Top Down Specialization on Apache Spark$^{\text{TM}}$

Macarious Abadeer
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
macariousabadeer [at] scs.carleton.ca

March 30, 2020

Abstract

The abundance of data as well as regulations protecting people's privacy created a need for protecting private and personal information in a scalable and efficient way. Personal data includes sensitive and private information such as health records, banking transactions and frequent locations. One of the challenges of data anonymization is when the data anonymity increases its usefulness for analytics or research decreases. This paper provides an implementation of Top-Down Specialization algorithm for data anonymization in parallel using Apache Spark which aims to balance data utility and data privacy. Performance evaluation is done on large datasets of up to 20-million rows in a variety of different cluster environments.

1 Introduction

Since the introduction of multi-core processors in 2004 by Intel$^{®}$, parallel computing evolved to exploit the advantages of multiple processing units that became the norm for personal computers. This evolution was also expanded and accelerated by the advancements in Cloud Computing that supported running compute-intensive applications over a network of clusters. Parallel computing enabled the development of solutions to different real world applications that were hindered by scalability limitations such as big data analytics, machine learning and artificial intelligence. One of the problems that parallel computing provided scaleable solutions for is data anonymization, especially for big data.

In today's abundance of big data ranging from retail and banking transactions, health care, social media interactions and sensor data, a need was created for measures that protect people's most private and sensitive data. One of the most popular theories that were developed in this area was k-anonymity developed by Samarati and Sweeney in 1998 [14]. Sweeney argued that an individual in a dataset can be identified when the dataset is linked with other public datasets even if the original dataset did not contain identifying information such as name, date of birth and social insurance number. Sweeney was able to show that when linking voter registration cards and health care data, individuals can be identified with 87% accuracy. Those potentially identifying attributes are called Quasi-Identifiers (QID). k-anonymity states that a dataset is called k-anonymous when for a given record, there exists at least $k-1$ records in the same dataset with the same QID values. Further modifications to k-anonymity were made to overcome its shortcomings such as introducing ℓ-diversity [8] and t-closeness [7]. ℓ-diversity ensures that sensitive attributes, such as diagnosis in a health care dataset, need to have diverse values so that an adversary with foreknowledge of a given QID set cannot deduce their diagnosis. t-closeness ensures that the distribution of these diverse values is close to their distribution in the original dataset.

While these theories contributed immensely to the practices of data anonymization, k-anonymity was proven to be $\mathcal{NP}$-hard by Meyerson and Williams [10]. Further research used these models as a baseline to develop scalable parallel algorithms that can handle big data.

The paper is organized as follows: in Section 2, I will go over the different ideas that were proposed to scale k-anonymity. Section 3 defines the problem and Section 4 details a proposed Top Down Specialization solution in parallel. Section 5 presents the experimentation results of the algorithm and the paper finally concludes in Section 6.

2 Literature Review

There are three different masking types that are used to satisfy k-anonymity: interval, taxonomy tree and suppression [1]. Suppression requires certain outlier tuples to be removed to satisfy k-anonymity [14]. Intervals and taxonomy trees are generalization techniques applied to numerical and categorical attributes respectively [14]. For example two records with birth year of 1971 and 1973 can be generalized to 1970-1975. For a taxonomy tree, a categorical attribute such as education level can have, for example, post-graduate as a parent node which can have PhD, Masters and Post Graduate Diploma as its child leaves so that records with these values can be generalized to the parent node. The majority of research papers on anonymization with respect to big data involved taxonomy trees thus this is where I focus my literature survey.

One of the techniques that researchers attempted to optimize was Bottom-Up Generalization (BUG) which involves traversing the taxonomy tree of attribute hierarchies from the bottom (most specific) upwards (most general) [6]. Wang suggested that the taxonomy tree would be provided by the data supplier or the data recipient [6]. As the tree is traversed, two metrics are calculated to ensure a high quality generalization: information loss and anonymity gain. An indexed approach to bottom-up generalization was proposed by Hoang [5] where the taxonomy tree for numerical attributes was generated automatically at runtime. Hoang's indexed approach could also handle numerical as well as categorical attributes. Indexed BUG starts with collecting statistical information about the dataset as well as partition it so it can be used in the generalization step which was further broken down to four steps: calculate the best generalization score based on the least information loss, calculate k-anonymity for every partition, generate an indexed generalization map which maps every value to its generalized value, and the last step creates the anonymized dataset using this map. Hoang's experiments showed that the generalization time did not increase with the dataset size due to the use of indexed generalization map however performance was impacted by the distinct values count for each QID [5].

Parallel BUG was introduced to address the limitations of traditional and indexed BUG approaches. Pandilakshmi attempted to solve the limitations of indexing structures since they are centralized and hard to parallelize and cannot run on distributed systems such as the Cloud [11]. Pandilakshmi introduced Bi-Level BUG algorithm where MapReduce framework was used to take advantage of job-level and task-level parallelization. Job-level parallelization was achieved by using multiple MapReduce jobs and task-level parallelization was achieved by using multiple mapper/reducer tasks for every MapReduce job so they are executed in parallel on every partition. Data is partitioned according to a random number generated between 1 and p where p is the number of partitions. Pandilakshmi then runs MapReduce BUG driver (MRBUG) iteratively on the partitioned datasets and calculates generalization score (least information loss with the most anonymity gain) and stops until it finds the best generalization with the highest score that satisfies k-anonymity. Pandilakshmi experiments performed on varying datasets of up to 3GB showed that execution time was virtually capped at $\approx$33 minutes regardless of dataset size.

Another technique is Top-Down Specialization (TDS). TDS traverses the taxonomy tree from the top downwards where it starts with the most generalized values and specializes the value and stops when it violates k-anonymity [4]. Multiple solutions have been developed such as a scalable two-phase TDS introduced by [13] and [18]. The first phase involves partitioning the original dataset to p partitions using random sampling. A MapReduce TDS job runs in parallel on each partition. Each job specializes the data iteratively while calculating information gain and privacy loss metrics and creates an intermediate anonymized dataset. In the second phase the intermediate datasets are merged and further anonymized if necessary to satisfy k-anonymity. In [18], Zhang et al. adopted Hadoop® and took advantage of distributed cache capability to pass the intermediate anonymized dataset to each mapper/reducer node. The experiments for this solution showed an overhead in the partitioning phase of the dataset.

