Shade: A Differentially Private Wrapper Around Apache Spark

by

Alexander G. Heifetz

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2017

© Massachusetts Institute of Technology 2017. All rights reserved.

Author

Department of Electrical Engineering and Computer Science

August 18, 2017

Certified by

Lalana Kagal
Principal Research Scientist
Thesis Supervisor

Accepted by

Christopher Terman
Chairman, Masters of Engineering Thesis Committee
Shade: A Differentially Private Wrapper Around Apache Spark

by

Alexander G. Heifetz

Submitted to the Department of Electrical Engineering and Computer Science on August 18, 2017, in partial fulfillment of the requirements for the degree of Masters of Engineering in Computer Science and Engineering

Abstract

Enterprises usually provide strong controls to prevent external cyberattacks and inadvertent leakage of data to external entities. However, in the case where employees and data scientists have legitimate access to analyze and derive insights from the data, there are insufficient controls and employees are usually permitted access to all information about the customers of the enterprise including sensitive and private information. Though it is important to be able to identify useful patterns of one’s customers for better customization and service, customers’ privacy must not be sacrificed to do so. We propose an alternative - a framework that will allow privacy preserving data analytics over big data. In this paper, we present an efficient and scalable framework for Apache Spark, a cluster computing framework, that provides strong privacy guarantees for users even in the presence of an informed adversary, while still providing high utility for analysts in an interactive wrapper. The framework, titled Shade, includes two mechanisms - SparkLAP, which provides Laplacian perturbation based on a user’s query and SparkSAM, which uses the contents of the database itself in order to calculate the perturbation. We show that performance of Shade is substantially better than earlier differential privacy systems without loss of accuracy, particularly when run on datasets small enough to fit in memory, and find that SparkSAM can even exceed performance of an identical non-private Spark query.

Thesis Supervisor: Lalana Kagal
Title: Principal Research Scientist
Acknowledgments

This thesis would not have been possible without the support of many other people and organizations. I would like to thank first and foremost my supervisor Dr. Lalana Kagal for her helpful guidance and patience over the course of the thesis process. To the rest of the Decentralized Information Group, particularly Amy van der Hiel and Sir Tim Berners-Lee, thank you for making me feel at home at CSAIL and for some extremely insightful questions during the spring semester. I am grateful to Professor Murat Kantarcioglu and Kanchan Waikar of UT Dallas for advice and guidance on Spark, to Professor Adam Smith of Penn State University for advice on the Sample and Aggregate framework, and to the TIG staff and Justine Jang for help with creating and maintaining the Spark cluster. Darwins Ltd and Lord Hobo in Cambridge were the site of many hours of writing and the friendly staff kept me well-supplied in coffee and beer. My friends, girlfriend, and siblings provided endless support and distraction during the writing process, and never failed to remind me just how far I was from ever graduating. Finally, the largest thanks go to my parents Dabney and Peter Heifetz for unending love, guidance, and inspiration. This thesis is dedicated to you.
# Contents

1 Introduction .......................................................... 11

2 Background ..................................................................... 13
   2.1 Spark ................................................................. 13
   2.2 Differential Privacy .................................................. 15
   2.3 Perturbation Mechanisms .......................................... 16
      2.3.1 Laplacian Mechanism ....................................... 17
      2.3.2 Sample-and-Aggregate Mechanism ....................... 19

3 Related Work ............................................................ 23
   3.1 PINQ/wPINQ ........................................................ 23
   3.2 Airavat ................................................................. 24

4 Approach ....................................................................... 25
   4.1 Overview ............................................................... 25
   4.2 Assumptions and Threat Model .................................... 26
   4.3 Configuration .......................................................... 27
   4.4 Usage .................................................................... 28
   4.5 Sample Workflow ..................................................... 29
      4.5.1 Data Provider .................................................. 30
      4.5.2 Data Analyst ................................................... 30

5 Implementation .......................................................... 31
   5.1 PrivateSparkContext ................................................. 31
5.2 Budget ................................................................. 32
5.3 SparkLAP ............................................................. 34
  5.3.1 Interface ......................................................... 35
  5.3.2 Type Enforcement ............................................. 38
  5.3.3 Range Enforcement .......................................... 38
  5.3.4 Key Enforcement .............................................. 39
  5.3.5 Stability Enforcement ....................................... 40
  5.3.6 Sample Queries ............................................... 40
5.4 SparkSAM .......................................................... 41
  5.4.1 Partitioning .................................................... 42
  5.4.2 Aggregation .................................................... 43
  5.4.3 Sample Queries ............................................... 44
5.5 Adversary Prevention ............................................. 44
5.6 Known Vulnerabilities ............................................ 47
  5.6.1 Function Impurity ............................................. 47
  5.6.2 Accessible Endpoints ....................................... 48
  5.6.3 Timing Attacks ............................................... 48

6 Evaluation ................................................................. 51
  6.1 Goals .............................................................. 51
    6.1.1 Performance .................................................. 51
    6.1.2 Privacy and Accuracy ..................................... 52
  6.2 Configuration ..................................................... 52
    6.2.1 Cluster ........................................................ 52
    6.2.2 Storage and MapReduce ................................... 53
    6.2.3 Spark .......................................................... 53
  6.3 Datasets .......................................................... 54
    6.3.1 AOL Dataset .................................................. 54
    6.3.2 Netflix Dataset ............................................... 54
  6.4 Experiments ....................................................... 55
7 Conclusion

7.1 Discussion .................................................. 75
  7.1.1 SparkLAP Performance ............................... 75
  7.1.2 SparkSAM Performance .............................. 78
  7.1.3 Privacy And Accuracy ............................... 80

7.2 Future Work .................................................. 81

7.3 Conclusion .................................................. 83
List of Figures

2-1 Result of \( \text{Lap}(\frac{1}{4}) \) added to 1000 .......................................................... 18

4-1 Diagram illustrating components of Shade and the roles of the Data Analyst and Data Provider ................................................................. 27

6-1 Performance of MapReduce, SparkLAP, Spark Core on Average Rating (n=5) ............................................................. 56

6-2 Performance of MapReduce (MR), SparkLAP (SL), Spark Core (SC) on 1st, Subsequent iterations of Iterative Average Rating (n=5, i=10) 57

6-3 Performance of MapReduce (MR), SparkLAP (SL), Spark Core (SC) on 1st, Subsequent iterations of Expanded Iterative Average Rating (n=5, i=10) ............................................................. 59

6-4 Performance of MapReduce, SparkLAP, Spark Core on Many Average Ratings (n=5) ............................................................. 60

6-5 Performance of MapReduce, Spark Core, SparkLAP on Relative Keyword Occurrence (n=5) ............................................................. 62

6-6 Performance of MapReduce, Spark Core, SparkLAP on Efficient Relative Keyword Occurrence (n=5) ............................................................. 64

6-7 Performance of MapReduce, Spark Core, SparkSAM on Median Rating (n=5) ............................................................. 66

6-8 Performance of MapReduce, Spark Core, SparkSAM on Median Hash (n=5) ............................................................. 67

6-9 Performance of MapReduce, Spark Core, SparkSAM on Sparse Median Hash (n=5) ............................................................. 69
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-10</td>
<td>Performance of SparkLAP, SparkSAM on Many Average Ratings (n=10)</td>
<td>70</td>
</tr>
<tr>
<td>6-11</td>
<td>Accuracy of SparkLAP on Movie 2 Count (n=10000)</td>
<td>71</td>
</tr>
<tr>
<td>6-12</td>
<td>Accuracy of SparkLAP on Movie 2 Average Rating (n=10000)</td>
<td>72</td>
</tr>
<tr>
<td>6-13</td>
<td>Accuracy of SparkSAM on AOL Median Hash (n=1000)</td>
<td>73</td>
</tr>
<tr>
<td>6-14</td>
<td>Accuracy of SparkLAP, SparkSAM on Average Hash (n=10)</td>
<td>74</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Consider a social networking site (LifeSpace) that allows companies to create and run advertisements, which are then shown to users. LifeSpace has recently started tracking and storing information about how its users interact with these ads. A wide variety of useful information can be learned from ad interactions [20] - one client might want to analyze how a certain ad performs with 18 year olds as compared to 40 year olds, while another client might want to analyze what times of day and locations its ads are most successful in, and another still might want to run machine learning queries to make predictions on how well its next ad will perform.

Naturally, LifeSpace wishes to monetize this information. However, it is aware that releasing the interaction dataset directly will compromise the trust of users and could lead to bad publicity if it becomes public that LifeSpace is "selling data". Even leaking a single user’s information could be disastrous; consider the backlash that LifeSpace would face if an adult videos company were to release the name of its most frequent inbound user. Simply releasing an "anonymized" dataset of ad interactions still leaks significant user information - there are many recent cases of individuals being identified from so-called anonymous datasets [27]. Generating aggregations for each ad themselves would be prohibitively expensive for LifeSpace, and it’s infeasible to choose a finite set of aggregations that covers all possible information that companies might want to learn (e.g. the machine learning case). In addition, it has been
shown that even the release of non-noisy aggregations can still occasionally compromise user privacy and allow individuals to be identified [14].

Overall, LifeSpace needs a system that will let analysts learn as much information about ad performance as possible, while still being able to provide a guarantee to its users that their information is being kept private and safe. Such a system should be efficient, scalable, and fault-tolerant with ability for LifeSpace to control exactly how much information is released. Finally, it should not require significant expertise in either privacy or data analysis to be able to use, and should provide an industry-standard interface that many analysts are already familiar with.

In this thesis, we present Shade: a system for private data analysis. Chapter 2 provides background on the state of the art in both privacy and data analysis, focusing on differential privacy and Apache Spark. In Chapter 3 we examine previous attempts to implement differentially private systems and the relative benefits and drawbacks of each. Chapter 4 outlines Shade, a differentially private wrapper around Spark that improves on existing differentially private systems through performance gains, query expressivity, and ease of use. Chapter 5 then discusses the implementation of Shade and its two different mechanisms: SparkLAP and SparkSAM. In Chapter 6 we present experimental results showing significant advances from previous systems in performance and query expressivity, as well as accuracy on certain globally-sensitive workloads such as median-finding. Finally, Chapter 7 explores the implications of these results and proposes future extensions to Shade.
Chapter 2

Background

2.1 Spark

Frameworks for large-scale parallel data processing such as MapReduce [4] have been highly useful in allowing researchers to run acyclic data flow computations in a distributed manner. However, more recent systems have proved able to better handle modern scale of data through optimizations such as lazy evaluation, in-memory caching, and automated fault-tolerance. Apache Spark [35] is one such system that has seen wide adoption across both academia and industry in recent years, and is the current benchmark holder across a number of data analysis competitions (e.g. the Daytona Grey 100TB sort). Its main contribution is an abstraction called a Resilient Distributed Dataset (RDD) [34] which represents a read-only collection of objects partitioned across a set of machines. RDDs are implicitly fault-tolerant and can be lazily manipulated through a set of Transformations such as Map, Filter, Partition, GroupBy, and Join. Information can then be returned to the user through Actions such as Count, Collect, and Reduce. Jobs in Spark are known as "driver programs" and are written as Scala programs that define the control flow of the program, usually involving construction of RDDs, transformations on them, and actions taken to return information back to the user. Spark can be used for a wide variety of data analysis tasks such as statistical querying or machine learning, and one of the goals of Spark's creation was to make cluster computations easier to write [33]. This improved
performance and usability has led to Spark being one of the most popular data processing engines used in industry today, with over 500 active production deployments as of 2015 [1].

