Large-scale Complex Analytics on Semi-structured Datasets using AsterixDB and Spark

Wail Y. Alkowaileet #1, Sattam Alsubaiee #1, Michael J. Carey +*2, Till Westmann *3, Yingyi Bu *3
#Center for Complex Engineering Systems at KACST and MIT
+University of California, Irvine
*Couchbase
1{walkowaileet, ssubaiee}@kacst.edu.sa, 2mjcarey@ics.uci.edu,
3{till,yingyi}@couchbase.com

ABSTRACT

Large quantities of raw data are being generated by many different sources in different formats. Private and public sectors alike acclaim the valuable information and insights that can be mined from such data to better understand the dynamics of everyday life, such as traffic, worldwide logistics, and social behavior. For this reason, storing, managing, and analyzing "Big Data" at scale is getting a tremendous amount of attention, both in academia and industry. In this paper, we demonstrate the power of a parallel connection that we have built between Apache Spark and Apache AsterixDB (Incubating) to enable complex analytics such as machine learning and graph analysis on data drawn from large semi-structured data collections. The integration of these two systems allows researchers and data scientists to leverage AsterixDB capabilities, including fast ingestion and indexing of semi-structured data and efficient answering of geo-spatial and fuzzy text queries. Complex data analytics can then be performed on the resulting AsterixDB query output in order to obtain additional insights by leveraging the power of Spark's machine learning and graph libraries.

1. INTRODUCTION

In the last decade, the MapReduce programming model led to a breakthrough in terms of the wide scale adoption of distributed computing. By transparently providing automatic parallelization, load balancing, and fault tolerance, Hadoop (which is the open-source implementation of MapReduce) has allowed developers to focus on writing map( ) and reduce( ) functions without worrying about the complexity of distributed processing. However, due to its rigid policies of materializing intermediates results (for fault-tolerance purposes) and always shuffle-sorting those results, Hadoop was found to be inefficient for running machine learning algorithms and interactive analytics, both which require repeated reads over the same data [4] [3] [7].

In response to this limitation, Spark [6] was introduced. By caching its intermediate results in main memory, Spark can outperform Hadoop by orders of magnitude when running iterative algorithms such as Logistic Regression. In addition to its performance advantages, Spark provides developers with more complex operators such as filter and join. These factors have resulted in the recent wide adoption of Spark, causing it to become the new de facto standard for big data processing. One limitation of Spark, however, is that it does not provide a storage engine with advanced indexing capabilities; it only scans data from HDFS. To address this issue, Spark has been connected with other systems that do offer indexing capabilities, such as Cassandra, HBase, MongoDB, and ElasticSearch. Each of these systems has its own characteristics and features that make it suitable for a particular workload. This paper describes how we have connected the AsterixDB [1] Big Data Management System (BDMS) with Spark.

AsterixDB has leveraged ideas drawn from parallel databases, semi-structured data stores, and first-generation big data platforms to create a new breed of Big Data Management System (BDMS). It can store, index, and query multiple datasets, with structures ranging from rigidly-typed relational datasets to flexible and complex datasets whose objects are heterogeneous and self-described. AsterixDB differs from other NoSQL datastores by having a powerful query language that supports full, multi-collection queries as opposed to just simple, single-collection selections and aggregates. A direct connection between Spark and AsterixDB can thus provide additional input-forming and filtering power. Given such a connection, Spark developers and data scientists will be able to leverage AsterixDB's powerful query language and advanced indexing capabilities to efficiently explore their big semi-structured data.

In this paper, we demonstrate the parallel connection of these two big data systems, AsterixDB and Spark, using a new AsterixDB-Spark connector (called "the connector" hereafter). The connector allows users to run queries against AsterixDB datasets to efficiently fetch and process only the subset of data that they need and then to process the resulting output in Spark. The connector leverages the parallelism and coexistence of both systems by enabling Spark to read the results of AsterixDB queries in parallel, and it exploits data-locality when possible. Such parallel inter-system connectivity is crucial in the world of big data. We also introduce an AsterixDB Schema Inferencer, one that is capable of inferring the schema of homogeneous or heterogeneous semi-structured data that results from an AsterixDB query.

The rest of the paper is structured as follows: Section 2 provides an overview of the user models and internal components of both Spark and AsterixDB. Section 3 discusses the details of our AsterixDB-Spark parallel connection, covering the whole path from AsterixDB query submission to running complex analytics on Spark. Finally, Section 4 describes the demonstration scenarios that will be shown to illustrate the combined capabilities of the two systems.

