GraphFrames

Large-scale graph processing

Limitations of **data-parallel models** (e.g Map/Reduce, Spark):

- Graphs have irregular structure
  - Difficult to extract parallelism based on partitioning of the data → Poorly partitioned data: unbalanced computation loads
  - Power-law degree distributions for large real-world graphs (social networks, www) → Computation and data access patterns have poor locality of memory access

- The graph is shuffled at each iteration (vertex object N (including the OUT adjacency list) passed as a parameter to map and reduce methods → Better to send only the new importance value and not the structure of the graph

- The iterations are controlled outside M/R (termination conditions and programme logic)

- Little computation required for each vertex

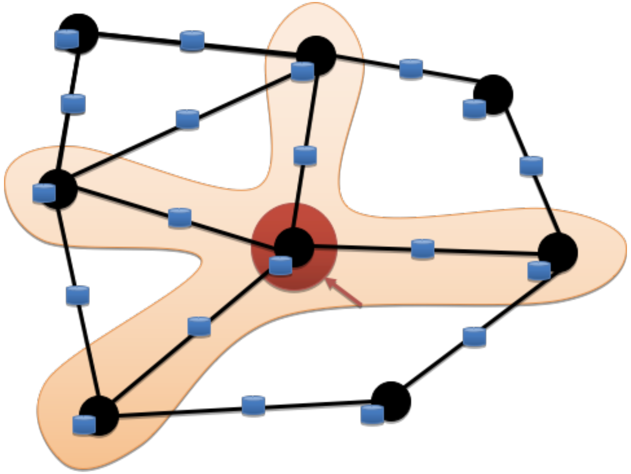
- Degree of parallelism changes during the execution

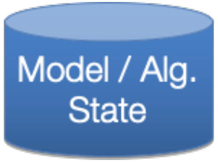
Graph-Parallel Abstraction

Program graph computations:
“**Think like a vertex**” (Malewicz et al. [SIGMOD’ 10])

Principles:

- each vertex has a small neighborhood to maximize parallelism
- effective graph partitioning to minimize communication

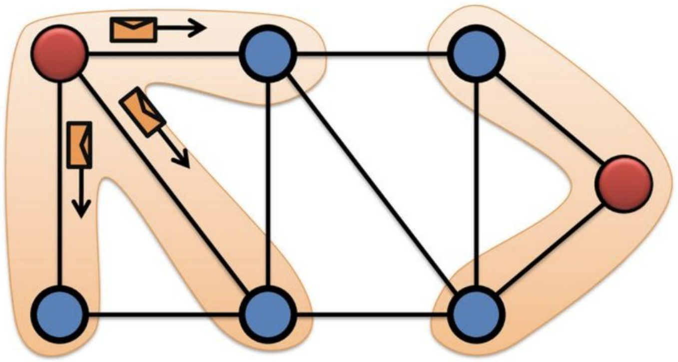




Computation only depends on **neighbors**

Graph-Parallel Abstraction

- A user-defined **Vertex-Program** runs on each vertex
- Graph constrains interactions along edges:
  - using **messages** (e.g. **Pregel**)
  - through shared state (e.g., **GraphLab)**
- **Parallelism:** run multiple vertex programs simultaneously



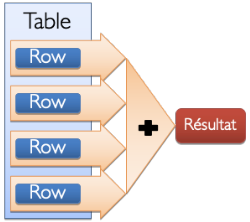
Data-Parallel vs. Graph-Parallel Computation

Data-parallel computations:
- Record-centric view of data
- Parallelism: processing independent data on separate sources

Graph-Parallel computations:
- Vertex-centric views of graphs
  - Specific graph partitioning techniques (graph-dependent)
  - Resolve dependencies (along edges) by iterative computation
  - Restrict the types of computation
  - Communication along edges
  - Exploit graph structure to achieve performance gains of several orders of magnitude compared with more general data-parallel systems.

