Interactive Graph Analytics with Spark

A talk by Daniel Darabos about the design and implementation of the LynxKite analytics application

About LynxKite

Analytics web application

AngularJS + Play! Framework + Apache Spark

Each LynxKite “project” is a graph

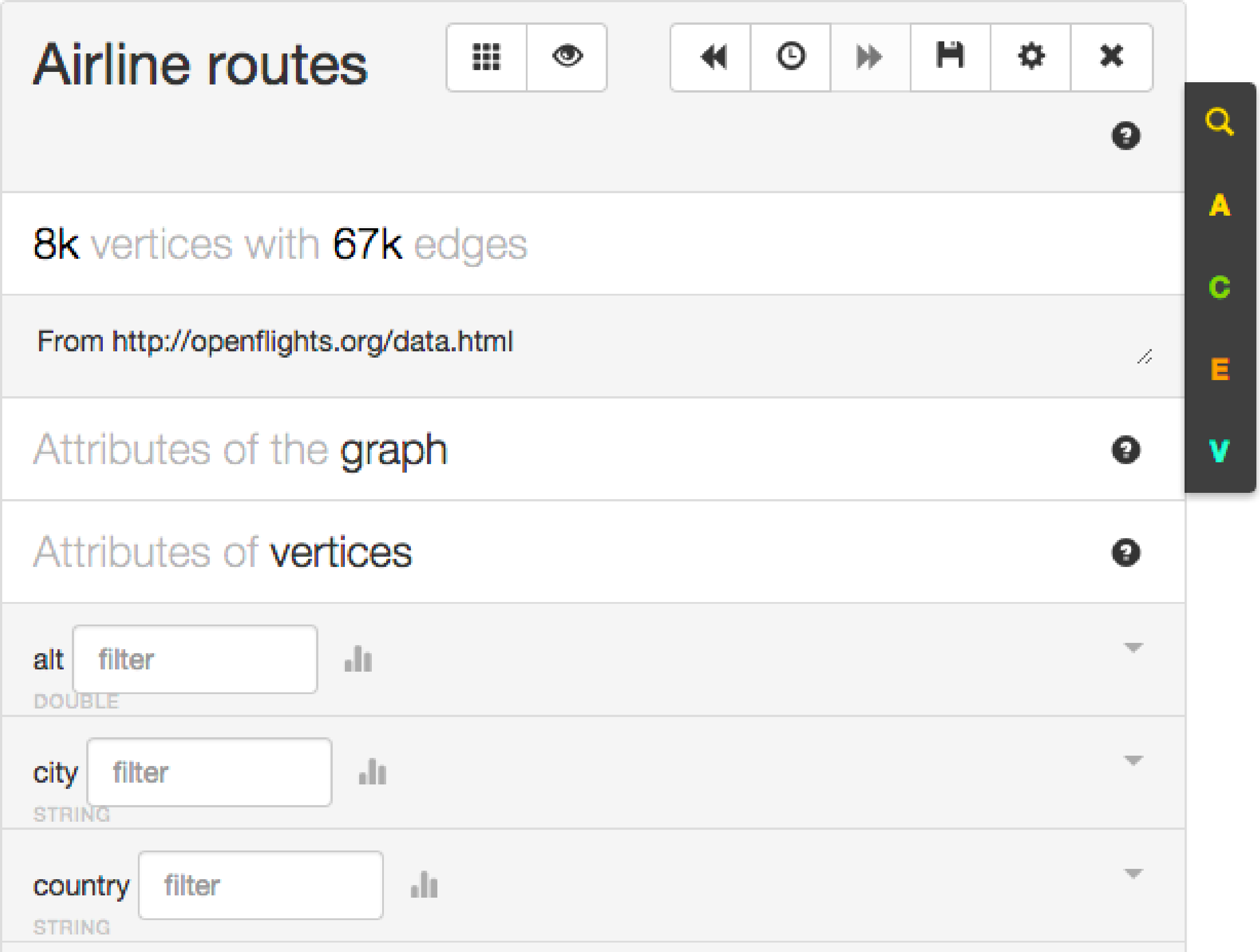
Graph operations mutate state

- Typical big data workload
- Minutes to hours

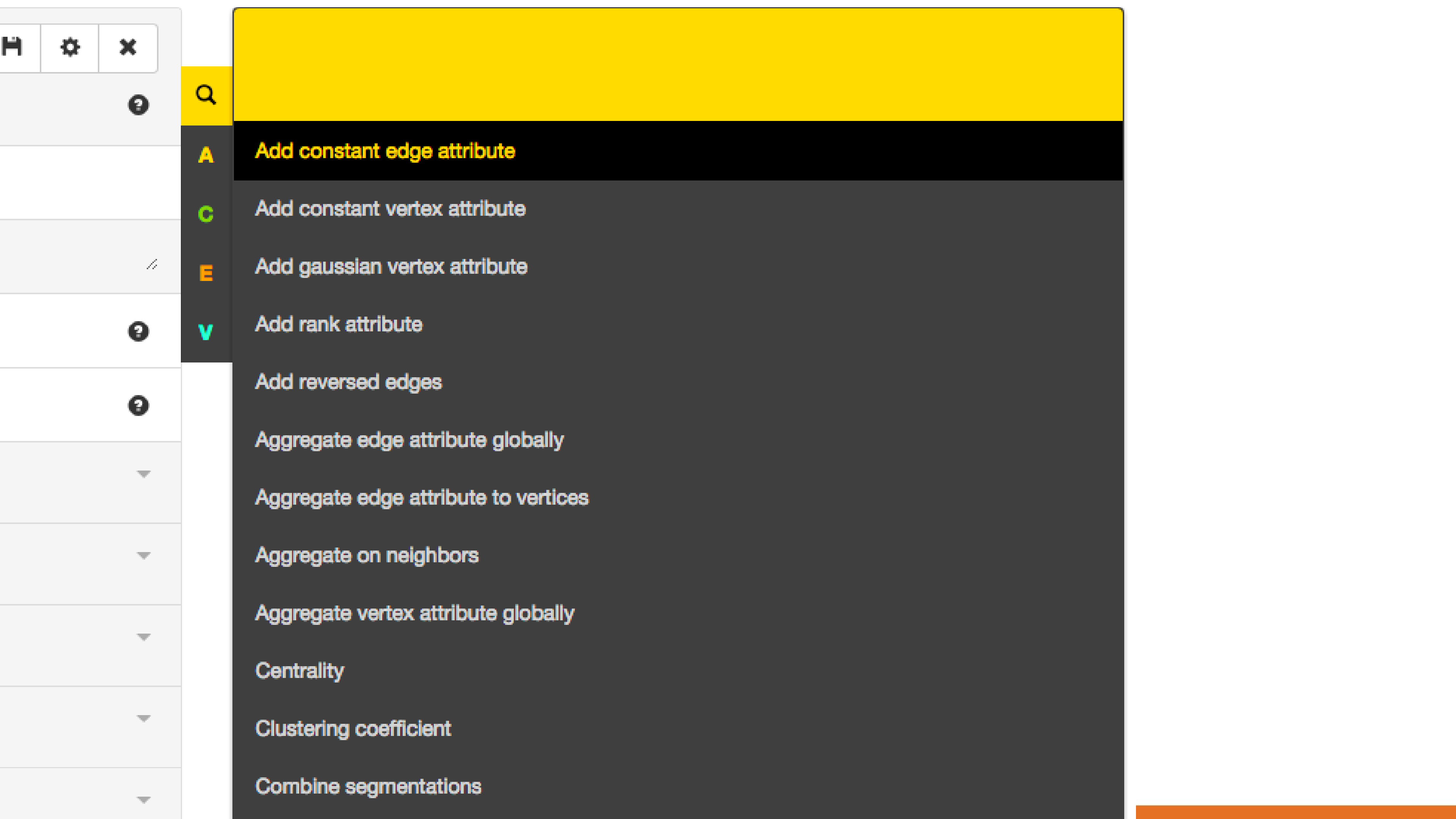
