IEMS5730/ IERG4330/ ESTR4316
Spring 2022

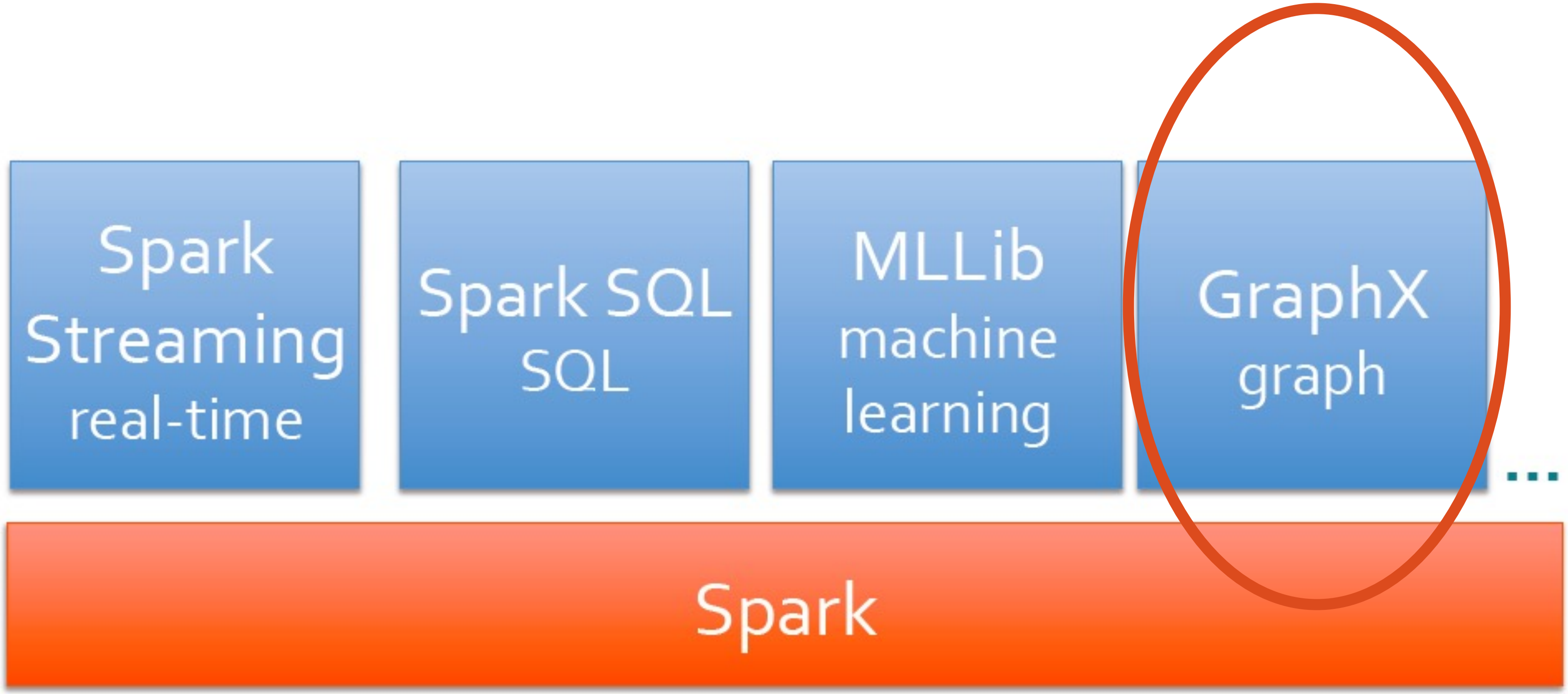
Big Graph Analytics

Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

Acknowledgements

- These slides are adapted from the following sources:
  - Matei Zaharia, “Spark 2.0,” Spark Summit East Keynote, Feb 2016.
  - Reynold Xin, “The Future of Real-Time in Spark,” Spark Summit East Keynote, Feb 2016.
  - Michael Armburst, “Structuring Spark: SQL, DataFrames, DataSets, and Streaming,” Spark Summit East Keynote, Feb 2016.
  - Ankur Dave, “GraphFrames: Graph Queries in Spark SQL,” Spark Summit East, Feb 2016.
  - Michael Armburst, “Spark DataFrames: Simple and Fast Analytics on Structured Data,” Spark Summit Amsterdam, Oct 2015.
  - Michael Armburst et al, “Spark SQL: Relational Data Processing in Spark,” SIGMOD 2015.
  - Michael Armburst, “Spark SQL Deep Dive,” Melbourne Spark Meetup, June 2015.
  - Reynold Xin, “Spark,” Stanford CS347 Guest Lecture, May 2015.
  - Joseph K. Bradley, “Apache Spark MLlib’s past trajectory and new directions,” Spark Summit Jun 2017.
  - Joseph K. Bradley, “Distributed ML in Apache Spark,” NYC Spark MeetUp, June 2016.
  - Ankur Dave, “GraphFrames: Graph Queries in Apache Spark SQL,” Spark Summit, June 2016.
  - Joseph K. Bradley, “GraphFrames: DataFrame-based graphs for Apache Spark,” NYC Spark MeetUp, April 2016.
  - Joseph K. Bradley, “Practical Machine Learning Pipelines with MLlib,” Spark Summit East, March 2015.
  - Joseph K. Bradley, “Spark DataFrames and ML Pipelines,” MLconf Seattle, May 2015.
  - Ameet Talwalkar, “MLlib: Spark’s Machine Learning Library,” AMPCamps 5, Nov. 2014.
  - Shivaram Venkataraman, Zongheng Yang, “SparkR: Enabling Interactive Data Science at Scale,” AMPCamps 5, Nov. 2014.
  - Tathagata Das, “Spark Streaming: Large-scale near-real-time stream processing,” O’Reilly Strata Conference, 2013.
  - Joseph Gonzalez et al, “GraphX: Graph Analytics on Spark,” AMPCAMP 3, 2013.
  - Jules Damji, “Jumpstart on Apache Spark 2.X with Databricks,” Spark Sat. Meetup Workshop, Jul 2017.
  - Sameer Agarwal, “What’s new in Apache Spark 2.3,” Spark+AI Summit, June 2018.
  - Reynold Xin, Spark+AI Summit Europe, 2018.
  - Hyukjin Kwon of Hortonworks, “What’s New in Spark 2.3 and Spark 2.4,” Oct 2018.
  - Matel Zaharia, “MLflow: Accelerating the End-to-End ML Lifecycle,” Nov. 2018.
  - Jules Damji, “MLflow: Platform for Complete Machine Learning Lifecycle,” PyData, Jan 2019.
-

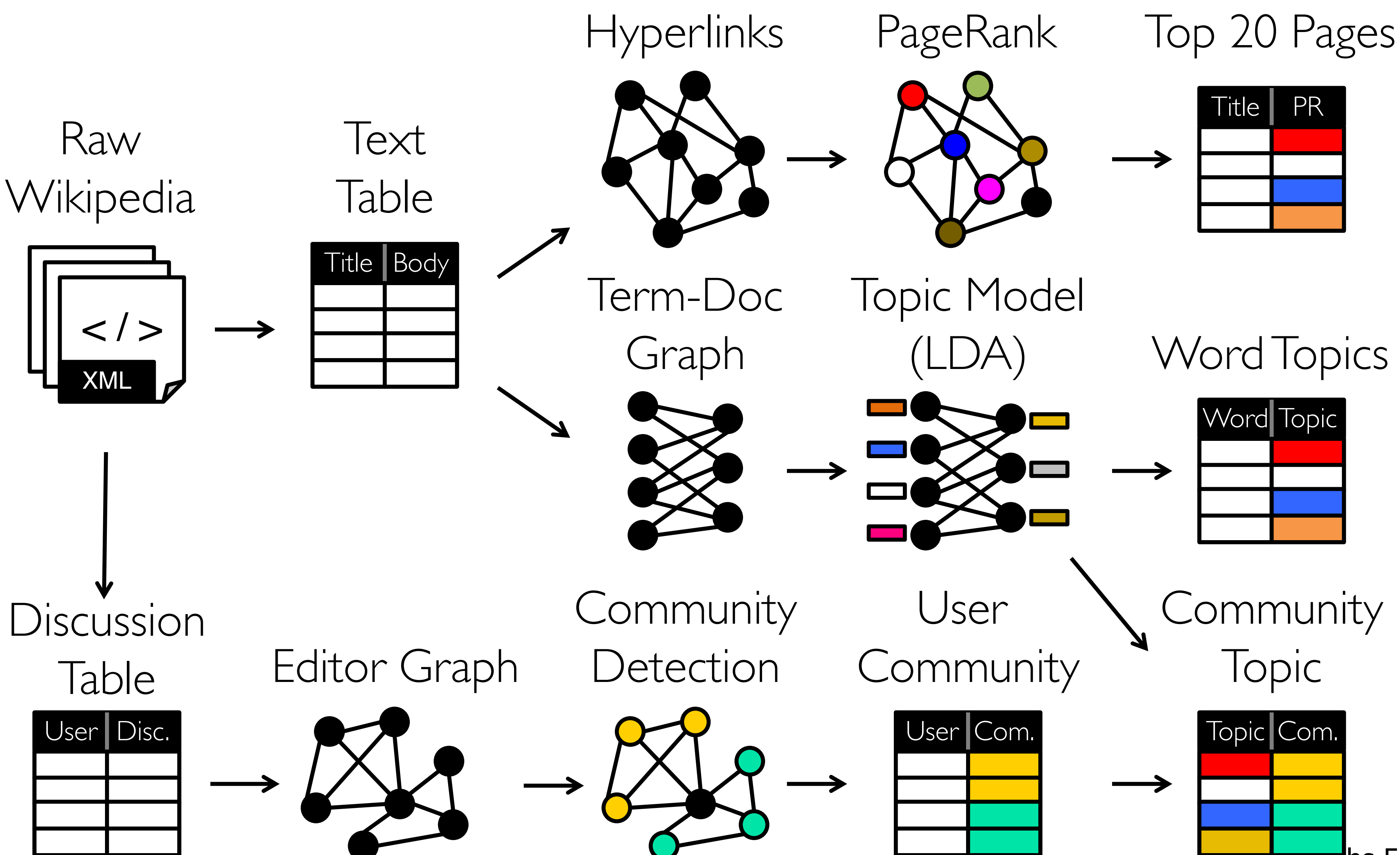
Major Modules in Spark



GraphX:

