Web-Scale Graph Analytics with Apache Spark

Tim Hunter, Databricks

#EUds6

About Me

- Tim Hunter
- Software engineer @ Databricks

- Ph.D. from UC Berkeley in Machine Learning
- Very early Spark user

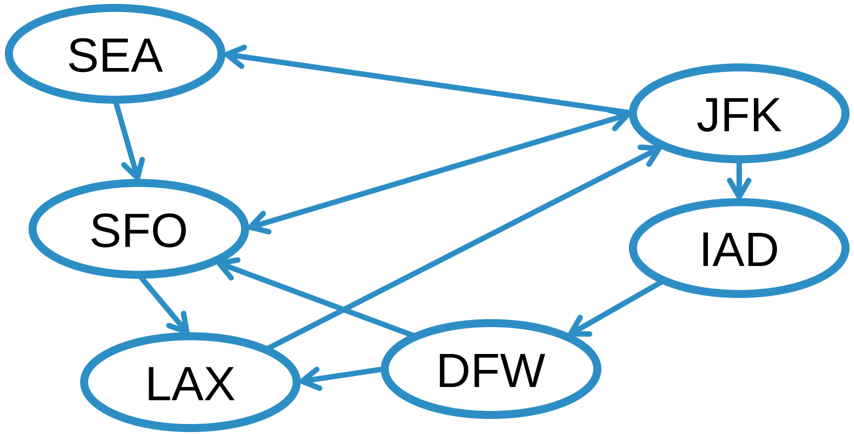
- Contributor to MLlib
- Co-author of TensorFrames, GraphFrames, Deep Learning Pipelines

Outline

- Why GraphFrames
- Writing scalable graph algorithms with Spark
  - *Where is my vertex?* Indexing data
  - *Connected Components*: implementing complex algorithms with Spark and GraphFrames
  - *The Social Network*: real-world issues
- Future of GraphFrames

Graphs are everywhere

Example: airports & flights between them



Vertices:

id	City	State
"JFK"	"New York"	NY

Edges:

src	dst	delay	tripID
"JFK"	"SEA"	45	1058923

Apache Spark's GraphX library

- General-purpose graph processing library
- Built into Spark
- Optimized for fast distributed computing
- Library of algorithms: PageRank, Connected Components, etc.

Issues:

- No Java, Python APIs
- Lower-level RDD-based API (vs. DataFrames)
- Cannot use recent Spark optimizations: Catalyst query optimizer, Tungsten memory management

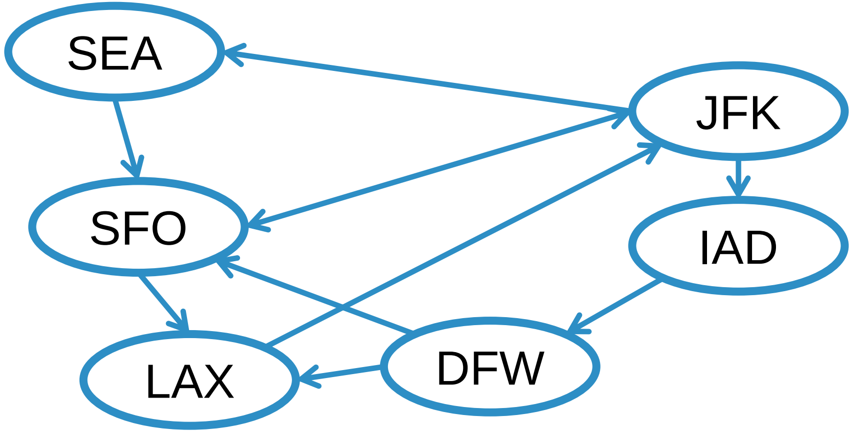
The GraphFrames Spark Package

Brings DataFrames API for Spark

- Simplifies interactive queries
- Benefits from DataFrames optimizations
- Integrates with the rest of Spark ecosystem

