# Socia*Lite*: A Python-Integrated Query Language for Big Data Analysis

**Jiwon Seo** *Jongsoo Park Jaeho Shin Stephen Guo Monica Lam

STANFORD UNIVERSITY
MOBISOCIAL RESEARCH GROUP

*Intel Parallel Research Lab

# Why another Big Data Platform?

Existing platforms are …

- Not fast enough (not network bandwidth)
- Too difficult (low-level primitives)
- Too many (sub) frameworks
  - Graph analysis
  - Data mining
  - Machine learning

# Introducing SociaLite

SociaLite is a high-level query language

- Compiled to parallel code
  - 1,000x hadoop
- Hadoop compatible
- Python integration
- Designed for graph analysis
- Good for data mining & machine learning

# Outline

- Language
  - Tables
  - Queries
  - Python integration
  - Approximation
- Analysis algorithms
  - Shortest paths, PageRank
  - K-Means, Logistic regression
- Evaluation
- Demo

# Distributed In-Memory Tables

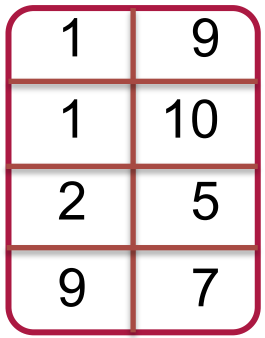
- Primary data structure in SociaLite
- Column oriented storage

*Table* (<type> $c_x$, ..., (<type> $c_y$, ... (<type> $c_z$...))).

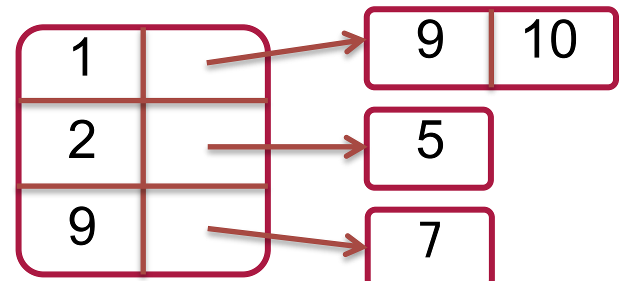
- <type>
  - Primitive types
  - Object types

# Distributed In-Memory Tables

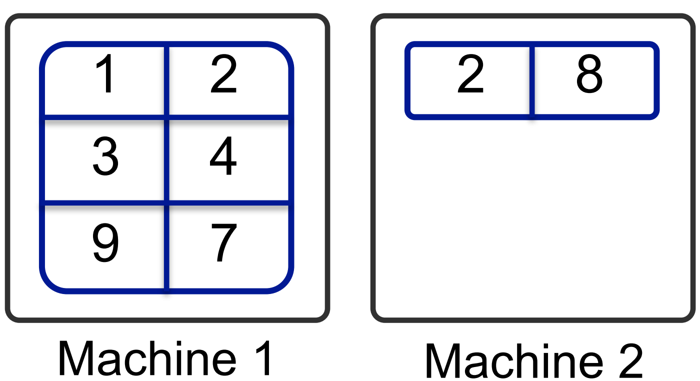
Foo(int x, int y).



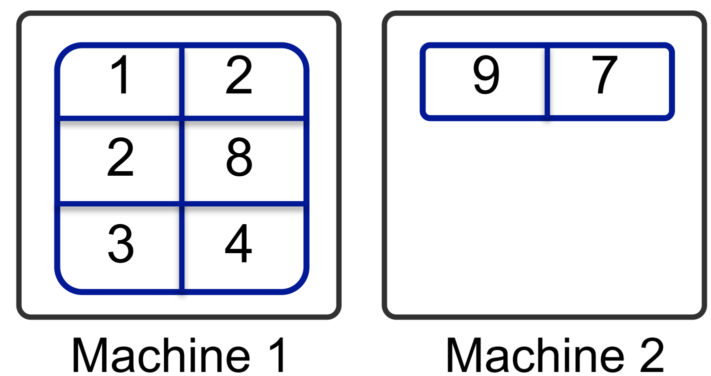

Foo(int x, (int y)).




Bar[int x](int y).




Bar[int x:0..10](int y).



# Distributed In-Memory Tables

## Table options

- indexby *<column>*
- sortby *<column>*
- multiset

```
Foo(int x, int y) indexby x.
```

```
Foo(int x, int y) sortby x.
```

```
Foo(int x, int y) multiset.
```

## Column options

- range
- (distributed) partition

```
Foo(int x:0..100, int y).
```

```
Foo[int x](int y).
```

# Rules (Queries)

$$\text{Foo(a, c)} \ \text{:-} \ \text{Bar(a, } \mathbf{b}\text{), Qux(}\mathbf{b}\text{, c).}$$

$$\underbrace{\text{Foo(a, c)}}_{\text{Rule head}} \ \text{:-} \ \underbrace{\text{Bar(a, } \mathbf{b}\text{), Qux(}\mathbf{b}\text{, c).}}_{\text{Rule body}}$$

# Rules

Foo(a, c) :- Bar(a, **b**), Qux(**b**, c).

