The STARK Framework for Spatio-Temporal Data Analytics on Spark

Stefan Hagedorn[1] Philipp Götze[1] Kai-Uwe Sattler[1]

Abstract: Big Data sets can contain all types of information: from server log files to tracking information of mobile users with their location at a point in time. Apache Spark has been widely accepted for Big Data analytics because of its very fast processing model. However, Spark has no native support for spatial or spatio-temporal data. Spatial filters or joins using, e.g., a *contains* predicate are not supported and would have to be implemented inefficiently by the users. Also, Spark cannot make use of, e.g., spatial distribution for optimal partitioning. Here we present our STARK framework that adds spatio-temporal support to Spark. It includes spatial partitioners, different modes for indexing, as well as filter, join, and clustering operators. In contrast to existing solutions, STARK integrates seamlessly into any (Scala) Spark program and provides more flexible and comprehensive operators. Furthermore, our experimental evaluation shows that our implementation outperforms existing solutions.

1 Introduction

Spark has been widely accepted as the data processing platform for big data sets providing better performance than classic Hadoop MapReduce as well as a more powerful expressiveness due to its large set of operators, support for cyclic data flow, and language-integrated queries. Spark supports a rather general data model, which allows to easily create programs for processing any type of data. Typically, data is loaded as text files and converted into their respective type of the chosen programming language or custom classes to represent complex data types.

One important class of complex data types are spatio-temporal data. For example, such data is created by sensors that record the location or movement of users or objects that periodically announce their current position. Another example for spatio-temporal data are *event data*, describing “something that happens at some place at some time”. Event data is not generated by sensors only, but can also be found in many textual sources like news articles, blogs, tweets, and social media. Here, so-called taggers detect, extract, and normalize spatial and temporal expressions from the sources and prepare the event data for further analysis steps.

A typical use case for analyzing spatio-temporal event data is to find correlated events in terms of their spatial and/or temporal components, i.e. if they are close to each other – or to a reference object – in time and space. Example applications are among others recommenders for events (e.g., in the cultural domain or for location-based services), mining crime data

[1] TU Ilmenau, Databases & information systems Group, Ilmenau, Germany, *first.last*@tu-ilmenau.de

(e.g., for predictive policing or homeland security) or political news (e.g., from the GDELT data set).

When event data is extracted in an automated process from large document repositories or the Web, event analysis is an interactive and exploratory process dealing with huge and often unknown data sets. Loading, indexing, and processing this data in a spatial or relational DBMS is often too time-consuming and requires a lot of preprocessing. Instead, tools like Apache Spark can directly process this data and can be used to implement complex analytical pipelines step-by-step. However, because Spark does not provide native support for spatial or temporal data, users have to define custom classes to represent spatial and/or temporal features and additionally implement special operators for spatio-temporal processing. Furthermore, efficient processing of large data sets requires to exploit data parallelism by partitioning the data accordingly, e.g., on its proximity, which is also not supported directly in Spark.

In order to address these challenges, a few extensions to Spark have been proposed recently. However, these solutions lack in an intuitive and integrated DSL, in support for spatio-temporal instead of only spatial data, as well as in a comprehensive and complete set of operators.

Our contribution is the following:

1. We present the STARK[2] framework for spatio-temporal data processing,
2. that provides an intuitive DSL and seamless integration with Apache Spark.
3. STARK includes an expressive set of operators, including filters, joins, and clustering,
4. and it supports spatial partitioning and indexing for efficient computations.

The remainder of the paper is organized as follows. After a discussion of related work in Sect. 2, we identify important requirements of spatio-temporal processing in Spark in Sect. 3. We then describe the architecture of the STARK framework in Sect. 4, followed by details for partitioning and indexing in Sect. 5. In Sect. 6 we show the internals of the spatial operators. Results of our experimental evaluation and comparison with GeoSpark and SpatialSpark are presented in Sect. 7. Finally, we conclude the main results of our work in Sect. 8 and point out to future work.

2 Related Work

Spatial data support has been implemented in presumably every type of data storage and processing system. Traditional relational database systems have built-in geometry data types and operations: Oracle's DBMS has data types for, among others, points, lines, and polygons, which can be defined using the `SDO_GEOMETRY` type. The system uses R-trees for indexing and supports *within distance*, *nearest neighbors* and other types of queries [Or14]. IBM DB2 contains a spatial extender which provides various spatial data types that all share a parent type called `ST_GEOMETRY`. It supports a Grid Index where the grid cells are indexed

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

using a B-tree [IBM13]. Microsoft SQL Server provides similar functionality and also uses a hierarchical grid index. The open source systems MySQL and PostgreSQL also support spatial data, where PostgreSQL uses the PostGIS extension, and support indexing using R-trees.

With the advance of Big Data, the MapReduce paradigm and its open source implementation in Hadoop became very popular and support for geospatial operators and indexing was needed on this platform as well. In [WP+14] Whitmann et al. present a framework to index spatial data for the Hadoop platform that uses quadtrees to support spatial queries. On each node, a partial tree is created which is then shuffled to other nodes and combined to a subtree of a complete index.

The first approach to implement spatial operations as an extension for Hadoop MapReduce is SpatialHadoop [EM13; EM15]. The framework provides spatial operators for range queries, k nearest neighbors, and joins. SpatialHadoop employs two index levels: on a global level an index partitions data across all nodes while a second index organizes data inside each partition. These indexes are used on read to eliminate records that do not contribute to the final result. As index structures, SpatialHadoop supports grid files, R-tree, and R+-trees.

Another approach that extends the plain Hadoop MapReduce framework with spatial operators is HadoopGIS [AW+13]. Similarly to SpatialHadoop, HadoopGIS utilizes a two level indexing: a global partition indexing and an optional local spatial indexing. The query processing engine, called RESQUE, uses these indexes to identify partitions to load and to speed up processing the required partitions. The RESQUE engine provides spatial operators like *intersects*, *contains*, *distance*, etc. The HadoopGIS system is integrated into Hive to provide a declarative SQL-like query language as user interface.

Accumulo[3] is a key-value store on top of Hadoop and is based on the design on Google's BigTable. In GeoMesa [F+13] the keys are created as a combination of the temporal value and the Geohash[4] representation of the spatial component. It is primarily designed for point data and non-point data has to be decomposed into multiple disjoint geohashes, resulting in duplicated entries in the index. It seems that data always has to have a spatial and a temporal component. When querying data, only those data items are considered, that intersect with the query region, based on the computed geohashes. GeoWave [Na] is a geospatial index that is also based on Accumulo or HBase. Like GeoMesa, it uses Space Filling Curves to represent multidimensional objects as 1-dimensional keys.

