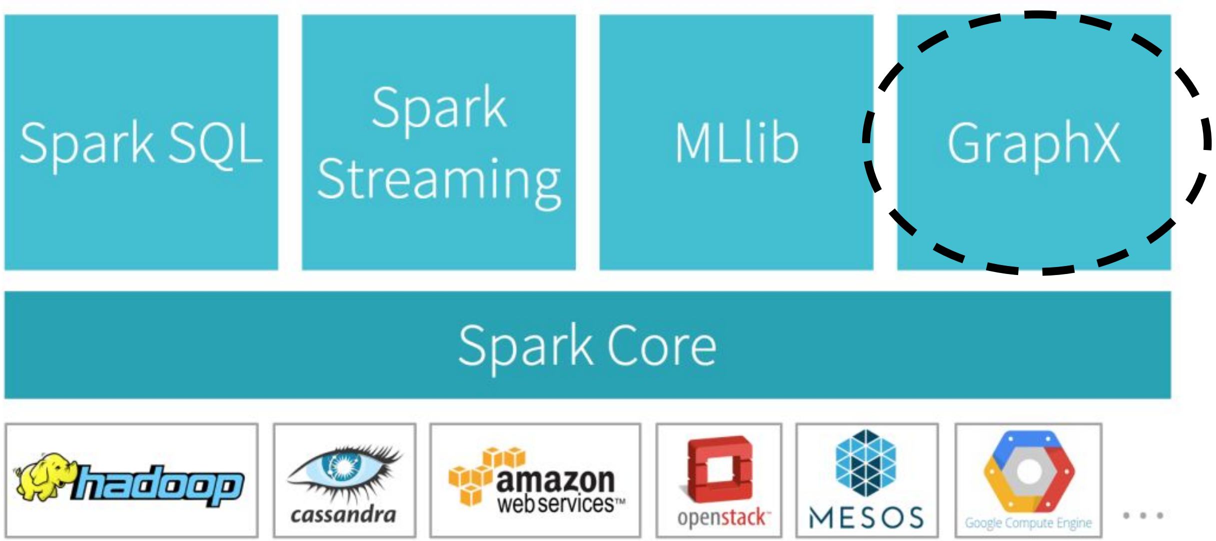
Graphs on Spark

Julián Arenas Guerrero
julian.arenas.guerrero@upm.es

Dec. 2024

Graphs in the Spark Stack



Spark GraphX

- GraphX is a Spark module for graph data processing
- It implements optimizations to efficiently work with the highly connected nature of graph data
- Provides a variety of graph algorithms: page rank, connected components, triangle count…
- *Standing on the shoulder of giants*: similarly to Spark SQL, it is built on top of Spark Core
- Retains the key features of RDDs: fault tolerant, immutability, lazy evaluation, partitioning, in-memory computation…