Foo

| | |
|---|---|
| | |

$\leftarrow$

Bar

| | |
|---|---|
| 1 | 2 |
| 1 | 3 |
| 8 | 4 |
| 8 | 7 |
| 9 | 11 |

$\bowtie$

Qux

| | |
|---|---|
| 2 | 9 |
| 2 | 10 |
| 5 | 4 |
| 10 | 7 |
| 11 | 9 |

# Rules

Foo(a, c) :- Bar(a, **b**), Qux(**b**, c).

Foo

| | |
|---|---|
| 1 | 9 |
| 1 | 10 |

←

Bar

| | |
|---|---|
| 1 | 2 |
| 1 | 3 |
| 8 | 4 |
| 8 | 7 |
| 9 | 11 |

⋈

Qux

| | |
|---|---|
| 2 | 9 |
| 2 | 10 |
| 5 | 4 |
| 10 | 7 |
| 11 | 9 |

# Rules

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

Foo $\leftarrow$ Bar $\bowtie$ Qux

**Foo**

| | |
|---|---|
| 1 | 9 |
| 1 | 10 |
| 9 | 9 |

**Bar**

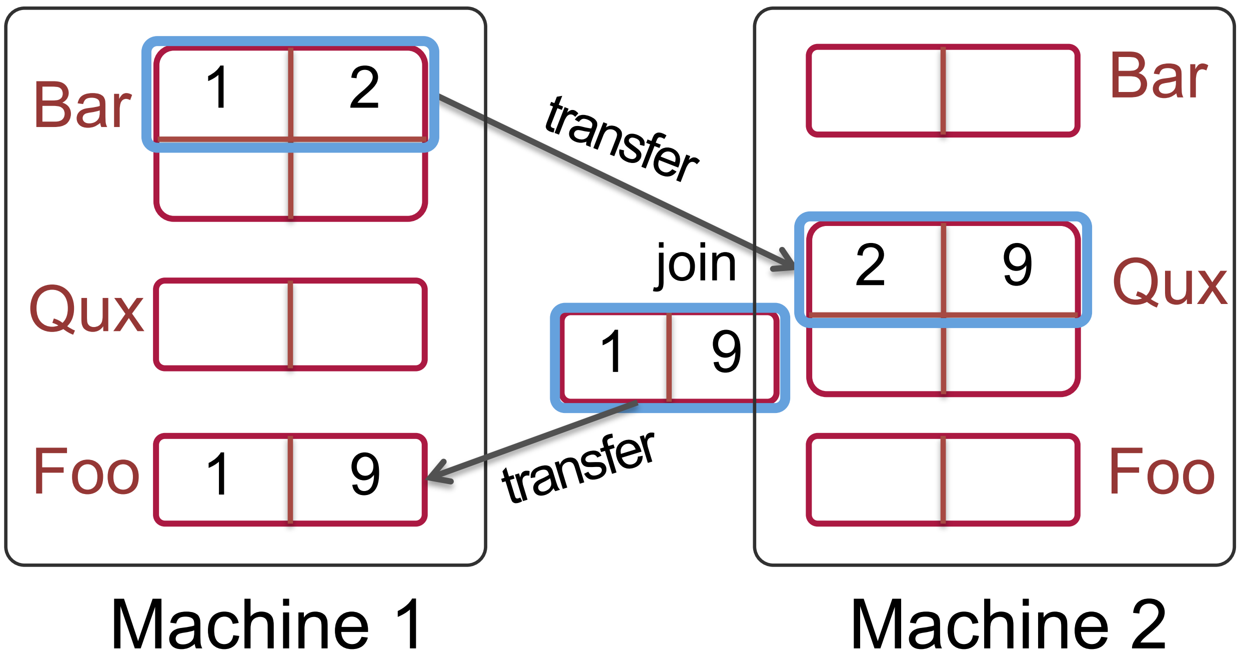
| | |
|---|---|
| 1 | 2 |
| 1 | 3 |
| 8 | 4 |
| 8 | 7 |
| 9 | 11 |