The in-memory execution model of Spark became very popular as it reduces the execution time dramatically, compared to MapReduce jobs. Currently, there are two systems that implement spatial operators for Spark: GeoSpark and SpatialSpark.

GeoSpark [YWS15; YWS16] is a Java implementation that comes with four different RDD types: `PointRDD`, `RectangleRDD`, `PolygonRDD`, and `CircleRDD`. These special RDDs internally maintain a plain Spark RDD that contains elements of the respective type, i.e. points, rectangles, polygons, and circles. GeoSpark supports k nearest neighbor queries,

[3] https://accumulo.apache.org
[4] https://en.wikipedia.org/wiki/Geohash

range queries, and join queries with *contains* or *intersects* predicates and each of these queries can be executed with or without using an index. The predicate *withinDistance* is only supported for joins. As described in [YWS15], GeoSpark supports R-trees and quadtrees to create an ad hoc index the RDDs. However, during the evaluation it showed that choosing quadtrees is not implemented. A persistent index does not seem to be possible since there is no index load functionality. Internally, GeoSpark uses the JTS library[5] which also provides the index structures. JTS does not support nearest neighbor queries on the indexes, though, and thus, GeoSpark uses their own extension to JTS, called JTSplus[6], to support these type of queries. GeoSpark comes with several partitioning techniques: R-Tree partitioning as well as Voronoi, Hilbert, and fixed grid partitioning. The main drawbacks of GeoSpark are that their spatial RDDs can only hold geometries of one certain type. On the one hand, this makes it impossible to load a data set that contains different geometry types in one column and on the other hand all other columns are removed when putting the data into these spatial RDDs. This also means that it is not possible to process the data in subsequent steps since related columns such as an ID are not available anymore. Furthermore, GeoSpark only has support for spatial data and temporal aspects cannot be modeled or processed. From a user perspective, GeoSpark provides an API which does not integrate tightly into Spark. RDDs are not created by transformations or actions, but by creating new objects and passing in values. Also operations like joins are not implemented as functions on these RDDs, but as extra classes. This way, the user has to adopt yet another API.

The goal of the SpatialSpark approach described in [YZG15] is to provide a parallel join technique for large spatial data sets with main focus on parallel hardware like multi-core CPUs and GPUs. To compute a join, the complete right relation is indexed using an R-tree which is then made available to all worker nodes using Spark's broadcast variables. After that, all items of the left relation are probed against that R-tree to find join partners. If the right relation does not fit into memory, SpatialSpark provides Fixed Grid Partitioning, Binary Space Partitioning, and Sort Tile Partitioning with and without using an R-tree as index [YZG]. As filter operation, SpatialSpark supports range queries with the predicates *contains*, *whithin* (*containedBy*), and *withinDistance*. The partitioning techniques from above, however, can not be used for filter operations. When querying a persistent index for these range queries the *intersects* predicate is compulsorily used. On top of that, k nearest neighbor queries are not possible. While SpatialSpark provides spatial operations for Spark, the core work of the authors is to integrate spatial operations into Impala. Thus, there is no real API and many things have to be done still by hand. Internally, they expect RDDs with an ID and a geometry object, which are processed when calling the specific query object (like `RangeQuery` or `BroadcastSpatialJoin`). Similar to GeoSpark, no other payload but the ID is allowed and, furthermore, the result of a join returns only the matched pairs IDs, which requires additional joins afterwards to retrieve the complete tuple in the application.

In Tab. 1 we outline the key differences from our STARK approach to the two frameworks for Spark GeoSpark and SpatialSpark. SpatialSpark and GeoSpark both provide some core functionality for dealing with spatial data like filter, joins and indexing. However, these frameworks both do not support spatio-temporal data and additionally, only GeoSpark

[5] http://tsusiatsoftware.net/jts/main.html
[6] https://github.com/jiayuasu/JTSplus

Tab. 1: Comparison of our STARK approach to GeoSpark and SpatialSpark

	GeoSpark	**SpatialSpark**	**STARK**
Language integrated DSL	x	x	✓
Support for Temporal Data	x	x	✓
Data Partitioning	✓	✓	✓
Indexing	✓	✓	✓
Persisted Indexes	x	✓	✓
Filter		no partitioning	
Contains	✓	(✓ - w/o Index)	✓
Intersects	✓	(✓ - w/ Index)	✓
WithinDistance	x	(✓ - w/o Index)	✓
Join	(✓ - pred. limitations)	(✓ - returns IDs)	✓
Nearest Neighbors	✓	x	✓
Clustering	x	x	✓
Skyline	x	x	(✓ - development)

supports nearest neighbors search as a more sophisticated data analysis operator. Both implementations do not provide a clustering or skyline implementation. Because of the respective design decisions and the resulting limitations of GeoSpark and SpatialSpark, we decided not to build our spatio-temporal engine on top of these platforms as this would have meant to rewrite a lot of code and rather built STARK from scratch. This way, we were able to fully integrate the DSL into Spark and build a flexible set of operators.

3 Requirements for Spatio Temporal Data Processing

Based on the application examples sketched in Sect. 1 and the current state of the art in spatio-temporal data processing we can identify several requirements for spatio-temporal Spark extensions:

native support for spatio-temporal data: As provided by spatial relational database systems and also defined in the SQL standard, spatial Spark should support appropriate data types in order to represent a set of standard geometries. Particularly, for the language-integrated APIs these data types should be mapped to native classes with similar APIs across the different language interfaces.

seamless integration into the Spark framework: In Spark, data processing is implemented as transformations of immutable distributed datasets (RDD, DataFrame) as well as actions returning data to the driver program. Together with the functional style of the Scala (or even Python and Java) API this composes a powerful and expressive way of formulating dataflow programs. Thus, a spatial/temporal extension should follow the same approach by supporting the RDD/DataFrame interface and defining spatial/temporal operations as transformations.

efficient and scalable processing of spatio-temporal data: Efficient processing of spatial data with non-trivial geometries requires appropriate spatial indexes such as the R-tree. The same holds for temporal data where temporal indexes such as interval trees are used. However, in Big Data analytics we cannot assume that data is always indexed in advance. Therefore, indexing should be optional as well as adaptive in the sense that indexes can be built on the fly before performing spatial/temporal operations and that such indexes are materialized for later reuse.

