BIG DATA SANITIZATION USING SCALABLE IN-MEMORY FRAMEWORKS

by

Kanchan Prakash Waikar

APPROVED BY SUPERVISORY COMMITTEE:

_____

Dr. Murat Kantarcioglu, Chair

_____

Dr. Latifur Khan

_____

Dr. Bhavani Thuraisingham

Dedicated to, my husband, my parents, and my family.

BIG DATA SANITIZATION USING SCALABLE IN-MEMORY FRAMEWORKS

by

Kanchan Prakash Waikar, B. Tech

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

May 2017

ACKNOWLEDGMENTS

First and foremost, I would like to thank Dr. Murat Kantarcioglu, for being an inspirational mentor, motivator, and for his constant support throughout my master's in computer science. I feel privileged to have had the opportunity to learn so much from him. Apart from the curricular course, I was fortunate enough to learn from him advanced concepts in the semantic web, machine learning, and big data frameworks. I cannot compare learnings I received while doing master's thesis with any coursework, and I will always remain grateful towards him for the same. I am also thankful to Dr. Bhavani Thuraisingham and Dr. Latifur Khan for their feedback, encouragement and for supervising my master's thesis. Discussions with Dr. Lalana Kagal and Alec Heifetz from MIT were extremely helpful, and I would like to thank them for their ideas. I would also like to thank Dr. Balaji Raghavachari, Dr. Mithun Balakrishna, Dr. Anjum Chidha, and Dr. Keven Ates for their guidance throughout my master's at UT Dallas. I feel I have learned a lot in the last two years and I would like to thank all my professors for being excellent teachers.

Living in a foreign country becomes easier when you have great friends and a supportive family. I would like to thank my husband, and my family, for constant support and encouragement. Thanks, Kavya, Sruti, Rachna, Tushar, Saagarikha, Javnika, Ravali, Kruthika, Madhuri, Ameya, Maryam, Himanshu, Parth, Dharmam, for always being there for me. Moreover, last but not least, I would like to thank the staff at UT Dallas for helping me in all possible ways.

March 2017

BIG DATA SANITIZATION USING SCALABLE IN-MEMORY FRAMEWORKS

Kanchan Prakash Waikar, MSCS
The University of Texas at Dallas, 2017

Supervising Professor: Dr. Murat Kantarcioglu

As more and more data is collected, it is growing beyond the scale humans could ever have imagined. Not only data but also data collection and analysis techniques have evolved and have enabled researchers to advance many fields such as medical science. Although health data can have a huge impact on the future success of research, data is usually distributed among various stakeholders. Organizations need to share this data to help research move forward, but health data sharing is a regulated domain. Due to privacy concerns, the U.S. Department of Health and Human Services (HHS) has taken steps to ensure privacy protection of individuals by regulating data sharing through Health Insurance Portability and Accountability Act of 1996 (HIPAA). HIPAA policy restricts publishers from sharing identification information as well as any auxiliary information that can be used for record re-identification.

To make data sharing compliant with the HIPAA policy, various data privacy protection techniques evolved. Differential privacy techniques focused on query accuracy maximization in statistical databases while minimizing the "risk" of record identification, whereas, data anonymization allows the publisher to share original data at lesser precision, i.e., sharing attribute value of age as 25-35 instead of 30. These techniques are considered as an industry standard.

Newer risk-based models determine record anonymization level based on the hidden "risk" of re-identification of the record. With constantly increasingly sanitization requests around big data, sanitization algorithms need to be adapted for distributed computing frameworks. Frameworks like Hadoop-MapReduce achieve parallelism by distributing tasks on multiple machines and executing them in parallel. Apache Spark is a Hadoop-MapReduce based in-memory distributed framework with support for data caching making it more suitable choice for iterative anonymization algorithms. This study focuses on developing distributed in-memory data sanitization techniques.

To extend traditional k-anonymity methods, we implemented Mondrian k-anonymization algorithm for Apache Spark. The Mondrian algorithm performs multidimensional partitioning cuts until data cannot be divided further without violating k-anonymity property. We propose a locality sensitive hashing (LSH) based one pass anonymization algorithm in which we use LSH functions for the formation of clusters of size k and finding a summary statistic for each cluster.

To support newer data anonymization methods, we implement an in-memory version of risk estimation based anonymization algorithm that leverages game theoretical approach for deciding optimal generalization level for each record. We then propose a hybrid risk anonymization algorithm that uses LSH bucketing to minimize the number of risk estimation algorithm executions.

To support online sanitization, we propose an aspect-oriented approach for modifying Apache Spark RDD's computation at runtime. We show how an aspect can suppress identifier field based on predefined policy at runtime.

With evolving functional requirements like within-dataset anonymization vs within population anonymization, centralized vs distributed anonymization, risk-based vs strict k-anonymization, it

is crucial to select the method that fits the requirement correctly. This study offers different solutions that are suitable for different functional requirements. The analysis and comparison of above methods would enable data publishers to make efficient computation cost anonymization decisions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Increasingly, medical research is becoming highly data driven and it uses big data sets ranging from hospital discharge data to genomic data. Genomic datasets are so huge that they are claimed to be beyond what big data technologies can handle today (Stephens, et al., 2015), thus leading to high-performance big data sanitization requirements. The research data is usually scattered and collection typically involves gathering data from different health organizations. However, due to privacy concerns, the process of health data sharing is highly regulated. In the USA, Health Insurance Portability and Accountability Act of 1996 (HIPAA) was introduced to regulate data distribution and for the protection of individual's privacy (HHS, 2000). In essence, HIPAA states that individual's privacy must be protected while sharing data (HHS, 2000). In order to enable privacy protection in shared data, data sanitization is used. Sanitization is a process of removing the sensitive information so that the data can be distributed to a larger audience. Sanitization is achieved through data anonymization, i.e., anonymizing identification information. Data sanitization is typically done out of two intents – secrecy protection and privacy protection. Secrecy protection is about converting secret/top secret data into less harmful versions that can be shared at a lower protection level. Privacy protection is about protecting the privacy of the individuals about whom data is being shared. The privacy of individuals is considered to be a human right (Wogara, 2001) and thus needs to be enforced. Countries have defined laws for the protection of individual's privacy, although these laws differ from country to country, the essence remains the same – to protect individual's privacy. In the United States, the law permits lawsuits

to be filed against the individual/institution that intruded aggrieved party's private affairs (Gostin, Lazzarini, & Neslund, 1996).

With HIPAA, the Department of Health and Human Services addresses the issue of protection of privacy in data distribution and use of individual's health information. The failure to implement and comply with these policies results in penalties, and in some cases, even imprisonment (HHS, 2000). In their earliest versions of implementation specification, HIPAA shared a list of attributes that must be removed or generalized to make sure that the data meets HIPAA safe harbor policy. HIPAA primarily covered individually identifiable information and stated that data is safe as long as it is de-identified. The list included SSN, telephone numbers, account ids, license numbers, zip code and all possible identifier fields. At glance, the HIPAA list looks pretty exhaustive, but it is not. Several successful re-identification attempts made and it has been shown that the above approach is just not sufficient to protect privacy (El Emam, Jonker, Arbuckle, & Malin, 2011). This lead to HIPAA acknowledging that de-identification is more of a risk analysis problem rather than a simple process of removing identifiers (HHS, 2000). In order to rectify this issue, HIPAA came up with a second implementation specification that states that an individual with appropriate qualification and knowledge of accepted scientific and statistical methods for identification, can claim that health information is not individually identifiable if and only if, the individual is able to confidently state that data has extremely low risk of re-identification based on generally available knowledge, and documents the methods used for coming to this conclusion (HHS, 2000). This specification accepts data with relatively low risk of re-identification as de-identified data.

In order to comply with HIPAA policies, data publishers have to share data with relatively low risk of re-identification. To address this challenge, multiple approaches were proposed and

several algorithms evolved (Dwork, 2008; Ramakrishnan, LeFevre, & DeWitt, 2006). An evolution of serial sanitization algorithms happened during the phase when the world was still getting ready for the big data. Since then, the data has grown exponentially and current data processing proposals focus on building algorithms that can run in distributed mode on distributed frameworks like Apache Spark. Most of the published data sanitization algorithms work well in a serial setting; however, they do not scale in a distributed setting. An experiment conducted as part of this study, outlined in Chapter 7 confirms this conclusion. Due to the very nature of these algorithms itself, they do not scale well in a distributed environment. Future data sanitization requirements are going to be big data based, which is why we need to analyze existing data sanitization algorithms and test them in a distributed environment.

The key contribution of this study is the development of efficient big data sanitization algorithms for distributed in-memory frameworks. As a result, we developed four unique algorithms for four different use cases. First is a scalable, distributed sanitization solution using a game theoretical approach (Wan, et al., 2015), second is a risk-based locality sensitive hashing bucketing sanitization solution. The third is an LSH bucketing based strict k-anonymity solution and fourth is a novel, aspect-based on-the-fly data sanitization technique. To our knowledge, these techniques have never been proposed for data sanitization using distributed in-memory frameworks, before.

1.1 Outline of the thesis

We discuss big data frameworks available as part of Chapter 2 and identify Apache Spark as the distributed framework that one could use for implementing iterative distributed algorithms. We then elaborate how Apache Spark performs distributed in-memory job processing. Chapter 2 also

outlines attribute types, i.e. identifiers, quasi-identifiers, sensitive fields, and domain specific attributes; quasi-identifiers being the focus of sanitization algorithms proposed in this study. We then continue to define the related work that has been done in the field of big data, data sanitization using LSH, and online data sanitization approaches using aspect-based methods.

Chapter 3 introduces currently known clustering mechanisms and identifies the clustering technique that can be used for performing clustering in high dimensional spaces. While evaluating strategies for big data problems, it is important to select the technique, which provides maximum information preservation while having low time complexity. This chapter suggests the use of locality sensitive hashing (LSH) bucketing technique for performing data anonymization. Locality sensitive hashing provides sub-quadratic time complexity for performing bucketing. This chapter then defines the internal mechanism of locality sensitive hashing.

Before introducing a new model for performing anonymization, we must understand popular methods used for performing k-anonymity. Chapter 4 introduces Mondrian multidimensional k-anonymity algorithm that achieves k-anonymity by performing repetitive multidimensional cuts until data becomes indivisible. Chapter 4 explains the algorithm and modifies it further to make use of distributed computing capabilities. In Chapter 4, we also introduce a one pass k-anonymity algorithm, using locality sensitive hashing bucketing technique.

Current k-anonymizations approaches involve computations on a subset of data while the risk-based game theoretical approach works on single data record at a time. In Chapter 5, we elaborate a game theoretical approach used for data sanitization using risk estimation techniques (Wan, et al., 2015). In this chapter, we also elaborate information loss evaluation technique that we would use in Chapter 7, in order to compare and contrast different anonymization approaches

suggested in this study. We then show how to modify lattice based search algorithm, to make use of distributed computing capabilities. We further extend it by using LSH bucketing in order to evaluate whether the performance of LBS algorithm can be further improved with the help of LSH bucketing algorithm.

There can be scenarios, which specifically demand online approaches. In order to provide solutions for such problems, we introduce an on-the-fly sanitization solution in Chapter 6. On-the-fly sanitization typically requires modification of the execution environment itself. However, it is not practical to modify the framework, as it requires re-work every time we want to upgrade framework to a newer version. Also, RDD computation being a cross cutting concern, needs a cross-cutting solution. We propose an aspect-based approach in Chapter 6, which focusses on modification of Apache Spark execution environment at runtime. We introduce an aspect called BlueRay for suppressing identifiers in a structured file. We elaborate how to extend BlueRay aspect to perform several things like attribute generalization and attribute suppression.

As part of Chapter 7, we document the experimental evaluations performed for both – batch as well as online approaches. We start by comparing how LSH based technique performs compared to Mondrian k-anonymity, both being strict k-anonymity methods. We contrast the information preserved in above-mentioned approaches with the LBS algorithm and study how LBS algorithm scales with increasing file size. We then shift the focus of the study to online approaches and study performance impact of BlueRay aspect itself on different operations like reading a dataset, writing a sanitized dataset and performing a group by query. We also compare and contrast the performance of two policy management techniques suggested in the study. As part of Chapter 8, we document our conclusions and highlight possible future work to take the study ahead.

CHAPTER 2

BACKGROUND AND RELATED WORK

To be able to discuss and propose newer methods, we need to first elaborate existing data sanitization algorithms along with related research done in the big data sanitization field. In this chapter, we first discuss current data sanitization approaches, and then analyze available platforms and choose the platform that is right for performing big data sanitization. We also define characteristics of the data before describing related work that has been done in the field of big data sanitization.

2.1 Current big data sanitization approaches

Based on HIPAA §164.514 specification, various anonymization techniques have been suggested. T-closeness, l-diversity, Mondrian are some of the examples of such techniques (Ramakrishnan, LeFevre, & DeWitt, 2006; Machanavajjhala, 2007; Ruggieri, 2014). K-anonymity is a strategy that relies upon increasing anonymity in the data in order to make the probability of re-identification in de-identified data very small; higher value of k implies lower risk.

These algorithms anonymize data just enough to get the risk to a smaller value. However, some address only part of the problem like numeric quasi-identifiers (Chakravorty, 2016) whereas some perform anonymization of only personal or identifying information (PII) data (Judson, 2015), etc. Many of the approaches have very high time complexity, which makes them infeasible for data size beyond few megabytes. We need algorithms, which are more feasible for big data. In this study, we focus on distributed high-performance anonymization algorithms. Considering data growth rate, as well as increasing performance expectations from software, soon serial

anonymization approaches just are not going to be enough. Typically, anonymization algorithms are iterative in nature and given that big datasets are usually in gigabytes, a sanitization solution needs a computing framework with built-in performance. For this reason, it is important that problem of data privacy be researched for distributed big data frameworks.

2.2 Big data frameworks

With the innovation of disruptive technology known as the internet, data has grown beyond the scale humans could imagine, two decades ago. Big data has become common term everywhere. The field of distributed computing has risen in order to provide for high compute/high memory big data algorithms. Hadoop-MapReduce cluster computing field has gained and flourished enormously. There are several distributed big data processing tools and techniques available in the market that serve specific needs. MapReduce works mainly by providing two methods; map method runs in parallel on multiple smaller splits of the original data, and reduce method aggregates result produced by the map function. In last few years, MapReduce framework has become extremely popular because of the parallel computing capability. One of the major reasons behind the popularity of the framework is data code locality. The framework optimizes the execution of the task in such a way that computation takes place on the node on which data is stored. The reduce function is executed on the reducer and it typically aggregates map function's output. Two or more map-reduce sequences can be chained, but in order to do so, we need to persist first map-reduce function's output and pass it as input to the second map function. For this reason, MapReduce does not typically work well for iterative algorithms.

Compute heavy algorithms typically involve complex calculations and little I /O. Iterative algorithms typically belong to this category; they iterate over and reuse intermediate data. Writing

a complex transformation of data typically involves chaining multiple map-reduce functions and thus leading to a lot of I/O. For this reason, we need a distributed computing framework appropriate for distributed iterative processing. Based on analysis of big data frameworks available and our requirement of iterative distributed processing, we decided to go ahead with Apache Spark.

2.2.1 Apache Spark: Introduction

Apache Spark is an open-source, in-memory, and distributed computing framework. It enhances Hadoop's data locality principle by adding a provision for data caching. It provides programmers an application program interface (API) around a central, fault-tolerant data structure known as Resilient Distributed Datasets (RDD). RDDs are lazily evaluated parallel immutable data structures (Zaharia, et al., 2012). RDDs are created by combining multiple RDDs or by reading files. RDDs let us chain multiple map-reduce/map-map commands without requiring read/write of intermediary outputs from or to an I/O device. Apache Spark achieves fault tolerance in RDDs by maintaining lineage graph for each RDD. Lineage graph explains RDD creation process.

2.2.2 Apache Spark Execution framework

When a program is submitted to an Apache Spark cluster, the node on which it is submitted is referred to as Driver. When the script is launched, driver coordinates with cluster manager and assigns executors to the program. A DAGScheduler running on driver reads user's program and creates a DAG, a directed acyclic graph of operations involved (map or reduce). Typically, it forms a graph of jobs where one job consists of multiple stages, and a stage depends on another stage or file(s). A stage typically consists of several tasks. DAG is then sent to TaskScheduler for task assignment. TaskScheduler coordinates with the cluster manager and launches tasks on the cluster.

A task includes task's closure, i.e. all input variables and methods used by the task. The driver creates a copy of input variables and methods and sends the same to the executor. Executors are launched on worker nodes present in the cluster. Each executor executes the task and sends the result to the driver. The executor can also cache intermediary results and improve the overall performance of the program. Executors execute more than one task during their lifetime on the worker node.

In Figure 2-1, inspired from the Apache Spark documentation (Zaharia, Spark Internals documentation, 2012), we can see a simple execution flow in Apache Spark framework. Here we are joining two RDDs and then executing the action called count. Execution of an action leads to the evaluation of the DAG and hence execution of the code. We can see in Figure 2-1 that both the RDDs contain 3 input splits each. Each input split is read and emitted by a task. This forms the stage 1 of the DAG.

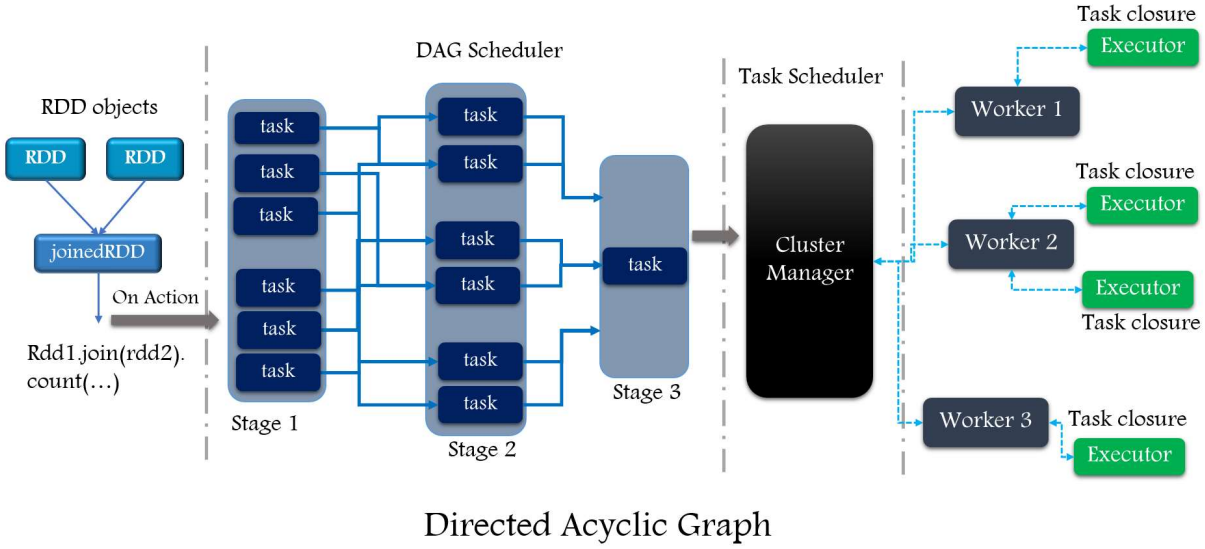


Figure 2-1. Apache Spark Execution framework

Stage 2 is a join stage; here we are joining one input split each of both the RDDs. Assuming that there is one to one join, we can see a set of tasks getting created in stage 2. These are map tasks. This forms as an input to the stage 3, in which count task gets executed. These all tasks are sent to cluster manager stage by stage. Once a stage has been executed, and inputs for next stage are ready, next stage gets executed.

Cluster manager is aware of all spark slaves, as well as the data required by tasks. Cluster manager communicates with HDFS and gets the details of data nodes on which data resides and sends tasks to slaves located on corresponding data nodes.

As we can see, Apache Spark framework is suitable for iterative data processing, respects data locality, and is also widely adopted. The framework, originally launched in 2010, has become mature and has stood the test of time. These attributes make Apache Spark a good candidate for implementation of distributed algorithms.

2.3 Dataset and metadata for privacy analysis

Defining data characteristics is the first step we take before we define, elaborate and evaluate distributed sanitization algorithms. Data typically consists of four parts, identifiers, quasi-identifiers, sensitive attributes, and domain specific attributes.

Identifiers are attribute(s) that uniquely identify the individual the record corresponds to. These must be suppressed before sharing, as they may be universally unique identifiers and typically do not follow a specific pattern. Typical examples of this column are social security number, license number, full name, etc. Most of them were included in the first list released by HHS as part of HIPAA policy (HHS, 2000).

Sensitive attributes are the attributes that are extremely sensitive and must not be revealed or associated with the identity of the individual. These attributes are the ones which data consumer is interested in, and hence cannot be generalized/suppressed by data publisher.

Typical examples of sensitive columns are "bank balance of an individual", "individual's medical condition", "number of parking tickets received", etc.

Quasi-identifiers are a group of attributes that together can uniquely identify the identity of the individual data belongs to. Knowledge of these attributes can lead to very high possibility of successful re-identification of the individual corresponding to the record. These attributes need to be either suppressed or generalized in order to reduce the risk of attack. E.g., Consider the attribute list of zip code, age, gender, race. In sparsely populated regions, knowledge of all four of these can easily lead to re-identification of individual data belongs to.

Domain attributes are remaining attributes which data publisher is interested in, but are not particularly harmful to individual's privacy. Although knowledge of these can help get insight, the information is considered harmless, and it does not always lead to the breach of privacy. E.g., does individual prefer spring over fall?

The data sanitization approach studied in this thesis is focused on providing anonymization techniques for solving problem outlined in §164.514 of HIPAA (HHS, 2000). While studying anonymization techniques, we assume that data has already been pre-processed and does not contain any identifiers. For removal of identifier fields, we propose an aspect-based online model in Chapter 6. In sanitization algorithms suggested, we focus only on quasi-identifiers.

