Efficient spatio-temporal event processing with STARK

Stefan Hagedorn
TU Ilmenau, Germany
stefan.hagedorn@tu-ilmenau.de

Timo Räth
TU Ilmenau, Germany
timo.raeth@tu-ilmenau.de

ABSTRACT

For Big Data processing, Apache Spark has been widely accepted. However, when dealing with events or any other spatio-temporal data sets, Spark becomes very inefficient as it does not include any spatial or temporal data types and operators. In this paper we demonstrate our STARK project that adds the required data types and operators, such as spatio-temporal filter and join with various predicates to Spark. Additionally, it includes k nearest neighbor search and a density based clustering operator for data analysis tasks as well as spatial partitioning and indexing techniques for efficient processing. During the demo, programs can be created on real world event data sets using STARK's Scala API or our Pig Latin derivative *Piglet* in a web front end which also visualizes the results.

1. INTRODUCTION

Spatio-temporal data is used in various application areas: for example by (mobile) location aware devices that periodically report their position as well as in news articles describing *events* that happen at some time and location. Spatio-temporal event data can, e.g., be extracted from text documents using spatial and temporal taggers that identify the respective expressions in a text corpus. The extraction of the structured event data from text is just a first step and data needs to further be analyzed using appropriate data mining operations to gain new insight.

As the event data sets may become very large, scalable tools are needed for the event analysis pipelines. Apache Spark has become a very popular platform for such Big Data analytics because of its in memory data model that allows much faster execution than with Hadoop MapReduce programs. However, Spark has a general data model which does not take the spatial and temporal aspects of the data into account, e.g., for partitioning. Furthermore, dedicated data types and operators for this spatio-temporal are missing.

In this paper we demonstrate our STARK[1] framework for scalable spatio-temporal data analytics on Spark, with the following features:

- STARK is built on top of Spark and provides a domain specific language (DSL) that seamlessly integrates into any (Scala) Spark program.
- It includes an expressive set of spatio-temporal operators for filter, join with various predicates as well as k nearest neighbor search.
- A density based clustering operator allows to find groups of similar events.
- Spatial partitioning and indexing techniques for fast and efficient execution of the data analysis tasks.

In contrast to similar existing solutions for Spark, STARK is the only framework that addresses not only spatial but also spatio-temporal data. Unlike other frameworks, STARK is seamlessly integrated into the Spark API so that spatio-temporal operators can directly be called on standard RDDs. Furthermore, we provide a Pig Latin extension in our Piglet engine to create (spatio-temporal) data processing pipelines using an easy to learn scripting language. A web front end supports users with interactive graphical selection tools and also visualizes the results. We evaluated STARK in a mirco benchmark against other solutions and showed that we can outperform them.

2. THE STARK FRAMEWORK

STARK is tightly integrated into the Apache Spark API and users can directly invoke the spatio-temporal operators and their RDDs. To achieve this, we created new data type and operator classes that make use of already existing Spark operations, but also extend internal Spark classes. Figure 1 gives an overview of STARK's architecture and its integration into Spark.

In the following, we describe the internal components for spatial partitioning and indexing as well as the API/DSL for spatio-temporal operations and integration into Spark.

2.1 Partitioning

Partitioning has a significant impact in data parallel platforms like Spark. If the partitions sizes, i.e., the number of elements per partition, are not balanced, a single worker node has to perform all the work while other nodes idle.

Spark already includes partitioners, but they do not exploit the spatial (or spatio-temporal) characteristics. Spatial-temporal partitioning means that partitions are not created by using, e.g., a simple hash function, but by considering the location in space and/or time of occurrence. Thus, after

[1] https://github.com/dbis-ilm/stark

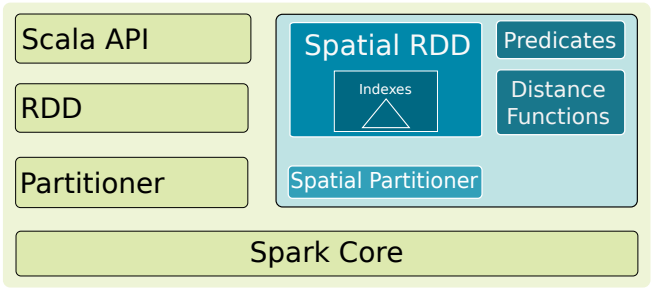

Figure 1: Overview of STARK architecture and integration into Spark.

