. When the simplex's hyperplane has been obtained, we simply can plug the known feature values into Equation 4 to get the estimated runtime at that specific configuration. A formal procedure can be found in Algorithm 2.

Note that this prediction algorithm relies heavily on the assumption that any potential prediction values will lie inside the convex hull of the model. This is precisely why in the approach that we picked 2^d *boundary samples* at the extremes of the feature configuration space. This step ensures that any future predictions will fall inside the convex hull, and thus have a simplex for prediction. Points lying outside the convex hull ("outer-hull" points) are difficult to predict because their values must be extrapolated from a hyperplane inside the hull. By selecting boundary samples at the beginning, we can avoid this situation.

Algorithm 2: Runtime Prediction with Delaunay Model

```
Result: Predicted runtime t for configuration $F$
Input : Model $M$ in $\mathbb{R}^d$ space containing a set of $d$-simplexes and attached $\mathbb{R}^{d+1}$
        hyperplanes
Input : $F = \langle f_1, f_2, ..., f_d \rangle$
foreach $s_i \in M$ do                                   // simplex $s_i$
    if $F \in s_i$ then
        $h \leftarrow$ hyperplane($s_i$);
        $t \leftarrow h(F)$ ;                            // predict runtime by its hyperplane
        return $t$;
    end
end
```

5.7 Sampling

Bootstrapping the initial model is an essential consideration for our system. If there is no previous historical data, what is the smallest set of samples that can be gathered to achieve the most-accurate model? Likewise, if some historical data exists, which subset of this data should be chosen for the initial model? The model should minimize the number of initial samples required for construction. In higher-dimensional configuration spaces, it is crucial to identify those samples which have the highest impact on the model. Furthermore, once the initial model has been constructed, how can the model be improved to some goal utilizing the least amount of samples? These are all important considerations, which our sampling technique addresses.

Naive sampling approaches in situations like this are random and gridding sampling. Random sampling selects points at random from the configuration space without replacement. Gridding sampling divides the configuration space into a uniform grid and then selects samples equidistant from each other. These sampling methods produce sub-optimal models because they do not consider the underlying data values, resulting in over-sampling of regions where there are little change and under-sampling where there is high change. We introduce a novel sampling technique, Adaptive Sampling, which selects more samples in regions where the rate of change is high, and fewer samples where there is no change in the topography. To build an accurate performance model, it is crucial to quickly identify the regions of high-change in the performance surface, which we accomplish with Adaptive Sampling.

Our sampling technique is divided into two phases: seed sampling and Adaptive Sampling. In seed sampling, we select an initial set of points to seed the model with. After the initial model has been created, it can be iteratively improved by adding more samples using the Adaptive Sampling technique.

5.7.1 Seed Sampling

We go through the following steps to generate the seed samples:

1. Determine d features to include in the model and their bounds, thereby forming the *feature space*, F, for the model. We normalize the feature space $\in [0, 1]$.

2. Use Latin Hypercube Sampling (LHS) [Ima08] to select m feature configurations. In the LHS technique, samples are chosen in a way such that the complete range of value elements is fully represented. However, LHS may generate bad spreads where all samples are spread along the diagonal. Therefore, we maximize the minimum distance between any pair of samples. Suppose we have n samples, we will select the sample set X^* such that:

$$X^* = arg \max_{1<i<n} \min_{f_1^{(x_1^i)},..f_d^{(x_d^i)} \in \mathcal{D}_{LHS}, x_1 \neq ... \neq x_d} \mathbf{Dist}(f_1^{(x_1^i)}, ..., f_d^{(x_d^i)}) \tag{5}$$

   where `Dist` is a typical distance metric, e.g., Euclidean distance in our implementation.

3. Combine LHS samples with 2^d boundary samples (taken from the minimum and maximum points of each feature axis) to gather $2^d + m$ seed samples for the initial model. There are two reasons for including boundary points. First, we include boundary points to ensure that all possible unknown runtime

configurations lie within the convex hull, thereby avoiding making predictions outside of the convex hull ("outer-hull" points), where there is no appropriate hyperplane for reference. Second, by selecting boundary points along with the initial LHS samples, we ensure an even distribution of points for the initial Delaunay Triangulation and avoid the complexity pitfalls of highly-symmetric point distributions.

5.7.2 Adaptive Sampling

The intuition behind our iterative sampling technique is to select new points for the model which are close to the greatest runtime changes. These areas need samples the most because they indicate an area where small changes in feature values may result in significant changes in runtime. We introduce a *utility* metric to compute the distance between predicted point and its hyperplane. Intuitively, a higher *utility* value indicates a larger distance to the points of its prediction hyperplane and thus a higher potential improvement to the model. Given samples X from LHS domain $\mathcal{D}_{LHS}$ and the n samples $\mathcal{S}$ with d features, we achieve predicted runtime $\hat{T^{(i)}}$ by sample $X^{(i)}$ and its hyperplane $\mathcal{S}' \subseteq \mathcal{S}$ with h sample $\langle F_k^{(i)}, T(F_k^{(i)})\rangle, 1 < k < h$. The utility $U(X^{(i)})$ of sample $X^{(i)}$ is defined as follows:

$$U(X^{(i)}) = \frac{1}{h}\sum_{k=1}^{h}(\frac{1}{d+1}(\sum_{j=1}^{d}(f_{k,j}^{(i)} - X_j^{(i)})^2 + (T(F_k^{(i)}) - \hat{T^{(i)}})^2)) \tag{6}$$

where h is the number of points constructing its hyperplane. The iterative sampling technique proceeds as follows until a stopping condition has been met:

1. Use Latin Hypercube Sampling across the entire feature space to re-sample m sample points X.

2. Use the current model to compute the utility $U(\cdot)$ of X.

3. Rank all samples by utility and pick the largest $U(X^{(i)})$ as the next sample to add to the model. The next sample's features $F_{(n+1)}$ is achieved as follows:

$$F_{(n+1)} = arg \max_{X \in \mathcal{D}_{LHS}} U(X) \tag{7}$$

4. Adaptive Sampling continues until a stopping condition has been met. We define an explicit threshold of prediction error as the stopping condition. In our empirical experiments, we continue selecting more sampling points into the model until the average error (i.e. MAPE) $\leq 5\%$ has been reached.

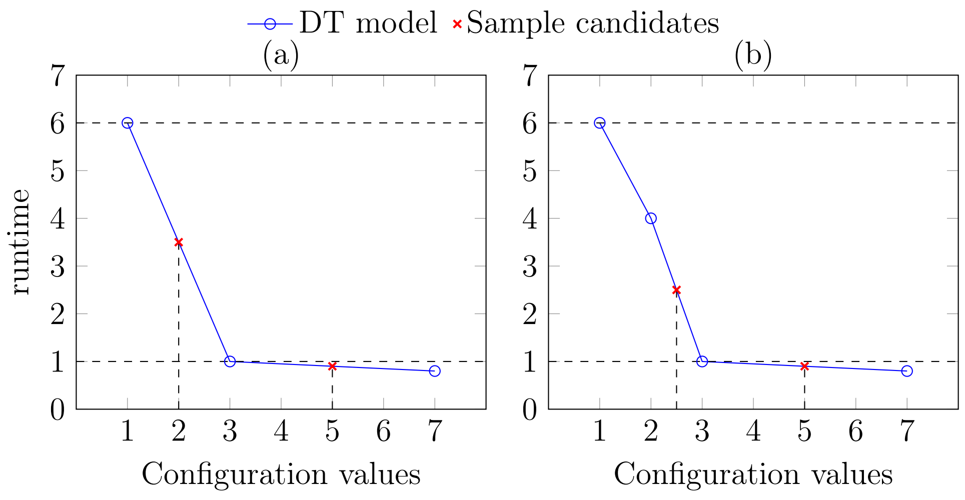

Figure 11: The next sample point to pick based on the current DT model. (a) Pick configuration value 2. (b) Pick configuration value 5.

It is important to note that the above sampling technique balances the conflicting tasks of exploration (understanding the global surface of the model) and exploitation (going regions where the performances of adjacency points change fast) that arise in model building. Achieving this balance is nontrivial. We provide an example to illustrate the idea of picking next point by the utility as follows.

Figure 11 depicts an example scenario with synthetic data. Suppose three experiments have been done, and the collected data points shown in sub-figure (a). Suppose the runtime is 3.5 for configuration value 2 by linear regression from points $(1, 6)$ and $(3, 1)$. Based on the points $(1, 6)$, $(3, 1)$, we compute the utility $U = 3.625$ for this candidate point 2 by Equation 6. Similarly $U = 2.005$ for point 5. We choose point 2 to be the next sample point for experiments because the run-time between 1 and 3 decreases steeply (by exploitation). Further, suppose the actual runtime for

Algorithm 3: Adaptive Sampling

```
Result: prediction model: $PM$
Input: sample set: $\mathcal{S}$; initial prediction model: $PM$
repeat
    $\mathcal{D}_{LHS} \leftarrow LHS(\mathcal{S})$
    $F_{(n+1)} = arg\ \max_{X \in \mathcal{D}_{LHS}} U(X)$        // Eq. 7
    $PM \leftarrow update(PM, F_{(n+1)}, T(F_{(n+1)}))$
    $\mathcal{S} \leftarrow \mathcal{S} - F_{(n+1)}$
until stopping condition is met
return $PM$
```

configuration value 2 is 4. Then we updated the model shown in sub-figure (b). At that moment, we compute 1.25 and 2.005 utility for configuration value 2.5 and 5, respectively. We chose the configuration value 5 as our next sample, because of the large uncertainty of points between 3 to 7 (by exploration).

5.8 Implementation

We implement the Delaunay Triangulation model as a Python project organized as a set of modules[13]. We implement Delaunay Triangulation by using a Python wrapper[14] for the `qhull`[15] library which internally uses the Quickhull Algorithm (Algorithm 1) to determine the Convex Hull and resulting Delaunay Triangulation for a set of points. The `qhull` library also contains useful abstractions for the components in the triangulation, such as determining if a simplex contains a point. On top of `qhull` we utilize `numpy` for linear algebra primitives, and `scikit-learn`[16] [PVG$^+$11] for LR [Nas07] and GP [Ras04] implementations.

