Simba: Efficient In-Memory Spatial Analytics.
Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou and Minyi Guo
SIGMOD’16.

Andres Calderon

November 10, 2016
There has been an explosion in the amount of spatial data in recent years...
There has been an explosion in the amount of spatial data in recent years...
There has been an explosion in the amount of spatial data in recent years...
There has been an explosion in the amount of spatial data in recent years...
The applications and commercial interest is clear...
But remember that “Spatial is Special”...
But remember that “Spatial is Special”...
But remember that “Spatial is Special”...
Is there room for improvements?

- Why do we need a new tool???
Yes, there is!!

Problems of Existing Systems...

- Single node database (low scalability)
  ArcGIS, PostGIS, Oracle Spatial.

- Disk-oriented cluster computation (low performance)
  Hadoop-GIS, SpatialHadoop, GeoMesa.

- No sophisticated query planner and optimizer
  SpatialSpark, GeoSpark

- No native support for spatial operators
  Spark SQL, MemSQL
Simba: **Spatial In Memory Big data Analytics.**

1. Extends Spark SQL to support spatial queries and offers simple APIs for both SQL and DataFrame.
2. Support two-layer spatial indexing over RDDs (low latency).
3. Designs a SQL context to run important spatial operations in parallel (high throughput).
4. Introduces spatial-aware and cost-based optimizations to select good spatial plans.
Simba: **Spatial In Memory Big data Analytics.**

1. Extends Spark SQL to support spatial queries and offers simple APIs for both SQL and DataFrame.
2. Support two-layer spatial indexing over RDDs (low latency).
3. Designs a SQL context to run important spatial operations in parallel (high throughput).
4. Introduces spatial-aware and cost-based optimizations to select good spatial plans.
Simba: **Spatial In Memory Big data Analytics.**

1. Extends Spark SQL to support spatial queries and offers simple APIs for both SQL and DataFrame.
2. Support two-layer spatial indexing over RDDs (low latency).
3. Designs a SQL context to run important spatial operations in parallel (high throughput).
4. Introduces spatial-aware and cost-based optimizations to select good spatial plans.
Simba: **Spatial In Memory Big data Analytics.**

1. Extends Spark SQL to support spatial queries and offers simple APIs for both SQL and DataFrame.
2. Support two-layer spatial indexing over RDDs (low latency).
3. Designs a SQL context to run important spatial operations in parallel (high throughput).
4. Introduces spatial-aware and cost-based optimizations to select good spatial plans.
Outline

1 Simba Architecture Overview
   - Programming Interface
   - Indexing
   - Spatial Operations
   - Optimization

2 Experiments
   - Comparison with Existing Systems
   - Comparison against Spark SQL
   - Join Methods vs Dimensionality

3 Conclusions
Outline

1 Simba Architecture Overview
   - Programming Interface
   - Indexing
   - Spatial Operations
   - Optimization

2 Experiments
   - Comparison with Existing Systems
   - Comparison against Spark SQL
   - Join Methods vs Dimensionality

3 Conclusions
Spark SQL is Apache Spark’s module for working with structured data.

- Seamlessly mixes SQL queries with Spark programs.
- Connects to any data source the same way.
- Includes a highly extensible cost-based optimizer (Catalyst).
- Spark SQL is a full-fledged query engine based on the underlying Spark core.
Spark SQL Overview

Spark SQL is Apache Spark’s module for working with structured data.

- Seamlessly mixes SQL queries with Spark programs.
- Connects to any data source the same way.
- Includes a highly extensible cost-based optimizer (*Catalyst*).
- Spark SQL is a full-fledged query engine based on the underlying Spark core.
# Apply functions to results of SQL queries.
context = HiveContext(sc)
results = context.sql(""
    SELECT
    *
    FROM
    people"")

names = results.map(lambda p: p.name)

# Query and join different data sources.
context.jsonFile("s3n://...").registerTempTable("json")
results = context.sql(""
    SELECT
    *
    FROM
    people
    JOIN
    json ...
    ")
Simba is an extension of Spark SQL across the system stack.