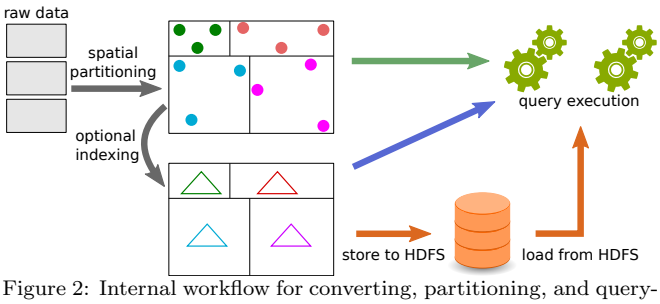

Figure 2: Internal workflow for converting, partitioning, and querying spatio-temporal data

a spatio-temporal partitioner was applied on a data set, a partition contains all elements that are near to each other in time and/or space and the bounds of a partition represent a spatial region and/or temporal interval which cover all items of that partition. This bound is very useful to determine what partitions actually have to be processed for a query. For example, an *intersects* query only has to check the items of partitions where the partition bounds themselves intersect with the query object. Such a check can decrease the number of data items to process significantly and thus, also reduce the processing time drastically.

When the spatial and temporal objects of a data set are not points or instants, respectively, these regions and intervals may span across multiple partitions. There are two options to handle such scenarios:

- The item is replicated into every of these partitions and the resulting duplicates have to be pruned afterwards.
- The items are assigned to only one partition and the partition bounds are adjusted accordingly which results in overlapping partitions.

STARK uses the latter approach by assigning polygons to partitions based on their centroid point. Beside the partition bounds, we keep an additional *extent* information that is adjusted with the minimum and maximum values of the respective objects in each dimension. We decide which partition has to be checked during query execution based on this *extent* information and prune partitions that cannot contribute to the final result.

In its current version, STARK only considers the spatial component for partitioning. The partitioners implement Spark's `Partitioner` interface and can be used to spatially partition an RDD with the RDD's *partitionBy* method.

Grid Partitioner.

The first partitioner included in STARK is a fixed grid partitioner. Here, the data space is divided into a number of intervals per dimension resulting in a grid of rectangular cells (partitions) with equal dimensions. The bounds of these partitions are computed in a first step and afterwards with a single pass over the data, each item is assigned to a partition by calculating in which grid cell this item is contained.

Cost-Based Binary Space Partitioner.

As the fixed grid partitioner created partitions of equal size over the data space, it might create some partitions that contain the majority of the data items, while other partitions are empty. As an example consider the world map where events only occur on land, but not on sea. With a grid partitioning, there might be empty cells on sea and overfilled partitions in densely populated areas. To overcome this problem, we implemented a cost based binary space partitioning algorithm, based on [1]. This partitioner divides the space into two partitions with equal cost (number of contained items). If the cost for one partition exceeds a threshold, it is recursively divided again into two partitions of equal cost. This way, large regions with only a few items will belong to the same partition, while dense regions are split into multiple partitions. The recursion stops when a partition does not exceed the cost threshold or the algorithm reached a granularity threshold, i.e., a minimum side length of a partition.

2.2 Indexing

Just as in relational DBMS, indexing the content can significantly improve query performance. STARK uses the JTS[2] library for spatial operations. This library also provides an R-tree implementation (more accurately, an STR-tree) for indexing. STARK can use this index structure to index the content of a partition. A spatial partitioning is not mandatory to use index, but might bring additional performance benefits. Basically, STARK has three indexing modes, that can be chosen by the user:

No Indexing.

