SociaLite: Query Language for Big Data Analysis

Jiwon Seo, Jongsoo Park*, Jaeho Shin, Stephen Guo, and Monica S. Lam

Stanford MobiSocial Group * Intel Research Lab

I. Motivation

- Many big data platforms
- Still big data analysis is hard!
  - Low-level primitives
  - Too static (hard to make changes)
  - Complex configuration
  - Not efficient
- We want big data analysis as easy as Python (or Ruby)

II. Introducing SociaLite

SociaLite is a query language that is

- As easy as Python
- As efficient as optimized C++/Java
- Parallel execution
- High-level primitives for parallelization
- Python integration

High-level queries compiled to fast parallel/distributed code!

III. Syntax and Semantics of SociaLite

Tables

Local Tables

```
Foo(int i, String s).
```

This declares a table named *Foo* with two columns; each machine in the cluster has its own instance of the table.

Distributed Tables

```
Bar[int i](String s, double d).
```

With the partitioning operator [ ], this declares a distributed table *Bar*; it is horizontally partitioned (or sharded) and stored across machines in the cluster.

Table Options

- Nesting

```
Bar[int i](String s, (double d)).
```

- Ranges

```
Bar[int i:0..100](String s, double d).
```

- Indexing and sorting

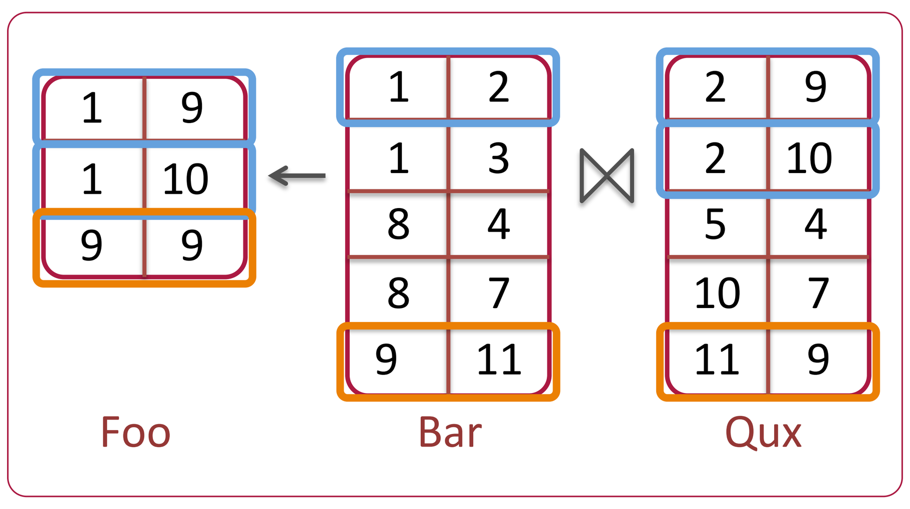
```
Bar[int i](String s, double d) indexby i.
Bar[int i](String s, double d) sortby d.
```

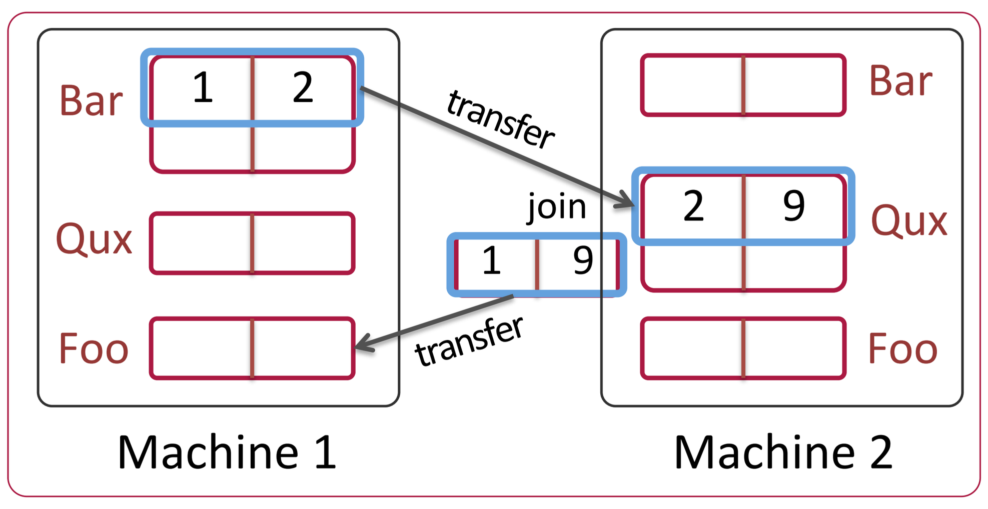
Rules

```
Foo[a](c) :- Bar[a](b), Qux[b](c).
```

Rule head — Rule body

Tables in the rule body are joined, and the result tuples are inserted to the table in the rule head.