Figure 1: Simba architecture.
Simba is an extension of Spark SQL across the system stack.

**Figure 1: Simba architecture.**
Simba is an extension of Spark SQL across the system stack.

Simba supports a number of popular spatial indexing strategies, such as R-trees and hash maps, over different data sets to achieve better query performance. To the best of our knowledge, Simba leverages the spatial index support in Spark SQL to its kernel. In particular, Simba introduces new logical and physical operators inherited from Spark SQL, such as constant folding, predicate pushdown, and cost-based optimization, as well as new spatial-aware operators for operations over point and rectangular objects. These spatial operations allow Simba to optimize complex spatial queries, such as kNN joins, by exploiting partition tuning and query optimizations.

Figure 1: Simba architecture.
Simba is an extension of Spark SQL across the system stack.

Simba Architecture Overview

Figure 1: Simba architecture.
Simba is an extension of Spark SQL across the system stack.

**Figure 1: Simba architecture.**

- CLI
- JDBC
- Scala/Python Program
- Simba SQL Parser
- Extended DataFramed API
- Extended Query Optimizer
- Cache Manager
- Index Manager
- Physical Plan (with Spatial Operations)
- Table Caching
- Table Indexing
- Apache Spark
- RDBMS
- Hive
- HDFS
- Native RDD

**Indexing**
- Optimizations
- Spatial operations
- Programming interface
Outline

1. Simba Architecture Overview
   - Programming Interface
     - Indexing
     - Spatial Operations
     - Optimization

2. Experiments
   - Comparison with Existing Systems
   - Comparison against Spark SQL
   - Join Methods vs Dimensionality

3. Conclusions
• Support rich query types natively in the kernel...
  • The 5 nearest entries to point (2,3).

```sql
SELECT *
FROM points
SORT BY (x - 2) * (x - 2) + (y - 3) * (y - 3)
LIMIT 5
⇒
SELECT *
FROM points
WHERE POINT(x, y) IN KNN(POINT(2, 3), 5)
```
Spatial Predicates

- **RANGE, CIRCLERANGE and KNN...**

  - Show me the points inside a rectangle:

    ```sql
    SELECT *
    FROM points p
    WHERE POINT(p.x, p.y) IN RANGE(POINT(10, 5), POINT(15, 8)).
    ```

  - Show me the points laying 10m around:

    ```sql
    SELECT *
    FROM points p
    WHERE POINT(p.x, p.y) IN CIRCLERANGE(POINT(4, 5), 10)
    ```

  - Show me the 3 nearest points:

    ```sql
    SELECT *
    FROM points p
    WHERE POINT(p.x, p.y) IN KNN(POINT(4, 5), 3)
    ```
Spatial Joins

- **KNN JOIN and DISTANCE JOIN...**
  - List the 5 nearest hotels around Points of Interest.
    ```sql
    SELECT *
    FROM hotels AS h
    KNN JOIN pois AS p
    ON POINT(p.x, p.y) IN KNN(POINT(h.x, h.y), 5)
    ```
  - Show me drones that are close to each other (less than 20m).
    ```sql
    SELECT *
    FROM drones AS d1
    DISTANCE JOIN drones AS d2
    ON POINT(d2.x, d2.y, d2.z) IN CIRCLERANGE(POINT(d1.x, d1.y, d1.z), 20.0).
    ```
Index Management

- **CREATE INDEX and DROP INDEX...**
  - Create a 3D index on the sensor table using a R-tree:
    ```sql
    CREATE INDEX pointIndex ON sensor(x, y, z) USE RTREE
    DROP INDEX pointIndex ON sensor
    ```
  - **Generic use:**
    ```sql
    CREATE INDEX idx_name ON R(x_1, ..., x_m) USE idx_type
    DROP INDEX idx_name ON table_name
    ```
**Compound Queries**

- Fully compatible with standard SQL operators...
- Let’s count the number of restaurants around 200m of a POI (sort locations by the count):

```sql
SELECT
    p.id, count(*) AS n
FROM
    pois AS p
DISTANCE JOIN
    restaurants AS r
ON
    POINT(r.lat, r.lng) IN CIRCLERANGE(POINT(p.lat, p.lng), 200.0)
GROUP BY
    p.id
ORDER BY
    n
```
Same level of flexibility for DataFrames...