Property Graphs

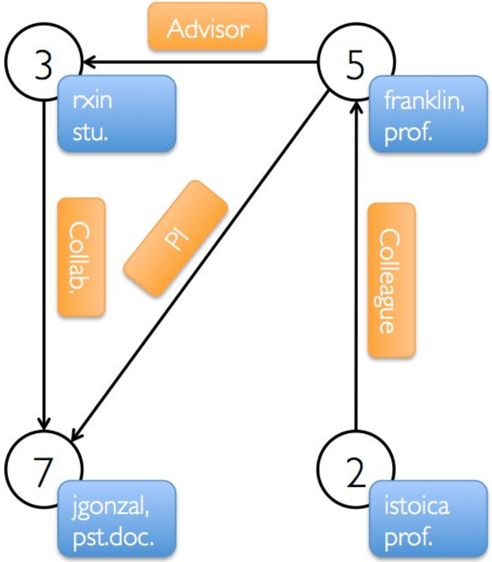
- Directed multigraph with labels attached to each vertex and edge
- A directed multigraph is a directed graph with potentially multiple parallel edges sharing the same source and destination vertex
- To construct a property graph we need to define a set of edges and properties

```scala
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

Property Graphs

- Property graph consisting of the various collaborators on the GraphX project

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

Property Graphs

Graph views:

id	name	role
3	rxin	student
7	jgonzal	postdoc
5	franklin	prof
2	istoica	prof

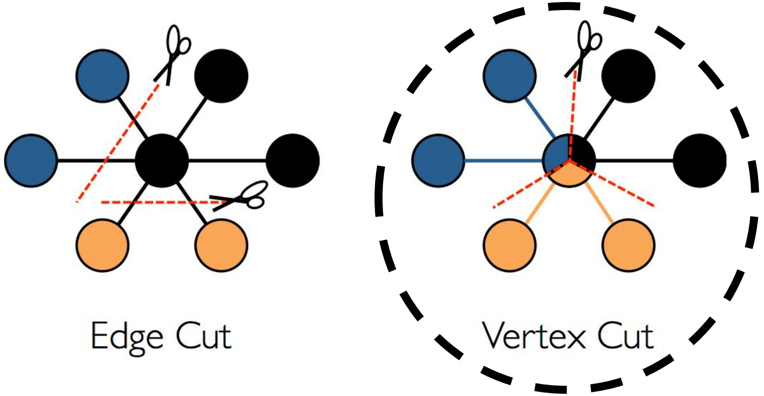
src	dst	relationship
3	7	collab
5	3	advisor
2	5	colleague
5	7	pi

src	edge	dst
{5, franklin, prof}	{5, 3, advisor}	{3, rxin, student}
{5, franklin, prof}	{5, 7, pi}	{7, jgonzal, post...
{3, rxin, student}	{3, 7, collab}	{7, jgonzal, post...
{2, istoica, prof}	{2, 5, colleague}	{5, franklin, prof}

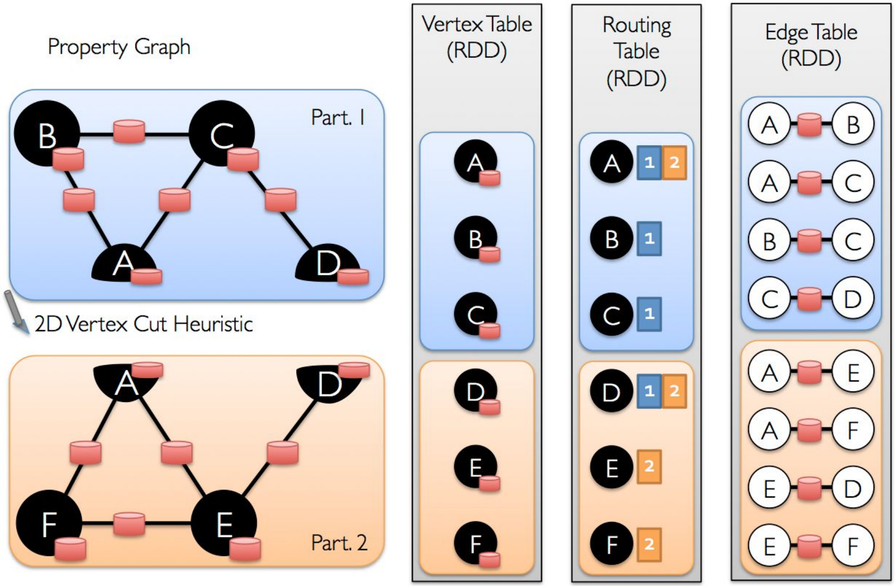
Triplets can be expressed as:

```sql
SELECT src.id, dst.id, src.attr, e.attr, dst.attr
FROM edges AS e LEFT JOIN vertices AS src, vertices AS dst
ON e.srcId = src.Id AND e.dstId = dst.Id
```

Partitioning on GraphX



- GraphX adopts a **vertex-cut** approach
- Edges are assigned to machines and vertices span multiple machines
- Challenge: joining vertex attributes with the edges
  - Because real-world graphs typically have more edges than vertices, we move vertex attributes to the edges



GraphFrames

- GraphFrames is a package for graph processing in Spark
- It has a Python API (GraphX does not)
- Graphs can be created from vertex and edge DataFrames:
  - A vertex DataFrame should contain a special column named “id” which specifies unique IDs for each vertex in the graph
  - An edge DataFrame should contain two special columns: “src” and “dst”
  - Both can have arbitrary other columns representing vertex and edge attributes

```python
# Vertex DataFrame
v = spark.createDataFrame([
  ("a", "Alice", 34),
  ("b", "Bob", 36),
  ("c", "Charlie", 30),
  ("d", "David", 29),
  ("e", "Esther", 32),
  ("f", "Fanny", 36),
  ("g", "Gabby", 60)
], ["id", "name", "age"])
```

```python
# Edge DataFrame
e = spark.createDataFrame([
  ("a", "b", "friend"),
  ("b", "c", "follow"),
  ("c", "b", "follow"),
  ("f", "c", "follow"),
  ("e", "f", "follow"),
  ("e", "d", "friend"),
  ("d", "a", "friend"),
  ("a", "e", "friend")
], ["src", "dst", "relationship"])
```