Visualizations

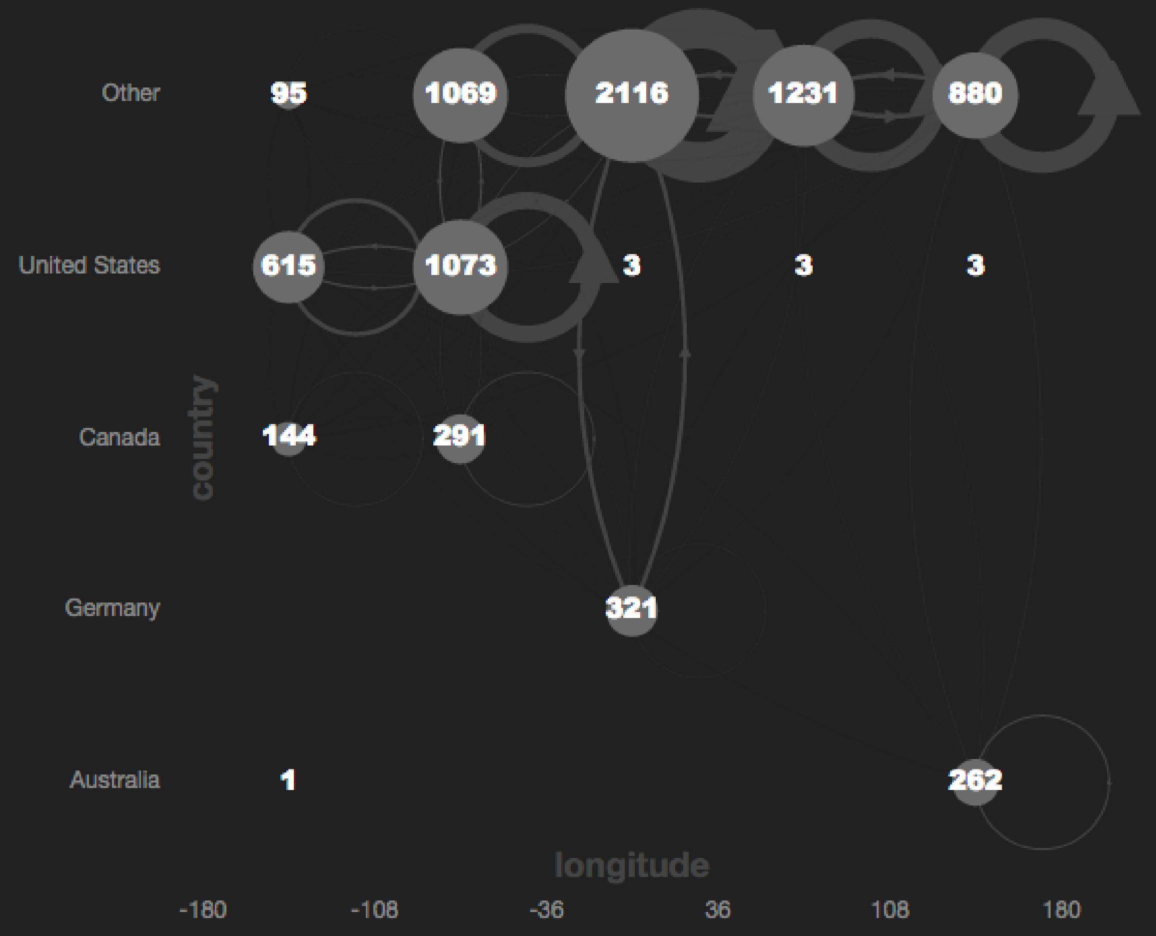
- Few seconds

The topic of this talk



Spark summit 2015


A
C
E
V
Add constant edge attribute
Add constant vertex attribute
Add gaussian vertex attribute
Add rank attribute
Add reversed edges
Aggregate edge attribute globally
Aggregate edge attribute to vertices
Aggregate on neighbors
Aggregate vertex attribute globally
Centrality
Clustering coefficient
Combine segmentations



Other
United States
Canada
Germany
Australia
country
95
1069
2116
1231
880
615
1073
3
3
3
144
291
321
1
262
longitude
-180
-108
-36
36
108
180

Idea 1:

Avoid processing unused attributes

Column-based attributes

```
type ID = Long
case class Edge(src: ID, dst: ID)
type VertexRDD = RDD[(ID, Unit)]
type EdgeRDD = RDD[(ID, Edge)]
type AttributeRDD[T] = RDD[(ID, T)]
// Vertex attribute or edge attribute?
// Could be either!
```