Let’s count the number of restaurants around 200m of a POI (sort locations by the count):

```scala
pois.distanceJoin(restaurants, Point(pois("lat"), pois("lng")), Point(restaurants("lat"), restaurants("lng")), 200.0)
  .groupBy(pois("id"))
  .agg(count("*").as("n"))
  .sort("n").show()
```
Simba Architecture Overview

1. Simba Architecture Overview
   - Programming Interface
   - Indexing
     - Spatial Operations
     - Optimization

2. Experiments
   - Comparison with Existing Systems
   - Comparison against Spark SQL
   - Join Methods vs Dimensionality

3. Conclusions
Table Indexing

- In Spark SQL:
  - Record → Row
  - Table → RDD[Row]
- Spark SQL makes a full scan of RDDs.
  - Inefficient for spatial queries!!!
- Solution: native **two-level** indexing over RDDs
Table Indexing

- **IndexRDD**
  - Pack all `Row` objects within a RDD partition into an array (O(1) cost for access).
  - **IPartition** data structure:
    - `case class IPartition[Type](Data: Array[Type], I: Index)`
    - Index can be HashMap, TreeMap or RTree.
  - So, by using `Type=Row`:
    - `type IndexRDD[Row] = RDD[IPartition[Row]]`
Two-level indexing strategy

Partition

Local Index

Global Index

IPartition[Row]

Array[Row]

IndexRDD[Row]

On Master Node

Global Index

Packing & Indexing

Row

i_0

i_1

i_2

i_3

i_4
Three-Phases Index Construction

- **Partition**
  - Concerns: Partition size, Data locality and Load balancing.
  - Partitioner abstract class.
  - STRPartitioner (based on Sort-Tile-Recursive algorithm).

- **Local Index**
  - RDD[Row] → IndexRDD[Row].
  - Collects statistics from each partition (number of records, partition boundaries, ...).

- **Global Index**
  - Enables to prune irrelevant partitions.
  - Can use different types of indexes and keep them in memory.
Outline

1. Simba Architecture Overview
   - Programming Interface
   - Indexing
   - Spatial Operations
   - Optimization

2. Experiments
   - Comparison with Existing Systems
   - Comparison against Spark SQL
   - Join Methods vs Dimensionality

3. Conclusions
Range Queries

- $\text{range}(Q, R)$
- Two steps: Global filtering + Local processing.

```
SELECT * FROM points p WHERE POINT(p.x, p.y) IN RANGE(POINT(5,5), POINT(10,8))
```
kNN Queries

- **kNN(q, R)**
- Good performance thanks to:
  - Local indexes.
  - Pruning bound that is sufficient to cover global kNN results.

(a) Loose Pruning Bound  (b) Refined Pruning Bound

```
SELECT * FROM points p WHERE POINT(p.x, p.y) IN KNN(POINT(5,8), 5)
```
Distance Join

- $R \bowtie_{\tau} S$
- DJSpark algorithm.

SELECT * FROM R DISTANCE JOIN S ON POINT(S.x, S.y) IN CIRCLERANGE(POINT(R.x, R.y), 5.0)
kNN Join

- \( R \bowtie_{kNN} S \)
- General methodology:
  1. Producing buckets: R and S are divided into \( n_1 \) (\( n_2 \)) equal-sized blocks. Every pair of blocks \((R_i, S_j)\) are shuffled to a bucket.
  2. Local kNN join: Performs \( kNN(r, S_j) \) for every \( r \in R \)
  3. Merge: Finds global \( kNN \) of every \( r \in R \) among its \( n_2k \) local \( kNNs \).
kNN Join