In order to exploit the benefits of a data-parallel platform like Spark, data partitioning is a crucial task. Particularly, spatial data requires space partitioning ensuring that closely located data objects are assigned to the same partition. Furthermore, the paritioning strategy should also handle data skew for a better load balancing, i.e. producing partitions with similar numbers of objects.

expressive spatial and temporal operations: Finally, a spatio-temporal framework should support a standard set of spatial and temporal predicates and query operators including filters, spatial and temporal joins, nearest neighbor search, and clustering. These operators should make use of indexes if available but also work on non-indexed data.

As described in Sect. 2, existing solutions do not fulfill these requirements completely yet. Thus, we have developed the STARK framework aiming at overcoming the deficiencies of current approaches.

4 The STARK Framework: Architecture & API

One of the main design goals of STARK is the seamless integration of spatio-temporal data and query operators into Apache Spark meaning that the user does not see any difference to the standard Spark API. For this purpose, we introduce new classes that add spatio-temporal operators to standard RDDs and encapsulate spatio-temporal data by a special class `STObject`. Fig. 1 gives an overview of the architecture of the STARK framework and its integration into the Spark ecosystem.

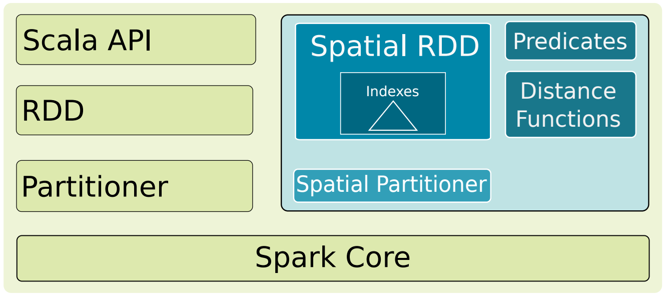

Fig. 1: Overview of STARK architecture and integration into Spark.

In the following, we describe the API of STARK comprising these classes as well as the RDD transformations for spatio-temporal operations. Furthermore, we present extensions to our dataflow compiler Piglet which extends the Pig Latin language by appropriate concepts.

4.1 DSL

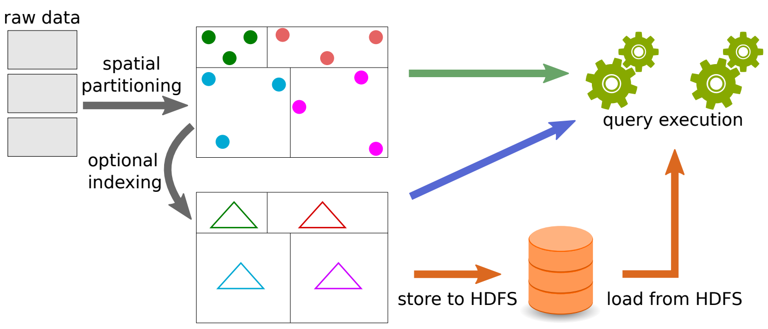

Fig. 2: Internal workflow for converting, partitioning, and querying spatio-temporal data

The goal of the STARK project is to create an easy to use DSL that can be used within any Spark program (written in Scala). This DSL should contain all necessary operations, as identified in Sect. 3 to comfortably work with spatio-temporal data. The possible workflow is shown in Fig. 2. An RDD is partitioned using a spatial partitioner and can optionally be indexed. Queries can be run on unindexed data as well as on indexed data. Indexes can be persisted and loaded again in other scripts.

As a basic data structure, STARK uses an `STObject` class. This class is used to represent the spatial and/or temporal component of any real world object. The class provides only two fields: (1) `geo` for storing the spatial component and (2) `time` to hold the temporal component of an object. To support spatial data without a temporal component, the `time` field may be left empty. Like many other Java based open source projects that deal with spatial data, STARK uses the JTS library with the JTSplus extension for internal representation of spatial objects and index structures.

The `STObject` class also provides various functions to test relations to other instances:

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*

The above functions all consider both, the spatial and the temporal component of the objects. Thus, a check of the three above mentioned is only true, iff:

1. the check yields true for the spatial component, and
2. both temporal components are *not* defined or
3. both temporal components are defined and they also return true for the respective check.

Or, expressed more formally: for two objects o and p of type `STObject` and a predicate Φ:

$$\begin{aligned}\Phi(o,p) \;\Leftrightarrow\;\; & \Phi_s(s(o), s(p)) \wedge (\\ & (t(o) = \perp \wedge t(p) = \perp) \vee \\ & (t(o) \neq \perp \wedge t(p) \neq \perp \wedge \Phi_t(t(o), t(p))))\end{aligned}$$

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`.

Spark's core component are resilient distributed datasets (RDDs). An RDD is a generic in-memory partitioned collection that can be distributed among the cluster nodes. To add support for spatio-temporal operations into Spark, STARK provides `SpatialRDDFunction` classes that wrap a traditional RDD and implement the spatial operators. This follows the concept that Spark implements for special operators on Pair-RDDs, like a join. A join can only be executed if the input RDDs are Pair-RDDs, i.e. they contain 2-tuples `(k,v)`, where `k` is used as join attribute. Spark automatically creates a `PairRDDFunction` object with the input RDD as parameter, using Scala's implicit conversions. The `PairRDDFunction` class then implements the join method.

STARK also provides such implicit conversions that create a `SpatialRDDFunction` object of a Pair-RDD, where the key `k` is of type `STObject` [7]. The value `v` of that pair can be of any type and is maintained during all operations. The `SpatialRDDFunction` class implements all spatial operations supported by STARK: filtering, join, kNN, clustering, as well as indexing.

The implicit conversion is transparent to the users and creates a seamless integration into any Spark program. Users don't have to explicitly create an instance of any of STARKs classes (except `STObject`) to use the spatial operators.

We now show how to transform a normal RDD loaded from a text file into a `SpatialRDD` and how to use it: Consider an example where we have a dataset given as a CSV file that contains a list of events from various categories. The schema of that file might be: `(id: Int, description: String, category: String, wkt: String, time: Long)`. After loading, preprocessing, and transforming, we get an RDD of exactly that type: `RDD[(Int, String, String, String, Long)]`. We then create an `STObject` representing the location and time of occurrence from the WKT string and time field, respectively, of each entry:

```
val events = rdd.map { case (id, desc, category, wkt, time) =>
                    ( STObject(wkt, time), (id, desc, category) ) }
```

The resulting RDD is of type `RDD[(STObject, (Int, String, String))]`. We can now simply use this RDD to call the functions to filter with a predicate.