For our study, we consider a simple dataset that consists of 4 quasi-identifiers - age, zip code, race, and gender. We used the data from adult dataset found in UCI machine learning

repository (Lichman, 2013). We extracted quasi-columns from the dataset and used the census dataset from U.S. Census Bureau's website, 2010 Census tables PCT12A through PCT12G (Census Summary File prepared by the US Census Bureau., Oct). We also used the transformed data set used in the game theoretical framework for analyzing risk paper (Wan, et al., 2015), in order to validate our algorithm implementations. We use the census data to calculate the risk probability for a given generalization level in the entire population. Sample data format, generalization hierarchy, as well as metadata used for expediting processing of algorithm, is described in Appendix A. Apart from data definition, we also define metadata generalization hierarchy. Generalization hierarchy contains all possible variants at the bottommost level and completely suppressed row at the topmost level.

2.4 Related Work

In this section, we summarize studies relevant to our work. In Section 2.4.1, we elaborate previously known aspect-based solutions and document the learnings from each of the paper. In section 2.4.2, we discuss known use cases in which LSH was used for purpose of anonymization of data. In section 2.4.3, we discuss currently known scalable anonymization models and highlight their shortcomings. In order to setup the environment for distributed in-memory sanitization algorithms, we explain LSH, Mondrian k-anonymity, and risk-based approach algorithms in initial sections of Chapter 3, Chapter 4, and Chapter 5. Once done, we introduce the in-memory distributed versions along with LSH based anonymization algorithms.

2.4.1 Aspect-oriented approaches

The idea of AspectJ for runtime modification of bytecode is extremely popular and has multiple utilities. In past, aspect-based approaches have been considered for runtime modification of the behavior of the system. In Vigiles (Ulusoy, Kantarcioglu, Pattuk, & Hamlen, 2014), a fine-grained access control system for MapReduce, the aspect-based approach was used in order to enforce fine-grained access control (FGAC) in Hadoop MapReduce environment. The system incorporates a policy manager and demonstrates how aspect-oriented approach can be used in Hadoop-MapReduce ecosystem. The paper also states that using AspectJ is a wise way of extending a functionality, as it removes the need to maintain a copy of the software. However, the paper elaborates technique only for MapReduce framework and does not explain how it can be achieved in Apache Spark.

2.4.2 LSH for Privacy Preservation

Locality sensitive hashing has been a known de-facto method for Approximate nearest neighbor search (Indyk & Motwani, 1998). In a paper on the anonymous publication of sensitive transactional data (Ghinita, Kalnis, & Tao, 2010), LSH was suggested for anonymization of high dimensional data. The paper suggests the use of LSH, but for sparse data. The paper (Zhang, et al., 2016) suggests using local recoding anonymization using LSH for Hadoop Map-Reduce systems. The approach, however, does expect data to be converted into binary data leading to imprecise numeric data distance calculation. The technique does not involve any normalization and uses agglomerative approach for forming a cluster of size k, whereas LSH based algorithm proposed in this study focusses on increasing number of buckets in order to cluster more precisely.

2.4.3 Anonymization in big data

Apart from the paper (Zhang, Yang, & Liu, Hadoop based Anonymization , 2013), several approaches have been suggested for performing anonymization in Hadoop-MapReduce ecosystem. However, these algorithms are designed keeping the serial MapReduce type of execution model in mind. They do not leverage intermediary data caching nor do they provide iterative in-memory solutions. Hortonworks does offer a solution for data masking (Syed & Srikanth, 2016), but it does not offer a complete anonymization solution.

CHAPTER 3

CLUSTERING: APPROXIMATE NEAREST NEIGHBOURS

Clustering is a technique for finding the group of elements that are close to each other in some aspect. Consider sample data in Figure 3-1 for gender, age, race, zip code combination. We can clearly identify two clusters, each one having two points very close to each other. We can see that first two entries in Figure 3-1 form the first cluster and remaining two the second.

Male, 50, Asian, 38212
Female, 63, Asian, 38219
Male, 49, Asian, 35312
Female, 50, White, 38218

(a) Sample data

Male, 50, Asian, 38212
Male, 49, Asian, 35312
Female, 63, Asian, 38219
Female, 50, White, 38218

(b) Clustered sample data

Figure 3-1. Clustering of sample data

The technique of clustering has several advantages and one can infer that all the data points in a cluster have a special relationship and that they all are within some small Euclidean distance of each other. We can use this information in order to apply an identical generalization to close points and lower total information loss.

3.1 Clustering techniques

A numerous number of algorithms have been suggested for performing clustering and these algorithms are mainly based on following four categories (Jain, M.N., & P.J., 1999).

Hierarchical clustering: Hierarchical clustering is a technique of building hierarchical clusters. Clustering is performed based on two primary approaches - top down and bottom up. A divisive approach is a top-down approach that focuses on dividing the cluster recursively into separate clusters whereas agglomerative approach is bottom-up clustering approach that joins small clusters

into bigger one until only one cluster remains. The joining of the clusters is done based on two metrics, single linkage clustering, and complete linkage clustering. Based on the performance, the agglomerate clustering techniques are considered to be good as they have a time complexity of $O(n^3)$ as compared to the divisive approach that has a time complexity of $O(2^{n-1})$ (Everitt, 2011). Hierarchical clustering is an approach that provides a specific type of clustering solution.

Centroid-based clustering: Centroid-based technique typically relies on the distance of points from centroids. An example of this technique is Kmeans. In Kmeans, we define K centroids and identify the nearest centroid for each element, and we continue with cluster formation until centroids do not change. This technique typically uses squared error for optimization of the distances within the cluster. Kmeans++ is a variant of Kmeans that uses probabilistic distribution in order to select k centroids in the initialization step.

Distribution based clustering: These algorithms typically rely on the distribution of the data within clusters. An example of this type is Gaussian mixture models, in which we assume a certain number of Gaussian distributions in the data and use the iterative model to fit the model to the dataset. These algorithms work pretty well when there are unobserved/latent variables or missing values in the data, however, this method cannot always be used because data distribution may not always be known.

Density-based clustering: Density-based clustering methods rely on a technique known as density-reachability. E.g., DBScan starts by grouping points that are close to each other, to form a cluster, leaving all the points that are far as outliers. This type of clustering suffers when data has varying densities (Mumtaz K, 2010).

3.2 Clustering technique evaluation

Overall, we are not really interested in the hierarchy of the clusters itself; we just want the bottommost level of clusters, which is why hierarchical clustering does not sound like a right choice. Distribution based clustering "assume" distribution of the data which may not always be known, which is why distribution based clustering may not be the right choice either. Density-based clustering technique works when there are distinct classes available, but because of lack of knowledge about the data, we cannot assume that that distinct classes would be available. We can consider centroid-based clustering as it does not really make any specific assumptions about data distributions or density. Kmeans is an extremely popular centroid-based clustering algorithm and its time complexity is O (n^{dk+1}) (Inaba, Katoh, & Imai, 1994) where

N = total number of points to be clustered

K = number of clusters

D = dimensions in the point

The time complexity of Kmeans suffers from the curse of dimensionality. We cannot afford such high time complexity, as a number of quasi-identifier columns present in the dataset can be very high. Having four quasi-identifier attributes in the dataset is definitely not a rare use-case. We can clearly see that these algorithms do not scale as dimensionality in the data increases. We need solutions that scale well in high dimensional spaces.

The intent of evaluation of clustering technique was to find out nearest neighbors and see if the generalization level applied to a point can be extended to its nearest neighbors. For such problems, we do not really need a concrete cluster boundary. Due to these reasons, we consider

locality sensitive hashing, which is an effective way to find out nearest neighbors for a particular point.

3.3 Locality sensitive hashing

Locality sensitive hashing is a high-performance approximate nearest neighbor (ANN) search algorithm for high dimensional data. It relies on probabilistic hashing of data in such a way that elements that are near to each other have a higher probability of being hashed into the same bucket. It uses multiple LSH family hash functions, hashes all elements into their respective buckets, in turn reducing the neighbor search space. LSH family of hash functions satisfies the property defined in equation (3.1).

$$\begin{aligned} &\forall \text{ row vectors } a, b \in N, \\ &\text{distance}(a, b) \leq d_1 \quad \Rightarrow \quad P(h(a)=h(b)) \geq P_1 \\ &\text{distance}(a, b) \geq d_2 \quad \Rightarrow \quad P(h(a)=h(b)) \leq P_2 \\ &\text{Where} \quad d_1 < d_2 \ \& \ P_1 > P_2 \end{aligned} \tag{3.1}$$

The above function is said to be (d_1, d_2, P_1, P_2) sensitive. For all row vectors a, b belonging to total search space N, if the distance between points a and b is less than or equal to d_1, then the probability of a and b being hashed to the same bucket is greater than or equal to P_1 and if the distance between a and b is greater than d_2, the probability of a and b being hashed into the same bucket is less than P_2 (Locality Sensitive Hashing, 2016).

There are two very popular hash functions that follow the above property. MinHash, based on Jaccard distance, and bucketed random projection, based on Euclidean distance. Jaccard distance typically works better when the distance measure is purely binary and it does not work well with quantitative data. Euclidean distance accommodates quantitative data better by considering the granularity in the data. For our experimentation, we consider Euclidean distance-

based measure $d(x, y) = \sqrt{\sum_i (x_i - y_i)^2}$. Where x = ($x_1$, x_2, x_3 .. x_k) and y = (y_1, y_2, y_3 …. y_k) are any two vectors, each having k attributes. Bucketed random projection method involves multiplying vector x with random unit vector v and assigning it to a bucket by dividing it with bucket length r. Please find the detailed bucketing algorithm explained in section 3.3.1.

3.3.1 LSH - bucketing algorithm

In LSH, we first perform bucketing, i.e. apply all hash functions on all records and store the result in a hash table. We extend this further by forming individual clusters of all elements which were hashed into the same bucket by all hash functions. In ***getBuckets*** method, we implement the formula h(x) =$\frac{x*v}{r}$. This formula is based on LSH scheme for p-stable distributions paper (Datar, Immorlica, Indyk, & Mirrokni, 2004). The paper states that this formula can be used on stable distributions in order to get locality sensitive hashing of data. As per this formula, we generate a random unit vector, multiply the same with the row vector and then divide the dot product by a value "r"; result is the bucket id that vector should be hashed into. LSH ANN search uses multiple such hash functions and narrows down the list of elements on which approximate neighbor search needs to be performed. We use identical mechanism, but instead of executing ANN search, we simply use buckets in order to get summary statistic. In step 3.1 of the algorithm described in Listing 1, we create a concatenated hash as the master hash and use grouping method in order to extract clusters. We use these clusters in algorithms defined in sections 5.5 and 4.4 for defining LSH bucketing based batch anonymization algorithms.

Listing 1:Algorithm for bucketing using LSH

```
Input: LinesRDD
Output: Buckets: RDD [Array [Row]]
getBuckets (linesRDD, precision)
1. Get random unit vectors.
2. For each Row in linesRDD, apply normalization transformation, i.e.,
each row gets converted into an array of Double.
3. for each row vector, do following,
      3.1 Create a "concatenatedHash".
      3.2 For each unit vector do following,
            3.1.1 Multiply row with the unit vector obtained in step 1
and store result in sum.
            3.1.2 Divide sum by "r", this is the bucketID to which the
row gets hashed into based on the current unit vector.
            3.1.3 Round the bucketID to precision provided.
            3.1.3 Append it to concatenatedHash.
      3.3 emit (concatenatedHash, row) tuple.
3. Group rows by concatenatedHash and lose concatenatedHash.
4. Return the RDD [Array [Row]] as buckets.
5. Stop.
```

3.3.2 Categorical data mapping, normalization of quantitative attributes

Before we execute the LSH algorithm, we must preprocess the data. We normalize all numeric attributes. We use normalization formula $\frac{value-mi}{max-min}$ for calculating the normalized value. Apart from numeric values, we convert qualitative attributes, gender, and race, into quantitative data by adding a number of columns corresponding to the number of unique possible values in the column. E.g., gender column has two possible values, male and female, so we create two columns, one corresponding to each value, and map each to a unique index. According to the mapping, attribute male gets mapped to <1, 0> whereas female gets mapped to <0, 1>. Similarly race attribute is converted into quantitative form too.

Apache Spark MLlib 2.1.0 has a built-in provision for LSH algorithm with bucketed random projection technique for numeric data, but it does not expose access to buckets. Also, the buckets are in non-materialized form. For this reason, we chose to implement the bucketing algorithm as part of the study.

CHAPTER 4

K-ANONYMITY TECHNIQUES

According to k-anonymity mechanism, quasi-identifiers must be generalized or suppressed in order to make records “k-anonymous”. That is, there should be at least k records for any given record that have exactly the same attribute generalization level for all quasi-identifier attributes. Attribute generalization is a process of generalizing a value, to a higher dimension value, in the hierarchy of the domain value of the record. E.g., consider the hierarchy for age attribute from Figure 4-1, a generalization of the tuple <age, 45> one level up would mean its conversion into <age, 31-60>.

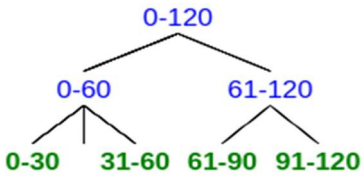


Figure 4-1. Three level age generalization hierarchy

Generalization can lead to information loss, thus it is important to define granular hierarchies for quantitative as well as qualitative attributes. Typically, these hierarchies can be defined based on statistics available for the data. Suppression incurs the highest possible information loss in generalization, i.e. generalizing a record to the topmost level. Suppression of <age, 45> would mean conversion of <age, 45> to <age,*>

4.1 Existing k-anonymity approaches

Optimal k-anonymity is a known NP-hard problem, In order to implement k-anonymity, several heuristic/approximation based algorithms have been suggested (Ramakrishnan, LeFevre, &

DeWitt, 2006). Since we have high dimensionality in data, we decided to implement Mondrian k-anonymity algorithm, which is a high-performance global recoding based greedy approach for implementing k-anonymity in the data. **Global recoding** is a greedy technique for achieving k-anonymity. Global recoding is of two types, single dimensional global recoding, and multidimensional global recoding. Single dimensional global recoding is a process of coming up with generalization or suppression rules for each of the quasi-identifier attributes and applying the same to all rows individually. This involves converting a tuple of the form T (a, b, c), into T'(a',b', c'). **Single dimensional global recoding** involves partitioning the data based on a single region. Single dimensional partitioning typically takes a value as an input and applies a function that maps the value to its region. **Multidimensional global recoding** is the process of defining a single rule for specific tuples in the dataset and applying the same to all. Multidimensional global recoding involves creating rules that are more granular, and thus gives better results. Multidimensional partitioning extends the concept of single dimensional global recoding to multiple dimensions. A multidimensional partition requires multiple attributes $<att_1, att_2, att_3, ..att_d>$ for definition of a region. We call a partitioning as strict partitioning when these regions when do not overlap, and relaxed partitioning when they do. E.g., with relaxed partitioning, <zip code, 51223> can be generalized into <zip code, 51233-51234> or <zip code, 51232-51233> whereas, with strict partitioning, zip code of 51223 can be generalized only into one of the two.

Table 4-1.Single dimensional partitioning

Sr. No.	Age	Gender	Zip Code	Salary
1.	36-39	Male	22710-22711	>50k
2.	36-39	Female	22712	<=50k
3.	36-39	Male	22710-22711	>50k
4.	36-39	Male	22710-22711	<=50k
5.	36-39	Female	22712	>50k
6.	36-39	Male	22710-22711	<=50k

Table 4-2. Multidimensional partitioning

Sr. No.	Age	Gender	Zip Code	Salary
1.	36-37	Male	22710	>50k
2.	36-38	Female	22712	<=50k
3.	36-37	Male	22710	>50k
4.	38-39	Male	22710-22711	<=50k
5.	36-38	Female	22712	>50k
6.	38-39	Male	22710-22711	<=50k

In Table 4-1 and Table 4-2, we can clearly see that single dimensional anonymization has higher loss as compared to multidimensional 2-anonymization.

4.2 Mondrian multidimensional k-anonymity algorithm

Mondrian k-anonymity algorithm is a greedy algorithm that performs multidimensional cut until no more cuts can be made without violating the k-anonymity property. A multidimensional cut is the division of data into two subsets in such way that both the subsets follow the k-anonymity property. Please find the algorithm in Listing 2.

Listing 2. Top-Down greedy algorithm for multidimensional recoding from (Ramakrishnan, LeFevre, & DeWitt, 2006)

```
Input: Partition of rows
Output: k-anonymized Partition
k-anonymize (partition)
1. If multidimensional cut is not possible,
     a. Return summary (partition).
2. Else
     a. dimension ← selectDimension().
     b. leftSet & rightSet ← splitOnMedian(partition, dimension)
     c. return k-anonymize (leftSet) U k-anonymize (rightSet)
```

When it comes to handling huge data, non-distributed frameworks typically do not scale well, because of their linear nature, which is why we decided to use Apache Spark.

4.3 Distributed Mondrian multidimensional k-anonymity algorithm

Apache Spark converts dataset into the form of an RDD and RDD allows Apache Spark to execute code in parallel. Along with the data, we also read metadata file that contains hierarchy metadata for each of the column in the data. The metadata, as well as data used for the experiment, is defined in APPENDIX A. We read the metadata and use Apache Spark broadcast variable feature in order to remove the need for the variable to be distributed with each of the tasks. Before we start the algorithm, we preprocess the data and convert textual CSV data into following map format.

i.e., Row of <15, Female, Asian, 38363> gets converted into RDD[0,Map[(0,15),(1,Female),(2, Asian),(3,38363)]…].

In Mondrian multidimensional k-anonymity algorithm, selection of dimension is made through a method called ***selectDimension***. Please find algorithm for the same described in Listing

3. The Mondrian multidimensional k-anonymity paper (Ramakrishnan, LeFevre, & DeWitt, 2006) suggests that dimension selection can be made in two ways, one based on a number of normalized values and second based on anticipated workload. We choose the prior, with strict partitioning. That is, we select attribute with the maximum number of unique values, and we do not overlap ranges. Range based selection does tend to give preference to the numeric attribute selection due to inherent variability present in the numeric data.

Listing 3. Distributed selectDimension method

```
selectDimension (linesRDD, k)
1. For Each record, emit (attribute-index, attribute-value) tuple and
   cache the result.
2. Aggregate unique values for each column.
3. Aggregate the number of occurrences for each value for each column
   and store it as frequency.
4. Select column with the maximum number of unique values from step 2
   as the column on which cut should be performed.
5. If column type is Categorical
   1. Loop over frequency map of categorical attribute and based on
      frequency, divide the set into two parts, leftSet and rightSet and
      return the same.
6. If column type is Numeric,
   1. Sort values corresponding to selected column, select min, median,
      and max and return the same.
7. In the case of error, return -1 as the dimension.
```

In the Mondrian k-anonymity algorithm, when we partition, we associate the summary statistic with the partition itself. Paper suggests that there are two possible ways to associate the summary statistic and range statistic. For this study, we choose range statistic instead of the summary statistic. As we can see, the algorithm in Listing 4 looks very much identical to the Mondrian multidimensional k-anonymity algorithm from paper (Ramakrishnan, LeFevre, &

DeWitt, 2006) but it does take advantage of distributed processing supported by Apache Spark. It is important to note that algorithm relies heavily on the selection of dimension.

Listing 4. Distributed Mondrian k-anonymity algorithm

```
distributed-K-anonymize (linesRDD: RDD, K)
1. Set dim = selectDimension (linesRDD, k).
2. If dim is less than 0, execute assignSummaryStatistic (linesRDD).
3. Else do following,
      a. If dim is of type categorical, find leftRDD and rightRDD by
         using leftSet and rightSet for selected dimension.
      b. Else, divide linesRDD based on numeric ranges into leftRDD and
         rightRDD.
      c. If both datasets are of a size greater than or equal to K,
         then follow following logic otherwise call
         assignSummaryStatistic on linesRDD itself.
           i. If the size of leftRDD is greater than "K" then call K-
              distributed-K-anonymize (leftRDD), if it is equal to K,
              then call assignSummaryStatistic (leftRDD).
          ii. If the size of rightRDD is greater than "K" then call K-
              distributed-K-anonymize (rightRDD), if it is equal to K,
              then call assignSummaryStatistic (rightRDD).
```

The ***assignSummaryStatistic*** method calculates the summary statistic for a region and assigns the summary to all quasi-identifier attributes.

4.4 LSH based k-anonymity algorithm

In this section, we propose a distributed one pass algorithm for implementing k-anonymity using LSH bucketing approach. The main idea behind k-anonymization is finding the optimal set of entries that should be part of the k-anonymized set. The problem of optimal k-anonymity remains NP-hard mainly because it is impossible to find the optimal set of K entries that should belong the set to be anonymized. However, we can take a greedy approach and find a set of entries that

definitely should belong together. This is where LSH comes into the picture. As explained in Section 3.3, LSH is an algorithm for finding approximate nearest neighbors in sublinear query time, because of its bucketing technique. We can use this property of LSH to find out optimal set of elements of size k and then generalize them by simply calculating a range statistic. In the algorithm described in Listing 5, we first execute ***getBuckets*** method on an entire dataset with very high precision. This leads to almost identical values being hashed into same buckets. We convert all those buckets that have at least k elements into summarized versions by calling the ***assignSummaryStatistic*** method. This method is same as the one from Mondrian k-anonymity implementation. We now simply suppress all buckets that have less than k elements. To ensure we get good accuracy, we do simulation in order to decide the optimal value for number of hash functions to be used as well as bucket id precision.

