Web-Scale Graph Analytics with Apache Spark

Joseph K Bradley

Bay Area Apache Spark Meetup
September 7, 2017

About me

Software engineer at Databricks

Apache Spark committer & PMC member

Ph.D. Carnegie Mellon in Machine Learning

About Databricks

TEAM

Started Spark project (now Apache Spark) at UC Berkeley in 2009

MISSION

Making Big Data Simple

PRODUCT

Unified Analytics Platform

Try Apache Spark in Databricks!

UNIFIED ANALYTICS PLATFORM

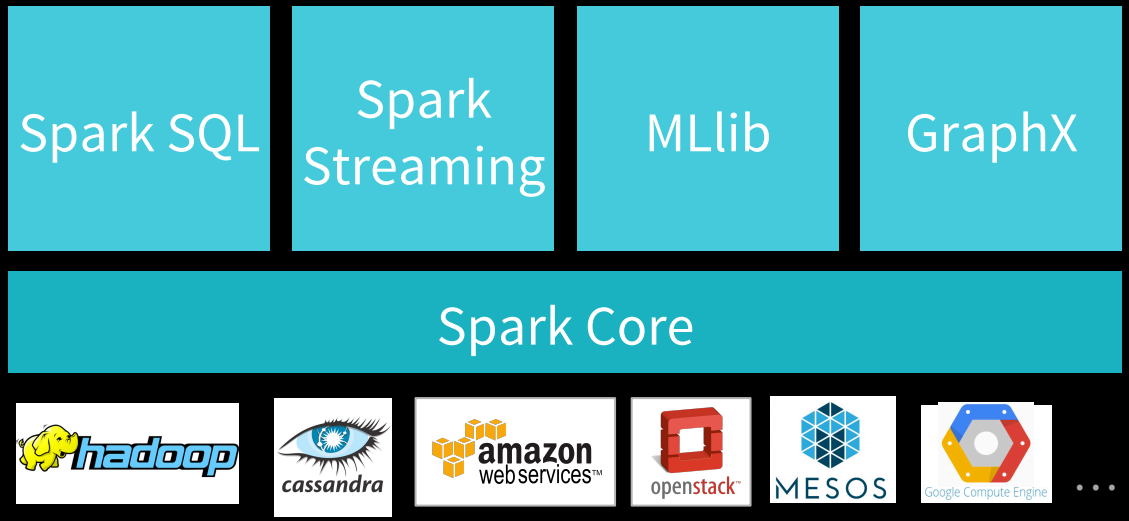
- Collaborative cloud environment
- Free version (community edition)

DATABRICKS RUNTIME 3.2

- Apache Spark - optimized for the cloud
- Caching and optimization layer - DBIO
- Enterprise security - DBES