Unifying Data-Parallel and Graph-Parallel Analytics

Graphs are Central to Analytics

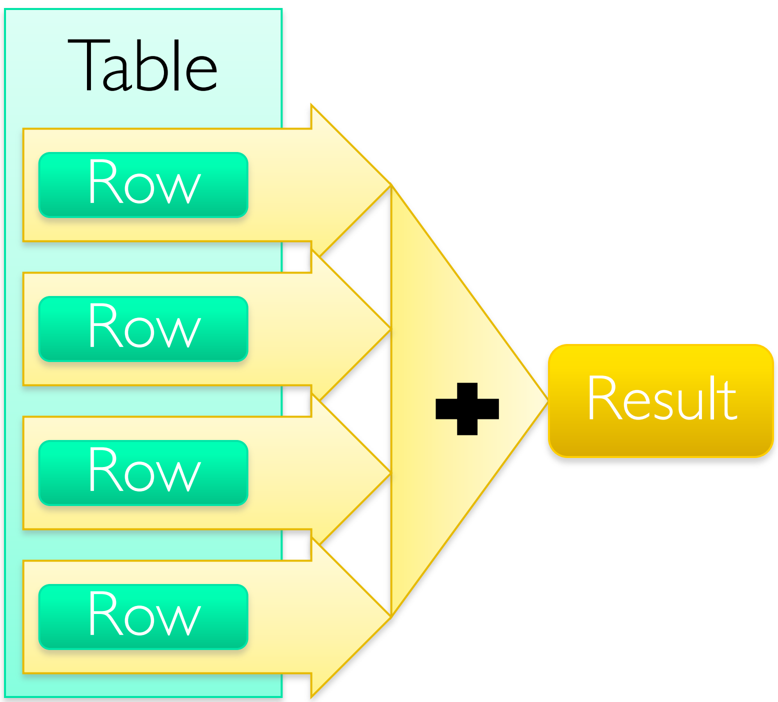


Separate Systems to Support Each View

Table View

hadoop

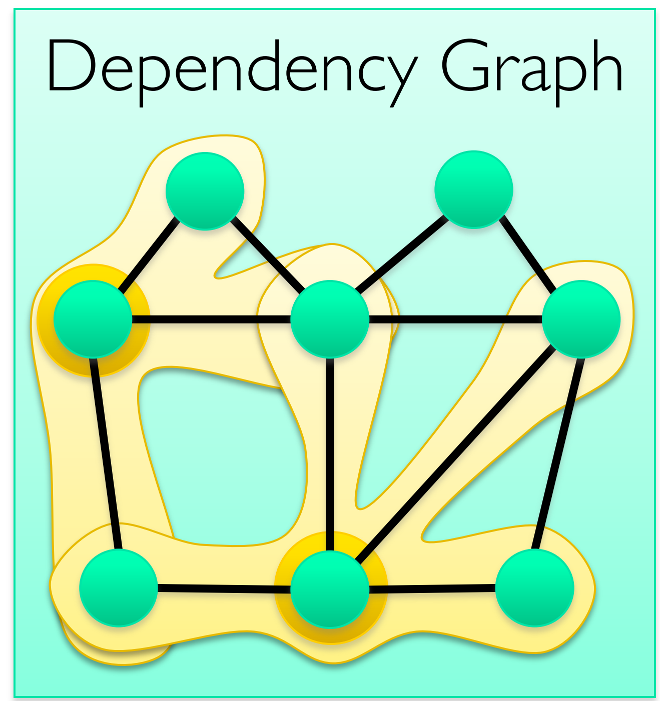
Spark



Graph View

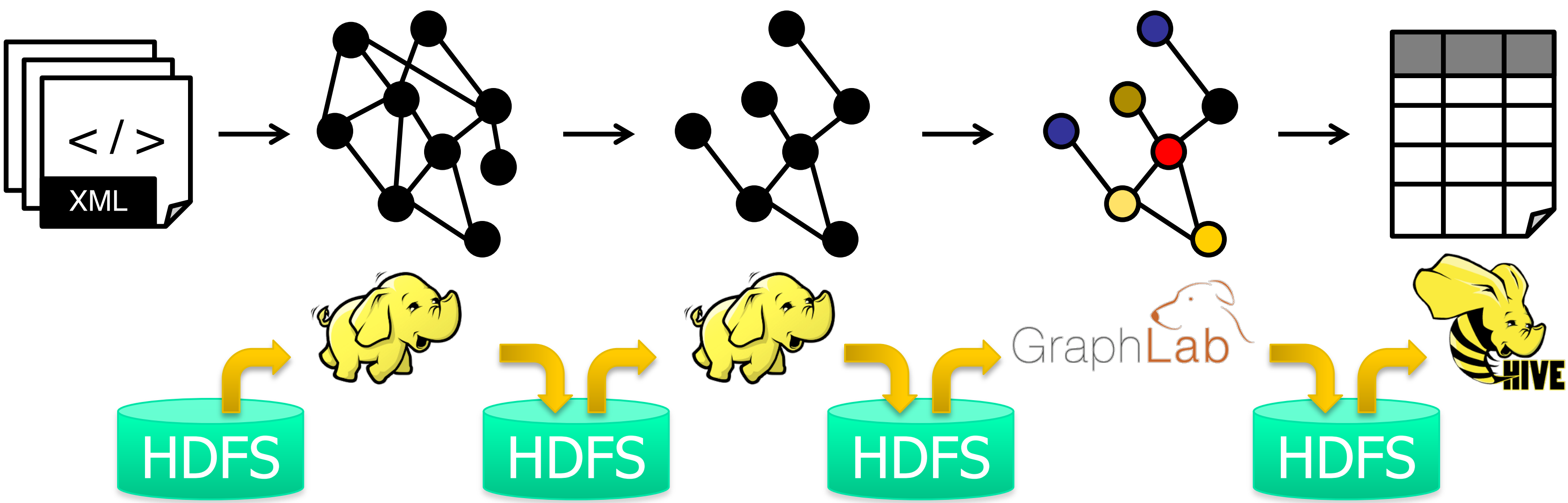
Pregel GraphLab

APACHE GIRAPH



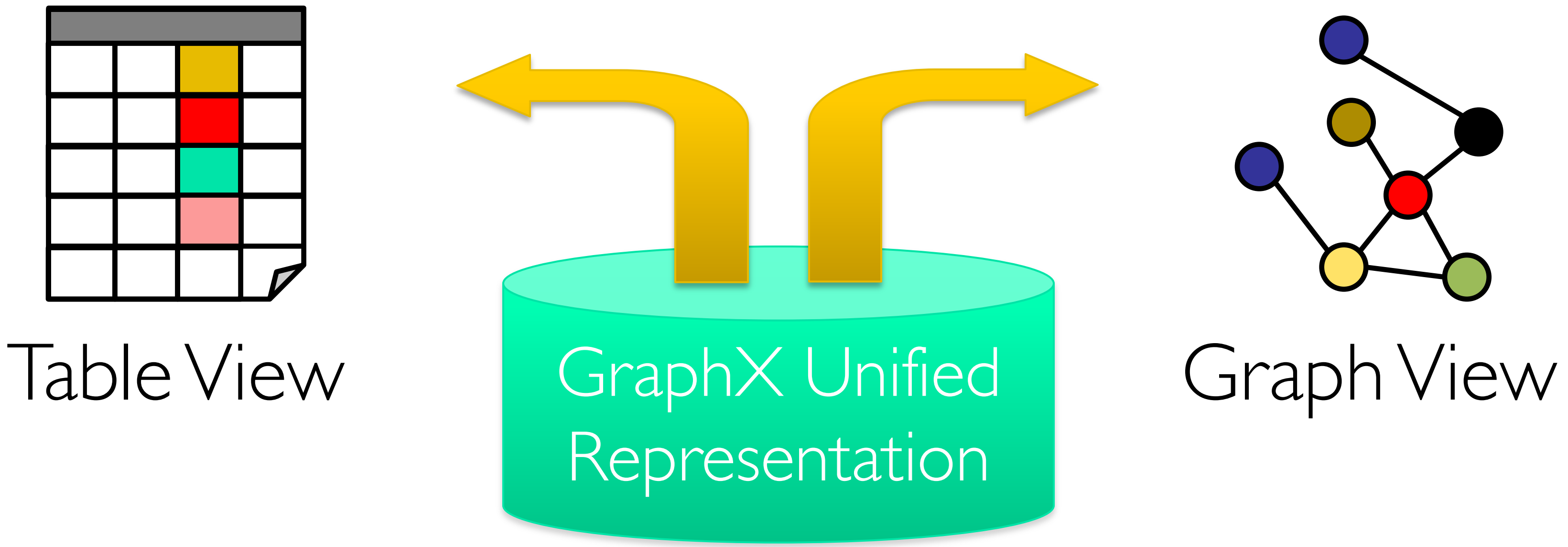
Inefficient

Expensive data movement and duplication across the network and file system



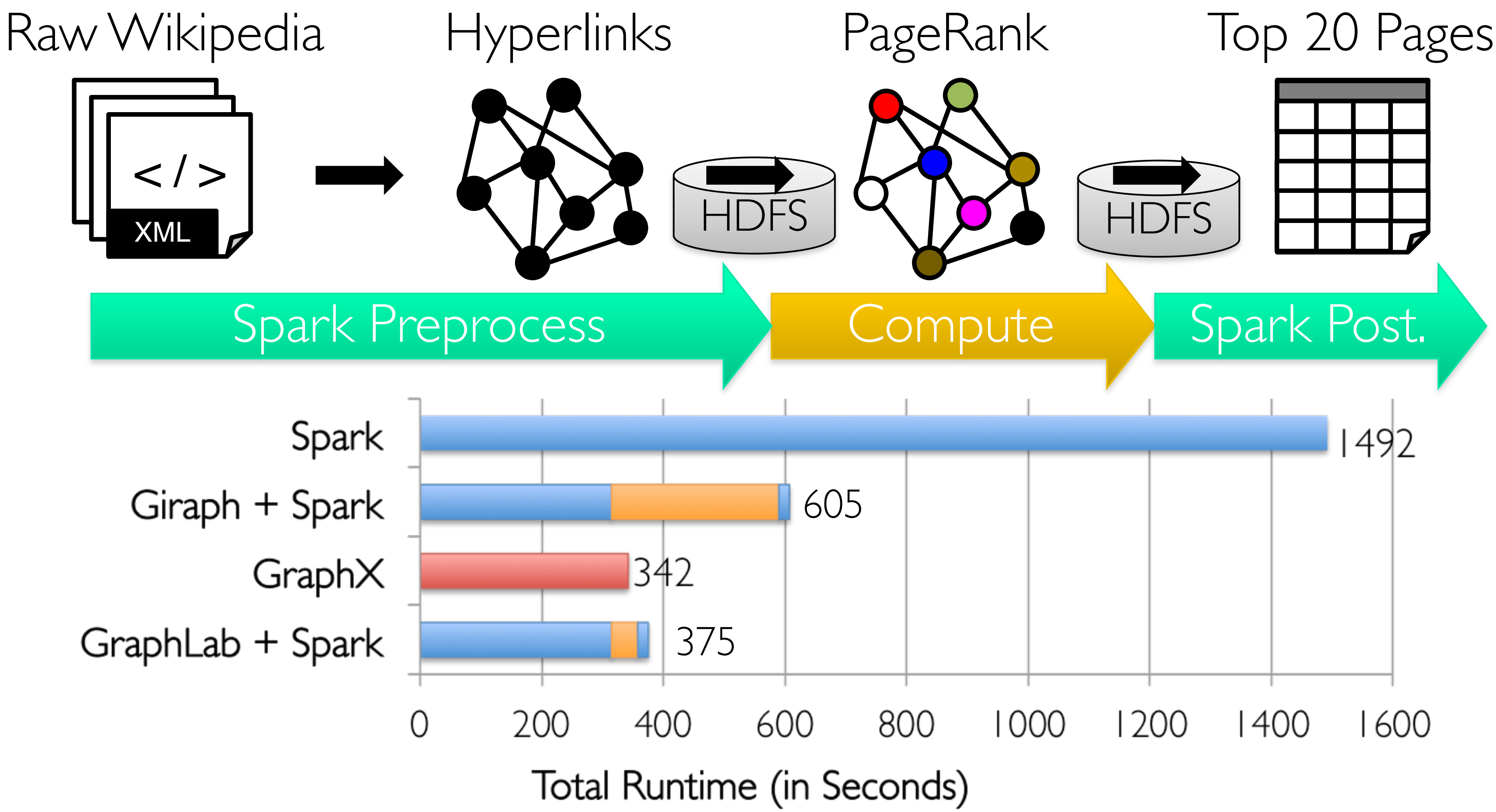
Limited reuse internal data-structures across stages

Tables and Graphs are composable *views* of the *same physical* data



Each view has its own operators that exploit the semantics of the view to achieve efficient execution

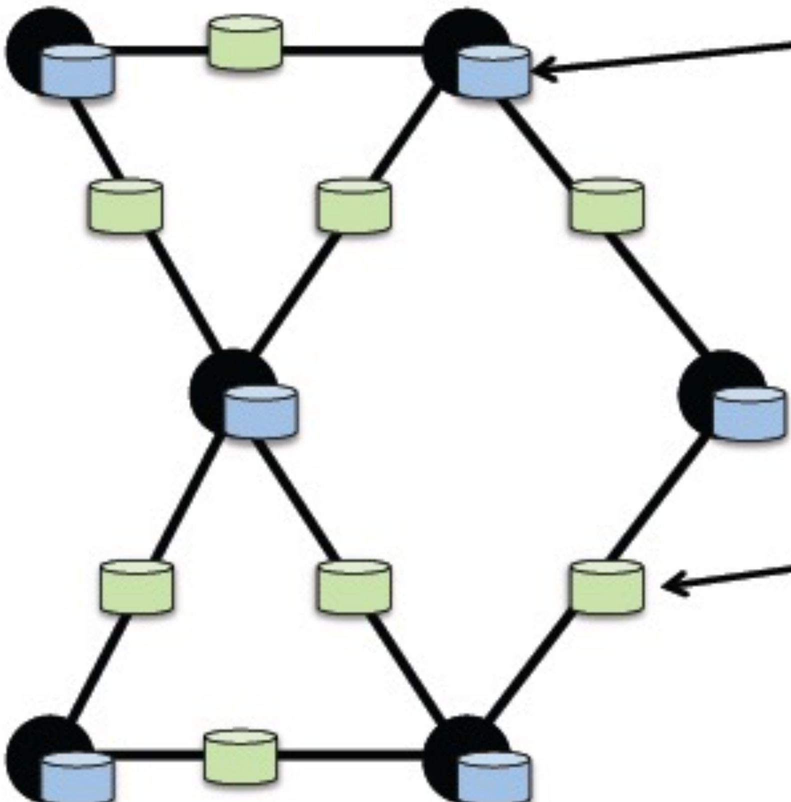
A Small Pipeline in GraphX



Timed end-to-end GraphX is *faster* than GraphLab

The GraphX API

Property Graphs



Vertex Property:

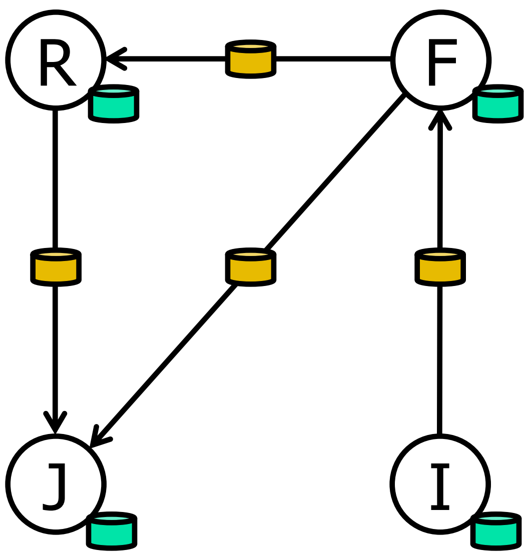
- User Profile
- Current PageRank Value

Edge Property:

- Weights
- Relationships
- Timestamps

View a Graph as a Table

Property Graph



Vertex Property Table

Id	Property (V)
Rxin	(Stu., Berk.)
Jegonzal	(PstDoc, Berk.)
Franklin	(Prof., Berk)
Istoica	(Prof., Berk)

Edge Property Table

SrcId	DstId	Property (E)
rxin	jegonzal	Friend
franklin	rxin	Advisor
istoica	franklin	Coworker
franklin	jegonzal	PI

Distributed Graphs as Tables (RDDs)