Collaboration between Databricks, UC Berkeley & MIT

Dataframe-based representation



`vertices` DataFrame

id	City	State
"JFK"	"New York"	NY
"SEA"	"Seattle"	WA

`edges` DataFrame

src	dst	delay	tripID
"JFK"	"SEA"	45	1058923
"DFW"	"SFO"	-7	4100224

Supported graph algorithms

- Find Vertices:
  - PageRank
  - **Motif finding**
- Communities:
  - **Connected Components**
  - Strongly Connected Components
  - Label propagation (LPA)
- Paths:
  - **Breadth-first search**
  - Shortest paths
- Other:
  - **Triangle count**
  - SVD++

(Bold: native DataFrame implementation)

Assigning integral vertex IDs

… lessons learned

Pros of integer vertex IDs

GraphFrames take arbitrary vertex IDs.

→ convenient for users

Algorithms prefer integer vertex IDs.

→ optimize in-memory storage

→ reduce communication

Our task: Map unique vertex IDs to unique (long) integers.

The hashing trick?

- Possible solution: hash vertex ID to long integer

- What is the chance of collision?
  - 1 - (N-1)/N * (N-2)/N * …
  - seems unlikely with long range $N=2^{64}$
  - with 1 billion nodes, the chance is ~5.4%

- Problem: collisions change graph topology.

Name	Hash
Sue Ann	84088
Joseph	-2070372689
Xiangrui	264245405
Felix	67762524

Generating unique IDs

Spark has built-in methods to generate unique IDs.

- RDD: `zipWithUniqueId()`, `zipWithIndex()`
- DataFrame: `monotonically_increasing_id()`

Possible solution: just use these methods

How it works

Partition 1	
Vertex	**ID**
Sue Ann	0
Joseph	1

Partition 2	
Vertex	**ID**
Xiangrui	100 + 0
Felix	100 + 1

Partition 3	
Vertex	**ID**
Veronica	200 + 0
…	200 + 1

… but not always

- DataFrames/RDDs are immutable and reproducible by design.

- However, records do not always have stable orderings.
  - –distinct
  - –repartition

- `cache()` does not help.

Partition 1	
Vertex	**ID**
Xiangrui	0
Joseph	1

repartition
distinct
shuffle

Partition 1	
Vertex	**ID**
Joseph	0
Xiangrui	1

Our implementation

We implemented (v0.5.0) an expensive but correct version:

1. (hash) re-partition + distinct vertex IDs
2. sort vertex IDs within each partition
3. generate unique integer IDs

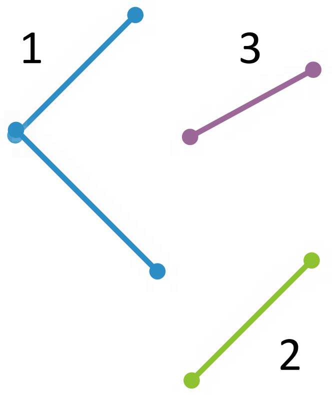
Connected Components

Connected Components

Assign each vertex a component ID such that vertices receive the same component ID iff they are connected.

Applications:

- fraud detection
  - [Spark Summit 2016 keynote from Capital One]
- clustering
- entity resolution



Naive implementation (GraphX)