Try for free today.
databricks.com

Apache Spark Engine



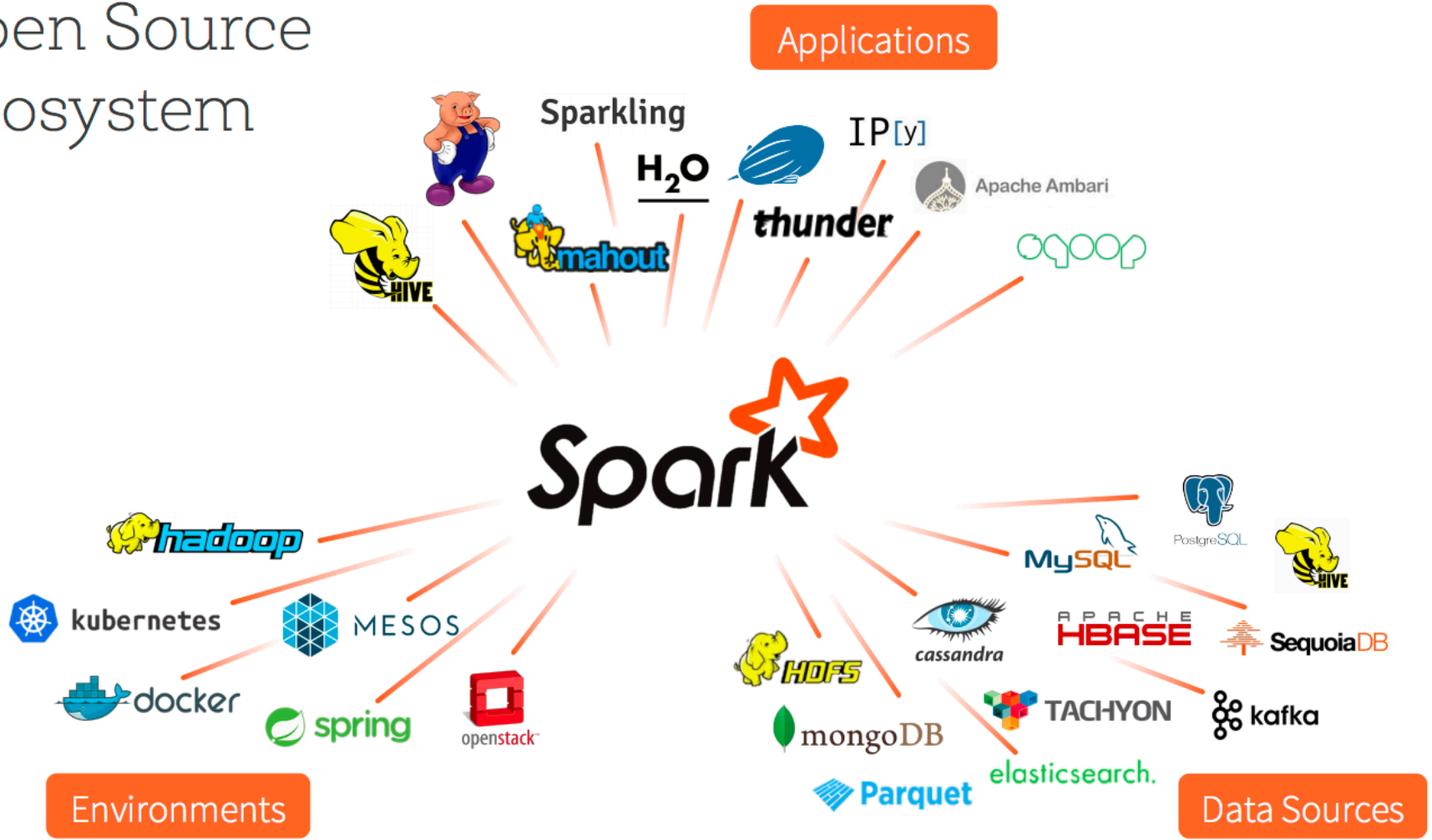
Scale out, fault tolerant

Python, Java, Scala, & R APIs

Standard libraries

Unified engine across diverse workloads & environments

Open Source Ecosystem



Spark Packages

spark-packages.org

340+ packages written for Spark

80+ packages for ML and Graphs

E.g.:

- **GraphFrames: DataFrame-based graphs**
- Bisecting K-Means: now part of MLlib
- Stanford CoreNLP integration: UDFs for NLP

Outline

Intro to GraphFrames

Moving implementations to DataFrames

- Vertex indexing
- Scaling Connected Components
- Other challenges: skewed joins and checkpoints

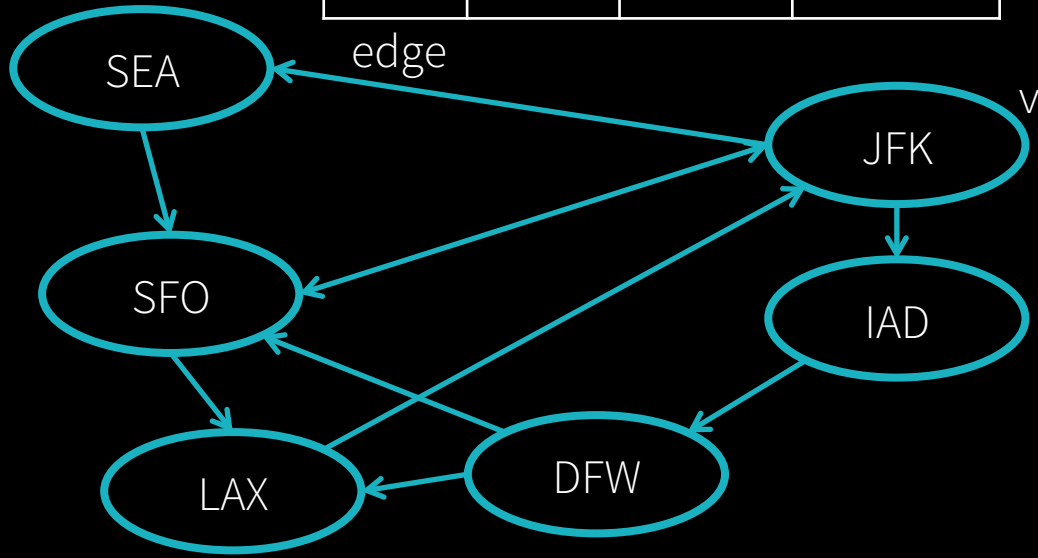
Future of GraphFrames

Graphs

Example: airports & flights between them

src	dst	delay	tripID
“JFK”	“SEA”	45	1058923

edge



vertex

id	City	State
“JFK”	“New York”	NY

Apache Spark's GraphX library

Overview

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

Challenges

- 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

Goal: DataFrame-based graphs on Apache Spark

- Simplify interactive queries
- Support motif-finding for structural pattern search
- Benefit from DataFrame optimizations