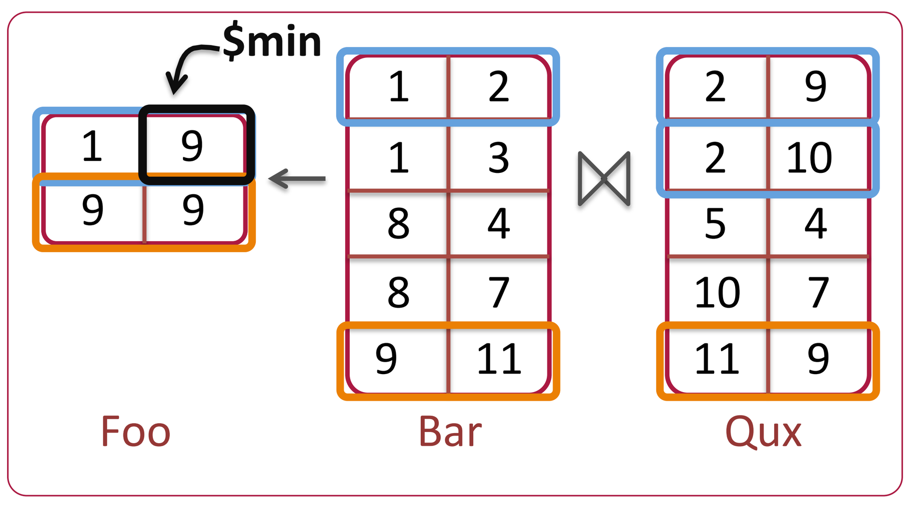




Aggregation

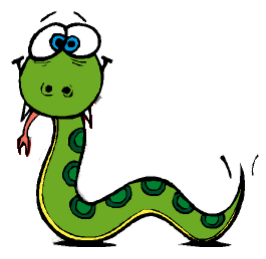
```
Foo[a]($min(c)) :- Bar[a](b), Qux[b](c).
```

The $min in the rule head is an aggregate function that is applied to tuples in Foo having the same first column value.



We have built-in aggregate functions such as $min, $max, $sum, $avg, and $argmin; also user-defined aggregate functions are supported.

Python Integration



SociaLite queries in Python code are marked with a pair of backticks, as shown in following.

```
print "This is Python code."
`Foo(int i, double d).
 Foo(a,b) :- a=10, b=42.0 .`
```

Tables can be accessed from Python code as following

```
for a,b in `Foo(a,b)`:
  print a, b
```

Access Python in SociaLite

Python variables are referenced in SociaLite queries with a preceding dollar sign.

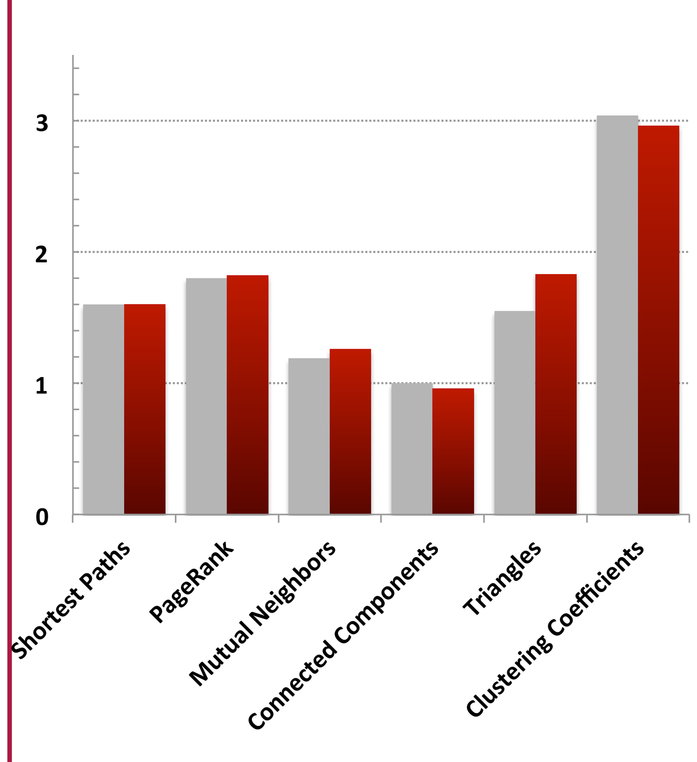
```
var=42
`Foo(int i, double d).
 Foo(a,b) :- a=$var, b=42.0 .`
```