[7] In the following we will refer to such an `RDD[(STObject,V)]` as `SpatialRDD` and exclude the implicit conversion.

```
val qryTime = 1481287522
val qry = STObject("POLYGON((....))", qryTime)  // create a query object
val contain = events.containedBy(qry)  // events contained by the query region
val intersect = events.intersect(qry)  // events that intersect with the query
```

Unlike in GeoSpark, in STARK a single RDD can hold objects of different geometries. This means that an `STObject` in `events` can represent a simple point, but also a polygon or any other geometry that can be represented as WKT.

In addition to the filtering operators shown above, STARK has also built-in support for spatial joins, k nearest neighbor search, and clustering as well as for computing skylines, which is currently under development and not released yet. These operators will be explained in Sect. 6.

4.2 Piglet Integration

In addition to the language-integrated DSL for the Scala programming language we also provide support for our dataflow compiler Piglet [HS16] that implements an extended Pig Latin dialect. Piglet generates code not only for Apache Spark but also for Flink and the streaming variants Spark Streaming and Flink Streaming. In addition to the backend specific operators it offers a lot of extensions to support, e.g., Linked Data, Basic Graph Patterns (BGP), matrix data, R scripts, and embedded code.

The goal of Piglet[8] is to simplify the development of data processing programs. Scripts can be compiled, packaged, and executed on the local machine or a YARN/Mesos cluster with a single command avoiding the tedious tasks of compiling and deploying all dependencies to a cluster.

In order to add support for spatial data into Piglet, we have introduced a new Pig data type called `geometry` and added new operators for filter, join, and indexing. The `geometry` data type is mapped to the Scala class `STObject` and instances can be constructed, e.g., from a WKT string.

Spatial data processing is supported by two operators:

- In order to filter a bag using some spatial predicate the `SPATIAL_FILTER` operator can be used.
- A spatial join is performed using the `SPATIAL_JOIN` operator. Same as for the `SPATIAL_FILTER` the join predicate can be defined by the user.

In the following example we assume two data sets: The first contains the names of states and their respective border as a WKT string. The second data set contains a list of events given with their ID and the respective latitude and longitude coordinates. To find the countries that each event occurred in, we can join these two data sets using the `contains` predicate:

[8] https://github.com/dbis-ilm/piglet

```
countriesRaw = LOAD 'countries.csv' as (name: chararray, poly: chararray);
countries = FOREACH countriesRaw GENERATE name, geometry(poly) as cntry;

eventsRaw = LOAD 'events.csv' AS (id: chararray, lat: double, lon: double);
events  = FOREACH eventsRaw GENERATE id, geometry("POINT("+lat+" "+lon+")") as loc;

eventCountries = SPATIAL_JOIN countries, events ON contains(cntry, loc);
DUMP eventCountries;
```

Internally, Piglet represents the script as a dataflow plan – the logical representation of the operator graph. The plan is passed to a rewriter which applies optimization and transformation rules and then sends the rewritten plan to the code generator which uses a template file to create the source code for the selected target platform. When generating the code for the `SPATIAL_JOIN` in the example above, the code generator has to produce code to ensure the correct input/output format, i.e., in this example conversion from `(chararray, geometry)` to `(geometry, (chararray, geometry))` which is required to for the implicit conversion to the `SpatialRDDFunctions` object of which we can call the spatial functions.

The Piglet integration also supports indexing (see Sect. 5) which can be done by the `INDEX` operator. The operator allows to specify the parameters needed to partition and index the data:

```
eventsIdx = INDEX events ON loc USING RTree(maxCost=100, cellSize=1.0, order=5);
```

Here, an R-tree index is created after applying a cost-based partitioning (see next section), where `maxCost` defines the maximum cost per partition, `cellSize` the side length of a cell used for the partitioning, and `order` is the order of the tree, i.e., the number of entries per node. Using the index is transparent in the Pig script: users can simply apply a filter or join operation on the `eventsIdx` bag.

Currently, the Pig Latin extension for spatial data is available only for the Spark target platform as it uses the STARK library. Though, we are working on a Flink port of the STARK project with which we are able to activate this extension for the Flink target, too.

5 Partitioning and Indexing in STARK

5.1 Partitioning

Spark leverages the data parallelism of the Hadoop environment and a program is executed in parallel on different nodes, where each node processes a (small) portion of the complete dataset, called a partition. Spark uses built-in partitioners, e.g., a hash partitioner, to assign each data item to a partition. However, while for text data it might be enough to create partitions of equal size, for spatial data one would also like to exploit the locality of the spatial feature of the data items. Consider for example a hash partitioner and a spatial grid partitioner: while both partitioners may create partitions of equal size, i.e. the same number of data items in each partition, the partitions created by the hash partitioner contain points from all over the data space disregarding their neighborhood, because the partition assignment is solely decided upon the hash value of the ID of the point, *h(x)*. Although we still can compute the result for, e.g., a spatial join operation, we cannot discard any partition apriori, because all partitions potentially contain join candidates. On the other hand, the spatial partitioner, in this case a simple spatial grid partitioner, divides the space using a grid and each resulting grid cell is a partition. If we now compute the minimum and maximum values for each dimension (e.g, latitude and longitude or x,y,z) we can easily decide if a partition contains potential join candidates or filter results, even without creating real indexes. We will talk about partition bounds for pruning at the end of this sub-section and describe how it is implemented in Sect. 6. Currently, STARK only uses the spatial component for partitioning (and indexing). However, in our current work we integrate the temporal component into both, partitioning and indexing.

Grid Partitioner. As briefly described before, the spatial grid partitioner creates partitions based on a grid over the data space. The partitioner accepts two parameters: the number of partitions in each dimensions (partitions per dimension, *ppd*) and the number of dimensions, where the latter has a default value of 2. To compute the cell size, the partitioner has to know the minimum and maximum values for each dimension. These values can be given as parameters or easily be computed by the partitioner in one single pass over the data. The side length in dimension *i* of a cell is determined by the minimum and maximum values for this dimension and the *ppd*:

$$length_i = \mid max_i - min_i \mid /ppd$$

To compute the partition a given point *p* belongs to, the partitioner has to simply calculate the partition ID $partitionId$. For a two dimensional scenario the formula would be:

$$x = \lfloor \lfloor \mid p_1 - min_1 \mid \rfloor \ / \ length_1 \rfloor$$

$$y = \lfloor \lfloor \mid p_2 - min_2 \mid \rfloor \ / \ length_2 \rfloor$$

$$partitionId = y * ppd + x$$

Binary Space Partitioner. The disadvantage of the naïve Grid Partitioner is that if the data points are skewed with only a few outliers throughout a large data space, the partitioning will result in lots of partitions were many of them are empty, some containing only very few data points, maybe only one or two, and only a small number of partitions with almost all data points. In an environment like Spark, this means that one executor has to perform all the work, i.e. process all data points, while other executors that were assigned an empty partition have no work to do. Thus, we additionally implemented the Binary Space Partitioner presented by He et al. in [H+14]. The advantage of this partitioning method is that one can define a maximum cost for a partition. First, the data space is divided into small quadratic cells of a given side length. Then, all possible partitioning candidates along the cell bounds in all dimensions are evaluated. After testing all possible partitioning candidates, the partitioning with smallest cost different between both candidate partitions is applied.