Listing 5. LSH k-anonymity algorithm

```
Input: linesRDD [Long, Map [Integer, String]]
Output: k-anonymized dataset
LSH-k-anonymity (linesRDD, k)
1. Compute buckets by calling getBuckets(linesRDD)
2. Store buckets that have a size greater than or equal to k, in a
   variable called neighbors.
3. Execute assignSummaryStatistic(neighbors) and add result to
   outputRDD.
4. Suppress all buckets with size less than k and add the result to
   outputRDD.
5. Merge all outputRDD values and write result to output.


Input: lines [Long, Map [Integer, String]]
Output: Summarized dataset
assignSummaryStatistic (lines)
1. For each line, emit (attribute-index, attribute) tuple.
2. For each Attribute index, do following
      a. If the attribute is a categorical type of attribute, find
         distinct entries. Use utility functions to find the common
         ancestor of all entries and set it as the range.
```

```
      b. If the attribute is a numeric type of attribute, find min and
         max. Set min_max as the range.
3. For all rows, update value to the range calculated in step 2.
```

Summary statistic calculation is done on numeric data based on min and max found in the neighbor list and range of categorical data is decided based on the nearest common ancestor. We then replace the value of attributes by its corresponding range. These records are then written to the output. The output of algorithm can be further improved by adding multiple iterations of LSH bucketing but at the cost of performance. Based on the data, tradeoff analysis can be done and a number of iterations can be added in order to improve the accuracy of this algorithm.

CHAPTER 5

RISK-BASED APPROACH

Several data sanitization approaches have been suggested for protecting privacy, most revolving around k-anonymity. These algorithms focus on anonymization of data to the extent that the data becomes unidentifiable. These approaches, however, lead to a huge information loss. In recent past, a completely new approach towards the problem was suggested, risk-based approach, according to which, we release the data at a certain generalization level based on the risk of re-identification of the data (Wan, et al., 2015). We consider one such solution identified in an article published on a game theoretical approach for analyzing re-identification risk (Wan, et al., 2015).

In order to illustrate the risk-based approach, let's consider a hypothetical scenario. Imagine there is a publishing company that wants to publish some health data to potential data buyers. Although data buyers claim that they intend to use the data for research purpose, they might have the malicious intention of re-identifying identities of individuals corresponding to the data. In order to avoid the re-identification, publishers have the option of applying k-anonymity algorithms. Publishers, however, realized that sharing highly generalized data reduces monetary gain hence publisher is willing to take some risk in order to get a higher price for the shared data without wanting to lose money in the process. Adversary (or data buyer) does want to perform re-identification attack but not at the cost of losing money. In order to achieve such optimal level of generalization, the paper (Wan, et al., 2015) suggests an algorithm called lattice based search (LBS).

A record lattice can be imagined as a lattice that contains the original record at the top and attribute-generalized children at subsequent levels. The sample lattice from the paper is as shown

below. We can see that (Amer-Indian, 42) is the topmost level record. While constructing lattice, we generalize only one child at a time.

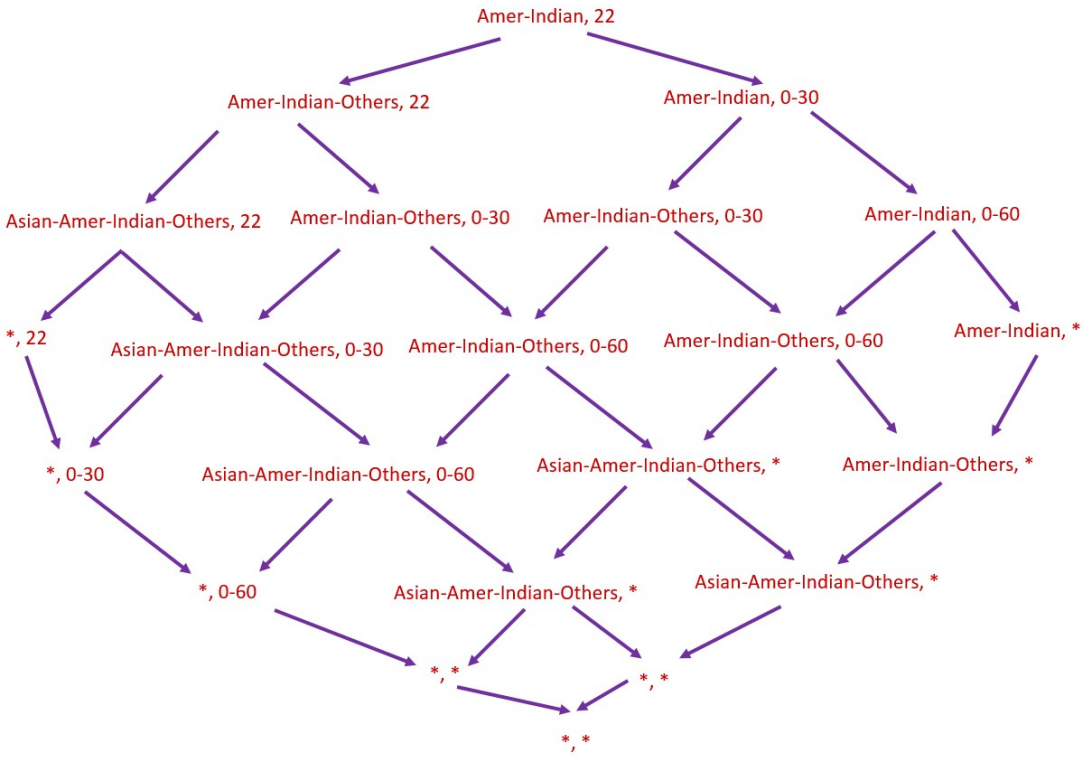


Figure 5-1. Sample lattice based on 3 layer hierarchy of age and race

In Figure 5-1. Sample lattice based on 3 layer hierarchy of age and race, we draw a lattice for the record (Amer-Indian, 22) based on the generalization hierarchy defined in Figure A-4. Age generalization hierarchy. Please note that in above lattice, Amer-Indian-Eskimo was renamed as "Amer-Indian" for keeping texts smaller. Based on generalization hierarchy, we know that age 22 can be generalized into the range 0-30, and similarly, race Amer-Indian can be generalized into Amer-Indian-Others. Hence, the topmost node has two children <Amer-Indian-Others, 22> & <Amer-Indian, 0-30>. Each of them has their own two children, <Amer-Indian-Others, 0-30> being the common child of the two. This continues until the hierarchy has been exhausted. All

generalizations end in the top most level of generalization, i.e., complete suppression, leading to the formation of a lattice instead of a tree. The risk-based approach involves evaluation of lattice for each record. We find the level of generalization for every record that reduces the risk of the record being re-identified while sharing maximum information.

5.1 Algorithm parameters

To be able to illustrate algorithm better, we first define algorithm parameters in Table 5-1.

Table 5-1. Algorithm parameters for evaluating a game theoretical model

Sr. No	Parameter	Description
1.	R	Record in its original form.
2.	V	The benefit publisher receives when he/she shares the record in original form(R) and adversary does not perform the attack.
3.	L	Amount publisher loses when an attacker succeeds in re-identification of the individual.
4.	C	Amount adversary pays for buying the record.
5.	g	The generalized form of the original record.
6.	V(g)	Publisher benefit when the record is shared at a generalization level 'g'.
7.	$\pi(g)$	The re-identification risk of the record. This would be 1 if individual corresponding to the record is clearly identifiable and 0 if there exist quite a few individuals at this level of generalization.
8.	IL(g)	Information loss incurred when record R is shared at generalization level 'g'.
9.	g_c	Child of generalization level 'g' obtained by a generalization of one of its attributes.
10.	f	Feature/attribute in the record.

5.2 Anonymization evaluation technique

The benefit publisher receives by sharing record in its generalized form 'g' is calculated based on formula shown in equation (5.1). The formula calculates V(g) based on the ratio of information loss [IL(g)] present in generalization level g and the max information loss possible[Max(IL(g))].

$$V(g) = V \text{ x } \left(1 - \frac{IL(g)}{Max(IL(g))}\right) \tag{5.1}$$

The information loss is derived by the total number of entries that are present at given generalization level.

$$IL(g) = \sum_f - \log\left(\frac{1}{size(Generalization\ range\ of\ f\ in\ g)}\right) \tag{5.2}$$

For numeric feature f, generalization range of the feature can be calculated by subtracting f.min from f.max, for selected generalization level g. For the categorical feature, generalization range would be a complete list of possible values at selected generalization hierarchy level. To understand this formula better, let us consider two simple examples. First, a single numeric attribute, and second, a single categorical attribute. Let us first consider numeric attribute. For a generalized record of (age, 40-49) containing a single feature age, the information loss would be calculated by calculating range.max minus range.min, i.e. 49 minus 40, taking the inverse of the value, taking the log of the inverted value, and negating the same. Similarly, for a generalization record (race, White OR Black OR African American), the information loss would be calculated by calculating the total number of unique categories (i.e. two), taking inverse and then taking the log of the value and negating the same. Typically, overall information loss can be calculated by taking summation of log values for each feature, as defined in equation (5.2). Similarly, we can calculate the max information loss by applying information loss formula on the suppressed record. The formula for the same is outlined in equation (5.3)

$$\text{Max (IL(g))} = \sum_f - \log\left(\frac{1}{size(Generalization\ range\ of\ f)}\right) \quad (5.3)$$

Apart from information loss, the algorithm also calculates the risk of the record by using the formula (5.4) on the entire population.

$$\pi(g) = \frac{1}{Population\ size(g)} \quad (5.4)$$

Calculation of population size for (40-49,*, White, 39360-39369) can be done by taking Cartesian product of each of the attribute, i.e. population size of [*(40,41,42,43,44,45,46,47,48,49) X (Male, Female) X (White) X (39360, 39361, 39362, 39363, 39364, 39365,39366,39367,39368,39369)]*. After counting total number of unique individuals, we take the inverse and obtain the risk of generalization level.

5.3 LBS algorithm

Listing 6 describes the LBS algorithm from the paper (Wan, et al., 2015). The algorithm starts with the initialization of g; it is initialized to value R, i.e. the original record itself. While g is not the completely suppressed form of the original record, the search continues by traversing through its most optimal child. If loss multiplied by the probability of successful identification exceeds the record cost, it means that adversary would be benefited if he chooses to perform re-identification attack. However, if that is not the case, we explore all the children of record g as long as we get better payoff than the record itself. If we see that any child has a better payoff than the parent, we select the child and explore its children in order to find the minimum risk child in the lattice. For selected strategy, if we find that $L*\pi(g) <= C$, then adversary won't have any reason to perform the attack hence algorithm would stop. The algorithm from Listing 6 looks performance intensive,

but the bottleneck actually exists in risk calculation function. This function returns a total number of entries present in current generalization level.

Listing 6. LBS algorithm (Wan, et al., 2015)

Input: Lattice entry point - R
Output: Risk optimal generalization level for R
findOptimalGeneralization (R)
1. Initialize g = R.
2. While g is not completely suppressed generalization, do,
   a. If L* $\pi(g)$ <=C, it means that if adversary attacks, he or she would not get any benefit, hence g is the optimal generalization level, return the same.
   b. Otherwise, initialize the highest payoff U_m =V(g)-L*$\pi(g)$, and assign g_m=g.
   c. For Each child g_c of g,
      i. If L* $\pi(g_c)$> C, then consider V(g_c) - L * $\pi(g_c)$ as payoff, else consider V(g_c) as payoff.
      ii. If the payoff is greater the highest payoff U_m, Then update U_m.
      iii. If none of the children of "g" had a greater payoff than its parent, then return "g" as the optimal generalization level.
      iv. Else update g=g_m and continue from step 1.
3. Stop.

The risk function calculates risk for every child in the lattice, and since lattice would be sparse, because of huge range in zip code column, the function would suffer when executed serially. The above algorithm can be easily extended to scale on distributed frameworks. We can leverage the inbuilt parallelism present in Apache Spark in order to compute the generalization levels for records in parallel.

5.4 Distributed LBS algorithm

Distributed LBS algorithm looks exactly like standalone version itself. Apache Spark reads CSV file in the form of RDD. The RDD supports parallelism inherently and launches ***findOptimalGeneralization*** on multiple worker nodes concurrently. Apart from this, instead of computing inputs required for calculation of risk on each node, we can use the broadcast variable feature of the Apache Spark framework and distribute the centrally computed population hash map across all executors.

Listing 7. Distributed LBS algorithm

```
Input: file path
Output: Risk optimal generalization level for all records
findOptimalStrategy ()
1. Read input file as an RDD.
2. Iterate over RDD in the following manner.
     Set generalizations = rdd.map({ case (x, y) (x,
     findOptimalGeneralization(y)) })
     This causes findOptimalGeneralization method to be executed in
     parallel on all executors.
3. Store output.
4. Stop.
```

We can see that distributed version specified in Listing 7, achieves the parallelism by converting the input into the form of an RDD and thus achieving parallelism at the record level. This means that each record can be evaluated in parallel and result will be aggregated in the end. Currently, Apache Spark does not support nested evaluations of RDD, but once it does, we can convert g's children into RDD, and evaluate all children in parallel by using a map and a filter method. Also, another compute heavy code is the one that calculates risk for selected generalization level. This portion of the code can be further parallelized by converting

combinations into RDD and evaluating in parallel. The approach we take for calculation of record risk has a direct impact on the performance of the LBS algorithm. Using map is the optimal way of performing this search. An optimal strategy must be chosen in order to balance the performance and memory for performing this search. After careful study, we decided to have two maps, one for top value lookup, HashMap ((race, gender, age, zip code), population size), and second, for generalization lookup, HashMap ((race, gender), TreeMap (Age, TreeMap (zip code, population size))).

5.5 LBS-LSH approximation technique for dense data

Distributed LBS-LSH algorithm tries to reduce the number of records for which LBS algorithm gets executed. As shown in Listing 8, We start by bucketing all records using method outlined in section 3.3.1 and then iterating over each bucket. While iterating, we first execute LBS on the first record and see if its generalization is applicable for all remaining entries in the bucket. For those records, for which entry is applicable, we apply the generalization, for rest we compute LBS. This approach leads to the LBS calculation complexity reduction proportional to the denseness of the cluster. If clusters present in the data are sparse, the above technique does not yield very good results. This is because LBS itself preserves very high information of original record, leading to lesser number of neighbors being capable of sharing the generalization level.

Listing 8. LBS-LSH algorithm

```
Input: LinesRDD, LBS parameters - pubBenefit, recordCost, loss, K –
Number of neighbors
Output: risk anonymized dataset
lbslsh (linesRDD, lbsParams, K)
 1. Call getBuckets method and store output in a variable called buckets.
```

```
2. For each bucket in buckets, do following,
      a. For the first entry in the bucket, perform lattice based search
         and get generalization hierarchy.
      b. Declare an empty List of rows and add the generalization from
         step 2.a to the same.
      c. For each remainingEntry in the bucket, do following,
           i. If generalization from step 2.a is applicable for
              remainingEntry, add the row_id and generalization to the
              list.
          ii. Else, perform a lattice-based search on remainingEntry and
              return the same.
      d. Return the list.
3. Group output and write it to a file.
```

Discussion:

Although LBS-LSH is designed to improve the performance of LBS algorithm, it does assume that the memory overhead of LSH is very small. It also assumes that LSH bucketing is performed with extremely high precision so that buckets contain only near-duplicate elements. LBS-LSH would suffer badly if any of the above two assumptions does not hold.

CHAPTER 6

SCALABLE ON-THE-FLY SANITIZATION ARCHITECTURE ON APACHE SPARK

Sometimes, batch sanitization is not an option, as data sanitization needs to be done on-the-fly. For such scenarios, we need to modify the execution framework itself, but this, however, is not always possible. Sometimes the requirement is to change the behavior without modifying the software. This can be achieved through aspects. Aspects provide a way to modify the behavior of the system by altering bytecode at runtime.

6.1 How do aspects work?

Aspect-Oriented Programming (AOP) is about the implementation of modularity to serve cross-cutting concerns. Aspects are typically used when a functionality needs to be added, amended, or removed at runtime. Function name logging, transaction management across a suite of classes, access control over a variety of classes are typical instances of cross-cutting concerns for which AOP is used. Typically, the elements on which cross-cutting concern needs to be applied are scattered. AOP enables us to add extra functionality to an existing method without modifying the method itself. Consider following example.

Problem Statement: *We have close to 1000 algorithms implemented in UTD's historical algorithm implementation package edu.utd.common.algo". Each class has a method called "describe" that prints the description of the algorithm. The describe method can be called by any other program directly, and execution takes place on a centralized Apache Spark cluster. We need to print a copyright message just after algorithm description has been printed.*

When we look at the above problem statement, the first thought that comes to our mind is to modify the source code of all 1000 algorithms. The solution will not work if the source code is not available. Even if somehow we find the source code, the strategy would still fail if there are a million such programs. In such a scenario, modifying each program is not a valid solution. This type of problem falls into the category of cross-cutting problems. For solving such types of problems, aspect-oriented programming is used. An aspect consists of an advice that describes how the method must be modified, and a pointcut that specifies which behavior needs to be modified. There are three types of advice – ***before***, ***after*** and ***around***. The ***before*** advice executes just before the method and thus allows us to modify inputs to the original method. The ***after*** advice allows us to modify the output of the original method. The ***around*** advice allows us to modify input as well as the output of the method. Aspectjweaver can be started in the background of the JVM. It scans for META-INF/aop.xml file that contains details of aspect to be weaved in, as well as a list of types to weave. Please find a sample aop.xml in Figure 6-1.

```
<aspectj>
        <aspects>
                <aspect name="edu.utd.security.blueray.AccessAuthorizerAspect" />
        </aspects>
         <weaver options="">
                 <include within="org.apache.Apache Spark.rdd..*"/>
                 <include within="edu.utd.security.blueray..*"/>
        </weaver>
<aspectj>
```

Figure 6-1. Sample AOP.xml

Aop.xml has two key tags - aspects and weaver. The ***aspects*** tag is used to specify aspects to be weaved in. All aspects that we want to weave must be specified in this tag. The ***weaver*** tag can be used for specification of types or packages that need to be woven. If no include tags are

specified, weaver tries to weave all classes thus degrading the overall aspect injection performance. Hence, we must specify the include tag. AspectJWeaver does the weaving of the aspect at load-time. This means that every time a method gets loaded in the JVM, AspectJWeaver will modify its bytecode and attach the additional behavior, defined in the advice, to the method. AspectJWeaver has access to all classes that are loaded in its JVM.

Benefits of aspect-oriented approach: Aspect-oriented approach lets us separate cross-cutting concerns. The overhead of modifying, maintaining local copies of modified code, and updating code as framework evolves is completely removed by implementing the cross-cutting approach using AOP. Aspects do incur little performance overhead, but it is extremely low considering advantages it offers. The aspect-oriented approach does provide us a convenient way for modifying data on-the-fly.

6.2 Scalable on-the-fly sanitization architecture - Apache Spark & AOP

For distributed on-the-fly sanitization, we need an innovative solution. Before we design a system for on-the-fly data sanitization approach, let us first look at how Apache Spark executes a program. For simplicity purposes, we submit the program to Apache Spark in a standalone mode. When we submit a program, a workflow that gets executed is described in Listing 9.

6.2.1 Current Apache Spark Work-Flow

Whenever an input file is being read, Apache Spark converts it into an RDD. RDD is a logical unit that gets mapped to MapPartitionsRDD depending upon the number of splits done on the input HDFS file. MapPartitionsRDD is computed whenever any of the Apache Spark actions is

executed. This RDD is typically computed on worker machines that are nearest to the data. Please find workflow described in detail, in Listing 9.

Listing 9. Apache Spark Workflow

Apache Spark Workflow

1. The driver reads input File and creates an RDD.
2. The driver then converts RDD into a set of MapPartitionsRDD.
3. Driver scans user program and computes a DAG.
4. Whenever any action gets called, its DAG is sent to DAGScheduler for execution.
5. DAGScheduler controls executions of stages.
6. Each stage consists of tasks on individual MapPartitionsRDDs.
7. Each MapPartitionsRDD is assigned along with its task closure, to an executor near the data node on which partition exists.
8. Executor invokes the ***compute*** method that has following signature.

```
override def compute(split: Partition, context: TaskContext):
                Iterator[U]
```

9. The ***compute*** method returns the iterator for given partition.
10. Task uses an iterator to iterate and performs the desired action.
11. Task completes.

We propose an aspect-based design for performing on-the-fly sanitization. In order to perform sanitization on-the-fly, we must inject our aspect in above workflow with minimum overhead. This is the reason why we modified the behavior of the ***compute*** method itself instead of modifying individual action methods.

6.2.2 Modified Apache Spark Execution workflow

As we can see in Figure 6-2, we have injected AspectJWeaver on each of the executors. AspectJWeaver is responsible for BlueRay aspect getting loaded in the executor's JVM. This Aspect has full access to executor JVM.