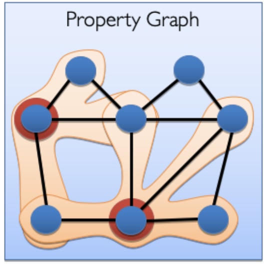
Data-parallel

hadoop Spark



Graph-parallel

Pregel GraphLab APACHE GIRAPH

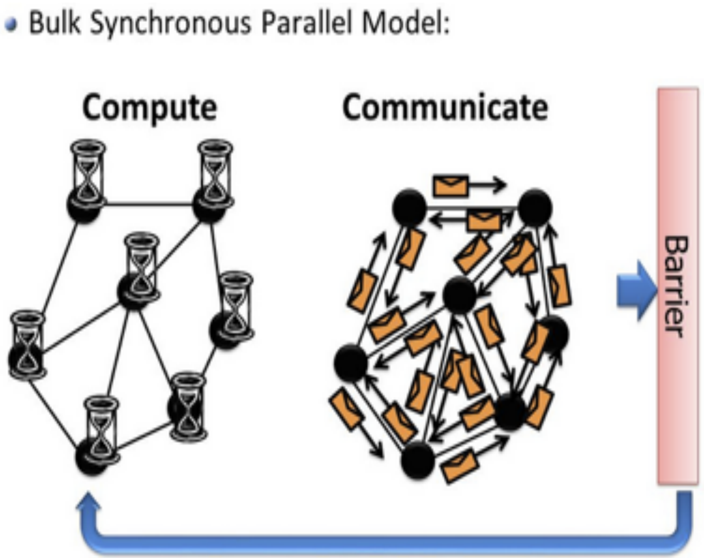


BSP Model

- Bulk synchronous parallel (**BSP**) : parallel programming model for designing data-parallel algorithms.
- **Vertex-centric System.**
- **Principle**:
  - Series of iterations (**supersteps**)
  - At each step (iteration) S, local execution for each vertex V:
    - invokes a function in parallel
    - can read messages sent in previous superstep (S-1)
    - can send messages to be read at next superstep (S+1)
    - can modify the state of itself and of the outgoing edges
    - can modify graph's topology
  - Messages:
    - Message value + destination vertex
    - Sent along outgoing edges, can be sent to any vertex with known id
    - Only available at the beginning of a superstep
    - Guaranteed to be delivered and not duplicated
    - Can be out of order
- **Pregel:**
  - Graph parallel computing framework *based on BSP*
  - Proposed by Google, open source implementations : Apache Giraph, Stanford GPS, Apache Hama

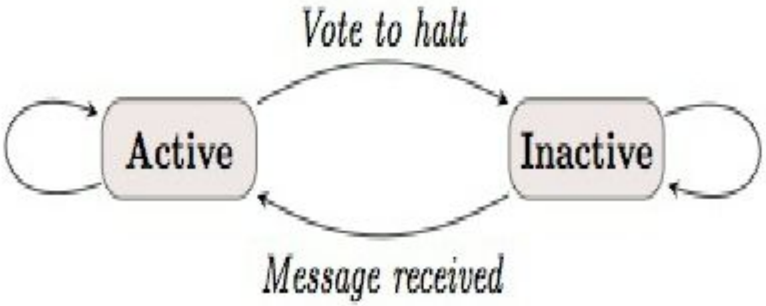
Computation model

- **Input**: a directed graph
  - Each vertex: an identifier and a modifiable value
  - Each directed edge: a source vertex, a target vertex, a modifiable value
- At each **superstep**: computation→communication→barrier synchronization
- **Synchronisation barrier**:
  - at the end of each superstep
  - ensures that all messages have been transmitted but not yet delivered
  - message delivery: beginning of the next superstep: ensures deadlock-free execution.