Proceedings of the VLDB Endowment, Vol. 9, No. 13

2. BACKGROUND

In this section, we briefly review the user model and internal components of Spark and AsterixDB.

2.1 Apache Spark

Spark uses an abstraction called Resilient Distributed Dataset (RDD) [7] for performing parallel in-memory computations on partitioned data collections with efficient fault-tolerance. There are two types of operations that can be performed on RDDs. The first is called a *transformation*, which is an operation that generates a new RDD from an existing RDD to reflect the desired data transformation. The second is called an *action*, which is an operation that generates a computed value from an existing RDD. Transformations (such as filter, map, join, union, etc.) can be chained to form a Directed Acyclic Graph (DAG) that represents a Spark Job. RDD transformations are executed lazily (i.e. no computation occurs) until there is some action operation (such as count, reduce, etc.) that starts the computation and returns the result.

Spark applications can be written in any of four programming languages: Scala, Java, Python and R. Additionally, Spark provides a programming shell where the user can interactively write a Spark application. Moreover, the Spark community has incrementally added many higher-level libraries on top of Spark Core (i.e., Spark RDDs) to execute SQL queries against Spark's DataFrames (which are RDDs with schemas), to run machine learning algorithms, and to perform graph analysis. Figure 1 shows how easy it is to write a scalable word count task in Spark using its transformations and actions; Figure 2 shows an alternative way to accomplish the same word-count task using SparkSQL.

```
val lines = sc.textFile("file.txt")
val words = lines.flatMap(line => line.split(" "))
val wordCount = words.map(word => (word,1))
    .reduceByKey(_ + _).collect()
wordCount.foreach(println)
```

Figure 1: A Spark word-count program using RDD transformations and actions.

```
val lines = sc.textFile("file.txt")
val words = lines.flatMap(line => line.split(" "))
case class Word(word: String)
val dataFrame = words.map(w => Word(w)).toDF
dataFrame.registerTempTable("wordsTable")
sqlContext.sql("""
      SELECT word, count(*) FROM wordsTable GROUP BY word
""").show()
```

Figure 2: A Spark word-count program using SparkSQL aggregate query on a DataFrame.

2.2 Apache AsterixDB

AsterixDB is a parallel big data management system that is capable of efficiently ingesting, storing, indexing, and querying large semi-structured datasets. It can manage relational-like datasets as well as complex and flexible ones. The AsterixDB Data Model (ADM) extends JSON to include a larger set of data types, such as `datetime` and `polygon`, that can be specified using the AsterixDB Data Definition Language (DDL). Figure 3 shows an example of three ADM types called `TwitterUserType`, `TweetType` and `StateType`, that can be used to model Twitter data and US States data, respectively. The example shows the ADM support for optional fields (`location`), nested types (`user`), collection types (`hashtags`) and various spatial types (`location` and `shape`). By specifying some or all the schema up front, a user can permit the ingestion of instances that must follow a certain structure without sacrificing the flexibility of allowing additional fields that are not described a priori. Queries for AsterixDB are written in AQL (AsterixDB Query Language). AQL is a declarative query language that draws ideas from XQuery, mainly its FLWOR expression and its composability, adapting it to query ADM datasets. Figure 4 shows a query that searches for tweets that contain certain keywords in a specific period of time. To determine the sender location, a spatial join is performed between TweetDataset and StateDataset using the `spatial-intersect` function.

AsterixDB accepts an AQL query through its HTTP-based API and returns the result either synchronously or asynchronously (where a handle to the result is returned and the user can inquire about the result status and retrieve the result using the handle). When a user submits an AQL query, AsterixDB compiles and optimizes it into a Hyracks job [3] for parallel execution. The role of Hyracks is to serve AsterixDB as its parallel runtime engine. AsterixDB utilizes partitioned LSM-Based [2] primary and secondary indexes (B^+-Tree, R-Tree, and text indexes) to answer queries efficiently and produces partitioned results in parallel. All partitioned results are then collected and returned to the user through the HTTP API.

```
create type TwitterUserType {
    screen-name: string,
    lang: string,
    friends_count: int32,
    statuses_count: int32,
    name: string,
    followers_count: int32
};
create type StateType {
  name: string,
  code: string,
  shape: polygon
};
```

```
create type TweetType {
    tweetid: int64,
    user: TwitterUserType,
    location: point?,
    time: datetime,
    hashtags: {{ string }},
    message: string,
    userid: int32
};
```