Latin Hypercube Sampling is developed in Python from scratch. More performance may be gained by writing more components from scratch, however, the implementations provided by the referenced modules have been sufficient for our evaluation presented in the next section.

6 Evaluation Methodology and Results

In this chapter, we present the empirical results to evaluate our approach systematically. We implemented against three sampling techniques: random, grid and adaptive sampling. We compared the Delaunay Triangulation framework with linear regression and Gaussian processing. Experimental results show the superiority of the adaptive sampling and the Delaunay Triangulation model. We achieve lower prediction error with fewer samples to build the model. The detail of the evaluation is following.

[13]Project Homepage: `https://github.com/HY-UDBMS/d-Simplexed`

[14]`https://pypi.org/project/pyhull/`

[15]`http://qhull.org/`

[16]`http://scikit-learn.org/stable/index.html`

6.1 Environment

On the software side, we used Apache Spark v2.1.0 on top of Hadoop v2.8.1. With Hadoop, we use HDFS as our distributed filesystem, and YARN as our resource manager. This is a typical open-source Spark software stack. From the hardware side, our experiments were run on the Ukko high performance computing cluster, made available by the Department of Computer Science at the University of Helsinki[17]. The cluster consists of over 200 Dell PowerEdge M610 servers, each having 32GB of RAM, 2 Intel Xeon E5540 2.53GHz CPUs and four cores per CPU, making (with hyperthreading) 16 virtual-cores available. For our experiments, we used one node as the YARN ResourceManager, with two separate worker nodes being running the YARN NodeManager process. It is worth noting that the cluster has a network file system (NFS) which we used as the primary HDFS storage node.

6.2 Data Generation

For our evaluations, we use a mixture of data collected from the previous benchmarks and synthetic data based upon the benchmark data. Due to the many points to collect, we collected enough initial points uniformly throughout the feature space, built a Delaunay Triangulation model from them, and used it to predict the remaining points in the feature space. We then use this data for the actual evaluation, as if it were collected as the result of a Spark workload. This approach allows us to emulate large data sets without having to perform hundreds or thousands of performance samples. It is key to understand that to evaluate the model, *it does not matter where the data comes from, so long as the same data used for building the model is the same used for verifying its accuracy.* By generating the evaluation data for the model, we enable a deeper analysis because there is orders-of-magnitude more granular data to test our model against.

6.3 Sampling Evaluation

The first set of experiments is performed to compare various sampling techniques to build a performance model. We implemented three approaches: random sampling,

[17]`https://www.cs.helsinki.fi/en/compfac/high-performance-cluster-ukko`

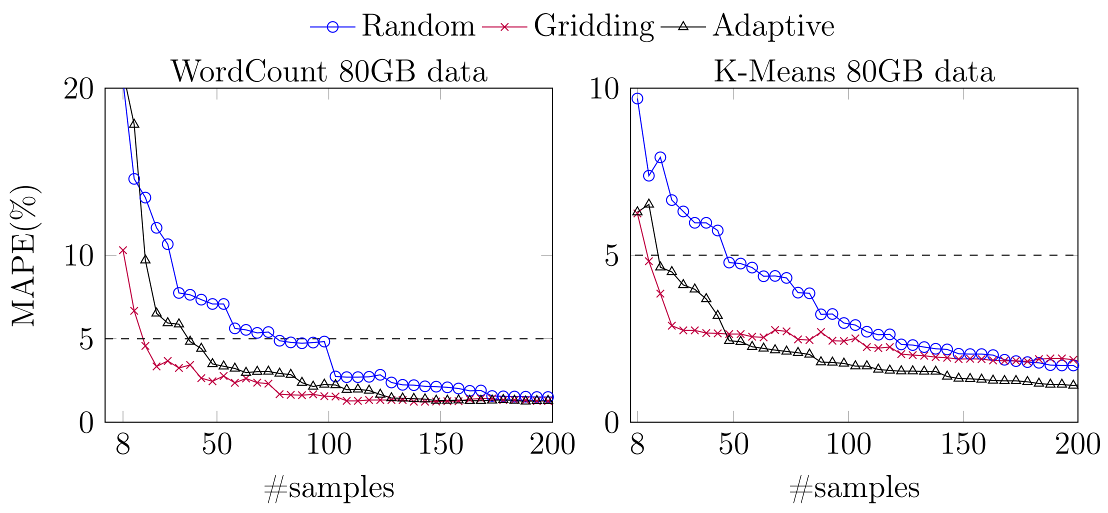

Figure 12: Comparison of sampling techniques; Random Sampling, Grid Sampling, and Adaptive Sampling. Adaptive Sampling excels in discovering the samples to improve the model.

grid sampling, and our adaptive sampling. A model is constructed with two different features: memory and vcores. As we have seen from our benchmarking results (refer to Chapter 3), the amount of memory and CPU cores given to a job has one of the most significant influences on its performance. For that reason, we have included memory and vcores as the initial feature set. In each sampling approach, we select four boundary points and four seed samples retrieved using LHS sampling, giving eight samples to create the initial model with. The model is then iteratively improved by adding new samples 1-by-1 and evaluating the Mean Absolute Prediction Error (MAPE) at each step. We used K-means workload with 5GB and 80GB data size. For the 80G data, the configuration space spans from 40-240GB memory and 60-160 vcores. For the 5GB data, the space includes 10-60GB memory and 5-25 vcores. In both settings, the unit size of each step is 2GB memory and one vcore. We randomly reserved 10% of configuration space samples as the test data (excluding the eight seed samples). This applies to the rest of the experiments.