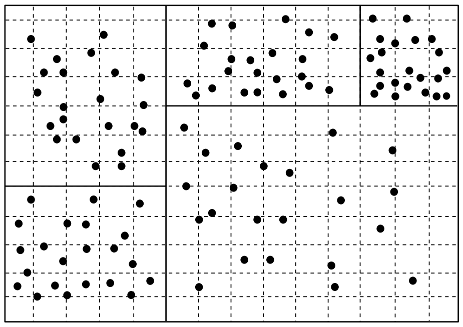
Fig. 3: A Binary Space Partitioning resulting in 5 partitions for a maximum cost of 22. Dashed lines represent cells and solid lines mark boundaries for generated partitions.

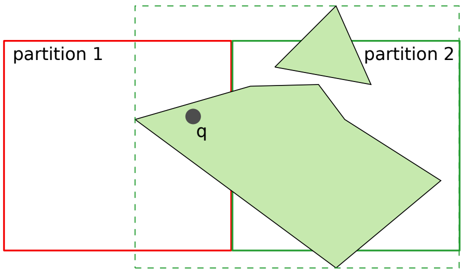

Fig. 4: Two neighbor partitions (solid lines) and the extent (dashed line) for partition 2 based on the contained geometries.

This results in two partitions for which the process is repeated recursively, if the according partition is longer than one cell length in at least one dimension and its cost is greater than the given maximum cost. While the authors of [H+14] introduce a sophisticated cost formula that also takes disk accesses and loading times into account, we simply consider the number of data points inside a partition as its cost. Applying this partitioning approach will result in partitions of almost equal cost, if the cell size is chosen reasonably according to the data. Fig. 3 shows a sample partitioning where the space is divided into 14×12 cells and results in 5 partitions with a maximum cost of 22.

Partitioning Polygons. The spatial partitioners decide to which partition a geometry belongs based on its position in space. Both introduced partitioners are based on grid cells of a fixed size, i.e. a width and height in two dimensional space. For points, the partitioners simply check which grid cell contains the current point and use this information to assign it to a partition. If the RDD contains not only points, but also polygons, these polygons may be larger than a grid cell. To decide to which partition a polygon belongs, the partitioners use the centroid point of that polygon.

While we can assign polygons to cells and therefore to partitions, this would create incorrect results when we use partitioning information to prune partitions before executing a join or filter operation: We might prune a partition based on its computed dimensions, the contained polygons, however, might span beyond the partition bounds and actually match the filter or join predicate. Fig. 4 shows an example of two neighbor partitions and a query point q, that lies within partition 1. If we wanted to find all geometries that contain q, we would prune partition 2 and falsely not find polygon p_1 (which is assigned to partition 2) as result. Therefore, in addition to the computed bounds, we also keep the *extent* of a cell/partition. This extent information is stored as a rectangle (in 2D space) and is created from the minimum and maximum x and y values of the cell/partition bounds and of all elements inside that cell/partition. For partition pruning, we then consider not the theoretical partition bounds, but the partition's extent. For the BSP, the extent of a partition is computed from all extents of the cells inside that partition.

5.2 Indexing

Repartitioning the data according to its spatial distribution can help to improve query performance. However, since all data items within a partition are candidates for the current spatial predicate of a query, nothing can be omitted and they all have to be evaluated. To further improve performance, STARK additionally can create an index for each partition. Theoretically, STARK can index a partition using any in-memory spatial index structure implementation. Currently, we support the R-tree index structure provided by the JTS library and plan to include more index structures in upcoming versions. In their Spark program, one can choose between the following three indexing modes:

No Index No index structure is used, and all data items are evaluated for query processing. This can be useful if the costs for creating and querying an index exceed the costs for processing all items (e.g., a full table scan). Note, in this case it does not matter how the RDD is partitioned.

Live Indexing During live indexing, the data is repartitioned using a given partitioner, if it was not partitioned before. Upon evaluation of a spatial or spatio-temporal predicate, all data items of a partition are first put into an index structure. Then this index is queried according to the spatial predicate of the query and, after pruning the candidates from the tree query, the overall result is returned.

Persistent Index For persistent indexing, the content of a partition is put into an index structure which is then used to evaluate the actual predicate. In contrast to live indexing, this execution mode changes the type of the input RDD from `RDD[(STObject , V)]` to `RDD[RTree[STObject , (STObject ,V)]]`. This means that the resulting RDD consists of R-tree objects instead of single tuples. This way, the index can be reused for subsequent operations. Furthermore, it can even be materialized to disk and loaded again for another execution. Thus, the index can be shared among different scripts to avoid the costly creation over and over again. Fig. 5 shows a data space which is partitioned into five partitions with equal costs. These partitions are indexed using the persistent index mode which results in an RDD with five entries.

As stated before, in our current work we investigate how to integrate the temporal component into indexing. One solution is to simply create two index structures, one for spatial and for temporal data, query both, and finally merge the results. However, in the literature there are dedicated approaches for spatio-temporal indexing ([MGA03; TVS96; ZSA99]), that we currently evaluate and will finally integrate into STARK.

6 Operator Implementation

Filter. The spatial filter operator for the different predicates is implemented as extra methods in the `SpatialRDDFunction` classes. An RDD can be filtered using the *contains*, *containedBy*, *intersects*, and *withinDistance* predicates. The first three mentioned predicates expect exactly one argument: the reference object (of type `STObject`) for which the respective predicate is to be evaluated. The operation is implemented as a Spark transformation and hence does not have any shuffling cost as it can be evaluated locally on a node's partition.