The following examples from [35] illustrate some possible Spark jobs that can be run.

**Example 1 (Error Count)**

```scala
val lines = spark.textFile(...)
val errors = lines.filter(line => line.startsWith("ERROR"))
println(errors.count())
```

**Example 2 (Map and Reduce)**

```scala
// Count the errors mentioning MySQL
errors.filter(line => line.contains("MySQL")).count()

// Fetch the MySQL errors as an array of strings
errors.filter(line => line.contains("MySQL")).collect()

// Fetch the time fields of errors mentioning PHP as an array
// (assuming time is field number 3 in a tab-separated format):
errors.filter(line => line.contains("PHP"))
  .map(line => line.split("\t")(3))
  .collect()
```

**Example 3 (Gradient Descent)**

```scala
// Read points from a text file and cache them
val points = spark.textFile(...).map(parsePoint).persist()

// Initialize w to random D-dimensional vector
var w = Vector.random(D)

// Run repeatedly over the same dataset cached in memory
for (i <- 1 to ITERATIONS) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  }.reduce((a, b) => a + b)
  w -= gradient
}

// Final separating plane
println(w)

2.2 Differential Privacy

Many possible approaches can be taken to preserve the privacy of individuals. Early techniques [7] included "anonymizing" datasets, by redacting or removing certain fields (name, address, etc) from datasets and operating on them normally. However, it quickly became apparent that an adversary with auxiliary information can still learn significant information from these datasets [24], and more formal approaches such as $k$-anonymity and $l$-diversity [7] were soon adopted. These too were shown to be susceptible to identification by adversaries with auxiliary information [14], and an even stricter approach became desirable. In the extreme, such an approach would be able to preserve privacy of any single individual in the dataset, regardless of queries performed or variance of the data. Differential privacy [6] is a formalization of this notion. Intuitively, for a given individual who is considering participating in a dataset, differential privacy requires that an analyst is able to learn no more information from a dataset that contains this individual’s information than one that does not. As no individual sample can affect the output, attackers can thus not infer the private information corresponding to any individual sample.

Definition 1 (Differential Privacy) Consider a mechanism $f(D)$ that takes a dataset $D$ as input, a distance function $d(D, D')$ denoting the minimum number of sample changes required to change dataset $D$ into dataset $D'$, and a non-negative number $\epsilon$. We say that $f$ satisfies $\epsilon$-differential privacy iff, for all neighbors $d(D, D') = 1$
and some set $S$ denoting the output range of $f$, the following relationship holds: $P(f(D) \in S) \leq e^\epsilon P(f(D') \in S)$. Thus, we can say that our system is $\epsilon$-differentially private if all queries made to the system satisfy $\epsilon$-differential privacy.

In general, a lower value of $\epsilon$ signifies that the standard for privacy protection is higher, but utility to analysts is likely lowered. There also exists a variant of approximate differential privacy (called $(\epsilon, \delta)$-differential privacy) that provides a probabilistic guarantee of the relationship's enforcement, however no components of Shade use this variant; we refer to "differential privacy" as exclusively $\epsilon$-differential privacy for the remainder of this thesis.

Even when given a differentially private system, privacy is still not guaranteed unless the amount of information released is also given a global limit, after which point the database can no longer be queried at all. We call this limit the "privacy budget", and any system that claims differential privacy must also provide support for a privacy budget, which, when exhausted, restricts the user from making any additional queries. The choices for the parameter $\epsilon$ and the privacy budget will vary depending on the privacy needs of the dataset, or even (in the case of the privacy budget) the number of input records. Choosing this value is outside the scope of this thesis, but is discussed at length in [15] and [18].

### 2.3 Perturbation Mechanisms

In order to be able to create a system that satisfies the constraints of differential privacy, it is essential to perturb (add noise to) any outputs from queries placed upon it. The magnitude of this perturbation, and the mechanism by which it is applied, can be highly variable [17]. Below, we will examine two major mechanisms by which this perturbation is applied, and discuss how the magnitude of the noise to apply is determined.
2.3.1 Laplacian Mechanism

Before discussing the Laplacian Mechanism [8], we first define the Laplacian Distribution.

**Definition 2 (Laplace Distribution)** The Laplace Distribution $\text{Lap}(\lambda)$ is a probability distribution with a single parameter $\lambda$ indicating the "scale" and has density function $h(x) = \frac{1}{2\lambda} \exp(-\frac{|x|}{\lambda})$. This distribution has mean $\mu = 0$ and standard deviation $\sigma = \sqrt{2}\lambda$.

The Laplacian mechanism is a perturbation that applies $\epsilon$-differential privacy for queries $f(D) \in \mathbb{R}^p$. It does so by taking the actual value $f(D)$ and adding a random variable $n$ whose value is taken from the Laplacian distribution with parameter $\lambda$, giving $f'(D) = f(D) + n$. In order to maximize accuracy, we want to choose the minimum $\lambda$ that still provides differential privacy, given the privacy parameter $\epsilon$ and the query $f$. [8] provides a proof that by choosing $\lambda = \frac{s(f)}{\epsilon}$, the result $f'(D)$ is differentially private. We now define the global sensitivity function $s$.

**Definition 3 (Global Sensitivity)** Intuitively, one can consider the global sensitivity of a function as the largest amount that the output of the function can change between any two databases that differ in only one row. Formally, $s(f) = \max_{d(D,D')=1} ||f(D) - f(D')||$.

**Example 4 (Laplacian Perturbation)** Consider a simple count of the number of records in $D$. The function $\text{count()}$ clearly has sensitivity 1, as each addition or deletion of a record changes the count by at most 1. If the true number of records is 1000 and the noise $n$ is chosen from $\text{Lap}(\frac{1}{\epsilon})$, the probability distribution of the result $f'(D) = \text{count}(D) + n$ is pictured below (see Fig. 2-1). Note that as epsilon decreases, the density function flattens, meaning that more noise is being applied.

Frequently, $s(f)$ must be estimated in order to not require a linear number of operations to determine - such estimation is generally considered acceptable in differentially private systems as long as the estimated sensitivity $s'(f) > s(f)$ [30].
Figure 2-1: Result of $\text{Lap}\left(\frac{1}{\epsilon}\right)$ added to 1000
2.3.2 Sample-and-Aggregate Mechanism

Unlike the Laplacian mechanism, the Sample-and-Aggregate mechanism [25, 32] differs in that the magnitude of noise is not solely determined by the query, but by the contents of the database itself. It is used to approximate the answers to arbitrary functions with an output $\mathbb{R}^d$ which must be fairly accurate when run on small subsets of the data (e.g. the median of a set of numbers). In order to understand the Sample-and-Aggregate framework, one must first understand the concept of local sensitivity.

**Definition 4 (Local Sensitivity)** Intuitively, the concept of local sensitivity is how much the output of the function can vary between the actual database and any database with one row added, removed, or changed. Formally, given the actual database $D$ and function $f$, let the local sensitivity $ls(f, D) = \max_{D', d(D, D') = 1} ||f(D) - f(D')||$.

Note how drastically different this is from the global sensitivity we discussed above! Using local sensitivity, one no longer needs to add noise relative to all possible databases, but only to the actual database; if your data is nicely distributed you can thus get away with adding far less noise and your results will be far more accurate.

**Example 5 (Local Sensitivity: Median)** Consider a query that seeks to find the median record in a dataset. $med(D)$, in general, has extremely high global sensitivity, as a single record could potentially change the median by the range of the entire dataset. For example, the result of $med(0, 0, \ldots, 0, k, k, \ldots k, k)$ could change from $k$ to zero if an additional zero is added, whereas $med(0, k, \ldots, k, k, k, \ldots, k, k)$ would require $\frac{n}{2}$ changes in order to give a result other than $k$. Given the range of these datasets, both queries have a global sensitivity of $k$ and Laplacian perturbation would add noise proportional to $\frac{k}{\epsilon}$, destroying nearly all utility. However, perturbation using local sensitivity would see that in the second example, the median is very insensitive to a change in a single record. Thus, minimal perturbation needs to be applied to the final aggregated result, and utility can be preserved.
Unfortunately, local sensitivity used naively can still result in a privacy leak: consider a malicious transformation that outputs very large values if a certain record is present and low values otherwise. Were noise to be applied to this query based on local sensitivity, the magnitude would vary hugely depending on whether or not the large value was present and the adversary could thus infer the record’s presence from the variance of a small series of repeated queries. Thus, local sensitivity is non-private.

The problems with local sensitivity arise because, while the query’s local sensitivity may be low, the sensitivity of the noise added is very high and a single value is able to greatly change the magnitude of the perturbation applied, leaking information solely by the amount of noise. By using a technique known as "Sample and Aggregate" (SAM) [25], we can "smooth" out the noise by running the user’s query on many representative subsamples and aggregating the results.

**Definition 5 (Sample and Aggregate)** Sample and Aggregate begins by dividing the original dataset into a number of smaller partitions. The analyst’s query \( f(D) \Rightarrow \mathbb{R}^d \) is applied to each partition \( D_i \), and an efficient private aggregation function \( g(f_1, f_2, \ldots, f_k) \) is used to approximate \( f \) using each result \( f_i = f(D_i) \). Finally, noise proportional to the sensitivity of \( g \) is added and the result is returned to the user.

The trick is to choose \( g \) such that accuracy is optimized but the guarantee of differential privacy still holds. [32] provides such an aggregation function which we use in SparkSAM. Below, we describe the function as applied to a query outputting a single real, however it can be trivially extended to a \( d \)-dimensional vector of reals by applying the below algorithm on each dimension independently.

**Definition 6 (Widened Winsorized Mean)** We first run the sampling procedure from above to generate \( m = n^k \) partitions, each of size \( n^{1-k} \), and run the user’s query on each partition to generate a set of outputs \( Z = (z_1, z_2, \ldots, z_m) \). The outputs are sorted and the exponential mechanism below is used to privately select values \((a, b)\) near the top and bottom quartiles respectively. The mean of the two quartiles \( \mu = \frac{a+b}{2} \) and the width \( w = |b - a| \) are computed. We then widen the interval by setting...
\[ l = \mu - 2w \] and \[ u = \mu + 2w. \] All values in \( Z \) are clamped to lie within the range \((l, u)\) and \[ x = \frac{\text{sum}(Z)}{m} \] is calculated. Finally, noise drawn from a Laplacian distribution with \[ \lambda = \frac{u-l}{2\epsilon m} \] is added to \( x \) and the result is returned.

**Definition 7 (Exponential Mechanism)** The exponential mechanism [9] is a method for selecting one element from a range of possible elements \( R \) (note that the element need not be numeric). Given a potential answer \( r \in R \) and a "utility function" \( H(a, D) \in \mathbb{R} \), we extend our definition of sensitivity to \( s(H) = \max_{a \in A} s(H(a)) \). Given that, we can then choose \( a \in A \) based on the following probability: \[ P(a \in A \text{ is selected}) \propto \exp\left(\frac{sH(D, a)}{2\epsilon s(H)}\right). \] Intuitively, choices that have higher utilities have a higher probability to be selected, and utility functions with higher sensitivity increase the randomness of the result.

As long as \( k > 0.3 \), this algorithm produces a differentially private result. The proof for the privacy and utility guarantees for this function are beyond the scope of this thesis, however further discussion can be found in [32].
Chapter 3

Related Work

3.1 PINQ/wPINQ

PINQ [22] is an influential implementation of differential privacy developed by Microsoft Research in 2009, shortly after differential privacy was first proposed. Built as an extension to Microsoft LINQ’s platform, PINQ is a wrapper that allows data analysts to interact with data using a SQL-like language while preserving differential privacy of individual records within the dataset. No privacy knowledge is required by analysts beyond a basic understanding of queries depleting a privacy budget, and data providers can be assured that their data is \( e \)-differentially private regardless of what queries analysts might choose to perform. Through composition of "stable" transformations (Select, Where, GroupBy, Join) and noisy aggregations (Count, Sum, Median, Avg), PINQ enables many statistical queries on structured and unstructured datasets. Laplacian noise is applied at the aggregation step, and aggregations are restricted to real values (conversion functions from records to doubles are an optional parameter to the aggregations). Note that sensitivity is controlled by clamping values to the interval \([-1, 1]\) prior to aggregation, limiting the impact of any single record. A later extension called wPINQ [29] lifts this restriction, instead controlling sensitivity by reducing the weight of records as they are used in transformations, thus limiting their influence on the output at the cost of accuracy in the final result. However, both PINQ and wPINQ remain restricted to relatively simple selections, projections,
and groupings; and remain tied to the proprietary DryadLINQ format.