Column-based attributes

Can process just the attributes we need

Easy to add an attribute

Simple and flexible

- Edges between vertices of two different graphs
- Edges between edges of two different graphs

A lot of joining

Idea 2:

Make joins fast

Co-located loading

Join is faster for co-partitioned RDDs

- Spark only has to fetch one partition

Even faster for co-located RDDs

- The partition is already in the right place

When loading attributes we make a seemingly useless join that causes two RDDs to be co-located

Co-located loading

```
val attributeRDD =
    sc.loadObjectFile[(ID, T)](path)
```

Co-located loading

```
val rawRDD =
    sc.loadObjectFile[(ID, T)](path)

val attributeRDD =
    vertexRDD.join(rawRDD).mapValues(_._2)
```

Co-located loading

```
val rawRDD =
    sc.loadObjectFile[(ID, T)](path)

val attributeRDD =
    vertexRDD.join(rawRDD).mapValues(_._2)

attributeRDD.cache
```

Scheduler delay

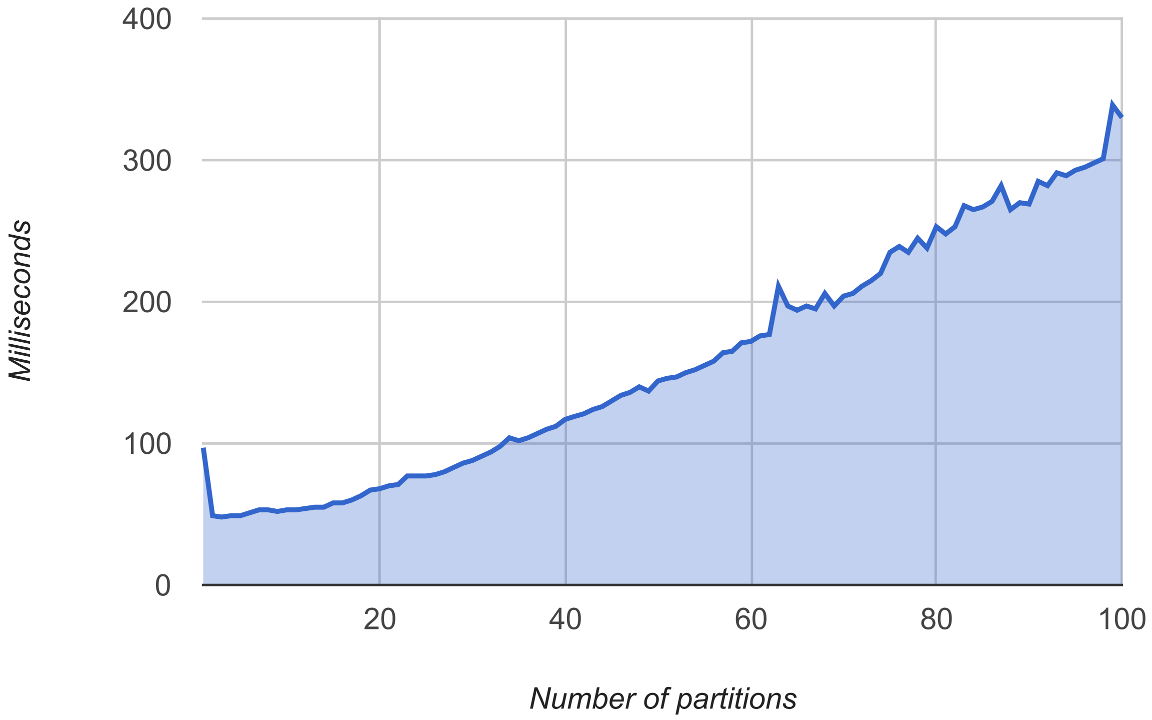
What's the ideal number of partitions for speed?

At least N partitions for N cores, otherwise some cores will be wasted.

But any more than that just wastes time on scheduling tasks.