1. Assign each vertex a unique component ID.
2. Iterate until convergence:
   – For each vertex v, update:

     component ID of v ← Smallest component ID in neighborhood of v

Pro: easy to implement

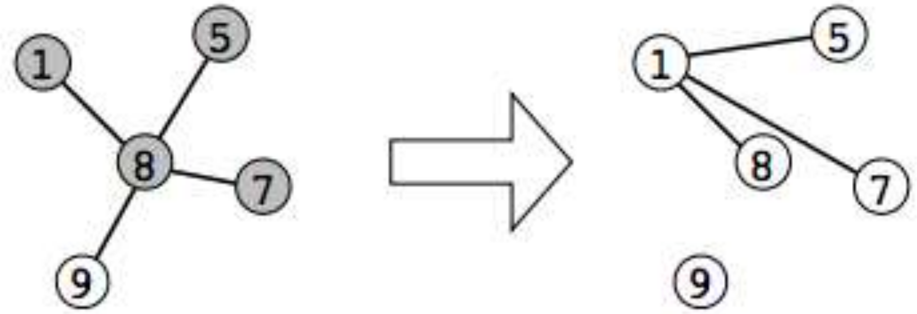
Con: slow convergence on large-diameter graphs

Small-/large-star algorithm

Kiveris et al. "Connected Components in MapReduce and Beyond."

1. Assign each vertex a unique ID.
2. Iterate until convergence:
   - (small-star) for each vertex,
     connect smaller neighbors to smallest neighbor
   - (big-star) for each vertex,
     connect bigger neighbors to smallest neighbor (or itself)

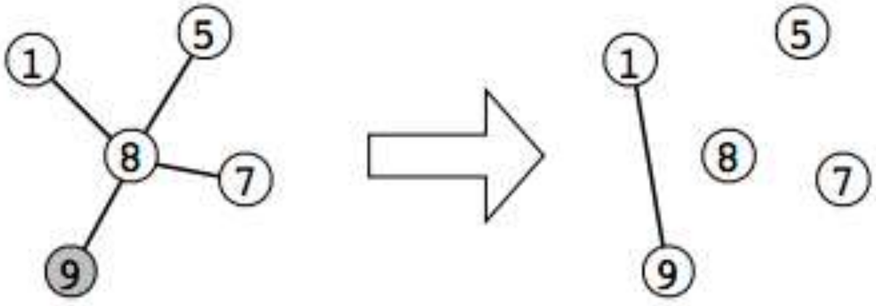
Small-star operation



(a) The small-star operation at node 8.

Kiveris et al., Connected Components in MapReduce and Beyond

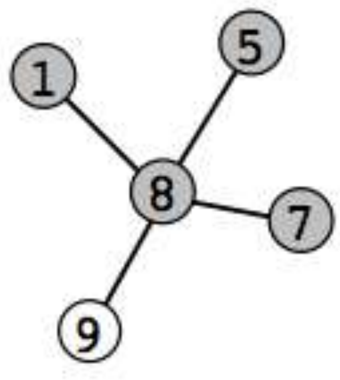
Big-star operation



(b) The large-star operation at node 8.