## 3.2 Airavat

Airavat [30] is a MapReduce-based [4] system which provides differential privacy guarantees for distributed computations. Following the MapReduce model, analysts are able to submit a job consisting of one or more arbitrary mapping functions ($f(k, v) => [(x_1, y_1), (x_2, y_2), ...]$) followed by one or more trusted reducers (aggregations similar to PINQ such as Count, Sum, etc, along with more advanced ones such as K-Count and Threshold). Jobs are written in the familiar MapReduce paradigm and can be run on a dataset of arbitrary key/value pairs, which can be grouped together into individual privacy groups by the provider (e.g. individual lines of a document can be grouped by the document they are from, thus guaranteeing differential privacy on the document level rather than the line level). Similar to PINQ, noise is applied using the Laplace mechanism relative to a global sensitivity estimation and an $(\epsilon, \delta)$ parameter and privacy budget are set by the provider. In order to prevent unacceptably-high privacy budget costs for multiple computations on disjoint portions of an input, partitioning of data by privacy group is possible through a Partition function (for a detailed discussion, see [30]). The analyst is required to specify the range of any untrusted mappers in a given job (enforced at runtime by Airavat), which is then combined with the known sensitivity of the trusted reducers to determine the magnitude of noise to apply. This system allows for more complex computations than PINQ, but places additional requirements on the analyst (specified ranges and output keys) as well as being bound by the performance characteristics of MapReduce.
Chapter 4

Approach

4.1 Overview

To their credit, both systems described above were seminal implementations of differentially private systems that enabled many privacy experiments and private data analyses. However, these systems are (as of this writing) both over 7 years old and are showing their age. Dryad (the backend to PINQ) has been discontinued since 2011 [28], making PINQ and wPINQ impractical for any organization to use in 2017. MapReduce is still in use as part of the Hadoop ecosystem, however its inability to store data in memory and its cumbersome query protocol have caused many organizations to abandon it in favor of Spark. The fact that differential privacy has not yet been widely adopted in industry [19] lends further evidence to the fact that higher performance and scalability is required, and that an implementation on a more modern and widespread platform is needed. Finally, the fact that both systems limit themselves to Laplacian noise (which is applied relative to global function ranges rather than calibrated to the actual data range) and real-valued outputs provide further barriers to mainstream adoption due to lack of accuracy in the system’s results.

It is clear that significant improvements can be realized by leveraging the benefits of the Spark cluster computing framework and of alternative mechanisms of differential privacy enforcement. We present Shade: a differentially private wrapper that can
be easily integrated around an existing Apache Spark cluster. Shade is implemented as a layer atop Spark Core [35], providing differentially private implementations of many of Spark’s core functions for data manipulation and analysis. Shade allows data providers (e.g. LifeSpace) to customize the degree of privacy guaranteed to individuals whose data is contained within the dataset, and allows data analysts (e.g. clients) to learn useful information from the database without compromising user privacy or exceeding the provider-determined privacy budget. We believe that the advantages provided by Shade are compelling enough to reinvigorate industry interest in differential privacy and inspire organizations to enforce this privacy in their data analysis pipelines. Below we first give the set of assumptions and threat model under which Shade operates. We then give an overview of the components of Shade that must be understood by both data providers and data analysts. Finally, we provide a sample workflow and diagram to illustrate the expected operation of Shade.

4.2 Assumptions and Threat Model

While a typical use case was outlined in (1), here we explicitly enumerate our assumptions and threat model. We assume that there exists a secured Spark cluster with a trusted communication network between the nodes within the cluster, maintained by a trusted data provider who has access to the raw sensitive datasets and possesses some expertise on differential privacy. Likewise, we assume that whatever storage engine the data provider uses to store these datasets is also secured, and that no analysts have access to the secret keys necessary to either operate the Spark cluster on their own or access the underlying datasets. For simplicity we assume that datasets are all in a .csv format with one line corresponding to one record - this is a reasonable assumption given that a simple ETL (Extract, Transform, Load) task [1] can be run in Spark prior to data loading which will ensure that this format is followed (we ran several of these ourselves in the evaluation process). We assume that each cluster is accessed in a read-only manner exclusively by Shade programs, that the users do not share the individual Shade programs that the provider has distributed
so as to circumvent the privacy budget restrictions, and that the only escape vector for information is through the output from the user’s submitted query. In short, this thesis does not concern itself with encryption or access control of the underlying dataset and leaves that as an implementation detail for the provider. We believe that this is a fair restriction, as a number of different access control mechanisms already exist (Kerberos, OAuth, etc) and we would not want to limit the utility of this system by including a dependency on specific security software. Note that this is a somewhat weaker threat model than Airavat (which includes a mechanism for file encryption and encryption of intermediate MapReduce files) but seems a fair one in light of the increasing variety of ways in which organizations store and secure their data.

4.3 Configuration

Certain components of the application must be first configured by the data provider before releasing the system to analysts. The data provider is in charge of acquiring the original private dataset(s), converting them into a format that can be read by Spark, and communicating Spark-accessible filepaths to the data analysts. The method of
storage of these datasets is outside the scope of this thesis, however we recommend using a distributed file system such as HDFS for performance reasons (6.2.2). Obviously, the data provider is also in charge of maintaining the Spark cluster that Shade is wrapped on top of, as anyone with direct access to the cluster will necessarily have access to the raw data in the process of configuring the system. Finally, the data provider is in charge of choosing $\epsilon$ (which controls the amount of perturbation applied to results) and $\text{scale}$ (which controls the number of information releases allowed before access is restricted). As discussed in (2.2), these parameters will vary from application to application and it is recommended that the data provider become familiar with some amount of privacy literature so as to be informed about best practices for choosing privacy parameters.

4.4 Usage

Once the data provider has performed the necessary configuration, data analysts can then begin making queries using their individual privacy budgets. A Shade query looks almost identical to a Spark query - the user first must create a SparkContext (in this case, a PrivateSparkContext) that provides a name for the application and various other details. From there, data analysts can instantiate one of two RDD types that provide different "views" of a dataset depending on the needs of their application. Various transformations can be lazily executed on the RDD using the Scala programming language, and information can be extracted from the RDD through a set of actions. The two views (SparkLAP and SparkSAM) share a privacy budget and some application data, but have very different perturbation mechanisms and use cases: we give a brief overview of each below.

SparkLAP has essentially identical functionality to Airavat [30]. Like Airavat, SparkLAP allows for arbitrary user mappings, filterings, and key-value groupings, followed by a single "trusted reducer" that returns a noised version of some numerical aggregation (e.g. count, sum, average). Also like Airavat, SparkLAP has a number
of constraints that are placed on users: only numerical values can be returned, sev-
eral Spark core operations (such as sorting) are disallowed, and users must specify
the output range of both keys and values. Failing to accurately specify any of these
constraints can result in high inaccuracy in results, as SparkLAP has a number of
prevention mechanisms built in to prevent information from being leaked to adver-
sarial analysts. The primary purpose of SparkLAP is to recreate full functionality of
previous systems in a modern framework, to provide performance improvements, and
to increase the ease-of-use of differential privacy. We show in (6.4) that SparkLAP
accomplishes these goals.

SparkSAM is a novel implementation of the Sample and Aggregate framework of
[25, 32]. Designed for queries that have high accuracy when run on random sam-
ples, SparkSAM partitions the raw dataset into many small partitions and runs the
user’s query on each sample, aggregating the results using the widened Winsorized
mean algorithm from (6). As privacy is applied at the aggregation step, rather than
via the sensitivity estimation of SparkLAP, many of SparkLAP’s requirements and
limitations can be lifted for SparkSAM. This can further reduce the amount of code
necessary for the analyst to write, as well as unlock certain classes of queries that
SparkLAP cannot accomplish at all. We show that on well-distributed datasets, accu-
ricy of high-sensitivity queries is vastly improved over SparkLAP. We also show that
in some cases (e.g. global sort and median finding), performance of SparkSAM can
actually exceed even that of non-private Spark and that future optimizations such as
locality-sensitive partitioning could provide even further improvements (7.2).

4.5 Sample Workflow

The following sections describe the work required to start using Shade from the per-
spective of a data provider and a data analyst. Note that this section omits many
details about configuring Spark, compiling programs using SBT, etc - refer to the
documentation for these tools for more details.
4.5.1 Data Provider

A data provider only needs to configure Shade once per organization and it should then be mostly a matter of maintaining the underlying cluster. However there are a few steps that must be taken at initial setup. First, the data provider must generate a JAR of all of Shade’s dependencies (specifically [10], see (5.4.1)) using SBT’s assemblyPackageDependency command and copy this JAR to an arbitrary path on each worker node. This allows users to avoid having to include a fully-compiled SBT in their programs, which would be hundreds of megabytes large. Once the dependencies are copied, the data provider must distribute that path to users so that they can modify their submission commands to include it (4.5.2). At the moment, note that Shade itself must be instead included as a part of the actual driver JAR submitted by users and thus the source code of Shade must be distributed to each user individually (e.g. through Github). A preferable method would be to also include Shade itself as a dependency and publish it to a Java package manager, preventing the need for analysts to perform this download.

4.5.2 Data Analyst

Data analysts must then download the source code for Shade and use SBT to package a Spark driver program that includes the files for Shade itself with Shade’s dependencies listed as a provided dependency (as mentioned above, this approach is less than ideal). Once the program is written, the spark-submit command can be used to submit the query using the address of the master and the filepath of the dependency JAR as a dependency parameter. Examples for all of these operations can be found in Shade’s Github repository in the examples folder.
Chapter 5

Implementation

5.1 PrivateSparkContext

As mentioned in (4.4), an analyst’s first step when using Shade is to create a PrivateSparkContext. A PrivateSparkContext is an object that contains information about the overall Shade query. This includes information about the underlying Spark query (currently only the query’s name in the Spark web UI) as well the various privacy parameters set by the Data Provider and a reference to the user’s Budget.

A PrivateSparkContext has two main methods:

getLapRDD(filepath : String) : Lap_RDD[String]

produces an RDD of the file at filepath manipulable in SparkLAP, and

getSamRDD(path : String) : Sam_RDD[String]

produces an RDD of the file at filepath manipulable in SparkSAM.

Recall from [34] that RDD interfaces can be created from arbitrary file paths (local files, HDFS files, etc) and multiple RDD interfaces of each type can be simultaneously active. Each RDD begins as an RDD of Strings, with each element corresponding to a single line of the original text file, but can then be affected by transformations and
actions as discussed below. Note that the passed in filepath must be reachable from all nodes in the cluster, as well as the master node, likely in some form of distributed file format e.g. HDFS. Spark can then take care of ensuring that each worker node fetches only their assigned portions of the RDD, and balancing the workloads to maximize throughput of the overall query given the cluster’s physical constraints. Each RDD also contains a reference to the same Budget instance, which is created at the same time as the original SparkContext; queries from any of them debit that same Budget, enforcing a global Budget limit (5.2). Finally, once all of the user’s desired queries have been written, the user must call `stop()` on the PrivateSparkContext so that the cluster can release the query’s requested resources for other waiting Shade jobs.

We feel it is worth pausing a bit here to discuss the internals of PrivateSparkContext, as many of the below classes which also wrap Spark functionality implement the same design pattern. Internally, a PrivateSparkContext is actually a "forwarding class" [3] that contains a reference to a SparkContext delegate. When a PrivateSparkContext is instantiated, it forwards that call to its delegate, thus instantiating the SparkContext as well. The same principle applies to creating RDDs: `getLapRDD` and `getSamRDD` both actually call the native Spark `textFile` method on the SparkContext delegate to construct native Spark RDDs, which are themselves then passed as delegates for the various RDD classes in SparkLAP and SparkSAM. In essence, when we say that Shade is a wrapper around Spark, we mean it: Shade requires no additional libraries or changes to the underlying Spark configuration because it actually forwards much of its work to the native functionality of Spark itself.

5.2 Budget

Recall that any differentially private system, no matter the value of $\epsilon$ chosen, must still limit the amount of information released to any given user in order to prevent the user from obtaining sufficient samples to infer a query’s true value [6]. This is