```python
# Create a GraphFrame
g = GraphFrame(v, e)
```

Basic Queries

```
>>> # Get a DataFrame with columns "id"
>>> # and "inDegree" (in-degree)
>>> g.inDegrees.show()
+---+--------+
| id|inDegree|
+---+--------+
|  b|       2|
|  c|       2|
|  f|       1|
|  d|       1|
|  a|       1|
|  e|       1|
+---+--------+
```

```
>>> # Find the youngest user's age in the graph.
>>> # This queries the vertex DataFrame.
>>> g.vertices.groupBy().min("age").show()
+--------+
|min(age)|
+--------+
|      29|
+--------+
```

```
>>> # Count the number of "follows" in the graph.
>>> # This queries the edge DataFrame.
>>> g.edges.filter("relationship = 'follow'").count()
4
```

Motif Finding

- Motif finding refers to searching for structural patterns in a graph.
- Example: `g.find("(a)-[e]->(b); (b)-[e2]->(a)")`

```
>>> g.find("(a)-[e]->(b); (b)-[e2]->(a)").show()
+----------------+--------------+----------------+--------------+
|               a|             e|               b|            e2|
+----------------+--------------+----------------+--------------+
|{c, Charlie, 30}|{c, b, follow}|    {b, Bob, 36}|{b, c, follow}|
|    {b, Bob, 36}|{b, c, follow}|{c, Charlie, 30}|{c, b, follow}|
+----------------+--------------+----------------+--------------+
```

Motif Finding

Declarative motif language:

- The basic unit of a pattern is an edge
- A pattern is expressed as a union of edges
- Within a pattern, variable names can be assigned to vertices and edges
  - The names can identify common elements among edges `(a)-[e]->(b); (b)-[e2]->(c)`
  - The names are used as column names in the result DataFrame
  - Variable names do ***not*** identify ***distinct*** elements: two elements with different names may refer to the same graph element
    `(a)-[e]->(b); (b)-[e2]->(c)` *(a and c can refer to the same vertex)*

```
>>> g.find("(a)-[e]->(b); (b)-[e2]->(c)").show()
+----------------+--------------+----------------+--------------+----------------+
|               a|             e|               b|            e2|               c|
+----------------+--------------+----------------+--------------+----------------+
| {e, Esther, 32}|{e, d, friend}|  {d, David, 29}|{d, a, friend}|  {a, Alice, 34}|
|  {d, David, 29}|{d, a, friend}|  {a, Alice, 34}|{a, b, friend}|    {b, Bob, 36}|
|  {f, Fanny, 36}|{f, c, follow}|{c, Charlie, 30}|{c, b, follow}|    {b, Bob, 36}|
|    {b, Bob, 36}|{b, c, follow}|{c, Charlie, 30}|{c, b, follow}|    {b, Bob, 36}|
|{c, Charlie, 30}|{c, b, follow}|    {b, Bob, 36}|{b, c, follow}|{c, Charlie, 30}|
|  {a, Alice, 34}|{a, b, friend}|    {b, Bob, 36}|{b, c, follow}|{c, Charlie, 30}|
| {e, Esther, 32}|{e, f, follow}|  {f, Fanny, 36}|{f, c, follow}|{c, Charlie, 30}|
|  {a, Alice, 34}|{a, e, friend}| {e, Esther, 32}|{e, d, friend}|  {d, David, 29}|
|  {d, David, 29}|{d, a, friend}|  {a, Alice, 34}|{a, e, friend}| {e, Esther, 32}|
|  {a, Alice, 34}|{a, e, friend}| {e, Esther, 32}|{e, f, follow}|  {f, Fanny, 36}|
+----------------+--------------+----------------+--------------+----------------+
```

Motif Finding

Declarative motif language:

- It is acceptable to omit names for (*anonymous*) vertices or edges in motifs when not needed

  `(a)-[]->(b)` These are called *anonymous* vertices and edges

a	b
{d, David, 29}	{a, Alice, 34}
{c, Charlie, 30}	{b, Bob, 36}
{a, Alice, 34}	{b, Bob, 36}
{f, Fanny, 36}	{c, Charlie, 30}
{b, Bob, 36}	{c, Charlie, 30}
{e, Esther, 32}	{d, David, 29}
{a, Alice, 34}	{e, Esther, 32}
{e, Esther, 32}	{f, Fanny, 36}

- An edge can be negated to indicate that the edge should *not* be present in the graph

  `(a)-[]->(b); !(b)-[]->(a)`

a	b
{a, Alice, 34}	{e, Esther, 32}
{e, Esther, 32}	{d, David, 29}
{e, Esther, 32}	{f, Fanny, 36}
{a, Alice, 34}	{b, Bob, 36}
{f, Fanny, 36}	{c, Charlie, 30}
{d, David, 29}	{a, Alice, 34}

Motif Finding

More complex queries can be expressed by applying filters to the result DataFrame