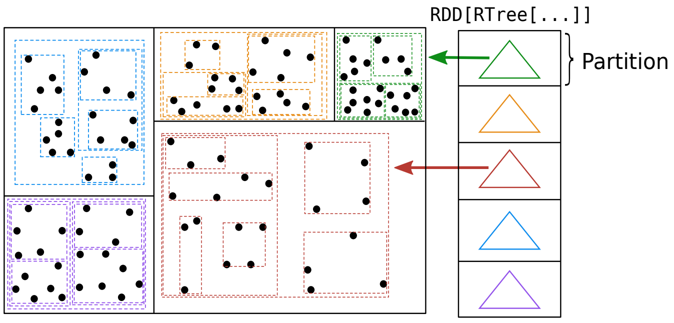

Fig. 5: A data space partitioned into five partitions which are indexed using an R-tree.

Algorithm 1 Filter operator and partition pruning - without indexing.

```
 1: procedure INTERSECTS($partition$, $qryObj$)
 2:     $checkPart \leftarrow true$
 3:     if rdd.partitioner isA SpatialPartitioner then          ▹ Pruning only for spatial partitioned RDDs
 4:         $sp \leftarrow rdd.partitioner$ as $SpatialPartitioner$
 5:         $extent \leftarrow sp.partitionExtent(partition)$
 6:         $checkPart \leftarrow extent.intersects(qryObj)$   ▹ Check only if qryObj intersects with partition
 7:     end if
 8:     if checkPart then                                         ▹ Partition can contain results
 9:         for each g in partition.iterator do
10:             if qry.intersects(g)) then                        ▹ Apply predicate
11:                 emit $g$
12:             end if
13:         end for
14:     end if
15: end procedure
```

The *withinDistance* predicate expects not only the reference object to which the distance should be computed, but also the distance function to use. The idea is to suppoert data of a cartesian as well as, e.g., a geodedic coordinate system which require different distance metrics for accurate computations.

The filter operators can prune partitions that cannot contain result tuples, if the RDD was partitioned using a spatial partitioner. Algorithm 1 shows the pseudocode of the *intersects* filter operator without indexing for a single partition. Spark executes the code for each partition in parallel without any additional programming effort. If the RDD was not partitioned using a spatial partitioner, we have to test each element in the partition with the predicate function. For the actual filtering, we iterate over all elements of that partition and apply the respective predicate function.

Nearest Neighbors. The nearest neighbor operator follows a straightforward implementation. Since the operators are executed in parallel without the nodes communicating with each other, an executor cannot decide alone, if its assigned partition contains elements that belong to the result. Thus, we compute the k nearest neighbors for each partition individually by computing the distance of each element to the query object, sort these elements by their respective distance to the query object in ascending order, and then return only the first k items. This results in n lists of k elements, if the RDD consists of n partitions. We then apply a global sort of these $n \times k$ elements and return only the first k items as the final result.

Algorithm 2 Join operator for contains predicate - with live indexing.

```
1: procedure GetPartitions($leftRDD$, $rightRDD$)
2:     for each l in leftRDD.partitions do        ▷ Check for spatial partitioner left out for brevity
3:         for each r in rightRDD.partitions do
4:             if l.intersects(r) then            ▷ Partitions must intersect to contain join results
5:                 emit new $JoinPartition(l, r)$
6:             end if
7:         end for
8:     end for
9: end procedure

10: procedure LiveIndexJoin($joinPartition$, $predicateFct$)
11:     $tree \leftarrow new\ RTree()$
12:     for each l in joinPartition.leftIterator do        ▷ Build index with all items of leftRDD
13:         $tree.insert(l)$
14:     end for
15:     for each r in joinPartition.rightIterator do
16:         $candidates \leftarrow tree.query(r)$        ▷ Query index with each entry in rightRDD
17:         for each c in candidates do        ▷ R-tree returns candidates only
18:             if predicateFct(c,r) then        ▷ Apply predicate
19:                 emit $(c, r)$        ▷ c is from leftRDD
20:             end if
21:         end for
22:     end for
23: end procedure
```

Join. A spatial join is a join operation using a spatial predicate like *contains* or *intersects*. In Spark however, only equi-joins are supported and θ-joins have to be implemented by the user. Hence, STARK provides its own implementation of join algorithms.

The join function accepts the other spatial RDD to join with as well as the join predicate. The predicate can be given as a function or as a identifier (an instance of a enumeration). Algorithm 2 shows the implementation of a join operator with partition pruning. To compute a join in Spark, we first have to produce the cartesian product of the partitions of both RDDs. When generating this cartesian product, we can check if the two partitions match the join predicate, e.g., intersect each other. If they do not match this predicate, this combination will not contain join partners and we can safely omit this combination. For combinations that can contain join partners, we create a new instance of a `JoinPartition` that contains references to these two partitions of the respective input RDDs.

For the actual computation of the join, the Spark framework will provide an executor with a partition which is now of type `JoinPartition`. For live indexing, we now iterate over the elements of the left RDD and insert the elements into an R-tree. Then, we query the R-tree with all elements of the right RDD and apply the predicate function to find join partners. Note, that the predicate may test the spatial or spatio-temporal components of the elements (same for the filter operator).

Clustering Another important operator for spatio-temporal data is clustering to identify groups of objects that occur close to each other in space and/or time. STARK comes with its own implementation of the density-based clustering algorithm DBSCAN for Spark. Our implementation is inspired by the MR-DBSCAN algorithm for MapReduce [H+14] and exploits Spark's data parallelism as well as makes use of the spatial partitioners.

The main idea of this algorithm is as follows: first, the data is partitioned in a way that all partitions are equally-sized for a better load balancing. Next, these partitions are extended in each direction by the value of the ϵ parameter of DBSCAN to overlap with their neighboring partitions. This partitioning step requires a shuffling of all data. Note, that some objects (which are contained in the overlap regions) are assigned to multiple partitions. Then, for each partition a local DBSCAN is performed in parallel to identify partition-local clusterings. Because this step produces different clusterings for each partition, an additional merge step is needed where objects from overlap regions assigned to multiple clusters are used to merge these clusters by constructing a graph of cluster pairs. In this graph, nodes represent local clusters and edges denote inter-partition relationships between clusters which can be merged. Based on this information, the objects are assigned to a single cluster in the final step. A crucial step in this algorithm is the partitioning: particularly, for skewed data a simple hash-based or even grid-based partitioning will result in an imbalance of numbers of objects per partition and, therefore, very different efforts for computing the local clustering. In order to avoid this problem, our DBSCAN implementation uses the spatial partitioner introduced in Sect. 5 to determine the initial partitions.

