Simba: Towards Building Interactive Big Data Analytics Systems

Feifei Li

THE UNIVERSITY OF UTAH®
Complex Operators over Rich Data Types Integrated into System Kernel

• For Example:

```
SELECT k-means from Population
WHERE k=5 and feature=age and income >50,000
Group By city
```

What are the impacts to query evaluation and query optimization modules?
e.g. Big Spatial Data is Ubiquitous!

Location-based Services

IoT Projects & Sensor Networks

Social Media

- Uber
- Dianping.com
- Facebook
- Twitter
- Yelp
- Sensor
- Routing Node

CONNECT THE WORLD
Problems of Existing Systems

- **Single node shared-state MPP database -> low scalability**
  - ArchGIS, PostGIS, Oracle Spatial

- **Disk-oriented cluster computation -> low performance**
  - Hadoop-GIS, SpatialHadoop, GeoMesa

- **No native support for spatial operators**
  - Spark SQL, MemSQL

- **No sophisticated query planner & optimizer**
  - SpatialSpark, GeoSpark
100 TB on 1000 machines

½ - 1 Hour \rightarrow 1 - 5 Minutes

Hard Disks \rightarrow Memory

In memory computation over a cluster
Apache Spark

“Fast and General engine for large-scale data processing.”

- **Speed**: By exploiting in-memory computing and other optimizations, Spark can be 100x faster than Hadoop for large scale data processing.

- **Ease of Use**: Spark has easy-to-use language integrated APIs for operating on large datasets.

- **A Unified Engine**: Spark comes packed with higher-level libraries, including support for SQL queries, streaming data, machine learning and graph processing.
Resilient Distributed Datasets (RDDs)

- Immutable, partitioned collections of objects
- Created through parallel transformations (map, filter, groupBy, join...) on data in stable storage: support pipeline optimization and lazy evaluation
- Can be cached in memory for efficient reuse.
- Retain the attractive properties of MapReduce:
  - Fault tolerance, data locality, scalability...
- Maintain lineage information that can be used to reconstruct lost partitions.
- Ex:
  ```scala
  messages = textFile(...).filter(_.startsWith("ERROR"))
            .map(_.split('\t')(2))
  ```

![Diagram of RDD processing]

- **HDFS File** → **Filtered RDD** → **Mapped RDD**
  - `filter (func = _.startsWith("ERROR"))`
  - `map (func = _.split(...))`
Spark scheduler

DAG based scheduler

Pipeline functions within a stage

Cache-aware work reuse & locality

Partitioning-aware to avoid shuffles

Stage 1
- A: (groupBy)
- B: (map)
- C: (union)

Stage 2
- D: (join)
- E: (cached data partition)

Stage 3
- F: (cached data partition)
- G: (cached data partition)

= cached data partition
Spark components
Spatial and Multimedia Data

```
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)
```
Simba: **Spatial In-Memory Big data Analytics**

*Simba is an extension of Spark SQL across the system stack!*

1. **Programming Interface**
2. **Table Indexing**
3. **Efficient Spatial Operators**
4. **New Query Optimizations**

![Diagram of Simba's Architecture](image)
## 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>Query Operation</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>In progress</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>
Query Workload in Simba

**Life of a query in Simba**

- **SQL Query**
- **DataFrame API**
- **Simba Parser**
- **Logical Plan**
- **Optimized Logical Plan**
- **Physical Plans**
- **Selected Physical Plan**
- **RDDs**

- **Analysis**
- **Logical Optimization**
- **Physical Planning**
- **Cost-Based Optimization**

- **Catalog**
- **Index Manager**
- **Cache Manager**

Statistics flow between Physical Plans and Selected Physical Plan.
Programming Interfaces

- Extends both SQL Parser and DataFrame API of Spark SQL
- **Support rich query types natively in the kernel**

```sql
SELECT *
FROM points
SORT BY (x - 2)*(x - 2) + (y - 3)*(y - 3)
LIMIT 5
```

- Achieve something that is impossible in Spark SQL.

```sql
SELECT *
FROM points
WHERE POINT(x, y) IN KNN(POINT(2, 3), 5)
```

```sql
SELECT *
FROM queries q KNN JOIN pois p
    ON POINT(p.x, p.y) IN KNN(POINT(q.x, q.y), 3)
```
Programming Interfaces (cont’d)