Figure 3: Modeling Twitter and US states data in ADM.

```
for $state in dataset StateDataset
for $tweet in dataset TweetDataset
let $keywords := ["NFL", "superbowl","semi final"]
let $location := $tweet.location
let $shape := $state.shape
let $message := $tweet.message
let $stateCode := $state.code
where spatial-intersect($location, $shape)
and (some $word in $keywords
   satisfies contains($message, $word))
and $tweet.time >
   datetime("2015-05-02T00:00:00Z")
and $tweet.time <
   datetime("2015-12-02T00:00:00Z")
return {"state-code":$stateCode, "message": $message}
```

Figure 4: An example of AQL query that involves spatial-join and textual search.

3. COMBINED SYSTEM OVERVIEW

The connector is a library that can be loaded into a Spark application (or shell) to allow AsterixDB and Spark to interact seamlessly as one combined system. It enables Spark users to leverage AsterixDB's data model and indexing capabilities to efficiently fetch only the subset of data that they need and then process it in Spark.

This is achieved by connecting the large, sharded query-result partitions from AsterixDB to Spark, as opposed to linking the two parallel systems through a thin (serial) pipe.

Consider the example configuration presented in Figure 5, which shows a cluster running both AsterixDB (Asterix Node Controller) and Spark (Spark Worker) instances. To keep this example simple, both AsterixDB and Spark instances coexist on every node in the cluster; however, our connector does not require the two systems to coexist on the same nodes. In the example shown, each node has three disks that are used to store AsterixDB datasets and each has three configured Spark Executors that execute Spark jobs. The Asterix Cluster Controller and Spark Driver are orchestrating, managing, and monitoring the execution of all of the AsterixDB (Hyracks) and Spark jobs on the cluster, respectively.

A typical coupled-system interaction will involve three consecutive steps: submission of an AQL query from Spark to AsterixDB, execution of the query in AsterixDB to produce the result partitions (with inferred schemas), and loading of the result partitions into Spark for analytics. In the following, we discuss each step in detail by using the sample Spark application shown in Figure 6. This application, which is written in Scala, performs sentiment analysis on a Twitter dataset that is stored in AsterixDB. Specifically, the analysis targets tweets about the Super Bowl posted within three days of the game. To do so, the application submits the AQL query shown in Figure 4 to efficiently fetch only those tweets that match the search criteria and to load it to Spark as a DataFrame collection. The application then utilizes Spark's machine learning library to count the number of positive tweets for each US state using a trained *Naïve Bayes* model.

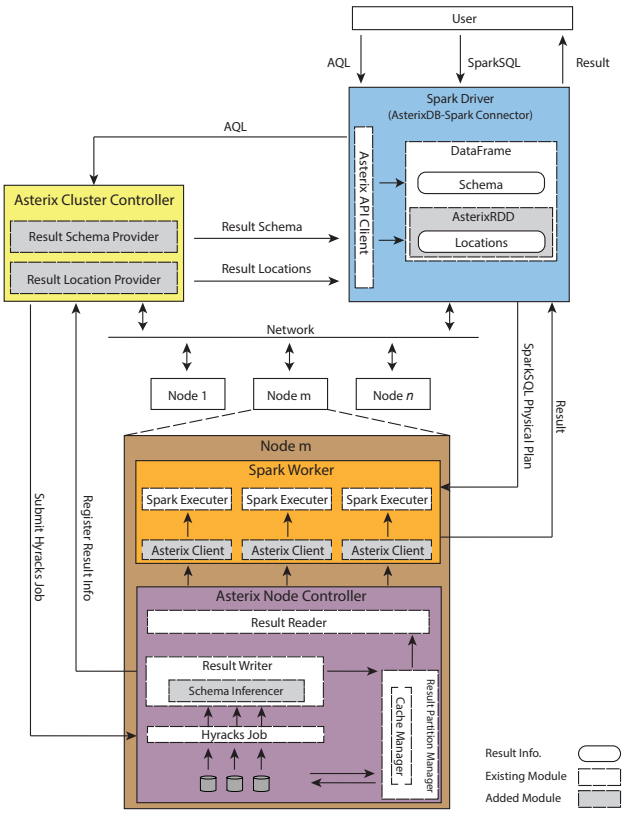

Figure 5: AsterixDB-Spark Connector detailed architecture.

```
val aqlQuery = /* AQL Query from Figure 4 */
val tweets = sqlContext.aql(aqlQuery)
val NBPipeline = Pipeline.load("/path/NBModel")
NBPipeline.transform(tweets).select("state-code, label")
      .filter("label = 1")
      .groupBy("state-code").agg(count("label"))
      .show()
```

Figure 6: A Spark application that submits a query to AsterixDB to fetch tweets that discuss the Super Bowl game and then perform sentiment analysis on them using a trained Naïve Bayes model.

