Graph Processing Using Spark's GraphX

Diego Rivera Correa
CS6240

Agenda

- Motivation
- GraphX’s Precursor
- GraphX!

Why Graphs?

Motivation: Graphs, Graphs, Graphs!

Graphs are general yet essential structures that allow us to model many real-world problems.

Motivation: Graphs, Graphs, Graphs!

Graphs are general yet essential structures that allow us to model many real-world problems.

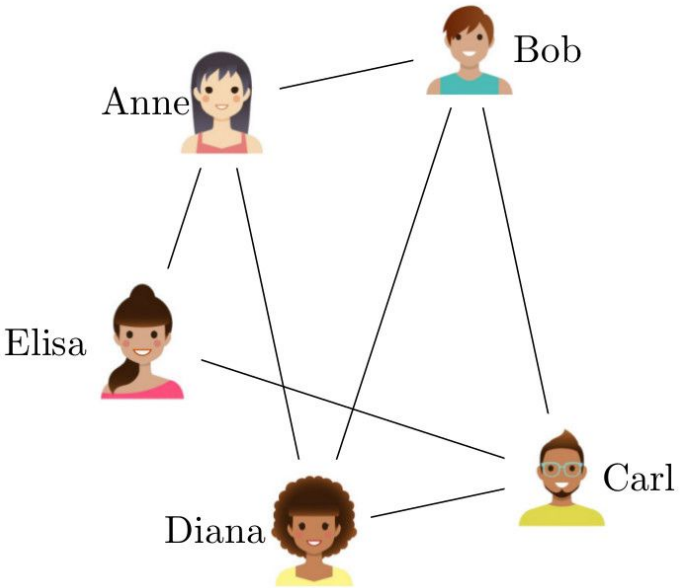
Let's take a look a few examples:

Motivation: Graphs, Graphs, Graphs!

Graphs are general yet essential structures that allow us to model many real-world problems.

Let's take a look a few examples:

- Social Networks (find friends of friends)

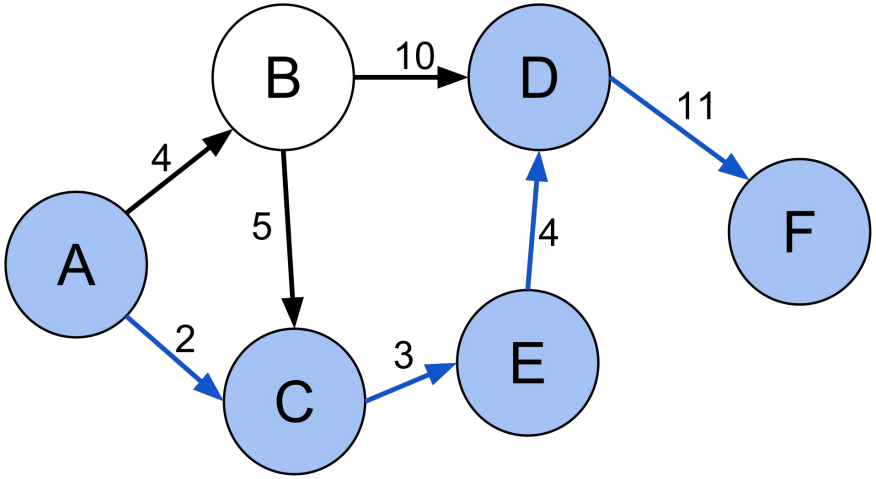


Motivation: Graphs, Graphs, Graphs!

Graphs are general yet essential structures that allow us to model many real-world problems.

Let's take a look a few examples:

- Social Networks (find friends of friends)
- Transportation Networks (shortest paths)

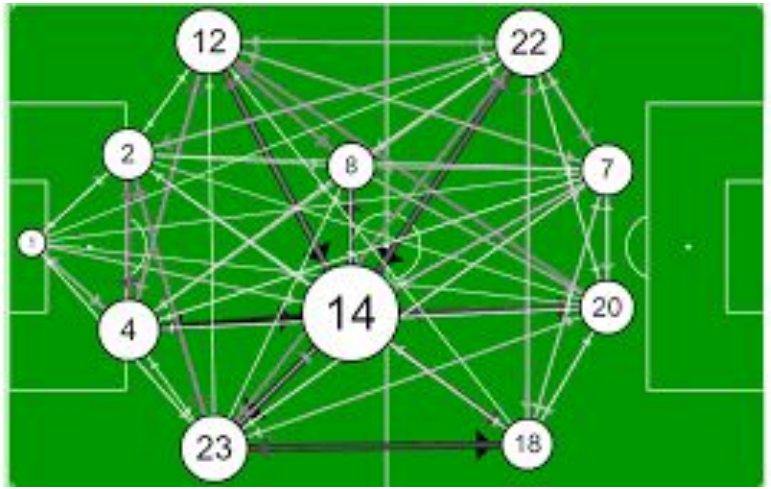


Motivation: Graphs, Graphs, Graphs!