Termination

- In superstep 0 all vertices are active
- Only active vertices participate in a superstep
- Active vertices can vote to halt → become inactive
- Inactive vertices can be reactivated upon receiving a message → become active
- **Algorithm termination**: when all nodes vote to halt (all vertices are inactive) and there are no messages in transit
- **Output**: set of values output by vertices



Example: PageRank in Pregel

in msgs

```
class PageRankVertex
    : public Vertex<double, void, double> {
  public:
   virtual void Compute(MessageIterator* msgs) {
     if (superstep() >= 1) {
       double sum = 0;
       for (; !msgs->Done(); msgs->Next())
         sum += msgs->Value();
       *MutableValue() =
           0.15 / NumVertices() + 0.85 * sum;
     }

     if (superstep() < 30) {
       const int64 n = GetOutEdgeIterator().size();
       SendMessageToAllNeighbors(GetValue() / n);
     } else {
       VoteToHalt();
     }
   }
};
```

out msgs

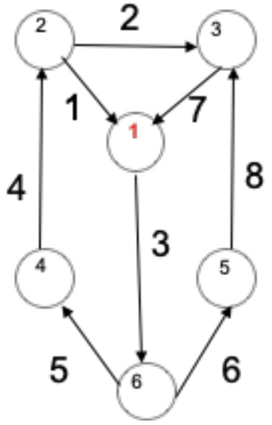
Vertex process executes **Compute()** during each superstep

Superstep 0 (Initialization): 1/NumVertices() for each vertex

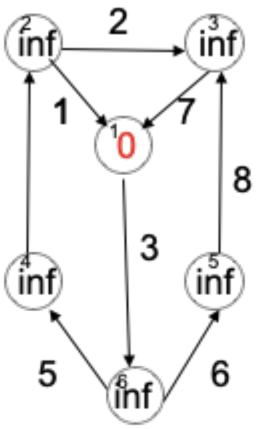
Example: Single Source Shortest Paths in Pregel

```
class ShortestPathVertex
        : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
        int mindist = IsSource(vertex_id()) ? 0 : INF;
        for (; !msgs->Done(); msgs->Next())
                mindist = min(mindist, msgs->Value());
        if (mindist < GetValue()) {
                *MutableValue() = mindist;
                OutEdgeIterator iter = GetOutEdgeIterator();
                for (; !iter.Done(); iter.Next())
                        SendMessageTo(iter.Target(),mindist +
iter.GetValue());
        }
        VoteToHalt();
  }
};
```