**Qux**

| | |
|---|---|
| 2 | 9 |
| 2 | 10 |
| 5 | 4 |
| 10 | 7 |
| 11 | 9 |

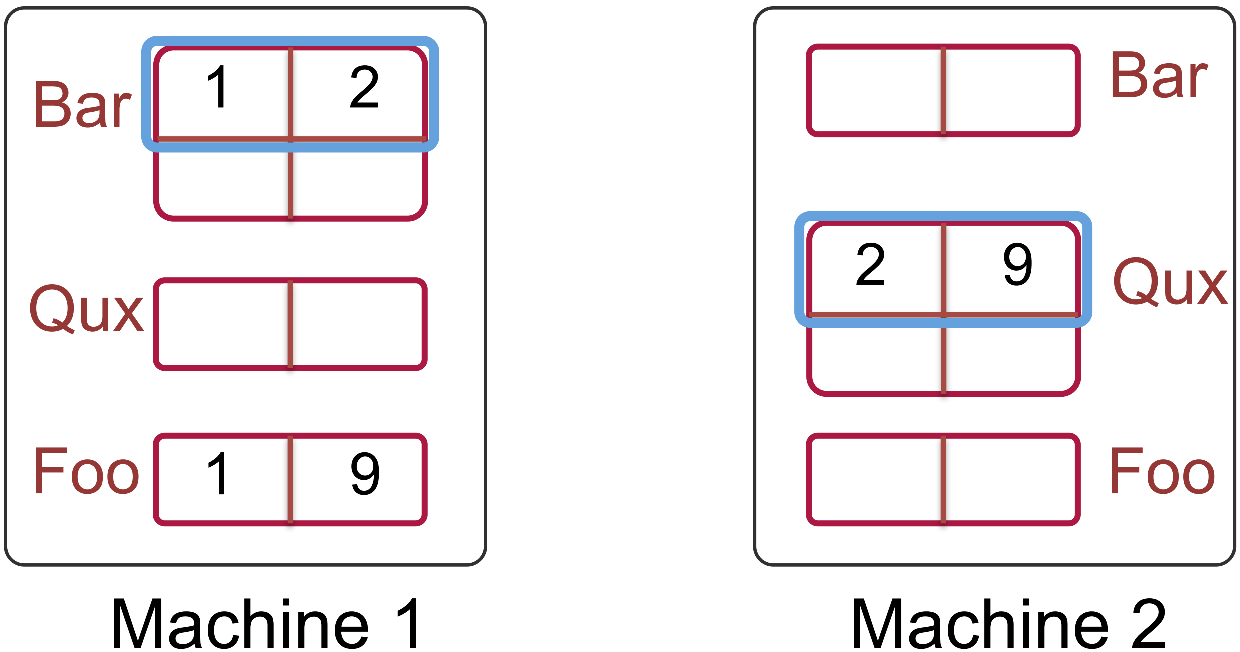
# Distributed Execution

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



# Distributed Execution

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



# Aggregation

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

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

- Built-in aggregate functions
  - min, max, sum, avg, argmin
- User-defined functions
  - in Java or Python

# Recursive Rules

- Head table also appears in rule body

**Foo**(a,c) :- **Foo**(a,b), Bar(b,c).

- Semantics
  - rule executed repeatedly until no changes to Foo

# Recursive Rules

```
`Edge(int s, (int t, double len)) indexby s.
 Path(int n, double dist) indexby n.`

`Path(t, $min(d)) :- t=$SRC, d=0;
                  :- Path(n, d1), Edge(n, t, d2), d=d1+d2.`
```

Shortest Path algorithm in recursion + aggregation

# Python (Jython) Integration

- SociaLite queries in Python code
  - `Queries are quoted in backtick`
  
  → Preprocessing with Python import-hook
- Python ←→ SociaLite
  - Python functions, variables are accessible in SociaLite queries
  - SociaLite tables are readable from Python

# Python (Jython) Integration

```
print “This is Python code!”
# now we use SociaLite queries below
`Foo[int i](String s).
 Foo(i, s) :- i=42, s=“the answer”.`
v=“Python variable”
`Foo(i, s) :- i=43, s=$v.`
@returns(str)
def func(): return “Python func”
`Foo(i, s) :- i=44, s=$func().`
for i, s in `Foo(i,  s)`:
   print i, s
```

# CPython Integration

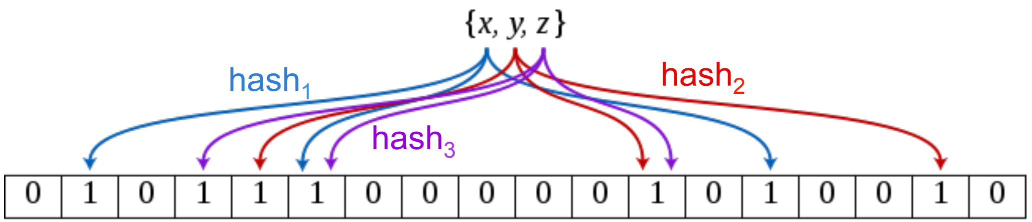
- JyNI – Jython Native Interface
  - Stefan Richthofer
  - http://jyni.org
- To support CPython extensions in Jython
  - NumPy, SciPy, Pandas, etc
- Tkinter works on Jython

# Approximation

## Approximate Computation

- Bloom Filter, FM Sketch

## BloomFilter




- Bitmap-based set
- Quickly check set membership

→ false positives, but no false negatives

- In SociaLite, useful to store large intermediate results approximately

# Approximation w/ Bloom Filter

```
Foaf(i, ff) :- Friend(i, f), Friend(f, ff).
LocalCount(i, $inc(1)) :- Foaf(i, ff), Attr(ff, "Some Attr").
```

# Approximation w/ Bloom Filter

```
Foaf(i, ff) :- Friend(i, f), Friend(f, ff).
LocalCount(i, $inc(1)) :- Foaf(i, ff), Attr(ff, "Some Attr").
```

(2nd column of Foaf table is represented with a Bloom filter)

# Approximation w/ Bloom Filter

Foaf(i, **ff**) :- Friend(i, f), Friend(f, ff).
LocalCount(i, $inc(1)) :- Foaf(i, **ff**), Attr(ff, "Some Attr").