Python functions are referenced similarly, but requires type annotation.

```
@returns(int)   ← return type annotation
def func(a,b):
  return a+b
`Foo(a,b) :- a=$func(0,1), b=$func(1,1).`
```

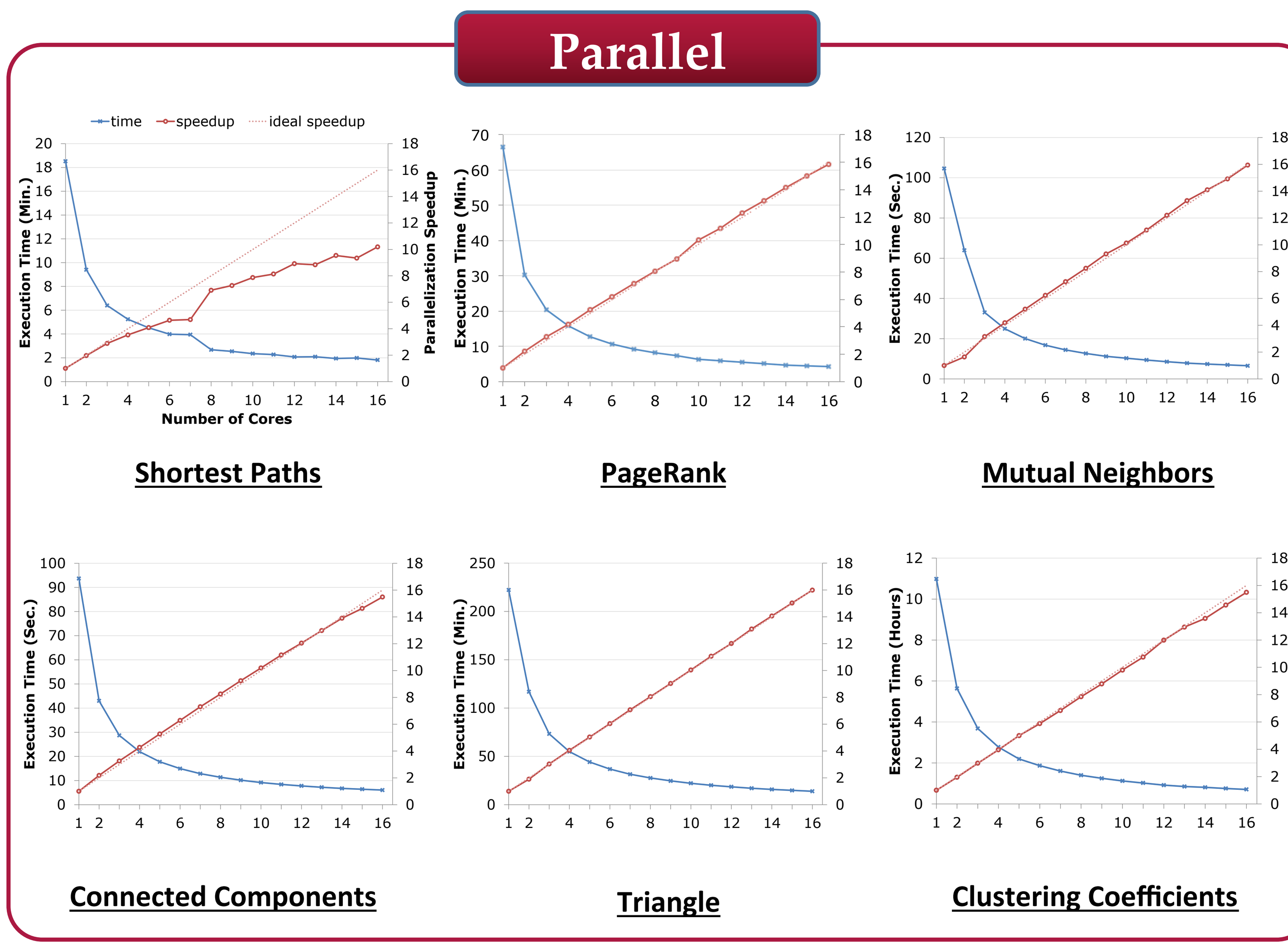
IV. Evaluation

Sequential

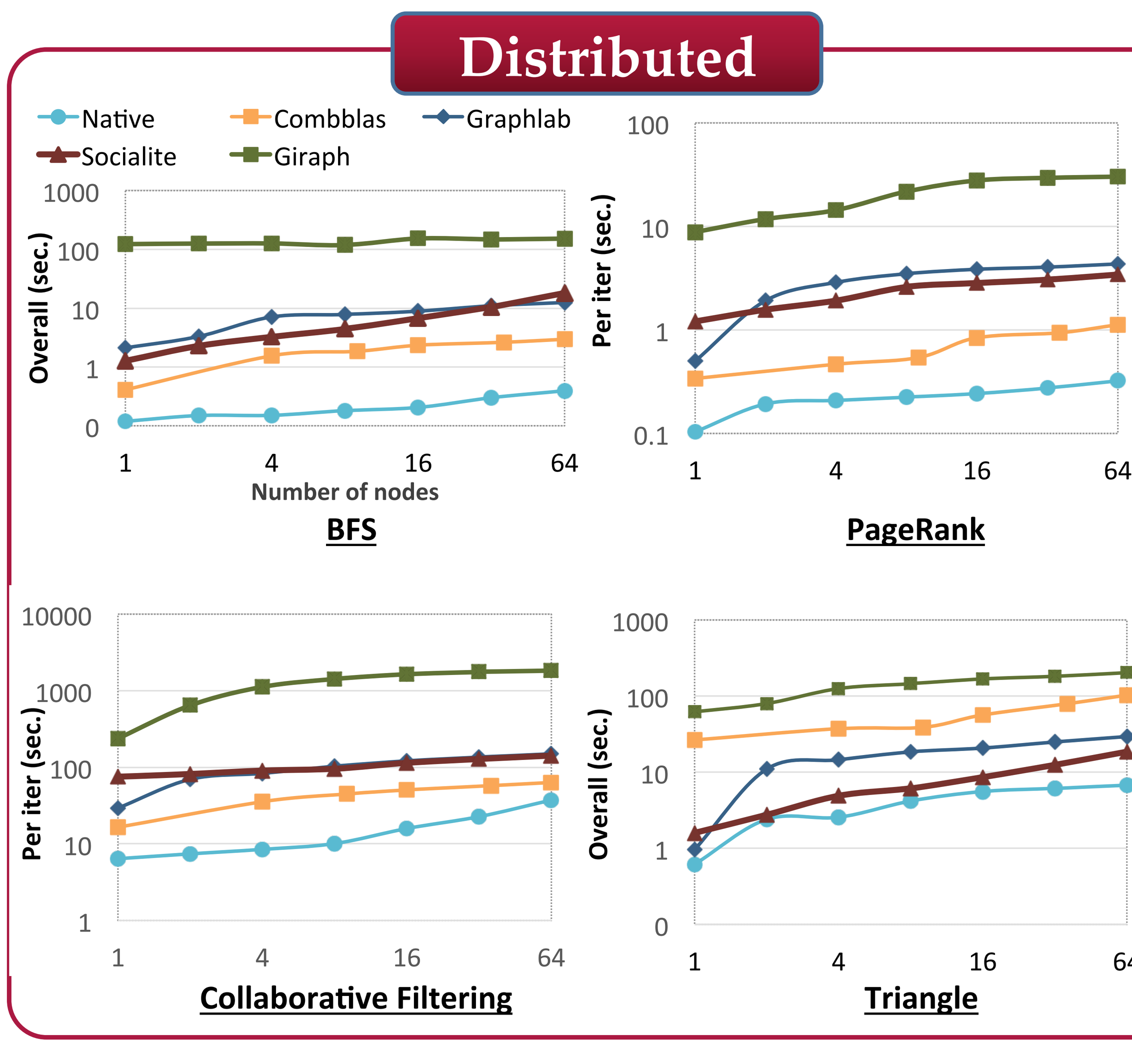


SociaLite vs Java

Parallel



Distributed



SociaLite, Technical Background and Algorithms

Datalog

- Logic programming language as well as Database query language
- Clean syntax and semantics
- Express graph algorithms very well
- Recently used in many domains
  - (programming analysis, network /distributed systems, security, etc)
- Slow performance
- Lack of control in data structure and execution

Recursive Aggregate Functions

- Aggregate functions that can be used in recursive rules
  - Must be a meet operator (idempotent, commutative, and associative)
  - e.g. Min, Max
- Optimizes execution of recursive rules
- Guarantees iterative solution is same as greatest fixed-point solution
- Semi-naïve evaluation can be defined, and gives same result as naïve evaluation.