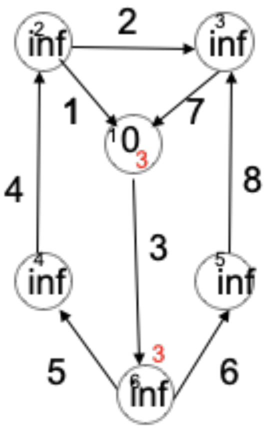
Example: Single Source Shortest Paths



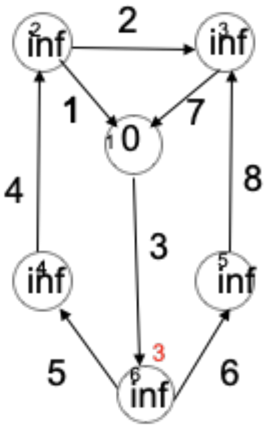
Superstep 0
Initialization



Superstep 0
Communication

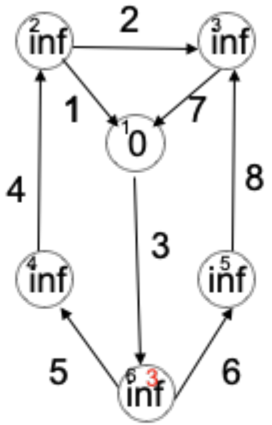


Superstep 0
Synchronisation

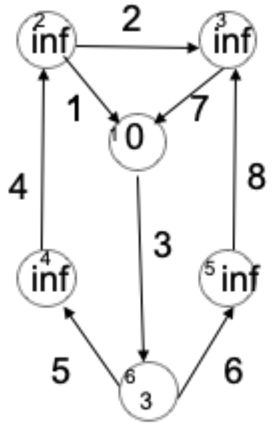


Example: Single Source Shortest Paths-Superstep 1

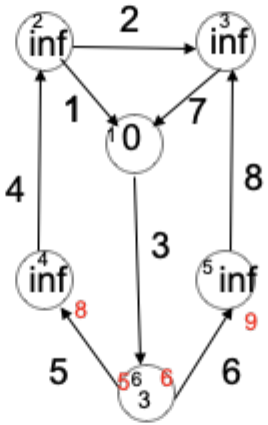
Superstep 1
GetMessages



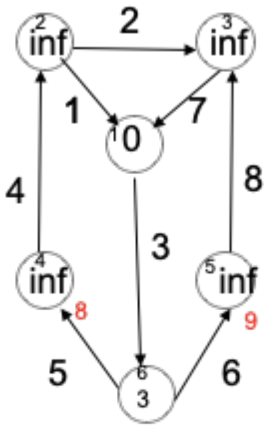
Superstep 1
Update Values



Superstep 1
Communicate

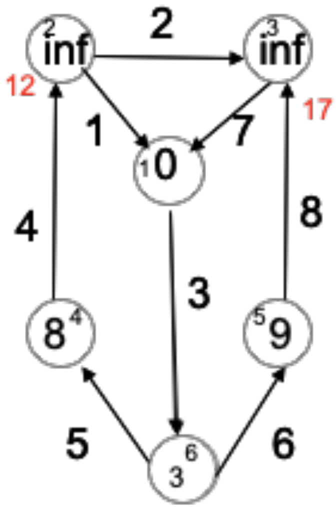


Superstep 1
Synchronisation

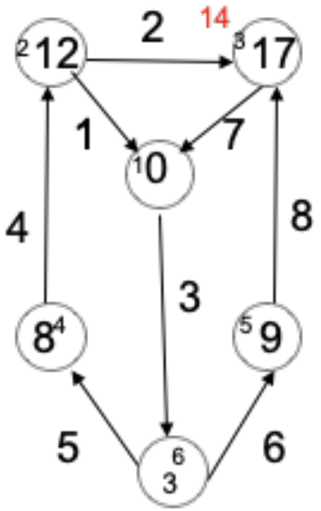


Example: Single Source Shortest Paths-Supersteps 2-4

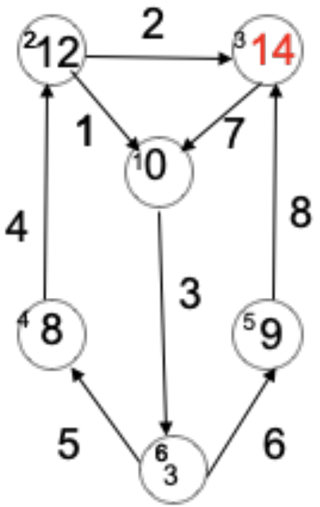
Superstep 2



Superstep 3



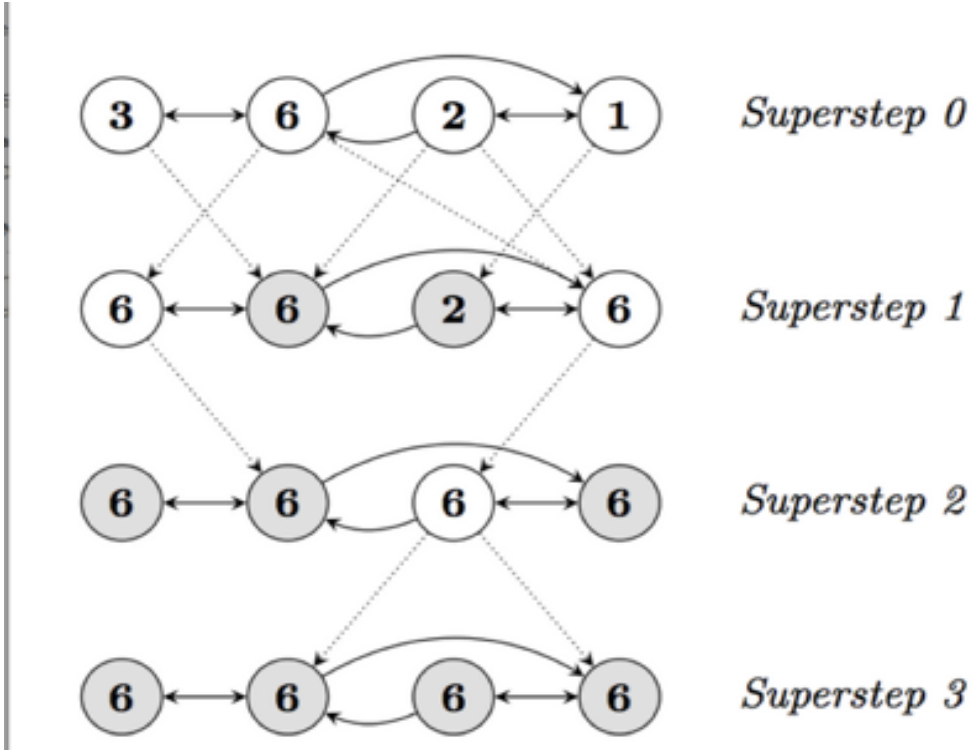
Superstep 4:
Halt for all vertices



Result :

(1,0.0) (2,12.0) (3,14.0) (4,8.0) (5,9.0) (6,3.0)

Example: finding maximum value



Dotted Arrows: messages

Grey nodes: inactive

Advantages of the BSP Model

Advantages:

- Vertex-centric approach:
  - Structured way to develop parallel algorithms
  - Users focus on a local action
  - Process each item independently
- Ease of Reasoning/Predictability of the performance/behavior of parallel algorithms: clear structure of BSP (computation, communication, synchronization)
- Better optimization and resource allocation: predict execution time of parallel algorithms
- Scalability: an be effectively used on small multi-core processors or large distributed systems.
- Synchronization barrier:
  - helps in managing workloads that are not uniformly distributed among processors, load balancing
  - reduces the overhead of frequent synchronization
- Abstraction of the communication layer: focus on algorithm development=>BSP algorithms portable across different parallel architectures
- No deadlocks and data races common in asynchronous systems

Limitations of the BSP Model

- Inefficient if different graph regions converge at different speed.
- Can suffer if one task is more expensive than the others
- Runtime of each phase: determined by the slowest machine
- Difficult to express the different stages of a *processing pipeline on graphs* (building/modifying the graph, computations on several graphs)