(2nd column of Foaf table is represented with a Bloom filter)

| | Exact | Approximation | Comparison |
|---|---|---|---|
| Exec time (min) | 28.9 | 19.4 | 32.8% faster |
| Memory usage(GB) | 26.0 | 3.0 | 11.5% usage |
| Accuracy(<10% error) | 100.0% | 92.5% | |

* LiveJournal (4.8M nodes, 68M edges)

# Analysis Algorithms

- Graph algorithms
  - Shortest Paths
  - PageRank
- Data mining/machine learning algorithms
  - K-Means Clustering
  - Logistic regression

# Graph Algorithm

- Shortest Path

```
`Edge(int s, (int t, double len)) indexby s.
 Path(int n, double dist) indexby n. `

`Path(t, $min(d)) :- t=$SRC, d=0;
                  :- Path(n, d1), Edge(n, t, d2), d=d1+d2.`
```

# Graph Algorithm

- PageRank

```
`Rank(n, 0, r) :- Node(n), r=1.0/$N.`
for t in range(30):
    `Rank(pi, $t+1, $sum(r))  :- Node(pi),  r=0.15*1.0/$N;
                              :- Rank(pj, $t, r1), Edge(pj, pi),
                                 EdgeCnt(pj, cnt), r=0.85*r1/cnt.`
```

# Graph Algorithm

- PageRank

```
`Rank(n, 0, r) :- Node(n), r=1.0/$N.`
for t in range(30):
    `Rank(p_i, $t+1, $sum(r))  :- Node(p_i), r=0.15*1.0/$N;
                               :- Rank(p_j, $t, r_1), Edge(p_j, p_i),
                                  EdgeCnt(p_j, cnt), r=0.85*r_1/cnt.`
```

At $t = 0$, an initial probability distribution is assumed, usually

$$PR(p_i; 0) = \frac{1}{N}.$$

At each time step, the computation, as detailed above, yields

$$PR(p_i; t+1) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j; t)}{L(p_j)}$$

# Data Mining Algorithm

- K-Means Clustering

```
for i in range(50):
  `Center(cid, $i+1, $avg(p)) :- Data(id, p), Cluster(id, $i, c),
                                 cid=c.value.`
  `Cluster(id, $i+1, $argmin(idx, d)) :-
                             Data(id, p), Center(idx, $i+1, a),
                             d=$getDiff(p, a).`
```

# Data Mining Algorithm

- Logistic Regression

```
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=$vecSum (w1, g).`
```

# Evaluation

Benchmark algorithms (graph algorithms)

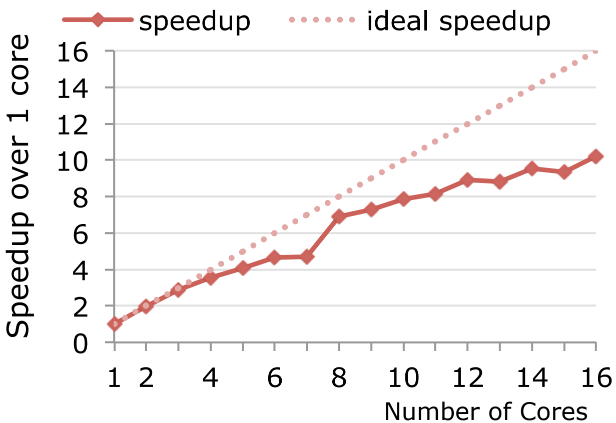
- Shortest-Paths
- PageRank
- Mutual Neighbors
- Connected Components
- Finding Triangles
- Clustering Coefficients

→ Evaluation on a multi-core & distributed cluster

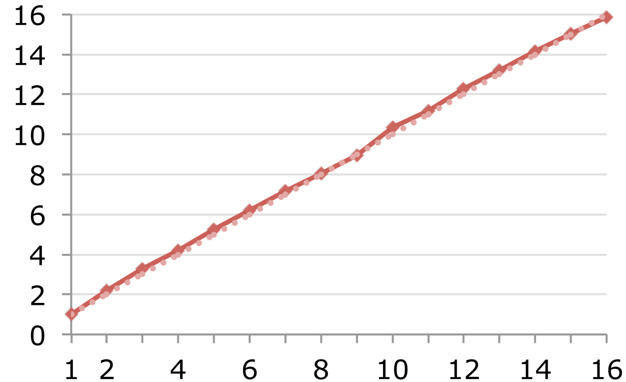
# Input Graph for Multi-Core

| Source | Size | Machine |
|---|---|---|
| Friendster | 120M nodes<br>2.5B edges | Intel Xeon E5-2670<br>16 cores(8+8)<br>2.60GHz<br>20MB last-level cache<br>256GB memory |