Figure 12 plots the prediction error (i.e MAPE) for three sampling techniques. It shows that given the same number of samples, adaptive sampling achieves the smallest error ratio among three techniques. It indicates that adaptive approach can be used to build a more accurate performance model.

The performance of random sampling is weak, especially when the model has very few sampling points. This is because when there are few points in the model, there is

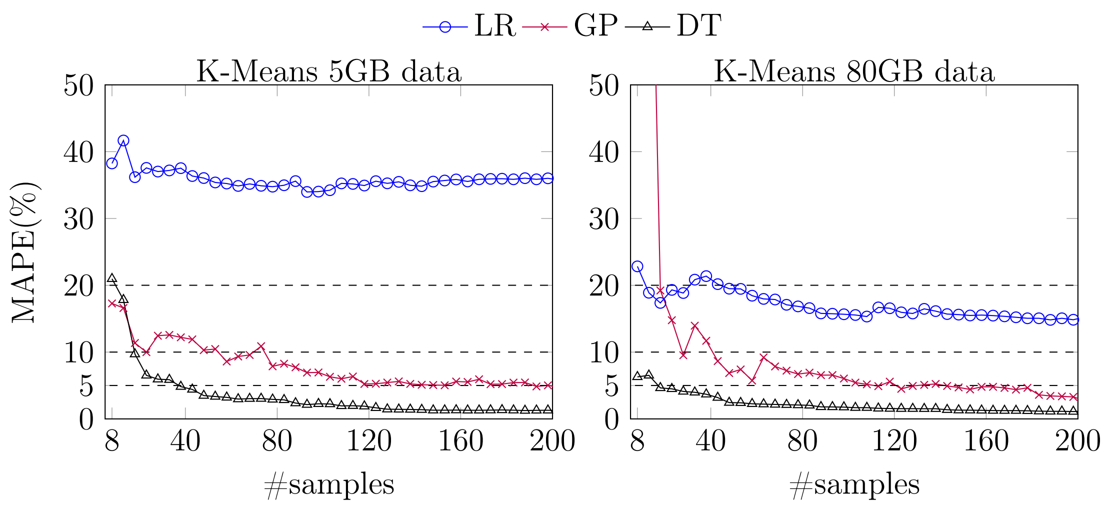

Figure 13: Comparison of the prediction models for K-means workload; Multivariate Linear Regression (LR), Gaussian Process (GP), and Delaunay Triangulation (DT). With 5GB and 80GB data, the Delaunay Triangulation model reaches less than 5% error by 8.6% and 0.18% configuration spaces, respectively.

a high probability that the next random point selected will fall in a region which has small performance change and therefore the new sample cannot make a significant contribution to predicting the critical turning point of performance. Grid sampling is better than random sampling, and it performs well at the beginning phase because it evenly distributes the chance of picking points in the model. In contrast to the grid and random sampling, our adaptive sampling selects the points that continuously improve the accuracy of the model. The adaptive sampling technique avoids picking points in the regions where there is little change in topography. In the 80GB data set, this behavior is especially pronounced, where adaptive sampling zooms-in on the critical region of the model which has an abrupt performance change, while grid and random sampling use a static strategy to pick points randomly or uniformly.

6.4 Model Evaluation

The second set of experiments compares our method to two machine learning methods, namely Multivariate Linear Regression (LR) [Nas07] and Gaussian Process (GP) [Ras04].

Figure 13 shows the experimental results against K-means workload with 5GB and 80GB data, respectively. The Delaunay Triangulation model is the best method in

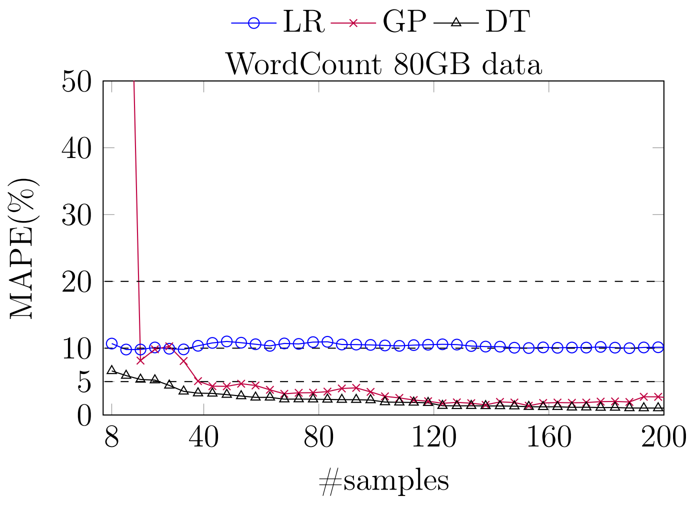

Figure 14: Comparison of the prediction models for WordCount workload; Multivariate Linear Regression (LR), Gaussian Process (GP), and Delaunay Triangulation (DT). With 80GB data, the Delaunay Triangulation model reaches less than 5% error by 0.22% configuration spaces.