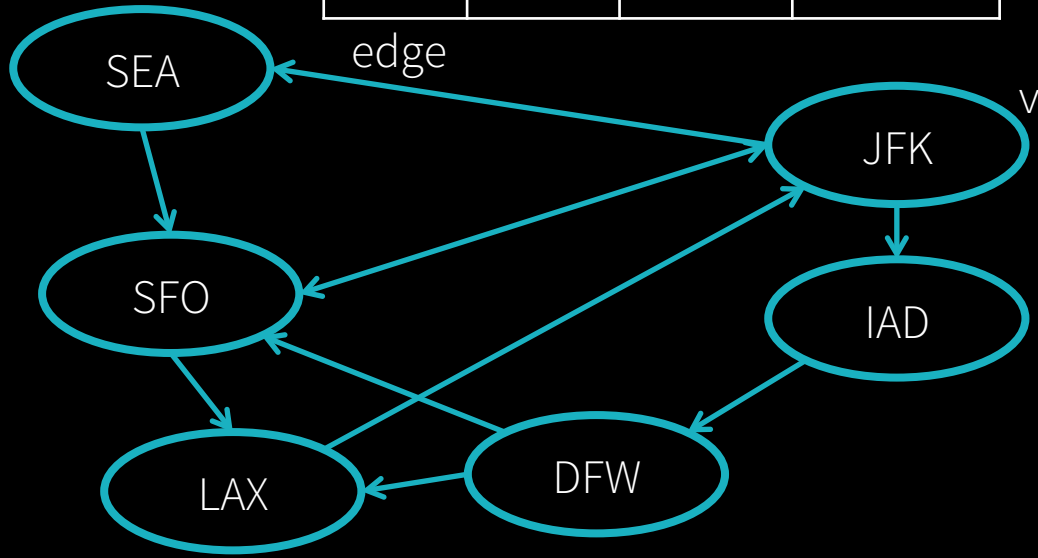
Collaboration between Databricks, UC Berkeley & MIT

+ Now with community contributors & committers!

Graphs

src	dst	delay	tripID
“JFK”	“SEA”	45	1058923

edge



vertex

id	City	State
“JFK”	“New York”	NY

GraphFrames

`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

Graph analysis with GraphFrames

Simple queries

Motif finding

Graph algorithms

Simple queries

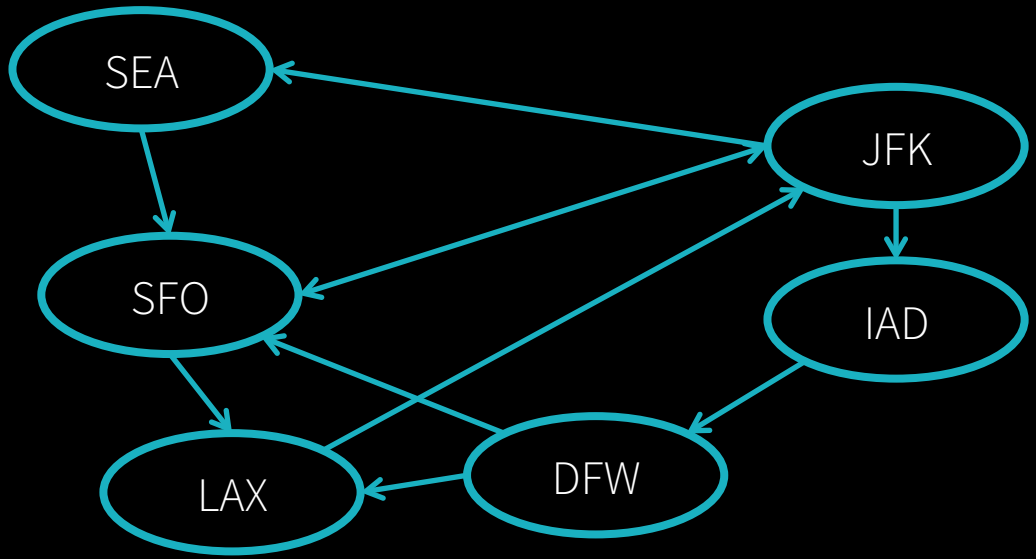
SQL queries on vertices & edges

Simple graph queries (e.g., vertex degrees)

Motif finding

Search for structural patterns within a graph.