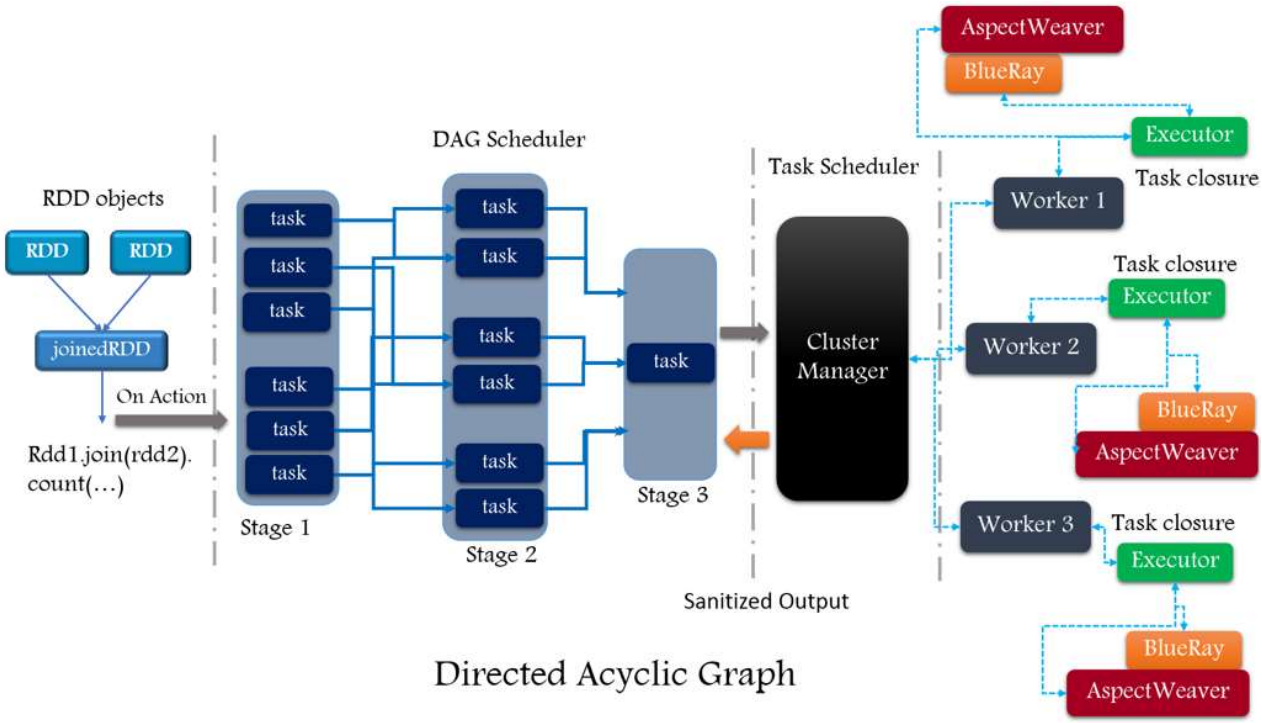


Figure 6-2. Modified Apache Spark workflow

6.2.3 BlueRay aspect

We can see that aspect described in Listing 10 is an ***around*** aspect which joins at the ***compute*** method of org.apache.spark.rdd.MapPartitionsRDD class. Here, we have used ***around*** method because we need to read the inputSplit's file path in order to apply policy, and output because we want to return the sanitized iterator instead of a normal iterator.

Listing 10. BlueRay aspect: Scala source code

```
@Aspect
class BlueRayAspect
{
     @Around("execution(*
     org.apache.apache.spark.rdd.MapPartitionsRDD.compute(..)))
     aroundAdvice(jp: ProceedingJoinPoint, partition: Partition,
     context: TaskContext)
        {
```

```
        val iterator = (jp.proceed(jp.getArgs()));
        val policy = getPolicy(context, jp);
        if (policy != None)
        {
           return new SanitizedIterator(context, iterator, policy);
        }
        return iterator
      }
}
```

SanitizedIterator is a subclass of org.apache.spark.InterruptibleIterator, a typical iterator class which has two methods ***hasNext*** and ***next***. ***SanitizedIterator*** overrides ***next*** method in order to return the sanitized version of the original value. The ***compute*** method's one of the inputs is partition details. We find the policy details by extracting the filename from the inputSplit attribute of the partition. Once found, we send the policy to ***SanitizedIterator***.

6.2.4 Architectural assumptions:

We assume that modified architecture is controlled by having complete control over how the user gets to submit the program. Also, we assume that program's output is a simple output that is written to the console. To facilitate this, we assume that the access to Apache Spark cluster is controlled through a web-application, where the user uploads the source code he/she intends to execute. This code is then executed on the Apache Spark cluster by injecting the AspectJWeaver at runtime. The output of the program is displayed on the web-console once it has finished execution. The architecture is assumed to be very simplistic and can be easily extended for meeting realistic expectations.

6.2.5 Policy manager

We assume a very simple policy model. A policy comprises of three things – role or username, file path, and text or REGEX (regular expression) to be sanitized. Typically, file path would be obtained from the partition, role or username can be extracted from task's context or system configuration and text or regex to be sanitized would be maintained per file, in the policy store. Policy manager can be local or centralized.

Distributing policy store on slaves increases headache of policy store maintenance. Although this can be mitigated by maintaining policies on a common file server that is accessible to all policy manager, it would lead to additional network cost for each policy store read.

Alternately, policy manager can be a web application deployed in Tomcat server residing outside the cluster, but on the same network. This way Apache Spark executors do not incur the network overhead while communicating with policy manager. As part of the study, we implemented both types of policy managers and evaluated their performance. Please find APIs supported by policy manager described in Table 6-1. The /enforcePolicy endpoint lets the user add a new policy to be enforced at runtime, the /deregisterPolicy endpoint simply lets the user take down a policy, and /policies lists currently registered policies. We deploy the BlueRay aspect in Apache Spark cluster by bundling BlueRay aspect and aop.xml in a JAR file and keeping it in <Spark_Home/jars> folder on all nodes in the Apache Spark cluster. This simply takes care of the loading process. Whenever Apache Spark program is executed, all JAR files present in Spark_Home/jars are loaded into the JVM.

Table 6-1. Policy manager API (RESTful)

Sr. No.	API endpoint	Type	Input	Output	Description
1.	/enforcePolicy	POST	filePath, REGEX, role OR username	Boolean	This API is used for registering new policy with the manager.
2.	/deregisterPolicy	POST	filePath, role OR username	Boolean	This API is used for deregistering the policy in real-time.
3.	/policy	GET	filePath, role OR username	Policy	This API is used for retrieving policy details.
4.	/policies	GET	None	List[Policy]	This API returns a complete list of policies that are registered with policy manager.

After deployment, along with Apache Spark jars, BlueRay.jar also gets loaded in the executor JVM. Now all that is required is registering the aspect present in the jar with AspectJWeaver, which would, in turn, weave the advice around ***compute*** method on the executor. This is done at runtime by specifying AspectJWeaver javaagent as shown in Listing 11. Apache Spark allows the user to specify extra JVM options through spark.executor.extraJavaOptions and driver-java-option properties. We use these properties to tell JVM to start AspectJWeaver daemons that would have access to all classes in its host JVM.

Listing 11. Apache Spark submit command with AspectJ injection

```
./Apache Spark-submit --conf "spark.executor.extraJavaOptions=-
javaagent:/data/blueray/aspectjweaver-1.8.5.jar"    --driver-java-
options "-javaagent:/data/blueray/aspectjweaver-1.8.5.jar" --class
<Class_Name>  --master "Apache Spark://cloudmaster3:7077"
<CLass_Jar_Path> <Class_Args>
```

We do so by injecting our aspect at executor as well as driver level. This is done by sending javaagent parameter containing the full path of aspectjweaver.jar. The assumption is that AspectJWeaver is present on all nodes of Apache Spark cluster. After setting up above environment, the Apache Spark execution flow gets modified as described in Listing 12.

Listing 12. Apache Spark - new workflow

1. Whenever a spark-submit command is executed, a driver is launched in the cluster.
2. The driver loads BlueRay aspect jar along with AspectJWeaver jar.
3. This makes sure that the bytecode representation of MapPartitionsRDD on the driver is consistent with executors.
4. The worker is responsible for launching executor. Worker launches executor and we can see the following command in worker log.
   ExecutorRunner: Launch command: "/home/cloud/pkg/jdk/bin/java" "-cp" "/Cloud/Apache Spark-2.1.0-bin-hadoop2.7/conf/:/Cloud/Apache Spark-2.1.0-bin-hadoop2.7/jars/" "-Xmx1024M" "-Dspark.driver.port=58584"* ***"-javaagent:/data/blueray/aspectjweaver-1.8.5.jar"*** *"-XX:MaxPermSize=256m" "org.apache.spark.executor.CoarseGrainedExecutorBackend" "--driver-url" "<driver_url>" "--executor-id" "0" "--hostname" "192.168.4.11" "--cores" "8" "--app-id" "<app_id>" "--worker-url" "Apache Spark://Worker@192.168.4.11:50547"*

Whenever a new executor is started, all jars present in Spark_Home/jars are loaded in the classpath of the JVM. This also loads BlueRay aspect and modifies the execution of ***compute*** method of MapPartitionsRDD.

5. The driver reads input File, creates an RDD, and then converts the RDD into MapPartitionsRDD.
6. Driver Scans user program and computes its DAG.

7. Whenever any action gets called, its DAG is sent to DAGScheduler for execution.
8. DAGScheduler controls executions of stages.
9. Each stage consists of tasks on individual MapPartitionsRDD.
10. Each MapPartitionsRDD is assigned to an executor present near or on the data node on which data exists.
11. Instead of executing the original MapPartitionsRDD class's ***compute*** method, the bytecode of the class created by the advice gets executed. This class follows the following process.

    1. It extracts the exact filename from the input split metadata of the partition.
    2. It then contacts policy manager and requests policy for the filename found in the partition.
    3. Policy manager returns policy if a policy exists.

12. If the policy is available, SanitizedIterator is returned. Otherwise, the original iterator or blocking iterator is returned – based on the requirement.
13. Tasks use iterator received and performs the desired action.
14. Task completes.

The ***SanitizedIterator*** now returns sanitized wrapper around original RDD iterator. The behavior of this wrapper can be defined in multiple ways. We can make it generalize a record, suppress a record, or even call a record level anonymization algorithm and make wrapper return the algorithm's output. Please find different approaches elaborated below.

6.2.6 Suppression with SanitizedIterator

The ***SanitizedIterator*** extends ***InterruptibleIterator [T] (context, delegate)*** class from Apache Spark and is expected to have two methods by contract. The first one is ***hasNext*** and the second one is ***next***. When we sanitize a value, we replace the value by some other value. A simple ***SanitizedIterator*** simply replaces the value by a string of predefined characters like a dash or a star. MapPartitionsRDD typically works on string data, so the replacement becomes pretty

straightforward while working on Apache Spark. However, for Apache Spark-SQL, iterator of UnSafeRow is returned. This is an instance of a special class, and the object must be carefully modified to return the new instance. Data suppression is a simple way for removing identifier fields at runtime. Identifier fields should always be removed from the data before it is shared otherwise it leads to the identification of individuals. We could define a simple policy that suppresses given columns completely. We can easily convert the original data into data with suppressed identifier by simply executing a program that reads a file, and then writes the RDD to another file and lets BlueRay aspect with ***SanitizedIterator*** take care of the suppression of identifiers.

6.2.7 Data generalization with GeneralizationIterator

GeneralizationIterator is an extension of sanitized iterator with a key difference; it generalizes columns instead of suppressing them. Along with the policy, it also requires metadata. ***GeneralizationIterator*** makes the assumption that the data to be processed is in accordance with the metadata provided in the metadata file. ***GeneralizationIterator*** also requires additional runtime data like column indices to be blocked and a total number of columns expected in the data. In order to specify the same, we have to set following environment variable in Apache Spark executor.

BlockColumns=Num_Columns [List of columns to be generalized]

If the RDD's parent data file has a policy associated and the RDD has a total number of columns same as specified in ***Num_Columns***, the iterator splits the value by commas and replaces the data at columns specified in the environment variable, by their parent categories. Implementing ***GeneralizationIterator*** for identifier field generalization is definitely not a great idea as keys are

arbitrary and they do not follow any specific hierarchy. However, it is a perfectly good solution for any general purpose anonymization requirement that focusses on reducing the amount of information provided in a certain column.

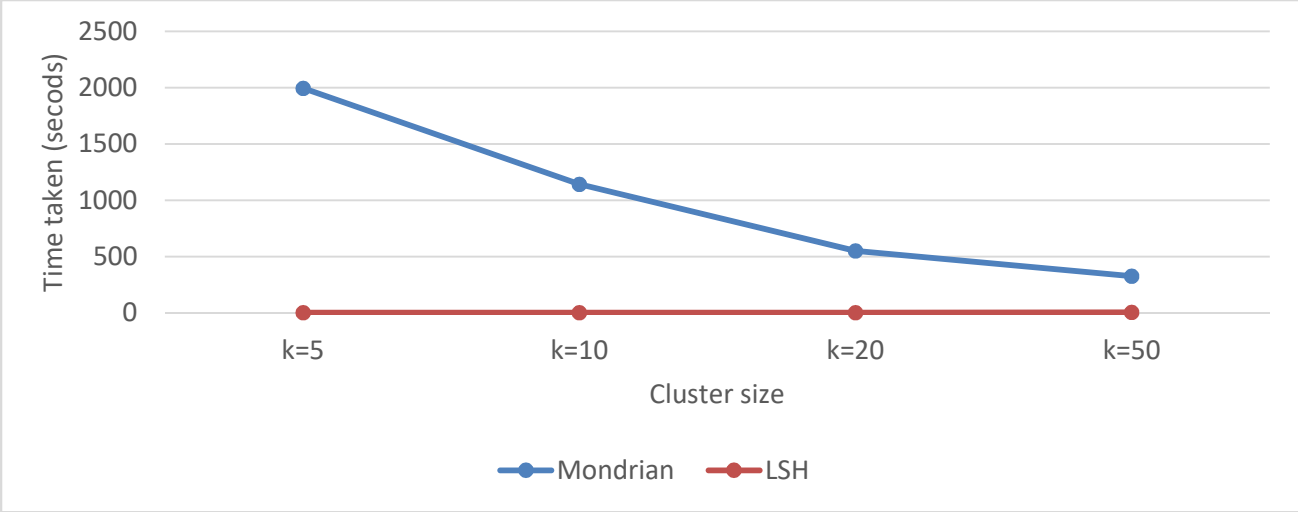
CHAPTER 7

EXPERIMENTAL EVALUATION

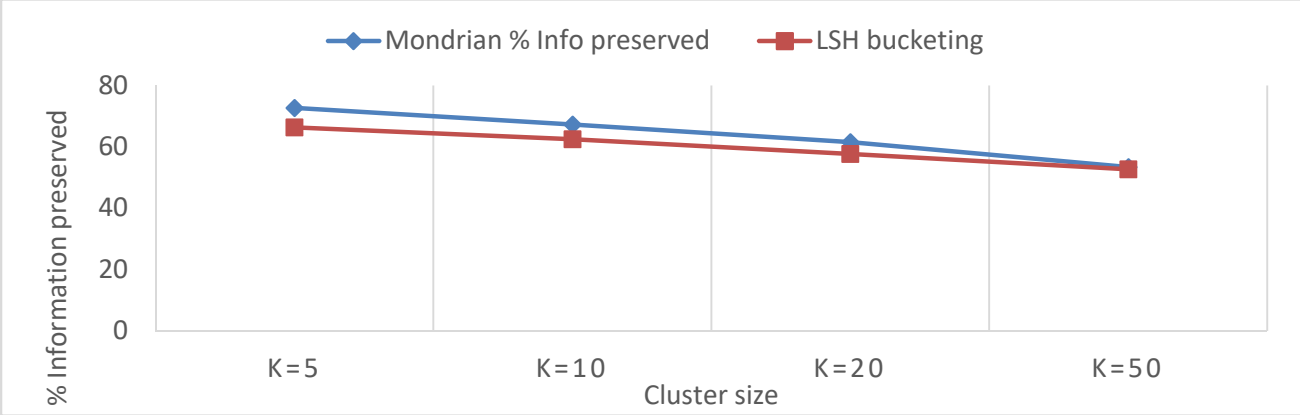
In this section, we compare different data sanitization implementations covered as part of this study. We start by comparing strict within-dataset k-anonymity approaches and then move onto comparison of the same with newer risk-based approaches. We also compare and contrast performance of BlueRay aspect for generalization and suppression. We compare algorithms based on the percentage of information preserved and time taken for sanitization.

7.1 Comparison of strict k-anonymization based algorithms

We start by comparing Mondrian k-anonymity algorithm with LSH bucketing based k-anonymity algorithm suggested in Section 4.4. Results from Figure 7-1 were obtained by executing both the algorithms on a single Apache Spark node with 8 cores, on an original dataset of size 32K. For LSH, we executed simulation for each of the value of k and found out the optimal value of a number of hash functions as well as the precision factor. We can see that % information preserved in Mondrian is better than LSH bucketing; however, the time taken by LSH is less than one hundredth of the time taken by LSH. For big data, Mondrian algorithm does not scale despite preserving a higher amount of original information. However, the amount of information preserved is low with LSH bucketing but only for a smaller population. We executed the same algorithm on a randomly generated adult dataset of size 50 million, with a precision factor of 10000 and three hash functions, and got result preserving ~87% of the original information.



(a) One pass LSH vs Mondrian: *Time taken vs cluster size*



(b) LSH vs Mondrian: percent information preserved

Figure 7-1. Mondrian k-anonymity vs. LSH bucketing based k-anonymity Model

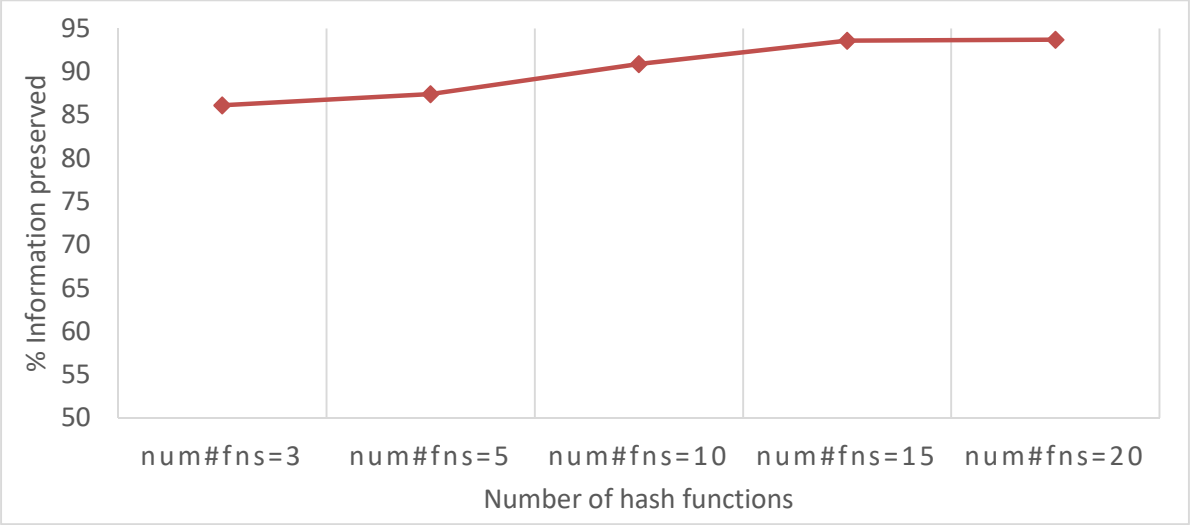
This means that as the size of dataset increases, LSH tends to preserve more information, thus standing out as a clear winner.

We can clearly see in the performance graph that LSH based k-anonymization performance is far superior to Mondrian k-anonymization. This happens because LSH uses the bucketing technique to find out the close points whereas Mondrian algorithm relies upon the "cut"

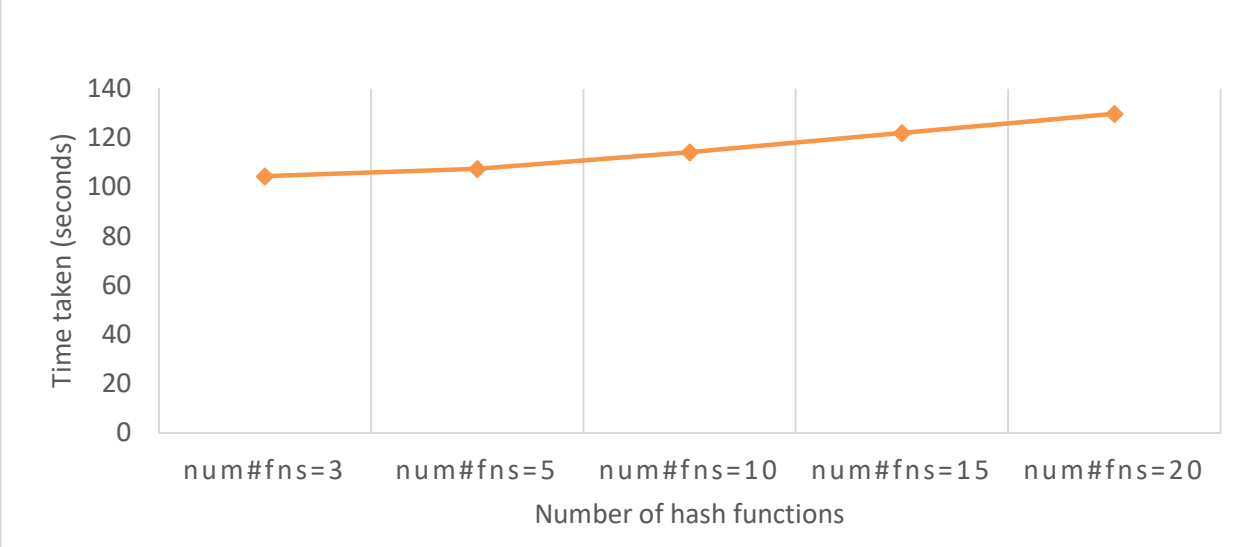
to decide which points are near to each other. The cut leads to the selection of maximum range but does not give any preference to the selection of points within the cluster. We also note that the value of k does not have much impact on LSH strict k-anonymity algorithm's time taken because of its constant time complexity. This is because the value of k simply decides the necessary cluster size for summarization; rest of the elements are simply suppressed. However, Mondrian algorithm time complexity is directly dependent upon the value of k. The height of the tree of stages to be cut is $\log_k(N)$. This means that as the value of k decreases, the depth of the tree increases. Higher depth leads to more number of jobs being formed hence leading to higher time complexity. As Mondrian is iterative in nature, the DAG formed in LSH k-anonymity algorithm is much simpler and smaller than the one formed in Mondrian algorithm.

