Comparison between MapReduce and Spark in Processing Sensitivity-based Anonymization Framework

One of the biggest concerns of big data and analytics is privacy. It is believed that the forthcoming frameworks and theories will establish several solutions for the privacy protection. One of the known solutions is the k-anonymity that was introduced for traditional data. Recently, two major frameworks leveraged big data processing and applications; these are MapReduce and Spark. Spark data processing has been attracting more attention due to its crucial impacts on a wide range of big data applications. One of the predominant big data applications is data analytics and anonymization. The previous chapters discussed a complete framework for data anonymization. The anonymization framework was merely implemented for MapReduce processing framework. This chapter will set-up a comparison between MapReduce and Spark in processing performance of data anonymization. Spark is a fast processing framework that was implemented in several applications such as; SQL, multimedia, and data stream. The main focus of this chapter is the SQL Spark, which is adequate for big data anonymization. Since Spark operates in-memory platform, we need to observe its limitations, speed, and fault tolerance on data size increase. Spark introduces an abstraction called resilient distributed datasets, which reads and serializes a collection of objects partitioned across a set of machines. Developers claim that Spark can outperform MapReduce by 10 times in iterative machine learning jobs. The experiments in this chapter compare between MapReduce and Spark. The overall results show a better performance for Spark’s processing time. However, in some limited cases, we prefer to implement the old MapReduce framework, when the cluster resources are limited and the network is non-congested [1].

# Analytics in Big Data

Big data evolution has formed new software tools and techniques to accelerate the processing speed, and increase the scalability. The new tools targeted many big data applications such as data analytics. The analytics has manifested some security concerns, as a reason for big data publicity prominence. In general, big data is beneficial when it is public, this means many organizations from different fields need to access this data for multiple purposes [2]. They all analyze, mine, and output statistical results. However, exposing any private data to public view carries a high-security risk. Personal re-identification is the main focus of researchers since decades. In data analytics, adversaries can easily re-identify and violate some private information. The information may contain sensitive information about patients, bank agents, census, or any other private information [3].

Anonymization methods, based on k-Anonymity, have been widely employed to prevent data re-identification [4]. Anonymization methods fall into two broad categories. The first category constitutes of techniques that generalize data from the bottom of the taxonomy tree towards its top and are referred to as the Bottom-Up Generalization (BUG) [5]. The second one is based on walking through the taxonomy tree from the top towards the bottom, known as the Top-Down Specialization (TDS) [6]. TDS and BUG methods were mainly developed for traditional data. Therefore, researchers upgraded the old methods to suit the new operations of big data. The operations should consider the parallel and distributed processing steps. Various methods of anonymization were specifically designed for parallel distributed processing. However, most resolutions fall short of a proper parallelization capability. The reason for this is further explained in section 7.6. Some of these recently developed anonymization methods are; Parallel BUG [7], Hybrid BUG and TDS [8], and Two-Phase TDS [9].

The previously mentioned methods focused on developing algorithms to find the best cut of the taxonomy tree, the optimal values of k, and/or the best option of anonymization technique either by TDS or by BUG. Implementing these algorithms may require high computation costs of continuous iterations with conditional statements, which means multiple times of heavy scan for the whole data records. However, these algorithms have ignored two main facts about big data processing; firstly, the key success factors of parallel processing is a proper parallelization algorithm [10]. This can be achieved by reducing the iteration to the minimal possible level. This is essential to avoid multiple scans for large data records. The major concern is not only the time consumption, but the unexpected failure that arbitrarily occurs during big data processing. Secondly, the contemporary methods have ignored the changes that occur in data growth. With the increased number of data records, data gain more similarity in attribute’s values. This is apparent in our life’s activities. For instance, if we sit in a data hall with 100 people, and the probability of finding a person’s age=33 is 10%, then the probability of finding the same age may go up to 20%, If the data hall contains 1000 people. This is because people’s age range is between [0 - 100], so more people will definitely increase the value equivalency. This concept was discussed in section 3.1.1.

The previously mentioned facts are actually essential to understanding big data nature and specifications. Applying heavy computation to a certain group of data records to find out the best anonymization cut, or even to decide which attribute that we need to anonymize, is inadequate. In big data, applying such accurate techniques may not affect the final results of statistical output. We even may ignore the small statistical value of small decimal results. The statistical results follow the principle of estimation prospect, which gives data miners a flexibility of approximating and rounding some numbers [11]. Therefore, pre-calculating the k value, and pre-determining the attributes needed to be anonymized is an advantage. Generally, this non-accuracy will not dramatically affect the data analytics results.

MDSBA is an anonymization method that uses BUG in *k-anonymity* that can address the previous two facts about big data. The main aim of this method is to improve the anonymization performance and to increase the usefulness of anonymized data. MDSBA is not only an anonymization algorithm or technique, but it provides a fine-grained access control for multi-level of user’s permissions.

In this chapter, experiments are conducted by using two different big data analytics tools, MapReduce and Spark. Anonymization operations will apply *k-anonymity* in MapReduce framework and compare it with Spark. Spark is an in-memory cluster computing framework for processing and analyzing large amounts of data. It exploits the increased size of hardware resources in CPU and RAM. Nowadays, Spark is the most popular processing framework for big data, by providing cost-effective and high scalable processes. MapReduce and Spark are both popular open source cluster computing frameworks. These frameworks are used in big data for large-scale data analytics, by applying parallel distributed processing tasks. Both frameworks provide programming API to users on managing major components of mapping, shuffling, execution, and caching [12].

# Spark Structure

Spark consists of masters and workers. The master uses the cluster manager to acquire the worker's resources. The cluster manager can be Mesos, YARN, or Spark standalone. Since Hadoop v2, YARN was developed with a wide range of flexibility, so it is able to collect information about the cluster resources and pass it to Spark, which in turn sends the command back to the cluster manager to create a certain number of JVMs or executers on each node [13].

In Spark two types of JVM are created, one is created on the master for Spark driver to contain the context operations, and the other type is created for each worker. Each worker may contain more than one JVM, and each JVM is known by a core. The worker executer may contain one or more cores. The number of cores on each worker can be determined automatically by the system, or statically configured by the user. Each Spark core is created by initiating a JVM, which reserves a specified size of memory space. The memory is reserved for; heap memory, cache, and JVM processes. Each core runs multi-task at the same time. The complete Spark structure is illustrated in Figure 7.1.

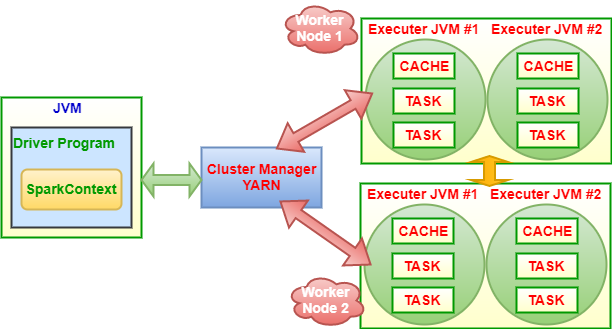


Figure 7.1. Spark structure in master and workers

Spark operates as JVMs in master and workers. The main connector between Spark master and workers is the cluster manager, such as YARN. On submitting a job by users, Spark in master, known as Driver Program, acquires resources from the cluster manager. The cluster manager collects information from workers, such as disk space, memory size, and processors. The driver Program splits the job into multiple tasks as per available resources. The small tasks are scheduled and executed accordingly. The execution is scheduled and divided into stages of Directed Acyclic Graph (DAG). The tasks are scheduled on the executors using a low-level scheduler provided by a cluster manager. The executors run the tasks submitted by Spark in parallel. To empower the fault tolerance structure, each core operates in an independent JVM to isolate the misbehaving JVM from the others. This structure is essential to reduce the dead nodes possibilities. However, this structure may become a cumbersome in some cases, where interaction between two processed data in two separate JVM’s is not possible without storing each executer data on the disk. This behavior degrades Spark efficiency and speed [14].

After processing each task, the results are either returned to the driver program JVM or proceeded to the shuffling. Every core can run multi-task concurrently. All Spark cores reserve similar partitions in the memory, in order to copy data blocks for execution. The size of partitions depends on the memory size. The number of executed tasks depends on the number of partitions created in each core. By default, Spark creates one task per data partition, therefore, programmers have options to decide the number of partitions on each task within the core. For a better parallelization, the number of partitions should consider many factors, such as the number of executers on each worker, the executer memory size, and the HDFS data block. Each partition reads one or more of data blocks for processing, so if HDFS data block is 128MB, while the number of partitions in the whole cluster is 32, then a data size of 8 GB will be split into 64 tasks. That is 32 partitions can load 32 blocks with 32 tasks, and each block is 128MB. Therefore, the data size of (8192 MB / 128 = 64). Moreover, more partitions can help the parallel processing of data. However, more partitions requires larger memory size [15].

Developers interact with Spark cluster through an API programming interface in multiple languages. Spark currently is available in Scala, Java, Python, and R. The API consists of SparkContext and Resilient Distributed Datasets (RDDs). The context is the main entry point for Spark functionality and cluster. Only one SparkContext may be active per master JVM per user, therefore, you must kill the active SparkContext process before creating a new one. The RDD represents the collection of partitions created in executers. Each partition contains some data blocks and cache. RDD is immutable or location unchangeable once data block is created on the memory. The data block or RDD is mapped to one or more of the data blocks on the disk. The RDD is fault tolerant, so if one node is dead, then similar RDD is created on another live node. The RDD contains a block of data types that can be a combination of integer, long, string and others. RDD creation and transformation operations are lazy. The lazy operations impose Spark not to immediately perform any computation when an application calls a method that returns an RDD. The operations of reading data, mapping, filtering, grouping, and others, are triggered by the action. Spark developers implement two types of methods, transformation and action. The transformation methods are noted by Spark of how an RDD was created. Thus, it maintains lineage information for each RDD. It uses this lineage information to construct or reconstruct an RDD when required. The action methods are the actual methods that perform real operations. Actions may include methods like count, foreach, max, min, save, print, and others. Action triggers a sequence of transformation methods in RDD [16].

RDD performs in a fault-tolerant fashion. On cluster installations, separate data partitions can be on separate nodes. Whenever a part of an RDD or an entire RDD is lost, the system is able to reconstruct the data of lost partitions by using lineage information. Lineage refers to the sequence of transformations used to produce the current RDD. As a result, Spark is able to recover automatically from most failures. Using the RDD handler, one can access all partitions and perform computations and transformations using the contained data. Spark supports a wider range of applications, from graph processing, SQL-like queries, and machine-learning libraries.

# MapReduce and Spark

Both frameworks are very similar in some core features. Spark runs on Hadoop, Mesos, or standalone, hence it is not possible to categorize Spark as a non-MapReduce framework. MapReduce core structure is YARN and Hadoop Distributed File System (HDFS). These two Hadoop native processes are intensively used in Spark. They provide reliability, performance, and scalability for Spark. It is worth mentioning that the distinguished difference between MapReduce and Spark is the processing methods. MapReduce wastes a considerable time on input/output transmission between memory and disks. The inefficiency of read/write from the disk and the high latency in each operation are the major inhibitors in MapReduce. On the contrast, Spark operations are executed over the built-in memory, and without a need for read/write on disks [1].

Before applying Spark on MDSBA, some concerns need to be verified about the efficiency of Spark in anonymizing data. Anonymization requires a large buffer size to accommodate the massive size of data flow. The comparison between MapReduce and Spark, tends to evaluate MDSBA approach’s performance in each framework. Also, it is essential to correlate between MDSBA architecture and Spark operation. This correlation measures the capability of MDSBA to operate on Spark framework. Moreover, all previous chapters on MDSBA were conducted in MapReduce environment, and within Hadoop ecosystems. Other frameworks have not been tested so far. There is a considerable number of cluster computation engines available in the market during the last decade. This includes; Hadoop, Spark, Flink, Storm, Heron, and Samza. Storm is a real-time processing engine that is able to stream SQL records from Hadoop Distributed File System (HDFS), from Kafka, or from MongoDB, or Redis [17]. We chose Spark over Storm for the following reasons: Storm is a popular framework worldwide, and it is implemented in enterprise company’s networks like Yahoo, Twitter, Flipboard, and yelp. It is written in Clojure Language, the Lisp-like functional language. Moreover, the structural design of Storm was built for the data stream, which means continuous infinitive processes to analyze data. Non-streamed data, or batch data, cannot be processed by infinitive processes. Anonymization methods are only applied to batch data. This is essential for grouping, counting the number of equivalent records and masking the non-equivalent. These operations cannot induce adequate anonymization results in the data stream. Hence, Storm is not preferable in k-anonymity methods [18].

Pig is an old Hadoop ecosystem that operates in MapReduce structure. Spark is a new framework that gained popularity for its speed and performance. Spark is much faster than Hadoop in performing operations. The speed is related to the in-memory processes. On the other hand, Hadoop inputs/outputs from/to the disk on starting or completing each map/reduce process. Hadoop reads data from the disk, executes in two stages of Map and Reduce, and returns output back to the disk. This data flow of in/out creates a high input/output latency, and keeps the scheduler busy with creating more tasks. This may make Spark faster than MapReduce, which may reach up to a hundred times faster. Speed is an important issue in big data for its large size. This new framework opens the door for various new applications that was not possible to develop with the old frameworks.

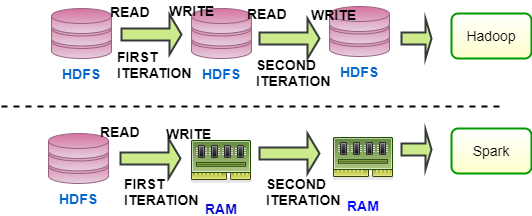


Figure 7.2. Comparison between Hadoop and Spark in dealing with memory and disks.

Figure 7.2 illustrates a comparison between Hadoop and Spark. Spark’s in-memory cluster computing capabilities are high, which boost performance, even with the large data magnitude. The time difference between reading data from disk or from memory is significant. Larger data size shows higher latency than smaller data size when reading from disks. Also, Spark implements caching technique to store data in memory to minimize the disk I/O. Figure 7.2 clearly describes the iterated stages in Hadoop operations. That is keeping two stages mapping or reading from the disk, and reducing or writing to the disk. A considerable time is consumed between memory/disk input/output. This two-stage principle is not available in Spark, instead, it map blocks from disks, processes them, keeps them in-memory for another staging process. Hence, several stages may save the disk I/O.

The second reason that makes Spark faster is the advanced job execution engine. Both Spark and MapReduce convert a job into a directed acyclic graph (DAG) of stages. The graph theory is an old theory that represents any graph with a collection of vertices connected by edges [19]. Graph theory was developed and implemented in many fields such as; computer science, and medical science. In Hadoop and Spark, graph theory is being used for scheduling tasks, with a similar technique to DAG. Each job is presented by a direct graph, which is a set of vertices connected by directed edges. Each vertex in the graph represents one processor, and each edge represents a communication link. The difference between Spark DAG and Hadoop DAG is the communication link. In Hadoop, each processor reads all incoming messages from the in-edge (disk), performs some computation, and writes messages to the out-edge (disk). The in-edge and out-edge are presented by disk I/O, which cause major delays. Hence, two stages can be processed at a time. In other words, there is no way you can start at some vertex in a DAG and follow a sequence of directed edges to get back to the same vertex [20]. Hadoop creates a DAG with only two pre-defined stages, map and reduce. Developers are forced to process their commands within one of these two stages. Based on this core structure, complex jobs need to be split into two or more jobs to fulfill the two-stage processes. In contrast, Spark can accomplish the job with multiple stages, and without splitting the single job into many sub jobs. This structure optimizes job scheduling and processing. For instance, if the processing algorithm contains, read from storage, filter, and group. Spark can accomplish this algorithm in one job with three stages of the map, shuffle, and reduce. In Hadoop, this will be divided into two jobs of (map, shuffle), and (map, reduce) for each job. Data will be read from the disk, filtered, and stored back to the disk with the first job. Again, data will be read from the disk, grouped and stored back to the disk.

However, some obstacles may degrade the efficiency of Spark. As mentioned earlier, if data size does not fit the total memory size of the cluster, then Spark uses disks to spill data. This action causes a higher delay on disk I/O, which is a similar delay reason in Hadoop framework. Therefore, Spark cluster always needs a large memory size. The average amount of processed data should be considered on building Spark cluster. This is because Spark has the capability of caching RDD in memory. Developers benefit from this feature to avoid disk I/O. However, caching features requires enough quite large memory size. For instance, if Spark cluster is expected to analyze data size of 3 TB, then Spark uses up to 60% of the total configured executor memory to cache RDDs. If data analyzers decided to cache half of the 3 TB in-memory at once, then there should be enough memory space to accommodate this large size of data. In this scenario, a Spark cluster with 20 workers and 128 GB memory for each worker is enough to cache 1.5 TB. A rough calculation can be. As noticed, Spark cluster infrastructure should consist of powerful servers and fast connectors between workers and storage units. Therefore, virtual machines connected to storage nodes by iSCSi or Fibre Channels are not the proper choice for building Spark cluster. Spark requires physical or non-virtual servers. Also, network connections between servers and storages should be through direct access connection (DAC), such as SAS or SATA.

Spark allows applications to cache data in memory for processing. This enables applications to minimize disk I/O. A MapReduce-based data processing pipeline may consist of a sequence of jobs, where each job reads data from disks, processes it, and writes the results back to disks. Since Spark allows caching of data in memory, the same application implemented with Spark reads data from disks only once. Once data is cached in memory, each subsequent operation can be performed directly on the cached data. Thus, Spark enables an application to minimize I/O latency, which can be a significant contributor to overall job execution time.

The (**persist**) or cache commands in Spark can be done automatically or by developers. Spark persists some intermediate data in some shuffle operations without users calling persist. This is done to avoid re-computing the entire input if a node fails during the shuffle. However, it is recommended for developers to call persist on the resulting RDD or during the map phase [21]. A common misconception is that Spark cannot be used if input data does not fit in memory. This is not true because Spark can process terabytes of data on a cluster that may have only 100 GB total cluster memory. However, this is not recommended for data anonymization by SQL Spark. The small memory size compared to the data size may degrade Spark operation’s performance. It is up to the developer to decide what data should be cached and at what point in a data processing pipeline. In fact, if a data processing application makes only a single pass over data, then there is no need for caching data at all. Developers may cache data, when data is read several times [3].

Data lifecycle is rotated between the network, disk, and memory. Data format and structure differ between disk storage, memory, and network transmission. The process of converting data to a storable format, in disks, is known by serialization, while releasing data in reverse structure, in memory, is known by de-serialization [22]. Data can be serialized using readable or binary format. The readable format may include CSV, XML, JSON, while the binary format may include multimedia and database. In Big data, implementing text file for storing and retrieving data in a text format is inefficient in terms of either storage space or parse time. The binary formats are more compact and can be parsed much quicker than text formats. However, in this paper we have limited resources in our conducted experiments, therefore, the dataset sizes we use are quite small, so there is no major impact regarding data type.

# Data Streaming versus Batch

Data should be stored in repository storage units. Data analytics operations fetch data from the repository and process them. Stored data may represent a production data or archived, where the production data is a live data with a real-time or close to real-time update. Archived data is an old data that is no longer actively used. From the analytics perspective, archived and backup data are similar. The difference between backup and archived data can be controversial, since the technical definition for each data type is slightly different. The archiving is applied to the original data, while the backup is a copy of the data. However, the aim is identifying the analytics operations on real-time and non-real time data. Regardless the originality of data, real-time data needs to be analyzed differently [23].

Different tools were designed for each data type. Data streaming tools fetch the storage location continuously to read the newly updated data. This data is collected in a temporary memory for further processing, hence, this operation runs indefinitely. On the other hand, data batching tools read all batched data only once, process them, and terminate after the process is completed. Streaming and batching tools vary in the core operation. Streaming needs to access the storage units on a regular basis to read the most recent records, with a small size of collected batches or records on each access. Hence, the operation latency should not cost more than few seconds or even milliseconds. Therefore, complex analytics cannot cope with the streaming principle.

Anonymity in data analytics is an example of complex analytics, where anonymization operations scan the data records many times during the; filtration, aggregation and masking operations. The anonymization latency is considerably high, therefore, data batching tools are more efficient to deal with the large data size, and long latency. Various big data tools were developed to accommodate both batching and streaming data. MapReduce framework, Hadoop, is unable to process the data streaming. The next generation was developed based on Lambda architecture, which is designed to handle both batching and streaming data. Lambda structure is based on a trade-off between latency, throughput, and fault-tolerance. The recently developed frameworks follow a similar structure of storing data temporarily in the memory, to reduce the disk input/output latency [24].

Many recent frameworks were developed based on Lambda architecture, such as; Spark, Storm, Flink, Samza, and others. Some of these frameworks can operate with stream and batch processing such as Spark. Some Lambda frameworks operate at the top of YARN, and read from HDFS. For this reason, Lambda tools can be considered as Hadoop ecosystems. Current available anonymization methods, including MDSBA, were specially implemented for batch processes. Implementing MDSBA in data streaming should follow different algorithms and techniques, which is left to the future work. So far, we can use big data framework for batch processing only. Hence, Spark is an appropriate choice over Samza, Flink, or Storm, since Spark can operate with batch processing.

# Implementing MDSBA in Spark

Spark has many advantages over Hadoop ecosystems. It mitigates latencies, and increases the performance. For instance, Pig divides jobs into small tasks, and for each task, Pig reads data from HDFS, and returns data back to HDFS once the process is completed. This in/out consumes a considerable time, and unlike Spark, which implements Resilient Distributed Dataset (RDD). RDD is the main distinguishing feature of Spark. RDD divides jobs into many DAG stages, and for each stage, Spark reprocesses RDD in memory without referring back to the disk. Spark may perform many times faster than MapReduce.

In this chapter, Pig and Spark performance are evaluated in MDSBA. To conduct the detailed evaluation, Spark’s transformation and functions, used in anonymization, should be properly defined. In this chapter, all Spark scripts are coded by Scala program. Some functions and transformations operate faster than the others. For example, inner **join** transformation commands may require a high computational cost. MDSBA was proposed for big data processing frameworks. Its core concept is applying optimized anonymization procedures and algorithms by splitting data into small tasks, so they can be parallelized among the cluster nodes.

In big data processing, the reduce phase is expensive since it involves data partitioning, data serialization and deserialization, data compression, and disk I/O. These operations require data transfer over the network to aggregate data over multiple nodes. Leveraging any application’s behavior should consider the size of data transferring between nodes during the reduce phase [15]. In data anonymization, reduce phase is presented by SQL grouping commands, which causes a high shuffling processes. In order to reduce the *groupBy* impact in the anonymization application, a filtration command is initiated to split data logically as per nominal values. This type of split reduces the performance degradation caused by shuffling process. This reduces the amount of shuffling among cluster nodes during the reduce stage.

The previous anonymization techniques, such as Bottom-Up Generalization (BUG) and Top-Down Specialization (TDS), were supposedly to work efficiently in the parallel distributed processing. Technically, anonymizing data with these algorithms may negatively create data overflow without considering the cluster resources and capabilities. For instance, Figure 7.3 illustrates a comparison between TDS and MDSBA algorithms. In TDS algorithm, the data flow may negatively affect parallelization. Firstly, grouping all records without filtering data are inefficient. This was experimentally approved in section 7.6. Secondly, the rest of operations should be implemented with the help of UDF. Spark and Pig are unable to run such intensive computations and conditional iterations. Eventually, the anonymization algorithm cannot complete the operations successfully without having more advanced methods. Scripting languages such as; Pig and Scala are not able to complete such intensive operations. Therefore, User-Defined Functions (UDF) are embedded in the script’s codes to execute the complicated operations. However, the operations of UDF are executed in a black box, and not related to Spark or Hadoop control and management. This is also true in Pig operations. UDF uses the resource of locally installed JVM, and does not process data within RDD. Therefore, implementing Spark with such an algorithm is inefficient.

In MDSBA, UDF was embedded in Spark and Pig scripts, with the minimal size of data processing. As explained in the next section, the anonymization is applied on fewer attributes at a time. This technique controls and minimizes the size of data flowing to the UDF. The UDF is executed in a local JVM beyond the source manager’s control. It is a memory pool located outside Spark JVMs. Therefore, there is a need to reduce the amount of data flowing outside Spark JVMs [13]. Moreover, the anonymized Q-ID in TDS is volatile. This means that the specialization is applied to different Q-ID attributes in each group of records. The chosen attribute, for specialization, is the one with the highest score value [6]. Calculating the highest attribute score for each group is an expensive computation process. On the contrast, in MDSBA the anonymized Q-ID is pre-determined based on the Q-ID probability. This saves a considerable amount of computation time. This solution may not provide the optimal anonymization for the pre-chosen Q-ID, by resulting in a higher percentage of anonymization loss. However, we may sacrifice some information gain to the benefit of data performance.

**Group** all records

Generalize all records

**UDF**

**UDF**

**Group** the highest QID probability

Anonymize the lowest QID prob.

**Group** the highest QID probability

**UDF**

YES

STOP

Anonymize the lowest QID prob.

If non-equivalent

**Filter** as per class value

If more cut is available

If non-equivalent

**Group** all records

NO

YES

Specialize the highest QID score by choosing the best cut

MDSBA

TDS

YES

Figure 7.3. Comparison between the anonymization algorithms of MDSBA and TDS

Figure 7.3 compares between TDS and MDSBA in anonymizing data. The TDS algorithm imposes various iteration on calculating the best cut and scores. This type of iteration is inefficient in big data, and even worse in using UDF, where the program executes the UDF code outside Spark framework. Furthermore, the UDF program needs to iterate large size of arrays. The UDF executes almost all anonymization process. Thus, there is no real benefit from the parallel distributed system. In contrast, MDSBA implements UDF with a limited data size flow. As shown in the figure, using UDF was reduced to the minimal operations. UDF reads only few data attributes to apply some masking operations. He data size flowing to the UDF is relatively small.

## 7.1. User Defined Function in MDSBA

MDSBA implements User Defined Functions (UDF) in different locations. This is essential for two main purposes; anonymizing and ungrouping. In anonymizing, three masking types of interval, taxonomy tree, and suppression are implemented. Figure 7.4 shows the algorithm for anonymizing any numerical group. In Scala, the group of objects can be a list or a sequence. In the shown algorithm in Figure 7.4, one attribute of a numerical type was used for anonymization. Minimizing the amount of data on accessing the UDF program is essential to reduce the processing cost and to avoid data overflow as described before.

It is hard to predict the failure of non-Spark JVM, but it is clear that we need to keep the data flow to the lowest level. For instance, JVM default installation may take up to 0.25 of the total memory. This size can be re-configured and enlarged if needed. If Spark worker memory is large enough to fit the data size, then the external JVM that handles the UDF may be able to handle up to 25% of data size located in Spark. The size of the UDF heap memory is not the only obstacle, but the complex iteration with several IF statements can be another cumbersome that degrades the data processes. MDSBA implements a swift algorithm to anonymize data with the minimal number of iterations.

Section 3.2.4.4 in chapter 3 described the interval masking algorithm in details. The algorithm principle is simple, which converts a numerical list of values into set of intervals. Each intervals consist of a minimum and a maximum value. In MDSA, the minimum values are found by rounding the minimum number in the list down to the nearest 5. The round down is different from the ordinary rounding. As shown in Table 7.1, the values of the example are rounded down to the nearest 5.

|  |  |  |
| --- | --- | --- |
| Number | Round down to the nearest 5 | Math. calculation |
| 2 | 0 | 2 – 2 mod 5 |
| 6 | 5 | 6 – 6 mod 5 |
| 12 | 10 | 12 – 12 mod 5 |
| 14 | 10 | 14 – 14 mod 5 |

Table 7.1- Round down to the nearest 5 example.

The round down is calculated by using MOD 5. The round value is only used as an initial value. In the interval, the minimum value is calculated by round down, while the maximum value is calculated by adding the interval range to the minimum value. For instance, if the interval range is 10, then the values of (4, 7) can be presented by a range of [0 – 10[. The initial value of 4 is 4-4 mod 4=0. The maximum value is 10 + 0 = 10. However, the value 10 is not included in the interval, instead it is included in the next interval of [10 – 20[. The interval range is derived from the sensitivity level ψ. If ψ=0.5, then interval range = 1 / 0.2 = 5. The interval range must be a multiple of 5, so if the number in the interval range was found to be smaller than 5, then it is rounded up to 5, and if the number was found to be larger than 5, then it is rounded up to multiples of 5. This concept is described in Table 7.2. The example shows a list of ψ values and the interval range calculation.

Table 7.2- Interval Range Rounding down to the nearest 5 example.

|  |  |  |
| --- | --- | --- |
| sensitivity level ψ | Interval Range | Interval Range Rounding |
| 0.4 | 1 / 0.4 = 2.5 | 2.5 +(5 – (2.5 mod 5))= 5 |
| 0.015 | 1 / 0.015 = 66.6 | 66.6 + (5 – (66.6 mod 5)) = 70 |
| 0.045 | 1 / 0.045 = 22.2 | 22.2 + (5 – (22.2 mod 5))= 25 |

Table 7.2 illustrates an example for calculating the interval range round. The equation used in calculation can be mathematically presented as:

(7.1)

It is possible to round the both of minimum and maximum value of the interval, while keeping the interval range. In this case, the minimum value is rounded down, and the maximum value is rounded up. However, rounding up the interval range or the maximum value conclude similar results.

In a similar algorithm, we can mask data with taxonomy trees as explained in section 3.2.4.3. The aim of this algorithm is reducing the size of data flowing to the UDF program. This is implemented by masking fewer attributes at a time, then attaching the rest of the data tuples, to the anonymized attributes. To find out the estimated size of data flow to the UDF, We may roughly estimate the size of ten data records by 1KB, while the size of one attribute of the ten records may not exceed the 20 bytes. This shows that the actual data size flowing to the UDF file may not exceed one-third of the total data size. For this reason, data processing in UDF is not expectedly expensive.

## 7.2. Differences between Pig and Spark Algorithms

Figure 7.4 shows two algorithms implemented by Pig and Spark. Both are quite similar with minor differences in the data flow between memory and disk. Spark does not need a disk read/write operations. In both programs, the fully equivalent records (SG) are stored in disks as output files. The major difference between both programs is the semi-equivalent records (SSG) and their management. In Pig, SSG records are stored temporarily in disks. Moreover, Pig is not provided with a built-in UDF capability, therefore, an external program, such as Java, should handle the UDF operations. Unlike Spark SQL, which provides a UDF built-in capability by defining a Scala function.

Pig is a script used for MapReduce framework, which executes any task in two stages only, map and reduce. Hence, every action requires a temporary storage in disks. In contrast Spark does not require such limited stages. It uses memory for a temporary cache for all stages. Spark can complete one task and move to the next task by creating a series of map and reduce. Each map and reduce process is known as a stage. This is very efficient technique, but requires enough memory to cache each stage in memory. If the memory does not fit the output size of data, then the memory spill the rest of data to the disk.

In Spark algorithm, it is difficult to concatenate the anonymized data with the rest of the attributes in the table. For instance, if a table T contains the following attributes of T={A,B,C,D}. The anonymization process has finished masking the attribute A, and transformed it to a, that is (A 🡪 a). Hence, the anonymized attribute {a} needs to be concatenated with the rest of the attributes {B,C,D}, so the new table TA will be TA={a,B,C,D}. In Scala's implementation of DataFrames, there is no direct way to zip two DataFrames into one. We can simply work around this limitation by adding indices to each row of the data frames. Then, we inner join DataFrame by these indices. Appendix 3 presents a full Scala program including UDF.

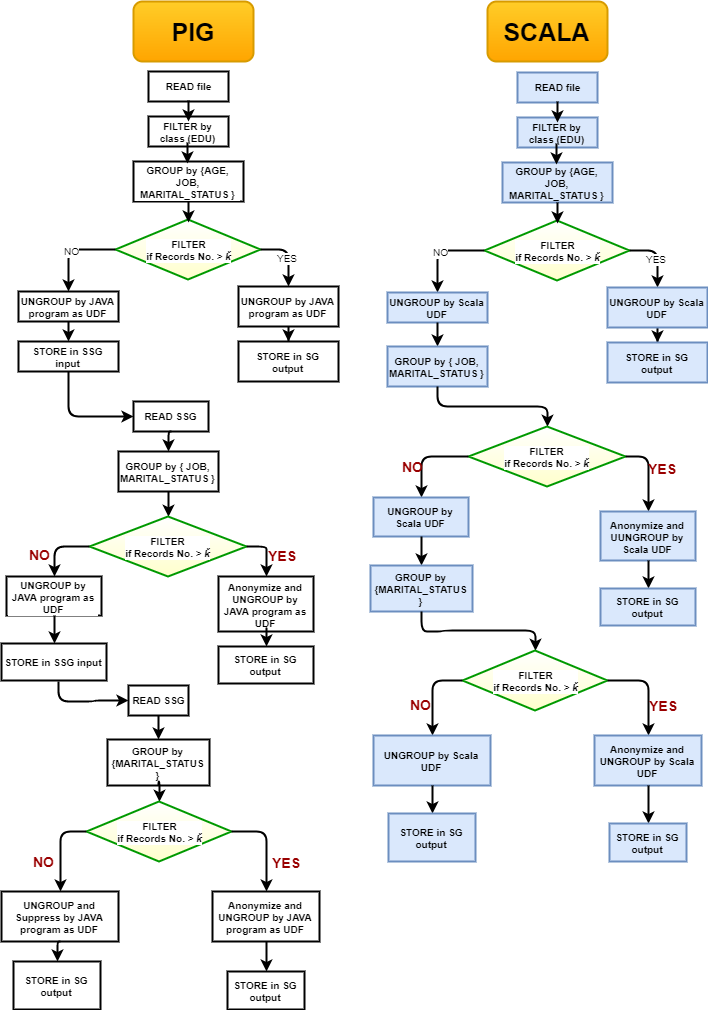


Figure 7.4- Anonymization algorithms in Pig and Spark.

However, all join operations are known as Cartesian join, which require high number of shuffling between nodes. Therefore, join operation is very expensive, and it is not recommended by Spark developers. The right solution is implementing another way of concatenating without creating an independent DataFrame. To understand the concept of this UDF script, let us give the following example, four Q-ID attributes and one non-Q-ID attribute, Teacher, are presented in Table 7.3. On grouping the records by three attributes (CLASS, SCHOOL, and LEVEL). The rest of the attributes, TEACHER and MARK collect multiple values in arrays. The script calls the anonymization UDF to anonymize the marks, and concatenates the rest of the attributes in one table. This can be implemented through (withColumn) command. The command syntax is;

*val anonymize\_MARK =Grouped\_QID.withColumn("MARK", AnonUDF($"MARK")).select("TEACHER","CLASS","SCHOOL","LEVEL",”MARK”).*

Table 7.3- Grouped and anonymized table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Q-ID Group | | | |
| **TEACHER** | **CLASS** | **SCHOOL** | **LEVEL** | **MARK** |
| Jones, Khan, Steve | A | School 1 | 1 | [70-80[, [70-80[, [70-80[ |
| Mark. Jane | B | School 2 | 2 | [90-95[, [90-95[ |

Table 7.3 should be ungrouped for storing or possible further processing. This format of grouped records cannot be easily managed for computation or statistical operations. Ungrouping the grouped data can be accomplished in various ways. The best-found way was creating another UDF that is able to map every sequence to a wrapped array, and rotate the direction of the wrapped array. The aim is converting Table 7.3 to Table 7.4 format. It is clear that the three Q-ID attributes are repeated according to the number of grouped objects in MARK and TEACHER.

Table 7.4- Illustrates the new state after ungrouping records

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Q-ID Group | | | |
| **TEACHER** | **CLASS** | **SCHOOL** | **LEVEL** | **MARK** |
| Jones | A | School 1 | 1 | [70-80[ |
| Khan | A | School 1 | 1 | [70-80[ |
| Steve | A | School 1 | 1 | [70-80[ |
| Mark | B | School 2 | 2 | [90-95[ |
| Jane | B | School 2 | 2 | [90-95[ |

The Ungrouping algorithm reads each wrapped array, counts the number of objects, and maps each array with indices. Each wrapped array has a various number of objects, therefore we need to define a function that can update the array size on each wrapped array. Scala defines functions by using (val) or (def) command. In our case, we implement (def), so the command can update the number of array objects, with the following syntax;

**def assertSameSize(arrs:Seq[\_]\*) = {assert(arrs.map(\_.size).distinct.size==1,"sizes differ")}**.

This definition is called by the UDF that is able to rotate the direction of the arrays with the following command:

*val ungroup = udf((xa:Seq[String],xb:Seq[String],xc:Seq[String],xd:Seq[String],xe:Seq[Integer]) => {*

*assertSameSize(xa,xb,xc,xd,xe)*

*xa.indices.map(i=> (xa(i),xb(i),xc(i),xd(i),xe(i)))*

*})*

The above UDF ungroups Table 7.3, and expands the wrapped array to the format of Table 7.4. In this example, the anonymization UDF outputs the range of marks in one string of values MARK={[70-80[, [70-80[, [70-80[}. This string should be converted to a wrapped array before ungrouping it. Converting a string to an array, in Scale, is implemented by the command split(col(MARK)). As noticed, implementing the fastest algorithm relies on several trials of execution before choosing the best method. In general, programming in big data should be carefully considered. This is quite similar to programming multi-task programs on a single computer with multiple processors. The program may not gain any advantage of the multi-core processor without a proper algorithm. For instance, an operation of total = a + b + c + d will run in one core only, while the rest of the cores are in ideal states. The same operation can be completed faster, if the algorithm was amended by tot1= a + b, tot2= c + d, and **total**= tot1 + tot2. Splitting the single operation to three operations of tot1, tot2, and total, enhances the performance and leverages the parallel processing on multiple cores. However, the operation of the **total** will be completed on one core only. The final result of the **total** will wait for the parallel operations of tot1 and tot2 to be completed. This operation is known as a sequence operation, which causes the major delay in algorithms [10]. Applying similar concept to parallel distributed operation mimics the mapping and reducing operations. Reduce is a sequence operation that waits for the mapping completion, and before the shuffling operation start-up. More shuffling leads to a higher operating cost.

The operation of the grouping process is implemented by the built-in transformation command “groupBy”. Alternatively, Scala permits the SQL embedded commands, so the grouping can be implemented in either way of groupBy or Select query. However, groupBy was found to be more efficient in performance wise. Each data tuple or record contains many attributes. As described before, MDSBA creates small Q-ID groups, which includes two to four attributes only. Also, each data record may consist of several G(QID), classes, and non-Quasi attributes. For this reason, we need to groupBy each G(QID) independently, while the rest of the attributes must be aggregated and concatenated with the anonymized G(QID).

# Comparison between Hadoop ecosystems and Spark

Experiments were conducted to compare between Hadoop ecosystem and Spark. The experiments aimed to measure the performance of the old MapReduce framework and the new Spark framework. The performance includes the computation cost, and scalability of each. Since Spark is a distributed in-memory platform, we need to observe Spark’s behavior on data growth. Eventually, Spark was developed for the new powerful servers that are provided with a large size of memory, and a considerable number of CPU cores. Spark is a memory consumer and CPU intensive operator, therefore, each worker should contain a reasonable size of memory and cores. Usually, memory size in each worker starts from 16 GB, with two quad-core processors. However, the required size of memory and processor in each worker of the cluster depends on three main factors, these are; the data size, the time required to complete the job, and the number of workers and masters within the cluster.

A lab was setup our in Western Sydney University, which includes five virtual machines, with one master and four workers. Each node’s CPU is a single core Intel(R) Xeon(R) CPU @ 2.40GHz, with a physical memory of 8 GB, and the operating system is CentOS 7. Both Spark and Hadoop were setup in the same cluster. Spark 2.1 was setup on Apache Hadoop 2.7. Also, Pig was setup in the NameNode to run the Pig Latin script. We created two different scripts programmed in Pig Latin and Scala. Both scripts must output similar results. To save resources, we executed Pig script first, then Scala script after the completion of Pig script. Adult data was used for the experiments. Data were randomly enlarged up to seven different sizes, and these are 500 MB, 1 GB, 2 GB, 3 GB, 4 GB, 5 GB, and 6 GB. The size of data sets was chosen related to the limited available resources in our cluster.

The first experiment aimed to compare the processing time between Spark and Pig. Both scripts are designed to read the data files, filter data by class value, group, anonymize and ungroup. The program concept is similar to the algorithm described in Figure 5. The adult data attributes include {AGE, JOB, MARITAL\_STATUS, EDU, SOCIAL, RACE, SEX, POSITION, COUNTY, COUNTRY, SALARY}. Attributes are divided into three groups of Q-IDs. The first Q-ID group is G(QID)1={AGE, JOB, MARITAL\_STATUS ,EDU}, where EDU is the class attribute. The second group is G(QID)2= {SOCIAL, RACE, SEX, POSITION}, where POSITION is the class attribute. And finally, the third group is G(QID)3= {COUNTY, COUNTRY, SALARY}, where SALARY is the class. For this experiment, group G(QID)1 is anonymized, by grouping Q-IDs of {AGE, JOB, MARITAL\_STATUS} in the first stage, then {JOB, MARITAL\_STATUS}, in the second stage, and finally { MARITAL\_STATUS} only in the third stage. Spark was setup with a total of three workers with 6 cores. Each worker node contains two cores, and each core contains 2 GB of memory. The three nodes have 12 GB of memory, with a total of 6 executers or cores.

The results with 3 workers show a large contrast in processing time between both scripts of Scala and Pig. Figure 7.5 shows the processing time for various data sizes. Spark is much faster than Pig in the relatively smaller data size. The relativity is described by comparing the data size with the memory size. In the proper memory size, Spark may reach up to 8x faster than Pig. On data size growth, Spark performance degrades dramatically, so the processing time becomes similar to Pig on data size= 3 GB. More data growth shows better performance for Pig speed in comparison to Spark. Actually, Pig does not speed up processes on data growth, but its processing time grows up steadily. In contrast, the discrete line in Figure 7.5 shows an exponential growth for the Spark processing time. SQL Spark consumes more processing time, when the memory is not large enough in comparison with the data size. For instance, when data size = 6 GB, the process time was around 570 minutes, which is much larger than Pig process time, which was around 390 minutes.

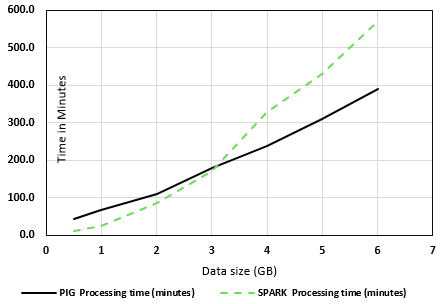


Figure 7.5. Comparison in process-time between Pig and Scala scripts with 3 workers

In the second experiment, one more worker was added to observe Spark’s behavior with the larger hardware capacity. The worker node consists of 8 GB memory and two core processors. The previous experiment was conducted by 3 workers and one master, while this experiment was conducted by 4 workers instead. The fourth worker was added to both Pig and Spark clusters. It showed a dramatic increase in Spark performance, which was expected. Figure 7.6 illustrates the same datasets with the extra worker added to Spark domain and Pig domain. Spark’s processing time showed a slight degradation, when data size exceeded 5 GB. However, the overall performance was better than using three workers only.

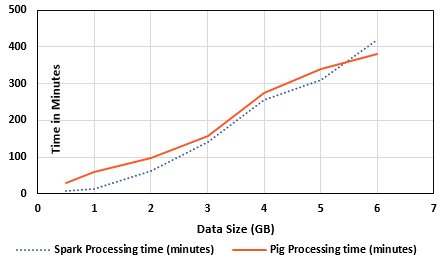


Figure 7.6- Comparison in process-time between Pig and Scala scripts with 4 workers

MapReduce is prone to network congestion during map or shuffle phases. Spark reduces this negative impact by reducing the data transmission between disks and memory. This feature is essential in the transmission of the network. Having many trials of experiments with Pig; several tasks took much longer time before throwing errors and terminating the tasks. The engineering structure of MapReduce and Spark is similar when handling the slow tasks. They both implement a speculative execution, which tags any task that takes longer on average than the other tasks from the same job. It clones this slow task and runs it on another node. It will not stop the slow task, but rather run another copy in parallel. This is beneficial in large clusters, whereas small clusters may lose their available resources. However, the university network is not dedicated to MapReduce structure, and suffers from a high network congestion most of the day. In both cases of enabling or disabling Pig’s speculative execution, almost similar delay has appeared in some tasks. Therefore, each experiment was repeated several times, mainly, in running Pig script. The failure jobs where repeated and excluded from the comparison time. The failure percentage of processing tasks in Pig script was around 4%, which increases with the data growth. Many factors may cause this congestion, such as university virtual environment and connection types between storages and virtual machines. Big data clusters rely on direct access connection (DAC) between nodes and storages. Also, the virtual environment is not recommended for big data structure. Figure 7.7 shows 319 tasks that were executed by Pig script for various data size. The tasks belong to more than 15 jobs with an average processing time between 10 seconds to 18 minutes for the successful tasks. The shown pulses represent the failure tasks after long processing time. The failure tasks increase in parallel with the data size increase.

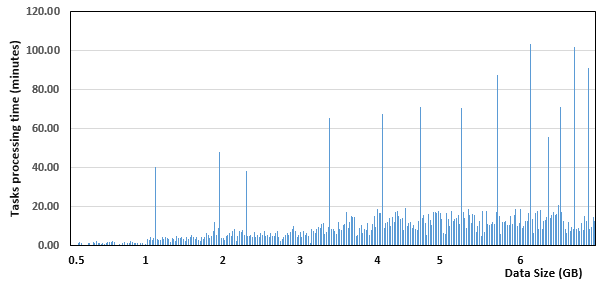


Figure 7.7- Failed tasks in Pig script are shown in a high processing time

## 7.1. Spark Tuning in MDSBA

Tuning Spark is one of the hardest tasks on managing Spark cluster. This requires knowledge, experience, and several experiments. There are no clear instructions on tuning Spark to gain the best performance. The focus here is on anonymization operations only. Different tasks and applications may require different steps of tuning. In MDSBA, the main operations conclude grouping, ungrouping, and masking data. These three main operations should be organized properly to give the best possible performance. Both ungrouping and masking require UDF programs. In masking, the algorithms should consider the least number of iteration and the smallest size of data. As explained earlier regarding the importance of reducing the data flowing to the UDF. Several tuning techniques were experimented and configured. Two main tuning concepts are found; these are: filter/group, and cache data.

In SQL grouping, experimentally it was found that filtering data, and then grouping it, may reduce the grouping time and enhance the performance. Hence, we need to filter and then group data, rather than jumping over to the GROUP command first. It is apparent that grouping data records can replace the command FILTER, so technically, we can group any records without the need for filtration first. However, to reduce the number of shuffling times and leverage the parallelization, it is better to filter records first. For a better understanding, let us consider a set of passenger’s records as shown in Table 7.5. If the grouping command included the attributes of; Depart, Arrive, and Flight, then the grouping results can be {ATL, DXB, (Adult, Child), K380}. We also gain similar results if we use filter command first. For instance, if we filter the records with the class value =K380, then data size will be reduced to two records instead of three. Using filter/group commands concludes the same results as using group command alone. However, in performance-wise they are not similar.

Table 7.5- Passenger’s records

|  |  |  |  |
| --- | --- | --- | --- |
| **Depart** | **Arrive** | **Passenger Type** | **Flight** |
| ATL | DXB | Adult | K380 |
| HND | SYD | Adult | D120 |
| ATL | DXB | Child | K380 |

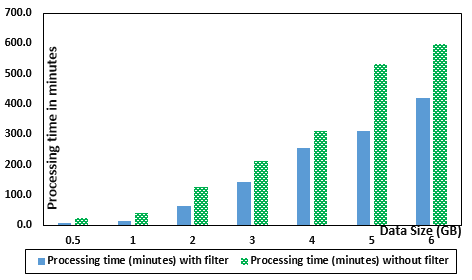


Figure 7.8- Process time between filter/group and group only.

Figure 7.8 shows the processing time difference between grouping data after filtration, and grouping data without filtration. It is clear that filtering data first increases the program performance. Filtering data first is beneficial, when the attribute values are known, so they can be added by programmers. In MDSBA, the attribute values are pre-added automatically in early preparation stages. MDSBA framework consists of three main services; core, initializer, and anonymizer. The Scala script is generated in the initializer stage by reading dataset parameters and user‘s access parameters from two different XML files. User’s XML file is received from the federation service, while dataset XML file is previously uploaded by data owners to the service provider’s servers, parallel with the dataset. MDSBA framework is further explained here [25].

The second tuning in Spark is data caching. The percentage of the failure tasks in Spark script was much lower than in Pig script. The percentage was even lower, when using persist command on reading data from disks. As mentioned before, the (**persist)** command caches the data in memory. Developers may assign the (**persist)**, if data will be read many times for multiple tasks. The comparison between (**persist)** and non-persist commands showed a tangible difference in performance between them. Persist command was implemented after reading a dataset from the disk. Figure 7.9 shows a large difference between both cases, where using (**persist)** reduces the processing time of tasks. The (**persist)** command is part of Spark tuning to increase the overall performance efficacy. However, the command is not recommended when the data size is larger than the available memory. If the data size is larger than the memory, then the overloaded memory will be spilled to the disk. Figure 7.9 illustrates the performance comparison between (**persist)** and non-persist. The time processing contrast increases with the increase of the data size.

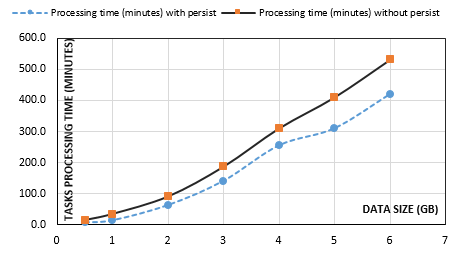


Figure 7.9- Performance comparison between caching and non-caching data

These two steps of tuning make a considerable difference in anonymization performance by MDSBA. Before tuning Spark, it is essential to building a robust Spark cluster and nodes. The hardware infrastructure is the major factor for big data operations success and performance. As mentioned earlier, the memory size of each worker, the number of cores, the number of workers, and the network connection between workers and storages, should be large enough to accommodate the massive size of data. Data anonymization and analytics are heavy processes that require a powerful and large cluster specifications.

# Summary

With the recent big data’s revolutionary growth, various processing frameworks were developed. Some of these frameworks best fit streaming data, while others can be applied to batch data. In this chapter, two popular frameworks were experimented for k-anonymity method. Both of MapReduce and Spark were examined in MDSBA anonymization process. The aim was paving a reliable ground for MDSBA state-of-the-art anonymization with the most reliable performance. The experiments show few hurdles in each framework. However, Spark is faster, and more fault tolerant as an in-memory operations framework. In congested networks, Spark reduces data transmission between memory and disks on serializing data with RDD. Spark can be many times faster than MapReduce in anonymization. To avoid SQL Spark performance degradation, the memory should be larger than the processed data. SQL Spark boosts performance on the large size of memory. Other Spark’s tuning methods that may leverage the anonymization performance are; UDF algorithm, filter/group commands, and caching data in memory.

On the other side, MapReduce is an old framework that can perform better when memory resources are quite small. This is conditioned by the network traffic and congestion level. MapReduce may operate on small memory resources, but it requires non-busy network, because it relies on a heavy transmission between memory and disks. In the conclusion, It is recommended to implement MDSBA in Spark framework, however, the cluster infrastructure must be prepared well, by providing enough memory and processor resources for each node.

Our future research will focus on finding a proper method for data stream anonymization. In big data, streaming is an essential type in most data applications. The current anonymization method of MDSBA does not support the data stream anonymization. Most recent big data frameworks provide complete solutions for the data stream. We may need to amend the current MDSBA method to cope with the continuous data streaming.

# References

[1] S. Gopalani and R. Arora, "Comparing apache spark and map reduce with performance analysis using K-means," *International Journal of Computer Applications,* vol. 113, 2015.

[2] H. Chen, R. H. Chiang, and V. C. Storey, "Business intelligence and analytics: From big data to big impact," *MIS quarterly,* vol. 36, 2012.

[3] M. Guller, *Big Data Analytics with Spark A Practitioner's Guide to Using Spark for Large Scale Data Analysis*: Berkeley, CA : Apress : Imprint: Apress, 2015., 2015.

[4] L. Sweeney, "ACHIEVING -ANONYMITY PRIVACY PROTECTION USING GENERALIZATION AND SUPPRESSION," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems,* vol. 10, pp. 571-588, 2002.

[5] P. S. Ke Wang, S. Yu, and S. Chakraborty. (2004) Bottom-up generalization: a data mining solution to privacy protection. 249-256.

[6] B. C. M. Fung, K. Wang, and P. S. Yu, "Top-down specialization for information and privacy preservation," ed. USA, 2005, pp. 205-216.

[7] A. Irudayasamy and L. Arockiam, "Parallel bottom-up generalization approach for data anonymization using map reduce for security of data in public cloud," *Indian Journal of Science and Technology,* vol. 8, p. 1, 2015.

[8] X. Zhang, C. Liu, C. Yang, J. Chen, S. Nepal, and W. Dou, "A hybrid approach for scalable sub-tree anonymization over big data using MapReduce on cloud," vol. 80, ed, 2014, pp. 1008-1020.

[9] L. T. Y. Xuyun Zhang, Chang Liu, Jinjun Chen,, "A Scalable Two-Phase Top-Down Specialization Approach for Data Anonymization Using MapReduce on Cloud," *IEEE,* 2014.

[10] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*: Morgan kaufmann, 2016.

[11] V. Govindaraju, *Big Data Analytics*: Oxford : Elsevier Science, 2015., 2015.

[12] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald*, et al.*, *Clash of the titans: Mapreduce vs. spark for large scale data analytics* vol. 8, 2015.

[13] M. Frampton, *Mastering Apache Spark*: Packt Publishing Ltd, 2015.

[14] G. P. Gibilisco, M. Li, L. Zhang, and D. Ardagna, "Stage aware performance modeling of DAG based in memory analytic platforms," in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*, 2016, pp. 188-195.

[15] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, 2015, p. 53.

[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," *HotCloud,* vol. 10, p. 95, 2010.

[17] K. Chodorow, *MongoDB*. Sebastopol: Sebastopol : O'Reilly Media, 2010.

[18] M. A. Abbasi, *Learning Apache Spark 2*, 1st ed. ed.: Birmingham : Packt Publishing, 2017., 2017.

[19] D. B. West, *Introduction to graph theory* vol. 2: Prentice hall Upper Saddle River, 2001.

[20] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041,* 2014.

[21] R. C. Ionescu, "A scalable system for primal-dual optimization," *arXiv preprint arXiv:1507.01456,* 2015.

[22] J. Lee, S. Cozens, and P. Wainwright, "Object-Oriented Perl," in *Beginning Perl*, ed: Springer, 2004, pp. 253-285.

[23] D. Chen and H. Zhao, "Data security and privacy protection issues in cloud computing," in *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*, 2012, pp. 647-651.

[24] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters," *HotCloud,* vol. 12, pp. 10-10, 2012.

[25] M. Al-Zobbi, S. Shahrestani, and C. Ruan, "Implementing A Framework for Big Data Anonymity and Analytics Access Control," in *Trustcom/BigDataSE/ICESS, 2017 IEEE*, 2017, pp. 873-880.