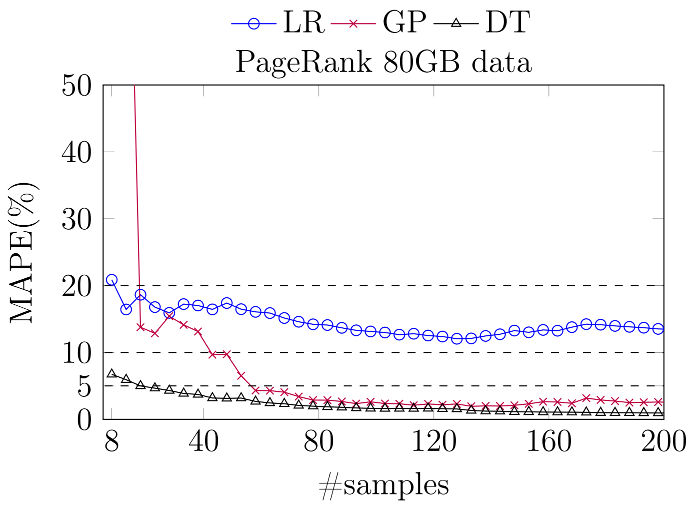

Figure 15: Comparison of the prediction models for PageRank workload; Multivariate Linear Regression (LR), Gaussian Process (GP), and Delaunay Triangulation (DT). With 80GB data, the Delaunay Triangulation model reaches less than 5% error by 0.27% configuration spaces.

Table 4: Workload evaluation. The Delaunay Triangulation model achieves less than 5% prediction error rate with less than 1% samples.

Workload	#configurations	Samples(%)	Error(%)
WordCount	40-240GB	0.16	4.40
K-Means	60-160Vcores	0.22	4.66
PageRank	step size: 2GB 1Vcore	0.27	4.43

both datasets with a varied number of samples. In particular, LR performs poorly since a linear model faces difficulty in capturing non-linear behavior. The GP model works relatively better than LR. However, it has the problem of over-fitting for the data as more points are loaded into the model. Our Delaunay Triangulation model outperforms the other two approaches. This is because of two reasons: 1) when new sample points are added, the Delaunay Triangulation model prefers to fit them locally, that is, their placement only impacts a few adjacent simplexes, rather than the entire model (in the case of GP and LR); and 2) the Delaunay Triangulation model uses utility (in Equation 6) function to judiciously pick the next point for sampling to improve the model as sample size increases continuously. Figure 14 and Figure 15 show the experimental results against WordCount and PageRank workload, respectively, which demonstrate the similar trend as K-means workload.

Table 4 shows the error ratios of our Delaunay Triangulation method with different sample ratios varied from 0.16 to 0.27 against three workloads. It indicates that the Delaunay Triangulation model selects a very small fraction of data (0.2%) while accurately predicting the performance on most of points (4.5% error ratio), which strongly motivates its application in practice.

6.5 Synthetic Workload Evaluation

In the last set of experiments, we sought to analyze the performance of models against a synthetic workload. We create a synthetic 120×120 =14400 point $\langle f_1, f_2 \rangle$ surface to demonstrate the model's flexibility under a hypothetical runtime condition, where the runtimes have massive turbulence as shown in Figure 16. We assume that some Big Data workloads may exhibit such behavior and it is an interesting surface to challenge with our model.

We evaluated this synthetic surface with three models and present the results in Figure 17. We did not plot LR performance here, because its corresponding MAPE is

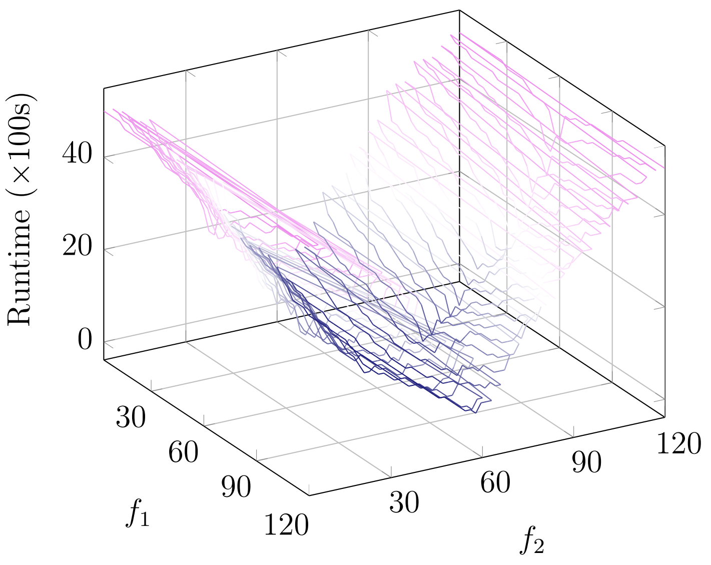

Figure 16: Synthetic workload with a massive turbulent surface illustrated in three dimensions.

more than 100% all the time. GP is much better than LR, but it still cannot achieve a smaller error than 10%. The reason is that GP considers the surface as a whole and cannot adapt to the local flexibility. On the contrary, our Delaunay Triangulation model keeps discovering the unknown regions and continuously improving the model (thanks to the utility function), and it finally achieves 6% error while selecting only 3% data points.