7 Evaluation

In this evaluation we will examine the performance of STARK and compare it to the competitors GeoSpark and SpatialSpark in their latest versions found on GitHub (GeoSpark: v0.3, SpatialSpark: v1.1.0). For our experiments, we used a cluster of 16 nodes where each node has an Intel Core i5 processor, 16GB RAM, and a 1TB disk. All experiments were executed in YARN mode with 16 executors and 2 cores per executor. On our cluster we run Ubuntu 14.04 with Hadoop 2.7, Spark 1.6.2 (because at the time of writing GeoSpark only supported this Spark version), Scala 2.11, and Java 1.8. Every test case was executed five times and the best result for each platform was chosen for comparison.

To create data sets, we had to comply with the requirements of the other two frameworks: GeoSpark can only work with data that contain only points or polygons, but not mixed data. SpatialSpark only supports RDDs with a Long value at the first position and the geometry at the second position. Other values are not allowed in addition to that. Since the other frameworks do not support spatio-temporal data, we only test spatial predicates.

We chose a sample of the `taxi` trip data[9] with trip information like pick up/drop off times and locations that represent our points-only data set and it contains 34 million entries. Additionally, we used a data set with only polygons that is created from an Open Street Map `world` dump. From this dump we exported the polygons of all borders, except of national borders, resulting in 322000 polygons. For our experiments with the join operator, we used a third data set called `blocks`[10] that contains around 38000 polygons representing all census

[9] http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml
[10] http://www1.nyc.gov/site/planning/data-maps/open-data.page

tracts and blocks of New York City. The *taxi* and `blocks` data sets were used in [YZG15], too. In addition to the results shown in this paper, we publish more results with other larger data sets on GitHub[11].

First, we conducted a test set for filter operations. In these experiments we created all combinations of supported partitioners with both live indexing and no index mode. Where applicable, all partitioners are executed with the same parameters. There are three filter operators that (1) find all polygons in `world` that *contain* a given query point, (2) find all polygons in `world` which *intersect* with a given query polygon, and (3) find all points in `taxi` that are *within a maximum distance* to a given point.

During the test construction some limitations and issues of the platforms came to light. In GeoSpark for example, one cannot combine spatial partitioning and indexing. Furthermore, the *contains* predicate was implemented the wrong way around and range filters with index only returned the candidates of an R-tree query without applying a predicate. We had to fix this to achieve comparability. For SpatialSpark, partitioning is not possible at all for range queries. On top of that, without index only *contains* and *withinDistance* and with persistent index only *intersects* is possible. The results of these filter operators for the best partitioner of each platform are shown in Fig. 6.

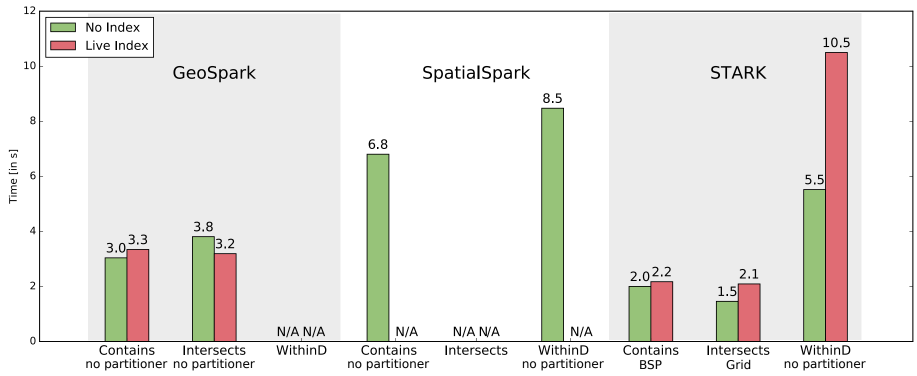

Fig. 6: Runtimes for filter operators with different indexing modes

Regarding the feature width, it can be seen that STARK offers the best use case coverage supporting all three operators both with and without using an index. STARK performs better for all executed filters than its competitors. Even for our relatively small test data set, the differences are clear and we expect them to be even greater for larger data sets. Especially for the *contains* and *intersects* queries partition pruning pays off. Due to internal implementation and design decisions, partition pruning cannot be applied with the *withinDistance* predicate and it will be addressed in future work. Different than expected the live index does not bring any benefit and is almost always slower in comparison to no index usage. Since the selectivity is quite high (there are less than 5 result tuples in each operation), the reason for this is probably the small size of the data and building the index requires more time than the faster lookup can improve the query. For larger data sets and larger partitions, the indexing will surely be much faster.

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

For filtering, GeoSpark performs best with no partitioning at all. SpatialSpark has no partitioning possibility for filter queries and, thus, the results for no partitioner is presented here. In the case of STARK, for each query another partitioning mode performs best. The reason that without spatial partitioning GeoSpark performs best is that the `taxi` data is highly clustered with only a few outliers. From the 34 million trip points, only a few hundred points are outside of Manhattan and some of them are very far away. Hence, millions of points are only a few centimeters away from each other. To achieve a good partitioning, a grid or BSP partitioner has to be configured with, e.g., a large number of very small cells to distribute those dense points to different partitions and hence create a balanced workload on the executors. With the highly clustered data, the partitioners tend to create a small number of partitions with a large number (millions) of points causing this executor to have all the work to do. Spark's hash partitioner, however, creates an even distribution, and thus, GeoSpark performs better without spatial partitioning. STARK however, was able to find a good spatial partitioning and therefore was able to apply partition pruning for additional speedup.

After examining the filter operators, we conducted a set of experiments for join operators. Since the `taxi` and `world` data sets contain a large number of entries each, we had to reduce the number of points and polygons in the respective input data sets to account for the limited hardware and to achieve reasonable execution times for repeatable experiments. We reduced the number of points by taking a sample of the original `taxi` data set, which contains 1328 points.

Also the implementation of the join tests revealed some limitations of the competitors. SpatialSpark for instance enforces the usage of an index when no partitioning is used. Outputting the result as ID tuple is also very limiting as already mentioned in Sect. 2. For GeoSpark partitioning is required for every join variant. Furthermore, when not using an index the predicate *contains* is automatically used while with indexing only the unfiltered candidates of the R-tree query are returned, which may be not the correct result. Although the *withinDistance* predicate is possible, it is limited to point geometries exclusively.