```
>>> # Search for pairs of vertices with edges in both directions between them
>>> motifs = g.find("(a)-[e]->(b); (b)-[e2]->(a)")
>>> # More complex queries can be expressed by applying filters.
>>> motifs.filter("b.age > 30").show()
+----------------+--------------+-------------+--------------+
|               a|             e|            b|            e2|
+----------------+--------------+-------------+--------------+
|{c, Charlie, 30}|{c, b, follow}|{b, Bob, 36}|{b, c, follow}|
+----------------+--------------+-------------+--------------+
```

Subgraphs

Subgraph selection is based on a combination of motif finding and DataFrame filters. Methods: `filterVertices(condition)`, `filterEdges(condition)`, and `dropIsolatedVertices()`

```
>>> g1 = g.filterVertices("age > 30").filterEdges("relationship = 'friend'").dropIsolatedVertices()
>>> g1.triplets.show()
+--------------+--------------+----------------+
|           src|          edge|             dst|
+--------------+--------------+----------------+
|{a, Alice, 34}|{a, e, friend}|{e, Esther, 32}|
|{a, Alice, 34}|{a, b, friend}|   {b, Bob, 36}|
+--------------+--------------+----------------+
```

Breadth-first search

Breadth-first search (BFS) finds the shortest path(s) from one vertex (or a set of vertices) to another vertex (or a set of vertices).

```
>>> # Search from "Esther" for users of age < 32.
>>> paths = g.bfs("name = 'Esther'", "age < 32")
>>> paths.show()
+--------------+-------------+-------------+
|          from|           e0|           to|
+--------------+-------------+-------------+
|{e, Esther, 32}|{e, d, friend}|{d, David, 29}|
+--------------+-------------+-------------+

>>> # Specify edge filters or max path lengths.
>>> paths = g.bfs("name = 'Esther'", "age < 32",\
...   edgeFilter="relationship != 'friend'", maxPathLength=3)
>>> paths.show()
+--------------+-------------+-------------+-------------+---------------+
|          from|           e0|           v1|           e1|             to|
+--------------+-------------+-------------+-------------+---------------+
|{e, Esther, 32}|{e, f, follow}|{f, Fanny, 36}|{f, c, follow}|{c, Charlie, 30}|
+--------------+-------------+-------------+-------------+---------------+
```

Shortest Paths

Computes shortest paths from each vertex to the given set of landmark vertices, where landmarks are specified by vertex ID. Note that this takes edge direction into account.

```
>>> results = g.shortestPaths(landmarks=["a", "d"])

>>> results.select("id", "distances").show()
+---+----------------+
| id|       distances|
+---+----------------+
|  g|              {}|
|  f|              {}|
|  e|{a -> 2, d -> 1}|
|  d|{a -> 1, d -> 0}|
|  c|              {}|
|  b|              {}|
|  a|{a -> 0, d -> 2}|
+---+----------------+
```

Triangle count

Computes the number of triangles passing through each vertex.

```
>>>
>>> results = g.triangleCount()
>>> results.select("id", "count").show()
+---+-----+
| id|count|
+---+-----+
|  a|    1|
|  b|    0|
|  c|    0|
|  d|    1|
|  e|    1|
|  f|    0|
|  g|    0|
+---+-----+
```

PageRank

Pagerank assigns a numerical weighting to each element in a graph with the purpose of measuring its relative importance within the graph. The underlying assumption is that more important nodes are more likely to receive links from other nodes.