6.6 Discussion

Building an accurate model of a Big Data system like Spark is a hard but essential task. As we have seen, the topography of Spark performance spaces can be very turbulent. Subtle changes in one feature can have a significant impact on the job's runtime. Likewise, there can also be vast areas in the topography of no change. Our models must capture the topography of this dynamic space using the least amount of resources and samples as possible. Our presented model utilizing the Delaunay Triangulation can handle these situations. Our results show its superiority in achieving not only high prediction accuracy with a low amount of samples, but also its tendency to approach on 0.0% prediction error when the others fail. Furthermore, the model's sparing use of points is especially noteworthy. In our results, we observe very high prediction accuracy even after the initial seed sampling phase, which is often only 8 points over a multi-thousand value configuration space. In most cases,

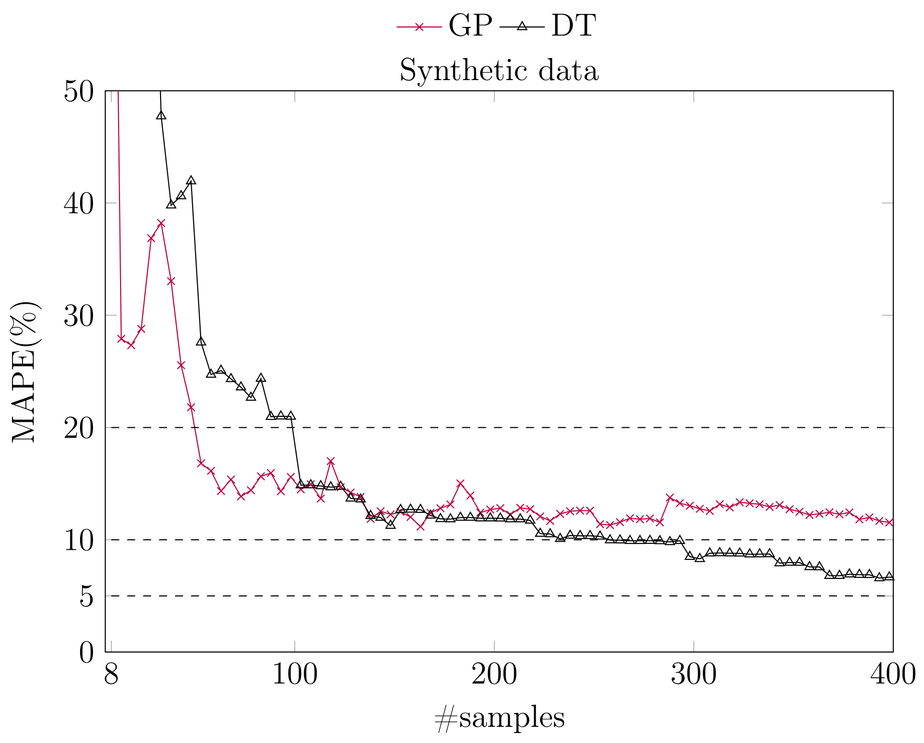

Figure 17: Comparison of the models for synthetic turbulent surface. The Delaunay Triangulation model achieves less than 6% error by sampling less than 3% data.

we observe that with <1% of the samples, we achieve <5% prediction error over the remaining space. That is remarkable.

The most significant benefit our model has over Linear Regression, Gaussian Process, and other models is its generality. In this work we have presented our Delaunay Triangulation model with $d = 2$ features, however, we have previously shown that it can be extended into higher dimensions. It is possible to extend the model to higher dimensions based upon previously gathered data. When moving to a higher dimension, the previously gathered data becomes a subset of the new higher dimensional space. So then, to efficiently build the model, one can gather the missing data around the existing subset of data. In this fashion, the existing data can be reused for future models. Other models (e.g., Machine Learning) are not as flexible when adding or subtracting dimensions. For example, a linear regression might work well with $d = 2$ features, however, a quadratic or cubic regression might perform better with $d = 5$ features. With many Machine Learning techniques, it is often the case that multiple variations need to be evaluated before an optimal one is found. Our Delaunay Triangulation model works sufficiently without variation in any space. The generality of the Delaunay Triangulation model is one of its biggest strengths.

Another strength of our Delaunay Triangulation model is its use of Adaptive Sampling to select new samples for the model optimally. We have developed and tested our Adaptive Sampling technique, showing its superiority over other trivial sampling techniques like random and gridding. The Adaptive Sampling technique works well when the Big Data administrator does not (yet) know the underlying performance characteristics for their workload. In other scenarios where more historical data is present, or the user is knowledgeable about the performance of their job under various feature configurations, it might be advantageous to write a custom sampler to achieve even a tighter fit to the data and/or a faster convergence rate. We recommend using Adaptive Sampling because it works well for most workloads, however, it is entirely possible with the Delaunay Triangulation architecture to plug-in a different sampler. The only constraint is to ensure that the model is seeded with points forming a convex hull over the feature space so that an outer-hull scenario is avoided (refer Section 5.6).