Scheduler delay

sc.parallelize(1 to 1000, n).count



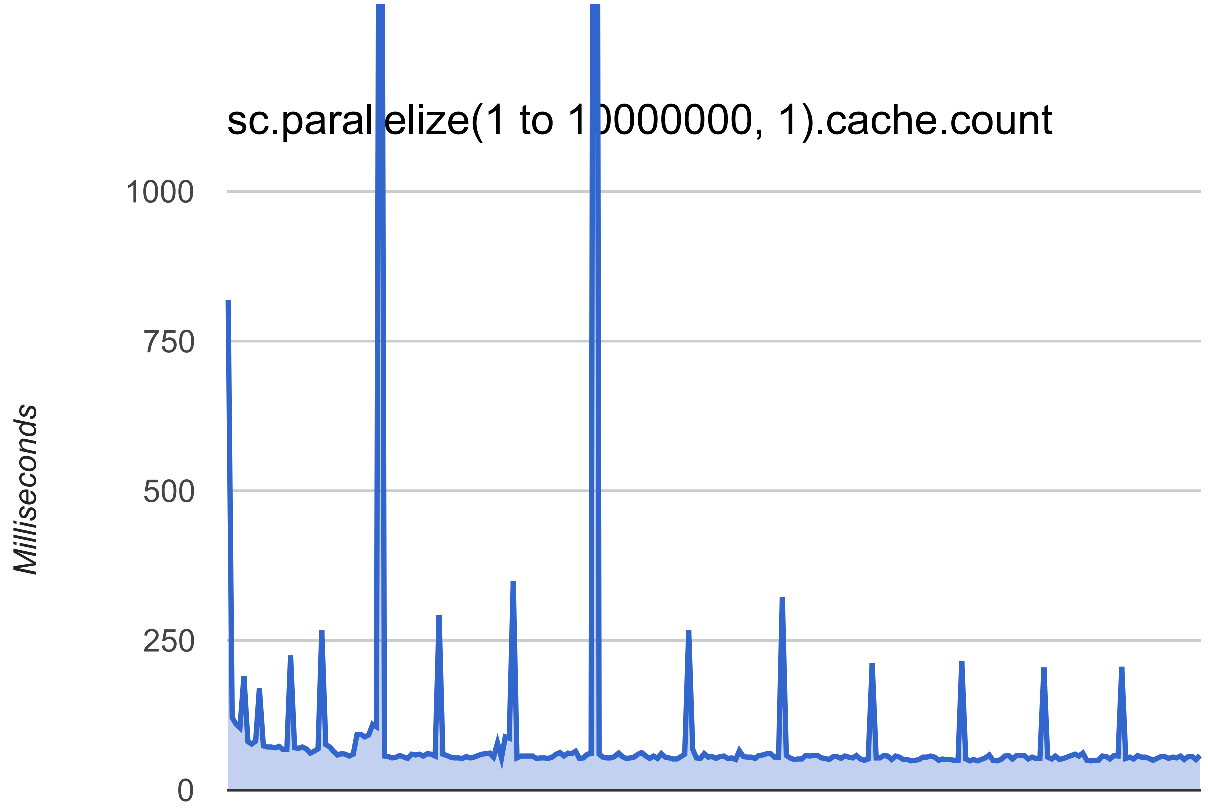
GC pauses

Can be tens of seconds on high-memory machines

Bad for interactive experience

Need to avoid creating big objects

GC pauses



sc.parallelize(1 to 10000000, 1).cache.count
Milliseconds
1000
750
500
250
0

Sorted RDDs

Speeds up the last step of the join

The insides of the partitions are kept sorted

Merging sorted sequences is fast

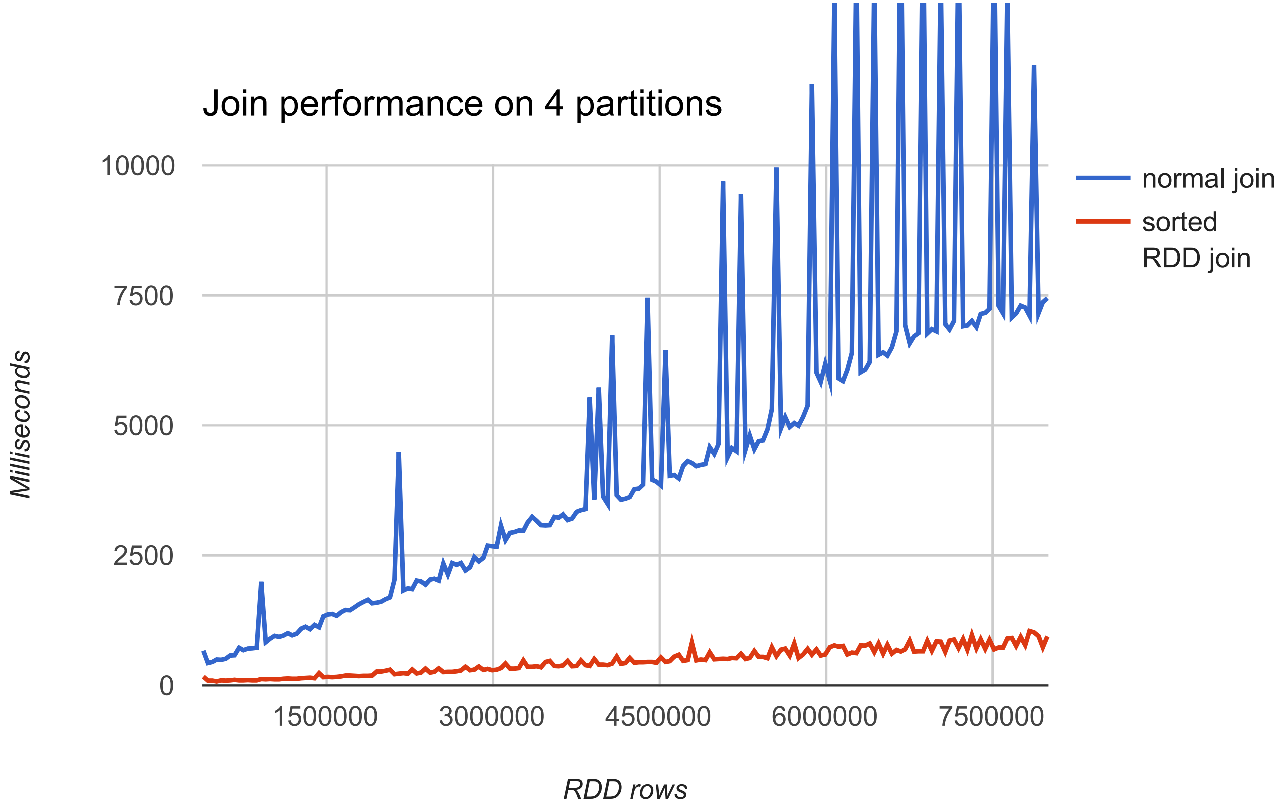
Doesn't require building a large hashmap

10 × speedup + GC benefit

- 2 × if cost of sorting is included
- sorting cost is amortized across many joins

Benefits other operations too (e.g. distinct)

Sorted RDDs


Join performance on 4 partitions
10000
7500
5000
2500
0
Milliseconds
1500000
3000000
4500000
6000000
7500000
RDD rows
normal join
sorted RDD join

Idea 3:

Do not read/compute all the data

Are all numbers positive?

```
def allPositive(rdd: RDD[Double]): Boolean =
  rdd.filter(_ > 0).count == rdd.count

// Terrible. It executes the RDD twice.
```

Are all numbers positive?

```
def allPositive(rdd: RDD[Double]): Boolean =
  rdd.filter(_ <= 0).count == 0


// A bit better,
// but it still executes the whole RDD.
```

Are all numbers positive?

```
def allPositive(rdd: RDD[Double]): Boolean =
  rdd.mapPartitions {
    p => Iterator(p.forall(_ > 0))
  }.collect.forall(_ == true)


// Each partition is only processed up to
// the first negative value.
```

Are all numbers positive?

```
def allPositive(rdd: RDD[Double]): Boolean =
  rdd.mapPartitions {
    p => Iterator(p.forall(_ > 0))
  }.collect.forall(identity)


// Each partition is only processed up to
// the first negative value.
```