• Fully compatible with standard SQL operators.

```sql
SELECT poi.id, count(*) as c
FROM poi DISTANCE JOIN data
  ON POINT(data.lat, data.long)
    IN CIRCLERANGE(POINT(poi.lat, poi.long), 3.0)
WHERE POINT(data.lat, data.long)
    IN RANGE(POINT(24.39, 66.88), POINT(49.38, 124.84))
GROUP BY poi.id
ORDER BY poi.id
```

• Same level of flexibility for DataFrame

```python
poi.distanceJoin(data, Point(poi("lat"), poi("long")),
  Point(data("lat"), data("long")), 3.0)
  .range(Point(data("lat"), data("long")),
    Point(24.39, 66.88), Point(49.38, 124.84))
  .groupBy(poi("id"))
  .agg(count("*").as("c")).sort(poi("id")) .show()
```
Table Indexing

• All Spark SQL operations are based on RDD scanning.
• Inefficient for selective spatial queries!

• In Spark SQL:
  • **Record** -> **Row**
  • **Table** -> **RDD[Row]**
• Solution in Simba: *native two-level indexing over RDDs*

• Challenges:
  • RDD is not designed for *random access*
  • Achieve this *without hurting Spark kernel and RDD abstraction*
Table Indexing (cont’d)

Two-level Indexing Framework: *local* + *global* indexing

```
CREATE INDEX idx_name ON R(x_1, ..., x_m) USE idx_type
DROP INDEX idx_name ON table_name
```
Table Indexing (cont’d)

**Representation for Indexed Tables (RDDs) in Simba**

case class IPartition[Type](data: Array[Type], index: Index)

type IndexRDD[Type] = RDD[IPartition[Type]]

Indexed tables are still RDDs (hence, fault tolerance is taken care of)!
Efficient execution of rich operations

• Indexing support -> efficient algorithms

• Global Index: **partition pruning**

• Local Index: **parallel pruning within selected partitions**

![Diagram]

global Index

partition pruning on the master node

local indexes

parallel pruning on selected partitions
Spatial operations

• Range Query: \( range(Q, R) \)
• Two steps: global Filtering + local processing
Spatial Operations (cont’d)

- $k$ nearest neighbor query: $\text{knn}(q, R)$
- Key to achieve good performance:
  - Local indexes
  - Pruning bound that is sufficient to cover global $k$NN results.
More Sophisticated Operations

• Distance Join: $R \bowtie_\tau S$

• Our solution: the \textbf{DJSpark} Algorithm

• $k$NN join: $R \bowtie_{kNN} S$

• Our solution: the \textbf{RKJSpark} Algorithm

• Details in the paper...
Query Optimizer

- **Index** and **geometry**-awareness optimizations

- Index scan optimization: for better index utilization

- **Selectivity estimation + Cost-based Optimization**
  - Selectivity estimation over local indexes
  - Choose a proper plan: scan or use index.

- Spatial predicates merging
Query Optimizer

- Index scan optimization: for better index utilization

\[
\text{Filter By:} \quad (A \lor (D \land E)) \land ((B \land C) \lor (D \land E))
\]

\[
\text{Full Table Scan}
\]

\[
\text{Result}
\]

\[
\text{Filter By:} \quad (A \land B \land C) \lor (D \land E)
\]

\[
\text{Optimize}
\]

\[
\text{Table Scan using Index Operators With Predicate:} \quad (A \land C) \lor D
\]

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

\[
\text{Optimize}
\]

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

\[
\text{Result}
\]

\[
\text{Filter By:} \quad (A \land B \land C) \lor (D \land E)
\]

\[
\text{Transform to DNF}
\]

\[
(A \lor (D \land E)) \land ((B \land C) \lor (D \land E))
\]
Query Optimizer (cont’d)

• Partition Size Auto-Tuning
  • Data Locality
  • Load Balancing

  A good Partitioner (e.g., STR Partitioner)

• Memory fitness \(<\) record-size estimator

• Broadcast join optimization: small table joins large table

• Logical partitioning optimization for RKJSparlk
  • provides tighter pruning bounds \(\gamma_i\)
Comparison with Existing Systems