3.1 Query Submission

Within a Spark application or a Spark shell, the user can submit AsterixDB AQL queries using the connector, as shown in Figure 6. The connector internally uses AsterixDB's HTTP API to submit the query to the Asterix Cluster Controller (CC) and specifies the required output result format as JSON[1]. Similar to any AQL query that is submitted to AsterixDB, the CC then compiles and optimizes the query to produce a Hyracks job and submits it to all registered Asterix Node Controllers (NCs) for execution.

3.2 Query Execution and Schema Inference

Once the Hyracks job is submitted, each NC will execute the compiled query and materialize the result as binary JSON partitions. The number of partitions generated in each NC will match the number of I/O devices (storage disks) configured in Hyracks. For example, in Figure 5, there are three I/O devices per node. Hence, the expected number of result partitions in each node is three. (The mapping between Hyracks result partitions and Spark partitions is explained in Section 3.3.)

Usually, when Spark loads a JSON file to create a DataFrame, the schema of the records in the file must be available. There are two options to obtain the schema: either the user provides it manually, which can be tedious especially when dealing with semi-structured datasets, or Spark infers the schema automatically by scanning the whole file, which can obviously be a costly operation. Some other Spark connectors (e.g., the MongoDB connector [8]) infer a schema by sampling records. However, sampling does not guarantee the correctness of the resulting schema when the data is semi-structured, which can cause subsequent Spark queries on the data to fail. To address this issue, we introduced a new Schema Inferencer module in AsterixDB that constructs the schema of the result partitions on the fly while materializing the result without an additional pass. The inferencer infers the types from the data as appropriate. For instance, the type of the field `name` in Figure 7 is inferred as `String`. In addition, the schema inferencer marks every field that does not appear in all records (e.g., the `salary` field appears only in the first record) as an optional field.

Some records in an AsterixDB dataset may contain fields with the same name but with different data types. For instance, the two JSON records shown in Figure 7 could coexist in a single AsterixDB dataset. As can be seen, the `dependents` field in the first record is an array of JSON objects representing the dependents' names. However, in the second record, the `dependents` field is an array of integers corresponding to the IDs of the dependents. We refer to fields with such a data type disparity as *heterogeneous fields*. In the case of Spark, heterogeneous fields are always inferred as `String`. To comply with Spark, our schema inferencer resolves heterogeneous fields as `String`. However, the schema inferencer

[1]AsterixDB supports other output formats such as CSV and ADM.

```
{                                    {
  "id": 1,                             "id": 2,
  "name": "Alice",                     "name": "Dane",
  "dependents":[                       "dependents": [5, 23]
     {"name": "Bob"},                }
     {"name": "Carol"}],
  "salary":68000
}
```

Figure 7: Two JSON documents with heterogeneous and homogeneous fields.

provides the developers with a generic interface that can be implemented to decide how to resolve data type disparities. For example, another implementation of this interface could resolve heterogeneous fields as the union of the conflicting field types.

3.3 Result Reading and Processing

When a user triggers the execution of the Spark job (e.g., by calling the `show()` function in the last line of Figure 6, which is an action that prints the results in a tabular format), the connector asks the CC for the result locations and the inferred schema. This information is used by Spark to construct the required DataFrame as follows: The SparkSQL optimizer generates an optimized physical plan and submits it as a Spark job to all Spark workers in the cluster. To exploit data locality, the locations of the AsterixDB result partitions are utilized by Spark to assign the job to those Spark workers that are co-located with the query result partitions. If there is an assigned Spark worker that does not coexist with a result partition, then the Asterix Client (shown in Figure 5) reads the result remotely from the other node. Within each Spark worker, multiple Spark Executors read the result partitions, deserialize the binary JSON results, and collectively construct an AsterixRDD. The connector then constructs a DataFrame by applying the inferred schema on the resulting AsterixRDD without needing an additional pass over the data. Since the number of Spark executors may exceed the number of result partitions, a one-to-one mapping could lead to some Spark executors being idle. To avoid this, the connector utilizes the RDD's `repartition` mechanism to let multiple executors read from the same partition (i.e., two or more Spark executors can read from the same result partition).

