The **STARK** Framework for Spatio-Temporal Data Analytics on Spark

**Stefan Hagedorn**  Philipp Götze  Kai-Uwe Sattler
TU Ilmenau

partially funded under grant no. SA782/22
Motivation

- data analytics for decision support
- include spatial and/or temporal information
  - sensor readings from environmental monitoring
  - satellite image data
  - event data
- data sets may be large
  or may be joined with other large data sets

Big Data platforms like Hadoop or Spark don't have *native* support for spatial (spatio-temporal) data
What do we want?

- event information (extracted from text)

- find correlations
- easy API/DSL
- fast Big Data platform (Spark/Flink)
- spatial & temporal aspects
- flexible operators & predicates
Outline

1. Existing Solutions
2. STARK DSL
   - Data representation
   - Integration
   - Operators
3. Make it fast
   - Partitioning
   - Indexing
4. Performance evaluation
Existing Solutions
... and why we didn't choose them

- **Hadoop-based**
  - slow
  - long Java programs
  - HadoopGIS, SpatialHadoop, GeoMesa, GeoWave

- **for Spark**
  - GeoSpark
    - special RDDs per geometry type (PointRDD, PolygonRDD)
    - no mix!
    - unflexible API
    - **crashes & wrong results!**
  - SpatialSpark
    - CLI programs
    - no (documented) API
Spark Scala API example

```scala
case class Event(id: String, lat: Double, lng: Double, time: Long)

val rdd: RDD[Event] = sc.textFile("/events.csv")
  .map(_.split(","))
  .map(arr => Event(arr(0), arr(1).toDouble, arr(2).toDouble, arr(3).toDouble))
  .filter(e => e.lat > 10 && e.lat < 50 && e.lng > 10 && e.lng < 50)
  .groupBy(_.time)
```

**Problem:** Spark does not know about spatial relations: no spatial join!

**Goal:**

- exploit spatial/temporal characteristics for speedup
- useful and flexible operators
  - predicates
  - distance functions
- integrate analysis operators as operations on RDDs
- seamless integration so that users don't see extra framework
STARK DSL
Data Representation

case class STObject(g: GeoType, t: Option[TemporalExpression])

- extra class to represent spatial and temporal component
  - time is optional
- defines relation-operators to other instances

```scala
def intersectsSptl(o: STObject) = g.intersects(o.g)

def intersectsTmprl(o: STObject) =
  (t.isEmpty && o.t.isEmpty ||
   (t.isDefined && o.t.isDefined && t.get.intersects(o.t.get)))

def intersects(t: STObject) = intersectsSpatial(t) && intersectsTemporal(t)
```

$$
\Phi(o, p) \iff \Phi_s(s(o), s(p)) \land (\\)
(t(o) = \bot \land t(p) = \bot) \lor
(t(o) \neq \bot \land t(p) \neq \bot \land \Phi_t(t(o), t(p)))
$$
STARK DSL
Integration

User program

```scala
val qry = STObject("POLYGON(...)", Interval(10,20))
val rdd: RDD[(STObject, (Int, String))] = ...

val filtered = rdd.containedBy(qry)
val selfjoin = filtered.join(filtered, Preds.INTERSECTS)
```

STARK

```scala
class STRDDFunctions [T](rdd: RDD[(STObject, T)]) {
  def containedBy(qry: STObject) = new SpatialRDD (rdd, qry, Preds.CONTAINEDBY )
}

implicit def toSTRDD[T](rdd: RDD[(STObject, T)]) = new STRDDFunctions (rdd)
```

- we do not modify the original Spark framework
- Pimp-My-Library pattern:
  - use implicit conversions
STARK DSL

Operations

- Predicates
  - `contains`
  - `containedBy`
  - `intersects`
  - `withinDistance`
- can be used for filters and joins
- with and without indexing

- clustering: DBSCAN
- k-Nearest-Neighbor search
- Skyline

- supported by spatial partitioning
Make it fast
Flow

raw data
spatial partitioning
optional indexing
store to HDFS
load from HDFS
query execution
Make it fast
Partitioning