Environment: A 10-node cluster with 54 cores and 135GB RAM

Query over 500M OpenStreetMap entries
Comparison with Existing Systems (cont’d)

Join operations performance

Join between two 3M-entry tables
Performance against Spark SQL: Data Size

$k$NN Query **Throughput**

$k$NN Query **Latency**
Performance of Joins: Data Size

**Distance Join** Performance

**kNN Join** Performance
Support for multi-dimension

$kNN$ **Throughput against Dimension**

$kNN$ **Latency against Dimension**
Index Building Cost: Time

Index Building Time **against** Data size

Index Building Time **against** Dimension
Index Building Cost: Space

Local Index Size

Global Index Size
100 TB on 1000 machines

½ - 1 Hour  1 - 5 Minutes  1 second

Hard Disks → Memory → ?

Online Query Execution
Complex Analytical Queries (TPC-H)

```
SELECT SUM(l_extendedprice * (1 - l_discount))
FROM customer, lineitem, orders, nation, region
WHERE c_custkey = o_custkey
  AND l_orderkey = o_orderkey
  AND l_returnflag = 'R'
  AND c_nationkey = n_nationkey
  AND n_regionkey = r_regionkey
  AND r_name = 'ASIA'
```

This query finds the total revenue loss due to returned orders in a given region.
Online Aggregation [Haas, Hellerstein, Wang SIGMOD’97]

\[
\begin{align*}
\text{SELECT} & \quad \text{ONLINE SUM} (l_{\text{extendedprice}} \cdot (1 - l_{\text{discount}})) \\
\text{FROM} & \quad \text{customer, lineitem, orders, nation, region} \\
\text{WHERE} & \quad c_{\text{custkey}} = o_{\text{custkey}} \\
& \quad l_{\text{orderkey}} = o_{\text{orderkey}} \\
& \quad l_{\text{returnflag}} = 'R' \\
& \quad c_{\text{nationkey}} = n_{\text{nationkey}} \\
& \quad n_{\text{regionkey}} = r_{\text{regionkey}} \\
& \quad r_{\text{name}} = 'ASIA'
\end{align*}
\]

\[
\Pr[\tilde{Y} - \varepsilon < Y < \tilde{Y} + \varepsilon] > 0.95
\]

Confidence Interval

Confidence Level
Accuracy vs. Speed Tradeoff

Continuous Query Evaluation and Feedbacks to the user

Execute on Samples

Execute on Entire Dataset

Error

Execution Time

5 sec

30 mins
Ongoing Works

• Native support to general geometric objects
  • Polygons, Segments, etc.
  • Geometric object filtering.
  • Spatial join over predicates such as intersect and touch
Ongoing works (cont’d)

- Online sampling and aggregation support.
- Provides uniform random samples / approximate aggregation results in an online fashion.

```sql
SELECT ONLINE 
SUM(l_extendedprice * (1 - l_discount)), COUNT(*) 
FROM customer, orders, lineitem 
WHERE c_mktsegment = 'BUILDING' 
AND c_custkey = o_custkey 
AND l_orderkey = o_orderkey 
AND l_quantity <= 1000 
WITHTIME 30000 CONFIDENCE 99 REPORTINTERVAL 500;
```
Ongoing works (cont’d)

• Trajectory (Spatial-Temporal) Data Analysis
  • Massive trajectory retrieval
  • Trajectory Similarity Search
Ongoing works (cont’d)

• Easy to use user interface integration
Conclusion

• Simba: A distributed in-memory spatial analytics engine

• User-friendly SQL & DataFrame API
• Indexing support for efficient query processing
• Spatial + multimedia + ML operator implementation tailored towards Spark
• Spatial and index-aware optimizations
• No changes to Spark kernel -> easier migration to higher version Spark
• Superior performance compared against other systems
• Support online query execution

• Now open sourced at: https://github.com/InitialDLab/Simba/
• Under active development....
https://github.com/InitialDLab/Simba/

Questions?
Supported SQL & DataFrame API

- **Point wrapper**
  
  \[
  \text{SQL: } \text{POINT}(\text{pois.x} + 2, \text{pois.y} \times 3) \\
  \text{DataFrame: } \text{Point}(\text{pois("x"): pois.y} \times 3)
  \]