Graphs are general yet essential structures that allow us to model many real-world problems.

Let's take a look a few examples:

- Social Networks (find friends of friends)
- Transportation Networks (shortest paths)
- Sports Analytics (pass movement)



Motivation: Graphs, Graphs, Graphs!

The beauty of graphs lies in its generic structure that is extendable to many problems.

Motivation: Graphs, Graphs, Graphs!

The beauty of graphs lies in its generic structure that is extendable to many problems.

Motivation: Graphs, Graphs, Graphs!

The beauty of graphs lies in its generic structure that is extendable to many problems.

This notion of extendability comes with 2 challenges: (1) Many graph problems can become prohibitively large and (2) Many graph problems are inherently complex.

> This means that this type of problem is ideal for distributed computing!

History of Graph Processing

Origins: NetworkX

Before GraphX (or any distributed framework), graph processing was done at the single-machine level.

It started in the early 2000's when NetworkX was born! However, it had some major setbacks:

- Did not scale well :(
- Was used for less intensive tasks: 2-hops, lookups, etc.

Boom! Big Data is Here!

Similarly in the early 2000s, with the expansion of web traffic and online stores, companies such as Amazon and eBay started to analyze EVERYTHING:

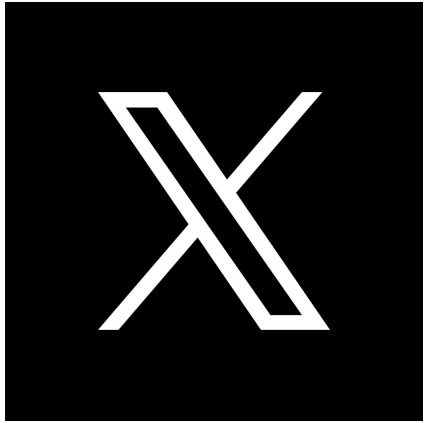
- click-rates, location data, search logs, etc.

In 2005, *Big Data* was coined by Roger Mougalas, referring to a large data that was almost impossible to manage using traditional single-machine tools.

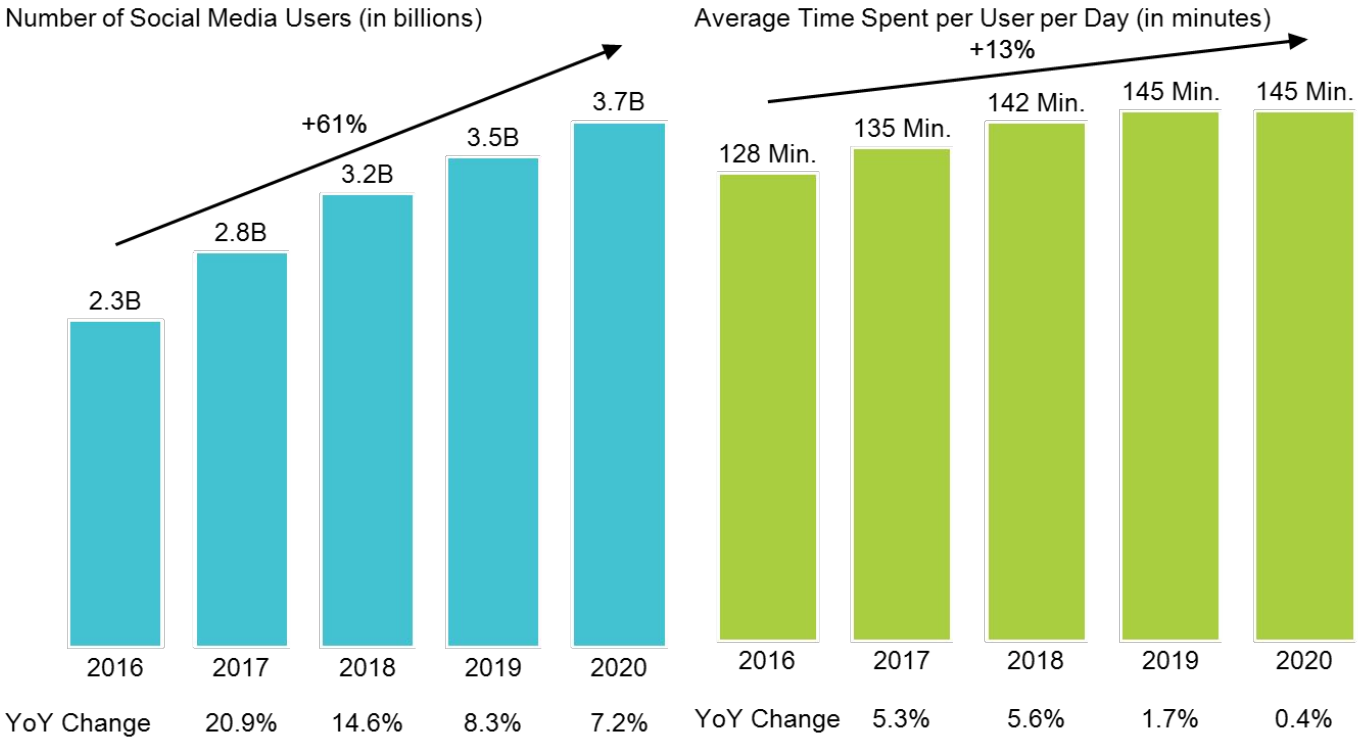
Boom! Big *Graph* Data is Here!