Our discussion throughout this work has been mostly theoretical, however, the applications of our Delaunay Triangulation model to Big Data analytics applications are numerous. The presented model can handle many different application scenarios. Here we list a few scenarios which can benefit from our model.

1. *Unknown or turbulent surfaces*: When the topography of the workload in any dimension changes rapidly, is unknown or is difficult to visualize, our method sees no difference because it can handle all scenarios above as equal.

2. *Complex dependent variables*: In the case of non-independent variables, many existing modeling techniques (such as Multivariate Linear Regression (LR) [Nas07]) fail because the relationship between variables is difficult to capture. However, our model excels in this scenario because the disjoint nature of the Delaunay Triangulation allows it to be an effective tool for capturing subtle changes and complex relationships in the workload's topography.

3. *Multiple variables and large configuration spaces*: Large Big Data systems can have many tuning knobs and a large range of tuning values, thereby leading to an exponential number of possible configuration combinations. Our method only uses few critical samples to discover the underlying performance surface of these systems, regardless of the number of features or size of the feature space.

Our presented model is well-adapted to handle the above scenarios. Its generic nature and ease-of-extensibility allow it to support many different modeling applications. While we have benchmarked it against Spark, it is certainly not limited there and can be applied to other Big Data frameworks with equal utility.

7 Conclusions and Future Work

Here we have presented a geometric approach to modeling runtimes for Big Data systems like Spark. Our Delaunay Triangulation model including Adaptive Sampling provides a highly-extensible alternative to Machine Learning, requiring less initial data and iterations to achieve a highly-accurate model. We have implemented the model in python and benchmarked it against common multivariate modeling solutions like Linear Regression and Gaussian Process, showing its superiority in most scenarios.

To the best of our knowledge, this is the first model proposed in the domain of Big Data modeling which applies a geometric (Delaunay Triangulation) approach to solving the modeling problem. For this reason, we believe there are significant research opportunities available in this area. Below we enumerate some of the interesting future work possible from this foundation:

1. *Alternative Meshing Methods*: The core idea of the Delaunay Triangulation method is to wrap a surface in polygons, which can then be used to interpolate values within. Though we have reasonably justified our choice for picking the Delaunay Triangulation, there exists many other polygon mesh techniques which could be investigated. Some techniques developed for applications in 2, 3 or 4 dimensions may produce a better model, but lack extensibility to higher dimensions. Likewise, there may exist some other solutions, like the Delaunay Triangulation, which work sufficiently in many higher dimensions. The field of Computational Geometry is rich in these methods to examine.

2. *Grey-box Approaches*: Earlier we have examined other models for Big Data platforms, and reached the conclusion that they can be either white-box or black-box in nature. There is however opportunity to mesh the properties of both types into a *grey-box model*. In a grey-box model, parts of the system that are easily reasoned are approximated in a white-box fashion, while the parts that are not clear or overly complex are approximated in a black-box

fashion. In this way, we take the best of both sides. For example, our black-box Delaunay Triangulation model could be modified to use white-box techniques to reduce its number of features used in the model, thereby using knowledge of the internals of the system to reduce its complexity.

3. *Iterative Model Computation*: In scenarios where performance is at a premium, such as in mobile and embedded devices, it may be beneficial to research the options for adding new samples into an existing Delaunay Triangulation *without* having to recompute it from scratch. When the samples in the model reach over 500-1000, it can become a costly operation to compute the Delaunay Triangulation. In this situation, it would be a large performance benefit to only recompute the region of the model which is changing.

4. *Performance Tuning*: Our model has been demonstrated to accurately predict runtimes for unknown runtime configurations. This provides a core component of a performance tuning framework. We can use this functionality to search the runtime configuration space for possible tuning optimizations. Once an accurate prediction model has been created, performance tuning reduces to a search problem. As we have discussed earlier, performance tuning is guided by the individual towards their own goals, whether that is lowest runtime, least resource usage etc. Our model provides enough features to support all of these possible tuning constraints.

5. *Applications to Other Domains*: In this work, we have modeled the performance of Big Data frameworks as a function of their inputs. Notice that this is, however, a very generic description of almost any complex system. All systems have some inputs that lead to some output(s). Our presented Delaunay Triangulation model could potentially be used in the future to model other systems where multiple inputs map to a single output. The applications are most-relevant in Big Data systems, however, they are equally applicable in other fields.

Our presented Delaunay Triangulation model utilizing Adaptive Sampling is a geometric solution to modeling Big Data frameworks such as Spark. It provides a simpler alternative to the current best practices and opens exciting avenues for future research.