Prefix sampling

Partitions are sorted by the randomly assigned ID

Taking the first N elements is an unbiased sample

Lazy evaluation means the rest are not even computed

Used for histograms and bucketed views

Idea 4:

Lookup instead of filtering for small key sets

Restricted ID sets

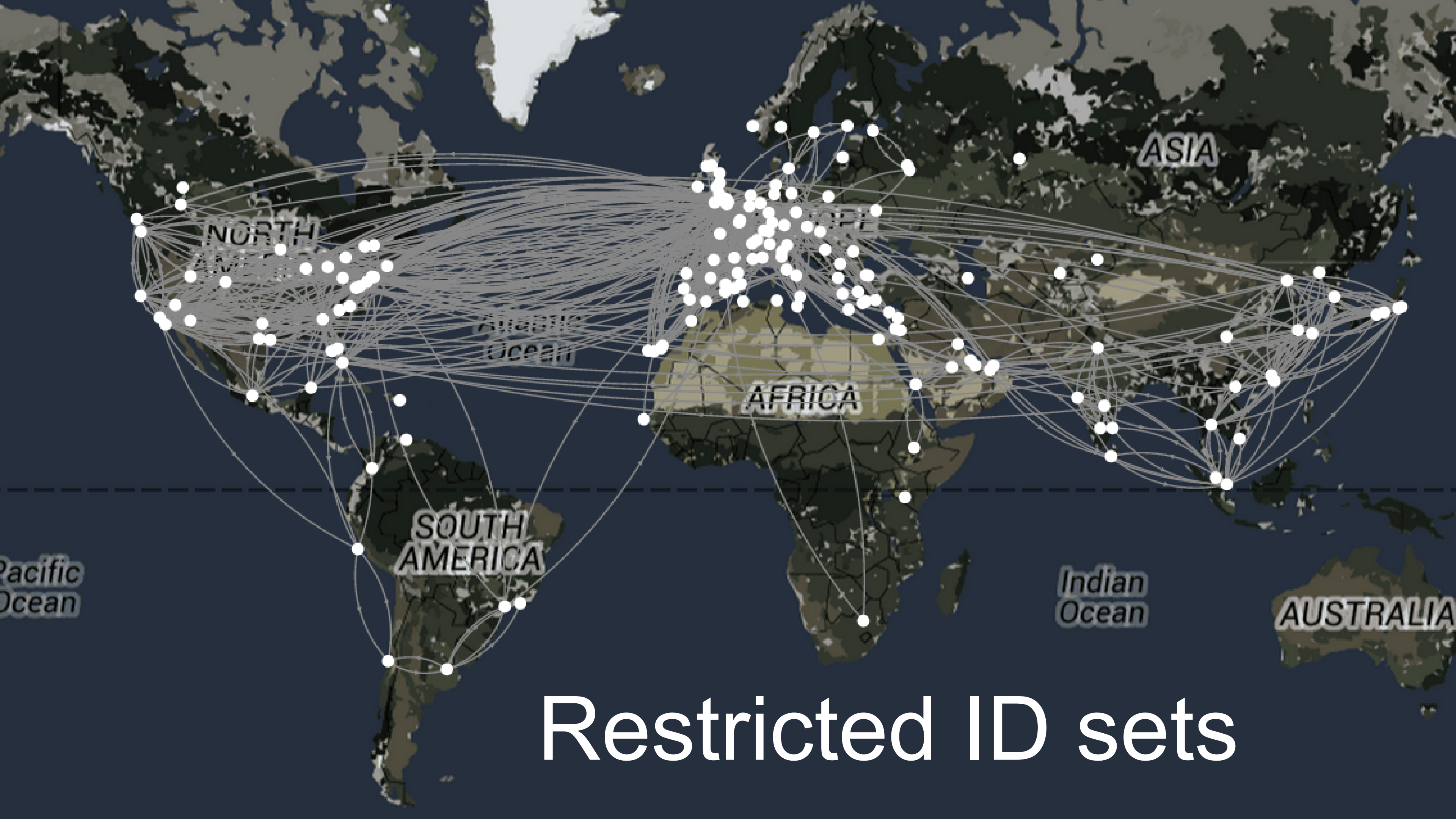
Cannot use sampling when showing 5 vertices