7.2 Effect of number of hash functions on strict LSH k-anonymity algorithm

In order to check the impact of a number of hash functions on performance, we conducted an experiment on dataset described in section 7.3, on the entire cluster. In Figure 7-2, we see that the effect of increasing number of hash functions remains more or less constant beyond five. When we use three hash functions, although it takes lesser time, the information preserved is lesser. This is because the concatenated bucket id is composed of less number of bucket ids. However, when we increase the number of hash functions, the overall information preserved increases. This is because using a higher number of hash functions leads to more specific concatenated bucket id. Since evaluation was done on a huge dataset, most of the buckets had more than "k" elements leading to a good amount of information being preserved. We also note that as number of hash functions increase, overall time taken by the algorithm increases too. We must select a number of hash functions to be used after doing time vs information preserved tradeoff analysis.



(a) Percent information preserved vs number of hash functions



(b) Time taken vs bucket precision

Figure 7-2. Effect of number of hash functions on LSH algorithm

7.3 Effect of bucket precision on strict LSH k-anonymity algorithm

In order to understand the value that must be specified for bucket precision, we ran an experiment. We created a random adult dataset with 4 attributes, of size 50 million and tried different values

for precision. In order to perform the experiment, we set the value of a number of hash functions as 3 and value of k as 3. As we can see in Figure 7-3, the optimal percent information preserved occurs when we preserve the higher number of digits in the bucket id. When we see following diagram we clearly see that the value of information preserved is lowest for one digit, this is because the hash function used is based on normal distribution(0,1) and dataset used just has 4 fields with values normalized in the range of (0.0,1.0). However, as we go beyond 4 decimal places, more and more unique hashes get generated leading to more buckets. This graph was generated on a big dataset. For smaller datasets, the percent information preserved does not increase with precision; in fact, it decreases because most of the buckets contain less than k elements leading to more suppression and hence higher data loss. The analysis must be performed on the data in order to decide the bucket id precision to be set for the algorithm.

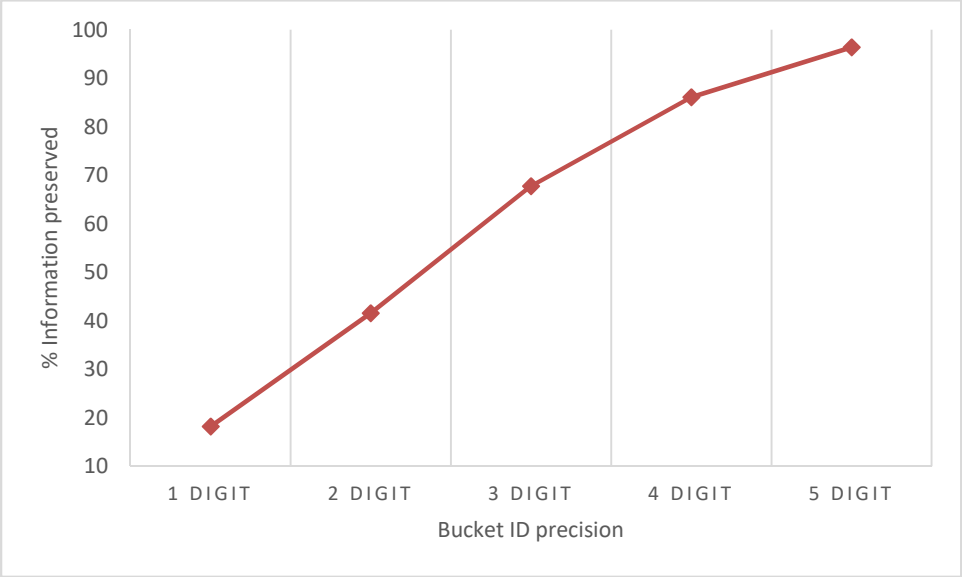


Figure 7-3. Percentage information preserved for different bucket precisions

7.4 Effect of k on strict LSH k-anonymity algorithm

In order to understand the effect of cluster size on information preserved by LSH k-anonymity algorithm, we used the same dataset as the one described in section 7.3. We set precision factor as 10000, number of hash functions as 3, and for different cluster sizes, we executed LSH. As we can see in Figure 7-4, the percentage information preserved reduces as cluster size increases. We see that for a huge dataset of 55 million, the cluster size of 80 works just fine but if we go beyond that, we see a sharp decline in the amount of information preserved.

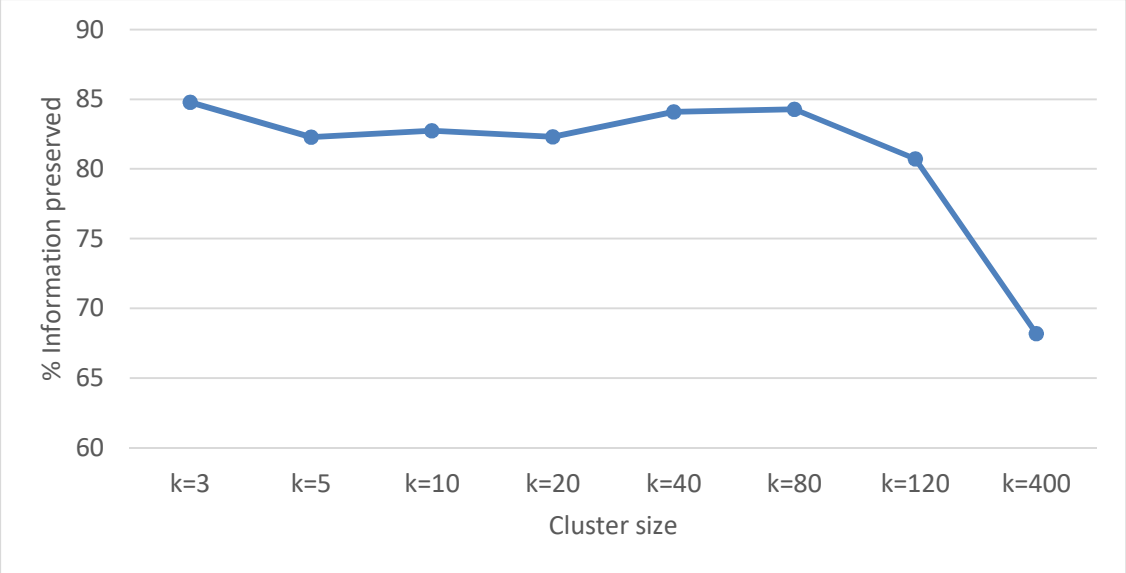
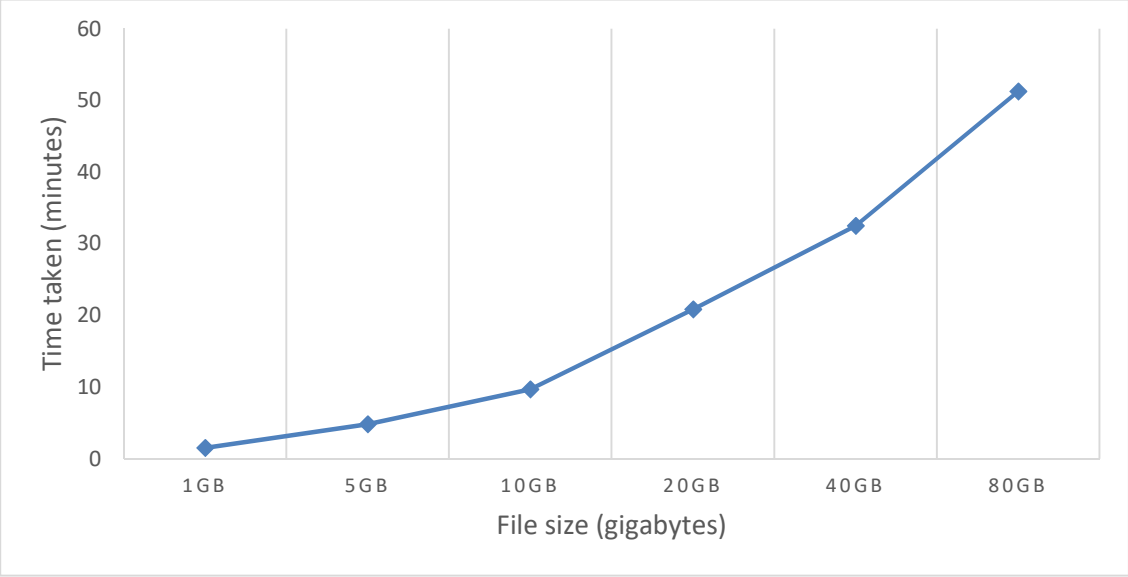


Figure 7-4. Percentage information preserved for different cluster sizes

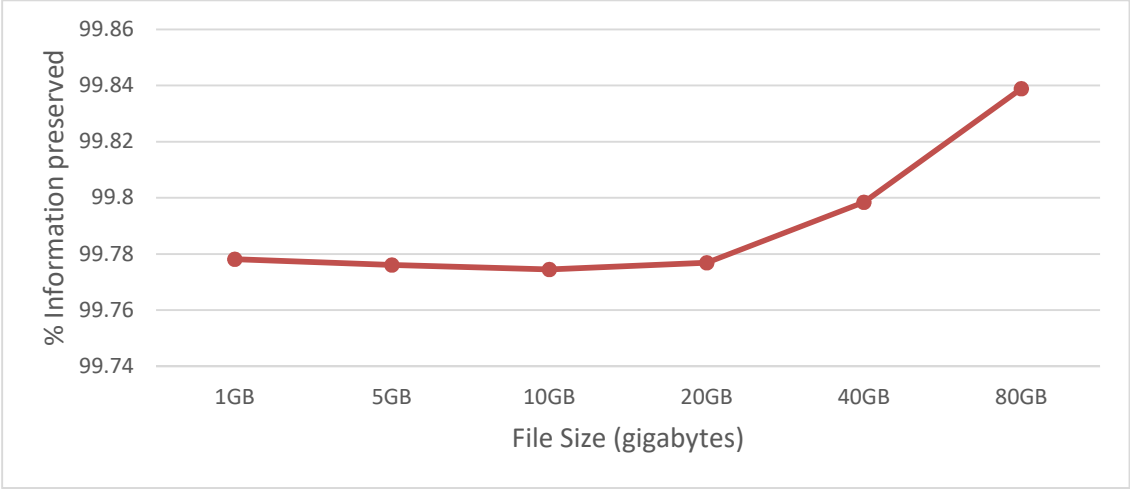
This happens because fewer clusters qualify for summary statistic generalization and all remaining clusters get completely suppressed, hence leading to higher information loss. The value of k must be chosen based on the size of the data.

7.5 Performance of risk-based LBS algorithm

Distributed LBS algorithm involves very few stages leading to a very simple DAG. DAG contains 5 jobs, first for reading data, second for executing LBS algorithm on each entry, third for calculation of mean of the publisher benefit, fourth for calculation of mean of the advisory benefit and fifth for writing output to a file. This leads to a very simple code execution flow requiring only single caching. Caching leads to RDD getting replicated on multiple nodes, hence leading to efficient computations even in the event of executor failure. In Figure 7-5, we can see that performance of LBS algorithm is linear. In order to perform this experiment, we scaled the LBS parameters from the paper (Wan, et al., 2015) to $1/12^{Th}$ of its value (i.e. V=100, L=8.333, C=0.333) and adjusted the number of partitions (1600 partitions for the 1GB file) as dataset size increased.



(a) LBS: Time taken vs file size



(b) LBS: percent information preserved vs file size

Figure 7-5. Performance of distributed LBS algorithm

Despite the LBS algorithm being compute intensive, it scales linearly. The distributed LBS algorithm executes LBS algorithm for each record concurrently on all executors. This leads to a high level of parallelism clearly reflecting in Figure 7-5. Also, we can clearly see that LBS algorithm preserves the high amount of information, ~99% as opposed to Mondrian and LSH bucketing based k-anonymity.

7.6 Comparison of data sanitization techniques

In order to compare how much of the original information is preserved after the anonymization process, we conducted a simple experiment on seven large files created by replicating original dataset. While executing these algorithms, we kept the same configuration for all algorithms. The value of k was set as 3 and the precision factor was set to 10000. The performance of LBS, LSH-k-anonymity, and LBS-LSH is as shown in Figure 7-6. We can clearly that LBS is a clear winner. It takes minimum time and overall, as dataset increases, the time increase remains within

proportion. In strict k-anonymity area, LSH based k-anonymity performs best. We executed Mondrian algorithm on the smallest dataset, and after an hour when it did not finish, we decided to end the program.

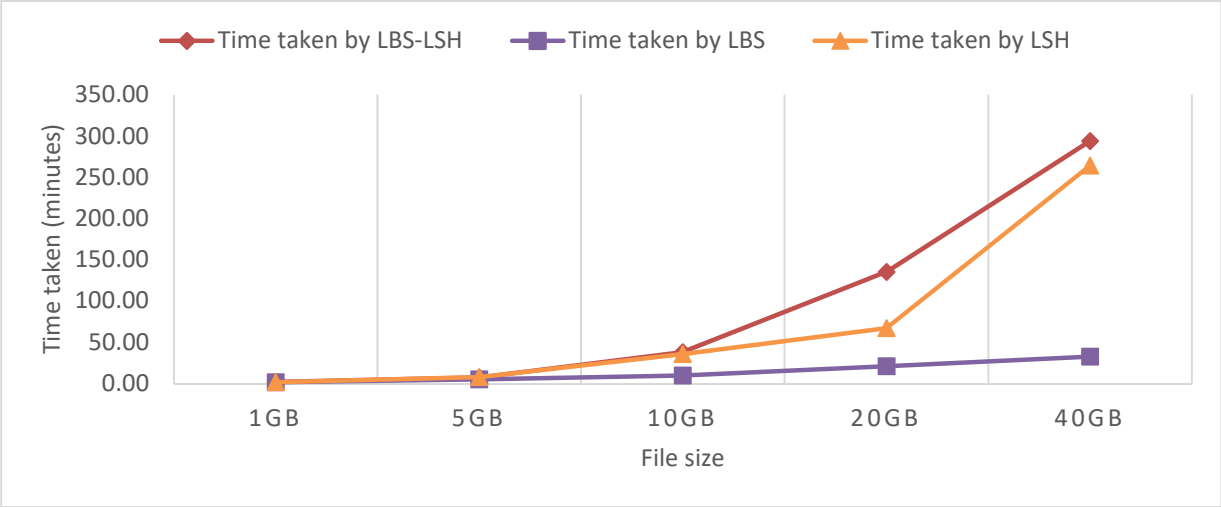


Figure 7-6. Time taken by LBS vs. LSH Vs one pass LBS-LSH anonymization techniques

We tried scaling all three algorithms to sanitize a file of size 80 GB. LBS finished in less than an hour, LSH finished in 9 hours whereas LBS-LSH failed to finish. LBS-LSH incurs the overhead of both the algorithms, leading to higher computation time. Space requirements of LBS-LSH are also higher; it needs space for caching buckets as well as space for caching intermediary RDD that gets computed thrice if not cached. This leads to extremely high memory requirements and LBS-LSH starts suffering the moment file size increases beyond 10 GB. This is because data does not fit in memory anymore and Apache Spark has to perform a lot of shuffle write operations. LSH takes a higher amount of time than LBS because of bucket grouping. Grouping of data leads to a lot of data shuffling leading to the high network I/O. Due to this reason, the computation time for LSH increases as data size increases.

Although it is not pictorially shown, LBS and LBS-LSH algorithms provide close to 99.72% information preservation, and LSH based k-anonymity preserved ~87% of the original information. Since datasets were created by replication of original data set, 100% information preservation is possible because of the existence of duplicates. LBS, as well as LBS-LSH, preserve most of the information. However, LSH bucketing based k-anonymity algorithm heavily depends on the master "concatenatedHash" key, in order to group, and this hash key is computed using floating point operations like division. Since the result of the floating point operation can differ slightly across machines, the algorithm suffers, leading to variation in concatenatedHash generated on different machines for the very same record.

7.7 Experimental evaluation of on-the-fly sanitization approach

In this section, we describe and compare the performance of on-the-fly sanitization technique described in Chapter 6. Since the technique was suggested for on-the-fly sanitization, we performed an experiment in order to analyze and report the impact of the aspect.

7.7.1 Experiment Setup

We used latest stable Apache Spark version available while performing the experiment. Please find configuration details below.

Number of executors: 18

Worker RAM: 10GB each

Apache Spark version: 2.1.0

Apache Spark add-ons:

- BlueRay aspect jar stored in <Apache Spark_Home>/jars

- Aspectjweaver jar – version 1.8.5, stored in /data/blueray folder
- Local policy store - policies.csv, stored in /data/blueray folder

Apart from Apache Spark, we also host Apache Tomcat server version 8.0 and deploy BlueRay web application, which serves as centralized RESTful policy manager. The BlueRay aspect communicates with RESTful policy manager using JSON data interchange format.

7.7.2 Comparisons: web policy manager vs local policy manager:

In order to compare the performance of web policy manager with local policy manager, we executed a simple count program that simply iterated over entire dataset once. Figure 7-7 shows the performance difference between local policy store and centralized web policy manager. We can clearly see that for small files, the performance of both the approaches is almost identical. This is because we specified fixed number of executors throughout, approximately 80. This means that the local policy store was read exactly eighty times, once by each executor. The time spent in reading local policy store was negligible, and this lead to the almost identical performance of both approaches.

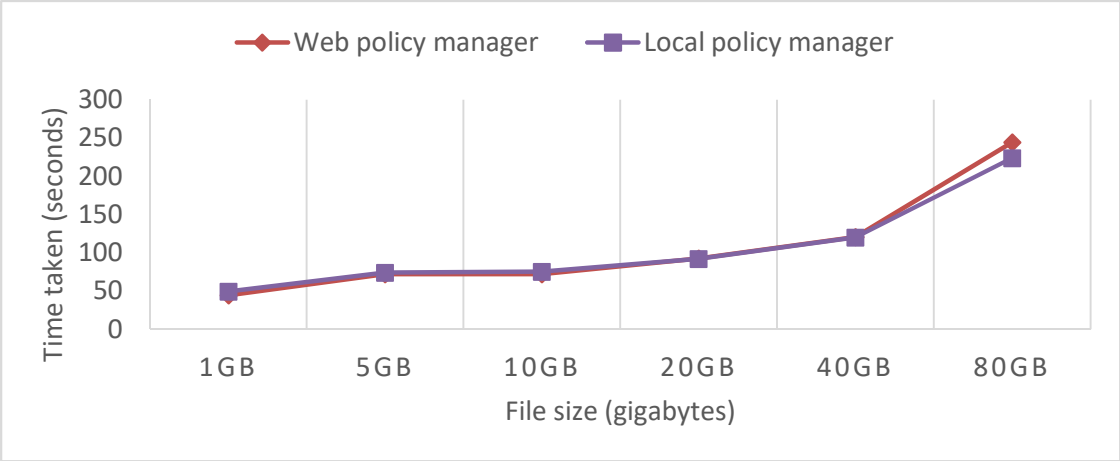


Figure 7-7. Local vs. web policy manager performance

As data size increases, Apache Tomcat's latency increases because of limitation of the load it can handle, which clearly explains why local policy performs slightly better for higher data loads. The policy store selection depends upon requirement too. For huge policy stores, it would be wise to have a web-based RESTful policy manager instead of local one. RESTful policy manager can be further load balanced to support a higher number of concurrent HTTP GET requests.

7.7.3 Performance comparisons: read, group by, read-write:

In order to analyze the impact of aspect on overall processing, we performed a simple experiment. We wrote small scripts that simply did read/write/group by operations. We executed BlueRay aspect on each of the files. We can see in Figure 7-8 that read query performance curves are parallel for sanitized and non-sanitized executions, and so are save functionality curves. However, group by internally can lead to several calls of read method leading to a higher performance overhead.

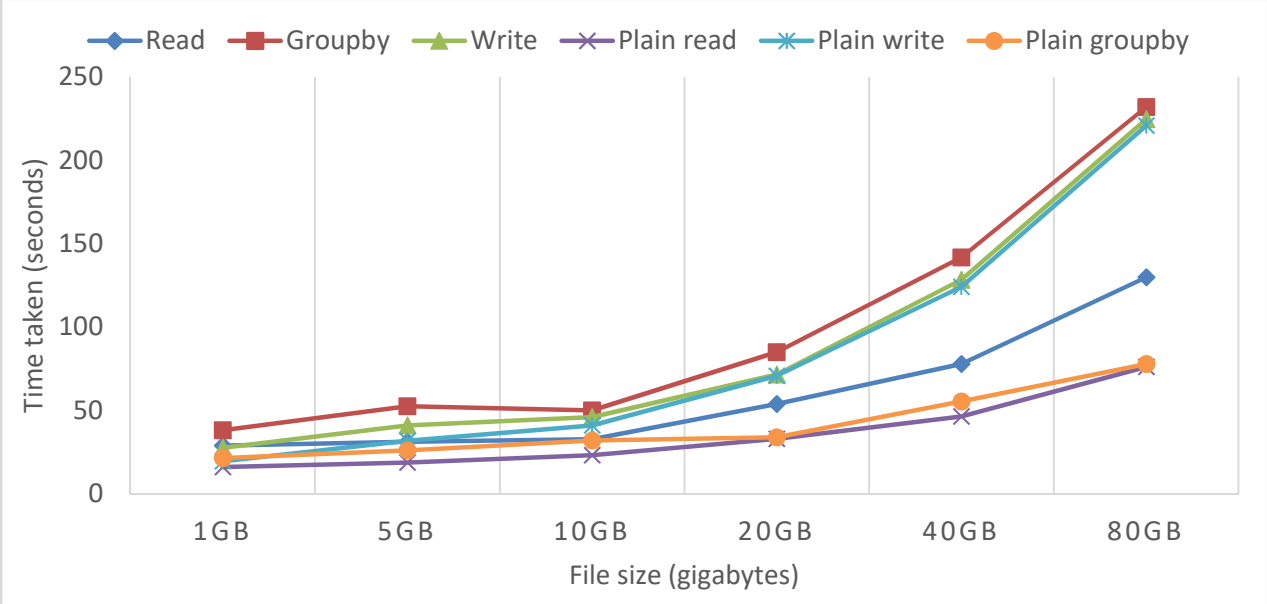


Figure 7-8. Sanitized vs plain functionality: Read vs. group by vs. write queries

We see that overhead caused by aspect scales linearly as data increases exponentially. This means that BlueRay aspect scales linearly even with exponentially growing data. In order to calculate sanitization overhead, we came up with the following formula.

$$\text{Overhead} = \frac{\text{Time taken for sanitization} - \text{Time taken without sanitization}}{\text{Time taken without sanitization}} \quad (7.1)$$

We can see in Figure 7-9 that for computational queries like write, as time passes, the overhead reduces whereas, for queries like read, it remains constant. Group by seems to be going up until we exhaust complete memory of all executors. This is mainly because, in a write operation, the cost of I/O dominates whereas in the group by operation, the cost of iteration dominates.