- $R \bowtie_{kNN} S$
- Explores several methods:
  - BKJSpark-N: Block nested loop $kNN$ join in Spark.
  - BKJSpark-R: Block R-tree $kNN$ join in Spark.
  - VKJSpark: Voronoi $kNN$ join in Spark.
  - ZKJSpark: z-value $kNN$ join in Spark.
  - RKJSpark: R-tree $kNN$ join in Spark.
Outline

1. **Simba Architecture Overview**
   - Programming Interface
   - Indexing
   - Spatial Operations
   - Optimization

2. **Experiments**
   - Comparison with Existing Systems
   - Comparison against Spark SQL
   - Join Methods vs Dimensionality

3. **Conclusions**
Why does it extend Catalyst?

1. The number of partition plays an important role in performance tuning.
2. Spatial indexes demands new logical optimization rules and spatial predicates management.
3. Indexing optimization cause more overheads than savings (Cost based optimization).
Partition estimation

- Cost model to estimate partition size:
  - Use of a sampling based approach to build estimators.

- Cost model + Partition strategy:
  1. Partitions are balanced.
  2. Each partition fits in memory.
  3. Number of partitions proportional to number of workers.
Index awareness optimizations

\((A \lor (D \land E)) \land ((B \land C) \lor (D \land E))\)

Full Table Scan

Optimize

Filter By:

\((A \land B \land C) \lor (D \land E)\)

Table Scan using Index Operators

With Predicate:

\((A \land C) \lor D\)

\((A \land B \land C) \lor (D \land E)\)

Transform to DNF

\((A \lor (D \land E)) \land ((B \land C) \lor (D \land E))\)

Result

DNF: Disjunctive Normal Form
- Geometric properties to merge spatial predicates.
  - i.e. $x > 3 \text{ AND } x < 5 \text{ AND } y > 1 \text{ AND } y < 6$ can be merged into a range query on $(\text{POINT}(3, 1), \text{POINT}(5, 6))$.
  - i.e. Two conjunctive range queries on $(\text{POINT}(3, 1), \text{POINT}(5, 6)) \text{ AND } (\text{POINT}(4, 0), \text{POINT}(9, 3))$ can be merged into a single range query on $(\text{POINT}(4, 1), \text{POINT}(5, 3))$. 

![Graph showing spatial predicates merging]
Selectivity + CBO

- Selectivity estimation + Cost-based Optimization.
  - Selectivity estimation over local indexes
  - Choose a proper plan: scan or use index.
- Broadcast join optimization: small table joins large table.
- Logical partitioning optimization for kNN joins.
  - Provides tighter pruning bounds.
Outline

1. Simba Architecture Overview
   - Programming Interface
   - Indexing
   - Spatial Operations
   - Optimization

2. Experiments
   - Comparison with Existing Systems
   - Comparison against Spark SQL
   - Join Methods vs Dimensionality

3. Conclusions
10 nodes cluster
- Processors: 6-core Intel Xeon E5 (1.6 to 2.0 GHz)
- RAM: 20 to 56 GB.
- Ubuntu 14.04 LTS, Hadoop 2.4.1, Spark 1.3.0
Datasets

- OSM (OpenStreetMap)
  - 2.2 Billion records, 132GB.
  - Five fields: ID, a two-dimensional coordinate and two text information.

- GDEL (Global Data on Events, Language and Tone)
  - 75 Million records
  - Seven attributes: timestamp, three two-dimensional coordinates (start, end and action of the event).