Kiveris et al., Connected Components in MapReduce and Beyond

Another interpretation



adjacency matrix →

	1	5	7	8	9
1				x	
5				x	
7				x	
8					x
9					

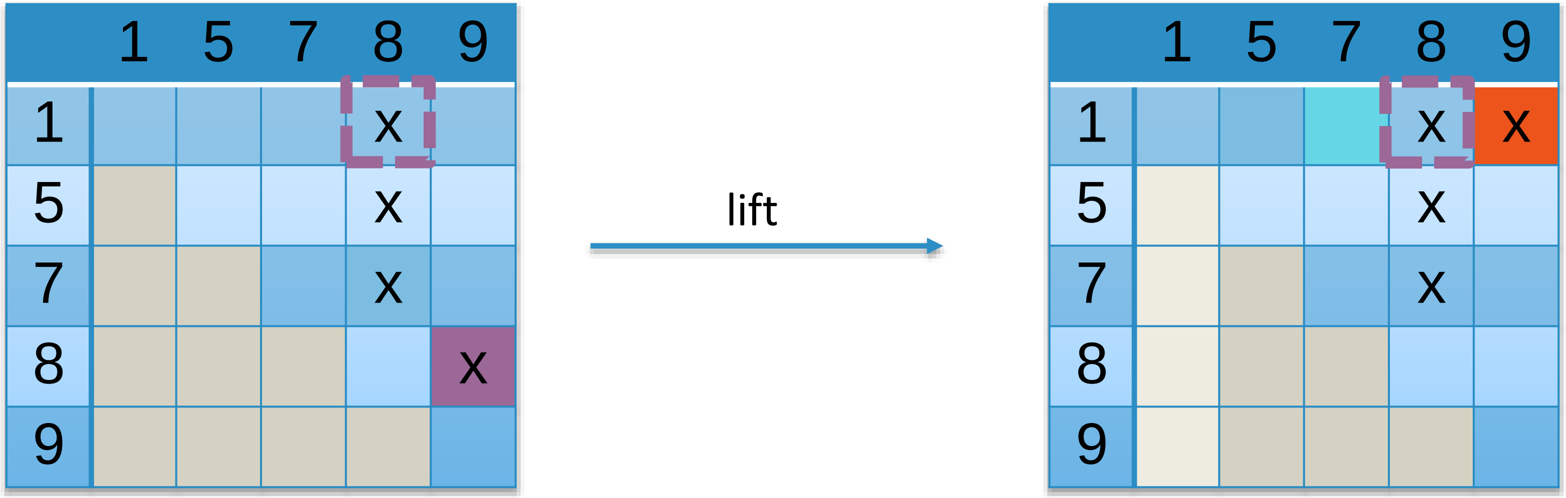
Small-star operation

	1	5	7	8	9
1				x	
5				x	
7				x	
8					x
9					

rotate & lift

	1	5	7	8	9
1		x	x	x	
5					
7					
8					x
9					

Big-star operation



Convergence

	1	5	7	8	9
1	x	x	x	x	x
5					
7					
8					
9					

Properties of the algorithm

- Small-/big-star operations do not change graph connectivity.
- Extra edges are pruned during iterations.
- Each connected component converges to a star graph.
- Converges in $\log^2$(#nodes) iterations

Implementation

Iterate:

- filter
- self-join

Challenge: handle these operations at scale.

Skewed joins

Real-world graphs contain big components.
The ”Justin Bieber problem” at Twitter

→ data skew during connected components iterations

src	Component id	neighbors
0	0	2,000,000
1	0	10
2	3	5

join

src	dst
0	1
0	2
0	3
0	4
…	…
0	2,000,000
1	3
2	5

Skewed joins

(#nbrs > 1,000,000)

src	Component id	neighbors
0	0	2,000,000

broadcast join

src	dst
0	1
0	2
0	3
0	4
…	…
0	2,000,000

union

1	0	10
2	3	5

hash join

1	3
2	5

Checkpointing

We checkpoint every 2 iterations to avoid:

- query plan explosion (exponential growth)
- optimizer slowdown
- disk out of shuffle space
- unexpected node failures

Experiments

twitter-2010 from WebGraph datasets (small diameter)

- 42 million vertices, 1.5 billion edges

16 r3.4xlarge workers on Databricks

- GraphX: 4 minutes
- GraphFrames: 6 minutes
  - algorithm difference, checkpointing, checking skewness

Experiments

uk-2007-05 from WebGraph datasets

- 105 million vertices, 3.7 billion edges

16 r3.4xlarge workers on Databricks

- GraphX: 25 minutes
  - slow convergence
- GraphFrames: 4.5 minutes

Experiments

regular grid 32,000 x 32,000 (large diameter)

- 1 billion nodes, 4 billion edges

32 r3.8xlarge workers on Databricks

- GraphX: failed
- GraphFrames: 1 hour

Future improvements

GraphFrames

- better graph partitioning
- letting Spark SQL handle skewed joins and iterations
- graph compression

Connected Components

- local iterations
- node pruning and better stopping criteria

Thank you!

