Challenging Web-Scale Graph Analytics with Apache Spark

Xiangrui Meng

Spark Summit 2017

databricks

About me

- Software Engineer at Databricks
  - machine learning and data science/engineering
- Committer and PMC member of Apache Spark
  - MLlib, SparkR, PySpark, Spark Packages, etc

GraphFrames

GraphFrames

- A Spark package introduced in 2016 (graphframes.github.io)
  - collaboration between Databricks, UC Berkeley, and MIT
- GraphX to RDDs as GraphFrames are to DataFrames
  - Python, Java, and Scala APIs,
  - expressive graph queries,
  - query plan optimizers from Spark SQL,
  - graph algorithms.

Quick examples

Launch a Spark shell with GraphFrames:

```
spark-shell --packages graphframes:graphframes:0.5.0-spark2.1-s_2.11
```

Or [try it on Databricks Community Edition (databricks.com/try)](databricks.com/try).

Quick examples

Find 2nd-degree followers:

```
g.find("(A)-[]->(B); (B)-[]->(C); !(A)-[]->(C)")
  .filter("A.id != C.id")
  .select("A", "C")
```

Quick examples

Compute PageRank:

```
g.pageRank(resetProbability=0.15, maxIter=20)
```

Supported graph algorithms

- breath-first search (BFS)
- connected components
  - strongly connected components
- label propagation algorithm (LPA)
- PageRank and personalized PageRank
- shortest paths
- triangle count

Moving implementations to DataFrames

- Several algorithms in GraphFrames are simple wrappers over GraphX RDD implementations, which do not scale very well.
- DataFrames are optimized for a huge number of small records.
  - columnar storage
  - code generation
  - query optimization

Assigning integral vertex IDs

… lessons learned

Pros of having integral vertex IDs

GraphFrames take string vertex identifiers, whose values are not used in graph algorithms. Having integral vertex IDs can help

- optimize in-memory storage,
- save communication.

So the task is to map unique vertex identifiers to unique (long) integers.

The hashing trick?

- It is easy to hash the vertex identifier to a 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%
- And collisions change graph topology.

Name	Hash
Tim	84088
Joseph	-2070372689
Xiangrui	264245405
Felix	67762524

Generating unique IDs

Spark has builtin methods to generate unique IDs.

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

So given a DataFrame of distinct vertex identifiers, we can add a new column with generated unique long IDs. Simple?

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 work

- DataFrames/RDDs are immutable and reproducible by design.
- However, records do not always have stable order.
  - distinct
  - repartition
- And cache doesn't help.

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

re-compute

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

Our implementation

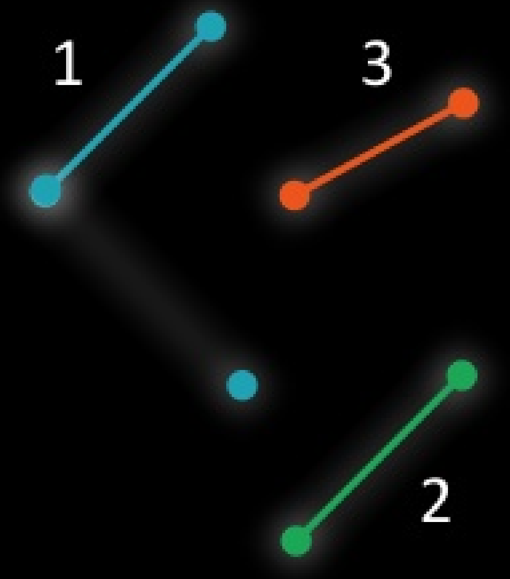
We implemented (v0.5.0) an expensive but correct version:

1. (hash) re-partition + distinct vertex identifiers,
2. sort vertex identifiers within each partition,
3. generate unique integral 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



A naive implementation

1. Assign each vertex a unique component ID.
2. Run in batches until convergence:
   - For each vertex v, update its component ID to the smallest component ID among its neighbors' and its own.