done by configuring a privacy budget in addition to an epsilon value. Each unit of "information release" debits a cost of $\epsilon$ from the overall budget, where the magnitude of an information release is defined as the maximum contribution of each line to the resultant output(s). Results that only take a single contribution per line (such as those using map, filter, or groupBy) only debit a single $\epsilon$, while results that can potentially have $k$ contributions from a given line (such as those using groupByMulti) necessitate $k$ debits of $\epsilon$. We note that it is possible for even queries with multiple values outputted to only have a single $\epsilon$ debited for the entire query (e.g. queries which use groupBy to partition lines by arbitrary keys and then output a set of keyed aggregations). Eventually, when the entire budget is exhausted, the system must prevent any further information releases, effectively rendering it useless.

We implement this behavior by having a single Budget instance shared across all RDD views. Each RDD keeps a reference to the Budget and the driver program debits the appropriate cost each time output(s) are released. If the cost cannot be debited, the RDD is responsible for cascading an exception back to the caller indicating that the requested operation has failed. By waiting until the result is generated to debit the Budget, we prevent concurrency issues, as the machine running the driver program is the only component of the cluster that actually interacts with the Budget. As the Budget instance is a public final parameter on the PrivateSparkContext class, clients if they so choose may query the amount of Budget remaining at no additional cost so as to better plan their remaining queries. (Note that this output is safe as the budgetary cost of a query is entirely insensitive to the database contents for both SparkLAP and SparkSAM).

Finally, it bears mentioning that the current implementation of Shade generates a new Budget instance each time a PrivateSparkContext is instantiated, potentially allowing analysts to subvert the budget functionality by creating many different PrivateSparkContexts in a single session. This attack could be trivially prevented by e.g. assigning each user a unique key and tying a persistent Budget instance to that key -
we choose to leave this security implementation to individual organizations following
the logic discussed in (4.2).

5.3 SparkLAP

Recall that SparkLAP is designed to replicate the functionality of Airavat at improved
performance and with the convenience of Spark. Before we discuss the interface and
implementation, we briefly refresh two important Spark concepts [35]: Single RDDs
vs Pairwise RDDs and Transformations vs Actions.

Spark makes the distinction between Single and Pairwise RDDs (unlike MapReduce
[4] which operates only on key-value pairs). Single RDDs can be thought of as a sim-
ple sequence of elements: an RDD[T] is thus a list of elements of type T distributed
across the cluster. Pairwise RDDs are what is generated when a single RDD has
a grouper executed on it (e.g. groupBy): an RDD[K, V] is thus a Map from each
unique key of type K to a list of values of type V. While a Single RDD can only have
Actions applied on the entire dataset, a Pairwise RDD can have actions applied on
individual keys - this is useful for many kinds of queries where an analyst wants to
learn information about \( n \) different keys without expending \( n \times \epsilon \) privacy budget.

While Spark Core obviously doesn’t have the direct concept of mappers and reducers
that MapReduce does, it does have a different distinction between Transformations
and Actions. The basics of this distinction are discussed in (2.1), but we repeat
them here. A Transformation (e.g. map, filter, groupBy) takes an arbitrary user-
provided function and applies it to each element in the dataset, creating a new RDD
from the old. Note that Transformations are lazily applied, such that an analyst can
chain multiple Transformations in a single query and the Spark scheduler will stack
them appropriately to avoid unnecessary scans of the full dataset or data shuffles
across machines. Actions, on the other hand, actually return something to the user
or have some other side effect (such as caching) - thus they necessarily result in ma-
terialization of the dataset and cannot be lazily applied. While the default Action in Spark Core is simply `reduce` (which takes arbitrary functions to combine all elements of an RDD), like Airavat SparkLAP requires Trusted Actions that have a known effect and are pre-defined as part of Shade. This is because noise is only applied at the time of materialization; when an Action is called, SparkLAP computes the result and then adds Laplacian noise scaled by the estimated sensitivity of the query. This sensitivity is inferred depending on the Action chosen as well as the size and range of the dataset.

Below, we discuss all Transformations and Actions supported by SparkLAP, along with the mechanisms for privacy enforcement and noise addition.

### 5.3.1 Interface

#### Single Transformations

Without loss of generality, we assume that all functions are called on a `Lap_RDD[T]`.

- **Map** \((f : T \Rightarrow U) : Lap_RDD[U]\)  
  Takes a function \(f\) and applies it to each item in the input RDD, returning a new RDD of transformed elements.

- **Filter** \((f : T \Rightarrow Bool) : Lap_RDD[T]\)  
  Restricts the input RDD to only elements \(e\) for which \(f(e)\) is true.

- **GroupBy** \((f : T \Rightarrow K) : Lap_PairRDD[K, V]\)  
  Creates a pairwise RDD from a traditional RDD where the keys are generated from a user-provided \(f\). A simplified version of `GroupByMulti` for cases where each element generates exactly one key.

- **GroupByMulti** \((f : T \Rightarrow Seq[(K, V)], outputs : Int) : Lap_PairRDD[K, V]\)  
  Creates a pairwise RDD from a traditional RDD where the keys and values are gen-
erated from a user-provided $f$. Each element must output at most outputs pairs; enforcement for this is done at runtime (5.3.5). While this function is not in native Spark Core, we add it to replicate the full functionality of Airavat (as this is identical to a Mapper in MapReduce).

$\text{Distinct}() : \text{Lap\_RDD}[T]$

Creates an RDD of all and only unique $T$ in the original RDD. Note that this functionality is actually not in the original Airavat function set, but as it unlocks a new set of operations without compromising any privacy guarantees, we have decided to add it.

**Single Actions**

$\text{Count}() : \text{Double}$

This function returns a noised version of the number of elements in the RDD. We note that $\text{count}$ is a special case as compared to other aggregations in that its result does not depend on the contents of any of the values in the database, only the number of values. Thus, count can be called on RDDs of any type and still has sensitivity of exactly 1, as each element can contribute at most 1 to the overall count. Also note that the sensitivity estimations for this and all later Actions can be potentially scaled up if the stability of the query is increased - see (5.3.5) for more details.

$\text{Sum}() : \text{Double}$

This function returns a noised version of the sum of all elements in the RDD. Like all other Actions besides $\text{count}$, $\text{sum}$ can only be called on numeric RDDs (5.3.2) as its sensitivity depends on the Range given for the RDD (5.3.3). Specifically, the sensitivity of the sum of a set of RDDs is the max absolute value within the Range [30], as this is the most that the sum can be affected by a single element.

$\text{Avg}() : \text{Double}$

This function returns a noised version of the average of all elements in the RDD. The
same numeric restriction as `sum` applies to `avg`, except in this case the sensitivity of `avg` is instead the width of the Range divided by the number of elements in the RDD [29]. Note that if naively applied, this is the one Action in which the magnitude of noise could indeed be dependent on the contents of the database and thus leak information! We prevent a privacy leak in this case by using the noisy count instead of the actual count when estimating sensitivity, thus ensuring that the noise magnitude is itself private as well.

**Pairwise Transformations**

WLOG, we assume that all functions are called on a `Lap_PairRDD[K, V]`.

`MapValues(f : T \Rightarrow U) : Lap_PairRDD[K, U]`

Identical to `Map`, but applied on all values in pairwise RDDs leaving key mappings untouched.

`FilterValues(f : T \Rightarrow Bool) : Lap_PairRDD[K, V]`

Identical to `Filter`, but applied on all values in pairwise RDDs leaving key mappings untouched.

`get(key : K) : Lap_RDD[V]`

Get a `Lap_RDD` of all values for the given `key`.

**Pairwise Actions**

`KCount() : Double`

Identical to `Count`, but returns a map of noisy counts for all keys that have been specified by the analyst (5.3.4).

`KSum() : Double`

Identical to `Sum`, but returns a map of noisy sums for all keys that have been specified by the analyst. Note that Ranges may be specified per-key or globally, potentially
giving different sensitivities for each key (this also applies to $K\text{Avg}$).

$K\text{Avg}() : \text{Double}$

Identical to $Avg$, but returns a map of noisy averages for all keys that have been specified by the analyst.

### 5.3.2 Type Enforcement

Necessarily, some of the actions above ($\text{sum}$, $\text{avg}$, $k\text{Sum}$, $k\text{Avg}$) must be called on RDDs of numbers, as it doesn’t make sense to have e.g. a sum of non-numeric elements. However, we want to keep this detail invisible to the user: the analyst may have a query going back and forth between numeric and non-numeric elements and we want the transition to be seamless without any additional type-specific methods that the client needs to call. We also don’t want to have to eagerly inspect the type of an element in the RDD prior to each Action as this must result in materializing the RDD, which will have negative performance implications.

We instead use Scala’s reflection features [26] to implicitly inspect the output type of the function passed to the most recent Transformation at runtime; we then store this class in the child RDD and inspect that as necessary when an Action is called (obviously this does not apply to type-preserving Transformations such as $\text{filter}$). If the inspected type is Numeric and the Action requires a numeric RDD, then the Action is allowed to proceed, otherwise the Action is rejected and an Exception is thrown. This allows us to keep a simple interface for the analyst without sacrificing either performance or functionality.

### 5.3.3 Range Enforcement

As discussed in [30], in order to be able to accurately bound a query’s sensitivity it is necessary for a range of values to be provided and enforced for any aggregation Actions that depend on the values of elements (in this case $\text{sum}$, $\text{avg}$, $k\text{Sum}$, $k\text{Avg}$).
We enforce this through the `setRange` method on single and pairwise RDDs, which allows a user at anytime to create and specify a Range for values in a single RDD or values for specific keys of a pairwise RDD. A Range is very simple: it consists of an upper and lower bound (as well as some inferred values such as the width or a method to draw a random value). As with (5.3.2), we implicitly restrict this method to only be permitted on RDDs of numeric type. While RDDs can be converted to and from numerics without having an associated Range, before any non-`count` Actions may be called, some Range must be assigned, otherwise the element Range will be presumed to be infinitely large and all utility will be destroyed by noise.

Once `setRange` is called, SparkLAP "enforces" the Range by clamping any values that lie outside of the Range to a randomly chosen value within the Range. As in Airavat, the Range can then be used to estimate the global sensitivity using the worst-case bounds for the values of various Actions - see (5.3.1) for examples. Thus, the accuracy of the Range is fairly important, as too-narrow Ranges will result in undesired value-clamping and too-wide Range will require additional noise, harming utility. For data in which the analyst is unaware of the Range it may be a good idea to use another tool such as SparkSAM to inspect e.g. quantiles before running a SparkLAP query, or the data provider could provide estimates of various column Ranges on an as-needed basis. Finally, note that enforcement is actually performed using more Spark primitives (specifically an additional `map` stage), thus minimizing performance costs of SparkLAP. This also applies to the later forms of enforcement discussed below.

### 5.3.4 Key Enforcement

Obviously, any differentially-private system cannot return arbitrary user-defined keys - consider an adversarial query that outputs a long String of the entire contents of a row, for example. Thus, it is necessary to restrict the set of keys output for pairwise Actions. We do this by providing a `setKeys` method (similar to the `setRange`) method that must be called before any of the pairwise Actions can be invoked. Only keys in
the provided set of keys will have values returned for the pairwise Actions, and each key will have a plausible noised value assigned if the dataset has no elements for that particular key. This ensures that information is not leaked either by the presence or absence of a given key.

5.3.5 Stability Enforcement

For this section, we use a variant of the concept of stability from [22]. For the purposes of SparkLAP, we say that a query has stability $x$ if any given input record contributes to the result of the query at most $x$ times. With the exception of \texttt{groupByMulti}, all Transformations in SparkLAP have stability of 1; in \texttt{map} for example, each input record results in exactly one child record, and in \texttt{filter} each record results in one or zero child records. However, for \texttt{groupByMulti}, recall from (5.3.1) that each input record can result in up to outputs key-value pairs; thus a \texttt{groupByMulti} Transformation has stability of outputs. [22] also provides a proof that chained queries have multiplied stabilities, which extends to our modified definition - if a \texttt{groupByMulti} Transformation with number of outputs $s$ is applied to an RDD that already has stability $t$, the new stability will be $st$.

Since SparkLAP uses global sensitivity, we must assume that if an input record can potentially influence an output by $n$ times, we must apply noise scaled by a multiple of $n$ to preserve the privacy bound [8]. Thus, we keep track of the stability history of an RDD throughout its transformation lifecycle and multiply the Laplacian scaling factor by that amount prior to adding noise.

5.3.6 Sample Queries

Here we present two queries to get the average of a given movie’s ratings in the Netflix dataset. The first is implemented with Single Transformations and Range Enforcement, and the other is implemented with Pairwise Transformations and Key Enforcement.
First Query:

```scala
val movie_id = "1"
val rdd = ctx.getLapRDD("hdfs:///path/to/dataset")

val average_rating : Double = rdd
  .map(x => x.split("\,""))
  .filter(x => x(0) == movie_id)
  .map(x => x(2).toDouble)
  .setRange(new api.Range(0, 5))
  .avg()
```

Second Query:

```scala
val movie_ids = List("1")
val rdd = ctx.getLapRDD("hdfs:///path/to/dataset")

val average_rating : Map[String, Double] = rdd
  .map(x => x.split("\,""))
  .groupBy(x => x(0))
  .setKeys(movie_ids)
  .setRangeForKeys(movie_ids, new Range(0, 5))
  .kAvg()
```

### 5.4 SparkSAM

Unlike SparkLAP, recall from (4.4) that due to the removal of the need for sensitivity estimation, SparkSAM is able to fully support the entire Spark Core set of Transformations and Actions listed in [34]. This includes support for sorting of values and for untrusted Actions. We also add a custom Action for discovering the median of a transformed dataset, as this is one of the key features advertised by [25, 32], although
it’s possible this could be accomplished with a (somewhat complex) analyst reduction as well. At time of writing, only the portions of the Spark Core library that are actually used in our experiments have been implemented, however adding additional operations can be done in a single line per operation and is trivially accomplished.

However, also recall that the "catch" is that not every query will return meaningful results. SparkSAM is designed for black-box queries that can be run on representative subsamples of the data and still produce an accurate answer; the Sample-And-Aggregate mechanism then takes the many (hopefully similar) samples and adds noise to a "private average" of the samples to produce the final result. What this means is that while statistical queries such as \texttt{mean} and \texttt{median} will work quite well on SparkSAM, queries that don’t scale to small samples such as \texttt{count} and \texttt{max} will result in answers that don’t make any sense. For queries that do fit the mold though, SparkSAM is able to apply noise calibrated to local sensitivity rather than SparkLAP’s global sensitivity and can frequently achieve far better than this worst-case estimation.

Below, we discuss the mechanisms for partitioning, aggregation, and noise addition.

\subsection*{5.4.1 Partitioning}

Recall that from [25] that, in order for Sample-and-Aggregate to be effective, there must be a sufficiently large number of partitions of the data. [32] shows that given $n$ records, any number of partitions $n^k$ with $k > 0.3$ should be sufficient; for this thesis we chose to use $k = 0.4$. Thus, we want to first split the dataset into $n^{0.4}$ partitions of size $n^{0.6}$, then perform the user’s entire query on each partition, and finally aggregate the results on a single machine and return to the user.

Our initial implementation of SparkSAM leveraged the Spark Core \texttt{randomSplit} method using $k$ as the randomization parameter. Unfortunately we experienced very poor performance on large datasets due to the $O(n^{1+k})$ runtime - using a third-party
function \texttt{splitSample} instead \cite{10}, we were able to realize $O(n)$ runtime for splits. \texttt{splitSample} is called once at the time of RDD creation and the sequence of pointers to the partitioned RDDs is then passed to the \texttt{Sam\_RDD} and stored as a delegate, preventing unnecessary repartitioning in extended queries.

We further optimized by transferring all data for each partition to a single machine prior to any query execution through the use of \texttt{coalesce}. This allows true parallelization in that once the initial data partitioning is complete, each partition resides entirely on a single machine, allowing that machine to devote its full computing resources to the query’s operations on this partition rather than the usual shuffle of data that accompanies a typical Spark job.

Finally, once the initial partitioning is complete, Transformations performed on a \texttt{Sam\_RDD} are then forwarded in parallel to each of the many different child partitions contained within the \texttt{Sam\_RDD}’s delegate. When run on a driver node with sufficient cores to communicate with all slave nodes in separate processes, even expensive queries can be run highly efficiently - in some cases more efficiently than even vanilla Spark Core (6.4).

\subsection{Aggregation}

When an Action is invoked on a \texttt{Sam\_RDD}, the Action is first invoked on each partition to produce either an $n$-dimensional vector of reals for each partition. Each vector is then sent to the driver program, and the resultant list of vectors is then aggregated. The aggregation algorithm used is described in detail in \cite{6} and \cite{25, 5}, so we will not reproduce it here. However, we note that, unlike SparkLAP, it is necessary for the privatization of the result to be done in multiple steps, meaning the data provider must choose how to divide the privacy payment even for a single query. For this implementation, we chose to spend $\epsilon/4$ on each of the two \texttt{PrivateQuantile} calculations, and the remaining $\epsilon/2$ on the Laplacian noise added to the Winsorized Mean, matching the default per-query privacy expenditure of SparkLAP at $\epsilon$ per
5.4.3 Sample Queries

Below, we present a query for finding the median rating of all ratings in the Netflix dataset. Note that this uses the custom-written `median` function, and at the moment is impossible to use with just an aggregation due to the need for the exact count to lookup the median element.

```scala
val rdd = ctx.getSamRDD("hdfs:///path/to/dataset")

val median : Double = rdd
 .map(x => x.split(","))
 .map(x => x(2).toDouble)
 .median()
```

We also present a query which uses a custom aggregation to find the maximum line length:

```scala
val rdd = ctx.getSamRDD("hdfs:///path/to/dataset")