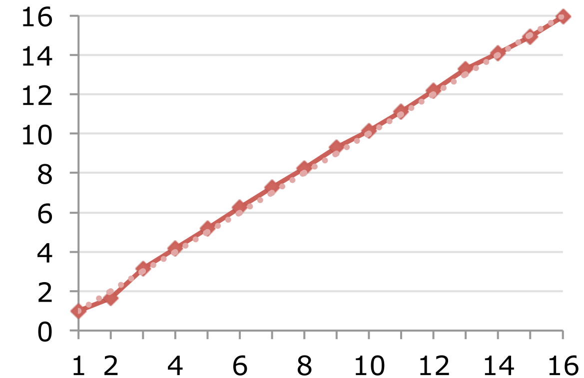
# Parallel Performance (Multi-Core)



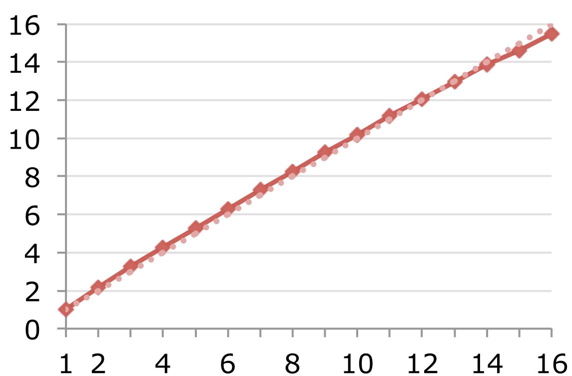

**Shortest Paths**



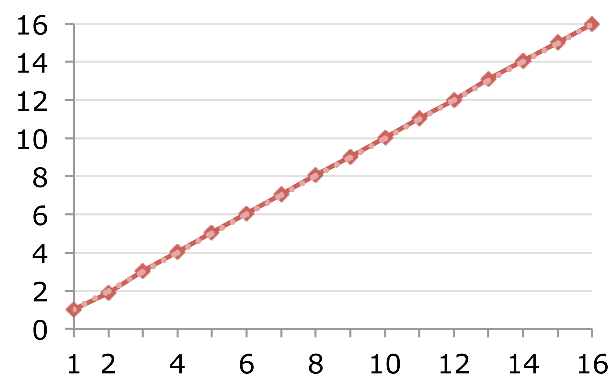
**PageRank**



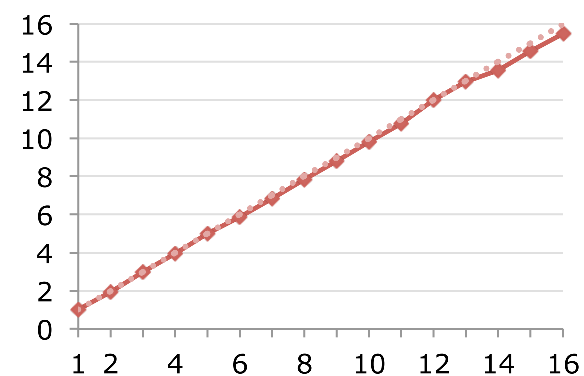
**Mutual Neighbors**



**Connected Components**



**Triangle**



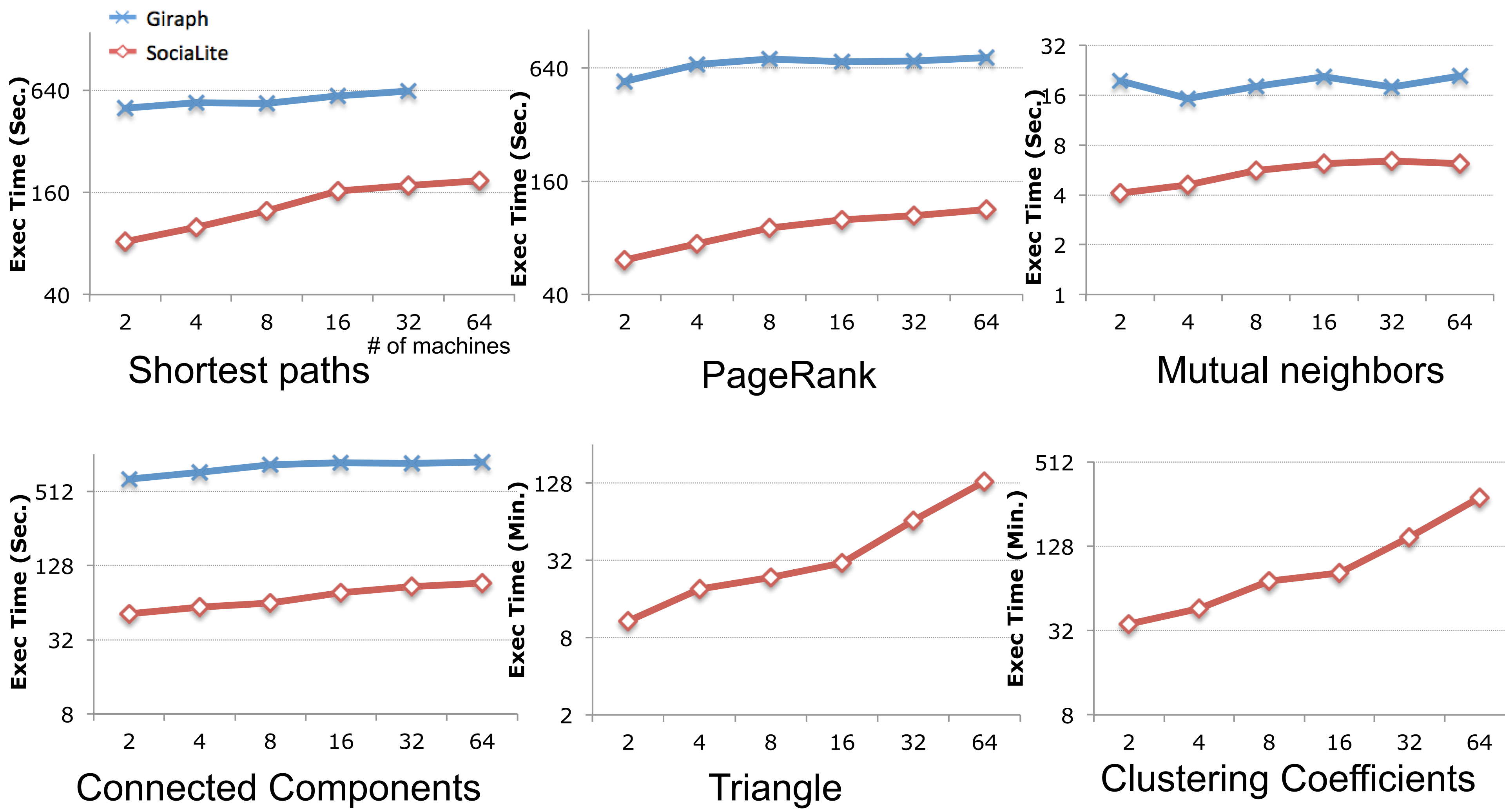
**Clustering Coefficients**

# Input Graph for Distributed Evaluation

| Source | Size | Machine |
|---|---|---|
| Synthetic Graph* | up to 268M nodes 4.3B edges (weak scaling) | 64 Amazon EC2 Instances Intel Xeon X5570, 8 cores 23GB memory |

*RMAT algorithm, Graph 500 Generator

# Giraph (Pregel) vs SociaLite



# Programmability Comparison

- Giraph vs SociaLite (lines of code)

| | Giraph | SociaLite |
|---|---:|---:|
| Shortest Paths | 232 | 4 |
| PageRank | 146 | 13 |
| Mutual Neighbors | 169 | 6 |
| Connected Components | 122 | 9 |
| Triangles | 181 | 6 |
| Clustering Coefficients | 218 | 12 |
| Total | 1,068 | 50 |

→ SociaLite is 20x simpler!

# Comparing More Graph Frameworks

- Collaboration with Intel Parallel Research Lab*
- Compared frameworks
  - SociaLite
  - Giraph
  - GraphLab
  - Combinatorial BLAS
- Native Implementation in C, assembly – optimal

* Navigating the Maze of Graph Analytics Frameworks using Massive Graph Datasets, Satish et al., *SIGMOD '14*

# Comparing More Graph Frameworks

- Benchmark Algorithms
  - BFS (Breadth First Search)
  - PageRank
  - Collaborative Filtering
  - Triangle
- Evaluation on Intel cluster