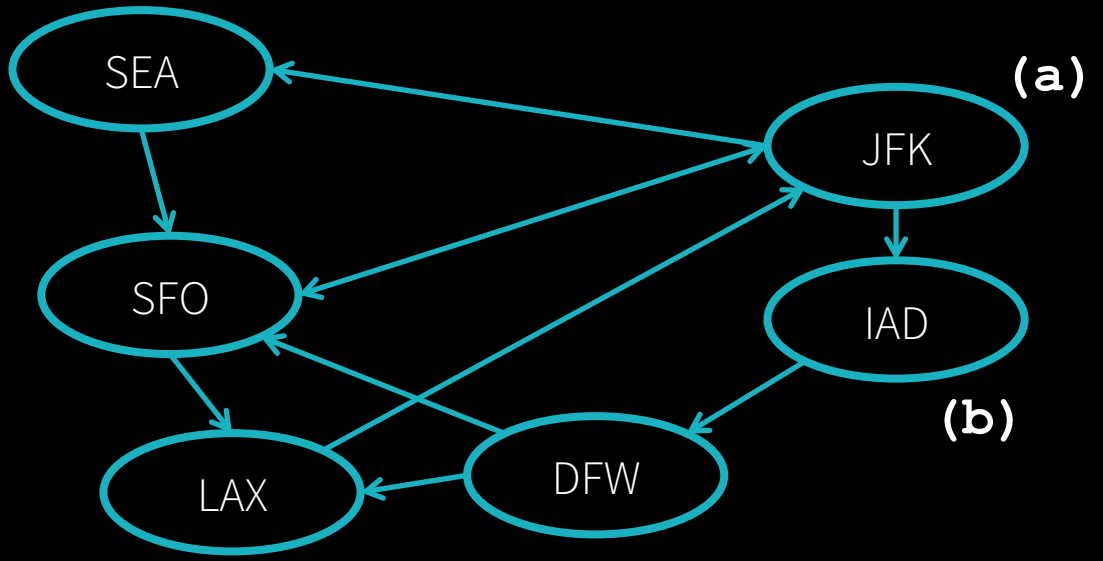
```
val paths: DataFrame =
  g.find("(a)-[e1]->(b);
          (b)-[e2]->(c);
         !(c)-[]->(a)")
```



Motif finding

Search for structural patterns within a graph.

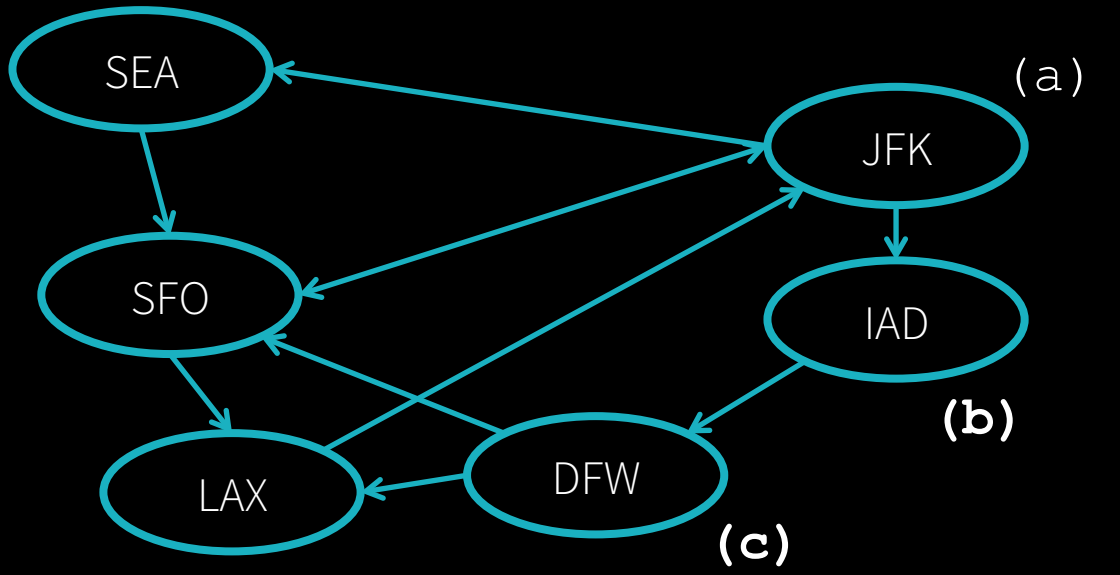
```
val paths: DataFrame =
  g.find("(a)-[e1]->(b);
          (b)-[e2]->(c);
         !(c)-[]->(a)")
```



Motif finding

Search for structural patterns within a graph.

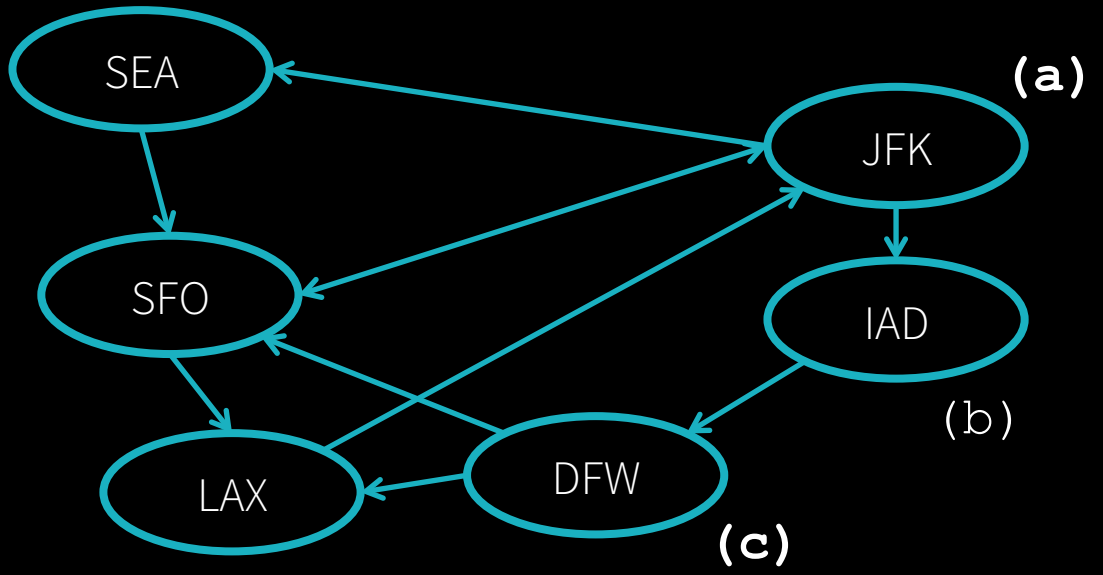
```
val paths: DataFrame =
  g.find("(a)-[e1]->(b);
          (b)-[e2]->(c);
         !(c)-[]->(a)")
```



Motif finding

Search for structural patterns within a graph.

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



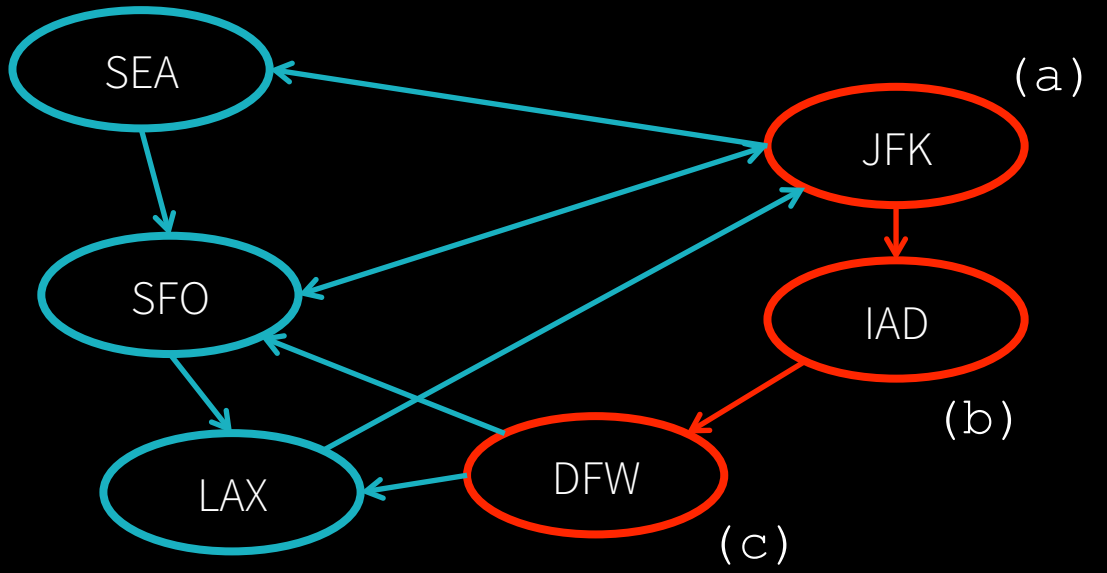
Motif finding

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(...)
```

GraphFrames vs. GraphX

	GraphFrames	**GraphX**
Built on	DataFrames	RDDs
Languages	Scala, Java, Python	Scala
Use cases	Queries & algorithms	Algorithms
Vertex IDs	Any type (in Catalyst)	Long
Vertex/edge attributes	Any number of DataFrame columns	Any type (VD, ED)

2 types of graph libraries

Graph algorithms

Standard & custom algorithms
Optimized for batch processing

Graph queries

Motif finding
Point queries & updates

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

Outline

Intro to GraphFrames

Moving implementations to DataFrames

- Vertex indexing
- Scaling Connected Components
- Other challenges: skewed joins and checkpoints

Future of GraphFrames

Algorithm implementations

Mostly wrappers for GraphX

- PageRank
- Shortest paths
- Strongly connected components
- Label Propagation Algorithm (LPA)
- SVDPlusPlus

Some algorithms implemented using DataFrames

- Breadth-first search
- Connected components
- Triangle counting
- Motif finding

Moving implementations to DataFrames

DataFrames are optimized for a huge number of small records.

- columnar storage
- code generation (“Project Tungsten”)
- query optimization (“Project Catalyst”)

Outline

Intro to GraphFrames

Moving implementations to DataFrames

- **Vertex indexing**
- Scaling Connected Components
- Other challenges: skewed joins and checkpoints

Future of GraphFrames

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 - (k-1)/N * (k-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
Tim	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

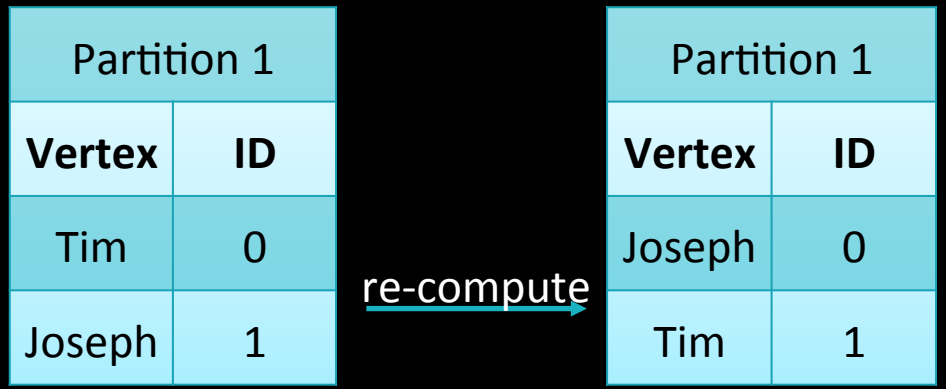
Partition 1	
Vertex	**ID**
Tim	0
Joseph	1

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

Partition 3	
Vertex	**ID**
…	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.



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

Outline

Intro to GraphFrames

Moving implementations to DataFrames

- Vertex indexing
- **Scaling Connected Components**
- Other challenges: skewed joins and checkpoints