- http://graphframes.github.io
- https://docs.databricks.com

2 types of graph representations

Algorithm-based

Standard & custom algorithms
Optimized for batch processing

Query-based

Motif finding
Point queries & updates

GraphFrames: Both algorithms & queries (but not point updates)

Graph analysis with GraphFrames

Simple queries

Motif finding

Graph algorithms

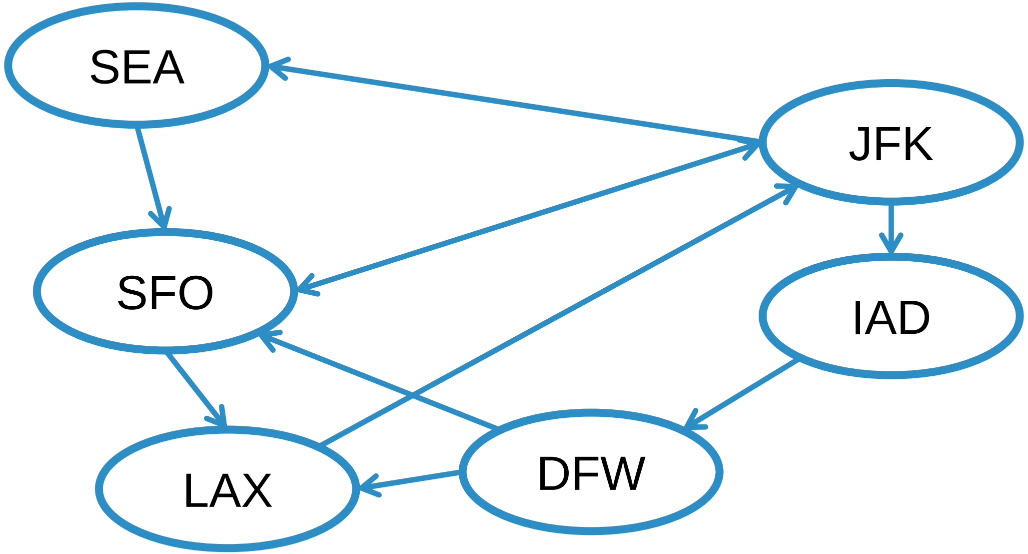
Simple queries

SQL queries on vertices & edges

Motif finding

Search for structural patterns within a graph.

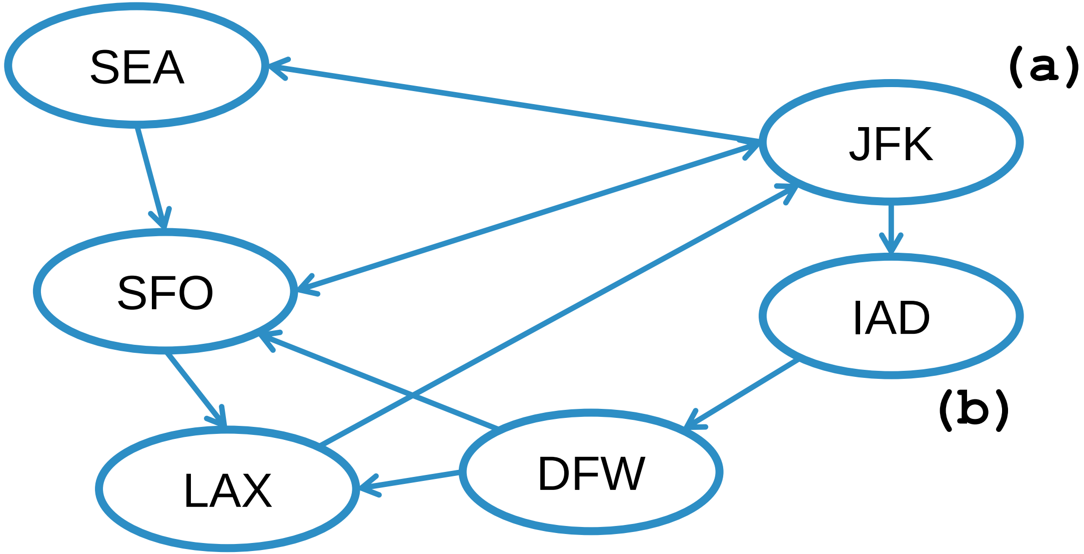
```
val paths: DataFrame =
  g.find("(a)-[e1]->(b);
          (b)-[e2]->(c);
         !(c)-[]->(a)")
```



Motif finding

Search for structural patterns within a graph.