– Intel Xeon, 24 cores 2.7GHz, 64GB memory, InfiniBand network

- Input Graph

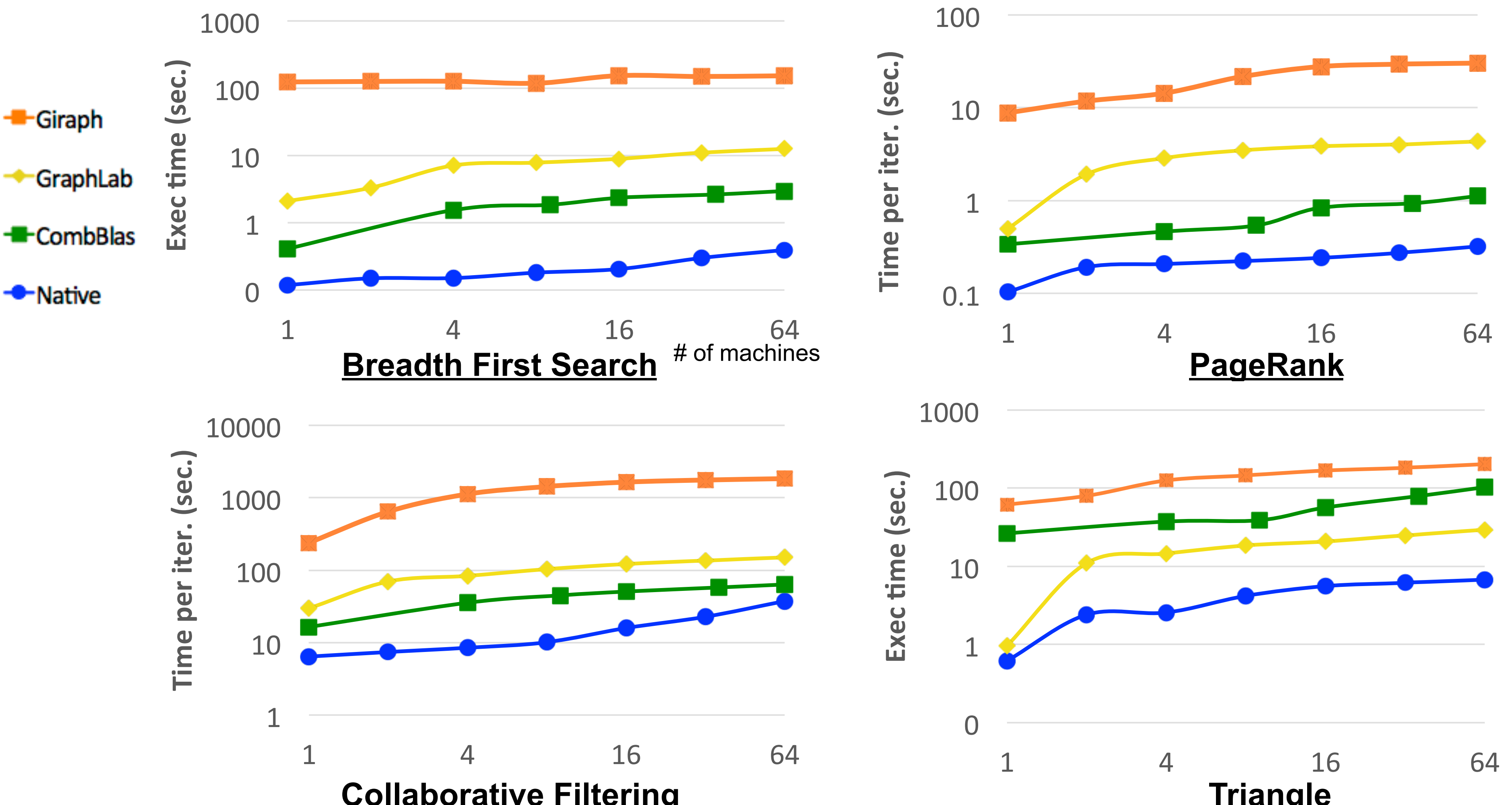
– up to 512M nodes, 16G edges (weak scaling)

# Programmability

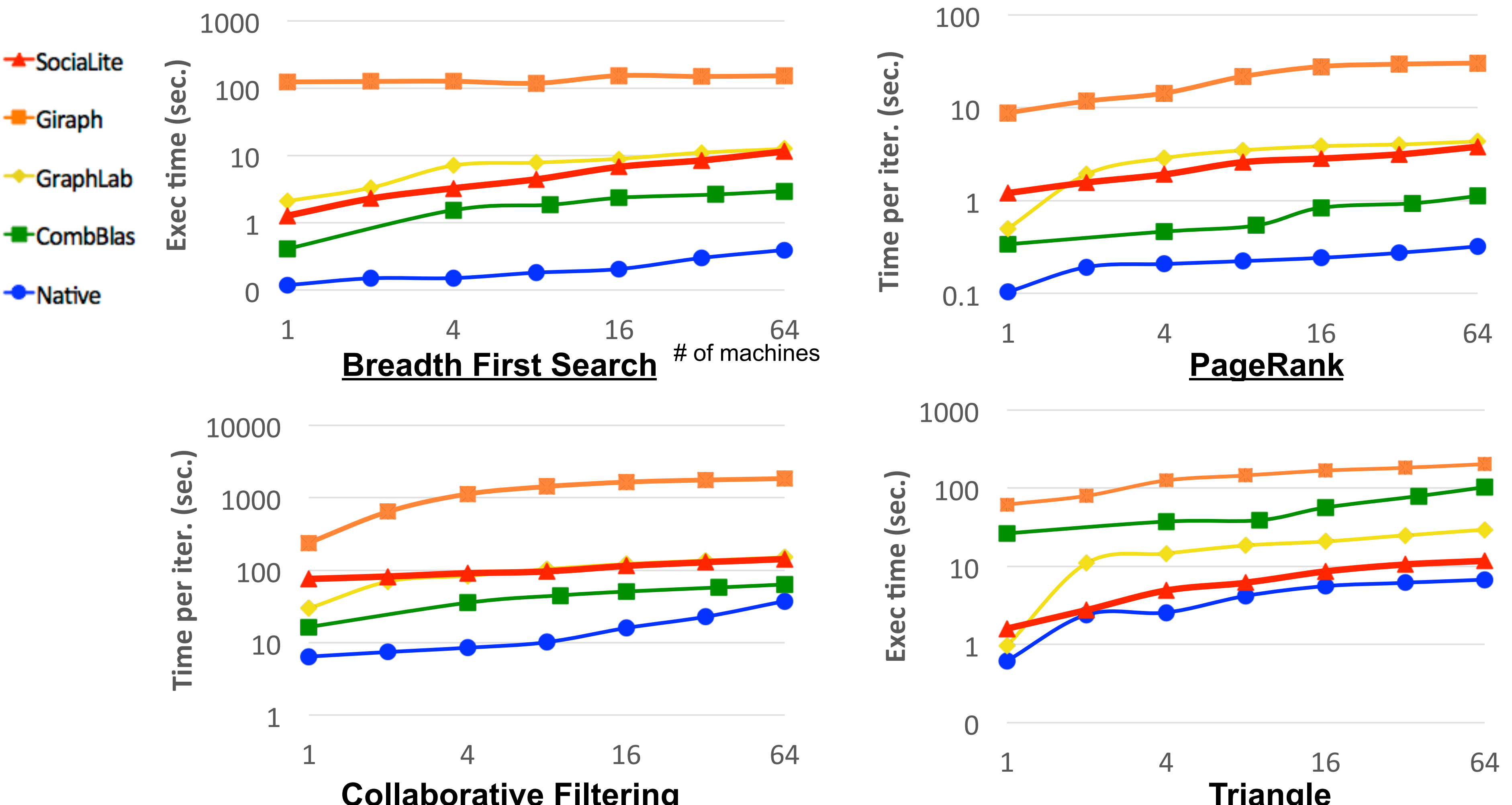
- BFS (Breadth First Search)