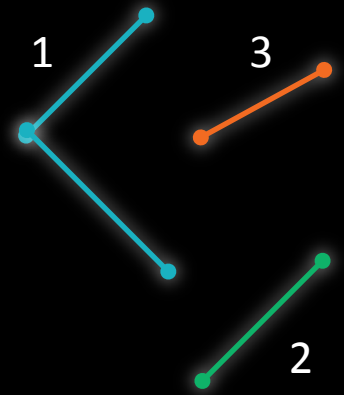
Future of GraphFrames

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

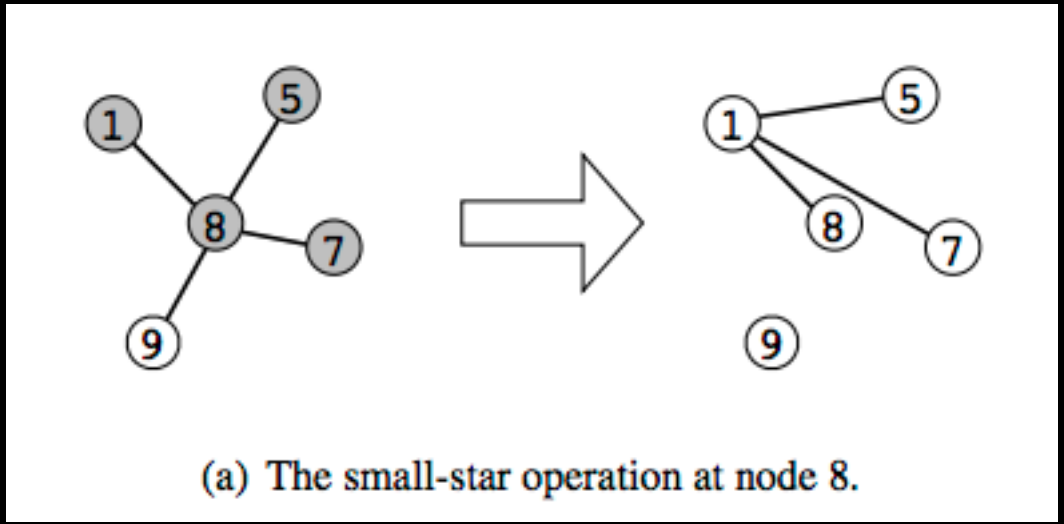
Con: slow convergence on large-diameter graphs

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)

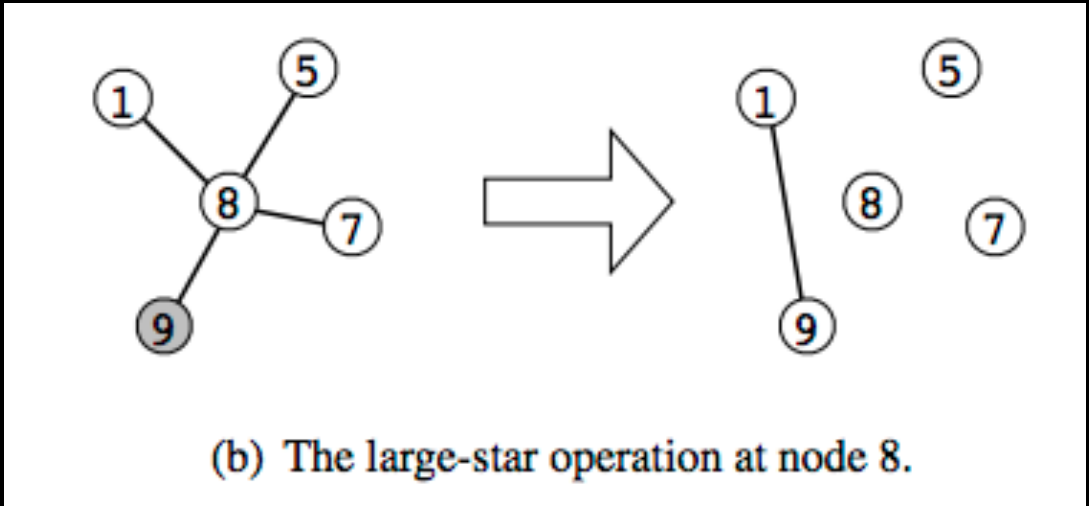
Small-star operation



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

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

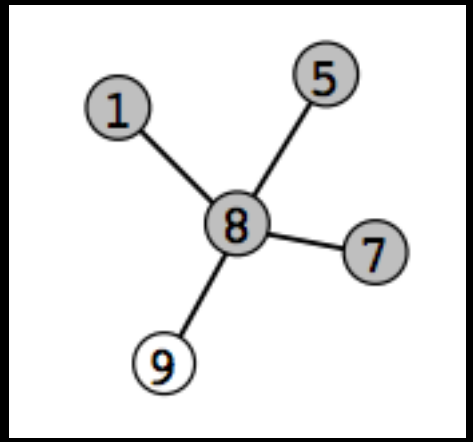
Big-star operation



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

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

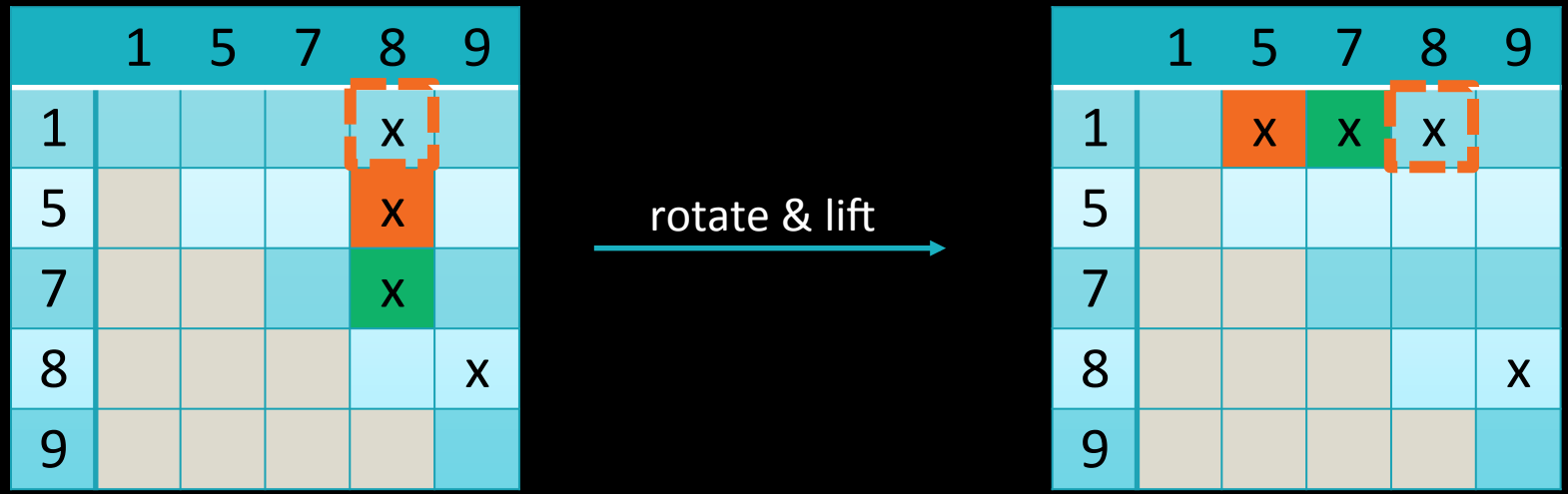
Another interpretation



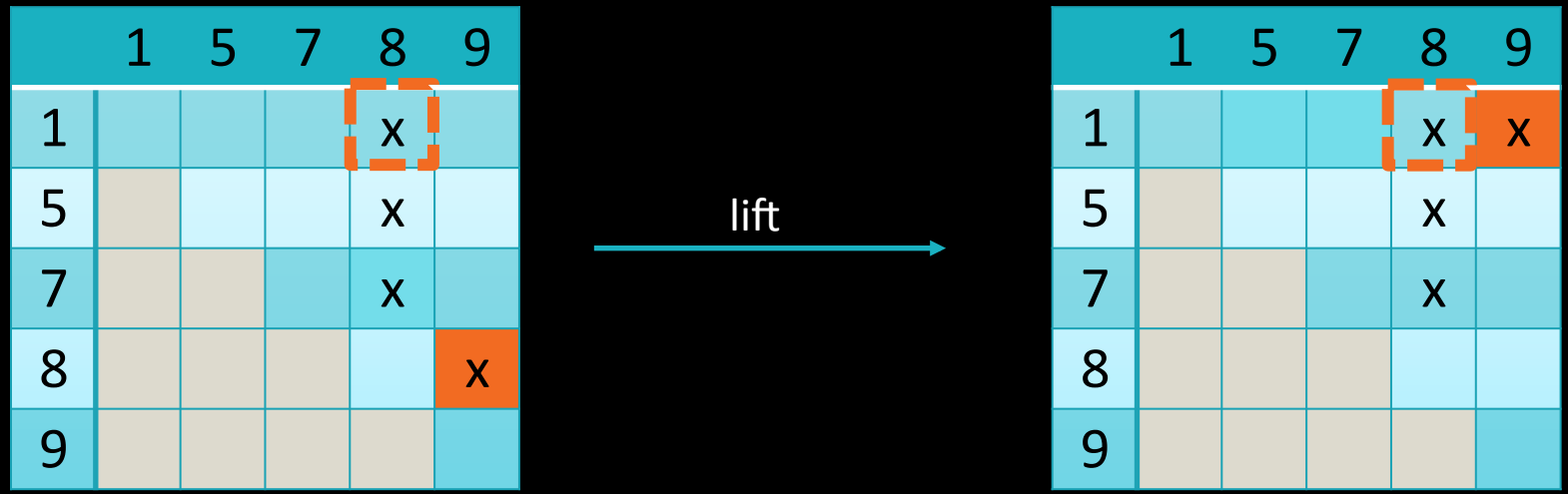
adjacency matrix

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

Small-star operation



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.

Outline

Intro to GraphFrames

Moving implementations to DataFrames

- Vertex indexing
- Scaling Connected Components
- **Other challenges: skewed joins and checkpoints**

Future of GraphFrames

Skewed joins

Real-world graphs contain big components.