val max_length : Double = rdd
 .map(x => x.length())
 .fold(0, math.max)
```

5.5 Adversary Prevention

Consider an adversary Eve who has auxiliary information that exactly one person with the name "John Doe" is in a dataset of company salaries, and wants to find out how much money John makes. A differentially private system should obviously prevent
her from being able to detect this; any query that allows Eve to discover Bob’s salary should thus be treated as a blatant violation of privacy. We now examine a few ways in which Eve might try to discover the salary and how Shade can prevent it using the previous sections’ enforcement techniques. Note that since SparkSAM’s potential for information leakage is far easier to bound due to the black-box nature of queries (if any query is private, they all must be so), we discuss only SparkLAP below. Assume that $\epsilon = 0.1$.

The naive approach would be for Eve to use the following query: first, filter out any records whose name does not equal "John Doe", then find the average salary of the result RDD. As the RDD is of size 1, the average will thus give John’s exact salary (note that the `sum` Action would work equally well). Since Eve did not provide a Range for the query, SparkLAP assumes the Range is infinitely wide and refuses to allow the `avg()` Action, preserving John’s privacy.

If Eve then sets the actual Range of the salary (say, 0 to $10^5$), SparkLAP will first calculate the width of the Range as $10^5$. Since only one record is in the dataset at the time of the `avg` Action, the sensitivity will also be $10^5$ (for a reminder of why this is the case, see 5.3.1). SparkLAP will thus add noise from a Laplacian scaled by $\frac{10^5}{0.1} = 10^6$, completely destroying the utility of the data and preventing Eve from learning anything at all about John’s salary. If Eve then decides to use a fake Range to minimize the width (say, 0 to 1), then John’s salary will be coerced to a random value within the Range and utility will again be destroyed.

Imagine that Eve tries an alternate approach: she first transforms each record into a (name, salary) tuple and then runs a `groupBy` query using this tuple as the key and the salary as the value. If she then outputs a `kAvg`, she thinks, she can just look for the key containing "John Doe" and see the salary right next to it! Unfortunately for Eve, recall that SparkLAP does not allow arbitrary user-defined keys and so this query will also be rejected.
Not to be deterred, imagine that Eve then calls \texttt{setKeys} with $10^5$ possible tuples, each containing a unique salary, in the hopes of being able to distinguish the one that is actually present. The obvious approach of "look for the one key that is output" won't work, as SparkLAP always creates and returns mappings for each possible key, even if it is not present in the actual RDD. Furthermore, values for non-present keys are either zero (in the case of \texttt{sum}, \texttt{count}) or randomly chosen from the provided Range (in the case of \texttt{avg}). Either way, we claim that John's privacy is still preserved.

Two values 0 and $x$ are certainly $\epsilon$-indistinguishable if $10x$ Laplacian noise is added, which would be the case for \texttt{sum}, \texttt{count}; and obviously a random value in the Range $(0, x)$ would be indistinguishable from John's salary given $10x$ noise as well. Thus, finding the key that actually contained John's salary would be impossible and privacy is still preserved.

Eve then tries to alter the stability of her query. For the purposes of this exercise, assume Eve has even stronger previous knowledge; she knows that John's salary is definitely either 50000 or 100000. Recall that \texttt{groupByMulti} can generate multiple \texttt{(K, V)} output pairs for each input record. If Eve constructs a malicious query that returns 1000 identical ("John Doe", 1) pairs for an input of ("John Doe", 100000), and nothing otherwise, she can then get the \texttt{sum} of the "John Doe" key; if the answer is 1000 then she assumes that his salary is 100000, otherwise she assumes it is 500000.

However, recall that SparkLAP increases the noise applied for queries of higher stability by using the max outputs per record - in this case, with a stability of 1000, the noise would be scaled by 1000 as well and the utility would again be destroyed. Similarly to previous enforcement mechanisms, an absent or incorrect number of max outputs provided will result in either query rejection or truncation of any results that don't conform to the expected value, preventing that attack vector as well.

Finally, recall that both the magnitude of noise applied and the amount of privacy budget debited are derived entirely from the contents of the query and not from the
contents of the database, preventing those common attacks \cite{12} as well.

5.6 Known Vulnerabilities

5.6.1 Function Impurity

Note that the privacy calculations for both SparkLAP and SparkSAM assume that all functions are free of side effects. Being free of side effects means that functions must have no effects at all beyond returning the value specified in the function header. Side effects such as printing statements, persisting data to disk, or even accessing system time could easily leak information to an informed adversary (imagine a query that printed out the contents of every line, for example). Airavat \cite{30} implemented this no-side-effect enforcement through a combination of a secure OS (preventing writing or reading from system files entirely) and modifications to the JVM (removing any functions that could be used to cause side effects). At the moment, Shade does not implement the same level of enforcement and is thus vulnerable to side effect attacks. However, we believe that a combination of two features could be used to prevent side effects without the need for either a custom JVM or a custom file system, both of which are highly burdensome to a data provider.

We first note that any attack vectors that rely on caching values in the JVM or exporting values in stdout would not work in a typical Spark cluster anyway, as the values would only reside uselessly in the memory of a node rather than being returned to the client: this means that utility does not have to be lost by unnecessarily restricting e.g. \texttt{println}. By instead configuring a SecurityManager \cite{16}, we believe that it is possible for a data provider to prevent access to an arbitrary set of classes, which can be used to restrict any classes or functions that might invoke actual side effects (e.g. Broadcast variables and Accumulators in Spark). \cite{13} suggests that clever tuning can bound the performance penalty of the SecurityManager significantly, implying that the performance metrics of (6.4) may continue to hold, however further investigation
is out of scope for this thesis.

The second mitigation technique is to use the Spores [23] library to prevent user code from accessing or modifying variables beyond those explicitly declared as parameters. A malicious user could potentially create some closure (a variable declared outside a function but accessible within) that is mutated in response to inputs and read from in order to modify outputs. Spores instead requires the client to explicitly specify in the function declaration precisely which variables will be accessed as part of the function’s operation - Shade can then use reflection at runtime to verify that the parameters declared are only those that are present in the function declaration, and reject any queries that don’t conform to this restriction. A proof-of-concept "Cleaner" function is implemented in this thesis that demonstrates using reflection to audit the set of variables visible to the function, however integration of the Spores environment would be necessary to actually enforce this audit. The cleaner function can be found in the Cleaner.scala file in the project repository https://github.mit.edu/dig/Shade.

5.6.2 Accessible Endpoints

At the moment, Shade is actually accessed directly by the client, by submitting a jar containing the Shade library and the user’s desired code to the master node of the cluster. Amusingly, this could also allow information to be leaked to a user who submits a regular Spark job that doesn’t use Shade at all, and instead just opens an RDD to the file directly. This can be trivially prevented by the data provider e.g. creating a web portal for submission of jobs and verifying that the jobs actually use Shade instead of vanilla Spark, however we leave that implementation to the discretion of individual providers.

5.6.3 Timing Attacks

Recall that we explicitly excluded timing attacks [12] from our threat model as they are harder to derive a one-size-fits-all solution for. At the moment, like PINQ and
Airavat, our system is indeed vulnerable to nearly every kind of timing attack - even a
naive attack that simply looped endlessly upon encountering a particular value could
be easily exploited for blatant non-privacy. One possible approach could be returning
arbitrary (yet realistic) values for queries that extend past a certain time $t$ (along
with noising $t$ as well!), or perhaps doing this on a per-stage granularity, but this too
we leave at the discretion of the data provider.
Chapter 6

Evaluation

Here we discuss the various dimensions on which Shade should be evaluated, the details of that evaluation, and the results.

6.1 Goals

6.1.1 Performance

The primary goal of Shade was to demonstrate improved performance over existing differentially-private systems, so as to further entice organizations to adopt privacy principles. As DryadLINQ (the engine for PINQ [22] and wPINQ [29]) has been discontinued for many years, we decided to focus our performance comparison on Airavat [30]. For a variety of reasons (primarily lack of access to a secure OS or to Airavat’s modified JVM) we were unable to directly run an Airavat system for comparison against Shade. Instead, we compare Shade against a MapReduce cluster directly, without any privacy wrapper; we then use these measurements to infer our performance against Airavat by adding in the overhead of Airavat given in [30].

We initially planned to focus the majority of our performance comparisons on SparkLAP vs MapReduce, as SparkLAP is the portion of Shade designed to provide increased performance at identical functionality. However, we also compared both
SparkLAP and SparkSAM against a Spark Core cluster (aka Spark without using the Shade wrapper) in order to benchmark the costs for an organization that has already adopted Spark. We vary both source datasets and query workloads in our experiments, as the performance of a cloud computing system can change hugely across both dimensions.

### 6.1.2 Privacy and Accuracy

Although performance is the primary goal of Shade and the perturbation mechanisms themselves are not novel, it is still important that the mechanisms are implemented correctly. We ran two queries of known sensitivity on SparkLAP and recorded sufficient results (with the privacy budget disabled) so as to be highly confident that $\epsilon$-differential privacy is preserved. While we were unable to complete an equivalent number of samples on SparkSAM to achieve suriety of privacy preservation (due to the far larger datasets and longer runtimes necessary for SparkSAM to be effective), we collected a smaller number of results on a pseudo-uniformly distributed dataset which hints that differential privacy is preserved. Finally, we also examine the resultant values to ensure that utility is still preserved despite the privacy perturbations, and compare utility of identical queries on SparkLAP and SparkSAM.

### 6.2 Configuration

#### 6.2.1 Cluster

Shade was evaluated on a cluster of 15 slave nodes and one master node provisioned using the Openstack cluster management system. All nodes were equipped with two Intel cores, 64GB of storage space and 8GB of RAM, were linked with a 10-gbit Ethernet connection, and were co-located in the MIT Stata Center. A different machine (also co-located) was designated as the "client node" and was used for interacting
with the cluster; the client node was provisioned with 32GB RAM and 16 Intel cores. We note that a client machine with a number of cores at least equal to the number of slave nodes was found to result in more efficient communication with slaves when using SparkSAM - this is likely due to the inability of Spark to manage parallel stages and there possibly exists a better solution than our approach of relying on the native Scala parallel collections to force driver-level parallelism.

6.2.2 Storage and MapReduce

Files were stored using the Hadoop Distributed File System with a replication factor of 3 and a block size of 64MB, managed by the YARN resource navigator. Early experiments by the author found that a Network File Storage system was capable of almost equal performance when only using SparkLAP, however due to the need for frequent random-access reads during the partitioning phase of SparkSAM, HDFS was a significantly more efficient choice for SparkSAM. Thus, all experiments in this section were performed using HDFS storage.

Hadoop MapReduce was also configured on the same cluster as part of the YARN framework, however it was deactivated while running Spark jobs so as not to cause competing resource demands. MapReduce, HDFS, and YARN were all installed as part of the Hadoop 2.7 distribution.

6.2.3 Spark

Spark version 2.1.0 (pre-built for Hadoop 2.7) was run on the master and all slaves, using Spark Standalone mode (and was deactivated when running MapReduce jobs). While Shade is designed to be used in "client" mode, with the submitting machine directly communicating with all slaves to drive the program, "cluster" mode was used for some repetitive experiments with the client machine manually designated as the driver program. A single third-party library used for partitioning and discussed in [10] was declared as a third-party dependency. Both Shade and the third-party
dependency were copied to all nodes as an optimization to prevent the need for copying at program startup. SBT was used to assemble a JAR of the user’s application on the client machine and the native Spark submission protocol was used to submit the job to the master node for running on the cluster (see (5.6) for discussion on this).

6.3 Datasets

For our experiments, we used two datasets: the AOL dataset and the Netflix dataset. We discuss both below.

6.3.1 AOL Dataset

The AOL dataset [31] was released by America Online in 2006 and consists of approximately 35 million searches conducted by various users on the AOL search engine along with links they chose to follow. AOL replaced each user’s username with an anonymous identifier and stripped out location information, but left the full contents of the searches and dates in the raw dataset. Shortly after the dataset’s release, the New York Times was able to use specific searches (“landscapers in Lilburn Ga”, "people with last name arnold") to identify user 4417749 as 62 year-old Thelma Arnold from Georgia [2]. AOL later discouraged any further analysis of the raw dataset due to concerns about far more personal searches being linked to real identities. Due to the sensitivity and scale of the data, the AOL dataset presents a compelling use case for private data analysis systems and is analyzed in the Airavat system [30].

The dataset is formatted as a tab-separated file with each row structured as follows:

user_id, text_of_search, date, clicked_link_id, clicked_link_url

6.3.2 Netflix Dataset

The Netflix dataset was released by Netflix also in 2006 and consists of approximately 100 million ratings given to movies by Netflix users. Like AOL, Netflix replaced each
user’s ID with a number, and like the AOL dataset, researchers were able to identify individual users despite the redacted usernames [24]. While de-anonymization is likely less damaging than for the AOL dataset, there is still risk to users (e.g. identification of political preferences via political movie ratings). Several researchers have developed algorithms for private analysis of the dataset [21], and the dataset is also evaluated in the Airavat system.

While the original dataset was distributed as multiple files (one per movie), for analysis in Shade we concatenated all movie ratings into a single comma-separated file with the following format:

```
movie_id, user_id, rating, date
```

### 6.4 Experiments

In the below sections, for each query we provide a description of the query, the dataset used, and the services (SparkLAP, SparkSAM, Spark Core, or MapReduce) under comparison. We also include either the SparkSAM or the SparkLAP query; we omit MapReduce and Spark Core queries due to brevity, but note that they are available at [https://github.mit.edu/dig/Shade](https://github.mit.edu/dig/Shade). All queries assume the existence of a PrivateSparkContext $ctx$.

Before continuing, we recommend reviewing (5), specifically sections (5.1, 5.3, 5.4).

#### 6.4.1 Performance

**Average Rating**

We ran a query on SparkLAP, Spark Core, and MapReduce to find the global average rating of the Netflix dataset across all movies.

The query can be described as follows: create a SparkLAP RDD using the filepath
Figure 6-1: Performance of MapReduce, SparkLAP, Spark Core on Average Rating
(n=5)
Figure 6-2: Performance of MapReduce (MR), SparkLAP (SL), Spark Core (SC) on 1st, Subsequent iterations of Iterative Average Rating (n=5, i=10)

to Netflix dataset, split each row of the dataset into comma-separated components, extract the third column (movie ratings) and convert to Double format, set the Range of values to (0, 5), and compute the average.

val rdd = ctx.getLapRDD("hdfs:///path/to/dataset")
val ratings = rdd
  .map(x => x.split(","))
  .map(x => x(2).toDouble)
  .setRange(new api.Range(0, 5))
println(ratings.avg())

Iterative Average Rating

We then repeatedly executed the Average Rating query on the Netflix dataset cached in memory for SparkLAP, Spark Core, and MapReduce. We note that MapReduce
has no provision for caching, so the MapReduce code is instead implemented as a series of identical repeated queries with no caching.

The query can be described as follows: first create a SparkLAP RDD of the Netflix dataset and cache it in memory of the slave nodes. Then run Average Rating on the cached dataset ten times.