```
>>> # Run PageRank for a fixed number of iterations
>>> results = g.pageRank(resetProbability=0.15, maxIter=10)
>>> results.vertices.select("id", "pagerank").show()
+---+------------------+
| id|          pagerank|
+---+------------------+
|  g|0.1707317073170731 8|
|  f|0.32504910549694244|
|  e| 0.3613490987992571|
|  d|0.32504910549694244|
|  c| 2.6667877057849627|
|  b| 2.7025217677349773|
|  a| 0.4485115093698443|
+---+------------------+

>>> results.edges.select("src", "dst", "weight").show()
+---+---+------+
|src|dst|weight|
+---+---+------+
|  f|  c|   1.0|
|  e|  f|   0.5|
|  e|  d|   0.5|
|  d|  a|   1.0|
|  c|  b|   1.0|
|  b|  c|   1.0|
|  a|  e|   0.5|
|  a|  b|   0.5|
+---+---+------+
```

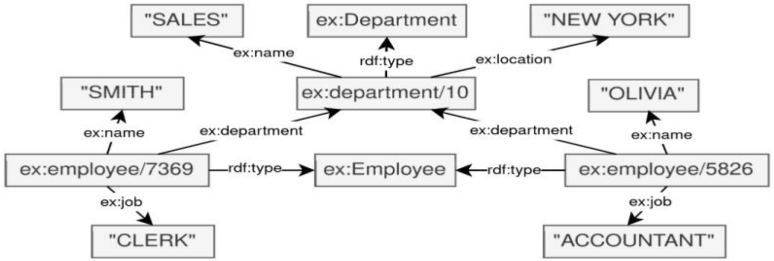
Querying with Cypher

- Cypher (now GQL): SQL for Property Graphs
- Cypher on Spark: **Morpheus** engine
- It exploits an relational representation of the data

What about RDF?

- RDF stands for Resource Description Framework
- RDF is to graphs what XML is to JSON
- Extremely flexible:
  - Good for representing very large knowledge bases (Wikidata, DBpedia, ...)
  - Integrates with ontologies
  - Good for integrating heterogeneous data sources
  - Standardized: querying (SPARQL), validation (SHACL), schema (OWL), mapping (R2RML)

RDF as a Triple Table

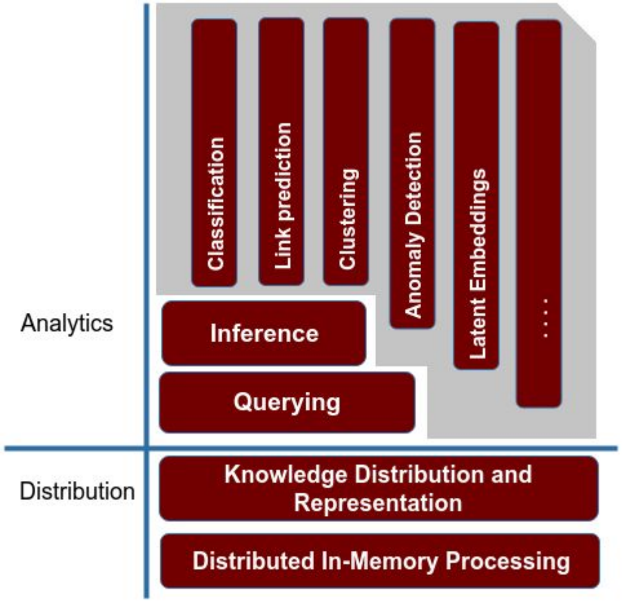



```
SELECT ?loc WHERE {
    ?emp rdf:type ex:Employee .
    ?emp ex:job "ACCOUNTANT" .
    ?emp ex:department ?dept .
    ?dept rdf:type ex:Department .
    ?dept ex:location ?loc .
}
```

```
SELECT
  loc
FROM
  ( ( ( ( ( (
          SELECT
            s AS dept, o AS loc
          FROM
            ITT
          WHERE
            p = 'ex:location'
        ) AS v950117
        INNER JOIN (
          SELECT
            s AS dept
          FROM
            ITT
          WHERE
            p = 'rdf:type' AND o = 'ex:Department'
        ) AS v208794 ON v950117.dept = v208794.dept
      ) AS v208794
      INNER JOIN (
        SELECT
          s AS emp, o AS dept
        FROM
          ITT
        WHERE
          p = 'ex:department'
      ) AS v320081 ON v208794.dept = v320081.dept
    ) AS v320081
    INNER JOIN (
      SELECT
        s AS emp
      FROM
        ITT
      WHERE
        p = 'ex:job' AND o = '"ACCOUNTANT"'
    ) AS v927160 ON v320081.emp = v927160.emp
  ) AS v927160
  INNER JOIN (
    SELECT
      s AS emp
    FROM
      ITT
    WHERE
      p = 'rdf:type' AND o = 'ex:Employee'
  ) AS v588866 ON v927160.emp = v588866.emp
  )
)
```

RDF on Spark

SANSA is a big data engine for scalable processing of large-scale RDF data on Spark. It provides the facilities for Semantic data representation, querying, inference, and analytics.



SANSA