- Spark uses Hash partitioner by default
  - does not respect spatial neighborhood

Fixed Grid Partitioning

- divide space into $n$ partitions per dimension
- may result in skewed work balance
Make it fast
Partitioning
Cost-based Binary Split

- divide space into cells of equal size
- partition space along cells
  - create partitions with (almost) equal number of elements
- repeat recursively if maximum cost is exceeded
SpatialFilterRDD(parent: RDD, qry: STObject, pred: Predicate)
extends RDD {

def getPartitions = {
  # Example partitions
}

def compute(p: ) = {
  for elem in p:
    if (pred(qry, elem))
      yield elem
}

Make it fast
Partition Pruning - Filter
SpatialJoinRDD(left: RDD, right: RDD, pred: Predicate) extends RDD {

    def getPartitions = {
        for l in left.partitions:
            for r in right.partitions:
                if l intersects r:
                    yield SptlPart(l, r)
    }

    def compute(p: (l, r)) = {
        for i in l:
            for j in r:
                if pred(i, j):
                    yield (i, j)
    }
}
Make it fast
Indexing

Live Indexing

- index is built on-the-fly
- query index & evaluate candidates
- index discarded after partition is completely processed

```python
def compute(p: Partition) {
    tree = new RTree()
    for elem in p:
        tree.insert(elem)

    candidates = tree.query()
    result = candidates.filter(predicate)
    return result
}
```
Make it fast

Indexing

Persistent Indexing

- transform to RDD containing trees
- can be materialized to disk
- no repartitioning / indexing needed when loaded again
Evaluation

Filter

10,000,000 polygons

 Execution time [s]

<table>
<thead>
<tr>
<th>Method</th>
<th>NONE</th>
<th>GRID</th>
<th>BSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Indexing</td>
<td>212.8</td>
<td>521.5</td>
<td>531.0</td>
</tr>
<tr>
<td>Live Indexing</td>
<td>284.4</td>
<td>270.0</td>
<td>332.9</td>
</tr>
<tr>
<td>Pers Indexing</td>
<td>218.7</td>
<td>9.0</td>
<td></td>
</tr>
<tr>
<td>Pre-parted Live Idx</td>
<td>6.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

16 Nodes, 16 GB RAM each, Spark 2.1.0
Evaluation
Filter - Varying range size

50,000,000 points

- No Partitioner, No Index
- No Partitioner, Live Index
- Grid, No Index
- Grid, Live Index
- Bsp, No Index
- Bsp, Live Index

Execution time [s]
Query range size
1x1, 5x5, 10x10, 50x50, 100x100, 360x180
Evaluation

Join
GeoSpark produced wrong results!

- 110,000 - 120,000 result elements missing
Conclusion

- framework for spatio-temporal data processing on Spark
- easy integration into any Spark program (Scala)

- filter, Join, clustering, kNN, Skyline
- spatial partitioning
- indexing

- partitioning / indexing not always useful / necessary
- performance improvement when data is frequently queried

https://github.com/dbis-ilm/stark
val rddRaw = ...
val partitioned = rddRaw.partitionBy( new SpatialGridPartitioner (rddRaw, ppD=5))
val rdd = partitioned.liveIndex(order= 10).intersects(STObject(...))

val rddRaw = ...
val partitioned = rddRaw.partitionBy( new BSPartitioner (rddRaw, cellSize= 0.5, cost = 1000*1000))
val rdd = partitioned.index(order= 10)
  rdd.saveAsObjectFile( "path"
val rdd1:RDD[RTree[STObject,(...)]] = sc.objectFile( "path"
Partition Extent

partition 1

partition 2

q
Clustering

```scala
val rdd: RDD[(STObject, (Int, String))] = ...
val clusters = rdd.cluster(minPts = 10,
                          epsilon = 2,
                          keyExtractor = { case (_, (id, _)) => id })
```

- relies on a spatial partitioning
- extend each partition by epsilon in each direction
  - to overlap with neighboring partitions
- local DBSCAN in each partition
- merge clusters
  - if objects in overlap region belong to multiple clusters => merge clusters