Hard to explain why showing 5 million is faster

Partitions are already sorted

We can use binary search to look up attributes

Put partitions into arrays for random access



Restricted ID sets

Summary

Column-oriented attributes

Small number of co-located, cached partitions

Sorted RDDs

Prefix sampling

Binary search-based lookup

Backup slides

Comparison with GraphX

Benchmarked connected components

Big data payload (not interactive)

Speed dominated by number of shuffle stages

Same number of shuffles ⇒ same speed

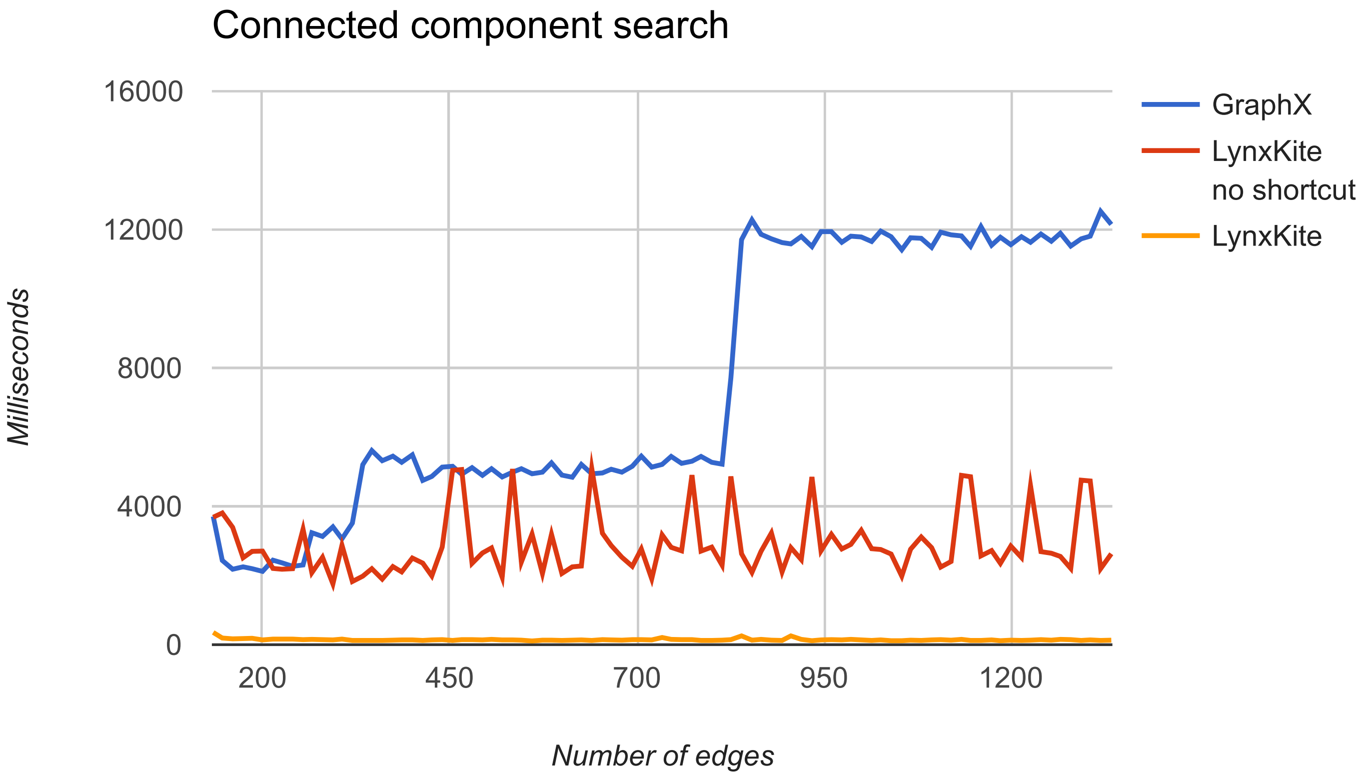
- Despite simpler data structures in LynxKite

Better algorithm in LynxKite ⇒ fewer shuffles

- From "A Model of Computation for MapReduce"

Benchmarked without short-circuit optimization

Comparison with GraphX



Comparison with GraphX