- RC (Synthetic dataset)
  - 1 Million to 1 Billion records, 2 to 6 dimensions.
  - Clusters randomly generated using Gaussian distributions.
Outline

1. Simba Architecture Overview
   - Programming Interface
   - Indexing
   - Spatial Operations
   - Optimization

2. Experiments
   - Comparison with Existing Systems
   - Comparison against Spark SQL
   - Join Methods vs Dimensionality

3. Conclusions
### Range and kNN Operations (OSM)

#### (a) Throughput

- **Simba**
- **GeoSpark**
- **SpatialSpark**
- **SpatialHadoop**
- **Hadoop GIS**
- **Geomesa**
- **DBMS X**

#### (b) Latency

- **Simba**
- **GeoSpark**
- **SpatialSpark**
- **SpatialHadoop**
- **Hadoop GIS**
- **Geomesa**
- **DBMS X**

The graphs illustrate the performance of range and kNN operations on different systems. Simba is shown to be efficient in terms of throughput and latency compared to other systems, indicating its suitability for handling spatial data efficiently.
Join Operations (OSM)

Simba, GeoSpark, SpatialSpark, SpatialHadoop, Hadoop GIS, Geomesa, DBMS X, and VKJHadoop.

Running time (s)

DistanceJoin: Simba 100, GeoSpark 10, SpatialSpark 1000, SpatialHadoop 100, Hadoop GIS 10000, Geomesa 100000, DBMS X 1000000, VKJHadoop 10000000

kNNJoin: Simba 100, GeoSpark 10, SpatialSpark 1000, SpatialHadoop 100, Hadoop GIS 10000, Geomesa 100000, DBMS X 1000000, VKJHadoop 10000000

Overall, Simba's index storage overhead is acceptable, for example, Simba provides 51x lower latency and 45x less index storage than Hadoop based systems because GeoSpark has not utilized global indexing (which uses R-tree for both local indexes and the global index).
Simba GeoSpark does not scale well for expensive join operations and large datasets (as evident from its index construction cost).

Overall, Simba’s index storage overhead is acceptable, for example, for datasets (as large as) tens of thousands) even for very large data.

In particular, it only consumes 653KB for the largest dataset (4.4 billion KB). In particular, the global index size in Simba is very small, 4-5x). Global indexing is only supported by Simba, SpatialSpark, and Geomesa. Most of the systems have the lowest storage overhead in their local indexes, while those systems that support global indexes (Simba, SpatialSpark, and Geomesa) have the higher storage overhead.

Next, we investigated the storage overhead (memory footprint or disk space used) of indexes in different systems, using OSM data of Simba GeoSpark. SpatialSpark, 12x faster than Hadoop GIS, and 15x faster than Geomesa.

SpatialHadoop is nearly as good as Simba. DBMS X is a single-node parallel database, only supports spatial operations. Thus, its analytical performance is not as good as Simba. DBMS X has outperformed Simba on spatial joins over geometric objects, we also compared Simba against the V oronoi-based join algorithm, which is 2.5x faster than SpatialHadoop, 3x faster than GeoSpark, and 3.5x faster than DBMS X and 7x faster than VKJHadoop.

The following bar charts indicates that the corresponding operation with the default settings described in Section 8.1 over the OSM data set (500 million records). A red cross mark in the plot indicates that the corresponding operation could not be completed.

The raw key size is about 40GB and the data itself is 132GB.
Outline

1. Simba Architecture Overview
   - Programming Interface
   - Indexing
   - Spatial Operations
   - Optimization

2. Experiments
   - Comparison with Existing Systems
   - Comparison against Spark SQL
   - Join Methods vs Dimensionality

3. Conclusions
Range Query Performance (GDELT)

(a) Throughput
(b) Latency
kNN Query Performance (GDELT)

(a) Throughput

(b) Latency

Figure 16: Effect of data size.

Figure 17: Comparison against Spark SQL
Outline

1 Simba Architecture Overview
   - Programming Interface
   - Indexing
   - Spatial Operations
   - Optimization

2 Experiments
   - Comparison with Existing Systems
   - Comparison against Spark SQL
   - Join Methods vs Dimensionality

3 Conclusions
Join Operations Performance (RC)

(a) Distance join

(b) $k$NN join
Outline

1. Simba Architecture Overview
   - Programming Interface
   - Indexing
   - Spatial Operations
   - Optimization

2. Experiments
   - Comparison with Existing Systems
   - Comparison against Spark SQL
   - Join Methods vs Dimensionality

3. Conclusions
Conclusions

- Simba: A distributed in-memory spatial analytics engine.
- Indexing support for efficient query processing.
- Spatial operator implementation tailored towards Spark.
- Spatial and index-aware optimizations.
- User-friendly SQL and DataFrame API.
- No changes to Spark kernel, easier migration to higher version Spark.
- Superior performance compared against other systems.
Simba: A distributed in-memory spatial analytics engine.