Example: Graph analysis pipeline

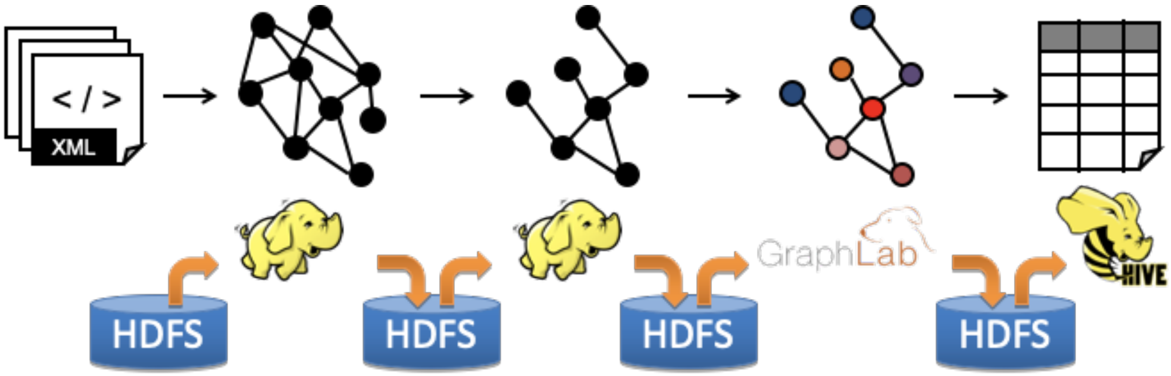
Difficult to use and program

- Users have to learn, deploy and manage multiple systems

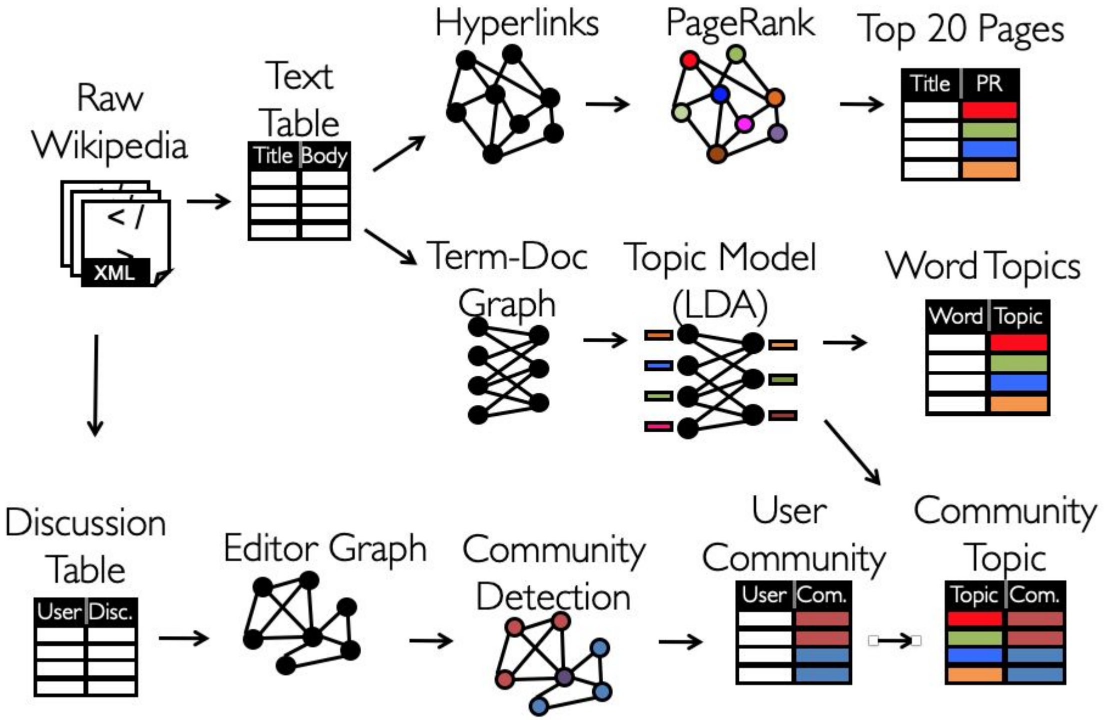
  Leads to interfaces that are complicated to implement and often complex to use

Inefficient :

- Generate large amounts of data movement and duplication across the network and file system
- Limited re-use of internal data structures from one stage to the next