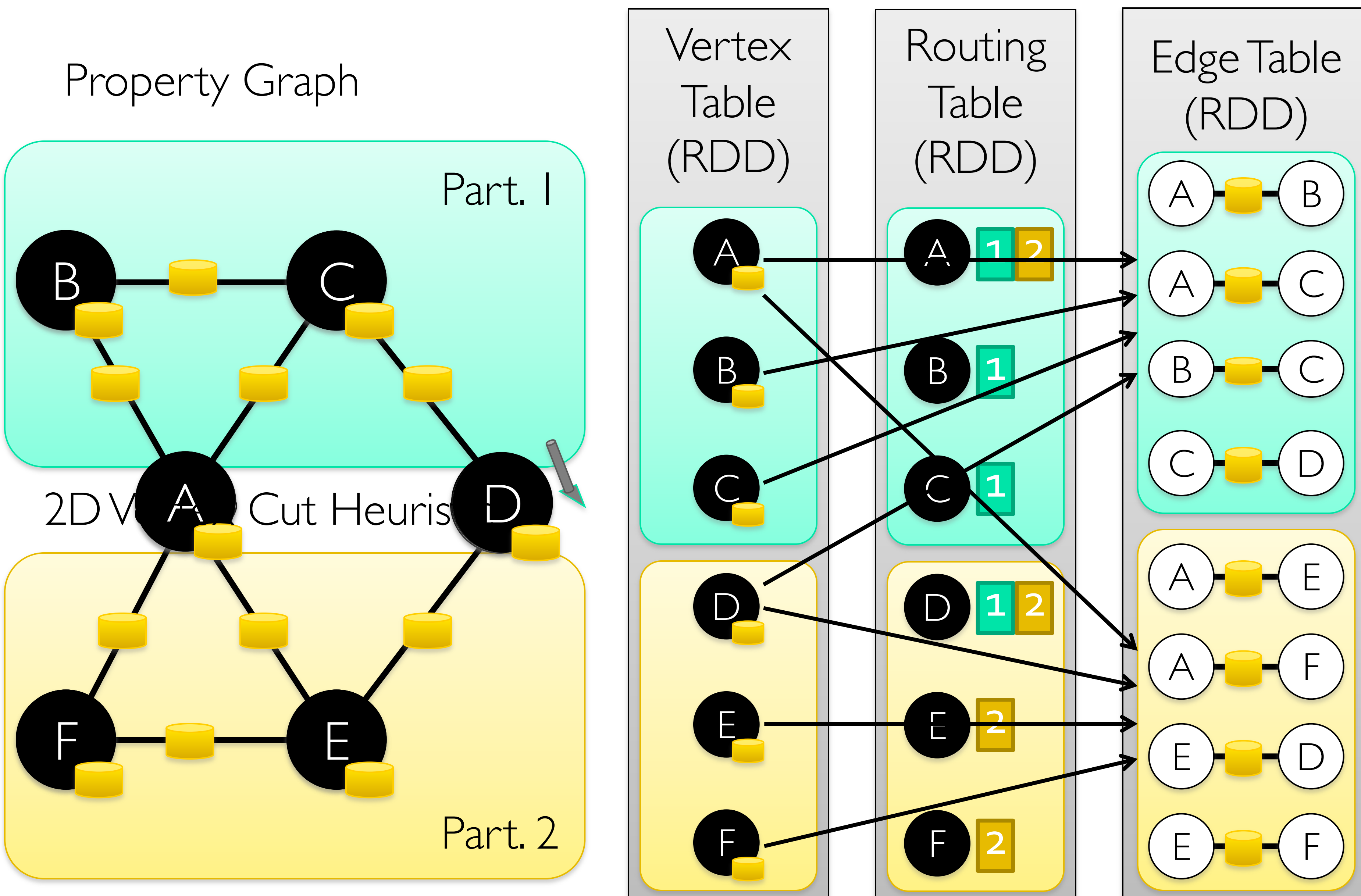


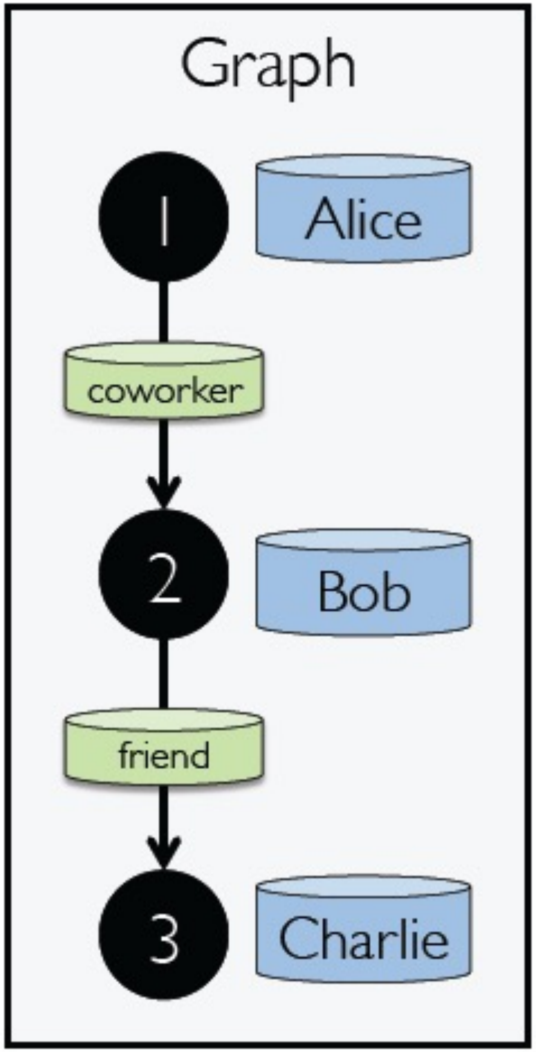
Table Operators

- Table (RDD) operators are inherited from Spark:

`map`	`reduce`	`sample`
`filter`	`count`	`take`
`groupBy`	`fold`	`first`
`sort`	`reduceByKey`	`partitionBy`
`union`	`groupByKey`	`mapWith`
`join`	`cogroup`	`pipe`
`leftOuterJoin`	`cross`	`save`
`rightOuterJoin`	`zip`	`...`

Creating a Graph (Scala)

```scala
type VertexId = Long

val vertices: RDD[(VertexId, String)] =
  sc.parallelize(List(
    (1L, "Alice"),
    (2L, "Bob"),
    (3L, "Charlie")))

class Edge[ED](
  val srcId: VertexId,
  val dstId: VertexId,
  val attr: ED)

val edges: RDD[Edge[String]] =
  sc.parallelize(List(
    Edge(1L, 2L, "coworker"),
    Edge(2L, 3L, "friend")))

val graph = Graph(vertices, edges)
```



Graph Operations (Scala)

```
/** Summary of the functionality in the property graph */
class Graph[VD, ED] {
  // Information about the Graph ===========================================================
==============
  val numEdges: Long
  val numVertices: Long
  val inDegrees: VertexRDD[Int]
  val outDegrees: VertexRDD[Int]
  val degrees: VertexRDD[Int]
  // Views of the graph as collections ===============================================
==============
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
  val triplets: RDD[EdgeTriplet[VD, ED]]
  // Functions for caching graphs =======================================================
==============
  def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
  def cache(): Graph[VD, ED]
  def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
  // Change the partitioning heuristic  ================================================
```

Graph Operations (Scala)

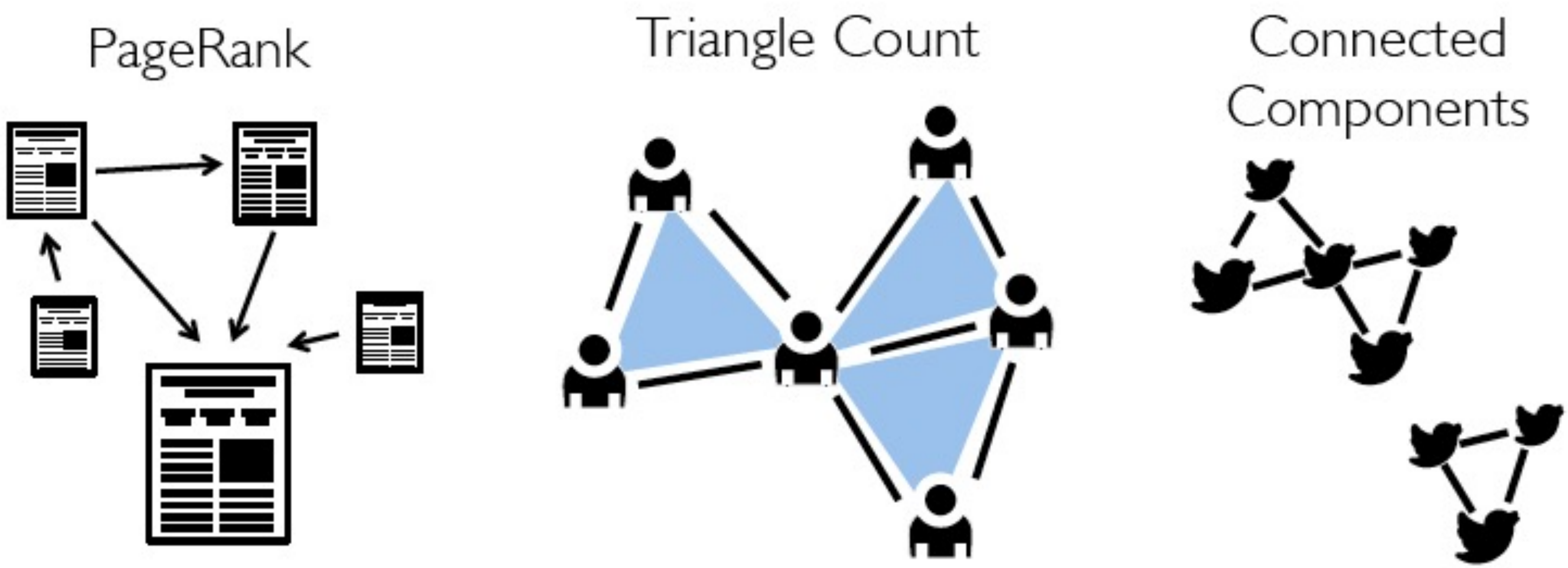
```
  // Transform vertex and edge attributes ==============================================
==============
  def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]) => Iterator[ED2]): Graph[VD
, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) => Iterator[E
D2])
    : Graph[VD, ED2]
  // Modify the graph structure ========================================================
==============
  def reverse: Graph[VD, ED]
  def subgraph(
      epred: EdgeTriplet[VD,ED] => Boolean = (x => true),
      vpred: (VertexID, VD) => Boolean = ((v, d) => true))
    : Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
```

```scala
  // Join RDDs with the graph ==============================================================
==============
  def joinVertices[U](table: RDD[(VertexID, U)])(mapFunc: (VertexID, VD, U) => VD): Gr
aph[VD, ED]

  def outerJoinVertices[U, VD2](other: RDD[(VertexID, U)])
      (mapFunc: (VertexID, VD, Option[U]) => VD2)
    : Graph[VD2, ED]
  // Aggregate information about adjacent triplets ==========================================
==============
  def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array[VertexID]]
  def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[Array[(VertexID, VD)]]
  def aggregateMessages[Msg: ClassTag](
      sendMsg: EdgeContext[VD, ED, Msg] => Unit,
      mergeMsg: (Msg, Msg) => Msg,
      tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[A]
  // Iterative graph-parallel computation ===============================================
==============
  def pregel[A](initialMsg: A, maxIterations: Int, activeDirection: EdgeDirection)(
      vprog: (VertexID, VD, A) => VD,
      sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexID,A)],
      mergeMsg: (A, A) => A)
    : Graph[VD, ED]
  // Basic graph algorithms ==============================================================
==============
  def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double, Double]
  def connectedComponents(): Graph[VertexID, ED]
  def triangleCount(): Graph[Int, ED]
  def stronglyConnectedComponents(numIter: Int): Graph[VertexID, ED]
}
```

Built-in Algorithms (Scala)

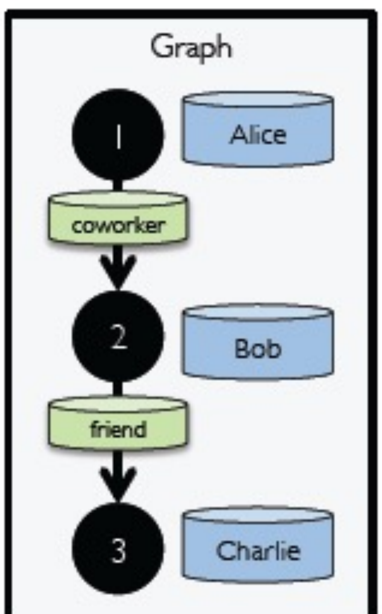
```
    // Continued from previous slide
    def pageRank(tol: Double): Graph[Double, Double]
    def triangleCount(): Graph[Int, ED]
    def connectedComponents(): Graph[VertexId, ED]
    // ...and more: org.apache.spark.graphx.lib
}
```



The "`triplets`" view

```
class Graph[VD, ED] {
    def triplets: RDD[EdgeTriplet[VD, ED]]
}

class EdgeTriplet[VD, ED](
  val srcId: VertexId, val dstId: VertexId, val attr: ED,
  val srcAttr: VD, val dstAttr: VD)
```



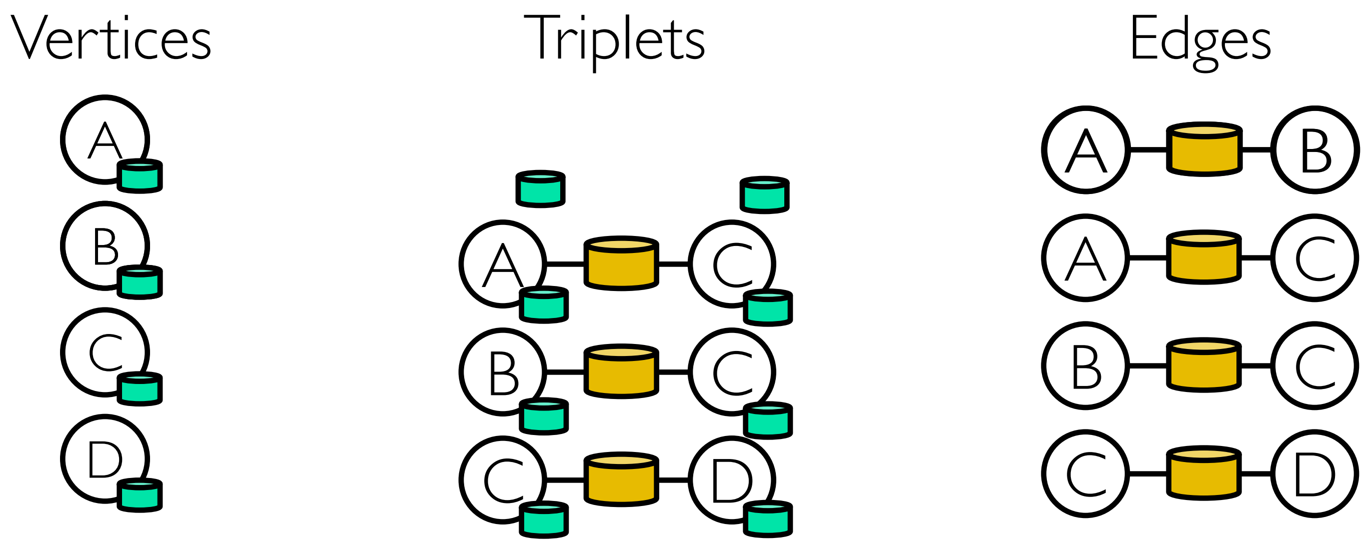
triplets →

RDD

srcAttr	dstAttr	attr
Alice	coworker	Bob
Bob	friend	Charlie

Triplets Join Vertices and Edges

- The *triplets* operator joins vertices and edges:

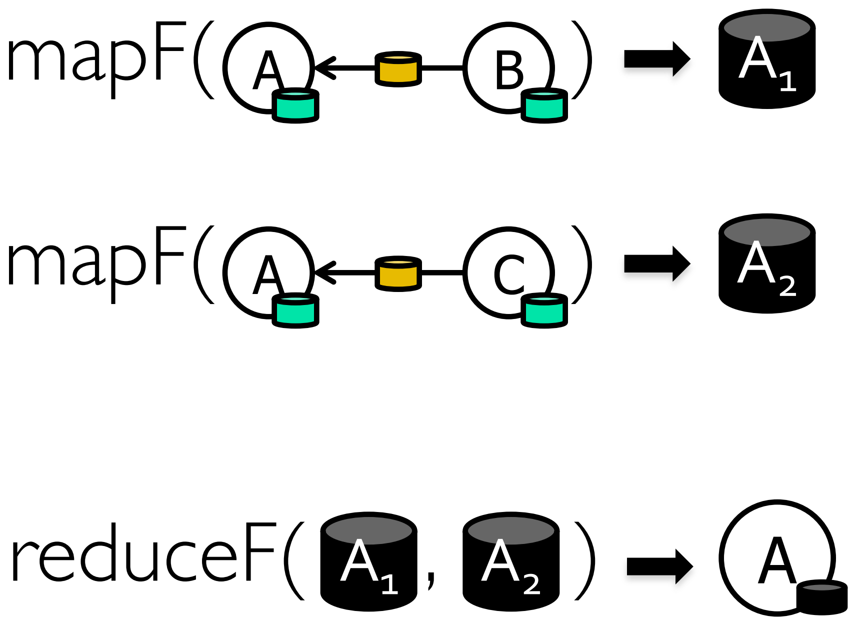


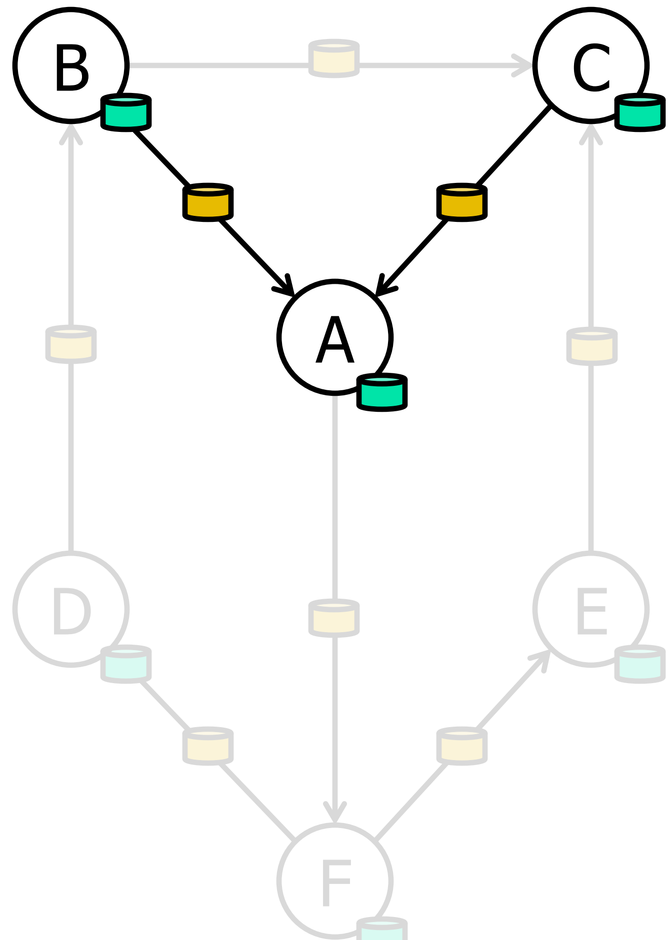
The *mapreduceTriplets* operator sums adjacent triplets.

SELECT t.dstId, *reduceUDF*(*mapUDF*(t)) **AS** sum
FROM triplets **AS** t **GROUPBY** t.dstId

Map Reduce Triplets

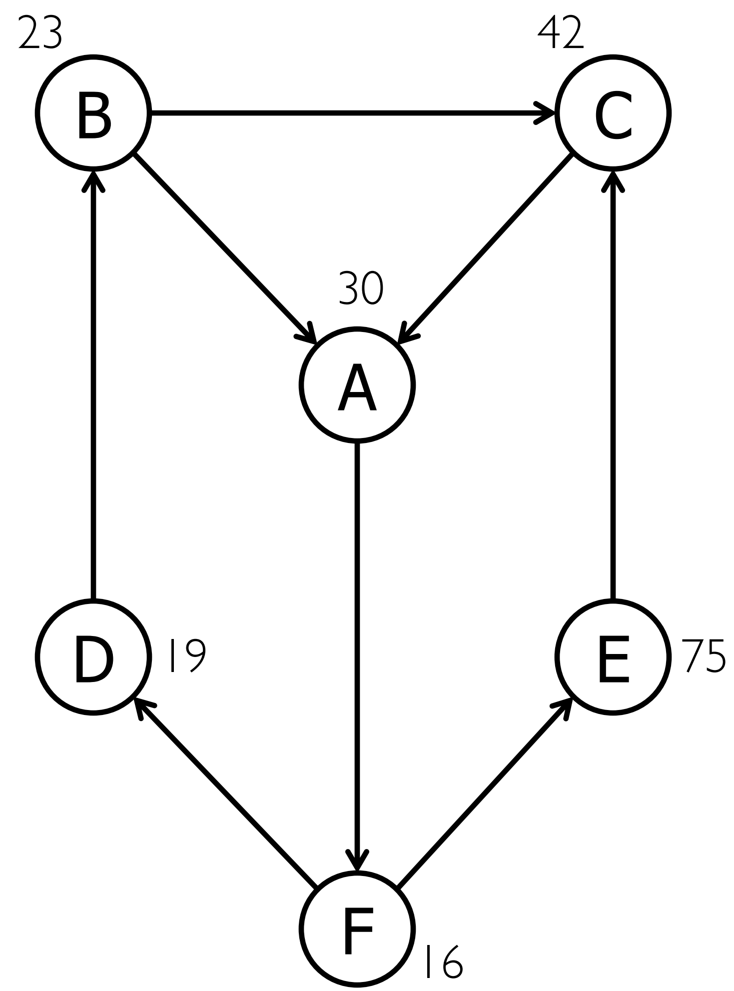
- *Map-Reduce* for each vertex





Example: Oldest Follower

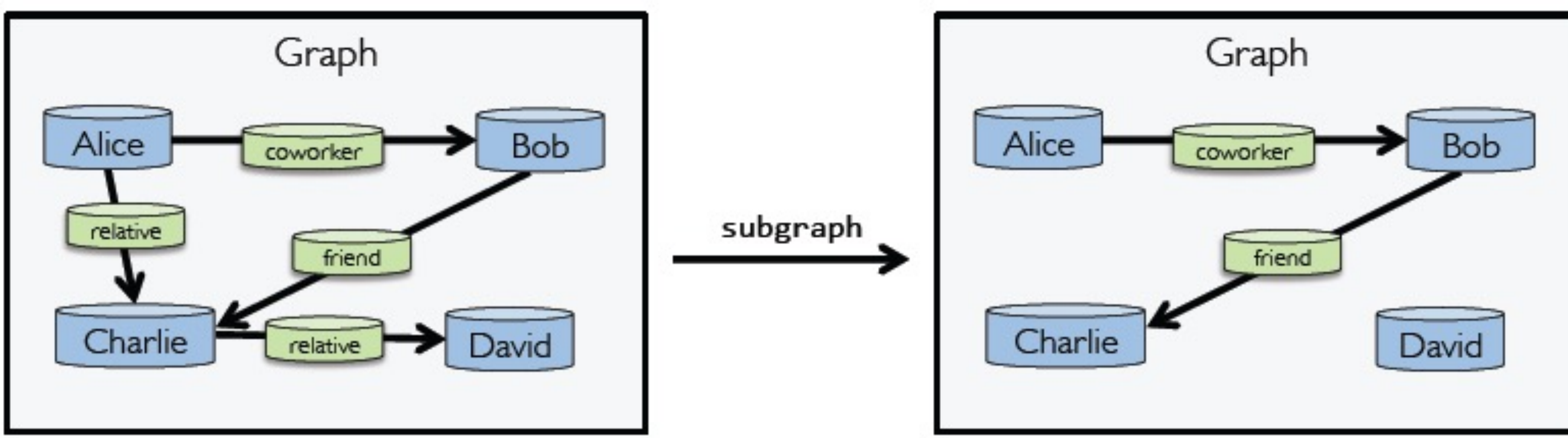
- *What is the age of the oldest follower for each user?*
- ```
  val oldestFollowerAge = graph
    .mapreduceTriplets(
      e=> (e.dst.id,
  e.src.age),//Map
      (a,b)=> max(a, b) //Reduce
    )
    .vertices
  ```


  ```

The `subgraph` transformation

```
class Graph[VD, ED] {
    def subgraph(epred: EdgeTriplet[VD, ED] => Boolean,
                 vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
}

graph.subgraph(epred = (edge) => edge.attr != "relative")
```

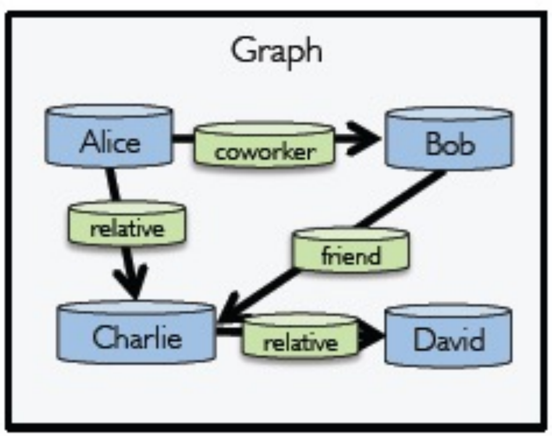


Computation w/ `mapReduceTriplets`

```
class Graph[VD, ED] {
  def mapReduceTriplets[A](
    sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
    mergeMsg: (A, A) => A): RDD[(VertexId, A)]
}
```

⚠ upgrade to `aggregateMessages` in Spark 1.2.0

```
graph.mapReduceTriplets(
  edge => Iterator(
    (edge.srcId, 1),
    (edge.dstId, 1)),
  _ + _)
```



`mapReduceTriplets` →

RDD

vertex id	degree
Alice	2
Bob	2
Charlie	3
David	1

Computation w/ `aggregateMessages`

```scala
class Graph[VD, ED] {
  def aggregateMessages[Msg: ClassTag](
      sendMsg: EdgeContext[VD, ED, Msg] => Unit,
      mergeMsg: (Msg, Msg) => Msg,
      tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[Msg]
}
```

The “aggregateMessages” operator:

(1) Apply a user-defined sendMsg function to each *edge triplet* in the graph and then
(2) Use the another user-defined mergeMsg function to aggregate those messages at their destination vertex.

Example: Compute Average Age of Older Followers of each node using **aggregateMessages**

```
import org.apache.spark.graphx.{Graph, VertexRDD}
import org.apache.spark.graphx.util.GraphGenerators

// Create a graph with "age" as the vertex property.
// Here we use a random graph for simplicity.
val graph: Graph[Double, Int] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapVertices( (id, _) => id.toDouble )
// Compute the number of older followers and their total age
val olderFollowers: VertexRDD[(Int, Double)] = graph.aggregateMessages[(Int, Double)](
  triplet => { // Map Function
    if (triplet.srcAttr > triplet.dstAttr) {
      // Send message to destination vertex containing counter and age
      triplet.sendToDst((1, triplet.srcAttr))
    }
  },
  // Add counter and age
  (a, b) => (a._1 + b._1, a._2 + b._2) // Reduce Function
)
// Divide total age by number of older followers to get average age of older followers
val avgAgeOfOlderFollowers: VertexRDD[Double] =
  olderFollowers.mapValues( (id, value) =>
    value match { case (count, totalAge) => totalAge / count } )
// Display the results
avgAgeOfOlderFollowers.collect.foreach(println(_))
```

Refer to "examples/src/main/scala/org/apache/spark/examples/graphx/AggregateMessagesExample.scala" in Spark repo for the full source code of this example

Have Expressed the Pregel and GraphLab abstractions using the GraphX operators in less than 50 lines of code!

By composing these operators we can construct entire graph-analytics pipelines.

Re-implementation of the Pregel abstraction using the GraphX API

```
def Pregel(g: Graph[V, E],
      vprog: (Id, V, M) => V,
      sendMsg: (Triplet) => M,
      gather: (M, M) => M): Collection[V] = {
 // Set all vertices as active
 g = g.mapV((id, v) => (v, halt=false))
 // Loop until convergence
 while (g.vertices.exists(v => !v.halt)) {
   // Compute the messages
   val msgs: Collection[(Id, M)] =
     // Restrict to edges with active source
     g.subgraph(ePred=(s,d,sP,eP,dP)=>!sP.halt)
     // Compute messages
      .mrTriplets(sendMsg, gather)
   // Receive messages and run vertex program
   g = g.leftJoinV(msgs).mapV(vprog)
 }
 return g.vertices
}
```

Finding Connected Components using the GraphX variant of Pregel

```
def ConnectedComp(g: Graph[V, E]) = {
  g = g.mapV(v => v.id) // Initialize vertices
  def vProg(v: Id, m: Id): Id = {
    if (v == m) voteToHalt(v)
    return min(v, m)
  }
  def sendMsg(t: Triplet): Id =
    if (t.src.cc < t.dst.cc) t.src.cc
    else None // No message required
  def gatherMsg(a: Id, b: Id): Id = min(a, b)
  return Pregel(g, vProg, sendMsg, gatherMsg)
}
```

Listing 6: **Connected Components:** For each vertex we compute the lowest reachable vertex id using Pregel.

GraphX System Design

Graph Partitioning Strategies

Edge Cut in GraphLab 1.0 vs.
Vertex Cut in GraphLab 2.0 in PowerGraph and GraphX

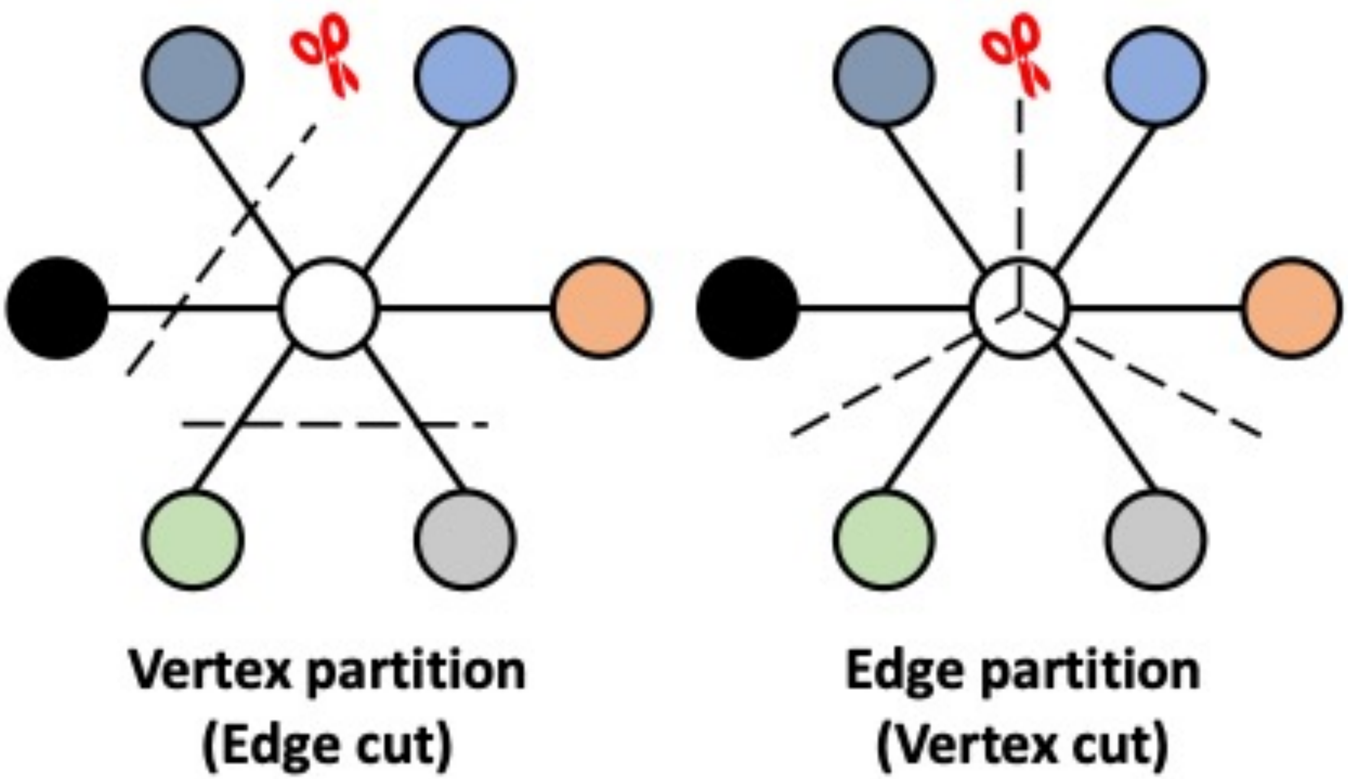
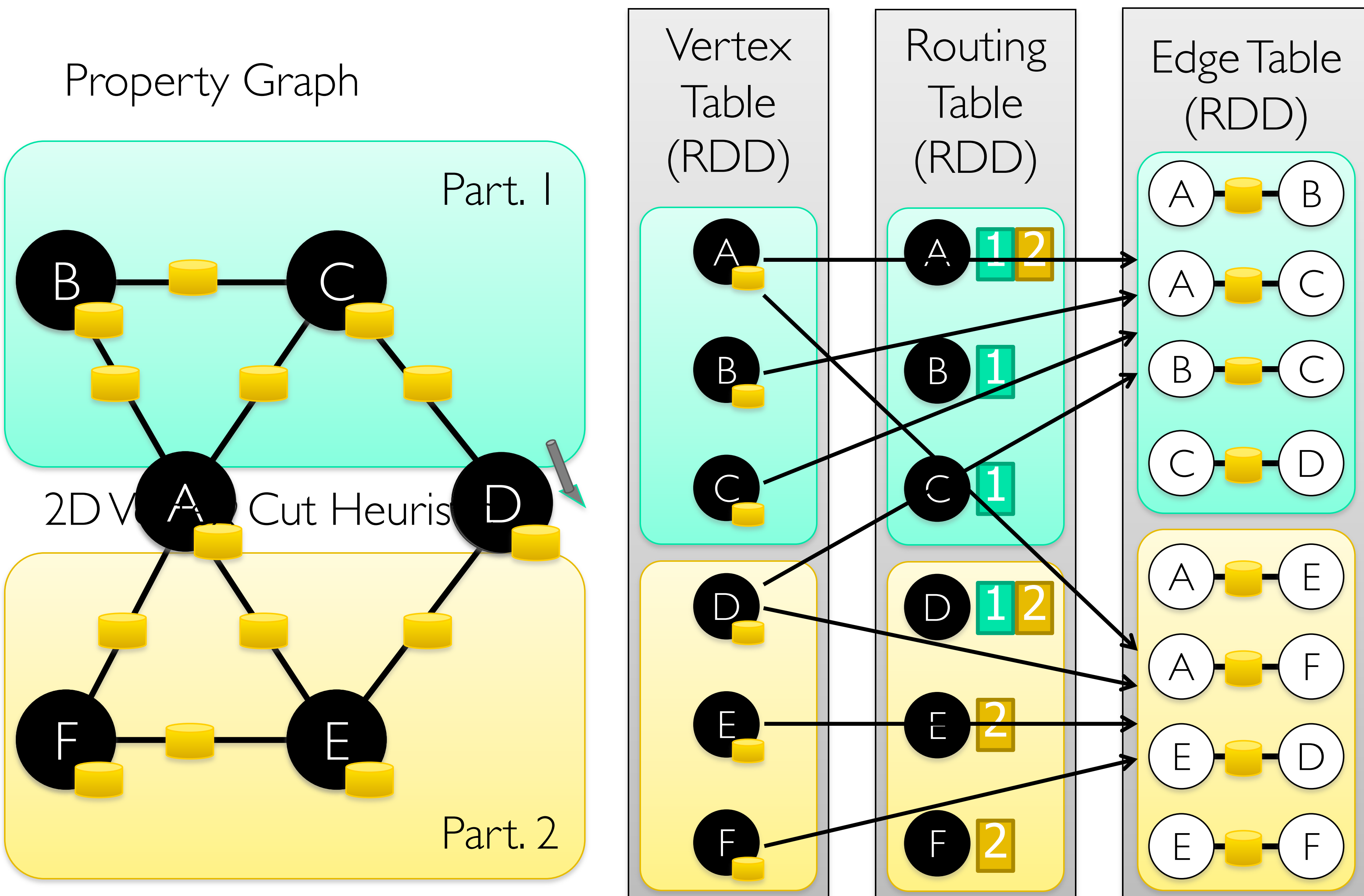


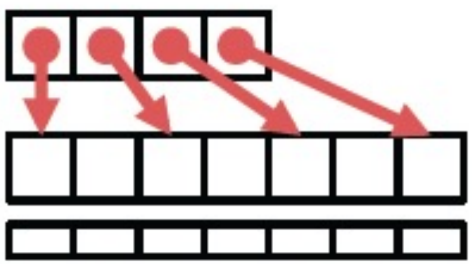
Fig. 2. Partitioning strategies.

Distributed Graphs as Tables (RDDs)

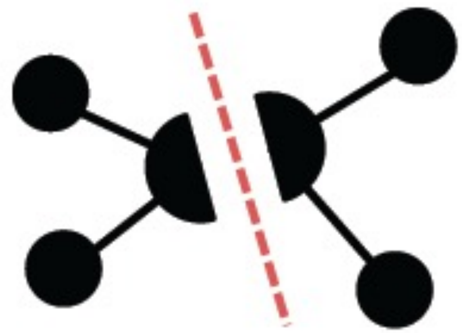


Graph System Optimizations

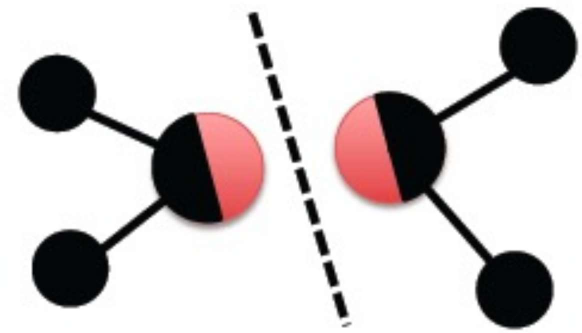
Specialized Data-Structures



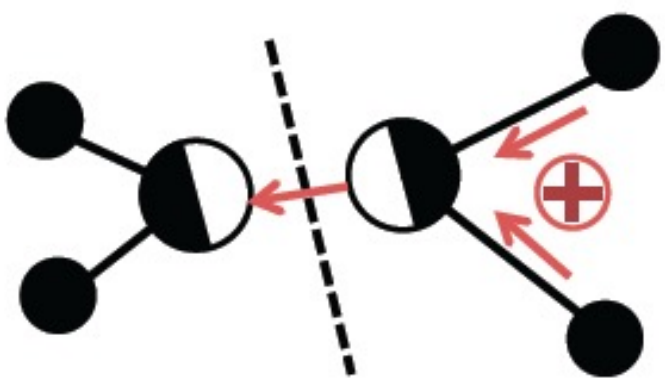
Vertex-Cuts Partitioning



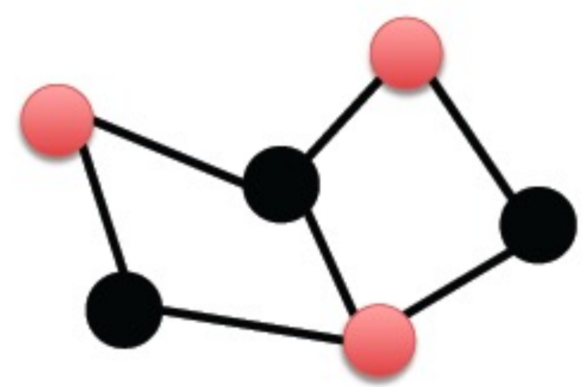
Remote Caching / Mirroring



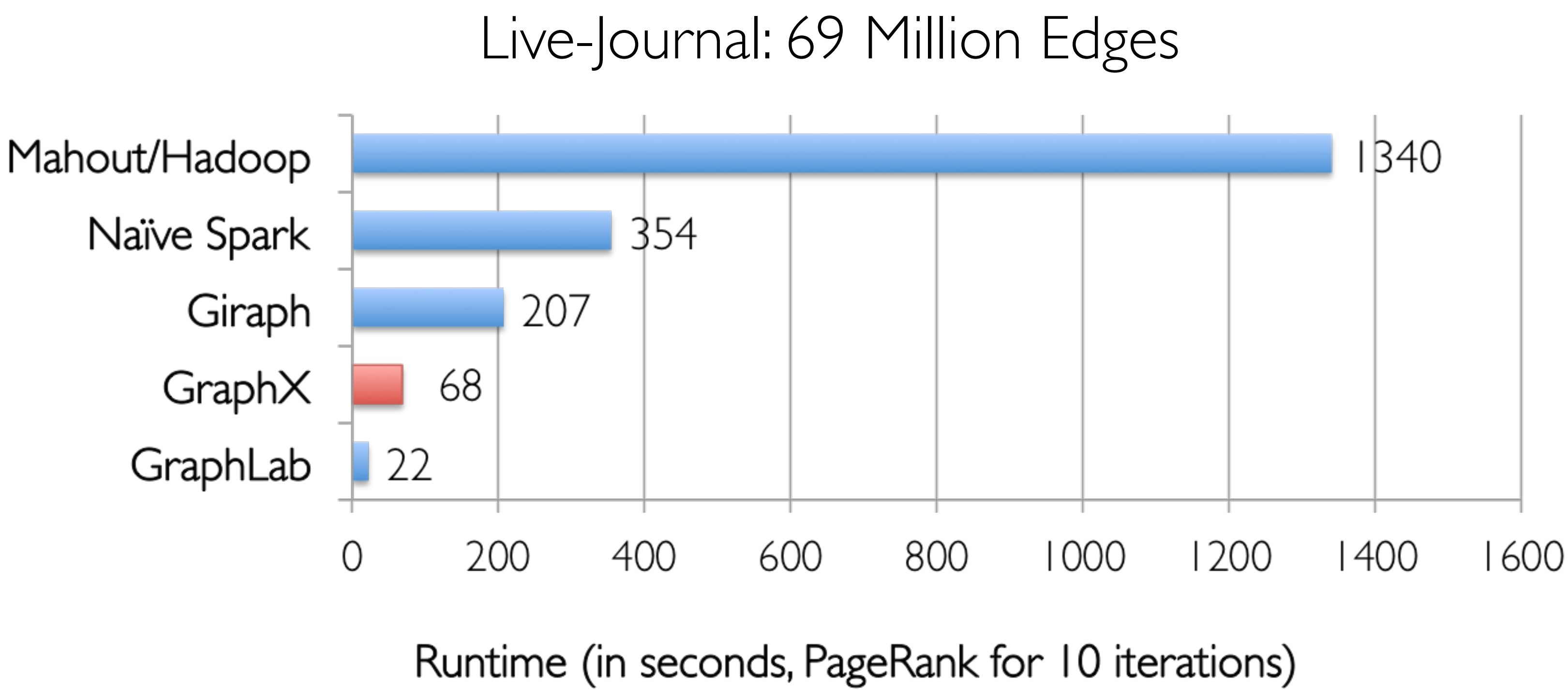
Message Combiners



Active Set Tracking

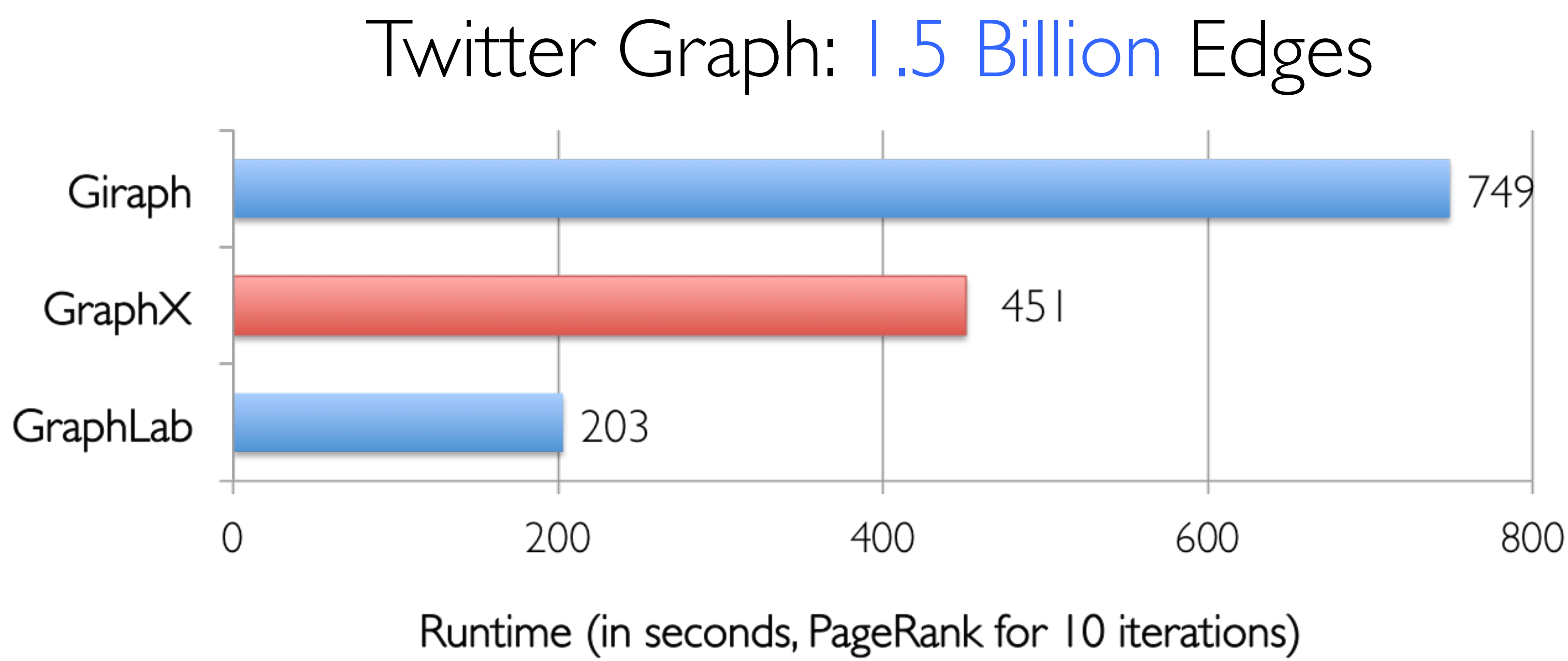


Performance Comparisons



GraphX is roughly 3x slower than GraphLab

GraphX scales to larger graphs

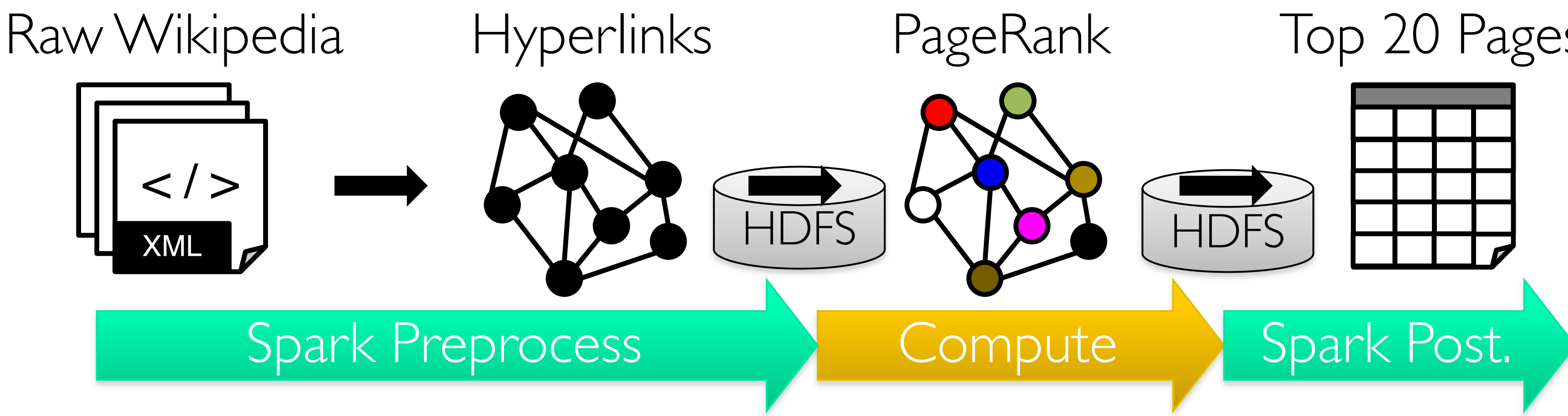


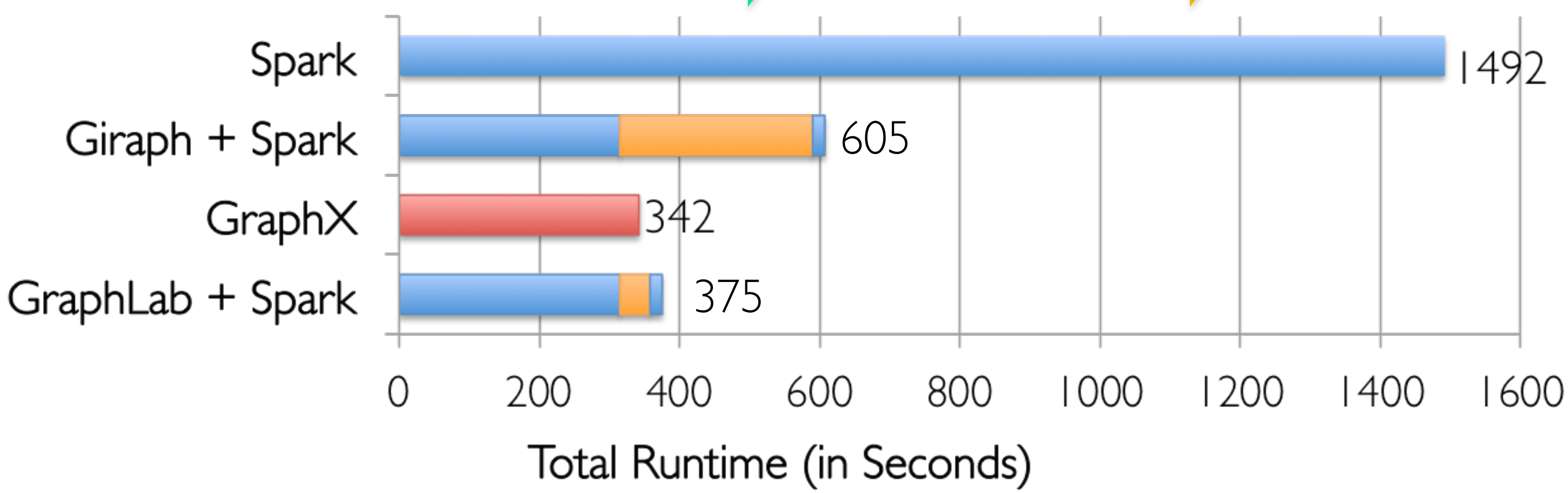
GraphX is roughly 2x slower than GraphLab

» Scala + Java overhead: Lambdas, GC time, …

» No shared memory parallelism: 2x increase in comm.

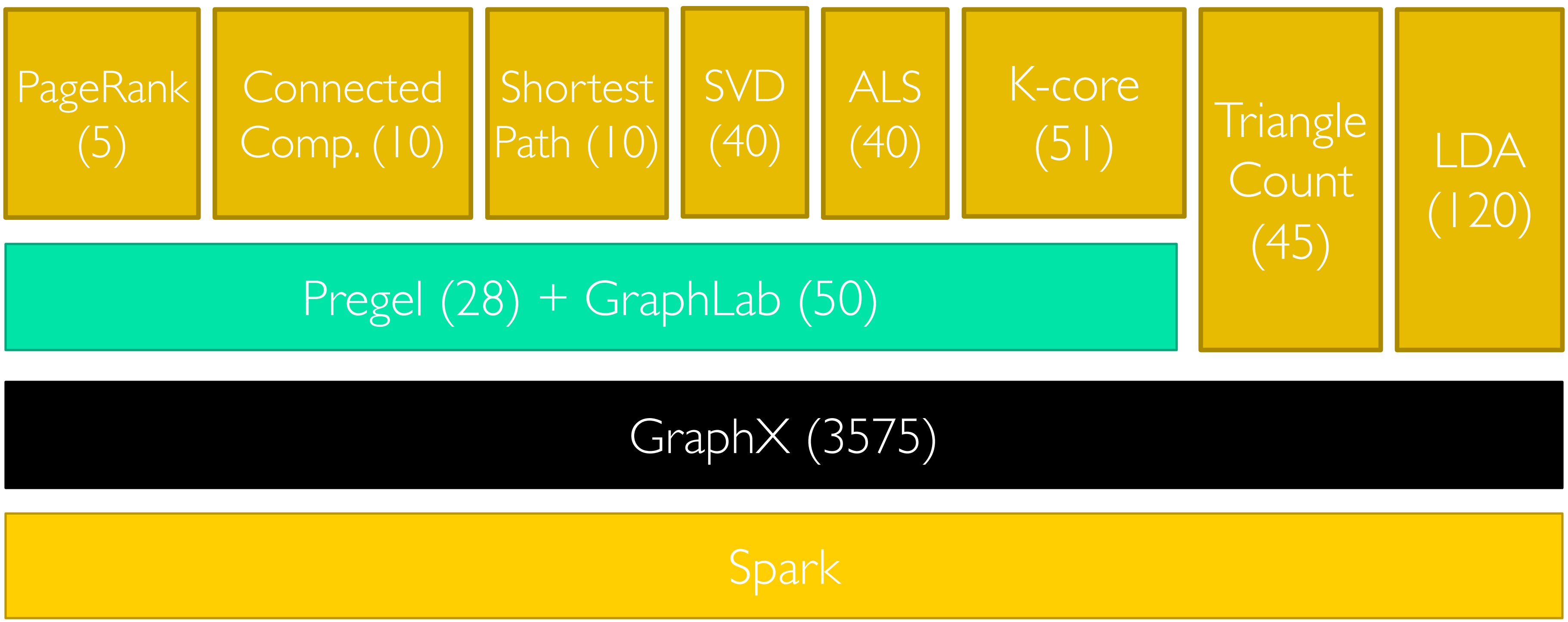
A Small Pipeline in GraphX





Timed end-to-end GraphX is *faster* than GraphLab

The GraphX Stack (Lines of Code)



GraphX: Summary and Observations

- Domain specific views: *Tables* and *Graphs*
  - tables and graphs are first-class composable objects
  - specialized operators which exploit view semantics
- Single system that efficiently spans the pipeline
  - minimize data movement and duplication
  - eliminates need to learn and manage multiple systems
- Graphs through the lens of database systems
  - Graph-Parallel Pattern → Triplet joins in relational alg.
  - Graph Systems → Distributed join optimizations

Directions for Further Development of GraphX

- Static Data → Dynamic Data, Time-Evolving Big Graphs
  - Apply GraphX unified approach to time evolving data
  - Model and analyze relationships over time

  => e.g. See the GraphTau paper in GRADES 2016.

- Serving Graph Structured Data
  - Allow external systems to interact with GraphX
  - Unify distributed graph databases with relational database technology

  => Refer to the next topic: Graphframes

Summary of Apache Spark's GraphX library

Strength

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

Limitations

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

See http://amplab.github.io/graphx/ for more details.

Enter GraphFrames

https://github.com/graphframes.graphframe

Motivations for GraphFrames

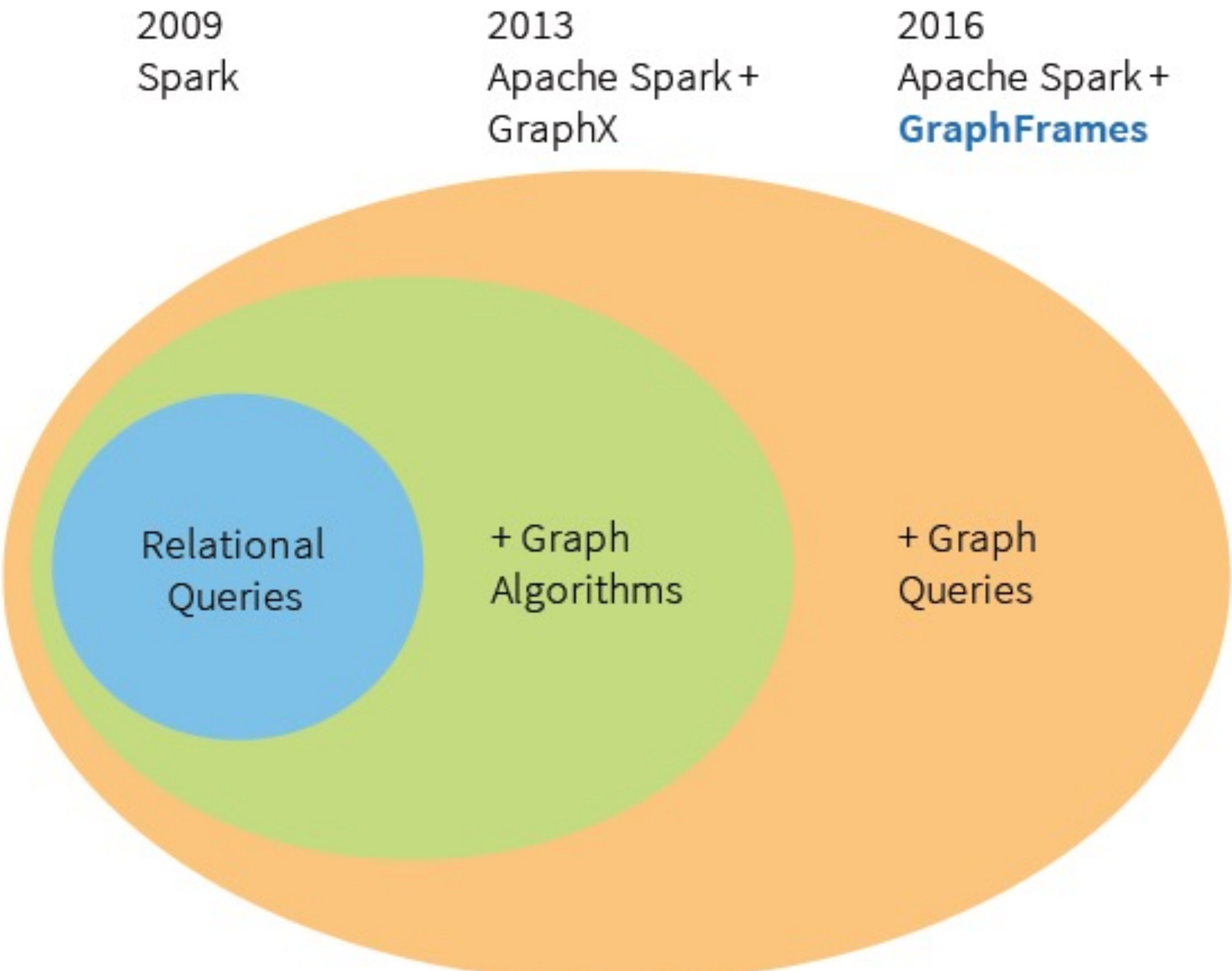
Goal: Support DataFrame-based Graph processing on Spark

- Simplify Interactive Queries
- Support Motif-finding for Structural Pattern Search
- Benefit from DataFrame Optimization

Collaborations between Databricks, UC Berkeley & MIT

- Now with open-source community contributors

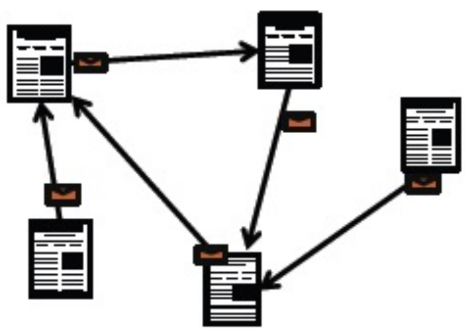
Evolution of Graph Processing Support in Spark



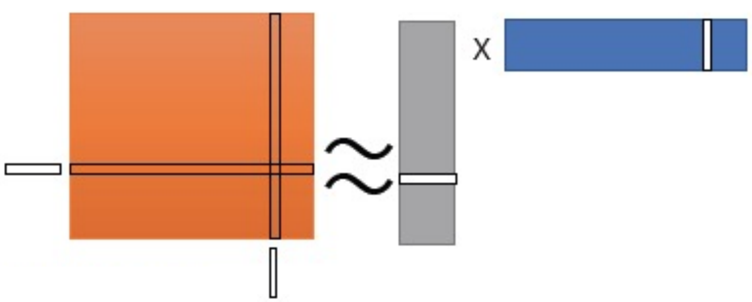
Graph Algorithms vs. Graph Queries

Graph Algorithms

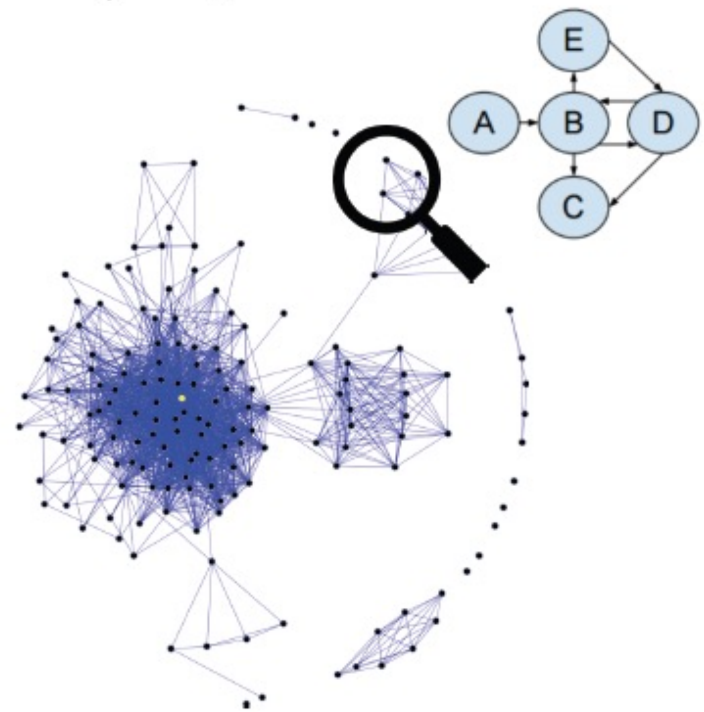
PageRank



Alternating Least Squares

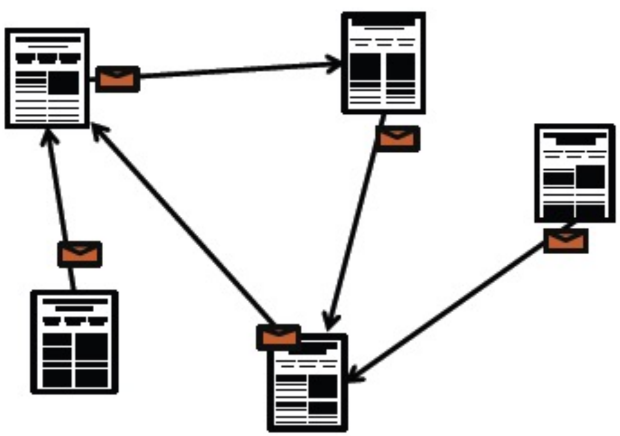


Graph Queries

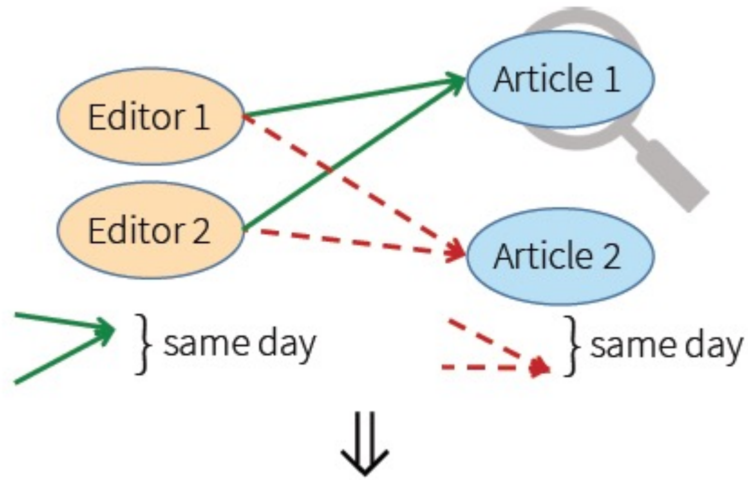


Graph Algorithms vs. Graph Queries, an Example

Graph Algorithm: PageRank



Graph Query: Wikipedia Collaborators



Editor 1	Editor 2	Article 1	Article 2

Graph Algorithms vs. Graph Queries, an Example

Graph Algorithm: PageRank

```
// Iterate until convergence
wikipedia.pregel(
  sendMsg = { e =>
    e.sendToDst(e.srcRank * e.weight)
  },
  mergeMsg = _ + _,
  vprog = { (id, oldRank, msgSum) =>
    0.15 + 0.85 * msgSum
  })
```

Graph Query: Wikipedia Collaborators

```
wikipedia.find(
  "(u1)-[e11]->(article1);
   (u2)-[e21]->(article1);
   (u1)-[e12]->(article2);
   (u2)-[e22]->(article2)")
 .select(
   "*",
   "e11.date – e21.date".as("d1"),
   "e12.date – e22.date".as("d2"))
 .sort("d1 + d2".desc).take(10)
```

Before GraphFrames:

Separate Graph Database/ Frameworks to support Graph Algorithms and Graph Queries

Graph algorithms

GraphX

APACHE GIRAPH

GraphLab

Standard & custom algorithms
Optimized for batch processing

Graph queries

neo4j

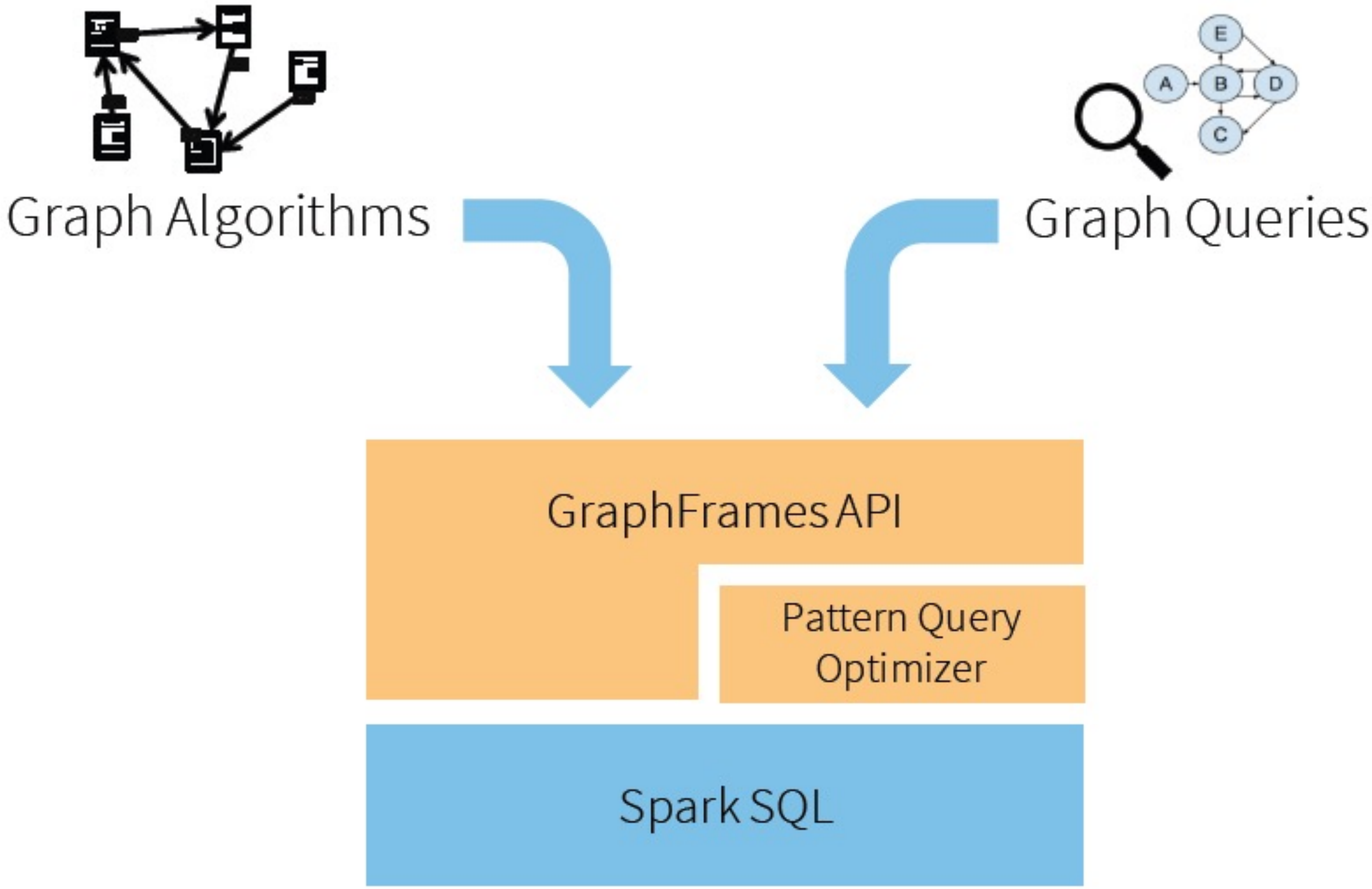
TITAN

OrientDB

Motif finding
Point queries & updates

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

System Architecture of GraphFrames



GraphFrames vs. GraphX

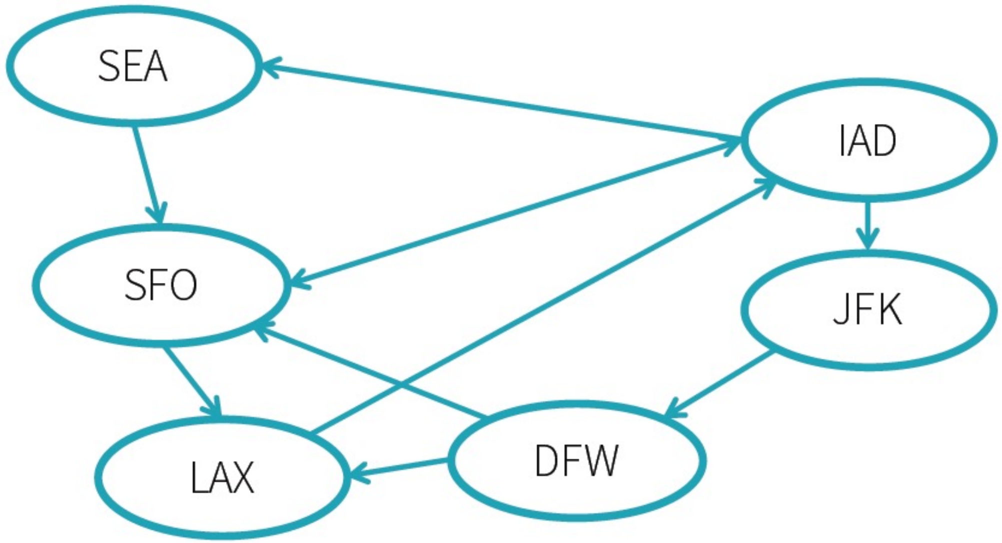
	GraphFrames	**GraphX**
Builton	DataFrames	RDDs
Languages	Scala, Java, Python	Scala
Use cases	Queries & algorithms	Algorithms
Vertex IDs	Any type (in Catalyst)	Long
Vertex/edge attributes	Any number of DataFrame columns	Any type (VD, ED)
Return types	GraphFrame or DataFrame	Graph[VD, ED], or RDD[Long, VD]

GraphFrames API

- Unifies graph algorithms, graph queries, and DataFrames
- Available in Scala, Java, and Python

```
class GraphFrame {
  def vertices: DataFrame
  def edges: DataFrame

  def find(pattern: String): DataFrame
  def registerView(pattern: String, df: DataFrame): Unit

  def degrees(): DataFrame
  def pageRank(): GraphFrame
  def connectedComponents(): GraphFrame
  ...
}
```

Representing a Graph in GraphFrames - an Example



“`vertices`” DataFrame

- 1 vertex per Row
- id: column with unique ID

id	City	State
“JFK”	“New York”	NY
“SEA”	“Seattle”	WA

Extra columns store vertex or edge data (a.k.a. attributes or properties).

“`edges`” DataFrame

- 1 edge per Row
- src, dst: columns using IDs from vertices.id

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

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(...)
```

In the future...

- SQL data sources for graph formats

Build and show a Graph in GraphFrames - an Example

```
> # Load & prepare vertices DataFrame  # Set File Paths tripdelaysFilePath = "/dat ...
> # Prepare edges DataFrame  # Build `departureDelays_geo` DataFrame #  Obtain key ...
```

```
> display(airports)
```

id	City	State	Country
LRD	Laredo	TX	USA
INL	International Falls	MN	USA
SAF	Santa Fe	NM	USA
MSO	Missoula	MT	USA
GRR	Grand Rapids	MI	USA

```
> from graphframes import *

  # Note that we already cached our Vertices and Edges
  tripGraph = GraphFrame(airports, departureDelays_geo)
> tripGraph.vertices
```

```
Out[6]: DataFrame[id: string, City: string, State: string, Country: string]
```

```
> # Airport and trip counts
  print "Airports: %d" % tripGraph.vertices.count()
  print "Trips: %d" %  tripGraph.edges.count()
```

```
Airports: 279
Trips: 1361141
```

Example of Simple Queries with GraphFrames

What trips are most likely to have significant delays?

```
display(tripGraph.edges
        .groupBy("src", "dst")
        .avg("delay")
        .sort(desc("avg(delay)")))
```

src	dst	avg(delay)
JFK	JAC	322
JAC	JFK	307
SYR	BTV	257
CRW	DTW	131

```
# States with the longest cumulative delays (with individual delays > 100 minutes) (origin: Seattle)
srcSeattleByState = tripGraph.edges.filter("src = 'SEA' and delay > 100")
display(srcSeattleByState)
```

```
display(tripGraph.degrees
        .sort(desc("degree"))
        .limit(10))
```

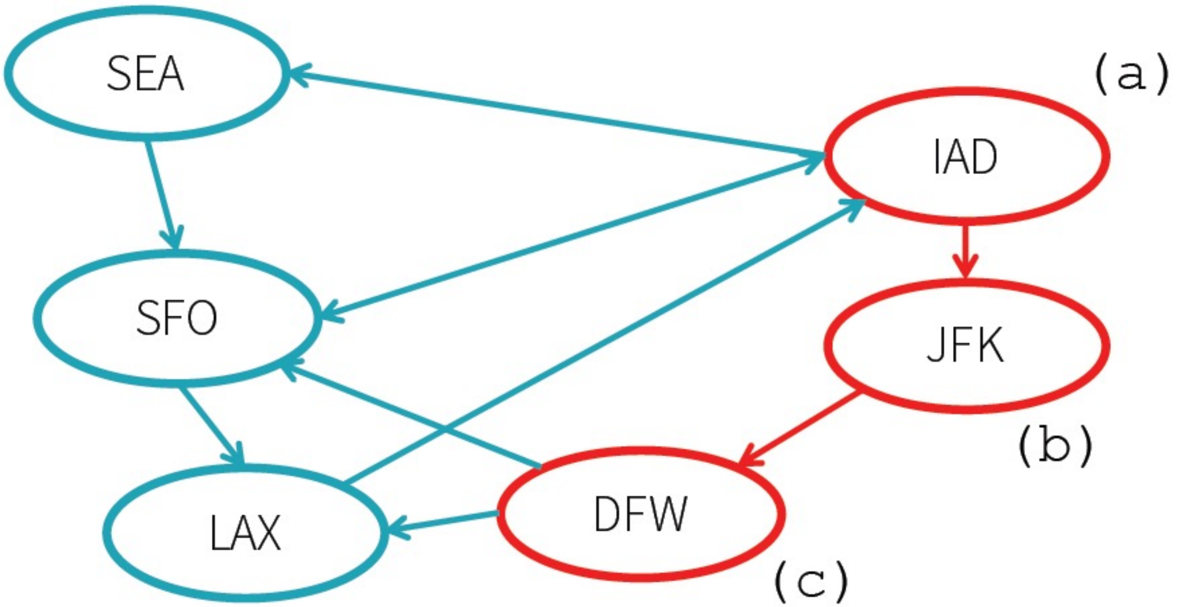
Motif Finding with GraphFrames

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")
```



E.G.: Study City/Flight Relationships via Motif-Finding

What delays might we blame on SFO?

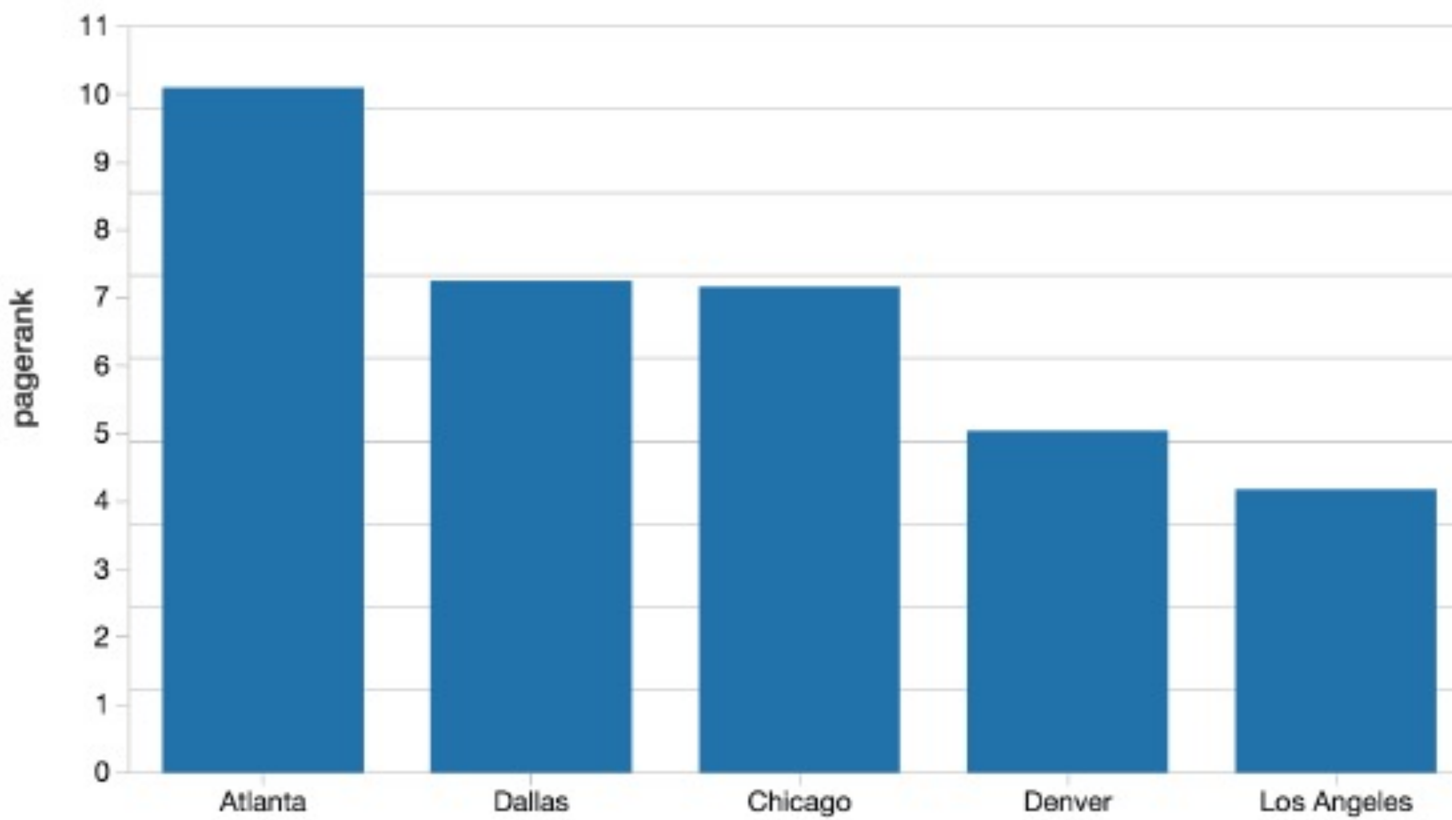
```
> motifs = tripGraph.find("(a)-[e1]->(b); (b)-[e2]->(c)")\
    .filter("(b.id = 'SFO') and (e1.delay > 500 or e2.delay > 500) and e1.tripid < e2.tripid")
  display(motifs)
```

e1	a
▸{"tripid":1011126,"localdate":"2014-01-01T11:26:00.000+0000","delay":-4,"distance":1421,"src":"IAH","dst":"SFO","city_src":"Houston","city_dst":"San Francisco","state_src":"TX","state_dst":"CA"}	▸{"id":"IAH","City
▸{"tripid":1011126,"localdate":"2014-01-01T11:26:00.000+0000","delay":-4,"distance":1421,"src":"IAH","dst":"SFO","city_src":"Houston","city_dst":"San Francisco","state_src":"TX","state_dst":"CA"}	▸{"id":"IAH","City
▸{"tripid":1011126,"localdate":"2014-01-01T11:26:00.000+0000","delay":-4,"distance":1421,"src":"IAH","dst":"SFO","city_src":"Houston","city_dst":"San Francisco","state_src":"TX","state_dst":"CA"}	▸{"id":"IAH","City
▸{"tripid":1011126,"localdate":"2014-01-01T11:26:00.000+0000","delay":-4,"distance":1421,"src":"IAH","dst":"SFO","city_src":"Houston","city_dst":"San Francisco","state_src":"TX","state_dst":"CA"}	▸{"id":"IAH","City

Showing the first 1000 rows.

E.G.: Determine Airport Importance via PageRank

```
ranks = tripGraph.pageRank(maxIter=5)
display(ranks.vertices
        .sort(ranks.vertices.pagerank.desc())
        .limit(10))
```



…

Built-in Graph Algorithms for GraphFrames

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

Built-in Algorithm Implementation for GraphFrames

Mostly wrappers for GraphX

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

Some algorithms implemented using DataFrames

- Breadth-first search
- Triangle counting

GraphX compatibility for GraphFrames

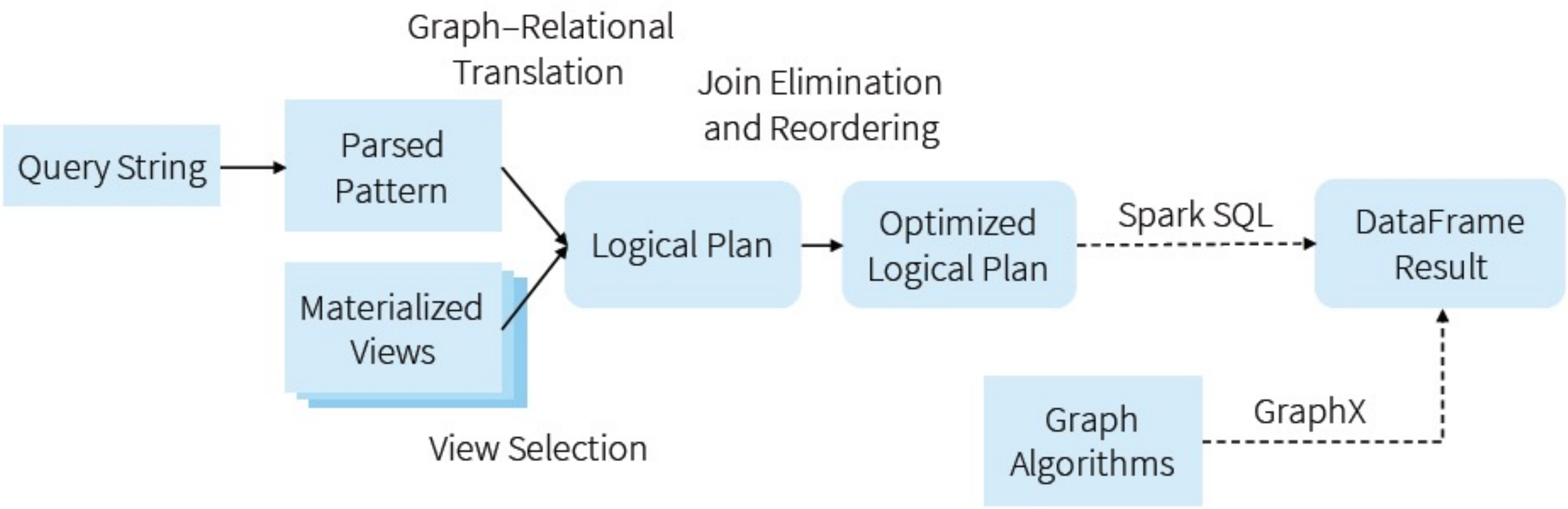
Simple conversions between GraphFrames & GraphX.

```
val g: GraphFrame = ...

// Convert GraphFrame → GraphX
val gx: Graph[Row, Row] = g.toGraphX

// Convert GraphX → GraphFrame
val g2: GraphFrame = GraphFrame.fromGraphX(gx)
```

Vertex & edge attributes are Rows in order to handle non-Long IDs

Wrapping existing GraphX code: See Belief Propagation example:
https://github.com/graphframes/graphframes/blob/master/src/main/scala/org/graphframes/examples/BeliefPropagation.scala

GraphFrames System Implementation



Resources for Learning more about GraphFrames

User guide + API docs http://graphframes.github.io/

- Quick-start
- Overview & examples for all algorithms
- Also available as executable notebooks:
  - Scala: http://go.databricks.com/hubfs/notebooks/3-GraphFrames-User-Guide-scala.html
  - Python: http://go.databricks.com/hubfs/notebooks/3-GraphFrames-User-Guide-python.html

Blog posts

- Intro: https://databricks.com/blog/2016/03/03/introducing-graphframes.html
- Flight delay analysis: https://databricks.com/blog/2016/03/16/on-time-flight-performance-with-spark-graphframes.html

https://www.datascience.com/blog/graph-computations-apache-spark

Another Graph Processing Framework for Spark:

Jiawei Jiang et al, "PSGraph: How Tencent trains extremely large graphs with Spark ?", IEEE ICDE 2020