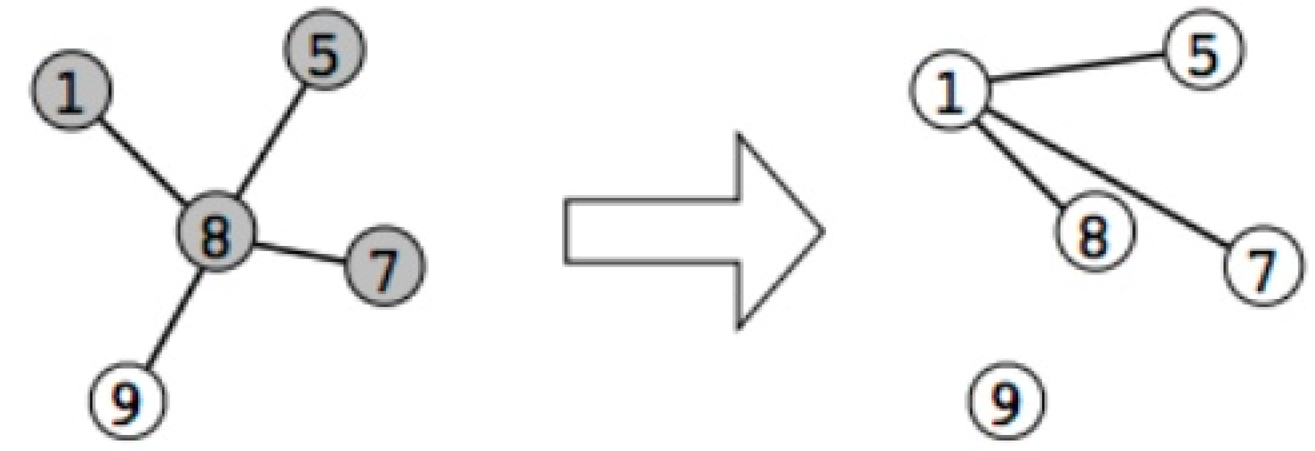
- easy to implement
- slow convergence on large-diameter graphs

Small-/large-star algorithm [Kiveris14]

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

1. Assign each vertex a unique ID.
2. Alternatively update edges in batches until convergence:
    - (small-star) for each vertex, connect its smaller neighbors to the smallest neighbor vertex
    - (big-star) for each vertex, connect its bigger neighbors to the smallest neighbor vertex (or itself)