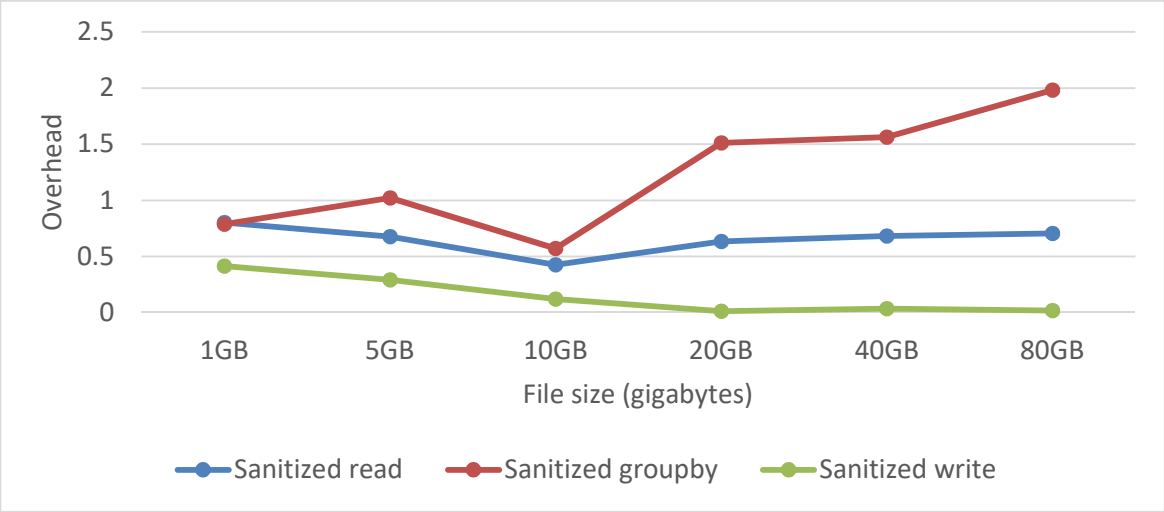


Figure 7-9. Overhead graph read vs. group by vs. write queries

Another reason behind this anomaly is because of the nature of Apache Spark platform itself. Apache Spark computes RDD for every action. If we are invoking two actions on single RDD, it would compute the RDD twice. This part of the framework definitely affects the performance. In order to remedy this Apache Spark suggests the use of the ***cache*** and ***unpersist***

function calls. However, incorrect use of these calls leads to values getting cached in memory on worker nodes, which may result in incorrect output.

7.7.4 Performance impact of generalization vs suppression

Graph in Figure 7-10 was created by generalizing and suppressing two of the quasi-identifier attributes, namely zip code, and age. We used performance of count method in order to accurately plot the cost of reading generalized or suppressed data. The process of variable suppression is achieved by reading data at a particular index through the BlueRay aspect and converting it into suppressed form. The process of generalization is performed by replacing the value with its parent. Although generalization involves lookup of the parent category, it can be easily fixed by using a hash map as the cache for attribute and their generalizations.

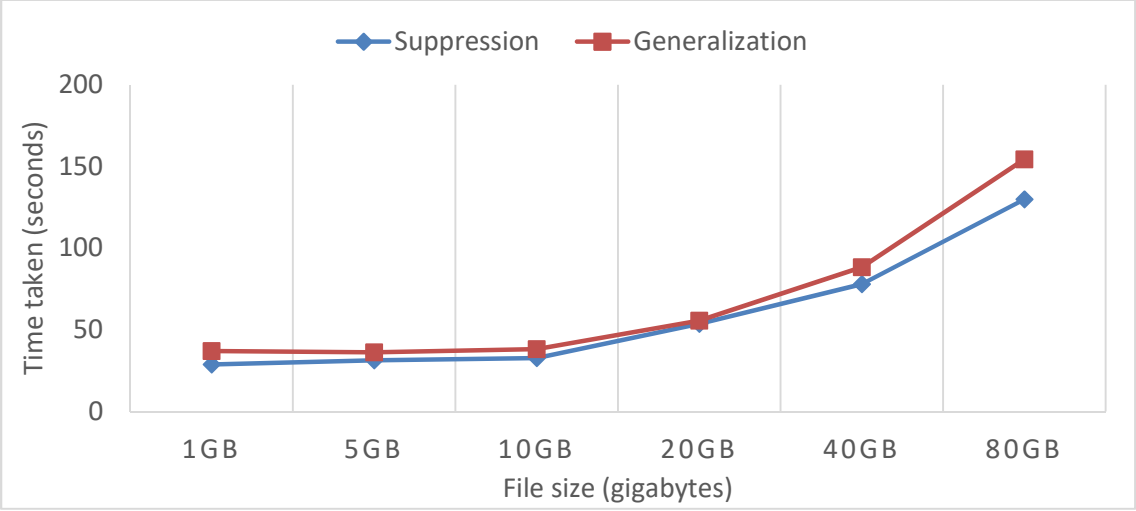


Figure 7-10. Performance of generalization vs. suppression

We can clearly see in above graph that generalization takes higher time as compared to suppression. Despite using a hash map, the diversity in the zip code causes cache misses leading to higher time consumption compared to suppression. We also notice that both the graphs start at

a high value because of the initial aspect setup cost and gradually go up as the size of the dataset increases. The overhead graph for generalization as well as suppression can be obtained using following formula.

$$\text{Overhead} = \frac{\text{Time taken for generalization OR suppression} - \text{Time taken for iteration}}{\text{Time taken for iteration}} \quad (7.2)$$

From overhead graph in Figure 7-11, we can see that generalization and suppression both have maximum overhead for smallest dataset. This is because the setup cost of injecting aspect itself and reading metadata, in the case of generalization, adds up. The impact of these additional costs decreases as we increase dataset size and the cost of BlueRay aspect starts adding up. We can see that the least overhead occurs for a file of size 10 gigabytes.

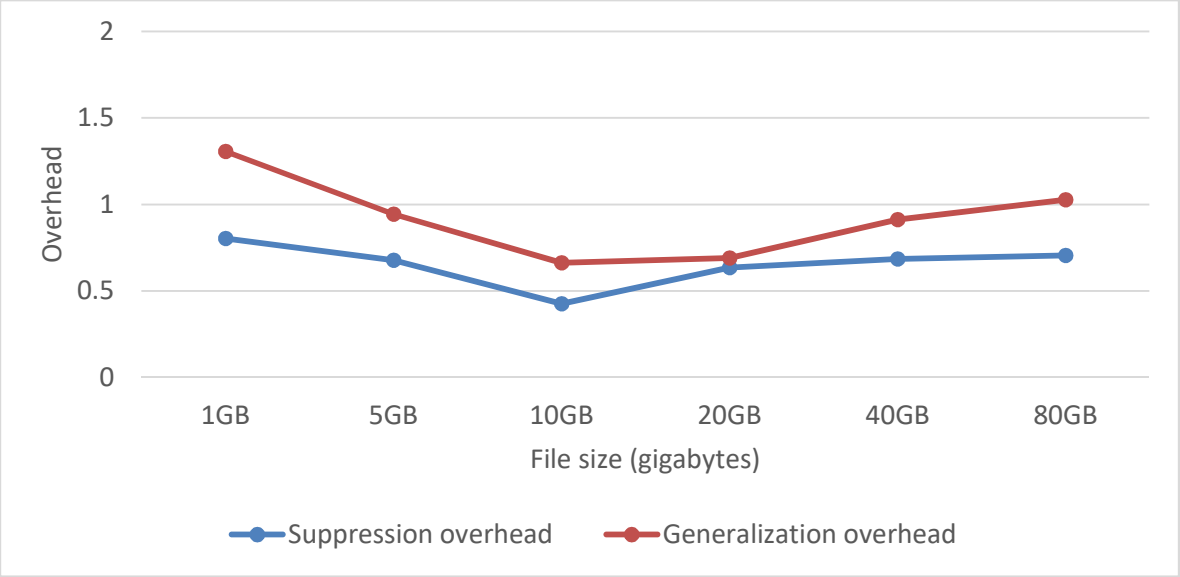


Figure 7-11. Overhead graph for generalization vs. suppression

CHAPTER 8

CONCLUSION AND FUTURE WORK

Distributed algorithms do better than centralized algorithms, and thus are a better fit for scalable big data anonymization. That being said, how distributed framework itself is architected has a direct correlation with how a particular algorithm is going to perform. Thus, it is important to select the right platform. Since most anonymization algorithms are typically iterative in nature, they demand big data solution tuned for in-memory, iterative requirements. When compared to Hadoop, Apache Spark provides additional support for iterative environments through data caching and thus are more suitable for data sanitization implementations. In order to enable big data sanitization, we implemented risk-based, strict k-anonymity based, batch, as well as online sanitization algorithms for distributed frameworks, and then compared and contrasted their performances.

In online big data sanitization space, the key contribution of this thesis is the BlueRay aspect, a non-intrusive aspect-based approach for modifying runtime behavior of Apache Spark RDD computation. To our knowledge, this is the first study that uses aspect-oriented programming on Apache Spark in order to perform online data sanitization. Although, as part of this study, we considered two utilities of the BlueRay aspect, namely, attribute suppression or generalization in a structured context, the model has several more uses. In future, the BlueRay aspect framework can be used to perform risk based sanitization, on-the-fly HDFS file encryption, role-based access control, on-the-fly encrypted HDFS file decryption, RDD computation logging, RDD performance metric analysis, etc. The approach does not require the developer to maintain a separate codebase for Apache Spark, thus removing headache associated with framework branch maintenance. The

BlueRay aspect framework can also be extended to other distributed in-memory frameworks like Impala, Apache Storm etc.

In the field of strict k-anonymity, the key contribution of this study is one pass distributed LSH bucketing based k-anonymity algorithm. The distributed LSH bucketing method uses locality sensitive hashing for performing bucketing in higher dimensional spaces. We further empirically show that one pass LSH with data normalization can yield results that are better than Mondrian k-anonymity. Not only it offers better scalability as well as efficiency compared to Mondrian algorithm, but it also provides superior original information preservation for larger datasets. Although it scales easily for datasets of size equal to cluster memory itself, improving the performance of LSH k-anonymity to scale beyond cluster memory capacity is a possible future work.

In risk-based modern data sanitization approaches, this thesis contributes LSH bucketing based LBS algorithm. Although the experimental section results show that LBS is superior to LBS-LSH when it comes to performance (because of smaller risk lookup cost), The LBS-LSH algorithm would definitely start performing better in two cases; when population lookups are extremely time-consuming, and when the population is too sparse leading to LBS exploring most of its lattice. In such scenarios, LBS-LSH may provide better performance.

While risk-based approach evaluates dataset against the population, k-anonymity based methods focus on attaining k-anonymization within the dataset itself. This is not always the requirement. A hybrid Mondrian in population k-anonymity algorithm can be formulated which would decide whether to select a given generalization level based on a number of individuals in the population. This approach would not only provide within population strict k-anonymity but

would also be more efficient because of its distributed nature. Apart from future work opportunities mentioned above, it would be worth comparing them with other clustering based models like Kmeans.

We evaluated all three batch algorithms for data set the size of approximately 2 billion rows, the combined population size of Europe and China, and showed how each of the algorithms can scale to meet such big data demands on a cluster of limited size. Optimizing performance of these algorithms to work beyond four attributes in limited time, on limited hardware is definitely something that can be addressed in future.

In summary, anonymization techniques need to scale to large data, and given advances in memory cost per unit reduction, in-memory distributed computing frameworks would be right tools for solving these problems. The decision of algorithm selection would simply remain to be a factor of the anonymization requirement. For strict k-anonymity, LSH bucketing can be used whereas for risk-based anonymization, LBS or LBS-LSH would be a perfect fit. The results reported in this thesis would enable data publishers to decide right tools and techniques for performing big data sanitization.

APPENDIX A

DATASET AND METADATA

In Appendix A, we describe the metadata used for performing experiments explained in CHAPTER 7.

Input dataset and metadata file format

All batch algorithms explained in this thesis expect input data as in the following format. Please find an example of the sample in Figure A-1.

```
Male,38019,19,White
Male,38114,19,Asian-Pac-Islander
Male,37887,18,Amer-Indian-Eskimo
Male,37212,18,Other
```

Figure A-1. Sample CSV file

Metadata file format

In order to process data efficiently, we also create a metadata file of the format shown in Figure A-2. This XML file describes the structure of the data. The root element of the XML is <columns>.

```
<columns>
    <column>
        <name>[column_name]</name>
        <type>[Numeric or Categorical]</type>
        <index>0</index>
        <num_unique>2</num_unique>
        <isQuasiIdentifier>true</isQuasiIdentifier>
        <min></min>
        <max></max>
        <hierarchy>
            <value>*</value> <!Implicitly assumed.-->
            <children>
                <value>[column_unique_value_2]</value>
            </children>
            <children>
                <value>[column_unique_value_1]</value>
            </children>
```

```
            </hierarchy>
        </column>
        <!-- ... Other columns   -->
</columns>
```

Figure A-2. Metadata file format

The <columns> element contains metadata for each column present in the CSV file. Along with other attributes, <column> element contains name or title of the column, index of the column, and type of the column. Type of the column is expected to be provided as "i" for numeric data and "s" for string data. For simplicity, the numeric data is assumed to be in form of double data type. It also contains a column indicating whether the column selected is a quasi-identifier or not. For numeric data, metadata is expected to include minimum and maximum value possible for data found in this column. There are no validations performed to enforce this, but we simply assume that data respects min-max range provided. Along with the above metadata, a generalization hierarchy is also provided for each column. Since granular generalization hierarchies play a major role in better anonymization of the data, we have defined granular hierarchies.

Metadata generalization hierarchy

Please find generalization hierarchy for the race attribute, denoted in the pictorial format in Figure A-3. Based on the data available, following generalization hierarchy was created for race attribute. Please note that all attributes have a common top most level of generalization, complete suppression.

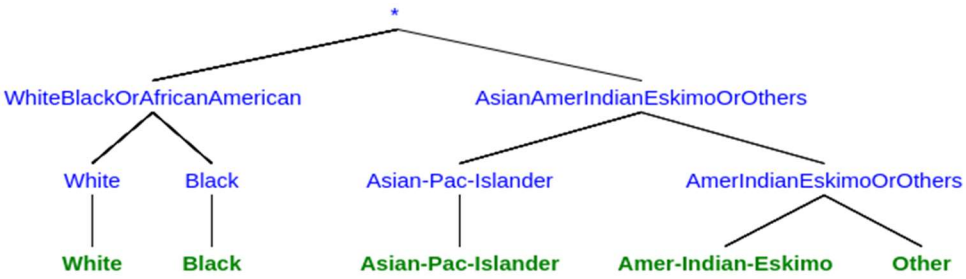


Figure A-3. Race generalization hierarchy

Similarly, for attribute age, we defined two high-level generalization hierarchies, 0-59 and 60-120, each of these levels are further subdivided into two equal portions until we reach a level where a number of entries become odd. At this level, we divide the range into 3 groups each and make it the leaf of the generalization hierarchy. Please find the same in Figure A-4

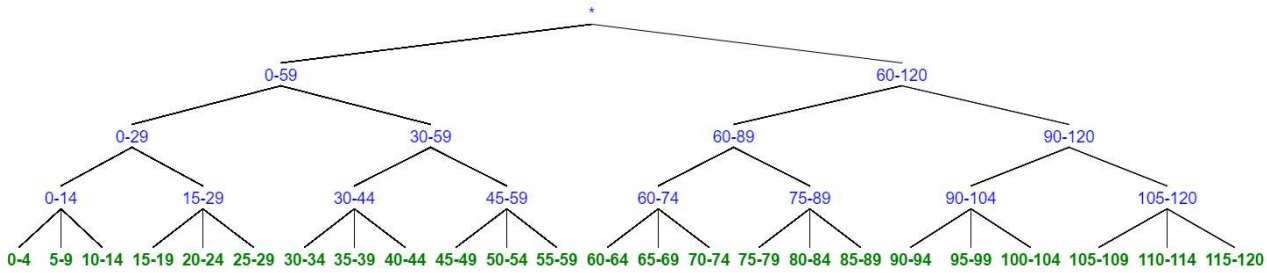


Figure A-4. Age generalization hierarchy

For gender attribute, a simple hierarchy was created. As shown in Figure A-5, both values can be generalized into a single common ancestor that suppresses the attribute completely.

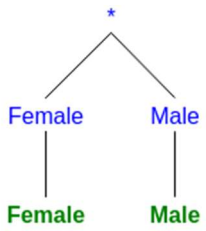


Figure A-5. Gender generalization hierarchy

For zip code, the data contained was found to be in the range of 37010-72338. The hierarchies defined for the zip code attribute are shown in Figure A-6. This was kept consistent with hierarchies used in the risk-based paper (Wan, et al., 2015).

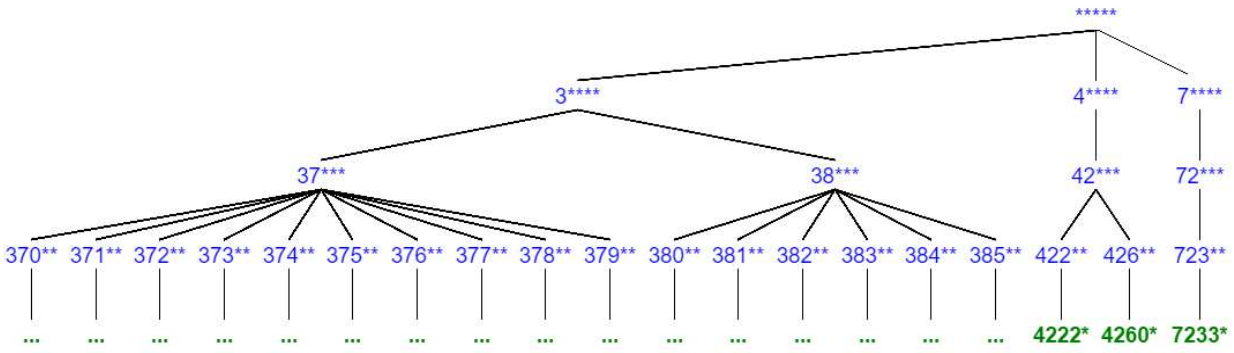


Figure A-6. Zip code generalization hierarchy (Top Level)

Because of the space constraint, the bottommost level was not included in Figure A-6. A sample generalization level for 370**, is expanded as shown in Figure A-7.