A hybrid approach of BUG and TDS using MapReduce was introduced by [17] where it was shown that when either TDS or BUG were used individually, they performed poorly for certain values of k. The hybrid approach applies TDS for large k values and BUG for smaller ones. The notion of Workload Balancing Point was introduced which is defined as the point where the amount of computation required for TDS is the same as BUG. Once that point is identified, the hybrid approach chooses TDS for k greater than the workload balancing point and chooses BUG when k is smaller. The workload balancing point is estimated using the height of the taxonomy tree as a reference.

Al-Zobbi et al [1] argued that finding the highest scoring generalization and specialization based on information gain and anonymity loss in BUG and TDS require high computational costs and impedes the ability to parallelize them. Al-Zobbi also argued that as the data grows in size, the high accuracy of these computations no longer make a statistical difference. Al-Zobbi proposed a multi-dimensional sensitivity-based algorithm on Apache Spark that uses a pre-determined QID attributes to anonymize as well as precalculated k value using linear regression. The solution also takes into consideration the probability value of each QID. For example assuming that age can range between 1 and 100, the probability of finding a given age is 1% which is much higher than a probability of finding a given job title assuming there are 200 different job titles. The solution prioritized the anonymization of higher probability attributes instead of calculating information gain and anonymity loss scores for every attribute. The solution also used a role-based access control equivalent system to set k based on context. For example a health care dataset maybe given a lower k (less anonymization) when shared with a doctor but a higher k value when shared with an insurance risk analyst. The solution was implemented on Spark and aimed at minimizing the use of User Defined Functions (UDFs) since they run outside of the Spark JVM which is beyond the resource negotiator's control. Al-Zobbi recognized that this solution would sacrifice the analytical value of the dataset for the performance improvement gained by not calculating the best generalization options.

In a research paper by [15], a survey was done on MapReduce vs. Spark for big data analytics. It concluded that Spark is better suited for problems that require accessing the same dataset multiple times such as the case with both TDS, BUG and their variants. The constant read and write by Hadoop to HDFS (Hadoop Distributed File System) is considered a significant overhead however Spark operates on datasets in memory and provides the capability of caching Resilient Distributed Datasets (RDDs) for faster access making it suitable for iterative algorithms. The experiments carried out by Shi et al in [15] found that Spark is 5 times faster than MapReduce for iterative algorithms regardless of data size.

Shi's findings in [15] are inline with other researchers that implemented anonymization algorithms on Spark such as [2] and [16]. For example, [16] proposed a TDS implementation for Apache Spark that partitioned the dataset to p partitions on n Spark nodes where $n = p$. The master node partitions the data and calculates the scores required by TDS such as information gain and privacy loss. The scores are sent to the driver node which performs aggregations required by further iterations until k is satisfied. The experiments carried out for this solution by [16] showed that there is an overhead cost incurred when having more than one partition in a single node. The experiments also showed performance gains regardless of k values and dataset sizes as long as Spark nodes are increased with the dataset size. As outlined by [1], ideally the partitioned dataset needs to fit in the node's memory in order to avoid spilling to disk.

The previously mentioned solutions are generic enough to be applied to any type of datasets. However, multiple other solutions have been proposed to address specific anonymization scenarios. I briefly include them here due to their relevance in terms of parallelization techniques. Parameshwarappa [12] for example proposed a solution to anonymize physical activity collected by wearable gadgets. It uses a multi-level clustering algorithm based on Maximum Distance to Average Vector (MDAV). It attempts to cluster data points so that every cluster satisfies k-anonymity. If a cluster does not satisfy k-anonymity, differential privacy technique is used to add statistical noise to the cluster in a way that does not skew the analytical value of the dataset.

Another solution was implemented to provide a parallel anonymization of transaction data such as retail and banking datasets in [9]. It uses an algorithm known as RBAT on MapReduce which uses set-based generalization to anonymize data based on user-provided set of rules. It partitions data in a way that ensures the workload of every partition is approximately the same across different partitions. The solution scans the whole dataset in order to achieve this efficient partitioning based on QID values to minimize data shuffling across partitions.

Other frameworks were also developed to address specific variations to k-anonymity mentioned such as t-closeness introduced by [7]. For example, [3] developed a framework called Incognito using MapReduce that generates a distribution of sensitive attributes based on their count in the dataset. Given the frequency histogram generated, subsets of the sensitive attribute values that have the same parent in the taxonomy tree are put together in the same data bucket. The tree is sorted from left to right nodes in an ascending order of their frequencies in the generated histogram. The anonymized dataset is then mapped to the generated tree in order to ensure anonymized dataset is close to the original dataset in terms of distribution of values.

3 Problem Statement

This paper tackles the scalability of anonymization algorithms for Big Data specifically Top Down Specialization algorithm. There was only a handful of papers in the literature reviewed in Section 2 that implemented anonymization algorithms on Spark and only one that implemented Top Down Specialization [16]. In that implementation the performance was assessed only up to 500 MB of data which is relatively small in the context of Big Data. This paper aims to assess the performance of Top Down Specialization on Spark for datasets larger than 500 MB. It will also use number of records as the gauge instead of size on disk. The question I aim to answer, how does Top Down Specialization scale up for datasets larger than 5 million rows or 500 MB? Are there any optimizations that can be done to improve speedups?

4 Proposed Solution

4.1 Introduction to k-anonymity

First, it is important to review definitions that will be used throughout this paper:

Definition 1 (k-anonymity) *A dataset is called k-anonymous if for every record there exists at least $k-1$ other records with the same Quasi-Identifier values.*

Definition 2 (Quasi-Identifiers) *Quasi-Identifiers are attributes that do not directly identify an individual, but when used together and linked with other datasets they have the potential of identifying an individual. They will be referred to as QID throughout the paper.*

Definition 3 (Sensitive Attributes) *Sensitive Attributes are attributes that should remain private so an adversary cannot deduce their values. They will be referred to as SA throughout the paper.*

Definition 4 (Taxonomy Trees) *Taxonomy Trees are logical hierarchies of distinct values in a dataset.*

For example, in Table 1 *Education*, *Gender* and *City* are examples of QIDs while *Income* is an example of SA. Table 1 does not satisfy k-anonymity since there are two unique records with the same QID values. In this case an adversary with foreknowledge of the existence of an Orleans Male with a Master's degree in the dataset will be able to deduce the individual's income. The two records violating k-anonymity are highlighted in Table 1.

Education	Gender	City	Income
Grade 12	Female	Nepean	\$65,000
Bachelor's	Male	Ottawa	\$50,000
Master's	Male	Orleans	\$50,000
PhD	Male	Gloucester	\$100,000
Grade 12	Female	Nepean	\$80,000
Associate	Female	Kanata	\$90,000
Associate	Female	Kanata	\$105,000
Bachelor's	Male	Ottawa	\$50,000

Table 1: Dataset violating k-anonymity

Figure 1 represents a taxonomy tree for *Education* column. The leaf nodes in the tree represent the distinct values present in the dataset. Taxonomy trees are provided by either the data provider or the data recipient for all QID in the dataset. The root node of all taxonomy trees is *Any*.

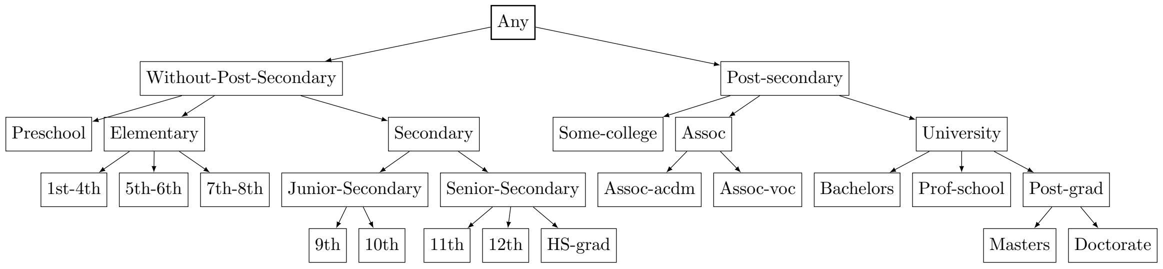


Figure 1: Education Taxonomy Tree

4.2 Introduction to Top Down Specialization

Top Down Specialization algorithm begins by removing all non QIDs from the dataset. The resulting dataset is then grouped by QID and SA and the count is calculated for every group. A set of trees called Anonymization Cuts $\{AC\}$ is created where it initially contains the taxonomy trees of all QID. The AC represents the level of the taxonomy tree to which each QID value will be generalized. The algorithm starts with generalizing all values to the root of the corresponding AC. The basic idea of the algorithm is that it starts from the top of the taxonomy tree and specializes the values until k is violated. However, it does not simply specialize all attributes. It calculates a score for every AC and only specializes the AC with the highest score. The score which is shown in Equation 1 calculates the information gain per privacy loss for every AC. In other words, which AC will provide the best information gain and the least privacy loss. Information gain involves calculating the entropy as well as counting all the values that get generalized to the AC's root ($|R_\nu|$), the count of all values that get generalized to the AC's root's children ($|R_{\nu c}|$), the count of all values that get generalized to the root for each value in $\{SA\}$ and finally the count of all values that get generalized to AC's root's children for each value in $\{SA\}$. The whole equation is shown in Equation 2 below. The privacy loss involves calculating k before specialization minus k after specialization. The reason for this is that as we specialize values along the AC trees, the data becomes more useful yet less private. The privacy loss equation is shown in Equation 4. Once the score is calculated, the root AC with the highest score is removed from $\{AC\}$ and its children are added as separate trees to the set of $\{AC\}$. That is, AC tree is specialized one level lower. The algorithm re-iterates with the new set of $\{AC\}$ until k is violated. The values in the dataset are finally generalized to the root of the final $\{AC\}$ set and that represents the anonymized dataset.

$$Score(\nu) = \begin{cases} \frac{InfoGain(\nu)}{PrivacyLoss(\nu)} & PrivacyLoss(\nu) \neq 0 \\ InfoGain(\nu) & otherwise \end{cases} \tag{1}$$

$$InfoGain(\nu) = I(R_\nu) - \sum_{c \in children(\nu)} \frac{|R_{\nu c}|}{|R_\nu|} I(R_{\nu c}) \tag{2}$$

$$I(R_\nu) = -\sum_{sv \in \{SA\}} \frac{|R_{\nu sv}|}{|R_\nu|} \times \log_2 \frac{|R_{\nu sv}|}{|R_\nu|} \tag{3}$$

$$PrivacyLoss(\nu) = |R_\nu| - \min(\{|R_{\nu c}|\}) \tag{4}$$

Education	Gender	Age	Income	count
9th	M	30	≤ 50k	3
10th	M	32	≤ 50k	4
11th	M	35	> 50k	2
11th	M	35	≤ 50k	3
12th	F	37	> 50k	3
12th	F	37	≤ 50k	1
Bachelors	F	42	> 50k	4
Bachelors	F	42	≤ 50k	2
Bachelors	F	44	> 50k	4
Masters	M	44	> 50k	4
Masters	F	44	> 50k	3
Doctorate	F	44	> 50k	1

Table 2: Sample preprocessed dataset

Fung et. al provided an example in [4] which helps illustrate the calculations. I include it here for completeness. Using Table 2 as our dataset, *SA* in this case is the *Income* attribute. *AC* element to score is the *Education* taxonomy tree shown in Figure 1 with *Any* at its root. Calculating the score of this particular *AC* is shown below:

Operand	Description	Value
$\lvert R_\nu \rvert$	Sum of counts of *Education* attribute when generalized to *Any*	34
$\lvert R_{\nu sv} \rvert$	Sum of counts of *Education* attribute when generalized to *Any* and *SA* is $> 50k$	21
$\lvert R_{\nu sv} \rvert$	Sum of counts of *Education* attribute when generalized to *Any* and *SA* is $\leq 50k$	13
$\lvert R_{\nu c} \rvert$	Sum of counts of *Education* attribute when generalized to *Without-Post-Secondary*	16
$\lvert R_{\nu c} \rvert$	Sum of counts of *Education* attribute when generalized to *Post-Secondary*	18
$\lvert R_{\nu csv} \rvert$	Sum of counts of *Education* attribute when generalized to *Without-Post-Secondary* and *SA* is $> 50k$	5
$\lvert R_{\nu csv} \rvert$	Sum of counts of *Education* attribute when generalized to *Without-Post-Secondary* and *SA* is $\leq 50k$	11
$\lvert R_{\nu csv} \rvert$	Sum of counts of *Education* attribute when generalized to *Post-Secondary* and *SA* is $> 50k$	16
$\lvert R_{\nu csv} \rvert$	Sum of counts of *Education* attribute when generalized to *Post-Secondary* and *SA* is $\leq 50k$	2
$\lvert R_\nu \rvert$	Anonymity before specialization	34
$min(\{\lvert R_{\nu c} \rvert\})$	Anonymity after specialization	16

Plugging these values in Equations 1 through 4 we get:

$$I(R_{Any\_Edu}) = (-\frac{21}{34} \times \log_2 \frac{21}{34}) + (-\frac{13}{34} \times \log_2 \frac{13}{34}) = 0.9597$$

$$I(R_{Without-Post-Secondary\_Edu}) = (-\frac{5}{16} \times \log_2 \frac{5}{16}) + (-\frac{11}{16} \times \log_2 \frac{11}{16}) = 0.8960$$

$$I(R_{Post-Secondary\_Edu}) = (-\frac{16}{18} \times \log_2 \frac{16}{18}) + (-\frac{2}{18} \times \log_2 \frac{2}{18}) = 0.5033$$

$$InfoGain(Any\_Edu) = 0.9597 - (\frac{16}{34} \times 0.8960 + \frac{18}{34} \times 0.5033) = 0.2716$$

$$PrivacyLoss(Any\_Edu) = 34 - 16 = 18$$

$$Score(Any\_Edu) = \frac{0.2716}{18} = 0.0151$$

Once the calculations are done for every tree in AC, let's say *Education_Any* has the highest score, then *Any* is removed from AC set and its subtree rooted at *Without-Post-Secondary* is added along with the subtree rooted at *Post-Secondary* so that the subsequent iteration calculates the scores of these two subtrees along with the rest of the AC set.

4.3 Pre-Processing

As illustrated by Section 4.2, Top-Down Specialization is an iterative algorithm that involves using the same dataset multiple times to perform different calculations. This type of algorithms is best suited for Apache Spark^{TM} as shown in Section 2. Spark is a fast and general-purpose cluster computing system that performs its computations in memory by default. In order to prepare our dataset for the main algorithm, first we need to remove all non-QID columns from the dataset. We then perform a group by query for $\{QID\} \cup \{SA\}$ with the count of every group as shown in the example in Table 2. Since there are multiple iterations that involve generalizing values to the root of the trees in AC, we ultimately need to access the cell value's root in $\mathcal{O}(1)$ time since this will be performed on every cell in the dataset. Therefore, I present an algorithm that runs during the preprocessing stage to build path maps from taxonomy trees.

Algorithm 1: Building Parent Child Mapping

```
Input: children, parentQueue
Output: parentChildMap
1  Function Traverse(children, parentQueue):
2      currentNode ← children[0];
3      currentParent ← Dequeue(parentQueue);
4      parentChildMap ← parentChildMap + ($currentNode$ : $currentParent$);
5      remainder ← children[1] to children[$length - 1$];
6      if IsEmpty(children) then
           /* No more children to traverse, we are done                 */
7          return parentChildMap;
8      if IsLeaf(currentNode) then
9          return Traverse(remainder, parentQueue);
10     else
11         currentNodeChildren ← GetChildren(currentNode);
12         foreach child c in currentNodeChildren do Enqueue(currentNode);
13         remainder ← remainder + currentNodeChildren;
14         return Traverse(remainder, parentQueue);
```

First, we need to build a parent-child mapping by traversing the taxonomy trees in its entirety in a breadth-first tail-recursive manner as shown in Algorithm 1. The Top-Down Specialization algorithm implemented for this paper is in Scala since it is the native language of Spark. Tail recursion is a feature in Scala which provides a way to calculate the intermediate results so that the stack does not dramatically increase in size by function calls as it happens in traditional recursive algorithms. The *Traverse* function traverses through every node in the tree. If a node is not a leaf node, it adds the current node to a queue as many times as its number of children. For example, when we are traversing the node *Any* in Figure 1, we see that it has two children, therefore after Line 12, *parentQueue* will contain [*Any*, *Any*]. This way, when we traverse *Without-Post-Secondary* and *Post-Secondary* we dequeue *Any* twice, and *parentChildMap* will contain two elements: [{*key*: *Without-Post-Secondary*, *value*: *Any*}, {*key*: *Post-Secondary*, *value*: *Any*}]

Algorithm 2: Get Path of a Given Node

```
Input: pathMap, node, currentPath
Output: path
1 Function GetPath(pathMap, node, currentPath):
2     currentParent ← Get(pathMap, node);
3     currentPath ← currentPath + node;
4     if currentParent is null then
5         return currentPath;
6     else
7         return GetPath(pathMap, node, currentPath);
```

Once we have our full parent-child mapping for every node in the tree, we can then tail-recursively build the full path for a given node as shown in Algorithm 2.

Algorithm 3: Build Path Map from Taxonomy Tree

```
Input: taxonomyTree
Output: fullPathMap
1 children ← GetChildren(taxonomyTree);
2 parentQueue ← foreach node n in children do Enqueue(node);
3 parentChildMapping ← Traverse(children, parentQueue);
4 currentPath ← $\emptyset$;
5 fullPathMap ← $\emptyset$;
6 for key $\in$ parentChildMapping do
7     path ← GetPath(parentChildMapping, key, currentPath);
8     fullPathMap ← fullPathMap + (key: Reverse(path));
9 return fullPathMap
```

Finally, we now have the necessary blocks to build a full path map from a taxonomy tree as shown in Algorithm 3. From the parent-child map we get the full path. We then update our *fullPathMap* with the node as the key and the reversed path as the value. The reason we reverse the path is that we need to generalize to the root of the tree. After Algorithm 3 runs, the map for node *9th* in Figure 1 will look like {*key*: *9th*, *value*: [*Any, Without-Post-Secondary, Secondary, Junior-Secondary, 9th*]}. Therefore, getting the root of a node is in constant time since the path map is indexed by the node's string value.

4.4 Main Algorithm

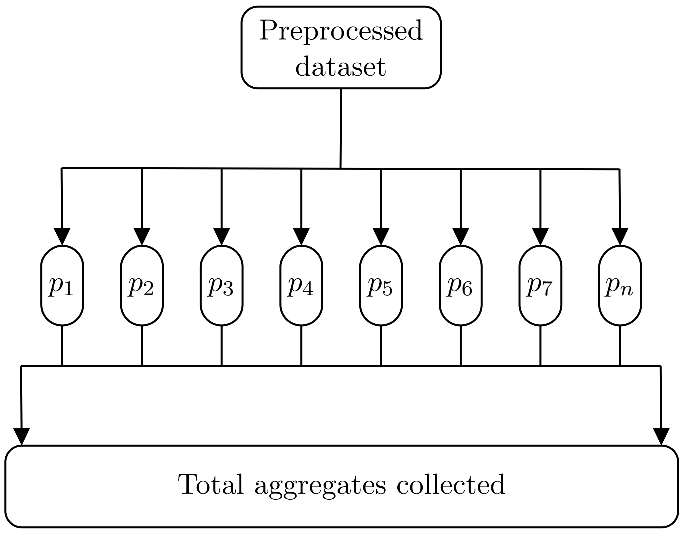


Figure 2: Performing Aggregations on Spark

Once the preprocessing is done, we are now ready to proceed with the actual anonymization. We start by partitioning the preprocessed dataset into multiple partitions as shown in Figure 2. For every partition, we transform the dataset by creating a column for every operand required in the score calculation equations. For example, we transform Table 2 to Table 3 shown below. The columns with _Y and _N postfixes represent those with $SA > 50k$ and $SA \leq 50k$ respectively. The _WPS postfix represents count of values that get generalized to Without-Post-Secondary node and _PS for Post-Secondary. This transformation is done for all QID on all worker nodes in parallel however due to the large width of the resulting dataset I show the example for *Education* attribute only.

Edu	Edu_Gen	Edu_Child_Gen	Edu_Any	Edu_Any_Y	Edu_Any_N	Edu_WPS	Edu_PS	Edu_WPS_Y	Edu_WPS_N	Edu_PS_Y	Edu_PS_N
9th	Any	WPS	3	0	3	3	0	0	3	0	0
10th	Any	WPS	4	0	4	4	0	0	4	0	0
11th	Any	WPS	5	2	3	5	0	2	3	0	0
12th	Any	WPS	4	3	1	4	0	3	1	0	0
Bachelors	Any	PS	6	4	2	0	6	0	0	4	2
Bachelors	Any	PS	4	4	0	0	4	0	0	4	0
Masters	Any	PS	4	4	0	0	4	0	0	4	0
Masters	Any	PS	3	3	0	0	3	0	0	3	0
Doctorate	Any	PS	1	1	0	0	1	0	0	1	0

Table 3: Education Transformation

Once we have all the transformations, we can then perform the aggregations that will provide us with the total numbers to be used in the score calculation equations. The aggregation dataset is just one row with all the aggregates as shown in Table 4. The values for the aggregates correspond to the same values in the example provided in Section 4.2. Spark performs these aggregations in parallel for every partition. The aggregation is followed by a *collect* call which triggers Spark to collect all the aggregates from the worker nodes into the master node which then gets stored in a local variable representing Table 4. The scores are then calculated using the local variable. The highest scoring AC (Anonymization Cut) tree is selected, its root is removed from the AC set, and its children's subtrees are added.

Algorithm 4: Parallel Anonymization

```
Input: originalPathMap, newPathMap, AC, partitionedDataset, k
Output: anonymizedCuts
1  Function Anonymize(originalPathMap, newPathMap, AC, k):
2      transformedPartitionedDataset ← Transform(QIDs, partitionedDataset);
3      bestScoreAC ← FindBestScore(transformedPartitionedDataset);
4      bestScoreChildren ← GetChildren(bestScoreAC);
5      newAC ← $AC - bestScoreAC + bestScoreChildren$;
6      originalMap ← originalPathMap;
7      newMap ← Dequeue(bestScoreAC);
8      generalizedDataset ← Generalize(partitionedDataset, newPathMap);
9      kCurrent ← CalculateK(generalizedDataset);
10     if $kCurrent > k$ then
           /* too general, repeat                                      */
11         return Anonymize(originalMap, newMap, newAC)
12     else if $kCurrent < k$ then
           /* violated k, return map before generalization             */
13         return originalPathMap;
14     else
15         return newMap;
```

The dataset is then generalized to the root of all respective *AC* trees, *k* is calculated, and the iteration continues until *k* is violated. The main algorithm can be summarized in Algorithm 4 above.

Edu_Any	Edu_Any_Y	Edu_Any_N	Edu_WPS	Edu_PS	Edu_WPS_Y	Edu_WPS_N	Edu_PS_Y	Edu_PS_N
34	21	13	16	18	5	11	16	2

Table 4: Education Aggregation

The path map that gets returned from Line 13 or 15 in Algorithm 4 once it is done can look as follows:

```
education: Map(
        7th-8th: Queue(Elementary, 7th-8th),
        Bachelors: Queue(University, Bachelors), ...
    ),
native-country: Map(
        Vietnam: Queue(Asia, Southeastern-Asia, Vietnam), ...
    ),
...
```

This anonymization map can then be passed to a Spark User Defined Function (UDF) that gets executed on every cell. For education column it will generalize *7th-8th* values to *Elementary* and *Bachelors* to *University*. For native country column, *Vietnam* values will be generalized to *Asia*.

4.5 Performance enhancements

Multiple considerations have been taken into account in order to improve performance. The first was generating path maps from taxonomy trees in the preprocessing step in order to speed up tree lookups performed during generalization. I also had to perform multiple experiments with a variety of different partition numbers in order to achieve the best performance. The best performance was achieved when number of partitions was set to total number of cores in the cluster. This finding corroborates findings of [16]. Tail recursion was also preferred over looping as a performance enhancement technique. Dataset was also partitioned over an arbitrary unique $rowId$ assigned to every row in the preprocessing stage. When partitioning is left completely up to Spark by only providing the number of partitions, it was found that this does not guarantee an equally distributed number of partitions across all worker nodes. An even distribution was achieved by partitioning over a unique ID column as well as providing the number of partitions.

Lastly, the biggest improvement enhancement was achieved by performing the aggregations of all taxonomy trees at once, in parallel, and then collecting the total aggregates to a local variable. The brute-force attempt of the implementation was only performing the aggregations of individual QIDs in parallel and then collecting aggregates for each QID separately. By doing all aggregations of all taxonomy trees in parallel, which essentially reduced aggregations and collect calls to only one per iteration, runtime was reduced from 4 hours to 15 minutes on a 5-million row dataset in an 8-node cluster. In Section 5 I review those performance results in detail.

5 Experimental Evaluation

Experiments were performed on an OpenStack cluster with 1, 2, 4, 8 and 16 worker nodes. Each node had 4 vCPUs, 8 GB RAM and 32 GB disk size. All nodes were running Spark version 2.4.2, Scala 12.10 and Java 8.

The dataset used is the same *Adult* dataset referenced in [16] however it was enlarged to 4 different datasets of 250,000, 5 million, 10 million and 20 million-row datasets. The enlargement process involved writing a program to inject rows using a random value from the list of distinct values in the original dataset for every column.

A total of 8 categorical QIDs were specified: *education*, *marital_status*, *occupation*, *native_country*, *workclass*, *relationship*, *race* and *sex*. The k value was set to 100.

The first experiment was determining the number of partitions. Using a 5 million-row dataset and running on an 8-node cluster, the following variations were tested for number of partitions: 16, 32 (total number of vCPUs in the cluster), 50, 100, 200 (Spark's default number of partitions), 300, 400 and 528.

As shown in Figure 3, the best performance was achieved when the number of partitions was set to the total number of vCPUs in the cluster. The dashed vertical line represents the best performing and the dotted represents Spark's default. The most likely explanation for this result is that when partitions are fewer than number of CPUs, not all cores are utilized. While when the number of partitions is much higher than number of cores, the partitioned dataset becomes too small to the point that the overhead of partitioning and scheduling the tasks exceed the time it takes to execute the task itself.

The following experiment evaluated the scale-up of Top-Down Specialization on Spark. The algorithm was executed using 3 dataset sizes on a 16-node cluster and 64 partitions.

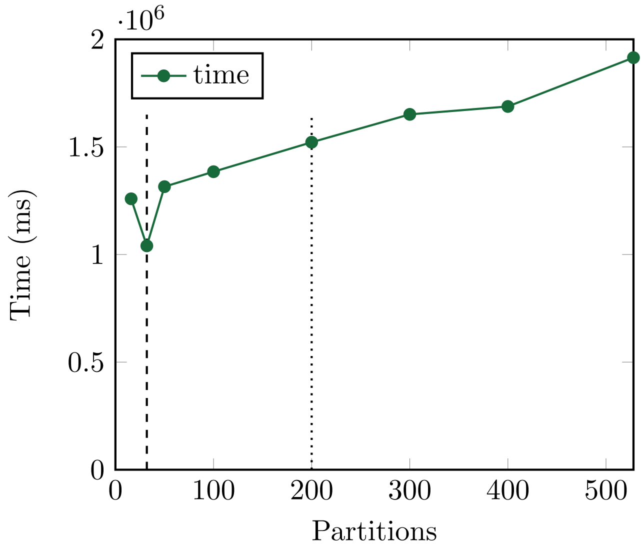


Figure 3: Determining Best Number of Partitions

When the dataset size increased by 100% from 5 to 10 to 20 million rows, the runtime only increased by 55-65%. This result is shown in Figure 4.

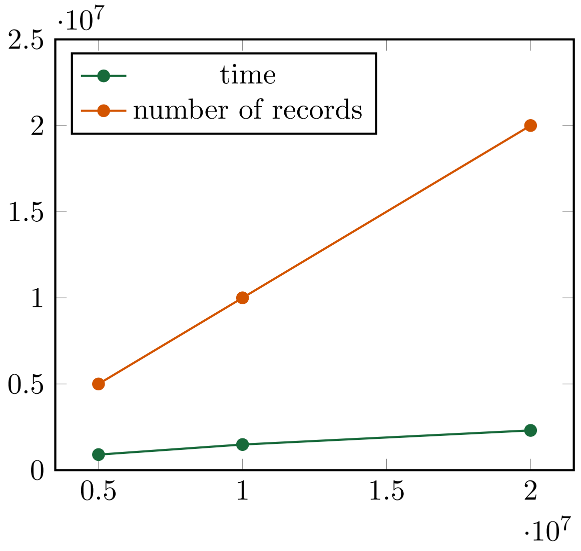


Figure 4: Scale-up of TDS on Spark

The next experiment assessed the speedup of the algorithm. Speed up experiments were conducted using four different dataset sizes: 250 thousand rows as well as 5, 10 and 20-million rows. As shown in Figure 5, the speedup significantly improved as the size of the dataset became larger. The 20-million row dataset provided the closest speedup to optimal. However, for the 250-thousand row dataset, increasing the number of nodes beyond 8 did not make a difference since the dataset was too small to be partitioned further. This is also most likely attributable to the assumption mentioned before where the overhead from partitioning the dataset and scheduling the task exceeded the computation cost of the task itself.