→ 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

Experiments

regular grid 50,000 x 50,000 (large diameter)

- 2.5 billion nodes, 10 billion edges

32 r3.8xlarge workers on Databricks

- GraphX: failed
- GraphFrames: 1.6 hours

Future improvements

GraphFrames

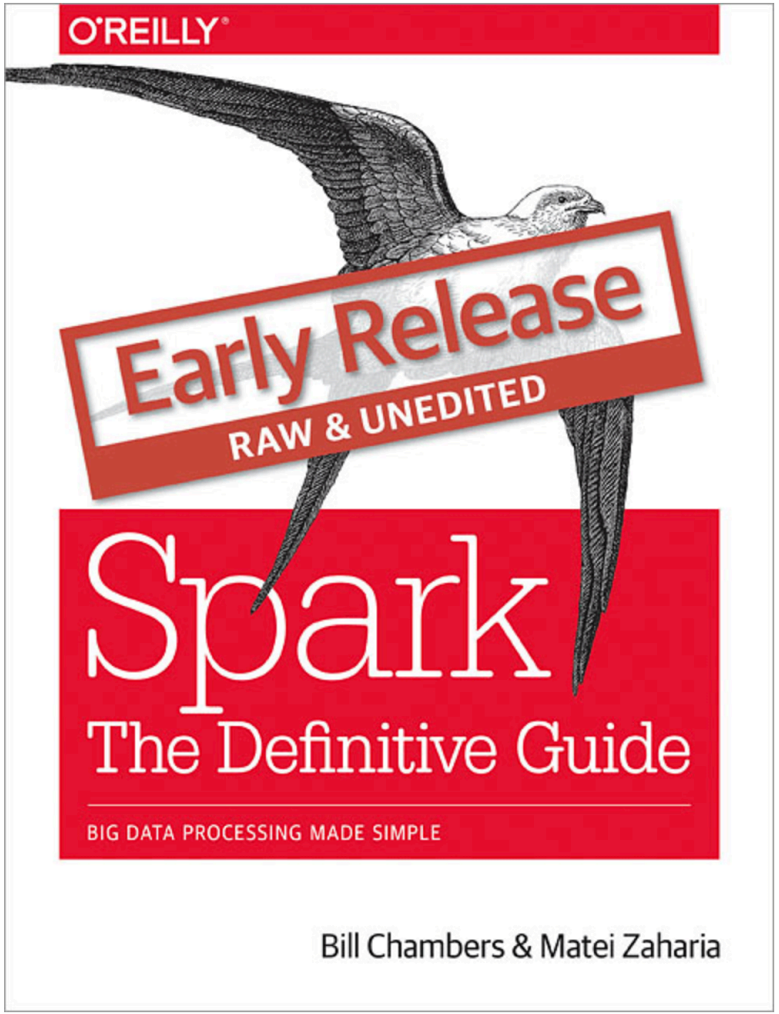
- update inefficient code (due to Spark 1.6 compatibility)
- better graph partitioning
- letting Spark SQL handle skewed joins and iterations
- graph compression

Connected Components

- local iterations
- node pruning and better stopping criteria

https://spark-summit.org/eu-2017/

15% Discount code: **Databricks**



source: O'Reilly

Sharing Knowledge with the Community in a Preview of Apache Spark: The Definitive Guide

by Bill Chambers and Matei Zaharia
Posted in **COMPANY BLOG** | June 5, 2017

http://dbricks.co/2sK35XT

databricks

https://databricks.com/company/careers

databricks

Get started with GraphFrames

- Docs, downloads & tutorials
  - http://graphframes.github.io
  - https://docs.databricks.com
- Dev community
  - Github issues & PRs

Thank you!

Twitter: @jkbatcmu → I'll share my slides.

databricks