The join operations were also executed for all combinations of partitioners and indexing modes and the results for the join with *contains* predicate is shown in Fig. 7. We see that

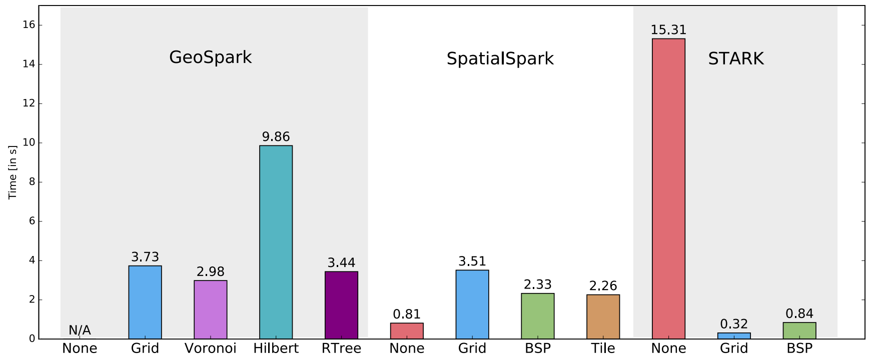

Fig. 7: Runtimes of the join operator with *contains* predicate for each partitioner using live index.

STARK outperforms GeoSpark and SpatialSpark if a spatial partitioner is used. Without spatial partitioning, we have to produce an expensive cartesian product and check all partition combinations for possible join partners. Without a spatial partitioner, SpatialSpark collects all entries of the right relation on one node and builds an R-tree. This tree is made available to all nodes using Spark's broadcast variable. This is obviously a good approach as long as data fits in memory. The runtime differences come from the time needed for partitioning and probably again from STARK's partition pruning ability.

Lastly, we ran an experiment with STARK's persistent indexing. Tab. 2 shows the runtimes for live and persistent indexes for a join. This join has to find all elements from `world` that contain points from an `event` data set which has 15000 events from all over the globe. This basically shows the overhead of creating the index. For persistent index, the index is already present to the operator which can use it for its operation. Also, a subsequent operator can make use of this index without recreating it.

Tab. 2: Comparison of live & persistent indexing

	Live Index	**Persistent Index**
BSP	12.6 sec	8.3 sec
Grid	20.6 sec	18.5 sec

8 Summary

In this paper we presented our STARK framework for analyzing large spatio-temporal data sets on Apache Spark. STARK integrates seamlessly into a Spark program by providing automatic conversion methods that, from a user perspective, add operators with spatial and spatio-temporal predicates to Spark's RDDs. STARK provides spatial partitioners and different indexing modes. A generated index can be stored persistently and loaded again by other scripts. Operators support these indexes, but also work on unindexed data. For further speedup, the operators can benefit from the spatial partitioning by not processing partitions that cannot contain any result tuples. In our experimental evaluation we compared STARK to GeoSpark and SpatialSpark and showed that it performs better and also provides a much more complete, flexible, and well integrated set of operators. In addition to the filter and join operators, STARK provides a DBSCAN clustering operator. In our ongoing work we extend the framework by more analysis operators like Skyline or Complex Event Patterns. For the skyline operator an angular partitioner, as described in [CHW12], is being implemented. Furthermore, we will integrate the temporal aspect more deeply into the framework and will use the temporal data for partitioning and indexing, too.

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

References

[AW+13] Aji, A.; Wang, F., et al.: Hadoop GIS: A High Performance Spatial Data Warehousing System over Mapreduce. VLDB Endow. 6/11, pp. 1009–1020, Aug. 2013.

[CHW12] Chen, L.; Hwang, K.; Wu, J.: MapReduce Skyline Query Processing with a New Angular Partitioning Approach. In. IPDPS, pp. 2262–2270, May 2012.

[EM13] Eldawy, A.; Mokbel, M. F.: A demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data. In. VLDB Endow., Aug. 2013.

[EM15] Eldawy, A.; Mokbel, M. F.: SpatialHadoop: A MapReduce Framework for Spatial Data. In: ICDE. Seoul, 2015.

[F+13] Fox, A.; Eichelberger, C., et al.: Spatio-temporal indexing in non-relational distributed databases. In: Big Data. 2013.

[H+14] He, Y.; Tan, H., et al.: MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data. FCS 8/1, pp. 83–99, 2014.

[HS16] Hagedorn, S.; Sattler, K.-U.: Piglet: Interactive and Platform Transparent Analytics for RDF & Dynamic Data. In. WWW 2016 Companion, 2016.

[IBM13] IBM: IBM DB2 10.5 Spatial Extender User's Guide and Reference, July 2013.

[MGA03] Mokbel, M. F.; Ghanem, T. M.; Aref, W. G.: Spatio-temporal access methods. IEEE Data Eng. Bull. 26/2, pp. 40–49, 2003.

[Na] National Geospatial-Intelligence Agency, N.: GeoWave, http://ngageoint.github.io/geowave, Accessed Dec. 13, 2016.

[Or14] Oracle: Oracle Database 12c: An Introduction to Oracle's Location Technologies, http://download.oracle.com/otndocs/products/spatial/pdf/12c/oraspatialandgraph_12c_wp_intro_to_location_technologies.pdf, Sept. 2014.

[TVS96] Theodoridis, Y.; Vazirgiannis, M.; Sellis, T.: Spatio-temporal indexing for large multimedia applications. In: ICMCS. Pp. 441–448, 1996.

[WP+14] Whitman, R. T.; Park, M. B., et al.: Spatial Indexing and Analytics on Hadoop. In. SIGSPATIAL, 2014.

[YWS15] Yu, J.; Wu, J.; Sarwat, M.: Geospark: A cluster computing framework for processing large-scale spatial data. In: International Conference on Advances in Geographic Information Systems. SIGSPATIAL, 2015.

[YWS16] Yu, J.; Wu, J.; Sarwat, M.: A demonstration of GeoSpark: A cluster computing framework for processing big spatial data./, pp. 1410–1413, 2016.

[YZG] You, S.; Zhang, J.; Gruenwald, L.: Large-scale spatial join query processing in cloud, https://github.com/syoummer/SpatialSpark, Accessed Sept. 13, 2016.

[YZG15] You, S.; Zhang, J.; Gruenwald, L.: Large-scale spatial join query processing in cloud. In. ICDEW, pp. 34–41, 2015.

[ZSA99] Zimbrão, G.; de Souza, J. M.; de Almeida, V. T.: The temporal R-tree, tech. rep., Technical Report ES492/99, COPPE/Federal University of Rio de Janeiro, Brazil, 1999.