```scala
val rdd = ctx.getLapRDD("hdfs:///path/to/dataset")
rdd.cache()
for (j <- 1 to 10) {
  val ratings = rdd
    .map(x => x.split(","))
    .map(x => x(2).toDouble)
    .setRange(new api.Range(0, 5))
  println(ratings.avg())
}
```

**Expanded Iterative Average Rating**

We created an expanded version of the Netflix dataset with the source ratings each cloned ten times, giving one billion total ratings. The resultant dataset was larger than the amount of memory available on the cluster. We ran the Iterative Average Rating query on this expanded dataset on SparkLAP, Spark Core, and MapReduce. Code and query description is the same as Iterative Average Rating, albeit with a different dataset path.

**Many Average Ratings**

We ran a query on SparkLAP, Spark Core, and MapReduce to calculate the average ratings of the first thousand movies in the Netflix dataset and output them all at once. This necessitated using `GroupBy` pairwise RDD to efficiently partition the dataset into a Pairwise RDD.
Figure 6-3: Performance of MapReduce (MR), SparkLAP (SL), Spark Core (SC) on 1st, Subsequent iterations of Expanded Iterative Average Rating (n=5, i=10)
Figure 6-4: Performance of MapReduce, SparkLAP, Spark Core on Many Average Ratings (n=5)
The query can be described as follows. First create a SparkLAP RDD of the Netflix dataset and split each line into columns. Group each line by the first column (movie ID) and extract the third column (movie rating) as the value, creating a pairwise RDD (id, rating). Set the keys of the pairwise RDD to be the ids 1 through 1000, and the Range for each rating to be (0, 5). Finally, report the average rating for each id in the provided key sequence.

```scala
val rdd = ctx.getLapRDD("hdfs://path/to/dataset")
val movie_ids = 1 to 1000

val split_rdd = rdd
  .map(x => x.split("",""))
  .groupBy(x => x(0).toInt)
  .mapValues(x => x(2).toDouble)

val grouped_rdd = split_rdd
  .setKeys(movie_ids)
  .setRangeForKeys(movie_ids, new Range(0, 5))

val averages = grouped_rdd.kAvg()
averages.foreach(x => println(x))
```

Relative Keyword Occurrence

We ran a complex query on the AOL dataset to compare the relative occurrences of two different words in the total search corpus. This necessitated first de-duplicating any duplicate search queries using `distinct` and then using a `GroupByMulti` query to split off each word into a separate key. We assumed that no search term was longer than thirty words and used "mac" and "pc" as the example terms. We ran the query on MapReduce, SparkLAP, and Spark Core.
Figure 6-5: Performance of MapReduce, Spark Core, SparkLAP on Relative Keyword Occurrence (n=5)
The query can be described as follows. First, create a SparkLAP RDD from the AOL filepath, split into columns using the tab delimiter, extract the first column (search text), and filter out all duplicate searches. Then, use GroupByMulti to create one key-value pair for each word in each search query, using the word as the key and 1 as the value. For example, the two queries ("pc and mac", "just pc") would result in the pairwise RDD ("pc", (1, 1)), ("mac", (1)), ("and", (1)), ("just", (1)). Finally, set the keys to consider to be the two words of interest and return the counts of each, which in this case would be ("pc", 2), ("mac", 1).

val first_word = "mac"
val second_word = "pc"
val max_words_per_query = 30
val rdd = ctx.getLapRDD("hdfs:///path/to/dataset")
val queries = rdd.map(x => x.split("\t")(1))
val unique_searches = queries.distinct()
val unique_words = unique_searches
    .groupByMulti(x => x.split(" ").map(y => (y, 1)), max_words_per_query)
    .setKeys(List(first_word, second_word))
val first_count = unique_words.get(first_word).count()
val second_count = unique_words.get(second_word).count()
println(
  first_word + ": " + first_count + ", ",
  + second_word + ": " + second_count
)

Efficient Relative Keyword Occurrence

We modified the query above to first filter the set of searches down to just the relevant small subset that actually contained one of the candidate words, thus greatly reducing
Figure 6-6: Performance of MapReduce, Spark Core, SparkLAP on Efficient Relative Keyword Occurrence (n=5)
the amount of data that needs to be de-duplicated, grouped, etc. The rest of the
query then proceeded as normal. We ran the query on SparkLAP, Spark Core, and
MapReduce.

```scala
val first_word = "mac"
val second_word = "pc"
val max_words_per_query = 30

val rdd = ctx.getLapRDD("hdfs://path/to/dataset")

val queries = rdd.map(x => x.split("\t")(1))

val filtered_queries = queries
  .filter(x =>
    x.contains(first_word) || x.contains(second_word)
  )

val unique_searches = filtered_queries.distinct()

val unique_words = unique_searches
  .groupByMulti(x =>
    x.split(" ").map(y => (y, 1))
  , max_words_per_query)
  .setKeys(List(first_word, second_word))

val first_count = unique_words.get(first_word).count()
val second_count = unique_words.get(second_word).count()

println(
  first_word + ": " + first_count + ", "
  + second_word + ": " + second_count
)```
Median Rating

We created a query to find the median rating of the Netflix dataset. We note that SparkLAP has no provision for median finding so this was instead run on SparkSAM, Spark Core, and MapReduce. Also note that while the median function here is unique to SparkSAM, the same could be accomplished with a composition of sortBy, zipWithIndex, and lookup.

The query is identical to Average Hash with the exception of a SAM RDD instead of a LAP RDD and a median Action instead of Average.

```scala
val rdd = ctx.getSamRDD("hdfs:///path/to/dataset")
```
val hashed_rdd = rdd.map(x => {
  val line = x.split","
  val rating = line(2).toDouble
  rating
})

val median_hash = hashed_rdd.median()
println(median_hash)

Median Hash

We mapped each line of the AOL dataset to a hash of its contents. We then ran a query to find the median of the hash codes on SparkSAM, Spark Core, and MapReduce.
The query can be described as follows: get a SAM RDD of the AOL dataset, map each line to its Java hash code, take the absolute value modulo one million, convert to a Double, and take the median of all hashes. The result should be the median of a dataset of values uniformly distributed between 1 and one million, which should be around 500000.

```scala
val hash_width = 1000000

val rdd = ctx.getSamRDD("hdfs:///path/to/dataset")

val hashed_rdd = rdd.map(x => {
  val result = math.abs(x.hashCode) % hash_width
  result.toDouble
})

val median_hash = hashed_rdd.median()
println(median_hash)
```

**Sparse Median Hash**

We then ran the same query on the Netflix dataset, but using a hash width of $10^8$, and ran on SparkSAM, Spark Core, MapReduce. Note that as the universe of hash keys is roughly equal to the cardinality of the dataset, we refer to this as a sparse hash. Code is identical with the exception of the different hash width and a different dataset.

**Average Hash**

Finally, we queried the average of the AOL hashes from Median Hash on both Spark-LAP and SparkSAM to compare their performance on the same query. Code is identical with the exception of the RDD type and the use of `average` instead of
Figure 6-9: Performance of MapReduce, Spark Core, SparkSAM on Sparse Median Hash (n=5)
Figure 6-10: Performance of SparkLAP, SparkSAM on Many Average Ratings 
\((n=10)\)
Figure 6-11: Accuracy of SparkLAP on Movie 2 Count (n=10000)

median.

6.4.2 Privacy and Accuracy

SparkLAP

We run a query to find the number of ratings for a given movie in the Netflix dataset and the average rating for that movie. We record the results on SparkLAP to check for the expected Laplacian distribution.

The query is identical to Average Rating with the addition of a filter to only consider ratings for movie movie_id. We also add in a count Action in addition to the avg Action.

val movie_id = "2"

val rdd = ctx.getLapRDD("hdfs:///path/to/dataset")
val transformed_rdd = rdd
  .map(x => x.split("",""))
  .filter(x => x(0).equals(movie_id))
  .map(x => x(2).toDouble)
  .setRange(new Range(0, 5))

val num_ratings = transformed_rdd.count()
println("Count: "+ num_ratings)

val avg_ratings = transformed_rdd.avg()
println("Average: "+ avg_ratings)
SparkSAM

We record the results of Median Hash and examine the balance between perturbation and utility. Code is identical to Median Hash.

Comparison

Finally, we record the results of Average Hash and compare the utility of SparkLAP and SparkSAM. Code is identical to Average Hash.
Figure 6-14: Accuracy of SparkLAP, SparkSAM on Average Hash (n=10)
Chapter 7

Conclusion

7.1 Discussion

Below we examine each of the workloads discussed in (6.4) and evaluate how effective Shade was in meeting its stated goals.

7.1.1 SparkLAP Performance

We observe that SparkLAP is noticeably faster than even a non-private MapReduce cluster across a large variety of workloads. Given that [30] claims that Airavat has an overhead of 40% from a MapReduce cluster, and that PINQ cannot even be used today at all [28], we are confident claiming that SparkLAP has conclusively met its goals of a substantial performance improvement from earlier differentially private systems. Individual query results are discussed below.

Average Rating

Examining Figure 6-1, which is a simple single-output single-pass query, it can already be seen that SparkLAP improves on MapReduce by around 350%, with a mean completion time of 20 seconds vs the 70 seconds of MapReduce. We also note that while Average Rating does exhibit a 10% discrepancy between the two, we believe that this is merely noise from a small sample set and observe that later experiments
show a more equal performance between Spark Core and SparkLAP.

**Iterative Average Rating**

When Average Rating is run iteratively (simulating the performance characteristics of a single-output multi-pass query such as a gradient descent), SparkLAP’s advantage becomes even more evident. This is especially true if the dataset fits in the memory of the cluster. Since SparkLAP is able to leverage the in-memory caching provided by Spark, the performance cost of later iterations is significantly cheaper than the first iteration, as later iterations are able to instead query the cached dataset stored in memory and not have to fetch from disk. Figure 6-2 illustrates this - while the first iteration has an average runtime of around 20 seconds, later iterations have a 500% improvement to around 4 seconds. Furthermore, even the slowest SparkLAP iteration is still also around 400% faster than the average MapReduce iteration. MapReduce, being unable to leverage the benefit of caching, demonstrates no noticeable decrease in runtime after the first iteration at all. As a result, SparkLAP has an implied 2000% performance increase over MapReduce on queries with many iterations, conditioning on the dataset fitting in the cluster’s total memory.