On a similar note, with the rise of social networks, the same problems transferred to the graph-processing world.

These graphs generated massive amounts of user data through posts, comments, likes, and shares, creating a need for new tools and techniques to analyze this large volume of unstructured data.



Big *Graph* Data!



The Rise of Distributed Graph Processing

Early efforts began in 2009 when **Pregel (Google)** was born. It focused on large-scale, distributed graph processing.

It introduced the **vertex-centric programming model**, where computation is performed from the perspective of each vertex.

Pregel: A System for Large-Scale Graph Processing

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn,
Naty Leiser, and Grzegorz Czajkowski
Google, Inc.
{malewicz,austern,ajcbik,dehnert,ilan,naty,gczaj}@google.com

ABSTRACT

Many practical computing problems concern large graphs. Standard examples include the Web graph and various social networks. The scale of these graphs—in some cases billions of vertices, trillions of edges—poses challenges to their efficient processing. In this paper we present a computa-

disease outbreaks, or citation relationships among published scientific work—have been processed for decades. Frequently applied algorithms include shortest paths computations, different flavors of clustering, and variations on the page rank theme. There are many other graph computing problems of practical value, *e.g.*, minimum cut and connected compo-

Vertex-Centric Programming Model

In a single machine environment, developers can easily implement graph analytics algorithms as they have a global view of the graph.

When the size of graph data grows beyond the memory capacity of a single machine, graph data must be partitioned to distributed memory.

A vertex-centric model is an intuitive way to process graphs where each vertex becomes the "center of computation".

Vertex-Centric Programming Model

The idea is that each vertex in the graph is treated as an independent computational unit. Computation occurs locally on each vertex, using information about its neighbors (e.g., adjacent vertices).

Communication between vertices occurs through message passing. Vertices send and receive messages to and from their neighbors.

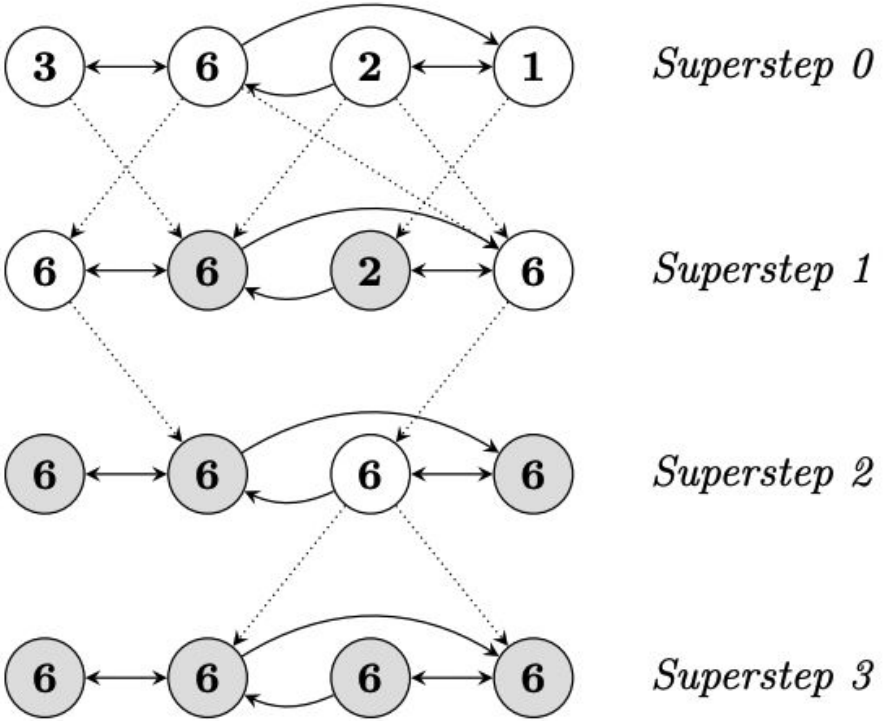
The computation proceeds in a series of iterations until a global convergence condition is met. These are called **supersteps**.