Returning to the example of Figure 6, the Naïve Bayes Model `NBPipeline` takes the DataFrame `tweets`, the result of the AQL query, as an input and classifies each tweet as positive (labeled by 1) or negative (labeled by -1). Finally, the tweets for each state are filtered for positive sentiment and a count of these positive tweets is printed.

4. DEMONSTRATION SCENARIO

In the demonstration, we will show several scenarios using two real datasets: a Twitter dataset and a publications dataset. During the demonstration, we will interact with AsterixDB and Spark through the connector and visualize the results using Apache Zeppelin notebooks [9]. Figure 8 shows a sample screen shot from the demonstration interface. In each scenario, the audience will be given a walk-through tour that interacts with both systems using the connector. Additionally, we will show the output of the AsterixDB schema inferencer. We will also show how the inferencer can be used to generate schemas for semi-structured datasets in order to produce the required Scala `case class` for the new experimental Spark `Datasets`[2].

The details of the two AsterixDB/Spark platform interoperation scenarios are as follows:

[2]https://spark.apache.org/docs/latest/sql-programming-guide.html#datasets

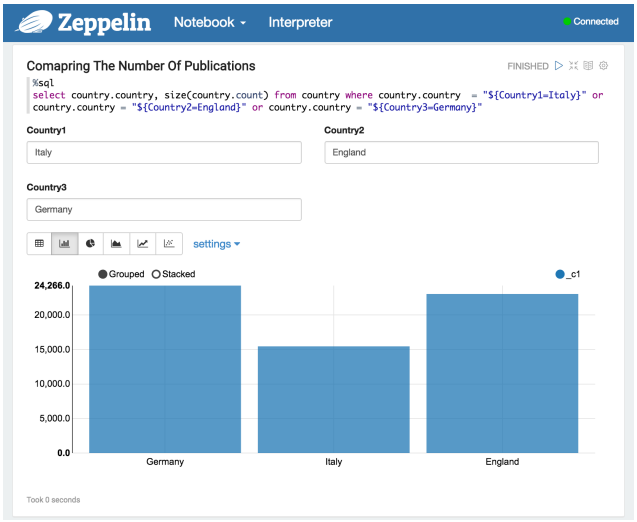

Figure 8: A sample visualization of the interface that will be presented in the demonstration.

Spatiotemporal Sentiment Analysis: In this scenario, we will show how to utilize both AsterixDB and Spark to detect the sentiment of tweets. In particular, the goal is to understand the impression about a topic in each US state during a certain time window. We will accomplish this using AsterixDB's capabilities to efficiently retrieve tweets based on their location, time, and textual content. Afterwards, we will use Spark to iteratively go over each state's tweets and run the trained sentiment analysis model. Finally, we aggregate the result and visualize it on a heat map.

Scholar Field Prediction: Many scholars publish in many disciplines during their career. For example, a computer scientist might start her career in theoretical computer science and later shift to computer systems. In this scenario, we will show a way of predicting the next field that a scholar might publish in using the methodology explained in [5]. To do so, we will use AsterixDB to construct the field relation graph by creating an edge between any two fields if there is an author who published in both of them. From the field network, we will use Spark to identify the statistical significance between fields which then can be used as a model for prediction.

5. REFERENCES

[1] S. Alsubaiee et al. AsterixDB: A scalable, open source BDMS. *Proc. VLDB Endow.*, 7(14), 2014.

[2] S. Alsubaiee et al. Storage management in AsterixDB. *Proc. VLDB Endow.*, 7(10), 2014.

[3] V. Borkar et al. Hyracks: A flexible and extensible foundation for data-intensive computing. *ICDE*, 2011.

[4] Y. Bu et al. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2), 2010.

[5] M. R. Guevara et al. The research space: using the career paths of scholars to predict the evolution of the research output of individuals, institutions, and nations. 2016. arXiv:1602.08409 [cs.DL].

[6] M. Zaharia et al. Spark: Cluster computing with working sets. In *Proc. HotCloud*, 2010.

[7] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI, 2012.

[8] MongoDB-Spark Connector: White Paper. https://www.mongodb.com/collateral/apache-spark-and-mongodb-turning-analytics-into-real-time-action.

[9] Apache Zeppelin. https://zeppelin.incubator.apache.org/.