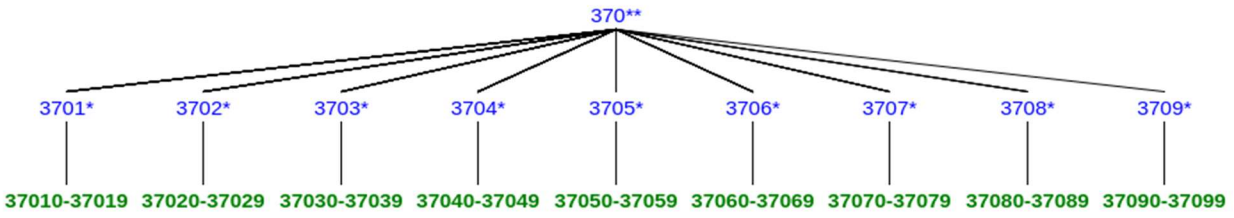


Figure A-7. Zip code generalization hierarchy (bottom level)

APPENDIX B

SOURCE CODE

This section includes source code implemented as part of the thesis. Shown below is the source code of BlueRay aspect, which is responsible for injection of aspect that performs data sanitization. The following code calls ***extractPathForSpark***, which is responsible for extraction of the name of the HDFS file on which aspect injection is being performed.

```
@Around(value = "execution(* org.apache.spark.rdd.MapPartitionsRDD.compute(..)) && args(theSplit,context)",
                    argNames = "jp,theSplit,context")
def aroundAdvice_spark(jp: ProceedingJoinPoint, theSplit: Partition, context: TaskContext): AnyRef = {

  val iterator = (jp.proceed(jp.getArgs()));
  if (sys.env.contains("BlockColumns")) {
    val blockCols = sys.env("BlockColumns");
    val metadataPath = blockCols.substring(blockCols.indexOf(']') + 1, blockCols.length());
    if (metadataPath != null && metadataPath.trim().length() > 3 && dataMetadata == null) {
      synchronized {
        if (dataMetadata == null) {
          val data = Source.fromFile(metadataPath).getLines().mkString("\n");
          if (data != null && data.trim().length() > 0) {
            dataMetadata = new DataReader().readMetadata(data);
          }
        }
      }
    }
    val columnBlockingIterator = new ColumnBlockingInterruptibleIterator(context,
                iterator.asInstanceOf[Iterator[_]], sys.env("BlockColumns"), dataMetadata);
    println("Returning   ColumnBlockingInterruptibleIterator : " + columnBlockingIterator)
    return columnBlockingIterator;
  }
  val policy = getPolicy(context, jp, PointCutType.SPARK);
  if (policy != None) {
    val authorizedIterator = new AuthorizedInterruptibleIterator(context,
                                    iterator.asInstanceOf[Iterator[_]], policy.get.regex);
    println("Returning new iterator")
    return authorizedIterator
  }
  return iterator
}
```

As we can see in the following code, we loop over fields of the joint point and extract the one which is either InputSplit or split, we then extract the entire path of the same.

```
def extractPathForSpark(jp: org.aspectj.lang.ProceedingJoinPoint): String = {
  var pathFound = false;
  var path: String = "";
  breakable {
    for (argument <- jp.getArgs()) {
      for (field <- argument.getClass.getDeclaredFields) {
        if (field.getName.equalsIgnoreCase("inputSplit") || field.getName.equalsIgnoreCase("split")) {
          field.setAccessible(true)
          val fullPath = field.get(jp.getArgs()(0)).toString()
          path = fullPath.subSequence(0, fullPath.lastIndexOf(":")).toString()
          pathFound = true;
          break;
        }
      }
      if (pathFound) {
        break;
      }
    }
  }
  path
}
```

The aspect chooses different iterators based on the requirement. Shown below is the declaration of one the iterators. Specifically, the ***ColumnBlockingInteruptibleIterator***, which blocks or generalizes columns provided. This iterator extends the ***InterruptibleIterator*** that is returned as the output of MapPartitionRDD's ***compute*** method.

```
class ColumnBlockingInterruptibleIterator[I]
      (context: TaskContext, delegate: Iterator[I], val columnsToBeBlocked: String, val dataMetadata: Metadata)
      extends InterruptibleIterator[I](context, delegate) {
```

We do not modify ***hasNext*** method but we do override ***next*** method. As we can see, the algorithm blocks column specified if the ***nextElement***'s length matches length specified.

```
override def next(): T = {
  val nextElement = super.next();
  if (nextElement != null) {
    var localNextElementStr = "";
    if (nextElement.getClass == classOf[UnsafeRow]) {
      var objectVal: Array[Byte] = nextElement.asInstanceOf[UnsafeRow].getBytes.asInstanceOf[Array[Byte]];
      for (c <- objectVal) {
        localNextElementStr += (c.toChar);
      }
    } else {
      localNextElementStr = nextElement.toString();
    }
    var split = localNextElementStr.trim().split(",");
    if (split.length == numColumns) {
      for (i <- blockCols) {
        split(i) = getStringOfLength(i, split(i));
      }
      if (nextElement.getClass == classOf[String]) {
        return split.mkString(",").asInstanceOf[T];
      } else if (nextElement.getClass == classOf[UnsafeRow]) {
        val unsafeRow: UnsafeRow = nextElement.asInstanceOf[UnsafeRow];
        var newElement: UnsafeRow = new UnsafeRow(unsafeRow.numFields());
        localNextElementStr = localNextElementStr.trim().r.replaceAllIn(localNextElementStr,
                              split.mkString(","));
        newElement.pointTo(localNextElementStr.map(_.toByte).toArray, unsafeRow.getBaseOffset,
                           unsafeRow.getSizeInBytes)
        return newElement.asInstanceOf[T];
      } else {
        return nextElement;
      }
    } else {
      return nextElement;
    }
  } else {
    return nextElement;
  }
}
```

In order to perform suppression or generalization, we use the following code.

```
def getStringOfLength(index: Int, value: String): String = {
  if (dataMetadata == null) {
    var sb: StringBuilder = new StringBuilder();
    for (c <- 1 to value.length()) {
      sb.append("-");
    }
    sb.toString;
  } else {
    return dataMetadata.getMetadata(index).get.getParentCategory(value).value();
  }
}
```

LBS source code for implementing risk-based approach implementation

Please find the implementation of the ***findOptimalStrategy*** method described in section 5.3, below. This method implements the risk-based sanitization approach paper (Wan, et al., 2015).

```
def findOriginalOptimalStrategy(top: scala.collection.mutable.Map[Int, String]):
(Double, Double, scala.collection.mutable.Map[Int, String]) = {

  var Um: Double = -1;
  var LPiG: Double = -1;

  var g = top;
  LPiG = L * Pi(g);
  var Gm = g;
  while (!isGLeafNode(g)) {

    if (LPiG <= C) {
      return (Vg(g), 0.0, g);
    }
    Um = Vg(g) - LPiG;
    Gm = g;
    val maxChildren = getChildren(g).map(Gc => {
      var LPiGc = L * Pi(Gc);
      if (LPiGc > C) {
        ((Vg(Gc) - LPiGc), LPiGc, Gc);
      } else {
        (Vg(Gc), 0.0, Gc);
      }
    }).filter(_._1 >= Um);

    if (maxChildren.isEmpty) {
      return (Um, LPiG, g);
    } else {
      val child = maxChildren.maxBy(_._1)
      Um = child._1
      LPiG = child._2
      Gm = child._3
    }
    g = Gm;
  }
  return (Um, LPiG, g);
}
```

The implementation of IL(g) method that calculates information loss for a generalized record, is shown below. This method computes information loss for each attribute individually and then adds it in order to compute total information loss for generalization level g.

```
def IL(g: scala.collection.mutable.Map[Int, String]): Double =
  {
    var infoLoss: Double = 0;

    for (column <- metadata.getQuasiColumns()) {
      var count: Long = 0;
      val value = g.get(column.getIndex()).get.trim()
      if (column.getColType() == 's') {
        val children = column.getCategory(value);
        if (children.leaves.length != 0) {
          infoLoss += (-Math.log10(1.0 / children.leaves.length));
        }
      } else {
        val minMax = LSHUtil.getMinMax(value);
        if (column.getName().trim().equalsIgnoreCase("age")) {
          infoLoss += (-Math.log10(1.0 / (1 + minMax._2 - minMax._1)));
        } else {
          val zipInfoLoss = LBSMetadata.getZip().filter { x => x >= minMax._1 && x <= minMax._2 }.size;
          infoLoss += (-Math.log10(1.0 / (zipInfoLoss)));
        }

      }
    }
    return infoLoss;
  }
```

Shown below is the source code of ***getChildren*** method used for creating a lattice. This method is called by the ***findOptimalStrategy*** method in order to form a lattice to compute generalization on.

```
/**
 * This method returns the list of immediate children from lattice for the given entry.
 */
def getChildren(g: scala.collection.mutable.Map[Int, String]): List[scala.collection.mutable.Map[Int, String]] =
  {
    /**
     * Iterate over each attribute, generalize the value one step up at a time, accumulate and return the list.
     */
    val list = ListBuffer[scala.collection.mutable.Map[Int, String]]();
    /**
     * Create child for lattice on each column one at a time.
     */
    for (column <- metadata.getQuasiColumns()) {
      var copyOfG = g.clone();
      val value = g.get(column.getIndex()).get.trim()
      val parent = column.getParentCategory(value).value();
      if (parent != value) {
        copyOfG.put(column.getIndex(), column.getParentCategory(value).value().trim());
        list += copyOfG;
      }
    }
    return list.toList;
  }
```

Following method is used in order to check whether lattice based search has reached the bottom of the lattice, i.e. every attributed is generalized to its highest possible value.

```
def isGLeafNode(map: scala.collection.mutable.Map[Int, String]): Boolean =
  {
    for (column <- metadata.getQuasiColumns()) {
      val value = map.get(column.getIndex()).get.trim()
      if (!value.equalsIgnoreCase("*")) {
        if (column.getColType() == 's') {
          if (!value.equalsIgnoreCase(column.getRootCategory().value())) {
            return false;
          }
        } else {
          if (value.contains("_")) {
            val range = value.split("_");
            if (!(range(0).toDouble == column.getMin() && (range(1).toDouble == column.getMax()))) {
              return false;
            }
          } else {
            return false;
          }
        }
      }
    }
    return true;
  }
```

Shown below is the source code of metadata class that is used for expediting the execution of LSH/Mondrian/LBS algorithms.

```
class Metadata(columnMetadata: Map[Int, Column]) extends Serializable {
  var columns: Array[Column] = null;
  var maximumInfoLoss: Double = 0;

  def getMetadata(columnId: Int): Option[Column] = {
    return columnMetadata.get(columnId);
  }
  def numColumns(): Int = {
    return columnMetadata.size;
  }
  def getMaximulInformationLoss(): Double =
    {
      if (maximumInfoLoss == 0) {
        for (column <- getQuasiColumns()) {
          maximumInfoLoss += (-Math.log10(1.0 / column.getNumUnique()));
        }
      }
      return maximumInfoLoss;
    }
  def getQuasiColumns(): Array[Column] = {
    if (columns == null) {
      var localColumns = ListBuffer[Column]();
      for (i <- 0 to numColumns() - 1) {
        val column = getMetadata(i).get;
        if (column.getIsQuasiIdentifier()) {
          localColumns += column;
        }
      }
      columns = localColumns.toArray;
    }
    return columns
  }
  override def toString: String = {
    return columnMetadata.mkString;
  }
}
```

Mondrian source code

Please find the source code for distributed Mondrian algorithm described in section 4.3, below.

```
def kanonymize(linesRDD: RDD[(Long, Map[Int, String])], k: Int, metadata: Broadcast[Metadata]) {
  linesRDD.cache();
  var leftRDD: RDD[(Long, Map[Int, String])] = null;
  var rightRDD: RDD[(Long, Map[Int, String])] = null;
  var leftPartitionedRange: String = null;
  var rightPartitionedRange: String = null;
  var dimAndMedian: Dimensions = selectDimension(linesRDD, k);
  if (dimAndMedian.dimension() >= 0) {
    if (metadata.value.getMetadata(dimAndMedian.dimension()).get.isCharColumn()) {
      leftRDD = linesRDD.filter({
        case (x, y) =>
          { dimAndMedian.leftSet().contains(y.get(dimAndMedian.dimension()).get) }
      });
      rightRDD = linesRDD.filter({
        case (x, y) =>
          { dimAndMedian.rightSet().contains(y.get(dimAndMedian.dimension()).get) }
      });
    } else {
      leftRDD = linesRDD.filter({ case (x, y) =>
                      y.get(dimAndMedian.dimension()).get.toDouble <= dimAndMedian.median().toDouble });
      rightRDD = linesRDD.filter({ case (x, y) =>
                      y.get(dimAndMedian.dimension()).get.toDouble > dimAndMedian.median().toDouble });
    }
    var leftSize = leftRDD.count();
    var rightSize = rightRDD.count();
    linesRDD.unpersist(true);
    if (leftSize >= k && rightSize >= k) {
      if (leftSize == k) {
        assignSummaryStatisticAndAddToList(leftRDD, metadata);
      } else {
        kanonymize(leftRDD, k, metadata);
      }
      if (rightSize == k) {
        assignSummaryStatisticAndAddToList(rightRDD, metadata);
      } else {
        kanonymize(rightRDD, k, metadata);
```

Please find the source code of ***selectDimension*** method called by the ***k-anonymize*** method, below.

```
def selectDimension(linesRDD: RDD[(Long, Map[Int, String])], k: Int): Dimensions = {
  val metadata = LBSMetadata.getInstance();
  val indexValuePairs = linesRDD.flatMap({ case (index, map) => (map) });
  val indexValueGrouped = indexValuePairs.groupByKey()
    .map({ case (index, list) => (index, list.toList) });
  indexValueGrouped.cache();
  val indexAndCount = indexValueGrouped.map({ case (index, list) => (index, list.distinct.size) })
  val sortedIndexAndCount = indexAndCount.sortBy(_._2, false);
  val dimToBeReturned: Int = sortedIndexAndCount.first()._1;
  if (metadata.getMetadata(dimToBeReturned).get.isCharColumn()) {
    val sortedListOfValues = indexValueGrouped.filter(_._1 == dimToBeReturned)
      .flatMap({ case (x, y) => (y) }).map(x => (x, 1))
      .reduceByKey((a, b) => a + b).sortByKey(true).collect();
    val total = sortedListOfValues.map(_._2).sum;
    var runningSum = 0;
    var leftList = List[String]();
    var rightList = List[String]();
    for (pair <- sortedListOfValues) {
      if (runningSum <= total / 2) {
        leftList = leftList.::(pair._1);
        runningSum = runningSum + pair._2
      } else
        rightList = rightList.::(pair._1);
    }
    indexValueGrouped.unpersist(true);
    return new Dimensions(dimToBeReturned, 0, 0, 0, leftList.toArray, rightList.toArray);
  } else {
    val sortedListOfValues = indexValueGrouped.filter(_._1 == dimToBeReturned)
      .flatMap({ case (x, y) => (y) }).sortBy(x => x.toDouble).zipWithIndex();
    val reverseIndex = sortedListOfValues.map({ case (x, y) => (y, x) });
    val min = reverseIndex.lookup(0)(0).toDouble;
    val median = reverseIndex.lookup((sortedListOfValues.count() / 2))(0).toDouble;
    val max = reverseIndex.lookup(sortedListOfValues.count() - 1)(0).toDouble;
    indexValueGrouped.unpersist(true);
    return new Dimensions(dimToBeReturned, min, median, max, null, null);
```

LBS

Please find the source code for distributed LBS algorithm explained in section 5.3, below.

```
val t0 = System.nanoTime()
val metadata = LBSMetadataWithSparkContext.getInstance(sc);
val zips = LBSMetadataWithSparkContext.getZip(sc);
val population = LBSMetadataWithSparkContext.getPopulation(sc);
val hashedPopulation = LBSMetadataWithSparkContext.getHashedPopulation(sc);
val params = sc.broadcast(lbsParam);

val file = sc.textFile(hdfsFilePath, numPartitions)
val lines = file.flatMap(_.split("\n")).zipWithIndex()
/**
 * Here, we calculate the optimal Generalization level for entire RDD.
 */
val output = lines.mapPartitions(partition =>
  {
    val algo = new LBSAlgorithm(metadata.value, params.value,
                    population.value, zips.value, hashedPopulation.value);
    partition.map {
      case (x, y) => {
        val strategy = algo.findOptStrategy(x);
        (y, strategy._1, strategy._2, strategy._3)
      }
    }
  });
output.persist(StorageLevel.MEMORY_AND_DISK_SER);

/**
 * We cache RDD as we do not want to be recomputed for
 * each of the following three actions.
 */
val publisherBenefit = output.map(_._2).mean();
val advBenefit = output.map(_._3).mean();
val records = output.sortBy(_._1).map(_._4);
println("Avg PublisherPayOff found: " + publisherBenefit)
println("Avg AdversaryBenefit found: " + advBenefit)
val fileName = outputFilePath + "/LBS_" + lbsParam.V() + "_" + lbsParam.L() + "_" + lbsParam.C() + ".csv";
new DataWriter(sc).writeRDDToAFile(fileName, records);
```

LSH bucketing (strict k-anonymity)

Please find the source code for LSH using strict k-anonymity algorithm outlined in section 4.4 described below.

```
val file = sc.textFile(hdfsFilePath, numPartitions)
val lines = file.flatMap(_.split("\n")).zipWithIndex()
val columnCounts = LSHUtil.getTotalNewColumns(metadata.value);
val unitVectors = sc.broadcast(getRandomUnitVectors(columnCounts));
var totalCols = sc.broadcast(LSHUtil.getTotalNewColumns(metadata.value));
val op = lines.mapPartitions({
  var nextStartCount = 0;
  val counts = ListBuffer[Int]();
  for (column <- metadata.value.getQuasiColumns()) {
    counts += nextStartCount;
    if (column.isCharColumn()) {
      nextStartCount = nextStartCount + column.getNumUnique();
    } else {
      nextStartCount = nextStartCount + 1;
    }
  }
  val countsArr = counts.toArray;
 _.map({
   case (x, y) => {
     var index = 0;
     var row = new Array[Double](totalCols.value);
     val split = x.split(",")
     var column = metadata.value.getMetadata(0).get;
     row((countsArr(0) + column.getRootCategory().getIndexOfColumnValue(split(0)))) = 1.0;
     column = metadata.value.getMetadata(3).get;
     row((countsArr(3) + column.getRootCategory().getIndexOfColumnValue(split(3)))) = 1.0;
     column = metadata.value.getMetadata(1).get;
     row(countsArr(1)) = ((split(1).toDouble) - column.getMin()) / (column.getRange());
     column = metadata.value.getMetadata(2).get;
     row(countsArr(2)) = ((split(2).toDouble) - column.getMin()) / (column.getRange());
     val concatenatedBucket = unitVectors.value.map(unitVect => {
     unitVect._1 *(  Math.round(
             ((unitVect._2.zip(row).map({ case (x, y) => x * y }).sum) / r) * precisionFactor) / precisionFactor)
     }).sum;
     row=null;
     (concatenatedBucket, (ListBuffer[Long](y),
           (Set[String](split(0)), Array.fill(2)(split(1).toInt),
             Array.fill(2)(split(2).toInt), Set[String](split(3)))))
   }
 })
}).reduceByKey({
  case ((id1, (a, b, c, d)), (id2, (p, q, r, s))) => {
    if (b(0) > q(0)) {
      b(0) = q(0);
    }
    if (b(1) < q(1)) {
      b(1) = q(1);
    }
    if (c(0) > r(0)) {
      c(0) = r(0);
    }
    if (c(1) < r(1)) {
      c(1) = r(1);
    }
    (id1.++=:(id2), (a.union(p), b, c, d.union(s)))
  }
 }).flatMap({
    case (bucket,(ids, (genders, zips, ages, races))) =>
      {
        if (ids.size >= numNeighbors.value) {
          var column = metadata.value.getMetadata(0).get;
          var generalization = column.findCategory(genders.toArray).value()
          var min = zips.min;
          var max = zips.max;
          if (min == max) {
            generalization += "," + min;
          } else {
            generalization += "," + min + "_" + max;
          }
          min = ages.min;
          max = ages.max;
          if (min == max) {
            generalization += "," + min;
          } else {
            generalization += "," + min + "_" + max;
          }
          column = metadata.value.getMetadata(3).get;
          generalization += "," + column.findCategory(races.toArray).value()
          ids.map(x => (x, generalization));
        } else {
          ids.map(x => (x, "*,37010_72338,29,*"));
        }
      }
 });
```

LBS-LSH source code

Please find the source code of LBS-LSH code below. As we can see, we first convert lines into the quantitative format and then we bucket them. Once done, we invoke LBS on each of the buckets individually based on the LBS-LSH algorithm defined in section 5.5.

```
val population = LBSMetadataWithSparkContext.getPopulation(sc);
val zips = LBSMetadataWithSparkContext.getZip(sc);
val hashedPopulation = LBSMetadataWithSparkContext.getHashedPopulation(sc);
val params = sc.broadcast(lbsParam);

val file = sc.textFile(hdfsFilePath, numPartitions)
val lines = file.flatMap(_.split("\n")).zipWithIndex()
val columnCounts = LSHUtil.getTotalNewColumns(metadata.value);
val unitVectors = sc.broadcast(getRandomUnitVectors(columnCounts));
var totalCols = sc.broadcast(LSHUtil.getTotalNewColumns(metadata.value));
val op= lines.mapPartitions({
  var nextStartCount = 0;
  val counts = ListBuffer[Int]();
  for (column <- metadata.value.getQuasiColumns()) {
    counts += nextStartCount;
    if (column.isCharColumn()) {
      nextStartCount = nextStartCount + column.getNumUnique();
    } else {
      nextStartCount = nextStartCount + 1;
    }
  }
  val countsArr = counts.toArray;
  _.map({
    case (x, y) => {
      var index = 0;
      var row = new Array[Double](totalCols.value);
      val split = x.split(",")
      var column = metadata.value.getMetadata(0).get;
      row((countsArr(0) + column.getRootCategory().getIndexOfColumnValue(split(0)))) = 1.0;
      column = metadata.value.getMetadata(3).get;
      row((countsArr(3) + column.getRootCategory().getIndexOfColumnValue(split(3)))) = 1.0;
      column = metadata.value.getMetadata(1).get;
      row(countsArr(1)) = ((split(1).toDouble) - column.getMin()) / (column.getRange());
      column = metadata.value.getMetadata(2).get;
      row(countsArr(2)) = ((split(2).toDouble) - column.getMin()) / (column.getRange());
      val concatenatedBucket = unitVectors.value.map(unitVect => {
      unitVect._1 * (  Math.round(((unitVect._2.zip(row).map({ case (x, y) => x * y }).sum) / r)
                    * precisionFactor) / precisionFactor)
      }).sum;
```

APPENDIX C

CONFIGURATION AND COMMANDS

This appendix shows commands used in order to execute online as well as batch sanitization implementations.

Apache Spark job configuration

The performance of job submitted to Apache Spark heavily depends on several parameters. Before submitting a job request, we must perform analysis and decide parameters for the job. Apache spark is extremely sensitive to these configuration parameters. Number of partitions to be performed on input dataset determines how many partitions will be created. A task gets created per partition and is assigned to an executor. The number of partitions chosen will be different based on dataset size and analysis must be performed to select the appropriate size for each partition. More executor memory causes heavy garbage collection breaks, leading to reduced performance. Smaller executor memory can cause OOM. Similarly, driver memory, as well as a number of cores on the driver, must be similarly tuned too. Some jobs can run for a long time, spark network timeout must be set in order to accommodate same. Also, some jobs get stuck, to not let that happen, we need to enable speculation. Speculation depends on multiplier and quantile. Quantile checks decide when speculation should be enabled. Enabling this early (0) can cause performance to degrade hence we set it to a higher value of 0.9, which means that speculation is started only after 90% of the jobs have completed execution. Similarly, speculation multiplier is used for stating that these many times slower job than the median should be killed. We modified the value to three because too many jobs were getting killed for a smaller value. This, however, is subjective

and is completely dependent on the state of the cluster as well the DAG. Please find a subset of parameters, which were tuned for this thesis, listed below.