The partitions are not indexed and all items within a partition have to be evaluated with the respective predicate function.

Live Indexing.

When a partition is processed for evaluating a predicate, the content of that partition is first put into an R-tree and then, this index is queried using the query object. Since the results of the R-tree query are only candidates where the minimum bounding boxes match the query, these candidates have to be checked again if they really match the query object. During this candidate pruning step, the temporal predicate is evaluated as well, if needed. Live indexing can be used in a program by calling the *liveIndex* method on an RDD. This method takes the order of the tree as well as an optional partitioner as parameters, in case the RDD should be repartitioned before indexing.

Persistent Indexing.

Creating an index may be time consuming and often the same index will be reused in subsequent runs of the same or in another program. For such cases, STARK allows to

[2] http://tsusiatsoftware.net/jts/main.html

persist the index to disk/HDFS using Spark's method to save binary objects. An indexing that should be persisted can also be used by that same program. Thus, users don't need to do an extra run to just persist the index, but can already perform their operations. Such an index mode is done using the *index* method, which also takes the order of the tree as well as an optional partitioner as parameter.

2.3 DSL

One important design goal of STARK was to create an DSL that can be intuitively used by users within any (Scala) Spark program. This DSL provides all required operations for flexibly working with spatio-temporal data. This means that raw data loaded from HDFS or any other source can easily be processed by spatio-temporal operators and may be spatially partitioned and optionally indexed. The partitioning and indexing is transparent to the subsequent query operators which means they can be executed with or without spatial partitioning and indexing (and any combination thereof). Furthermore, the created indexes can be materialized, e.g., to HDFS, and be re-used within other programs. Figure 2 gives an overview of these possibilities.

In order to represent spatio-temporal data, STARK provides the `STObject` class. This class has only two fields: (1) `geo` that stores the spatial attribute and (2) and optional `time` field which holds the temporal information of an object. The `time` is optional to support spatial-only data that does not need any temporal information.

Beside these fields, the `STObject` class provides methods which check the relation to other spatio-temporal objects:

intersect(o) checks if the two instances (*this* and `o`) intersect in their spatial and/or temporal component,

contains(o) tests if *this* object completely contains `o` in their spatial and/or temporal component, and

containedBy(o) which is implemented as the reverse operation of *contains*

A formal definition for two objects o and p of type `STObject` and a predicate Φ can be given as:

$$\Phi(o,p) \Leftrightarrow \Phi_s(s(o),s(p)) \wedge (\quad (1)$$
$$(t(o) = \perp \wedge t(p) = \perp) \vee \quad (2)$$
$$(t(o) \neq \perp \wedge t(p) \neq \perp \wedge \Phi_t(t(o),t(p)))) \quad (3)$$

Where $s(x)$ denotes the spatial component of x, $t(x)$ the temporal component of x, Φ_s and Φ_t denote predicates that check spatial or temporal objects, respectively, and $\perp$ stands for `undefined` or `null`. This says that the predicate Φ is true for two spatio-temporal objects o and p, if the predicate on the spatial components of o and p is true (1), and both temporal components are *not* defined (2), or they are defined and the predicate on the temporal components of o and p is true as well (3).

To add the spatio-temporal operations to an RDD, STARK implements a special helper class called `SpatialRDDFunction` that has one plain Spark RDD as attribute and implements the supported spatio-temporal operations. In plain Spark, when an RDD contains 2-tuples of `(k,v)` an implicit conversion method creates a `PairRDDFunction` object, which provides, e.g., the *join* functionality using `k` as the join key. STARK follows the same approach: for an RDD of 2-tuples `(k,v)` we create a `SpatialRDDFunction` object implicitly, if `k` is of type `STObject`[3]. This implicit conversion is transparent to users and creates a seamless integration into any Spark program. Users don't have to explicitly create an instance of any of STARK's classes (except `STObject`) to use the spatio-temporal operators.