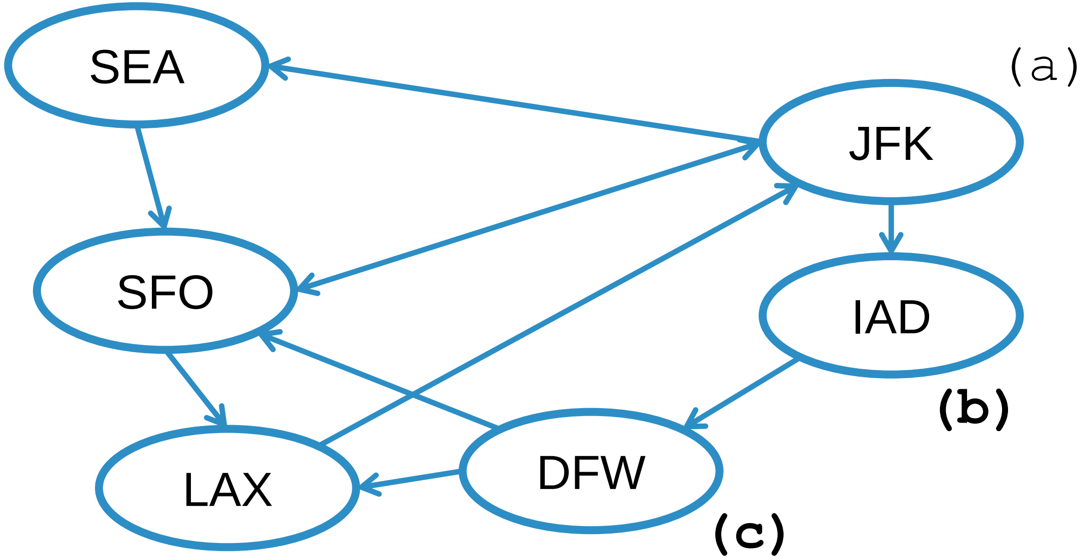
```
val paths: DataFrame =
  g.find("(a)-[e1]->(b);
          (b)-[e2]->(c);
         !(c)-[]->(a)")
```



Motif finding

Search for structural patterns within a graph.

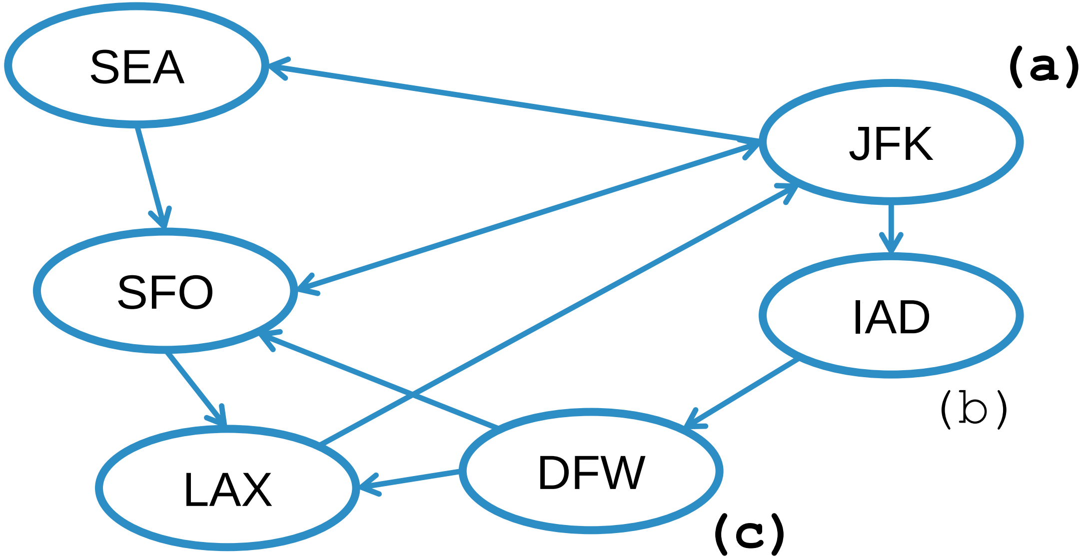
```
val paths: DataFrame =
  g.find("(a)-[e1]->(b);
          (b)-[e2]->(c);
          !(c)-[]->(a)")
```