**Expanded Iterative Average Rating**

However, if the dataset is unable to fit in memory, SparkLAP’s advantage lessens significantly. In Figure 6-3, note that SparkLAP has only around a 50% decrease in runtimes following the first iteration, consistent with the fact that the dataset can only be partially cached. Interestingly, the runtimes of the first iteration also converge when using the larger dataset, with MapReduce this time only being 50% slower than the slowest iteration of SparkLAP. Again though, MapReduce shows no decrease in average runtimes following the first iteration, and although SparkLAP’s decrease is significantly reduced, the advantage of SparkLAP is still significant: around 400%. We note that larger datasets would likely see further equalizing as less and less of Spark’s caching ability is able to be leveraged, however it seems unlikely given the vast difference that MapReduce would ever eclipse SparkLAP.
Many Average Ratings

We also evaluated a query with multiple outputs in a single pass, to see whether the large parallel workload changes the results. While Figure 6-4 shows that the advantage of SparkLAP is slightly lessened to around 200% (40 seconds vs 80 seconds), SparkLAP still provides a large gain over MapReduce.

Keyword Occurrence

We then shifted from aggregations on the large narrow Netflix dataset to targeted comparisons on the wider AOL dataset, to see how SparkLAP responds to more complex queries and pairwise RDDs. When running the relative keyword occurrence query without pre-filtering, SparkLAP still performs significantly better than MapReduce, with over a 500% improvement (see Figure 6-5). We hypothesized that this query in particular is a bit unfair to MapReduce - because of the lack of pre-filtering, MapReduce is forced to write each individual word count to disk prior to the Reduce stage, while Spark Core and SparkLAP have no such requirement due to the aforementioned memory caching.

Efficient Keyword Occurrence

However, when pre-filtering to reduce the amount of data processed, SparkLAP still demonstrated around a 500% improvement (10 seconds vs 50 seconds, see Figure 6-6). We hypothesize that this is because MapReduce has an inherently large startup overhead even for small jobs, meaning that even though the work done by the job was significantly less intensive than before, SparkLAP was still able to greatly outperform MapReduce. We note that no job we observed in MapReduce was ever completed in less than 40 seconds, while some SparkLAP jobs could be completed in as little as 6 seconds.
7.1.2 SparkSAM Performance

Although SparkSAM was not designed with performance as an explicit goal, in testing we found that its performance was surprising enough to warrant discussion as well. As expected, unlike SparkLAP, SparkSAM does add significant overhead to equivalent non-private jobs in Spark Core on some workloads; however, on others, SparkSAM actually meets or even exceeds Spark Core’s performance. For completeness, we also include reference times to MapReduce, and we include a comparison to SparkLAP as well.

Median Rating

Like SparkLAP, we began with a simple query to find the median rating in the Netflix dataset. This result ended up being the most fascinating, as SparkSAM exceeded Spark Core’s performance by almost 200%, and MapReduce exceeded SparkSAM by another 200%.

While this result was initially shocking, we realized that this case is an outlier caused by the extremely low cardinality of ratings: only 5 unique ratings (1-5) were observed from 100 million rows. Because of this, Spark Core is greatly disadvantaged; the Spark scheduler is only able to parallelize work at the individual key level, so since there are only 5 unique keys it means that only 5 nodes are able to do any of the sorting work - yet there are still 100 million records! SparkSAM avoids this by instead coalescing all data for a given partition on each node and sorting only within the partition, thus preventing any bottleneck of few nodes having to sort many values. Finally, MapReduce has no shuffle at all, instead doing aggregations in the map phase through use of a combiner; this means little data needs to be materialized (only the key-count pairs) and thus the reduction phase takes almost no time at all.
**Median Hash**

To more fairly evaluate the performance of SparkSAM vs Spark Core, we created a contrived query where we took the hash code of each line of the AOL dataset modulo one million, thus creating a semi-uniform distribution of values across the range (1, 1000000). As expected, this removed the unnatural bottleneck in Spark Core and advantage in MapReduce - we instead observe that Spark Core is now 200% faster than SparkSAM (which is approximately at parity with MapReduce). Still, with only twice the runtime of Spark Core in even the most pessimal of workloads for SparkSAM, this provides confidence that SparkSAM delivers at least respectable performance in a variety of cases.

**Sparse Median Hash**

For completeness and to confirm the trend, we then hashed each of the lines in the Netflix dataset to a uniform distribution between 1 and 100 million (so roughly 1:1 with the number of lines) in order to see the performance on sorting a very sparse dataset (see Figure 6-9). We see that MapReduce is further hurt by the trend, dropping to around 1.5x the runtime of SparkSAM, and SparkSAM improves to around 1.5x the runtime of Spark Core (likely causing decreased accuracy of results due to a decrease in uniformity of partitions).

**Average Hash**

Finally, we compared the performance of SparkLAP and SparkSAM on a query that both systems could accommodate: \( \text{avg} \). We performed Average Hash on both SparkLAP and SparkSAM; as expected, SparkLAP significantly outperformed SparkSAM by around a factor of 5 (see Figure 6-10). Perhaps a more complex query would work more in SparkSAM’s favor, however given the results of the above experiments and combined with the fact that SparkLAP is nearly at parity with Spark Core, we believe that SparkSAM is unlikely to ever outperform SparkLAP without the locality improvements discussed in (7.2).
7.1.3 Privacy And Accuracy

SparkLAP

Examining both the Count and Average accuracy graphs, we can clearly see that the Laplacian mechanism is successfully implemented (Figures 6-11, 6-12). The Count results nearly exactly fit the distribution of a Laplacian with \( \lambda = 10, \mu = 144 \), which is consistent with the true value of 144 and the Laplacian Mechanism (2.3.1) expecting sensitivity = 1 and \( \epsilon = 0.1 \). With the Average equation, we expect the sensitivity to be less consistent, since it is instead \( \frac{\text{range}}{\text{noisy}_\text{count}} \approx \frac{5}{144} \), but it also somewhat matches the expected distribution when accounting for the relatively noisy sensitivity.

We also observe that utility is still somewhat preserved, even considering that the dataset cardinality is relatively small (only 144 ratings exist for movie 2). A query run to measure the average of all movies would of course have far lower noise, but we chose to illustrate the privacy mechanism in a case where its effect was more visible due to relative sensitivity of the query.

SparkLAP

For SparkSAM, recall that the mechanism is more complex to model than the simple Laplacian discussed above. As discussed in (2.3.2), noise comes from three different sources: 1) variance introduced by the random partitioning, 2) randomness from the exponential mechanism used to privately select the first and third quantiles of the subsampled results, and 3) Laplacian noise added to the Winsorized mean scaled by the width between the private first and third quantiles. The exact mathematics of how this system preserves differential privacy are out of scope for this thesis and are deferred to [32], however, examining the graph we can see a roughly Gaussian curve that certainly perturbs the result in a manner that would make it near impossible to recover the influence of a single record. However, note that the dataset is uniformly, not normally, distributed - meaning that variance itself might also be a factor in the noise applied here. Still, this result in combination with the following result strongly
implies that privacy is preserved. We discuss utility in the following section.

**Utility Comparison**

Finally, consider Figure 6-14. In this, we compare the noise provided to an equivalent query performed by both SparkLAP and SparkSAM. We immediately note that SparkLAP has far lower noise applied than SparkSAM, to the point where the actual average is barely perturbed at all (note that this is in line with an estimated sensitivity of around $\frac{1000000}{34000000}$). On the other hand, SparkSAM has far higher noise, as evidenced by the much larger gap between the top and bottom quartiles.

While this is disappointing for SparkSAM’s utility for queries with low global sensitivity, it actually lends credence to the claim that privacy is preserved due to the higher amount of noise. In addition, note that SparkSAM’s noise application is fairly agnostic to the type of query made, so while SparkSAM might lose on avg, the median query would have $\lambda = 10000000$ in SparkLAP (a value which destroys all utility completely) but be around the same variance as the Average Hash result for SparkSAM (that is to say, still useful).

Overall, while SparkSAM did not end up increasing utility for queries computable by other differentially private systems (and by SparkLAP), it did unlock a whole new class of high-global-sensitivity queries not computable on other systems, and appears to have done so without compromises on either privacy or utility.

### 7.2 Future Work

Although Shade was successful in its intended goals, there are still several interesting avenues that would be beneficial for future researchers to investigate.

While SparkLAP is fairly close in performance to Spark Core and likely doesn’t have much room for improvement, various aspects of SparkSAM are still ripe for perfor-
mance gains. In particular, we observed during experiments that over half of the operation time of a SparkSAM query is spent coalescing each partition to reside on a single machine; if the partitioning function used to create random samples could be reworked to use locality-aware randomization (only creating partitions from data residing on a single machine), then performance could be improved by a significant margin.

An earlier goal of Shade was to also provide support for using the exponential mechanism to privately extract non-numeric values from datasets in addition to noised numeric results. While this did not end up coming to fruition in this thesis, an implementation of differential privacy with the ability to return non-numeric values is still absent from the literature and would remove a significant barrier to organizations adopting privacy principles.

While differentially private systems have historically not supported the `join` operator due to its unbounded stability [22], more recent work from e.g. [29] provides various techniques for alleviating this privacy cost. Future work could investigate the possible of adding a private version of `join` to Shade, through either an existing mechanism or novel work. Anecdotally, we found it slightly harder to write queries in SparkLAP than in Spark Core due to the absence of `join`; we believe as `join` is a significant component of many non-private analysis systems, that its absence is another large barrier to mainstream adoption of differential privacy.

Finally, while multipass queries in Shade do greatly outperform MapReduce, the privacy cost is still significant, costing budget linear in the number of iterations. Future work could explore the possibility of reducing this, perhaps through clever use of the composition properties of differential privacy [8] to alleviate the budgetary and privacy cost.
7.3 Conclusion

As organizations collect increasingly more data about individuals, it is clear that significantly more care must be given to preventing misuse of that data. While access control mechanisms and security protocols can prevent unauthorized access, the potential for even an authorized user to abuse sensitive data is still severe, particularly when that data is mined by a large number of analysts using computation platforms such as Apache Spark. Differential privacy provides a number of mechanisms for preventing this abuse by limiting the influence that any individual can have over any information provided by the database, but with the notable exception of Apple Inc [11], differential privacy has yet to be widely adopted in industry [19].

In an attempt to fix this, Shade provides an implementation of differential privacy atop the widely used Spark platform. Through SparkLAP, Shade allows analysts to execute statistical queries at performance that substantially exceeds previous work, particularly on small datasets or queries with many iterations. For queries with high worst-case global sensitivity such as median, SparkSAM instead applies noise calibrated to the local contents of the database, allowing a whole new class of queries that would be otherwise impossible to answer using previous systems. While Shade is currently not suitable for immediate deployment in industry, we believe that the results demonstrated inspire future work by both researchers and engineers to advance privacy principles without compromising on the performance and compatibility needs of large organizations.
Bibliography


[34] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster