Secured Array DBMS on Spark

LIN, Ching-Yu
The Hong Kong University of Science and Technology
Clear Water Bay, HK
clinah@connect.ust.hk

Keywords
Array Database, Security, Spark, SciSpark, SparkSQL, SparkArray, H5Spark, Opaque

ABSTRACT
A conceptual report for discovering a secured array DBMS in distributed environments. The report consists of three parts, including reviews on the existing variations of Spark, known security issues in the database areas, and the possible methods to implement a secured array DBMS in a distributed environment. The first part mainly discussed current approaches to tuning a well-known distributed in-memory data-base system, Spark, to ingest and manipulate array data, including SciSpark, SparkArray, H5Spark, and SparkSQL. The second part included some reviews on security issues in database management systems. The last part proposed a possible design for a secured ADBMS based on Spark.

1. INTRODUCTION
While having more and more scientific or time-geographical data collected every day, the security of DBMS which manages these data could become an issue. In this report, the first part reviews the approaches of Spark variations to accommodate multidimensional data onto the general-purpose distributed computing platform Spark. These DBMSs provide useful techniques for building the container to hold array data from the original Spark RDD. Also, they provide some critical aspects to consider, such as format converting, loading, transferring data between partitions. Further, in the second part, some critical security issues aside from data itself and Opaque, a secure DBMS on Spark, are discussed. From the experience of Opaque, the risk of access pattern leakage from the network I/O transferring data could be mitigated by reordering the optimized sequence of operators as well as adopting oblivious operations to shuffle the data while sorting. Moreover, in the last section, it demonstrates a proposed secure array database based on Spark to enhance security in the ADBMS area.

2. ARRAY DBMS ON SPARK
Spark is a well-known in-memory distributed database management system frequently used by the fields related to Big Data[3]. It provides a fault-tolerant and load-balancing environment for handling a huge volume of data among nodes[6]. For discovering the possibilities to implement a secured array database management system based on Spark, this part of the report explores the architectures and usages of Spark and the architectures, implementation methods, syntaxes, and performance evaluation of existing similar array DBMS on Spark, including SciSpark[4], SparkArray[5], H5Spark[3], and a popular variation of Spark, SparkSQL[1], which is not an array DBMS, but worth some discussions on it.

2.1 Spark
Spark is an in-memory database management system designed for a distributed environment based on Scala. It was proposed for having better utilization of the computing power of a commodity cluster for iterative and interactive computation works. Most of the costs for handling big data related computation jobs are from the repeated use of data and the data transferring overhead between the file system and memory and between nodes[6]. Spark alleviates the overhead by two methods. It introduces a data structure RDDs, Resilient Distributed Datasets, to represent the loaded data indicated by the users in an abstract shared memory within a cluster. Further, it adopts the computation strategy MapReduce to minimize the work to only the needed procedures. By only doing the needed processes, which is called the lazy way, it reduces the unnecessary steps to load unused data and to store the interim products during the computation. Spark also integrates Scala's interpreter for accommodating more user-defined classes, to have more variations of RDDs. This provides possibilities to build array data structures on top of Spark and take advantage of its distributed environment.

2.1.1 RDDs
The new data structure introduced, RDD, resilient distributed dataset is represented by a collection of read-only Scala objects[6]. It is designed for data reuse, scalability, and fault tolerance. It is resilient. By keeping its tracking information of how it is produced, if any parts of the RDD are lost, Spark could effortlessly recompute the needed parts. RDD lazily does the computation. Spark instantiates the

requested RDDs to memory only on demand. And, its existence is ephemeral in the memory. Once computations for RDDs are finished, Spark automatically discards them from memory.

Besides having lazy computation and ephemeral existence, it gives out two options for users to manipulate the persistence of RDD. One is "cache" to store RDD for future reuse. The cache command forces Spark to keep the result of RDD in the memory even after the computation task is finished. However, the computation strategy remains to be lazy. In some cases with nodes having insufficient memory, the cached RDDs will still be dropped but restored by recomputing once needed, which is similar to virtual memory in some sense. The other option is "save". Spark writes the RDDs back to the drive, HDFS (Hadoop Distributed File System), or local file systems, to persist the data. Map, flatMap, and filter are also used for transferring a RDD to another RDD.

2.1.2 MapReduce

Spark adopts the MapReduce strategy to solve two main issues for managing a large volume of data, I/O overhead in iterative jobs and uncombined MapReduce job allocations for interactive analytics on Hadoop[6]. MapReduce consists of two parts, Map and Reduce. Map relates to splitting the assigned datasets to nodes and tracing the sequence of queued jobs by mapping them into an directed acyclic graph. Reduce infers shuffling the tasks and reducing, which combines the result back to the main node, the job tracker. In the paper[6], authors indicated that shuffling had not yet been implemented by the time of publishing.

Spark has MapReduce in a parallel way. There are three parallel operations, reduce, collect, and foreach. Reduce gives out the result of the computation. Collect is used to update data on the users' side. Foreach passes elements in the RDD to the user-provided functions.

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val cachedErrs = errs.cache()
val ones = cachedErrs.map(_ => 1)
val count = ones.reduce(_+_)
```

Spark Uages Example[6].

2.1.3 Scala Interpreter

Scala interpreter used in Spark allows others to tune Spark to hold their user-defined data structure in RDDs. There are two main modifications realized in Spark. It outputs the definition of the user-defined classes to the shared file system, which allows working nodes to easily load the data. Furthermore, it sends the references of the objects to workers directly[6]. By using the service of the Scala interpreter in Spark, there are some opportunities to hold the array data and build an array DBMS on top of Spark.

2.2 SciSpark

SciSpark extends Spark for scalable scientific computations[4]. It introduces sRDD, Scientific Resilient Distributed Dataset to accommodate multidimensional data and process computations in the MapReduce paradigm. By utilizing sRDD, SciSpark could process those multidimensional scientific data stored in HDFS with the features provided by Spark, MapReduce, to attain better data reuse, scalability, and fault-tolerant while handling a large amount of data.

2.2.1 Architecture

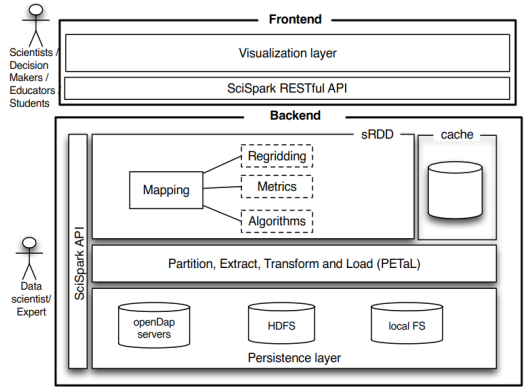

Figure 1: The Architecture of SciSpark[4].

There are three main layers in SciSpark's architecture, persistence layer, PETaL layer, and processing layer[4]. The system starts with the persistence layer. It ingests data from the HDFS, OpenDAP, or the local file system in a parallel manner. After recording the data with their URI, the system passes the sets of URI to the next layer, PETaL, Partition, Extract, Transform and Load layer, to transform the raw data into sRDD's structure. Once the sRDD is successfully created, the users could manipulate the multidimensional data with provided libraries, including Breeze and ND4J.

In the persistence layer, SciSpark is designed to ingest two types of scientific data formats, including netCDF and HDF formatted files. SciSpark locates the data with their URI, instead of loading them into the memory directly. The layer accumulates and aggregate the sets of URI with non-linear manner, to exploit the features of HDFS for speeding up the processing time, and return them to the next layer, PETaL layer.

In PETaL layer, Partition, Extract, Transform, and Load layer, there is a set of APIs called SciSparkContext. SciSpark uses the API to partition and distribute the URIs to the nodes in the cluster. After each node receives assigned URIs, they extract the data from HDFS, transformed them into arrays to enable them to be stored in the sRDD format, and passed the sRDD to the next layer, processing payer.

In the processing layer, since data have been transformed into the array container, sRDD, users could manipulate the

data with common multidimensional style by applying linear algebra functions from the chosen libraries, either Breeze or ND4J.

Apart from the backend of the system, SciSpark provides some RESTful Web API for users to manipulate the data. Also, besides netCDF and HDF, SciSpark could be used to contain more different kinds of multi-dimensional scientific data. The mapping functions in the persistence layer as well as partitioning and transforming functions in the PETaL layer should be extended to accommodate the need.

2.2.2 *Implementation*

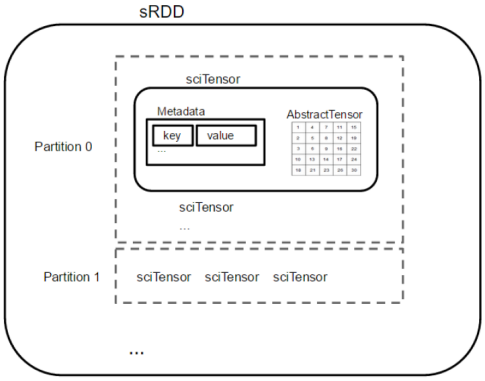

Figure 2: sciTensor in SciSpark[4].

The main approach in SciSpark is sRDD, Scientific Resilient Distributed Dataset, a newly introduced version of RDD, to contain and manipulate multidimensional data. sRDD has a substructure called sciTensor. "sciTensor" consists of two hash tables. One is for storing metadata in key-value pairs. It is used to store additional information, for example, data, area, minimal value, and maximal value. The other one stores the data itself. It containes some key-value pairs also. The keys correspond to the names of variables, while values store the array data. Several sciTensors could be in a partition, while several partitions form an sRDD. By storing array data in the partitions similar to RDD, SciSpark takes advantage of the distributed and MapReduce features provided by Spark.

sRDD attains the goal to store multidimensional data by storing supplementary metadata, such as date and time, into every basic storage unit, sciTensor. To produce sciTensor from data stored in OpenDAP, HDFS, or local file systems, SciSpark first navigates and records the needed OpenDAP URLs, HDFS path names, or paths to files in local file systems. The step produces the URI sets. After that, the partition functions group the URI sets and send subsets of URI to the nodes in the cluster. After receiving the URIs, SciSpark will utilize the loader functions provided by users, or netCDF and HDF loaders provided by SciSpark itself, to extract and transform the data into sRDD.

In the paper[4] and codes in the groups' GitHub repository, SciSpark could transform the data into a tensor-like structure either to Breeze or ND4J. Although Breeze has more succinct syntax by using operator overloading, it could only support two-dimensional arrays. On the other hand, ND4J supports n-dimensional arrays and Python Numpy style syntax. Nevertheless, it was still unmatured at that time while SciSpark developed[4]. Furthermore, by loading data into these two kinds of arrays, users could immediately call the functions in their linear algebra libraries for performing computations on the array data.

2.2.3 *Syntax*

SciSpark is implemented by Java and Scala, which is very similar to Spark. Hence, the usage of SciSpark has rather little difference from the way users are familiar with in Spark. Users could interact with SciSpark with Scala's functional programming paradigm[4]. Also, MapReduce syntaxes resemble the ways in Spark.

Indeed, the transferring syntaxes such as map, flatMap, and filter, are remained unchanged. Users should implement their commands by designing the lambda functions inside map or other similar transferring functions.

```
for(i = 0; i < n - 1; i++){
   FrameOneRDD = sRDD.filter(p => p_f == i)
   FrameTwoRDD = sRDD.filter(p => p_f == i + 1)
        ...
}
```

Figure 3: Syntax in SciSpark[4].

```
RDDDoubleKey = InputRDD.flatMap(p => List((p_f , p), (p_f + 1, p)))
ConsecutivePairRDD = RDDDoubleKey.GroupByKey
ConsecutivePairRDD.map((A, B) => {
    ...
})
```

Figure 4: Syntax in SciSpark[4].

Unlike other Spark extensions having a Python interface to interact with the system itself, SciSpark is not equipped with one. In the paper[4], the authors made the decision based on the known latency issue in PySpark environment. The communication overhead between Java and Python is due to the expensive cost to pass data between JVMs and Python daemon processes. It is originated from passing in and passing out the data to a local disk for facilitating the communication between two kinds of languages. In the testings with Spark and Opaque in the early stages of this UROP project, there is a significant communication overhead observed while using the Python interface to manipulate them.

From the experience of SciSpark, it is better to have succinct syntaxes by having overloaded operators. Additionally, building the interface directly with Scala instead of Python could avoid the communication latency between Python processes and JVMs.

2.2.4 Performance

Since the computations of SciSpark greatly relies on the original ways in Spark, mapping funcitons. To have better performance, it is critical to fully exploit the features of MapReduce by eliminating unused data in the map functions before entering the stage to really process the data. This is, it is better to use the lazy style of computation to traverse the directed acyclic graph for jobs first, instead of directly loading the whole dataset into the memory. Hence, the significant I/O overhead could be greatly mitigated. However, which kinds of mapping strategies to adopt highly depend on users. It would be relatively difficult to evaluate the performance of SciSpark, while having different kinds of mapping functions.

2.3 SparkArray

SparkArray[5] is another approach to handle multidimensional data in Spark's distributed environment. It proposes its data structure SparkArray and emphasizes more on evaluating the performance of the array operators based on SparkArray. It recognizes that there is a need for scientists to have an easy-to-use and flexible system to handle massive data from experiments and observations nowadays. The system should also have high throughput and low latency in numerical analysis, and linear scalability to accommodate different size of data[5].

2.3.1 Architecture

There are two main parts of SparkArray's architecture, SparkArray, and operators. SparkArray uses a schema to handle those stored multidimensional data. In the schema, there are two lists. One for storing values of dimensions, which could be used as an index. The other one is a list of attribute values. Besides the data structure introduced in SparkArray, there are three kinds of operators, unary independent, unary bounded, and binary operators. For independent operators within the unary category, it refers to the part for processing which is independent of other parts of the array. For the bounded operators, it handles the computation works related to values outside but close to the boundary of the chunk. For binary operators, it deals with two input arrays, which mainly are join operations.

$$Image < pixel : int, color : string > [i = 0 : 99, j = 0 : 99]$$

Figure 5: A 2-dimensional array representing an image with attributes color and pixel in SparkArray[5].

2.3.2 Implementation

SparkArray stores the multidimensional data into chunks, which each contains multiple cells. There are three options for nodes in the cluster to store the array data, independent, merge, and overlap. The independent option has separated chunks to store array data, which acts very similar to chunks in other array database management systems. For the merge, it has chunks stored in each node and extra boundary parts stores in a specified node. The overlap has chunks having parts overlapped to each other, which could speed up some the computations in some cases, but would lead to overhead in memory to keep the duplicated data.

For the independent operators, there are four kinds of operators implemented, including filter, apply, slice, and subsample. Filter returns an array at the same size as the input array, with only filtered values. Apply is used to add a new attribute to a cell. Slice returns the array data along a certain dimension. Subsample gives out a subarray within the input one.

For bounded operators, there are smooth, regrid, and cluster. Smooth acts as if the average selector cursing through an area and replace one of the cell's value with the average value. Regrid is used to compress or extend the input array. Cluster extracts the locations of value clusters in the array.

For binary operators, SparkArray mostly deals with optimizing join operations. There are three kinds of join implemented, broadcast join, repartition join, and repartition filter join. Broadcast join broadcasts one of the arrays to every node in the cluster and do the join operations afterward. However, this method could only be used for the case that one of the arrays is comparatively small to another one. Otherwise, it would have heavy overhead on memory and network I/O. A procedure to pre-process the task by first finding out the overlapped part of two arrays and only broadcast the overlapped part could slightly mitigate the overhead in this join method. More, repartition join divides two arrays into several partitions by using the same partitioning function. For each partition, the indexes are used as keys to these partitions where indexes are mostly the dimension values. Having some hashing functions to be the partition functions could speed up the process. Further, repartition filter join adds a procedure to filter out the intersected parts between two arrays before partitioning two arrays. Although there is a tradeoff to compute the overlapped part, "repartitioning join with filtering" still performs better and could be deemed as an improved version of the repartition join.

2.3.3 Syntax

Although in the paper[5] the real use cases of SparkArray are not mentioned, it should be very similar to the cases in SciSpark[4] but with more encapsulated operators since the goal for SparkArray is to provide scientists with easy-to-use and reliable array database management system based on Spark.

2.3.4 Performance

SparkArray uses Standard Science DBMS Benchmark (SS-DB) from MIT CSAIL lab as the testing dataset for the performance evaluation[5]. From the experience of the smooth operator, while partitioning the array, the number of partitions should not be too few. Otherwise, the CPU utilization would be relatively low. While dealing with overlapping chunks, the overhead of extra storage space to store the data of overlapping parts should be noticed. While for repartition filter join, it is generally better than repartition join in terms of the computation time. For broadcast join, it only works well with the part to broadcast is small. Otherwise, the communication overhead would be significantly huge.

Compared to SciDB, an array database management system built on top of a relational database structure, SparkArray outruns SciDB in the cases of data reusing but has a higher cost to start the tasks and prepare the materials related to computations compared to SciDB.

2.4 H5Spark

H5Spark[3] is an approach to introduce Spark to HPC, high-performance computing, environment. It supports HDF5 and netCDF formatted scientific data, which are commonly used by scientists. Aside from extending Spark to ingest HDF5 and netCDF, which are not natively supported, H5Spark also makes Spark able to link to Lustre, a parallel distributed file system generally used by many HPC systems, since Spark originally supports HDFS and local file systems only. H5Spark places more emphasis on I/O related issues, including network I/O, file I/O (disk I/O), file system access, and scalability while processing a huge volume of data.

2.4.1 Architecture

In H5Spark, there are five main components to transform data from HDF5 to RDD, including Metadata Analyzer, RDD Seeder, Hyperslab Partitioner, and RDD Constructor[3]. Metadata Analyzer is triggered first to obtain metadata from the targeted files. It goes through the file system and fetches the metadata of the HDF5 files by file names and dataset names provided by the user's input. Hyperslab Partitioner partitions data into chunks to make parallel I/O reading the data possible. RDD Constructor provides users several reading functions to easily transform data to RDD. There are several kinds of reading functions. Different reading functions correspond to different data types in RDD.

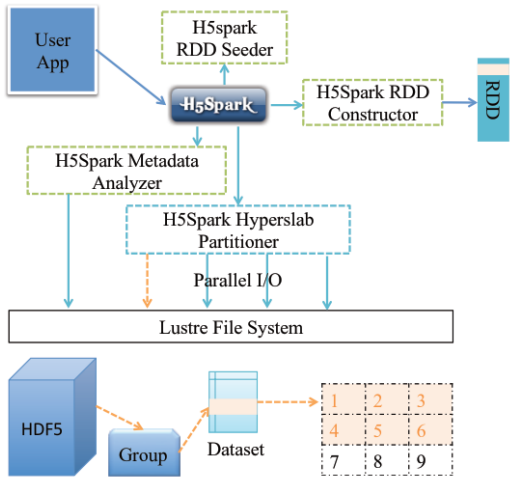

Figure 6: The Architecture of H5Spark[3].

2.4.2 Implementation

To improve the I/O, H5Spark uses the option cache in Spark to remain the data in the memory for future data reuse in iterative circumstances[3]. By forcing the system to keep the data in the memory, I/O overhead caused by repetitively loading the same portion of data could be greatly alleviated. However, users need to ensure there are sufficient memory space to contain the whole cached data. Otherwise, the data will still be dropped even cache is indicated.

Since in Spark only CPU utilization and memory space are taken into consideration for scheduling tasks, it often encounters I/O bottleneck[3]. H5Spark takes I/O cost, particularly the disk I/O into the consideration for scheduling tasks, which could be seemed as I/O optimization for Spark.

2.4.3 Syntax

H5Spark provides two kinds of interface for users to interact with it. Similar to Spark, users could use Python through PySpark or directly using Scala to interact with H5Spark. For importing HDF5 or netCDF files into H5Spark, the only thing for users to do is to provide the reading functions in RDD Constructor with file names and dataset names to transform the data into RDD. After successfully reading the data, users could then apply normal computation tasks in Spark on these RDD.

```
from h5spark import read
from pyspark import SparkContext
sc = SparkContext()
rdd = h5read(sc, file_list_or_txt_file,
      mode='multi', partitions=2000)
```

Figure 7: Syntax using H5Spark with PySpark[3].

```
import org.nersc.io._
val rdd = read.h5read (sc,inputpath,
          dataset name, partition)
```

Figure 8: Syntax using H5Spark with Scala[3].

2.4.4 Performance

Considering Spark was built with Scala, which works on JVMs, garbage collection could sometimes be an issue to the system's performance. While testing Spark and Opaque with a weather dataset in the early stages of this UROP project, garbage collection could be a bottleneck in some extreme data reuse cases with insufficient memory size. While in H5Spark, it is found that garbage collection is generally a constant factor to the performance[3].

While for the number of partitions, H5Spark also encourages to have more partitions and executors. In the cases with too few partitions, some nodes would need to digest a huge amount of data, while others would be idle, which lowers the system's performance. It is better to have both the number of executors and the number of partitions increased if encountering the low CPU utilization problems.

For the size of the cache, while using Scala, it is about the same as the original size of the data, which means that all data is in memory. While for Python, the cache size is restricted to a number, which is caused by a Python data structure[3]. Since it is unable for Python interface to cache

all data, there is more unnecessary I/O to repetitively load the reused data.

While comparing to a similar system, SciSpark[4], H5Spark has better data loading performance since it loads the data by chunks in a parallel manner, while SciSpark loads data in a serialized way[3].

2.5 SparkSQL

SparkSQL builds a relational database in the environment of Spark[1]. It provides the DataFrame API to simplify the usage for handling the data in Spark's RDD form. It uses Catalyst optimizer in the underlying part to optimize the performance in Spark. Also, by combining relational programming and Spark's native procedural programming, SparkSQL provides users a more comprehensive way to handle data. Although SparkSQL is not an array database management system, its approaches to modify Spark's workflow and optimizing the job sequences could still be reviewed.

2.5.1 Architecture

There are two main parts in SparkSQL[1], DataFrame, and Catalyst. DataFrame provides users a set of succinct and easy-to-use APIs to assign jobs to a dataset. Besides using DataFrame API, using Spark's native RDD mapping functions is also acceptable to manipulate the data. Both jobs assigned by DataFrame API and commands in Spark RDD mapping functions are listed and optimized by the optimizer Catalyst before passing the real tasks to Spark engine.

DataFrame in SparkSQL could be deemed as an RDD with a schema accompany. DataFrame corresponds to the concept of tables in relational databases. Each DataFrame serves as a structured collection of data with column data type defined. While it seems like a physical object from users' points, it is rarely materialized in reality since SparkSQL is based on Spark, which has the lazy style to do computations. DataFrame turns out to be a logical plan for recording the job sequence and is passed to Catalyst for optimization.

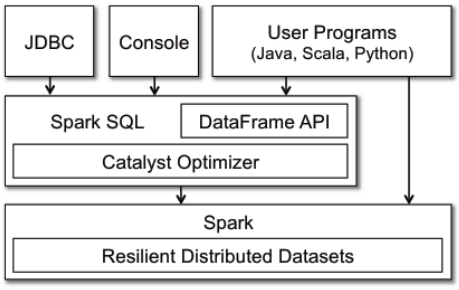

Figure 9: The Architecture of SparkSQL[1].

Catalyst optimizer serves as the handler to optimize the logical plan recorded in DataFrame. There are two kinds of strategies used in Catalyst, rule-based optimization, and cost-based optimization. After collecting the jobs assigned by DataFrame API and Spark RDD mapping functions, it tries to optimize the logical plan based on the defined rules. After finalizing the logical plan, it resumes to the physical planning stage. In the physical planning stage, it generates different plans to realize the logical plan and compare them based on the cost. Subsequently, the optimizer compiles the codes into Java bytecodes for better performance.

2.5.2 Implementation

For DataFrame there are two possible usages for users to manipulate data represented by DataFrame. One of them is to manipulate the data with relational operators provided by SparkSQL. There are four common kinds of operations implemented, select as projection, where as filter, groupBy as aggregation, and join. Besides the relational style operators, using Spark RDD's native mapping functions is also feasible. Users could manage the DataFrame as Spark RDD Row Objects directly. Although DataFrame seems to be the object directly managed by users, DataFrame is not directly materialized into memory after its creation. It is still processed with Spark RDD's lazy computing paradigm, which leaves some spaces for Catalyst to perform the optimization.

DataFrame could still be cached into memory. With a similar option cache in Spark, DataFrame could remain itself in the memory to improve the performance for iterative and interactive uses. Unlike Spark directly stores the whole and unprocessed data in memory, SparkSQL has some optimizations for the storage space taken by the cached datasets. It adopts columnar cache containing dictionary encoding and run-length encoding to compress the cached size.

Another important part of SparkSQL is the Catalyst optimizer. It optimizes the execution sequence with two kinds of strategies, rule-based optimization, and cost-based optimization. Rule-based optimization uses pre-defined rules to go through the representing trees of the predicates from the job sequences. It applies those rules on them to perform the optimization. It is executed with batches. This is, it only ends looping if the trees stopped changing after a round of the optimization. This could recursively go through all possibilities for the optimization. On the other hand, cost-based optimization handles the case with several plans by comparing their predicted cost and chooses the best one. There are four stages to transform the logical plan presented by DataFrame to real executable codes. First, the optimizer analyzes the logical plan for solving the data's references. After that, Catalyst optimizes the original logical plan by rule-based optimization. Further, it takes the optimized logical plan to produce several possible physical plans and compares them based on cost-based optimization. In this stage, only join operations would trigger the cost-based optimization. Others are passed directly to the next stage. In the end, Catalyst tries to generate Java bytecodes by utilizing quasiquotes and AST, abstract syntax trees, in Scala to perform pattern matching. The transformation transforms the codes to optimized Java bytecodes for better performance executing on Spark engine.

SparkSQL also provides users with UDFs, user-define functions, and UDTs, user-defined types. For UDFs, it is rela-

tively simple. After registering the function for SparkSQL to recognize it, users could pass the function directly into DataFrame APIs. For UDT, there are two mapping functions needed for SparkSQL to recognize the type. One is the mapping function from the user-defined type to a Catalyst Row built-in type, and another one is the reverse mapping function from the built-in type to the user-define type. After registering the type, users could manage the type as if it is native in SparkSQL.

```
object DecimalAggregates extends Rule[LogicalPlan] {
  /** Maximum number of decimal digits in a Long */
  val MAX_LONG_DIGITS = 18

  def apply(plan: LogicalPlan): LogicalPlan = {
    plan transformAllExpressions {
      case Sum(e @ DecimalType.Expression(prec, scale))
             if prec + 10 <= MAX_LONG_DIGITS =>
        MakeDecimal(Sum(LongValue(e)), prec + 10, scale)
    }
}
```

Figure 10: Optimizaiton for logical plans.[1].

```
def compile(node: Node): AST = node match {
  case Literal(value) => q"$value"
  case Attribute(name) => q"row.get($name)"
  case Add(left, right) =>
    q"${compile(left)} + ${compile(right)}"
}
```

Figure 11: Transforming to Java bytecodes with quasiquotes and AST.[1].

2.5.3 Syntax

SparkSQL provides Java, Scala, and Python interface for users to interact with the DataFrame API. The APIs are in concise and declarative SQL style, which are easy for users to structure their codes and debugging. Even if users have the codes as the mixture of DataFrame API and SparkRDD's mapping functions, Catalyst optimizer could still optimize them together. Before actually entering the stage to perform the real data loading and computation, the optimizer performs type check and other analytics to ensure some possible run-time errors could be discovered in the early stage, instead of encountering errors in the expensive execution stage. For the rules written for the rule-based optimization, SparkSQL retains an extension point for developers to extend the set of rules with their own optimizing rules, especially for working with user-defined types. SparkSQL also provides schema inference to construct DataFrame from data stored in RDD. Unlike ORM, object-relational mapping, which carries out the expensive transformation on data to another format, the inferred transformation is in place. However, while trying SparkSQL with a weather dataset, the inference encountered some failures. After having a precise schema, the problem was finally solved.

```
case class User(name: String, age: Int)

// Create an RDD of User objects
usersRDD = spark.parallelize(
  List(User("Alice", 22), User("Bob", 19)))

// View the RDD as a DataFrame
usersDF = usersRDD.toDF
```

Figure 12: Creating a DataFrame.[1].

```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

Figure 13: Syntax of DataFrame.[1].

2.5.4 Performance

SparkSQL's performance highly relies on the optimization from Catalyst. While in [1], it is shown that the performance after the optimization on the same sequence of operations, SparkSQL DataFrame outperforms the original Spark RDD operations. Additionally, with the help of Catalyst, the performance bottleneck using Python shown in other above-mentioned systems is greatly alleviated. The optimizer transforms the logical plan written by users in Python to physical plans, and to Java bytecodes run in the Spark engine. Since the execution parts of the program are translated into Java bytecodes, it could avoid the bottleneck in Python from passing the data between the Python daemon process and the JVM where the Spark engine runs on.

2.6 Conclusion

From SciSpark[4], SparkArray[5], H5Spark[3], and SparkSQL[1], they proposed some valuable examples to modify Spark[6] for specialized needs and some worth noticing factors to take into consideration while building an array database management system on Spark. Spark's RDD and MapReduce structure provides users a clever and effortless way to handle computations over Big Data. While SciSpark introduces sRDD with substructure sciTensor to hold array data and make it possible to manipulate the data in Spark with linear algebraic operations. SciSpark sciTensor's two hash tables help realized the array-like data structure in Spark RDD. One of them stores additional information, and another of them stores the array data. For SparkArray, it stores schema as metadata to record the piece of data. The schema has a list for storing dimension values and a list for storing attribute values, which could detail describe a piece of array data. Also, it shows that similar array DBMS optimization techniques could also be applied to the array DBMS built on Spark, including merging the boundaries, overlapping the chunks, and repartitioning the chunks. For H5Spark, it also creates the special RDD with extra metadata stored. However, it places more emphasis on I/O optimizations, which is critical for loading a huge volume of data from file systems to the memory. While for SparkSQL, it also has a modified RDD with a schema called DataFrame. It demonstrates that the optimization from translating a logical plan formed by the sequence of user's jobs to interpreting the plan to lower-level Java bytecodes could not only

enhance the overall throughput and efficiency but also mitigate the known communication overhead between languages while using different language's interface. The elements, including having RDD with metadata storing dimension and attribute information, taking care of the optimizations specialized for array data, placing more emphasis on I/O optimizations in distributed environments, and abstracting user job organizations from how the underlying system executes, could give out some ways to build a better array database on Spark.

3. SECURITY ISSUES IN DBMS

Although there have been many efforts in the field to enhance securities in DBMS, [2] pointed out there is a noticeable gap between the simulated attack models in academic areas and the real use cases on running DBMS. Many past works mentioned in [2] focused on keeping the data stored in DBMS safe and secured from attacks and leakages by performing encryption on it. However, most of the DBMS used other auxiliary components, such as logs, caches, and in-memory data structures, to improve the performance or handle concurrencies. These unprotected parts are also vulnerable in some attacks, which could also cause DBMS unsafe. Aside from the known security issues listed and elaborated in part one, the second part of this section delves into Opaque[7], a variation of SparkSQL using encrypted enclaves and oblivious operators to protect computations on the data and hide the access patterns from malicious attackers. By considering the security issues in the paper[2] and implementation emphasis in Opaque, the proposal for a secured ADBMS on Spark could be more thorough and profound.

3.1 Known Security Issues

The work [2] emphasized the gap between simulated situations and the real situations on running DBMS, especially relational databases, under snapshot attacks. Many academic attack simulations assumed the system has no information about part queries. However, it is generally untrue in practical systems. The assisting data kept by DBMS for better efficiency of the systems could be a cost in the security domain. Some frequently used items, including logs, diagnostic tables, and in-memory data structures, could leak some information to the attackers, which place the DBMS in an unsafe state.

Most of the relational databases keep logs to hold past queries for efficiency reasons. The information kept in log files often contains detailed reading and writing information, such as updates, inserts, and deletes. Others could use the knowledge gain from the log files to reconstruct the current DBMS state, which exposes the data. Also, for MySQL, it keeps binlog (binary log) to handle concurrencies. In binlog, the undo and redo information could leak information. The timestamp inferred by primary keys in MongoDB would also be a concern. Although applying public-key encryption on these files could enhance its security, other information such as size and timing could still be revealed.

The diagnostic tables, for example, information_schema and performance_schema tables in MySQL, could leak the information of internal states and query statistics to attackers. The information_schema contains records of DBMS internal states, while performance_schema stores the statistics about past queries. The information about internal states accompanied by information in log files could provide attackers partial knowledge to hack the DBMS. While the statistics of past queries would reveal sufficient information about the access pattern, which would be discussed in the latter paragraph.

The in-memory data structures could also be a concern to DBMS's security. Some caches or heap structure in the memory are not encrypted, which are possible to leak the data. For example, in InnoDB, it keeps an adaptive hash index in the cache, which contains the content of past queries. Even if the information would be immediately deleted after use, if the attack intervenes in the process of a continuing query, the temporarily stored data could also be exposed. If in the case of having multiple threads running the query, the problem would be more severe. While in Spark, the DAG, directed acyclic graph, kept by each node for job sequence ordering could leak the whole image of the planned queries. Further, the event history stored in Spark for fault-tolerance is also vulnerable to leak past query information.

The access patterns gained from statistics in diagnostic tables, such as performance_schema, or in-memory data structures, could expose the side-information of the DBMS. Attackers could perform frequency analysis on a bipartite graph to find the most possible pairing from ciphertext to plain text by calculating the frequencies gain from the access patterns[2]. It was asserted in [2], that without trusted hardware, the DBMS would somehow leak access pattern information.

Further, mentioned in [7], an inference attack in join operations is also an issue. Attackers could learn some knowledge of the dataset from the primary-foreign key relationship between tables during join operations.

3.2 Opaque

Opaque[7] is a system based on SparkSQL designed for secured and oblivious data operations in an untrusted cluster. It adopts Intel SGX Enclave as the secured hardware. The enclave provides Opaque with a secure execution environment and data encryption services. Further, with oblivious operators, Opaque could hide access patterns from potential attacks.

3.2.1 Architecture

Opaque is based on SparkSQL, a relational DBMS approach on Spark. It chose to modify the query optimization layer in SparkSQL only to minimize the implementation codes and preserve the same usage as in SparkSQL. It separated the query planner and the scheduler. The query planner is placed on the client-side, which is trusted because query planning is critical for having the correct result. While the scheduler remains on the server-side. Opaque implemented the interface of Opaque operators in Scala where Spark based on. The operators pass input data by JNI

to the underlying enclave written in C++. While for the enclave, it would pass the data to its substructure, the assembly codes to access enclave's memory, EPC, and encrypt data with AES-GCM by calling provided Intel SGX APIs.

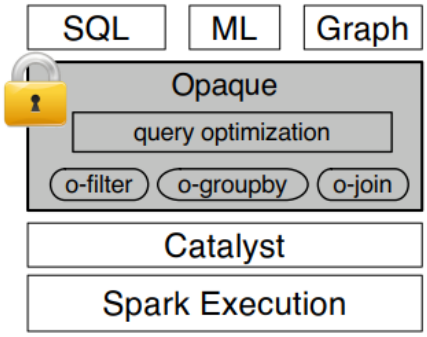

Figure 14: The Architecture of Opaque[7].

3.2.2 *Implementation*

The fundamental part in Opaque is the enclave created by Intel SGX APIs. It is created on the specially designed Intel SGX trusted hardware. Intel SGX promises data confidentiality and secure execution while processing data. Data within the enclave would be encrypted with AES in GCM mode. Further, Opaque also ensures data integrity. It applies a 128-bit MAC on each partition of data to achieve remote attestation for loading correct codes into the enclave, to prevent them being tampered.

The obliviousness was achieved by the basic oblivious operators. Regarding the fact that obliviousness is expensive, Opaque mitigates the overhead with the two-part solution, including distributed oblivious SQL operators and adjusted query planning optimization. Operators are decomposed to smaller distributed oblivious operators and optimized with the oblivious sense in the global view.

For the oblivious operators, they are redesigned to hide access patterns and size information. The most important oblivious operator is the oblivious sort operator. It could fulfill most of the need of existing relational operators. It adopts column sort which shuffles, reorders, and sorts data to wipe out the access pattern while sorting[7]. Another example of oblivious operator is filter. To ensure the size of the data would not be leaked, Opaque abandoned the common strategy of eliminating unused data in the early stage of the filtering. Instead, it adds a column to the row of data indicating whether or not the row should be filtered out in the future, and it performs the real filtering at the very last stage.

Additionally, Opaque also provides the padding mode. It uses dummy data to pad the dataset to a designed number once needed. And, it only drops the dummies at the very last stage to keep the size unchanged. With the paddings, it greatly protects the size information.

The optimization was based on Catalyst from SparkSQL with some changes to attain obliviousness. The optimization also contains two types, rule-based and cost-based. Rule-based optimization is very similar to the original version. It contains the two-part solution, which decomposes the operators to basic ones and optimizes them globally with obliviousness. Unlike original Catalyst performs filter push down to trim the size of the dataset at the early stages, Opaque performs filter push up to secure the size information. Also, the dummy lines in the padding modes as mentioned in the last paragraph would only be eliminated at the ending stage due to a similar reason. These strategies as well as the shuffling stage in the oblivious sorting increase the network I/O, which incurs great overhead. However, with the optimization, it greatly alleviates the known high cost for realizing obliviousness. Apart from the rule-based, cost-based optimization was redesigned to take the expensiveness of oblivious operations into account. It utilizes the second path analysis on join operations with tables in both sensitive and insensitive categories. The tables on the way from the leaves of sensitive tables to the root of the final joined table should all be noted as sensitive, to prevent possible inferences of any sensitive information. The optimization tends to lower the number of needed oblivious operations, which is the desired result.

3.2.3 *Syntax*

Since Opaque focused on modifying the query optimization layer, most of the relational operators in SparkSQL are preserved. Only a few lines of codes are needed for users to encrypt the data. Other usages are identical to the originals.

```
import edu.berkeley.cs.rise.opaque.implicits._
edu.berkeley.cs.rise.opaque
 .Utils.initSQLContext(spark.sqlContext)

val data = Seq(("foo", 4), ("bar", 1), ("baz", 5))
val df =
 spark.createDataFrame(data).toDF("word", "count")
val dfEncrypted = df.encrypted

dfEncrypted.filter($"count" > lit(3)).explain(true)
// [...]
// == Optimized Logical Plan ==
// EncryptedFilter (count#6 > 3)
// +- EncryptedLocalRelation [word#5, count#6]
// [...]

dfEncrypted.filter($"count" > lit(3)).show
// +----+-----+
// |word|count|
// +----+-----+
// | foo|    4|
// | baz|    5|
// +----+-----+
```

Example usage of Opaque.(From Opaque GitHub page, https://github.com/ucbrise/opaque.)

3.2.4 *Performance*

While Opaque provides encryption, secure execution, and obliviousness, the overhead is the main issue. From the testing on Opaque and Spark with a weather dataset, having some aggregation operators and filter-simulated slicing, subsetting operators, overheads ranging from 15X to 40X were observed, which are consistent with the range 1.6X-46X provided in the paper[7]. The huge overheads are resulting from having a huge sensitive dataset, which invalidates the cost-based optimization.

3.3 Conclusion

Data itself could be secured by applying encryptions on it. However, other by-products of a running DBMS, including logs, diagnostic tables, in-memory structure, could also be the concerns for security. Access pattern information could also make data in DBMS vulnerable. While for Opaque, it has enclaves to ensure data confidentiality and secure execution. It also realizes obliviousness against access pattern leakage by implementing the oblivious operators and redesigning the optimization rules for both hiding size information and mitigating the huge overhead incurred by obliviousness.

4. PROPOSAL FOR A SECURED ADBMS ON SPARK

The proposed secured ADBMS on Spark would be based on SciSpark's approach. The DBMS has the main structure, secured scientific RDD (ssRDD). It is a variation of Spark original RDD, and it contains a substructure, secured sciTensor (ssciTensor). The ssciTensors hold the array data in chunks. With Intel SGX Enclave and oblivious operations adopted in Opaque, the operations on ssciTensor could ensure data confidentiality, secure execution, and access pattern hiding. The system also provides the padding mode similar to Opaque's. However, the huge overhead for the padding mode is expected. Moreover, the following parts would demonstrate the target attack scenario, architecture of the proposed DBMS, possible implementation methods with an example on range queries, and expected performance evaluation.

4.1 Scenario

The scenario demonstrates how an attacker could confirm the relationship between which partition the time-geographic data is allocated and its dimension values from leaked access patterns. The attacker could even reveal the moving track of a certain item.

The dataset stores the time and location values (normalized coordinate x and y values) and is distributed to three partitions. For example, the dataset contains a four times four plane and three time slices representing the snapshot of items' locations at 3 equally spaced time. While the attacker could observe the query commands from the transferring from the client-side to the server cluster. He could also observe the transferring data and its size information from the network I/O between nodes.

From numerous different range queries, the attacker could obtain plenty of information on which partition the data is stored in. Since the attacker could observe the network I/O between nodes, by analyzing the result size and finding out the overlapping parts of each range query, he could understand the locations of each item in the partitions. The analysis includes finding out the overlapping data access in the same partition from the range query's access pattern. The precision of the locations found depends on the granularity of each range query. With some extra information, such as knowing when and where the items show up the most, the attacker could identify the paring relationship between the data allocation in the partitions and its dimension values. Although only the paring relationship could be identified, instead of knowing exactly the data's identity, it could still reveal the data distribution of the whole dataset. Similarly, attackers could obtain the knowledge of data distribution from the full reorder join, which finds out the data with different locations in different time slices. The method to reveal the data distribution is similar to the one used in the case for multiple range queries. The full reorder join itself leaks information of changing parts while having the network I/O between partitions, which mechanism is close to the overlapping part analysis in range queries.

With the knowledge of the data distribution, the attacker could even obtain the tracking information by comparing the distribution difference between time slices. If the items stored in the database are locations and time of some shared bikes, it would immediately fulfill an assumption that the displacement of the location of a bike between time slices is relevantly small. Hence, the attacker could utilize the assumption to track the bikes' movements.

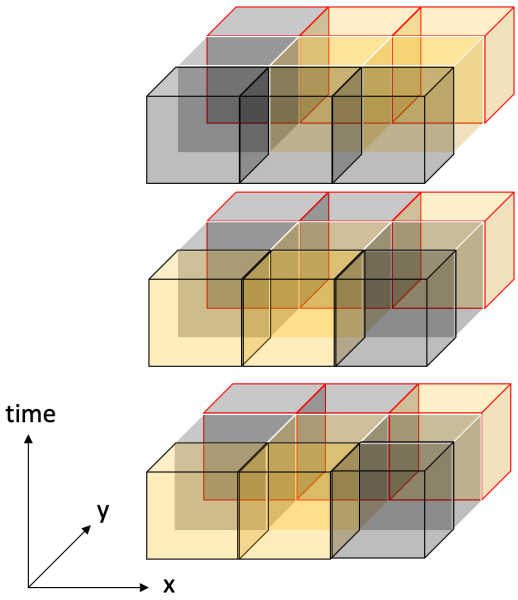

Figure 15: The Dataset in the Scenario. Yellow blocks represent locations containing items. While black, white, and red grid lines define three partitions.

4.2 Architecture

The architecture of the system would be based on SciSpark with the secured enclave and oblivious operations supported in Opaque. In the system, there are three main parts, including ssRDD as well as ssciTensor, Enclave, and oblivious executions. The ssRDD is based on Spark original RDD and contains ssciTensor as its substructure. An ssRDD contains multiple ssciTensor where ssciTensor holds a chunk of array data. In ssciTensor, the chunk stores cells containing attribute values. In the distributed environment provided by Spark, within ssRDD, ssciTensors are separated in partitions. Further, while performing operations, computations would be sent to the enclave's memory and fenced, providing secure execution. While for oblivious operations, in common sorting stages, the system adopts the column sort, which is assured to be oblivious[7].

4.3 Implementation

The substructure ssciTensor contains three fields for storing different kinds of data, for instance, dimensions, the chunk, and operator extensions. The dimension field stores the designated point of the array data chunk, for example, the point with smaller coordinate values. While the field of the chunk stores a cube of cells containing attribute values of each data point. Some linear algebraic operations could be easily applied to the array. However, at the current stage, only linear algebraic operations within the same chunk are allowed since the access pattern issues of the operations applying on data across different chunks still need some future development. Further, the operator extension field stores the result of every basic unit operations. The need for the field arises from the fact that the system does filtering and other operations at the ending stage and even pads dummy chunk of data to ssRDD for preventing the information leaked from the size changed. For example, for the range query, some chunks would be fully in the query range, while others would be partially in it or not in it. The system then assigns the field of the ssciTensor with 2 to fully, 1 to partial, or 0 to none in the range. After performing other operations, the real filtering stage filters the data depends on the extension fields.

The secure execution and the data encryption would rely on the enclave. The system sends the execution parts to the enclave to protect the information of the dataset itself and the in-memory products.

The DBMS would load the data from the file system with its original sequence. While during operations, the data would be shuffled. The oblivious operations would rely on the column sort adopted by Opaque. For example, for range queries, the system would first label the ssciTensor indicating how many data points in the range by recording the number in the operator extension field. After finishing labeling, the column sort would obliviously shuffle and sort the data. Finally, the range of the data would be filtered out based on the sorted sequence. Also, boundary cases should be handled.

The system optimizes the operations by breaking down the operators into basic unit operators and perform optimization globally, which is similar to Opaque.

While the system also provides padding mode for hiding the size information while transferring the data, the huge overhead demonstrated in [7] could be a concern.

4.4 Performance

While discussing the performance of the system, there are two main areas of concern. One is related to the size of the chunk stored in the ssciTensor. While having too big chunk size would lead to huge transferring overhead between partitions. While for smaller chunk size would lead to some security concern. Since the precision of overlapping analysis shown in the scenario depends on the granularity of the range query, having a smaller chunk size would make it easier for attackers to identify more precise data distribution. Also, having too small chunk size would make the linear algebraic operations on the dataset incur larger overhead. Besides the issues related to the chunk size, adopting oblivious operations would incur huge overhead. It could be an issue. Since the purpose to have an array operator is to speed up the processing for multidimensional data. While on the other hand, obliviousness requires shuffling, padding, and dummy data processing to protect access patterns, which slows down the overall performance.

4.5 Future Development

The system is still a proposal where more works would be needed to implement the prototype. Also, more aspects of range queries could be explored, such as comparing range queries with independent chunks and overlapping chunks. Further, more operations could be taken into consideration, such as join, broadcast join, and aggregations.

5. CONCLUSION

While Spark has the advantage to manipulate the huge volume of data, there are still some works needed to accommodate array data into Spark RDD. The proposal takes the approach of SciSpark, which separates the structure into two levels. ssRDD represents the whole dataset while handling the shuffling and data transferring works, while ssciTensor holds the chunk of array data to provide convenient multidimensional data related operations. However, more details needed to be discussed to have a suitable chunk size and handle the boundary cases in an oblivious manner while applying operations on array data. Although having secured and oblivious operations would incur relevant huge overhead, it is still worth a try to explore the possibility to build a secure and oblivious ADBMS on Spark, especially for those sensitive time-geographical datasets.

6. ACKNOWLEDGMENTS

The UROP project is supervised by DR. NG, WILFRED SIU HUNG and guided by Mr. CHEN, Guo with helpful assistance.

7. REFERENCES

[1] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD*

international conference on management of data, pages 1383–1394. ACM, 2015.

[2] P. Grubbs, T. Ristenpart, and V. Shmatikov. Why your encrypted database is not secure. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 162–168. ACM, 2017.

[3] J. Liu, E. Racah, Q. Koziol, R. S. Canon, and A. Gittens. H5spark: bridging the i/o gap between spark and scientific data formats on hpc systems. *Cray user group*, 2016.

[4] R. Palamuttam, R. M. Mogrovejo, C. Mattmann, B. Wilson, K. Whitehall, R. Verma, L. McGibbney, and P. Ramirez. Scispark: Applying in-memory distributed computing to weather event detection and tracking. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2020–2026. IEEE, 2015.

[5] W. Wang, T. Liu, D. Tang, H. Liu, W. Li, and R. Lee. Sparkarray: An array-based scientific data management system built on apache spark. In *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–10. IEEE, 2016.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[7] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 283–298, 2017.