GraphX: Unifying Table and Graph Analytics

Presented by Joseph Gonzalez

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

IPDPS 2014

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

Raw Wikipedia

Text Table

Term-Doc Graph

Hyperlinks

Topic Model
(LDA)

PageRank

Top 20 Pages

Discussion Table

Editor Graph

Community Detection

User Community

User|Com.

Topic|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
Mean Field Algorithm

\[ f(x_i, b_j) = \sum_{x_j} b_j(x_j) \log \phi_{i,j}(x_i, x_j) \]

\[ b_i(x_i) \propto \phi_i(x_i) \exp \left( \sum_{j \in N_i} f(x_i, b_j) \right) \]

\[ \phi_1(x_1) \quad \phi_2(x_2) \quad \phi_3(x_3) \]

\[ \phi_{1,2}(x_1, x_2) \quad \phi_{2,3}(x_2, x_3) \]

\[ \sum \text{ over Neighbors} \]
Recommending Products

Low-Rank Matrix Factorization:

Iterate:

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

SOCIAL NETWORK ANALYSIS

GRAPH ALGORITHMS
Graph-Parallel Systems

Expose *specialized APIs* to simplify graph programming.
“Think like a Vertex.”
- Pregel [SIGMOD’10]
The Pregel (Push) Abstraction

Vertex-Programs interact by sending messages.

\[
\text{Pregel\_PageRank}(i, \text{messages}) : \\
// Receive all the messages \\
total = 0 \\
\text{foreach} (\text{msg in messages}) : \\
\quad total = total + msg \\
// Update the rank of this vertex \\
R[i] = 0.15 + total \\
// Send new messages to neighbors \\
\text{foreach}(j \text{ in out\_neighbors}[i]) : \\
\quad \text{Send \ msg}(R[i]) \text{ to vertex } j
\]

Malewicz et al. [PODC’09, SIGMOD’10]
The GraphLab (Pull) Abstraction

Vertex Programs directly **access** adjacent vertices and edges

```python
GraphLab_PageRank(i):
    // Compute sum over neighbors
    total = 0
    foreach (j in neighbors(i)):
        total = total + R[j] * w_{ji}
    // Update the PageRank
    R[i] = 0.15 + total
```

Data movement is managed by the system and not the user.
Iterative Bulk Synchronous Execution

Compute

Communicate

Barrier
Graph-Parallel Systems

Expose *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

Spark is 4x faster than Hadoop
GraphLab is 16x faster than Spark
Triangle Counting on Twitter
40M Users, 1.4 Billion Links

Counted: 34.8 Billion Triangles

Hadoop
[WWW’11]
1536 Machines
423 Minutes

GraphLab
64 Machines
15 Seconds

1000 x Faster

S. Suri and S. Vassilvitskii, “Counting triangles and the curse of the last reducer,” WWW’11
Graph Analytics Pipeline

Raw Wikipedia

XML

Hyperlinks

PageRank

Top 20 Pages

PageRank

Com. User

Community Detection

Editor Graph

Term-Doc Graph

Discussion Table

User Disc.
Tables

Raw Wikipedia

Text Table

Hyperlinks

PageRank

Top 20 Pages

Term-Doc Graph

Topic Model (LDA)

Word Topics

Discussion Table

Editor Graph

Community Detection

User Community

Community Topic

Term-Doc Graph

Discussion Table

Editor Graph

Community Detection

User Community

Community Topic
Graphs

Raw Wikipedia → XML

Text Table → Term-Doc Graph

Hyperlinks → PageRank

Top 20 Pages

Topic Model (LDA)

Word Topics

Community Detection

Editor Graph

Discussion Table

User Community

Table

User Com.
Separate Systems to Support Each View

Table View

Graph View

Table
Row
Row
Row
Row

Result

Dependency Graph
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
GraphX Solution: Tables and Graphs are views of the same physical data.

Each view has its own operators that exploit the semantics of the view to achieve efficient execution.
Graphs $\rightarrow$ Relational Algebra

1. Encode graphs as distributed tables
2. Express graph computation in relational algebra
3. Recast graph systems optimizations as:
   1. Distributed join optimization
   2. Incremental materialized maintenance

Integrate Graph and Table data processing systems.
Achieve performance parity with specialized systems.
Distributed Graphs as Distributed Tables

Property Graph

Vertex Table

Routing Table

Edge Table

2D Vertex Cut Heuristic
Table Operators

Table operators are inherited from Spark:

<table>
<thead>
<tr>
<th>Table Operator</th>
<th>Table Operator</th>
<th>Table Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>reduce</td>
<td>sample</td>
</tr>
<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>
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 *triplets* operator joins vertices and edges:

\[
\text{SELECT } t.\text{dstId}, \text{reduce( map}(t)) \text{ AS sum}
\text{FROM triplets AS t GROUPBY t.\text{dstId}}
\]

The *mrTriplets* operator sums adjacent triplets.
Example: Oldest Follower

Calculate the number of older followers for each user?

```scala
def olderFollowerAge = graph
  .mrTriplets{
    e => // Map
      if(e.src.age < e.dst.age) {
        (e.srcId, 1)
      } else { Empty }
  ,
  (a,b) => a + b // Reduce
  }
  .vertices
```
We express enhanced Pregel and GraphLab abstractions using the GraphX operators in less than 50 lines of code!
**Enhanced Pregel in GraphX**

```java
pregelPR(i, messageSum):
    // Receive all the messages
    total = 0
    foreach (msg in messageList):
        total = total + msg

    // Update the rank of this vertex
    R[i] = 0.15 + total

combineMsg(a, b):
    // Compute sum of two messages
    return a + b

sendMsg(i, j, R[i], R[j], E[i, j]):
    // Compute single message
    Send msg(R[i]/E[i, j]) to vertex
```

- **Require Message Combiners**
- **Remove Message Computation from the Vertex Program**
PageRank in GraphX

// Load and initialize the graph
val graph = GraphBuilder.text("hdfs://web.txt")
val prGraph = graph.joinVertices(graph.outDegrees)

// Implement and Run PageRank
val pageRank =
    prGraph.pregel(initialMessage = 0.0, iter = 10)(
        (oldV, msgSum) => 0.15 + 0.85 * msgSum,
        triplet => triplet.src.pr / triplet.src.deg,
        (msgA, msgB) => msgA + msgB)
Join Elimination

Identify and bypass joins for unused triplet fields

```
sendMsg(i→j, R[i], R[j], E[i, j]):
  // Compute single message
  return msg(R[i]/E[i, j])
```

PageRank on Twitter

- Three Way Join
- Join Elimination

Factor of 2 reduction in communication
We express the Pregel and GraphLab \emph{like} abstractions using the GraphX \emph{operators} in less than 50 lines of code!

By composing these operators we can construct \emph{entire graph-analytics pipelines}. 
Example Analytics Pipeline

// Load raw data tables
val verts = sc.textFile("hdfs://users.txt").map(parserV)
val edges = sc.textFile("hdfs://follow.txt").map(parserE)
// Build the graph from tables and restrict to recent links
val graph = new Graph(verts, edges)
val recent = graph.subgraph(edge => edge.date > LAST_MONTH)
// Run PageRank Algorithm
val pr = graph.PageRank(tol = 1.0e-5)
// Extract and print the top 25 users
val topUsers = verts.join(pr).top(25).collect
topUsers.foreach(u => println(u.name + \\
\t' + u.pr))
The GraphX Stack
(Lines of Code)

GraphX (3575)
Spark

Pregel (28) + GraphLab (50)

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

LDA

ALS

K-core

SVD

Shortest Path

Connected Comp.

PageRank

GraphX Stack
(Lines of Code)
GraphX is roughly \textit{3x slower} than GraphLab.
GraphX scales to larger graphs

Twitter Graph: 1.5 Billion Edges

- Giraph: 749 seconds
- GraphX: 451 seconds
- GraphLab: 203 seconds

Runtime (in seconds, PageRank for 10 iterations)

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?
A Small Pipeline in GraphX

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

Part of Apache Spark

In production at several large technology companies
GraphX: Unified Analytics

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
A Case for Algebra in Graphs

A standard algebra is essential for graph systems:
- e.g.: SQL \(\rightarrow\) proliferation of relational system

By embedding graphs in *relational algebra*:
- Integration with tables and preprocessing
- Leverage advances in relational systems
- Graph opt. recast to relational systems opt.
Conclusions

Composable domain specific views and operators

Single system that efficiently spans the pipeline

Graphs through the lens of database systems
  » Graph-Parallel Pattern → Triplet joins in relational alg.
  » Graph Systems → Distributed join optimizations

Joseph E. Gonzalez
jegonzal@eecs.berkeley.edu
http://tinyurl.com/ampgraphx
Thanks!

http://amplab.cs.berkeley.edu/projects/graphx/

ankurd@eecs.berkeley.edu
crankshaw@eecs.berkeley.edu
rxin@eecs.berkeley.edu
jegonzal@eecs.berkeley.edu
Recommending Products

Low-Rank Matrix Factorization:

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

\[ f(x_i, b_j) = \sum_{x_j} b_j(x_j) \log \phi_{i,j}(x_i, x_j) \]
GraphX System Design
Caching for Iterative mrTriplets

Vertex Table (RDD)

Edge Table (RDD)

Mirror Cache

A
B
C
D

Mirror Cache

A
B
C
D
E
F
Incremental Updates for Iterative mrTriplets

Vertex Table (RDD)

- Change
  - A
  - B
  - C

- Change
  - D
  - E
  - F

Edge Table (RDD)

Mirror Cache

- A
- B
- C
- D

- A
- B
- C
- D

Scan
Aggregation for Iterative mrTriplets

**Vertex Table (RDD)**

- **A**
- **B**
- **C**
- **D**
- **E**
- **F**

**Edge Table (RDD)**

- **A**
- **B**
- **C**
- **D**
- **E**
- **F**

**Mirror Cache**

- **A**
- **B**
- **C**
- **D**
- **E**
- **F**

**Local Aggregate**

**Scan**
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**

- **Runtime (Seconds)**
- **Iteration**
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