Vertex-Centric Programming Model by Example

Each superstep is an iteration.
Assume we want to compute the largest value in a graph.

Grey node = “Inactive”

White node = “Active”



Vertex-Centric Programming Model

Does anyone see any challenges in this model?

Vertex-Centric Programming Model

Does anyone see any challenges in this model?

- **Scalability:** Processing large-scale graphs in a distributed environment can lead to significant communication overhead due to frequent message passing between vertices.

Vertex-Centric Programming Model

Does anyone see any challenges in this model?

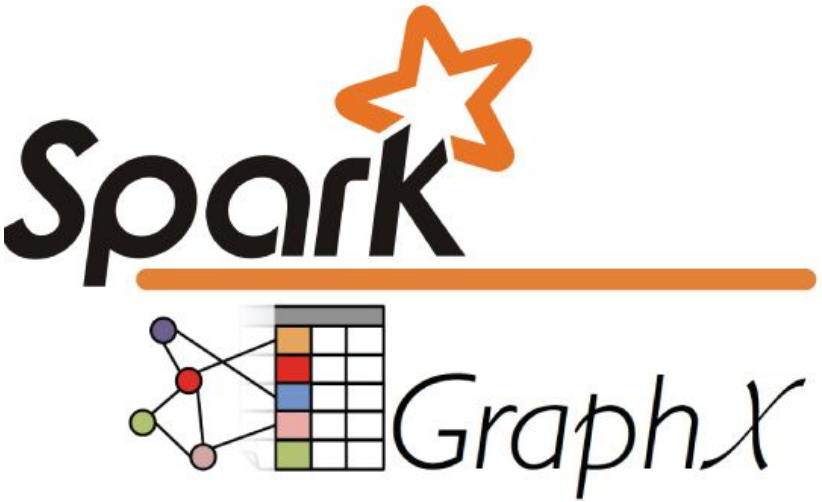
- **Scalability:** Processing large-scale graphs in a distributed environment can lead to significant communication overhead due to frequent message passing between vertices.

- **Performance Bottlenecks:** The overhead of iterative computation and communication can lead to slow convergence for algorithms that require many iterations.

Vertex-Centric Programming Model

Luckily, GraphX handles most of these challenges. Let’s see how…

GraphX



What is GraphX?

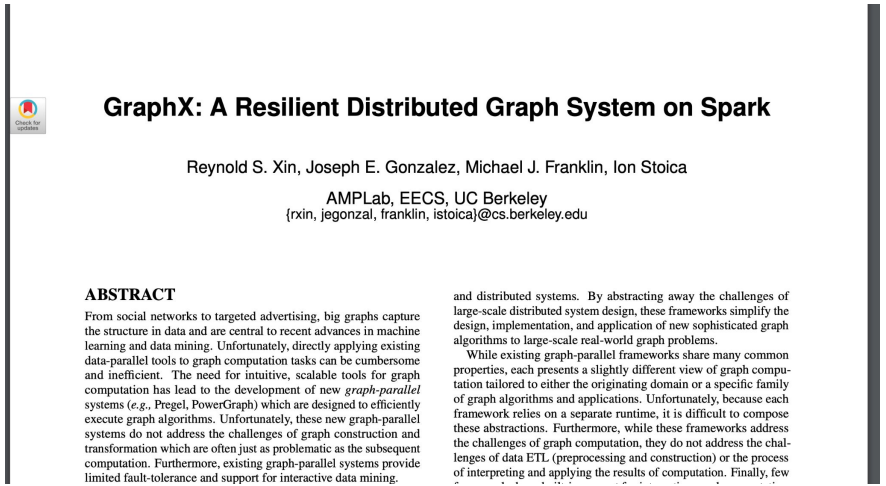
GraphX is one of the libraries built on top of Spark. It extends the Spark RDDs to support graph computations and analysis. GraphX introduces the concept of graphs as a **property graphs**.