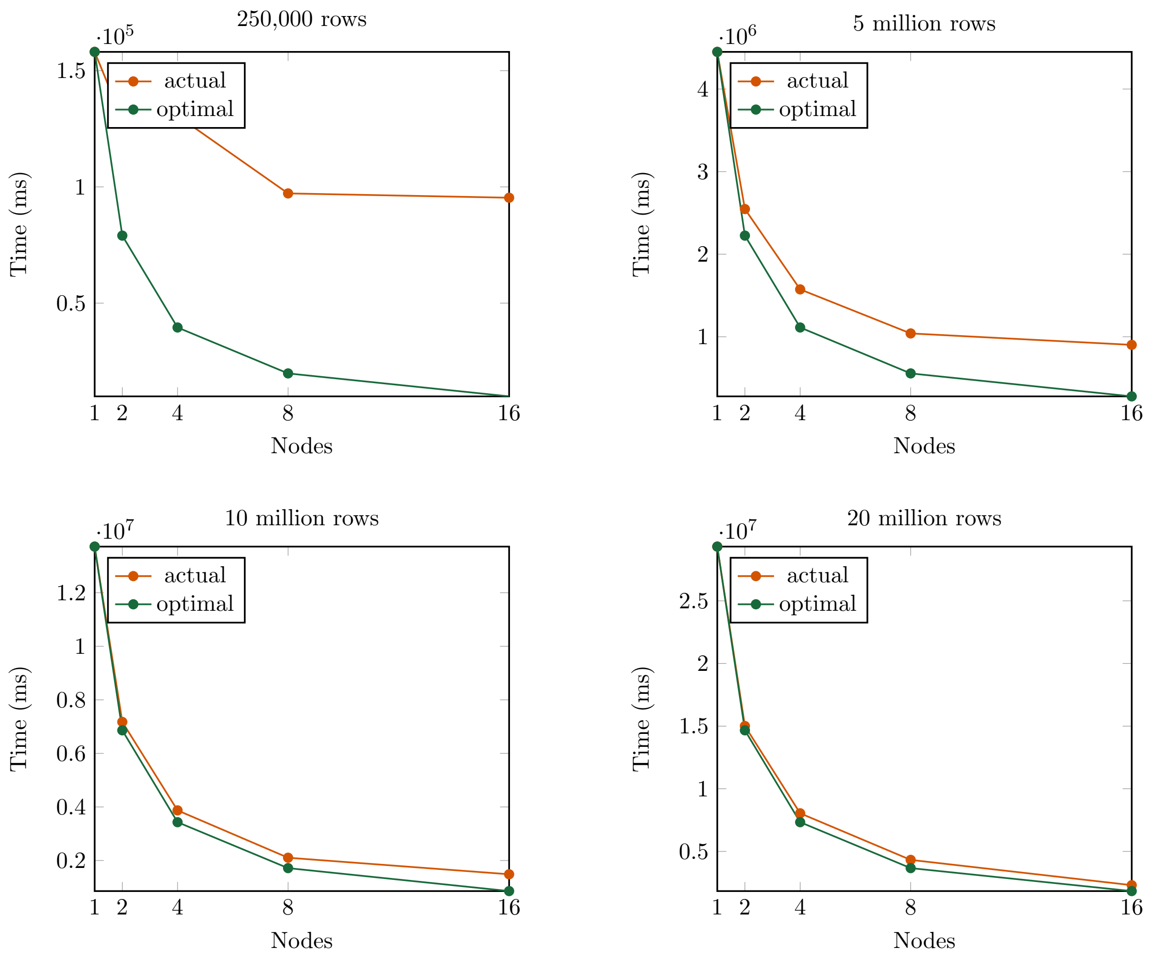


Figure 5: Speed-up curves for TDS

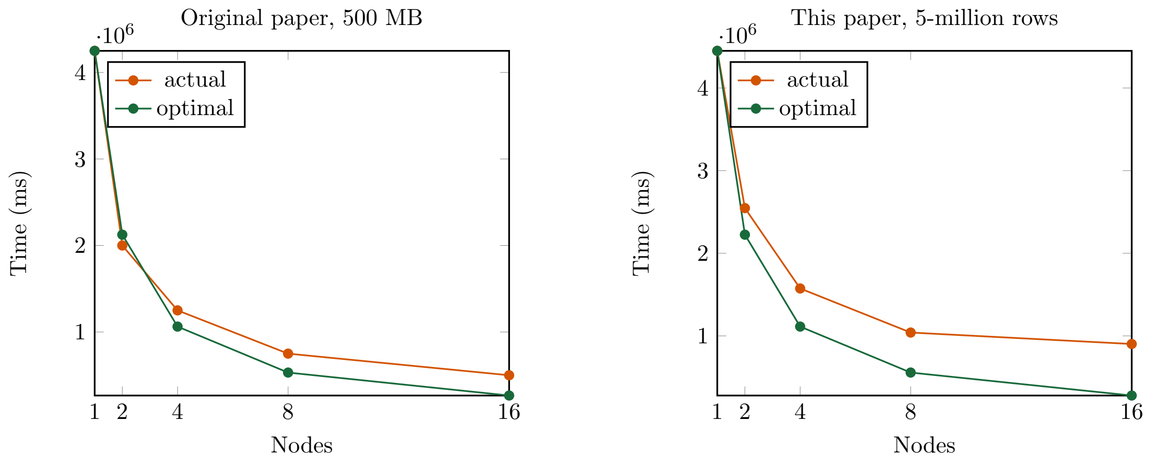
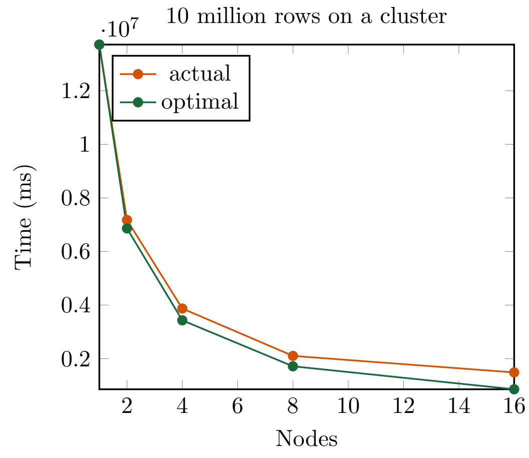


Figure 6: Comparison with original paper

In comparison with the original paper [16], we can see in Figure 6 that their speedup was slightly better for 5-million row dataset. Note that [16] used size on disk for measurement which was equivalent to this paper's 5-million row dataset. Also note that the test environment used in the original paper was slightly different as the authors installed Spark on Docker containers on the same machine. It is presumed that the communication between Docker images on the same machine will not suffer from the communication overhead between workers in a cluster where this paper's implementation was evaluated.