References

AFG$^+$10	Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I. et al., A view of cloud computing. *Communications of the ACM*, 53,4(2010), pages 50–58.
APGZ17	Aken, D. V., Pavlo, A., Gordon, G. J. and Zhang, B., Automatic database management system tuning through large-scale machine learning. *SIGMOD Conference*. ACM, 2017, pages 1009–1024.
BDH96	Barber, C. B., Dobkin, D. P. and Huhdanpaa, H., The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22,4(1996), pages 469–483.
CSL$^+$04	Cantrill, B., Shapiro, M. W., Leventhal, A. H. et al., Dynamic instrumentation of production systems. *USENIX Annual Technical Conference, General Track*, 2004, pages 15–28.
Del34	Delaunay, B., Sur la sphere vide. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, 7,793-800(1934), pages 1–2.
DG04	Dean, J. and Ghemawat, S., Mapreduce: Simplified data processing on large clusters. *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, 2004, USENIX Association, pages 10–10.
DN14	Doulkeridis, C. and Nørvåg, K., A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal*, 23,3(2014), pages 355–380.
For87	Fortune, S., A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2,1-4(1987), page 153.
For92	Fortune, S., Voronoi diagrams and delaunay triangulations. In *Computing in Euclidean geometry*, World Scientific, 1992, pages 193–233.
GGL03	Ghemawat, S., Gobioff, H. and Leung, S.-T., *The Google file system*, volume 37. ACM, 2003.
HAG$^+$16	Hashem, I. A. T., Anuar, N. B., Gani, A., Yaqoob, I., Xia, F. and Khan, S. U., Mapreduce: Review and open challenges. *Scientometrics*, 109,1(2016), pages 389–422.

HHD$^+$10 Huang, S., Huang, J., Dai, J., Xie, T. and Huang, B., The hibench benchmark suite: Characterization of the mapreduce-based data analysis. *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on.* IEEE, 2010, pages 41–51.

HKZ$^+$11 Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S. and Stoica, I., Mesos: A platform for fine-grained resource sharing in the data center. *NSDI*, volume 11, 2011, pages 22–22.

HLL$^+$11 Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F. B. and Babu, S., Starfish: a self-tuning system for big data analytics. *Cidr*, volume 11, 2011, pages 261–272.

HYA$^+$15 Hashem, I. A. T., Yaqoob, I., Anuar, N. B., Mokhtar, S., Gani, A. and Khan, S. U., The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, 47, pages 98–115.

Ima08 Iman, R. L., Latin hypercube sampling. *Wiley StatsRef: Statistics Reference Online.*

Lan01 Laney, D., 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6,70(2001).

Mas99 Mashey, J. R., Big data and the next wave of infrastress problems, solutions, opportunities, Jun 1999. URL `http://static.usenix.org/event/usenix99/invited_talks/mashey.pdf`.

MCB$^+$11 Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C. and Byers, A. H., Big data: The next frontier for innovation, competition, and productivity.

MS15 Morris, C. C. and Stark, R. M., *Finite Mathematics: Models and Applications.* John Wiley & Sons, 2015.

Nas07 Nasrabadi, N. M., Pattern recognition and machine learning. *Journal of electronic imaging*, 16,4(2007), page 049901.

NL91 Nitzberg, B. and Lo, V., Distributed shared memory: A survey of issues and algorithms. *Computer*, 24,8(1991), pages 52–60.

Puk93 Pukelsheim, F., *Optimal design of experiments*, volume 50. siam, 1993.

PVG$^+$11 Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. et al., Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12,Oct(2011), pages 2825–2830.

Ras04 Rasmussen, C. E., Gaussian processes in machine learning. In *Advanced lectures on machine learning*, Springer, 2004, pages 63–71.

SQM$^+$15 Shi, J., Qiu, Y., Minhas, U. F., Jiao, L., Wang, C., Reinwald, B. and Özcan, F., Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment*, 8,13(2015), pages 2110–2121.

SS17 Singhal, R. and Singh, P., Performance assurance model for applications on spark platform. *Technology Conference on Performance Evaluation and Benchmarking.* Springer, 2017, pages 131–146.

SZL$^+$14 Shi, J., Zou, J., Lu, J., Cao, Z., Li, S. and Wang, C., Mrtuner: a toolkit to enable holistic optimization for mapreduce jobs. *Proceedings of the VLDB Endowment*, 7,13(2014), pages 1319–1330.

VMD$^+$13 Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S. et al., Apache hadoop yarn: Yet another resource negotiator. *Proceedings of the 4th annual Symposium on Cloud Computing.* ACM, 2013, page 5.

VYF$^+$16 Venkataraman, S., Yang, Z., Franklin, M. J., Recht, B. and Stoica, I., Ernest: Efficient performance prediction for large-scale advanced analytics. *NSDI*, 2016, pages 363–378.

WB13 Ward, J. S. and Barker, A., Undefined by data: a survey of big data definitions. *arXiv preprint arXiv:1309.5821.*

Whi09 White, T., *Hadoop: The Definitive Guide.* O'Reilly Media, Inc., first edition, 2009.

WI98 Weiss, S. M. and Indurkhya, N., *Predictive data mining: a practical guide.* Morgan Kaufmann, 1998.

WK15 Wang, K. and Khan, M. M. H., Performance prediction for apache spark platform. *High Performance Computing and Communications (HPCC),* *2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on.* IEEE, 2015, pages 166–173.

ZCD$^+$12 Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S. and Stoica, I., Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation.* USENIX Association, 2012, pages 2–2.

ZCF$^+$10 Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. and Stoica, I., Spark: Cluster computing with working sets. *HotCloud*, 10,10-10(2010), page 95.

Zem10 Zemek, M., Regular triangulation in 3d and its applications. 2010.