What is GraphX?

It was developed by researchers from the AMPLab at UC-Berkeley.

Spark was also developed here!

It was first ‘revealed’ in 2013 in the GRADES workshop, and further improved in 2014 before being donated to the Apache Software Foundation.



GraphX: A Resilient Distributed Graph System on Spark

Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica

AMPLab, EECS, UC Berkeley
{rxin, jegonzal, franklin, istoica}@cs.berkeley.edu

ABSTRACT

From social networks to targeted advertising, big graphs capture the structure in data and are central to recent advances in machine learning and data mining. Unfortunately, directly applying existing data-parallel tools to graph computation tasks can be cumbersome and inefficient. The need for intuitive, scalable tools for graph computation has lead to the development of new *graph-parallel* systems (*e.g.,* Pregel, PowerGraph) which are designed to efficiently execute graph algorithms. Unfortunately, these new graph-parallel systems do not address the challenges of graph construction and transformation which are often just as problematic as the subsequent computation. Furthermore, existing graph-parallel systems provide limited fault-tolerance and support for interactive data mining.

and distributed systems. By abstracting away the challenges of large-scale distributed system design, these frameworks simplify the design, implementation, and application of new sophisticated graph algorithms to large-scale real-world graph problems.

While existing graph-parallel frameworks share many common properties, each presents a slightly different view of graph computation tailored to either the originating domain or a specific family of graph algorithms and applications. Unfortunately, because each framework relies on a separate runtime, it is difficult to compose these abstractions. Furthermore, while these frameworks address the challenges of graph computation, they do not address the challenges of data ETL (preprocessing and construction) or the process of interpreting and applying the results of computation. Finally, few

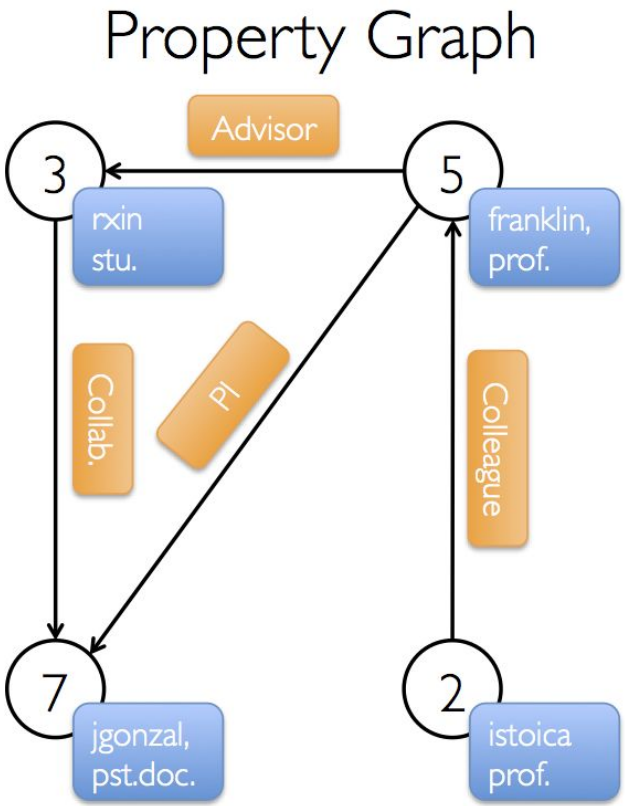
The Property Graph

The Property Graph is directed multigraph where each vertex and edge have user defined objects attached to it.

A directed multigraph is a directed graph with potentially multiple edges sharing the same source and destination vertex.

What is important about this model is its ability to support parallel edges simplifies modeling scenarios where there can be multiple relationships (e.g., co-worker and friend) between the same vertices.