- **Box range query**
  
  \[
  \text{SQL: } \text{p IN RANGE}(\text{low}, \text{high}) \\
  \text{DataFrame: } \text{range}(\text{base: Point, low: Point, high: Point})
  \]

- **Circle range query**
  
  \[
  \text{SQL: } \text{p IN CIRCLE_RANGE}(\text{c}, \text{rd}) \\
  \text{DataFrame: } \text{circleRange}(\text{base: Point, c: Point, rd: Double})
  \]

- **k nearest neighbor query**
  
  \[
  \text{SQL: } \text{p IN KNNSPHERE}(\text{q}, \text{k}) \\
  \text{DataFrame: } \text{knn}(\text{base: Point, k: Int})
  \]
Supported SQL & DataFrame API (cont’d)

• **Distance join**

  SQL: $R \text{ DISTANCE JOIN } S \text{ ON } s \text{ IN CIRCLERANGE}(r, \tau)$
  
  DataFrame: `distanceJoin(target: DataFrame, left_key: Point, right_key: Point, \tau: Double)`

• **$k$NN join**

  SQL: $R \text{ KNN JOIN } S \text{ ON } s \text{ IN KNN}(r, k)$
  
  DataFrame: `knnJoin(target: DataFrame, left_key: Point, right_key: Point, k: Int)`

• **Index management**

  SQL: `CREATE INDEX idx_name ON R(x_1, \ldots, x_m) USE idx_type`
  
  DROP INDEX `idx_name` ON `table_name`
  
  SHOW INDEX ON `R`
  
  DataFrame: `index(idx_type: IndexType, idx_name: String, attr: Seq[Attribute])`
  
  `dropIndex()`
  
  `showIndex()`
Spatial Operations: Distance Join

• Distance Join: $R \bowtie_{\tau} S$

• General theta-join in Spark SQL -> *Cartesian product!!!*

• Our solution: the *DJSpark* Algorithm
Spatial Operations: $k$NN join

- $k$NN join: $R \bowtie_{kNN} S$

- Solutions in Simba:
  - Block Nested Loop $k$NN join ($BKJSpark-N$)
  - Block Nested Loop $k$NN join with local R-Trees ($BKJSpark-R$)
  - Voronoi $k$NN join* ($VKJSpark$)
  - $z$-value $k$NN join+ ($ZKJSpark$) -> approximate $k$NN join
  - R-Tree $k$NN join ($RKJSpark$)

Spatial Operations: $k$NN join (cont’d)

- R-Tree $k$NN join (RKJSpark)
- Distributed hash-join like algorithm.
- **For each partition $R_i$, find $S_i \subset S$, s.t. $\forall r \in R_i$, $\text{knn}(r, S) = \text{knn}(r, S_i)$**

- $R \bowtie_{kNN} S = \bigcup R_i \bowtie_{kNN} S_i$

- Define $c_{r_i}$ as the centroid of partition $R_i$

- Take a uniform random sample $S' \subset S$, and suppose $\text{knn}(c_{r_i}, S') = \{s_1, \ldots, s_k\}$

- For each partition $R_i$:
  \[ u_i = \max_{r \in R_i} |r, c_{r_i}| \]
  \[ \gamma_i = 2u_i + |c_{r_i}, s_k| \]
  \[ S_i = \{s | s \in S, |c_{r_i}, s| \leq \gamma_i\} \]
Support for multi-dimension joins

**Distance Join against Dimension**

**kNN Join against Dimension**
Experiments

• OpenstreetMap Data, up to 2xOpenstreetMap.
  1xOpenstreetMap: 2.2 billion records in 132GB

• 10 nodes with two configurations: (1) 8 machines with a 6-core Intel Xeon E5-2603 v3 1.60GHz processor and 20GB RAM; (2) 2 machines with a 6-core Intel Xeon E5-2620 2.00GHz processor and 56GB RAM.

• Other datasets are used in high dimensions
Data Cube

• Jim Gray, Adam Bosworth, Andrew Layman, Hamid Pirahesh: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. ICDE 1996
Future Works

• Native support to general geometric objects
  • Polygons, Segments, etc.
  • Spatial join over predicates such as *intersect* and *touch*

• Data in very high dimensions (> 10d)

• More sophisticated cost-based optimizations.