Finally, a test was performed on a 32-core machine and 128 GB RAM for the 10-million row dataset. As shown in Figure 7, the 10-million row running on a cluster had a much better speedup compared to the same dataset running on a 32-core machine. The most likely reason for this is as the dataset becomes larger, and the number of cores increases, the bus communication traffic becomes congested which serves as a bottleneck not present in a cluster environment. Spark documentation also shows that running locally on the same machine is meant for testing purposes only and some out-of-the-box Spark performance enhancements are only available for cluster environments.



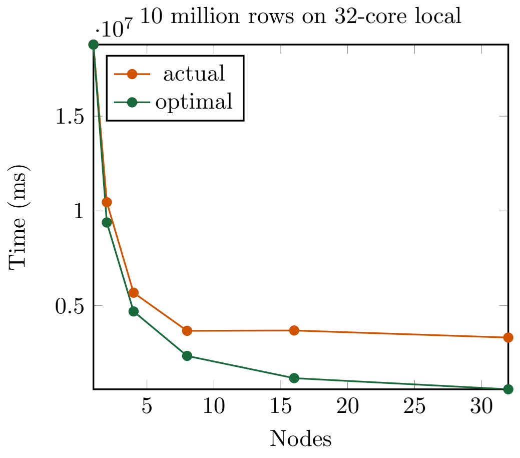


Figure 7: Comparison with 32-core local

6 Conclusions

This paper's implementation had speedups close to optimal for larger datasets of 20-million rows. The scale up curve was also virtually flat; when dataset size doubled, runtime only increased by half. In order to achieve the best performance, Spark partitions need to equal total number of cores in a cluster. Top-Down Specialization on Spark also performed better in a cluster environment compared to local machines even when locally the cores were more powerful than on the cluster. For iterative algorithms, minimizing the number of aggregations and collection from worker nodes to only one yields the best performance results. Even when the aggregations are performed on wide tables of $\approx$100 columns.

6.1 Summary of Contributions

This paper enhanced the preprocessing stage by implementing an algorithm that built path maps from taxonomy trees. This provided the option of performing look-ups in constant time during the iterations. It also evaluated the performance of Top-Down Specialization algorithm on Spark on datasets up to 4-times larger than tested in [16]. Evaluations of the algorithm were also made in a cluster environment which is more typical for production applications.

6.2 Future Research

The Top-Down Specialization algorithm starts by generalizing all values to the root of taxonomy trees. This introduces redundant iterations for larger datasets as it will iterate all the way from $k = n$, where n is the number of records, to a much lower number such as $k = 100$. Future research can enhance algorithms such as Hybrid Top-Down and Bottom-Up algorithm proposed in [17] by adapting it for Spark so that not all records get generalized to the root before starting specialization. It is also worth it to research starting the generalization at a level deeper than the root to avoid these redundant iterations.

References

[1] Ruan Chun Al-Zobbi Mohammed, Shahrestani Seyed. Experimenting sensitivity-based anonymization framework in apache spark. *Journal of Big Data*, 5(1):38, October 2018.

[2] Y. Canbay and S. Sağıroğlu. Big data anonymization with spark. In *2017 International Conference on Computer Science and Engineering (UBMK)*, pages 833–838, October 2017.

[3] A. Chakravorty, C. Rong, K. R. Jayaram, and S. Tao. Scalable, efficient anonymization with incognito - framework algorithm. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 39–48, June 2017.

[4] B. C. M. Fung, K. Wang, and P. S. Yu. Top-down specialization for information and privacy preservation. In *21st International Conference on Data Engineering (ICDE'05)*, pages 205–216, April 2005.

[5] A. Hoang, M. Tran, A. Duong, and I. Echizen. An indexed bottom-up approach for publishing anonymized data. In *2012 Eighth International Conference on Computational Intelligence and Security*, pages 641–645, November 2012.

[6] Ke Wang, P. S. Yu, and S. Chakraborty. Bottom-up generalization: a data mining solution to privacy protection. In *Fourth IEEE International Conference on Data Mining (ICDM'04)*, pages 249–256, November 2004.

[7] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 106–115, April 2007.

[8] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007.

[9] N. Memon, G. Loukides, and J. Shao. A parallel method for scalable anonymization of transaction data. In *2015 14th International Symposium on Parallel and Distributed Computing*, pages 235–241, June 2015.

[10] Adam Meyerson and Ryan Williams. On the complexity of optimal k-anonymity. In *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '04, pages 223–228, New York, NY, USA, 2004. ACM.

[11] K Pandilakshmi and G Rashitha Banu. An advanced bottom up generalization approach for big data on cloud. *Int J Comput Algor*, 3:1054–9, 2014.

[12] Pooja Parameshwarappa, Zhiyuan Chen, and Gunes Koru. A multi-level clustering approach for anonymizing large-scale physical activity data. *arXiv*, 2019.

[13] Zorige Priyanka, K Nagaraju, and Y Venkateswarlu. Data anonymization using map reduce on cloud based a scalable two-phase top-down specialization. *International Journal on Recent and Innovation Trends in Computing and Communication*, 2(12):3879–3883, 2014.

[14] Sweeney L. Samarati P. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and supression. Technical report, Massachusetts Institute of Technology and SRI International, 1998.

[15] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proc. VLDB Endow.*, 8(13):2110–2121, September 2015.

[16] U. Sopaoglu and O. Abul. A top-down k-anonymization implementation for apache spark. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 4513–4521, December 2017.

[17] X. Zhang, C. Liu, S. Nepal, C. Yang, W. Dou, and J. Chen. Combining top-down and bottom-up: Scalable sub-tree anonymization over big data using mapreduce on cloud. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 501–508, July 2013.

[18] X. Zhang, L. T. Yang, C. Liu, and J. Chen. A scalable two-phase top-down specialization approach for data anonymization using mapreduce on cloud. *IEEE Transactions on Parallel and Distributed Systems*, 25(2):363–373, February 2014.