- Indexing support for efficient query processing.
- Spatial operator implementation tailored towards Spark.
- Spatial and index-aware optimizations.
- User-friendly SQL and DataFrame API.
- No changes to Spark kernel, easier migration to higher version Spark.
- Superior performance compared against other systems.
Simba: A distributed in-memory spatial analytics engine.
- Indexing support for efficient query processing.
- Spatial operator implementation tailored towards Spark.
- Spatial and index-aware optimizations.
- User-friendly SQL and DataFrame API.
- No changes to Spark kernel, easier migration to higher version Spark.
- Superior performance compared against other systems.
Simba: A distributed in-memory spatial analytics engine.
Indexing support for efficient query processing.
Spatial operator implementation tailored towards Spark.
Spatial and index-aware optimizations.
User-friendly SQL and DataFrame API.
No changes to Spark kernel, easier migration to higher version Spark.
Superior performance compared against other systems.
Simba: A distributed in-memory spatial analytics engine.
Indexing support for efficient query processing.
Spatial operator implementation tailored towards Spark.
Spatial and index-aware optimizations.
User-friendly SQL and DataFrame API.
No changes to Spark kernel, easier migration to higher version Spark.
Superior performance compared against other systems.
Simba: A distributed in-memory spatial analytics engine.
Indexing support for efficient query processing.
Spatial operator implementation tailored towards Spark.
Spatial and index-aware optimizations.
User-friendly SQL and DataFrame API.
No changes to Spark kernel, easier migration to higher version Spark.
Superior performance compared against other systems.
Simba: A distributed in-memory spatial analytics engine.
- Indexing support for efficient query processing.
- Spatial operator implementation tailored towards Spark.
- Spatial and index-aware optimizations.
- User-friendly SQL and DataFrame API.
- No changes to Spark kernel, easier migration to higher version Spark.
- Superior performance compared against other systems.
Future ideas

- Native support to geometry objects and operations.
- Spatial joins over predicates (i.e. intersect or touch).
- Explore complex spatio-temporal patterns.
Future ideas

- Native support to geometry objects and operations.
- Spatial joins over predicates (i.e. intersect or touch).
- Explore complex spatio-temporal patterns.
Future ideas

- Native support to geometry objects and operations.
- Spatial joins over predicates (i.e. intersect or touch).
- Explore complex spatio-temporal patterns.
Future ideas

- Native support to geometry objects and operations.
- Spatial joins over predicates (i.e. intersect or touch).
- Explore complex spatio-temporal patterns.
Thank you!!!

Do you have any question?
### Comparison with existing systems

<table>
<thead>
<tr>
<th>Core Features</th>
<th>Simba</th>
<th>GeoSpark</th>
<th>SpatialSpark</th>
<th>SpatialHadoop</th>
<th>Hadoop GIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data dimensions</td>
<td>multiple</td>
<td>$d \leq 2$</td>
<td>$d \leq 2$</td>
<td>$d \leq 2$</td>
<td>$d \leq 2$</td>
</tr>
<tr>
<td>SQL</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>Pigeon</td>
<td>×</td>
</tr>
<tr>
<td>DataFrame API</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Spatial indexing</td>
<td>R-tree</td>
<td>R-/quad-tree</td>
<td>grid/kd-tree</td>
<td>grid/R-tree</td>
<td>SATO</td>
</tr>
<tr>
<td>In-memory</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Query planner</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Query optimizer</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Concurrent query execution</td>
<td>thread pool in query engine</td>
<td>user-level process</td>
<td>user-level process</td>
<td>user-level process</td>
<td>user-level process</td>
</tr>
</tbody>
</table>