Motif finding

Search for structural patterns within a graph.

```
val paths: DataFrame =
  g.find("(a)-[e1]->(b);
          (b)-[e2]->(c);
          !(c)-[]->(a)")
```



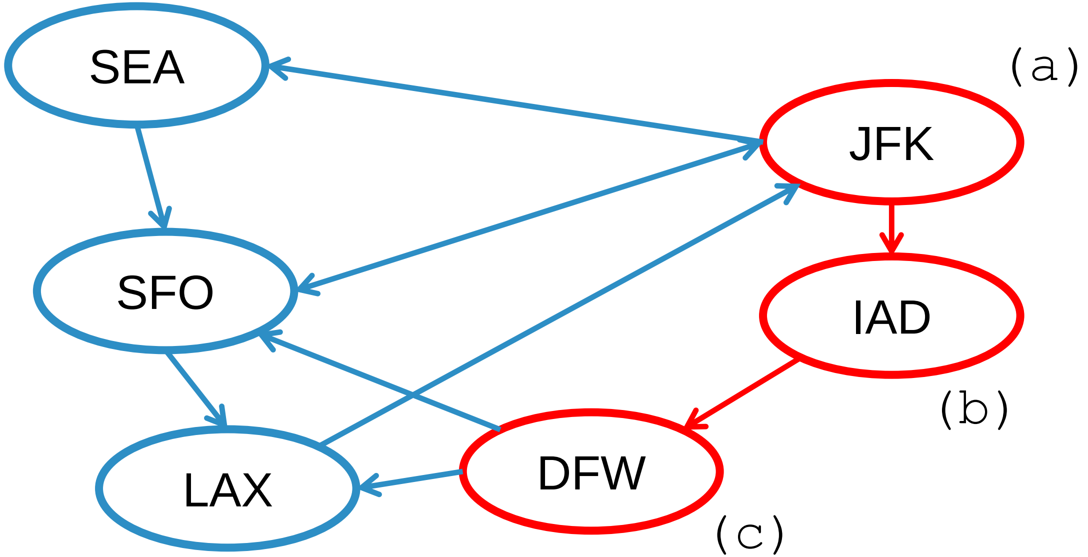
Motif finding

Search for structural patterns within a graph.

```
val paths: DataFrame =
  g.find("(a)-[e1]->(b);
          (b)-[e2]->(c);
         !(c)-[]->(a)")
```

Then filter using vertex & edge data.

```
paths.filter("e1.delay > 20")
```



Graph algorithms

Find important vertices

- PageRank

Find paths between sets of vertices

- Breadth-first search (BFS)
- Shortest paths

Find groups of vertices (components, communities)

- Connected components
- Strongly connected components
- Label Propagation Algorithm (LPA)

Other

- Triangle counting
- SVDPlusPlus

Saving & loading graphs

Save & load the DataFrames.

```
vertices = sqlContext.read.parquet(...)
edges = sqlContext.read.parquet(...)
g = GraphFrame(vertices, edges)

g.vertices.write.parquet(...)
g.edges.write.parquet(...)
```