GraphX: Unifying Data-Parallel and Graph-Parallel Analytics

Presented by Joseph Gonzalez

Joint work with Reynold Xin, Daniel Crankshaw, Ankur Dave, Michael Franklin, and Ion Stoica

Strata 2014

*These slides are best viewed in PowerPoint with animation.
Graphs are Central to Analytics

Raw Wikipedia -> Text Table

Discussion Table -> Editor Graph

Term-Doc Graph -> Community Detection

Hyperlinks -> PageRank

Topic Model (LDA) -> Word Topics

Top 20 Pages -> Community Topic

User Community -> User Com.
PageRank: Identifying Leaders

\[ R[i] = 0.15 + \sum_{j \in \text{Nbrs}(i)} w_{ji} R[j] \]

- **Rank of user** \( i \)
- **Weighted sum of neighbors’ ranks**

Update ranks in parallel
Iterate until convergence
The Graph-Parallel Pattern

Computation depends only on the **neighbors**
Many Graph-Parallel Algorithms

• Collaborative Filtering
  – Alternating Least Squares
  – Stochastic Gradient Descent
  – Tensor Factorization

• Structured Prediction
  – Loopy Belief Propagation
  – Max-Product Linear Programs
  – Gibbs Sampling

• Semi-supervised ML
  – Graph SSL
  – CoEM

• Community Detection
  – Triangle-Counting
  – K-core Decomposition
  – K-Truss

• Graph Analytics
  – PageRank
  – Personalized PageRank
  – Shortest Path
  – Graph Coloring

• Classification
  – Neural Networks
Graph-Parallel Systems

Exposé specialized APIs to simplify graph programming.

Exploit graph structure to achieve orders-of-magnitude performance gains over more general data-parallel systems.
PageRank on the Live-Journal Graph

GraphLab is 60x faster than Hadoop
GraphLab is 16x faster than Spark
Graphs are Central to Analytics

- Raw Wikipedia
  - XML
  - Table
- Text
  - Table
- Hyperlinks
- PageRank
- Top 20 Pages
- Term-Doc Graph
- Topic Model (LDA)
- Word Topics
- Discussion Table
  - User Disc.
- Editor Graph
- Community Detection
- User Community
- Community Topic
  - User Com.
  - Topic Com.
Separate Systems to Support Each View

Table View

Graph View

Table
Row
Row
Row
Row

Dependency Graph

Result

hadoop
Spark
Pregel
GraphLab
Apache Giraph
Having separate systems for each view is difficult to use and inefficient.
Difficult to Program and Use

Users must *Learn, Deploy, and Manage* multiple systems

Leads to brittle and often complex interfaces
Inefficient

Extensive data movement and duplication across the network and file system

Limited reuse internal data-structures across stages
Solution: The GraphX Unified Approach

New API
Blurs the distinction between Tables and Graphs

New System
Combines Data-Parallel Graph-Parallel Systems

Enabling users to easily and efficiently express the entire graph analytics pipeline
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.
View a Graph as a Table

Property Graph

Vertex Property Table

<table>
<thead>
<tr>
<th>Id</th>
<th>Property (V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rxin</td>
<td>(Stu., Berk.)</td>
</tr>
<tr>
<td>Jegonzal</td>
<td>(PstDoc, Berk.)</td>
</tr>
<tr>
<td>Franklin</td>
<td>(Prof., Berk)</td>
</tr>
<tr>
<td>Istoica</td>
<td>(Prof., Berk)</td>
</tr>
</tbody>
</table>

Edge Property Table

<table>
<thead>
<tr>
<th>SrcId</th>
<th>DstId</th>
<th>Property (E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rxin</td>
<td>jegonzal</td>
<td>Friend</td>
</tr>
<tr>
<td>franklin</td>
<td>rxin</td>
<td>Advisor</td>
</tr>
<tr>
<td>istoica</td>
<td>franklin</td>
<td>Coworker</td>
</tr>
<tr>
<td>franklin</td>
<td>jegonzal</td>
<td>PI</td>
</tr>
</tbody>
</table>
Table Operators

Table (RDD) operators are inherited from Spark:

<table>
<thead>
<tr>
<th>Map</th>
<th>Reduce</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td>Count</td>
<td>Take</td>
</tr>
<tr>
<td>GroupBy</td>
<td>Fold</td>
<td>First</td>
</tr>
<tr>
<td>Sort</td>
<td>ReduceByKey</td>
<td>PartitionBy</td>
</tr>
<tr>
<td>Union</td>
<td>GroupByKey</td>
<td>MapWith</td>
</tr>
<tr>
<td>Join</td>
<td>Cogroup</td>
<td>Pipe</td>
</tr>
<tr>
<td>LeftOuterJoin</td>
<td>Cross</td>
<td>Save</td>
</tr>
<tr>
<td>RightOuterJoin</td>
<td>Zip</td>
<td>...</td>
</tr>
</tbody>
</table>
Graph Operators

class Graph [ V, E ] {
    def Graph(vertices: Table[ (Id, V) ],
               edges: Table[ (Id, Id, E) ])

    // Table Views ----------------------
    def vertices: Table[ (Id, V) ]
    def edges: Table[ (Id, Id, E) ]
    def triplets: Table[ ((Id, V), (Id, V), E) ]

    // Transformations -----------------
    def reverse: Graph[V, E]
    def subgraph(pV: (Id, V) => Boolean,
                  pE: Edge[V,E] => Boolean): Graph[V,E]
    def mapV(m: (Id, V) => T): Graph[T,E]
    def mapE(m: Edge[V,E] => T): Graph[V,T]

    // Joins --------------------------
    def joinV(tbl: Table [(Id, T)]: Graph[(V, T), E ]
    def joinE(tbl: Table [(Id, Id, T)]: Graph[V, (E, T)]

    // Computation ---------------------
    def mrTriplets(mapF: (Edge[V,E]) => List[(Id, T)],
                   reduceF: (T, T) => T): Graph[T, E]
}
Triplets Join Vertices and Edges

The \textit{triplets} operator joins vertices and edges:

\begin{align*}
\text{SELECT} & \ t.\text{dstId}, \ \text{reduceUDF}( \ \text{mapUDF}(t) \ ) \ \text{AS sum} \\
\text{FROM} & \ \text{triplets AS t} \ \text{GROUPBY} \ t.\text{dstId}
\end{align*}
Map Reduce Triplets

Map-Reduce for each vertex

mapF( ) \rightarrow A_1

mapF( ) \rightarrow A_2

reduceF( A_1, A_2 ) \rightarrow A
Example: Oldest Follower

What is the age of the oldest follower for each user?

```scala
val oldestFollowerAge = graph
  .mrTriplets(
    e => (e.dst.id, e.src.age), //Map
    (a, b) => max(a, b) //Reduce
  )
  .vertices
```
We express 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.
DIY Demo this Afternoon

2. Introduction to the GraphX API

To get started you first need to import GraphX. Start the Spark-Shell (by running the following on the root node):

```
/spark/bin/spark shell
```

and paste the following in your Spark shell:

```scala
import org.apache.spark.graphx._
import org.apache.spark.graphxRDD
```

2.1. The Property Graph

The property graph is a directed multigraph (a directed graph with potentially multiple parallel edges sharing the same source and destination vertex) with properties attached to each vertex and edge. Each vertex is keyed by a unique 64-bit long Identifier (vertexId). Similarly, edges have corresponding source and destination vertex identifiers. The properties are stored as Scala/Java objects with each edge and vertex in the graph.

Throughout the first half of this tutorial we will use the following toy property graph. While this is hardly big data, it provides an opportunity to learn about the graph data model and the GraphX API. In this example we have a small social network with users and their ages modeled as vertices and likes modeled as directed edges.

We begin by creating the property graph from arrays of vertices and edges. Later we will demonstrate how to load real data. Paste the following code into the spark shell.

```scala
val vertexArray = Array(
  (1L, "Alice", 23),
  (2L, "Bob", 27),
  (3L, "Charlie", 60),
  (4L, "David", 42),
  (5L, "Eve", 55),
  (6L, "Fred", 35))
```
GraphX System Design
Distributed Graphs as Tables (RDDs)

Property Graph

2D Vertex Cut Heuristic

2D Vertex Cut Heuristic

Part. 1

Part. 2

Vertex Table (RDD)

Routing Table (RDD)

Edge Table (RDD)
Caching for Iterative mrTriplets

Vertex Table (RDD)

A
B
C
D
E
F

Edge Table (RDD)

A
B
C
D

Mirror Cache

A
B
C
D

Mirror Cache

A
B
C
D

A
E
F

A
F
E
D

A
F
E
D
Incremental Updates for Iterative mrTriplets

Vertex Table (RDD)

Edge Table (RDD)

Change

Mirror Cache

Scan
Aggregation for Iterative mrTriplets

- **Vertex Table (RDD)**
  - Change
    - A
    - B
    - C
    - D
    - E
    - F

- **Edge Table (RDD)**
  - A
  - B
  - C
  - D
  - E
  - F
  - Mirror Cache

- **Local Aggregate**
  - Scan
  - Local Aggregate

- **Change**
  - Change
  - Change
  - Change
  - Change
  - Change
  - Change
Reduction in Communication Due to Cached Updates

Connected Components on Twitter Graph

Most vertices are within 8 hops of all vertices in their comp.
Benefit of Indexing Active Edges

Connected Components on Twitter Graph

- Scan
- Indexed

Scan All Edges
Index of “Active” Edges
Join Elimination

Identify and bypass joins for unused triplets fields

Example: PageRank only accesses source attribute

PageRank on Twitter

Factor of 2 reduction in communication
Additional Query Optimizations

Indexing and Bitmaps:
» To accelerate joins across graphs
» To efficiently construct sub-graphs

Substantial Index and Data Reuse:
» Reuse routing tables across graphs and sub-graphs
» Reuse edge adjacency information and indices
Performance Comparisons

Live-Journal: 69 Million Edges

Runtime (in seconds, PageRank for 10 iterations)

GraphX is roughly **3x slower** than GraphLab
GraphX scales to larger graphs

Twitter Graph: 1.5 Billion Edges

<table>
<thead>
<tr>
<th>System</th>
<th>Runtime (in seconds, PageRank for 10 iterations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Giraph</td>
<td>749</td>
</tr>
<tr>
<td>GraphX</td>
<td>451</td>
</tr>
<tr>
<td>GraphLab</td>
<td>203</td>
</tr>
</tbody>
</table>

GraphX is roughly 2x slower than GraphLab

» Scala + Java overhead: Lambdas, GC time, ...
» No shared memory parallelism: 2x increase in comm.
PageRank is just one stage....

What about a pipeline?
Timed end-to-end GraphX is faster than GraphLab.
The GraphX Stack
(Lines of Code)

- PageRank (5)
- Connected Comp. (10)
- Shortest Path (10)
- SVD (40)
- ALS (40)
- K-core (51)
- Triangle Count (45)
- LDA (120)

Pregel (28) + GraphLab (50)

GraphX (3575)

Spark
Status

Alpha release as part of Spark 0.9

Seeking collaborators and feedback
Conclusion 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
Active Research

Static Data ➔ Dynamic Data
  » Apply GraphX unified approach to time evolving data
  » Model and analyze relationships over time

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

http://amplab.github.io/graphx/
ankurd@eecs.berkeley.edu
crankshaw@eecs.berkeley.edu
rxin@eecs.berkeley.edu
jegonzal@eecs.berkeley.edu
Graph Property 1
Real-World Graphs

Power-Law Degree Distribution

More than $10^8$ vertices have one neighbor.

Top 1% of vertices are adjacent to 50% of the edges!

Edges $>>$ Vertices
Graph Property 2
Active Vertices

PageRank on Web Graph

51% updated only once!
Graphs are Essential to Data Mining and Machine Learning

Identify influential people and information
Find communities
Understand people’s shared interests
Model complex data dependencies
Recommending Products

Users

Ratings

Items

- Bicycle
- Coffee
- Duck
- Camera
- Shoe
Recommending Products

Low-Rank Matrix Factorization:

Iterate:

\[ f[i] = \arg \min_{w \in \mathbb{R}^d} \sum_{j \in \text{Nbrs}(i)} (r_{ij} - w^T f[j])^2 + \lambda ||w||^2_2 \]
Predicting User Behavior

Conditional Random Field
Belief Propagation
Finding Communities

Count triangles passing through each vertex:

Measures “cohesiveness” of local community

Fewer Triangles
Weaker Community

More Triangles
Stronger Community
Example Graph Analytics Pipeline

Preprocessing

Raw Data

ETL

Initial Graph

Slice

Subgraph

Compute

PageRank

Post Proc.

GraphLab

XML

Hive

Analyze

Top Users

Repeat