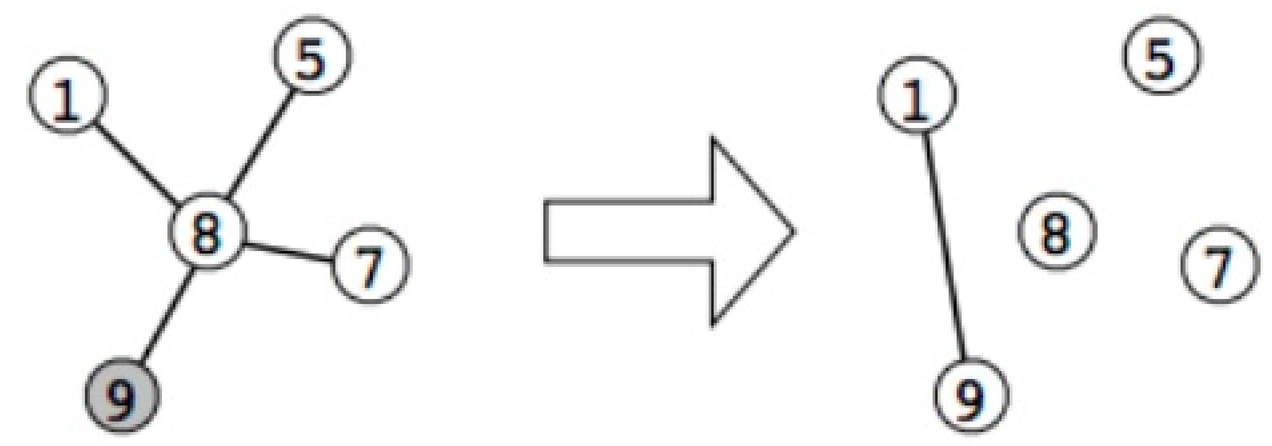
Small-star operation



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

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

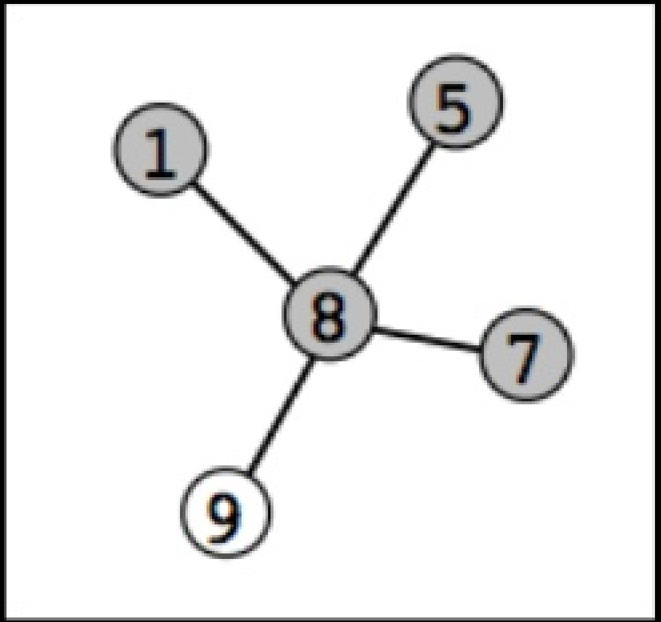
Big-star operation



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

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

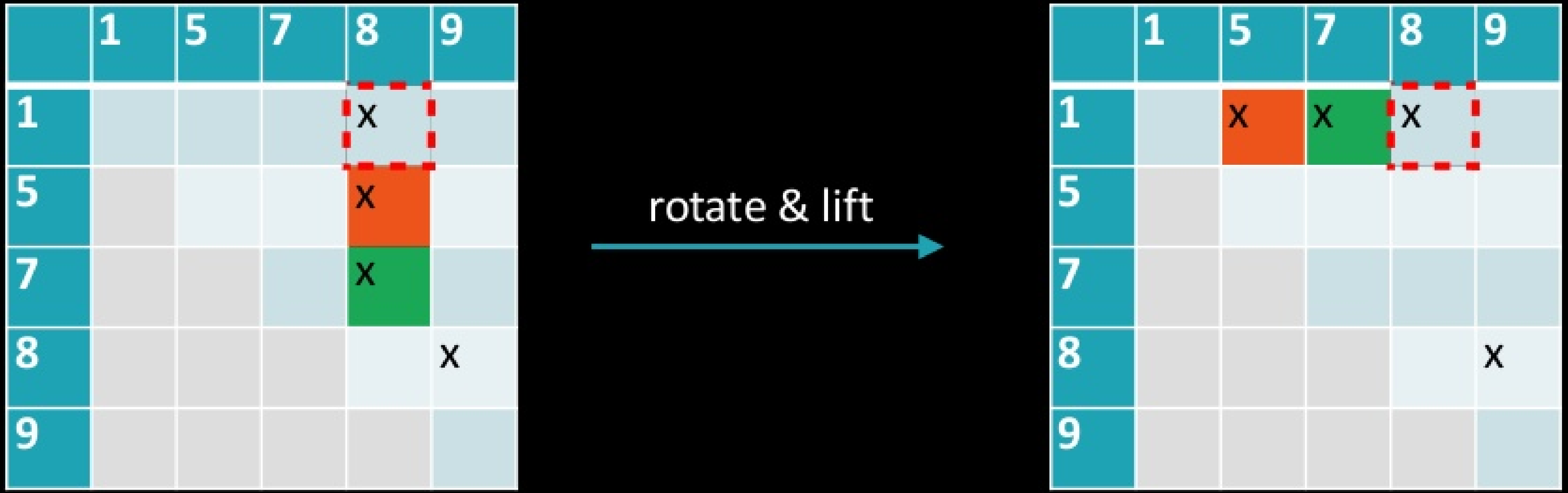
Another interpretation



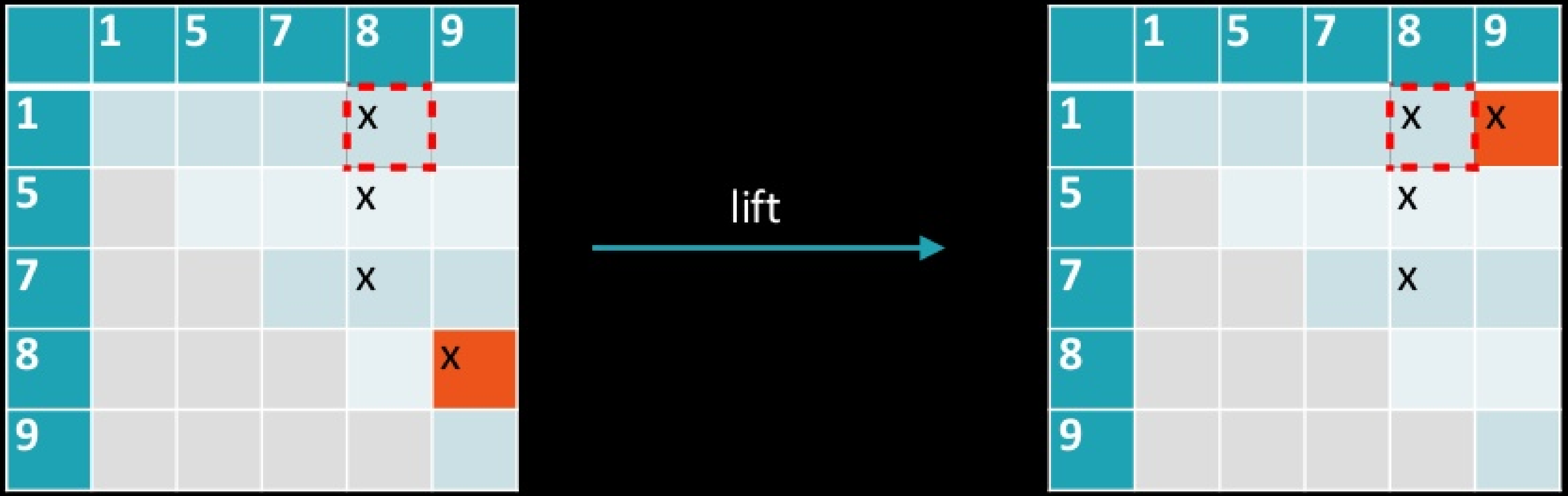
adjacency matrix

	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					

Small-/big-star algorithm

- Small-/big-star operations do not change graph connectivity.
- Extra edges are pruned during iterations.
- Each connected component converges to a star graph.

Kiveris et al. proved one variation of the algorithm converges in $\log^2(\#\text{nodes})$ iterations. We chose a variation that alternates small-/big-star operations in GraphFrames.

Implementation

Essentially the small-/big-star operations map to a sequence of filters and self joins with DataFrames. So we need to handle the following operations at scale:

- joins
- iterations

Skewed joins

A real-world graph usually contains big component, which leads to data skewness at connected components iterations.

src	id	nbrs
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	id	nbrs
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 do checkpoint at every 2 iterations to avoid:

- query plan getting too big (exponential growth)
- optimizer taking too long
- 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

- update inefficient code (due to Spark 1.6 compatibility)
- better graph partitioning
- local iterations
- node pruning and better stop criterion
- letting Spark SQL handle skewed joins and iterations
- graph compression
- prove log(N) iterations or maybe a better algorithm?

Thank You

- graphframes.github.io
- docs.databricks.com