- Number of dataset partitions
- spark.executor.cores=2/5/10
- spark.executor.memory=2G/5G/10G
- spark.network.timeout=800
- spark.speculation=true
- spark.speculation.multiplier=3
- spark.driver.memory=6G/8G
- spark.driver.cores=8
- spark.speculation.quantile=0.9
- spark.shuffle.consolidateFiles=true
- spark.executor.heartbeatInterval=50s

Please find commands used for executing LBS/LBS/LBS-LSH implementations, listed below. Please note that a single codebase was maintained for batch as well as online implementation. In order to invoke LBS/LSH/LBS-LSH, a single class LBSAndLSH needs to be executed. The format of attributes to be provided to this class is as shown below.

<spark_master><hdfs_data_file_path><output_file_path><recordcost><maxpublisherbenefi><publishersloss><numpartitions><algorithm (lbs/lbslsh/lsh)><lsh_num_neighbors>

Please find sample commands for execution of each of the algorithm, below.

Command for Invoking LSH

```
./spark-submit --class edu.utd.security.risk.LBSAndLSH --deploy-mode client --conf
spark.executor.cores=10  --conf spark.executor.memory=10G  --conf
spark.network.timeout=800 --conf spark.speculation=true --conf spark.speculation.multiplier=3 --
conf spark.driver.memory=6G --conf spark.driver.cores=8 --conf spark.speculation.quantile=0.9
--conf spark.shuffle.consolidateFiles=true --conf spark.executor.heartbeatInterval=50s --master
"spark://cloudmaster3:7077"  "/data/kanchan/blueray-1.2-ASPECT-SNAPSHOT.jar"
"spark://cloudmaster3:7077"
"hdfs://cloudmaster3:54310/user/adult_zip80G.csv"  "hdfs://cloudmaster3:54310/user/"  0.3333
100 8.3333 31000 lsh 3
```

Command for Invoking LBS

```
./spark-submit --class edu.utd.security.risk.LBSAndLSH --deploy-mode client --conf
spark.executor.cores=10  --conf spark.executor.memory=10G  --conf
spark.network.timeout=800 --conf spark.speculation=true --conf spark.speculation.multiplier=3 --
conf spark.driver.memory=6G --conf spark.driver.cores=8 --conf spark.speculation.quantile=0.9
--conf spark.shuffle.consolidateFiles=true --conf spark.executor.heartbeatInterval=50s --master
"spark://cloudmaster3:7077"  "/data/kanchan/blueray-1.2-ASPECT-SNAPSHOT.jar"
"spark://cloudmaster3:7077"
"hdfs://cloudmaster3:54310/user/adult_zip80G.csv"  "hdfs://cloudmaster3:54310/user/"  0.3333
100 8.3333 31000 lbs 3
```

Command for Invoking LBS-LSH

```
./spark-submit --class edu.utd.security.risk.LBSAndLSH --deploy-mode client --conf
spark.executor.cores=10  --conf spark.executor.memory=10G  --conf
spark.network.timeout=800 --conf spark.speculation=true --conf spark.speculation.multiplier=3 --
conf spark.driver.memory=6G --conf spark.driver.cores=8 --conf spark.speculation.quantile=0.9
--conf spark.shuffle.consolidateFiles=true --conf spark.executor.heartbeatInterval=50s --master
"spark://cloudmaster3:7077"  "/data/kanchan/blueray-1.2-ASPECT-SNAPSHOT.jar"
"spark://cloudmaster3:7077"
"hdfs://cloudmaster3:54310/user/adult_zip80G.csv"  "hdfs://cloudmaster3:54310/user/"  0.3333
100 8.3333 31000 lbslsh 3
```

Similarly, we can invoke online implementation by specifying following command. Please note that following commands assume that blueray has been deployed in /data/kanchan folder instead of /data/blueray. This was done in order to comply with the UTD's policy of storing user files in user folders. We can see that with the following command, we are trying to execute "BlueRayTest"

class while setting the user as “kanchan” in the system environment. This variable will be read by BlueRay aspect in order to apply the appropriate policy.

Command:

```
./spark-submit --conf spark.executor.cores=2  --conf spark.executor.memory=2G --master "spark://cloudmaster3:7077"  --class BlueRayTest  --conf "spark.executorEnv.USER=kanchan" --master "spark://cloudmaster3:7077" /data/kanchan/BlueRayTest-0.0.9-SAVE-SNAPSHOT.jar kanchan "hdfs://cloudmaster3:54310/user/adult_zip1G.csv" "hdfs://cloudmaster3:54310/user/GENER_12G.csv" "spark://cloudmaster3:7077"
```

Executing RESTFul policy manager on BlueRay aspect

In order to execute a specific type of iterator, we need to set appropriate parameters in the system environment of the driver as well as the executor. The BlueRay aspect offers following options to be selected based on the requirement.

POLICYMANAGER_END_POINT – For RESTful policy manager based sanitization.

BLUERAY_POLICIES_PATH – For local policy store based sanitization.

BlockColumns – for Identifier field generalization or suppression.

Please find a sample invocation of command provided below. Please note that following command assumes that RESTful policy manager is available on http://192.168.4.1:8084/bluerayWebapp link.

Command:

```
./spark-submit --conf spark.executor.cores=2  --conf spark.executor.memory=2G --master "spark://cloudmaster3:7077"   --conf "spark.executor.extraJavaOptions=-javaagent:/data/kanchan/aspectjweaver-1.8.5.jar"    --conf "spark.executorEnv.USER=kanchan" --conf "spark.executorEnv.POLICYMANAGER_END_POINT=http://192.168.4.1:8084/bluerayWebapp" --driver-java-options "-DPOLICYMANAGER_END_POINT=http://192.168.4.1:8084/bluerayWebapp -javaagent:/data/kanchan/aspectjweaver-1.8.5.jar" --class BlueRayTest  --master "spark://cloudmaster3:7077" /data/kanchan/BlueRayTest-0.0.9-SAVE-SNAPSHOT.jar  kanchan
```

"hdfs://cloudmaster3:54310/user/adult_zip20G.csv"
"hdfs://cloudmaster3:54310/user/GENER_12G.csv" "spark://cloudmaster3:7077"

Similarly, please find command to be used for invoking local policy manager, below. Please note that policies file must be kept in sync on all executors to ensure that this is done the right way.

Command for execution of local policy manager on BlueRay aspect

./spark-submit --conf spark.executor.cores=2 --conf spark.executor.memory=2G --master "spark://cloudmaster3:7077" --conf "spark.executor.extraJavaOptions=-javaagent:/data/kanchan/aspectjweaver-1.8.5.jar" --conf "spark.executorEnv.USER=kanchan" --conf "spark.executorEnv.BLUERAY_POLICIES_PATH=/data/kanchan/policies.csv" --driver-java-options "-DBLUERAY_POLICIES_PATH=/data/kanchan/policies.csv -javaagent:/data/kanchan/aspectjweaver-1.8.5.jar" --class BlueRayTest --master "spark://cloudmaster3:7077" /data/kanchan/BlueRayTest-0.0.9-SAVE-SNAPSHOT.jar kanchan "hdfs://cloudmaster3:54310/user/adult_zip1G.csv" "hdfs://cloudmaster3:54310/user/GENER_10G.csv" "spark://cloudmaster3:7077"

In order to perform multi-field generalization or suppression, we can specify BlockColumns.

Please find sample commands for both listed below.

Command for suppressing columns 2 and 3

./spark-submit --conf spark.executor.cores=2 --conf spark.executor.memory=2G --master "spark://cloudmaster3:7077" --conf "spark.executor.extraJavaOptions=-javaagent:/data/kanchan/aspectjweaver-1.8.5.jar" --conf "spark.executorEnv.BlockColumns=4[1,2]" --conf "spark.executorEnv.USER=kanchan" --driver-java-options "-DBlockColumns=4[1,2] -javaagent:/data/kanchan/aspectjweaver-1.8.5.jar" --class BlueRayTest --master "spark://cloudmaster3:7077" /data/kanchan/BlueRayTest-0.0.9-COUNT-SNAPSHOT.jar kanchan "hdfs://cloudmaster3:54310/user/adult_zip80G.csv" "hdfs://cloudmaster3:54310/user/GENER_10G.csv" "spark://cloudmaster3:7077"

The command for generalization is very much similar to suppression command except that as part of BlockColumns, we also provide a path of metadata file that contains generalization hierarchy for columns 2 and 3.

Command for generalizing columns 2 and 3:

```
do ./spark-submit --conf spark.executor.cores=2  --conf spark.executor.memory=2G --master
"spark://cloudmaster3:7077” --conf "spark.executor.extraJavaOptions=-
javaagent:/data/kanchan/aspectjweaver-1.8.5.jar"   --conf
"spark.executorEnv.BlockColumns=4[2,3]/data/kanchan/metadata_exp.xml"   --conf
"spark.executorEnv.USER=kanchan" --driver-java-options "-
DBlockColumns=4[2,3]/data/kanchan/metadata_exp.xml  -
javaagent:/data/kanchan/aspectjweaver-1.8.5.jar" --class BlueRayTest  --master
"spark://cloudmaster3:7077" /data/kanchan/BlueRayTest-0.0.9-SAVE-SNAPSHOT.jar  kanchan
"hdfs://cloudmaster3:54310/user/adult_zip1G.csv"
"hdfs://cloudmaster3:54310/user/GENER_1G.csv" "spark://cloudmaster3:7077"
```

REFERENCES

Census Summary File prepared by the US Census Bureau. (Oct, 2014). Retrieved from US Census Bureau: https://www.census.gov/prod/cen2010/doc/sf2.pdf

Chakravorty, A. (2016, June). *Incognito library of Apache spark based k-anonymization.* Retrieved 04 07, 2017, from https://github.com/achak1987/incognito

Datar, M., Immorlica, N., Indyk, P., & Mirrokni, V. S. (2004). Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. *SCG '04 Proceedings of the twentieth annual symposium on Computational geometry* (pp. 253-262). New York: ACM .

Dwork, C. (2008). Differential Privacy: A Survey of Results. *Lecture Notes in Computer Science, vol 4978* (pp. 1-19). Berlin Heidelberg: Springer .

El Emam, K., Jonker, E., Arbuckle, L., & Malin, B. (2011, December 2). A Systematic Review of Re-Identification Attacks on Health Data. *PLoS ONE, 6*(12).

Everitt, B. (2011). *Cluster analysis.* Chichester, West Sussex, U.K: U.K: Wiley.

Ghemawat, S., & Jeffrey, D. (2004, Dec). MapReduce: Simplified Data Processing on Large Clusters. *Sixth Symposium on Operating System Design and Implementation.* San Francisco, CA: OSDI'04.

Ghinita, G., Kalnis, P., & Tao, Y. (2010). Anonymous Publication of Sensitive Transactional Data. *IEEE Transactions on Knowledge and Data Engineering*, 161 - 174.

Gostin, L. O., Lazzarini, Z., & Neslund, V. S. (1996). A National Review of the Law on Health Information Privacy. *JAMA, 275*(24), 1921-1927.

HHS. (2000, December 28). *HIPAA Privacy Rule.* Retrieved from HHS.gov: https://www.hhs.gov/sites/default/files/ocr/privacy/hipaa/administrative/privacyrule/prdecember2000all8parts.pdf?language=en

Inaba, M., Katoh, N., & Imai, H. (1994). Applications of weighted Voronoi diagrams and randomization to variance-based k-clustering. *10th ACM Symposium on Computational Geometry* (pp. 332–339). doi:10.1145/177424.178042.

Indyk, P., & Motwani, R. (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. *Proceedings of the thirtieth annual ACM symposium on Theory of computing* (pp. 604-613). Dallas, Texas, USA: STOC.

Jain, A., M.N., M., & P.J., F. (1999). Data clustering: A review. *ACM Computing Surveys (CSUR), 31*(3), 264-323.

Judson, B. S. (2015, June). *Data Anonymizing in Hadoop: A TED Case Study.* Retrieved 04 07, 2017, from https://blogs.msdn.microsoft.com/partnercatalystteam/2015/06/04/data-anonymizing-in-hadoop-a-ted-case-study/

K-Anonymity. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/K-anonymity

Lichman, M. (2013). *Adult Data Set.* Retrieved 12 12, 2016, from UCI Machine Learning Repository: http://archive.ics.uci.edu/ml/datasets/Adult

Locality Sensitive Hashing. (2016, Dec 28). Retrieved 02 07, 2017, from Apache Spark Documentation: https://spark.apache.org/docs/latest/ml-features.html#lsh-operations

Machanavajjhala, A. a. (2007, March). L-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data (TKDD)*.

Mumtaz K, D. K. (2010). An Analysis on Density Based Clustering of Multi Dimensional Spatial Data. *1*(1), 8-12.

Ramakrishnan, R., LeFevre, K., & DeWitt, D. J. (2006). Mondrian Multidimensional K-Anonymity. *ICDE '06 Proceedings of the 22nd International Conference on Data Engineering* (p. 25). IEEE Computer Society Washington, DC, USA ©2006.

Ruggieri, S. (2014, August). Using t-closeness anonymity to control for non-discrimination. *Transactions on Data Privacy, 7*(2), 99-129.

Safran, C., Bloomrosen, M., Hammond, W. E., Labkoff, S., Markel-Fox, S., Tang, P. C., & Detmer, D. E. (2007, 01 01). Toward a National Framework for the Secondary Use of Health Data: An American Medical Informatics Association White Paper. *An American Medical Informatics Association, 14*(1), pp. 1-9.

Stephens, Z. D., Lee, S. Y., Faghri, F., Campbell, R. H., Zhai, C., Efron, M. J., . . . Robinson, G. E. (2015). Big Data: Astronomical or Genomical? *PLOS : Biology*.

Syed, M., & Srikanth, V. (2016, September). *FOR YOUR EYES ONLY: DYNAMIC COLUMN MASKING & ROW-LEVEL FILTERING IN HDP2.5*. Retrieved from https://hortonworks.com: https://hortonworks.com/blog/eyes-dynamic-column-masking-row-level-filtering-hdp2-5/

Ulusoy, H., Kantarcioglu, M., Pattuk, E., & Hamlen, K. (2014). Vigiles: Fine-grained Access Control for MapReduce Systems. *Proceedings of the 2014 IEEE International Congress on Big Data* (pp. 40-47). Washington DC, USA: IEEE Computer Society.

Wan, Z., Vorobeychik, Y., Xia, W., Wright, E., Clayton, Kantarcioglu, M., . . . A. Malin, B. (2015). A Game Theoretic Framework for Analyzing Re-Identification Risk. *PLOS*.

Wogara, J. (2001, 05 01). Human Rights and Patients' Privacy in UK Hospitals. *Human Rights and Patients' Privacy in UK Hospitals*. Retrieved 04 07, 2017, from https://en.wikipedia.org/: Human Rights and Patients' Privacy in UK Hospitals

Zaharia, M. (2012, 12). *Spark Internals documentation.* Retrieved 01 01, 2017, from Spark Internals documentation: http://files.meetup.com/3138542/dev-meetup-dec-2012.pptx

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., . . . Ion, S. (2012). Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (pp. 2-2). CA, USA: USENIX Association Berkeley. Retrieved 01 01, 2017, from A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Zhang, X., Leckie, C., Dou, W., Chen, J., Kotagiri, R., & Salcic, Z. (2016). Scalable Local-Recoding Anonymization using Locality Sensitive Hashing for Big Data Privacy Preservation. *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management* (pp. 1793-1802). Indianapolis, Indiana: ACM New York, NY.

Zhang, X., Yang, L., & Liu, C. (2013). A Scalable Two-Phase Top-Down Specialization Approach for Data Anonymization Using MapReduce on Cloud. *IEEE Transactions on Parallel and Distributed Systems*, 363 - 373.

Zhou, B., Pei, J., & Luk, W. (2008). A brief survey on anonymization techniques for privacy preserving publishing of social network data. *ACM SIGKDD Explorations Newsletter, 10*(2), 12-22. Retrieved 04 07, 2017

BIOGRAPHICAL SKETCH

Kanchan Waikar is a master's student in the Department of Computer Science at The University of Texas at Dallas. She received her Bachelor's degree in Computer Science and Technology from S.N.D.T University, Juhu, India in 2005. She has over 10 years of industry experience spanning NLP, J2EE, web designing, development and architecture management of technical products. Her career interests are data science, big data technologies and distributed system architectures.

CURRICULUM VITAE

Synopsis:

I am a curious technologist. I absolutely love development, design, architecting and analyzing software systems. I have been working as backend lead, technical product lead and have over 10 years of software experience. I have played several roles - UX specialist, NLP developer, java web developer, full stack engineer, project manager, dev-ops intern, technical product lead and recently ML/data security researcher. Spring, Hadoop, SOLR, Memcached, Hibernate, Spark, Scala, Docker are some of my most favorite tools/technologies apart from Java and J2EE.

At School, I have taken very interesting subjects like Machine learning, big data, advanced algorithms, Natural Language Processing, Semantic Web, etc. My master's thesis topic is big data sanitization using distributed in-memory frameworks.

Education

Master's in Computer Science (Data Science - Graduating in spring 2017)

University: The University of Texas at Dallas

CGPA: **3.85/4.0**

Bachelor's in Computer Science

College: Usha Mittal Institute of Technology, S.N.D.T. University, Mumbai

Degree: Bachelor's degree in Computer Science & Technology (distinction)

CGPA: 7.81/10 (University Topper in 7th Semester)

Recognitions

- Won spotlight award in June 2016
- Received thank you award from FirstFuel, in March - 2015
- CEO excellence award in February 2013
- Received several appreciation letters from Vice President, CEO while working at Credit Pointe.
- Exhibited **'Multi-term Entity Extraction Engine'** at Symantec's annual conference in Salt Lake City, Utah, **USA** in October 2008.
- Java Developer Journal/Sys-Con author - kanchanwaikar.sys-con.com

Experience Summary:

Organization	**Role/Duration**	**Tools & Technologies used**
The University of Texas at Dallas	Researcher [Aug-2016 – April 2017]	Spark, Hadoop, MapReduce, AspectJ, Scala, Java, and Junit.
Intuit	Summer Intern [May-2016- Aug 2016]	Docker, Docker Swarm, Jenkins, Shell scripts.
Firstfuel Account, Xpanxion, India	Backend Technical Lead /Scrum Master [March 2014 – July - 2015]	Spring 3.0.3, Java, MySQL, Junit, Maven, AGILE methodology, AWS, Confluence-Bamboo-Jira-GIT suite, and Hibernate.
Credit Pointe, Indian subsidiary of Rage Frameworks	Full Stack Technical Lead [December 2010 to March 2014]	Java, Spring 3.0.3, Apache Server, Apache **Tomcat**, **Memcached**, Oracle, PL-SQL, **SOLR**, **JSP**, EXT-JS, AM Charts, Fusion Charts, Jprofiler, Jersey, and **JQuery.**
Symantec corporation	NLP/Full stack Developer [April 2008 to November 2010]	**Java**, Struts, JSP, **Lucene**, **SOLR**, SVN Apache Tomcat, MySQL, Apache Tika, PDFBox, Apache James, **ANT**, and Perforce.
Wipro Technologies	On-site Review Team Lead, Developer [September 2005 to March 2008]	CSS, **Java**, **JSP**, Javascript, JSF, HTML, and Mainframes.

Projects

Firstfuel (http://www.firstfuel.com/) - Xpanxion/Firstfuel limited.

FirstFuel is a SaaS solution that dissects fuel usage data of buildings and applies advanced analytical models in order to identify actionable insight for decreasing net fuel usage.

I played the role of technical Epic Lead for several modules like end use bounds disaggregation, automatic monitoring execution, energy usage peer ranking determination, thermal regression, deep analysis, etc. I worked on building complex, highly scalable and maintainable backend pieces involving integration with statistical models energy analytics platform.

Real Time Intelligence (http://www.rageframeworks.com/capital_markets/) - Credit Pointe

RTI application solves the problem of information asymmetry by combining real-time data from news and social media sources, and by performing a comprehensive analysis in order to identify the performance trend of the company.

I owned the technology portion of RTI portal - (1.0, 2.0, and 2.1). I architected & developed RTI – 2.0, upgraded public facing website, and adjudication and business analytics layer of customer portal. I architected back-end in order to add support for up to 1000 concurrent users by using more advanced backend approaches that involved fairly advanced concepts like usage of SOLR, Memcached, and service level caching.

PricePucho (Personal Project)

This website's primary purpose was to bridge the gap between Indian consumers and small-scale businesses. It provided powerful features that let users contact small scale businesses.

A website that I developed, maintained, and hosted by renting a virtual private server. This website used SOLR for providing search, filtering and faceting features and Memcached for caching.

NUI/NUL Tech-Decom - Wipro

This project targeted migration of redundant systems formed because of business mergers and acquisitions. I worked as a System Analyst & Review Team Lead at Norwich Union's head office, located in the United Kingdom.

Internship Project - Lacerte/Docker - Intuit, Dallas

- Lacerte SCM Build process performance analysis and improvement. I improved the capacity of Build system by 60% and build job speed by 50% without the addition of any new hardware.
- I was one of the key members of the team that automated CI-CD pipeline implementation using Docker Swarm.

SymHelp - Symantec Corporation

SymHelp is a context-sensitive help system that performs content categorization and returns results based on user query sentiment.

I was involved in SymHelp 2.0 design, the addition of L10N support, an end to end release automation and development of several web modules.

Multi-term Entity Extraction Engine - Symantec Corporation

This tool parsed PDF/TXT/Word/presentation files, extracted noun variants, verb variant entities along with their TF/TDF, and identified domain specific multi-term entities along with their importance. These entities were consumed by the SymHelp search engine in order to perform user intent analysis.

I owned development of this project end-to-end and presented an exhibit of the same at Cutting edge 2008 annual conference in Salt Lake city, Utah, USA.

Extra-Curricular Activities:

- I own a painting artist page on Facebook.
- I won a 2nd prize at intra-university chess competition.
- I played chess for The SNDT University at the national level.
- I was event manager for several events in FirstFuel and Wipro.