Graph Algorithms

Shortest Paths

```
`Path(int n, double dist) indexby i.
 Edge(int s, (int t, double len)) indexby s.`
`Path(t, $min(d)) :- t=$SRC, d=0;
                 :- Path(n, d1), Edge(n, t, d2), d=d1+d2.`
```

The recursive rule computes shortest paths from source node ($SRC) to the rest nodes, and stores the distances in *Path* table. The base case of the recursion (written in red) sets the distance to the source node to be 0, and the recursive part (written in green) computes the distances to the rest nodes recursively.

Friends of Friends

```
`Friend(int i, (int f)) indexby i.
 Foaf(int i, (int ff)) indexby i.`
`Foaf(i, ff) :- Friend(i, f1), Friend(f1, ff).`
```

The table *Foaf* stores friends-of-friends relation after evaluating the rule.

PageRank

```
N=1000000 # number of nodes
`Rank(int n:0..$N, int i:iter, double rank).
 Node(int n:0..$N).
 Edge(int s:0..$N, (int t)).
 EdgeCnt(int s:0..$N, int t).`
`Rank(n, 0, $sum(r)) :- Node(n), r=0.15*1.0/$N.`
for i in range(20):
  `Rank(n, $i+1, $sum(r)) :- Node(n),  r=0.15*1.0/$N;
                          :- Rank(p, $i, r1), Edge(p, n), EdgeCnt(p, cnt),
                             r=0.85*r1/cnt.`
```

The *Rank* table stores the PageRank value for each node in the graph. The 2nd column of *Rank* is declared with *iter* option, which indicates that the table will store values at multiple iterations. For example, *Rank(n, 0, r)* has rank values at iteration 0, *Rank(n, 1, r)* has rank values at iteration 1. The first rule (in orange) initializes page rank values at iteration 0. The second rule in the loop computes rank values for 20 iterations; the first rule body (in red) computes the constant term in the algorithm, and the second rule body (in green) adds the contributions from neighbor nodes.

Data Mining Algorithms

K-Means

```
N = 10000 # data size
K = 100  # num of clusters
`Data(int n:0..$N, double[] p).
 Center(int k:0..$K, Avg[] avg).
 Cluster(int n:0..$N, int i:iter, ArgMin min) groupby (2).`
@returns(double)
def getDiff(p, avg):
  return (p[0]-avg[0].value)**2+(p[1]-avg[1]).value)**2
for i in range(100):
  `Center(idx, $avg(p)) :- Data(id, p), Cluster(id, $i, argmin),
                          idx=argmin.value.`
  `Cluster(id, $i, $argmin(idx, d)) :-
                          Data(id, p), Center(idx, a),
                          diff=$getDiff(p, avg).`
```

Data table contains data points to cluster, *Center* table stores K cluster centers, and *Cluster* table maps data points to their current cluster (at a specified iteration).
The Python function *getDiff* (in orange) computes the distance from a data point to a cluster center. The first rule in the loop (in red) finds a new center for each cluster by taking average of data points for each dimension. The second rule (in green) computes new mapping from data points to clusters with updated cluster centers.

Logistic Regression

```
`Data(int n:0..$N, double[] p).
 Weight(int n:iter, double[] w).
 Gradient(int n:iter, double[] g).`
@returns(double)
def sigmoid(x):
  return 1.0/(1+math.exp(-x))
@returns(double, double, double)
def computeWeights(p, y):
  r = 0.01 # learning rate
  w1=r*(p[3]-y)*p[0]
  w2=r*(p[3]-y)*p[1]
  w3=r*(p[3]-y)*p[2]
  return w1, w2, w3
for i in range(0, 100):
  `Gradient($i, $sum(w)) :- Data(id,  p), Weight($i, w1),
                            dot=$dot(w1, p),  y=$sigmoid(dot),
                            w = $computeWeights(p, y).`
  `Weight($i+1, w) :- Weight($i, w1), Gradient($i, g), w=$sum1(w1, g).`
```

Update rule (gradient descent) $\theta_j := \theta_j + \alpha(y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)}$
Data table contains data points, Gradient table stores gradient value (green in the equation), and *Weight* table stores weights that are used to predict the class of the data points (red in the equation).
The first rule in the loop computes the gradient values, and the second rule updates the weights for next iteration.

STANFORD UNIVERSITY