| | Lines of Code | Development Time |
|---|---|---|
| SociaLite | 4 | 1~2 min |
| Giraph | 200 | 1~2 hours |
| GraphLab | 180 | 1~2 hours |
| Combinatorial BLAS | 450 | a few hours |
| Native | > 1000 | > A few months |

# Distributed Execution – Comparison



# Distributed Execution – Comparison

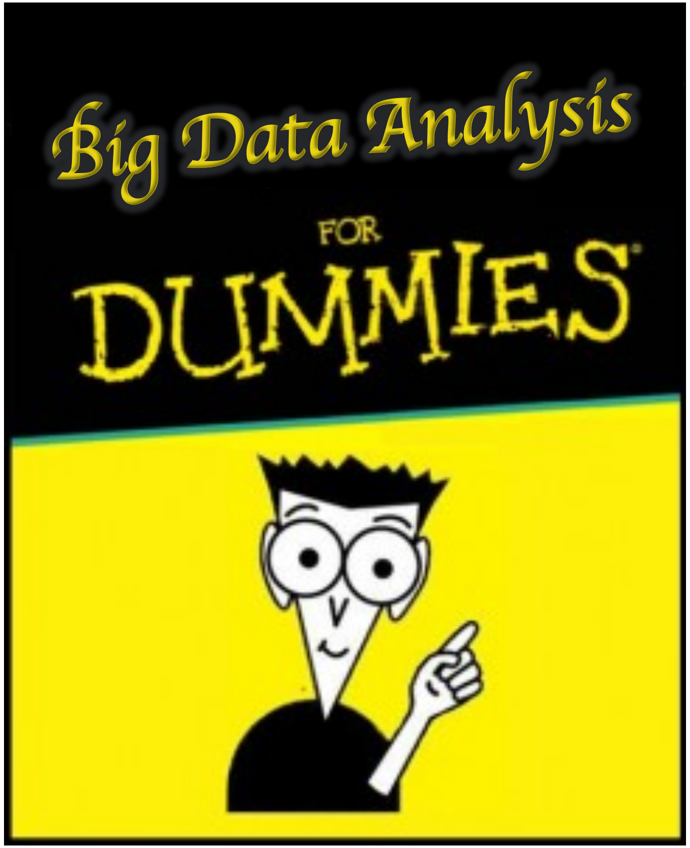


# Work In-Progress

- Collaboration with LinkedIn
  - Real-time pattern matching queries
  - Off-line analysis

- Discussing collaboration with other companies
  - Kakao
  - etc

# Summary




- 20x easier than Giraph
- 10x faster than Giraph
- As fast as, or faster than
  - GraphLab, CombBlas
- How?
  - High-level query interface
  - Compiler optimizations
  - Python integration

# Demo

DBLP (CS bibliography)

- Co-authorship graph
  - vertices: authors (1 million)
  - edges: co-authorship (10 million)
- Guido van Rossum's academic network
  - How Guido is connected to

    Armin Rigo (PyPy)

    Jim Hugunin (Jython, IronPython)
  - Run shortest-paths from Guido & visualize

# Questions?

- Visit http://socialite.stanford.edu for
  - Trying out
  - Participation (Apache v2)