Property Graph by Example



Abstractions

The main components of the graph are VertexRDD and EdgeRDD:

VertexRDD[(VertexId, O)]

VertexId = Unique ID (Long Int)

O = Object

Example:

(1L, (“Jayson Tatum”, “SF”))

(2L, (“Joe Mazzulla”, “HC”))

EdgeRDD[((From, To), O)]

From, To = VertexId

O = Object

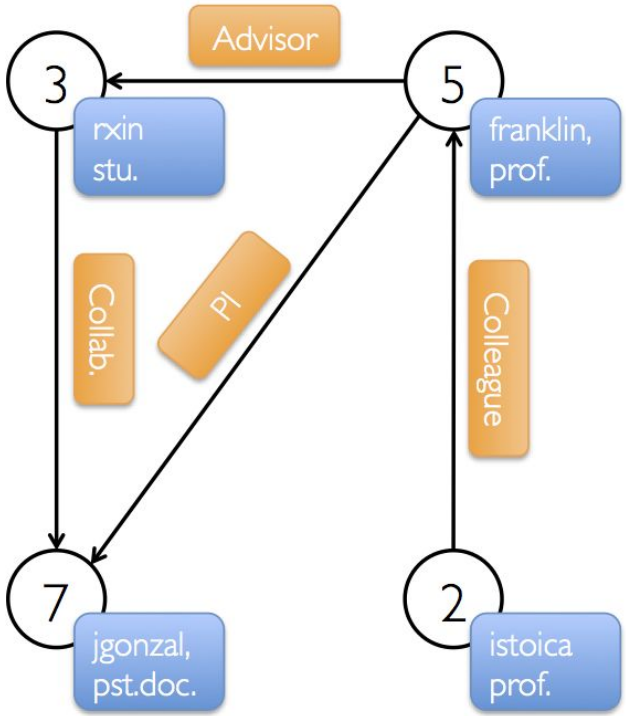
Example:

((2L, 1L), “Coaches”)

((1L, 2L), “Reports To”)

Abstractions

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Abstractions

Like RDDs, property graphs are immutable, distributed, and fault-tolerant.

Changes to the values or structure of the graph are accomplished by producing a new graph or RDDs with the desired changes.

Parallelism and Graphs

What Must We Partition? A large graph must be divided across multiple machines.

But vertices and edges are connected, so partitioning creates a choice:

- Allow edges to cross machines
- Allow vertices to appear on multiple machines

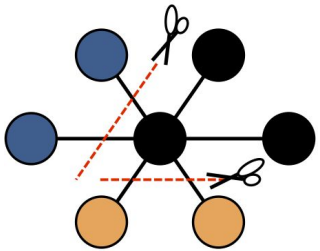
Partitioning determines which communication cost we pay.

“Cutting” a Graph: Edges

Edge-cut assigns every vertex to one machine.

An edge is cut when its endpoints are on different machines. Cut edges may require network communication.

Goal: balance computation while minimizing crossing edges.



Edge Cut

“Cutting” a Graph: Edges

Suppose we divide the vertices among machines:

- Machine 1: A, B, C
- Machine 2: D, E, F

The edge B → C is local. An edge such as B → E crosses the partition boundary.

“Cutting” a Graph: Edges

Whenever the algorithm needs information from both endpoints, the machines may need to communicate.

Objective is approximately: “*Balance the work while minimizing the number of edges whose endpoints are on different machines.*”

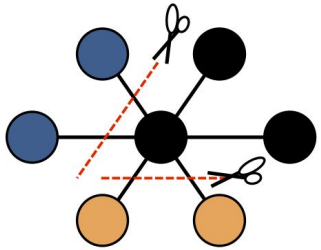
- Advantage = each vertex has exactly one authoritative location.
- Disadvantage = high-degree vertices can create both communication and load imbalance.

Graph Partitioning: Edge-Cut Approach

How do I determine in a principled way which edge(s) to cut?

When an edge is cut, communication between those machines is required whenever data or computation involving that edge is needed.

This is where overhead comes in: cutting more edges results in more data being transferred between machines.

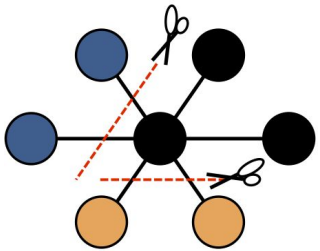


Edge Cut

Edge-Cuts in a Graph

Consider a celebrity with millions of followers.

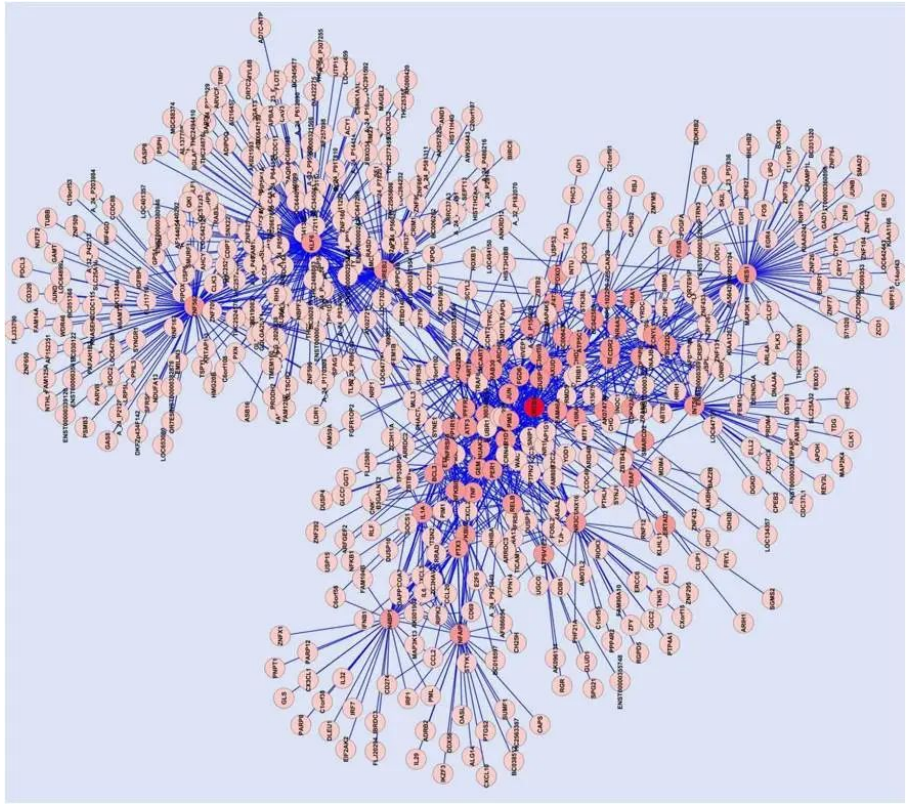
- Assigning the same number of vertices to each machine does not balance the work.
- The celebrity's machine may process millions of incident edges.
- Many of those edges may also cross machine boundaries.



Edge Cut

Balanced vertices != balanced edge work.

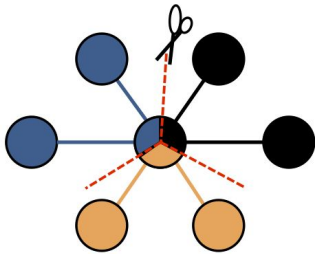
Challenge: Find the Optimal Edge-Cut!



Graph Partitioning: Vertex-Cut Approach

Spark said: “*Why not cut vertices instead?*”

In contrast to edge-cuts, vertex-cuts evenly assign edges to machines and allow vertices to span multiple machines.



Vertex Cut

The communication overhead of a vertex-cut is directly proportional to total number of machines spanned by each vertex.

Graph Partitioning: Vertex-Cut Approach

Suppose we divide the vertices among machines:

- Machine 1: A → B, A → C
- Machine 2: A → D, A → E

Vertex A appears in both machines. It is therefore replicated. Most edge processing is local, but GraphX must:

- Combine partial results computed by different replicas
- Update the authoritative vertex value
- Synchronize the new value with replicas that need it

Graph Partitioning: Vertex-Cut Approach

Therefore, the objective becomes: "*Balance the edges while minimizing how many partitions each vertex spans.*"

This is typically measures through a replication factor. This is computed as:

$$\frac{\sum_v \text{number of partitions containing } v}{|V|}$$

Value of 1 means no replication. Larger values means more synchronization overhead.

Hub Vertices

Suppose a social-network graph contains one celebrity vertex C with eight followers. Assume we have 4 machines.

- Hub C is followed by L1…L8

In an edge-cut approach, we assign each vertex to exactly one machine:

- M1: C, L1, L2
- M2: L3, L4
- M3: L5, L6
- M4: L7, L8

The edges from C to L1 and L2 are local. The other six edges cross machine boundaries.

Hub Vertices

We have:

- Local edges: 2
- Cross-machine edges: 6

More importantly, C is still a single computational unit. If processing C requires scanning all eight incident edges, much of the work associated with the hub is concentrated around M1.

We balanced the number of vertices, but we did not balance the amount of work.

Hub Vertices

In a vertex-cut approach, we assign two edges to each machine:

- M1: C → L1, C → L2
- M2: C → L3, C → L4
- M3: C → L5, C → L6
- M4: C → L7, C → L8

Now all machines receive the same amount of edge work. But C must appear on all four machines.

- C replication count: 4.
- Each leaf replication count: 1

Each machine processes its two local edges and produces a partial result for C. Those four partial results are then combined.