#### query operation support

<table>
<thead>
<tr>
<th></th>
<th>Simba</th>
<th>GeoSpark</th>
<th>SpatialSpark</th>
<th>SpatialHadoop</th>
<th>Hadoop GIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Box range query</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Circle range query</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>$k$ nearest neighbor</td>
<td>✓</td>
<td>✓</td>
<td>only 1NN</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Distance join</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>via spatial join</td>
<td>✓</td>
</tr>
<tr>
<td>$k$NN join</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Geometric object</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Compound query</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>
Index Construction Time (OSM)

- **Index construction time (s)**
  - **Data size (× 10^6)**
  - **Index construction time (s)**

**Legend:**
- Simba
- GeoSpark
- SpatialSpark
- SpatialHadoop
- Hadoop GIS
- Geomesa
- DBMS X

**Notes:**
- Simba, GeoSpark, and SpatialSpark show the best index construction time.
- Figure 9(a) shows the effect of data size on the OSM data set (500 million records).
- A red cross mark indicates the corresponding operation is unsupported.
- Simba is more efficient than GeoSpark and SpatialSpark.
- Simba, SpatialHadoop, and Hadoop GIS have higher throughput than SpatialSpark.
- Simba is 3x faster than GeoSpark.
- Simba is 7x faster than VKJHadoop.
- Simba is 10x faster than Hadoop GIS.

**Additional Information:**
- The raw key size is about 40GB and the data itself is 132GB.
- Simba and GeoSpark show the best index construction time, and SpatialSpark was not included in Figure 9(a) since local indexes are not supported.
- Geomesa utilizes a hash based indexing strategy, which is efficient for finding kNNs. But its range query is small.
- Simba's local index support is very efficient for finding kNN queries.
- Simba is more efficient than GeoSpark and SpatialSpark.
- Simba is 5-100x better than other key-value storage systems.
Range Query Performance (OSM)

(a) Effect of data size.

(b) Effect of query area size (percentage of total area).
kNN Query Performance (OSM)

(a) Effect of data size.

(b) Effect of $k$.

Figure 12(a) shows that Simba outperforms Spark SQL by about 10× for this is the result size becomes so large that there are less opportunities for Simba's query optimizer. Spark SQL for this case; roughly 1% of the space).

$k$NN join. It is impossible to express $k$NN join in Spark SQL in a

Spark SQL crashes after 10 hours when data size increases from 30 million to 4.4 billion records (2× OSM). The performance of both Simba and Spark SQL drops when the data size increases while that of Simba drops much slower, which implies Simba has much better scalability. This is due to Simba is also much more user-friendly with the native support for query planning, query optimization, and concurrency by other systems (except that SpatialHadoop supports a SQL-like engine based on Spark but without native support for spatial operations, to see the benefit of extending Spark SQL to Simba for spatial query processing through thread pooling. Whereas, GeoSpark always more efficient than the baseline method BDJSpark-R, un-

For Simba, we compared Simba's DJSpark algorithm with a nested range queries. In contrast, Simba's performance is almost not affected by the data size, because Simba is able to narrow the input $k$NN query. In contrast, Simba's performance is almost not affected by the data size, because Simba is able to narrow the input area to the expensive Cartesian product it has to perform to the change on $k$ each $\theta$ and running a distance join as a $k$NN join. Figure 13 shows the results for distance join using Simba's DJSpark algorithm with a nested $\tau$-join in Spark SQL (an example...
Distance Join Performance (OSM)

![Distance Join Performance (OSM)](image)

(a) Effect of data size.

(b) Effect of \( \tau \).

Simba SQL crashes after 10 hours.

Spark SQL crashes after 10 hours.

BDJSpark-R, DJSpark.
kNN Join Performance (OSM)

(a) Effect of data size.

(b) Effect of $k$. 

Simba