STARK has an *intersects*, *contains*, and *containedBy* predicate. In addition to that, we support a *withinDistance* operation, which finds all elements that are within a given maximum distance around the query object. Here, the distance function can be passed as a parameter so that users can implement their own function and adjust STARK to their requirements. However, we also include standard distance functions that can be used out of the box. Furthermore, there is a k nearest neighbor search operator.

An important data mining operation is clustering. STARK implements the DBSCAN algorithm for Spark inspired by MR-DBSCAN for MapReduce described in [1]. The implementation exploits the spatial partitioning: points that are within ϵ-distance from the partition border (where ϵ is the DBSCAN parameter), are replicated into the respective neighboring partition. In a next step a local partitioning is performed locally and in parallel on each partition. In a subsequent merge step, these local clusterings are merged using the replicated points, which may connect two clusters to a single one.

The following example shows the usage the spatio-temporal operator on an RDD with STARK. Consider an input file with a schema *(id: Int, category: String, time: Long, wkt: String)*. After pre-processing, we get an RDD of exactly that type: `RDD[(Int, String, Long, String)]`. We then create an `STObject` representing the location from the WKT string and time of occurrence from the `time` field of each entry:

```
val events = rawInput.map {
 case (id, ctgry, time, wkt) =>
           ( STObject(wkt, time), (id, ctgry) ) }
```

The `events` RDD of type `RDD[(STObject, (Int, String))]` and can now be used with any supported spatial-temporal predicate function:

```
val qry = STObject("POLYGON((...))", begin, end)
val contain = events.containedBy(qry)
val intersect = events.liveIndex(order = 5)
                      .intersect(qry)
```

We create a query object with a spatial polygon defined as a WKT string and a temporal interval. Here `begin` and `end` are `Long` values that describe the begin and end of a temporal time window for querying. With the *containedBy* function we can find all items in the events RDD that are contained by the query object. In the second example, the RDD is indexed using live indexing with an order of the R-tree of 5. We can then simply call the *intersects* (or any other supported function) on that indexed RDD.

3. EVALUATION

We evaluated our STARK implementation against other existing Hadoop- and Spark-based solutions for spatial data processing. In this evaluation we looked at provided features and further performed a micro benchmark. During this evaluation we found that not only do some systems have serious bugs and produce wrong results, but they are also not intuitively to use and have a very limited or even no API (only a

[3]i.e., `RDD[(STObject , V)`, where V can be any type.