Hub Vertices

For example, when counting messages sent to C:

- M1 computes partial count 2
- M2 computes partial count 2
- M3 computes partial count 2
- M4 computes partial count 2
- Merge: 2 + 2 + 2 + 2 = 8

The central trade-off is now visible:

- Edge-cut: keep one copy of C, but communicate across many cut edges and potentially overload its machine.
- Vertex-cut: replicate C, but distribute its incident edges and their computation.

GraphX Algorithms

Per Spark 3.5.3, GraphX includes some pre-built algorithms:

- PageRank
- Connected Components
- Triangle Counting
- Matrix Factorization

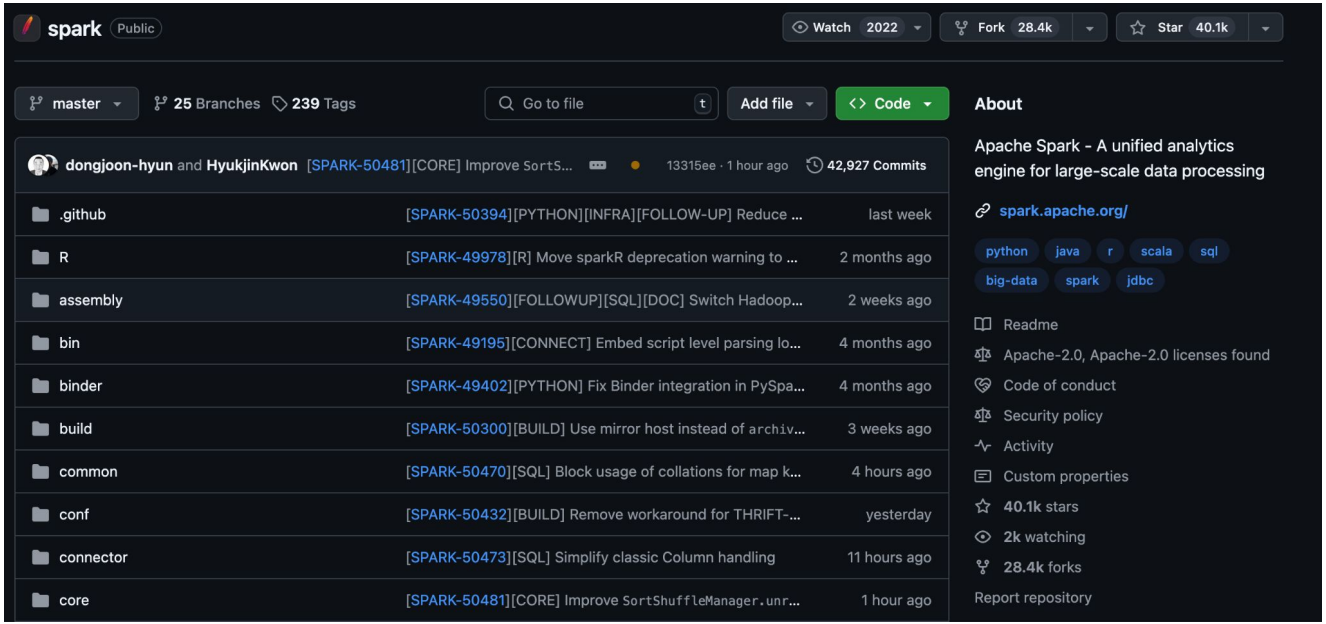
GraphX Triangle Count

```scala
import org.apache.spark.graphx.{GraphLoader, PartitionStrategy}

// Load the edges in canonical order and partition the graph for triangle count
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt", true)
  .partitionBy(PartitionStrategy.RandomVertexCut)
// Find the triangle count for each vertex
val triCounts = graph.triangleCount().vertices
// Join the triangle counts with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val triCountByUsername = users.join(triCounts).map { case (id, (username, tc)) =>
  (username, tc)
}
// Print the result
println(triCountByUsername.collect().mkString("\n"))
```

GraphX’s Level of Activity in 2026

Although the Spark github community is quite active, the last commit for GraphX was performed around 2 years ago :(



What’s Next for GraphX?

Actually, Spark has another graph processing library called **GraphFrames**.

It is exactly the same model as GraphX, but oriented towards DataFrames.
Unfortunately, this one is also not actively updated.

This one has so much potential given the documentation DataFrames have.

What’s Next for GraphX?

With the strong surge of graph databases, would we still need GraphX?

- Development has plateaued a bit, should it add more algorithms?
- Enable interoperability with databases like Neo4j?
- Add query language support to make GraphX accessible to data analysts and non-programmers (GQL).


Thank you!