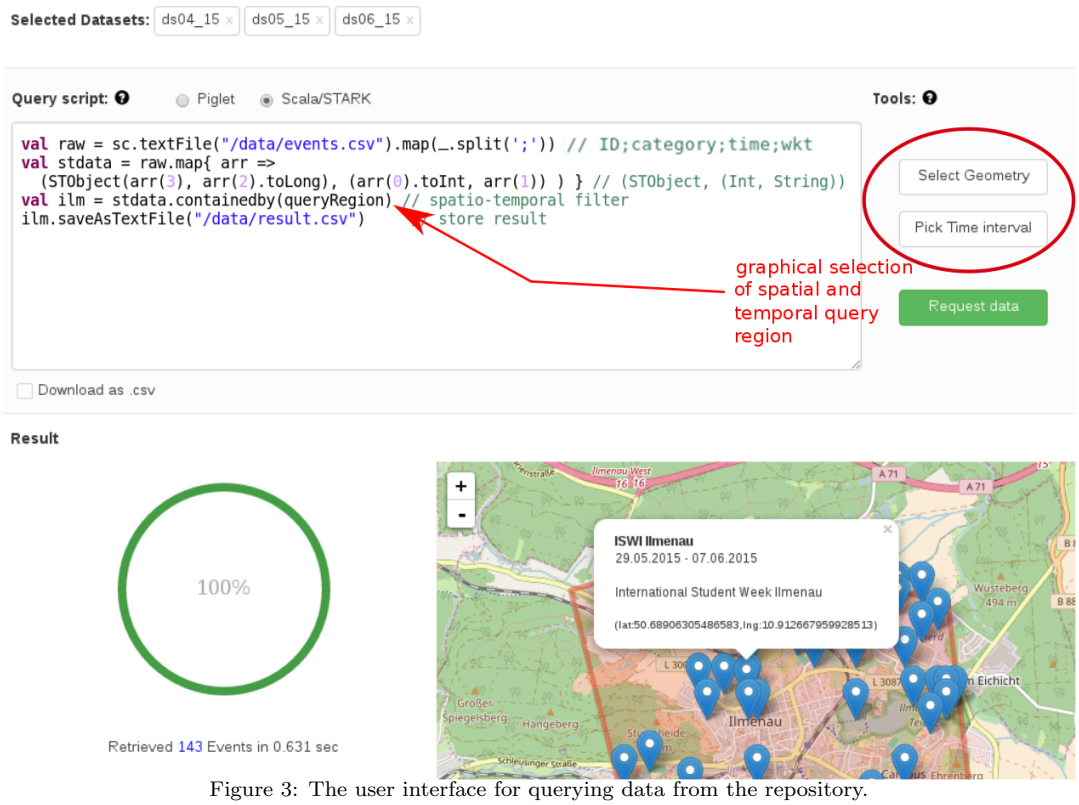

Figure 3: The user interface for querying data from the repository.

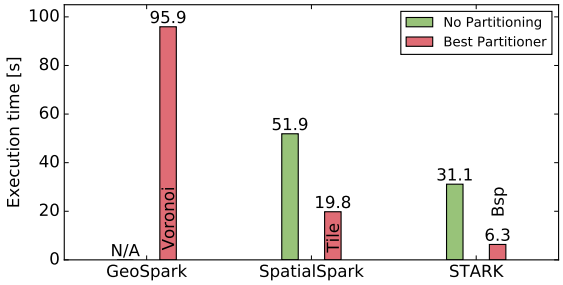

Figure 4: Execution times for self join operation for best partitioner and indexing.

command line interface). Figure 4 shows the result of a self join operation on a data set with 1,000,000 points comparing STARK with the Spark-based frameworks SpatialSpark [2] and GeoSpark [3]. The figure shows the execution time without partitioning as well as for the partitioner that resulted in the fasted execution time. For GeoSpark we experienced different result counts in each repetition of the experiment for two spatial partitioners. The results show that STARK outperforms the other frameworks in both cases. More results of the performance evaluation can be found in our GitHub repository[4].

4. DEMONSTRATION SCENARIOS

STARK is integrated into a larger project in which event information is extracted from text articles, stored as structured data, and analyzed using STARK's operators. We will prepare real world data sets with events from Wikipedia (created in the context of that project) as well as other spatio-temporal data sets with different contents. During the demonstration, visitors will be able to create and execute simple queries and complex data analysis pipelines or choose from prepared programs using our web front end. For that we will prepare different real world use case queries that include (reverse) geocoding, spatio-temporal join and aggregation, as well as clustering/co-location. Figure 3 shows the web front end with the query interface which supports the formulation of the spatio-temporal components by providing graphical selection tools using maps and date/time pickers that make the selected values available in the program. Queries and pipelines can be created as Scala programs, but we also allow to create these programs as Pig Latin scripts using our Piglet [4] engine that extends the original Pig Latin language with the before mentioned data types and operators. The results of the queries will be dynamically visualized in the web front end.

[4] https://github.com/dbis-ilm/spatialbm

Acknowledgments.

This work was partially funded by the German Research Foundation (DFG) under grant no. SA782/22.

5. REFERENCES

[1] Y. He, H. Tan *et al.*, "MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data," *FCS*, vol. 8, no. 1, pp. 83–99, 2014.

[2] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud." ICDEW, 2015.

[3] J. Yu, J. Wu, and M. Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data." SIGSPATIAL, 2015, p. 70.

[4] S. Hagedorn and K.-U. Sattler, "Piglet: Interactive and platform transparent analytics for rdf & dynamic data," in *WWW*, April 2016, pp. 187–190.