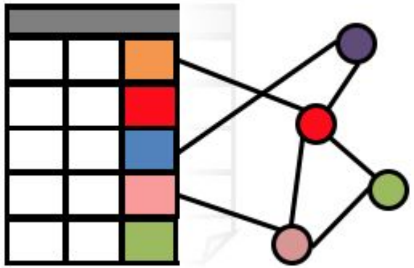
Problem: Mixed Graph Analysis



Same data, different vues (table or graph), easily change between them

GraphX: Unifying Graphs and Tables

New API
Reduces the distinction between Tables and Graphs

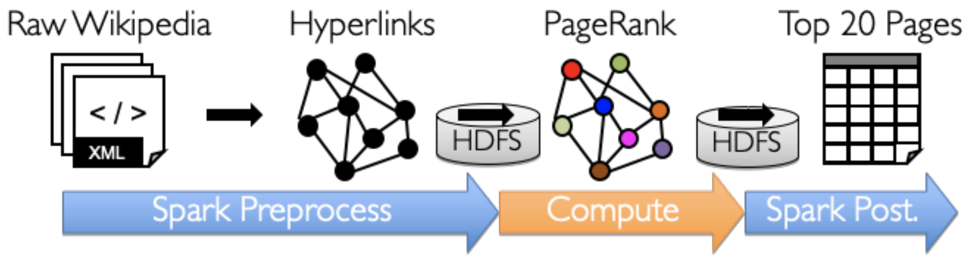


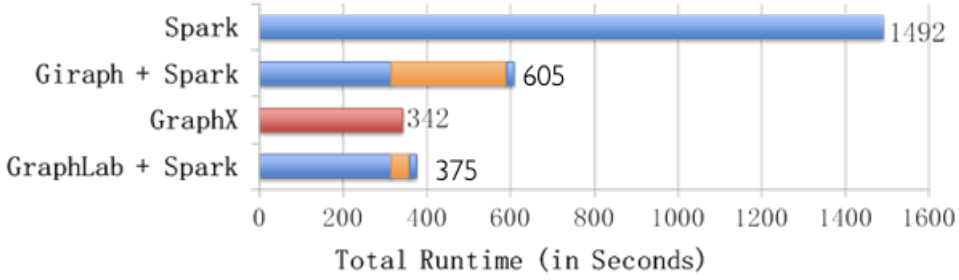
New System
Combines Data-Parallel and Graph-Parallel systems

Enables users:

- Easily and efficiently express the entire graph analysis pipeline.
- View data both as collections (RDD) and as a graph without data movement/duplication

Example: Graph analysis pipeline with GraphX

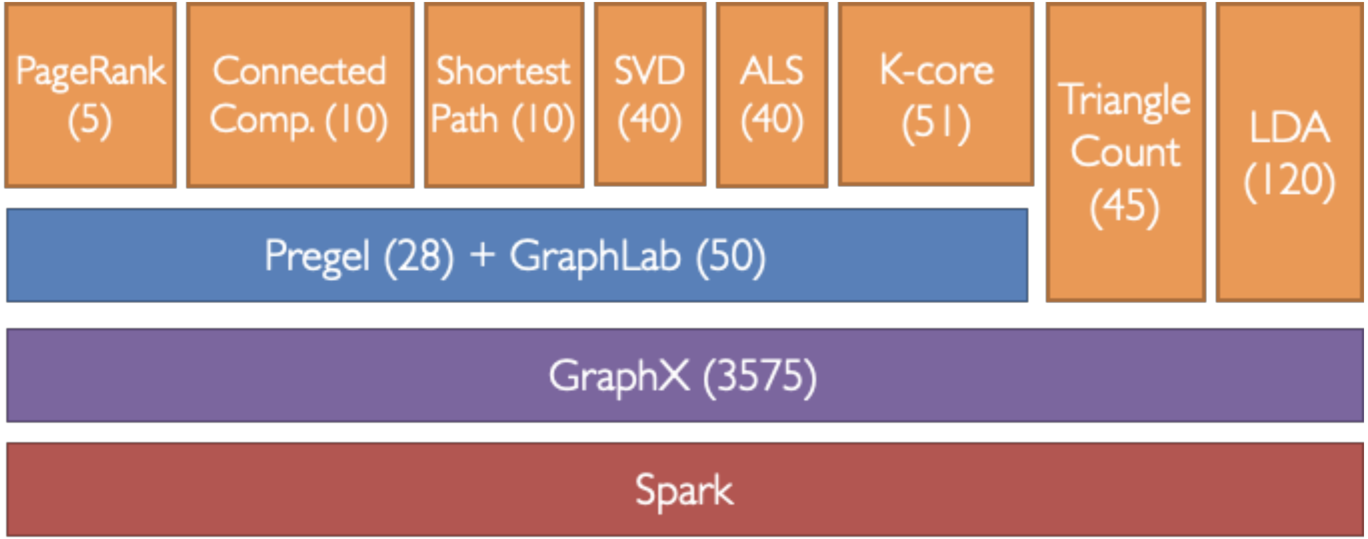




GraphX:

- processing time for the entire pipeline is faster than Spark+GraphLab

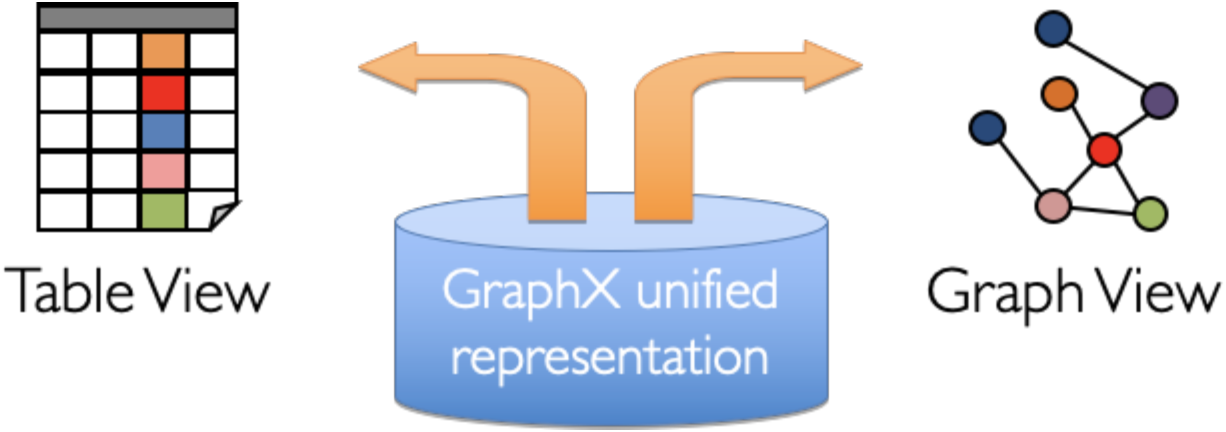
Graph algorithms in GraphX (lines of code)



Pregel and GraphLab algorithms are implemented with GraphX operators in less than 50 lines of code.

GraphX: different views

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



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

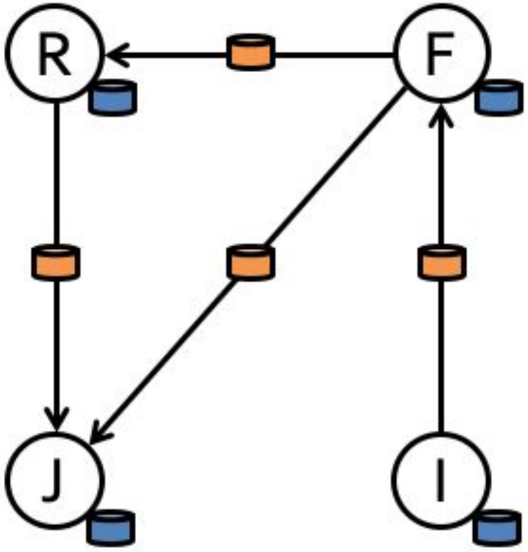
Property Graphs

- Directed multigraphs with user-defined objects attached to each edge and vertex (allow to model several relationships between nodes)
- Each vertex has a unique 64-bit **VertexID** key
- Each edge has the ID of the source vertex and the ID of the destination vertex
- Two RDDs: **VertexRDD[VD]** (for vertices) and **EdgeRDD[ED]** (for edges)
  - **VD, ED**: types of the objects associated with vertices/edges

Like RDDs, property graphs are:

- Immutable: changes to the values or structure of the graph are made by producing a new graph.
- Distributed: the graph is partitioned using a set of node partitioning heuristics
- Fault-tolerant: as with RDD, each partition on the graph can be recreated on another machine for fault tolerance purposes

Property Graph



Graph((String, String), String)

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

Vertex Table:
`VertexRDD[VD]`

`VD: (String, String)`

Src_Id	Dst_Id	Property (E)
rxin	jegonzal	Friend
franklin	rxin	Advisor
istoica	franklin	Coworker
franklin	jegonzal	PI

Edge Table: `EdgeRDD[ED]`

`ED: String`

RDD Operations

Table operators (RDD) 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`	`...`

Graph Operations (1)

```
/** 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]]
// 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 mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): 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]
 // Change the partitioning heuristic  ============================================================
   def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
```

Graph Operations (2)

```
// 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]
// Functions for caching graphs ==================================================================
  def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
  def cache(): Graph[VD, ED]
  def unpersistVertices(blocking: Boolean = false): Graph[VD, ED]
```

Apache Spark's GraphX Library

- General purpose graph processing library
- Built into Spark
- Optimized for fast distributed computing
- Library of algorithms: PageRank, Connected Components, etc

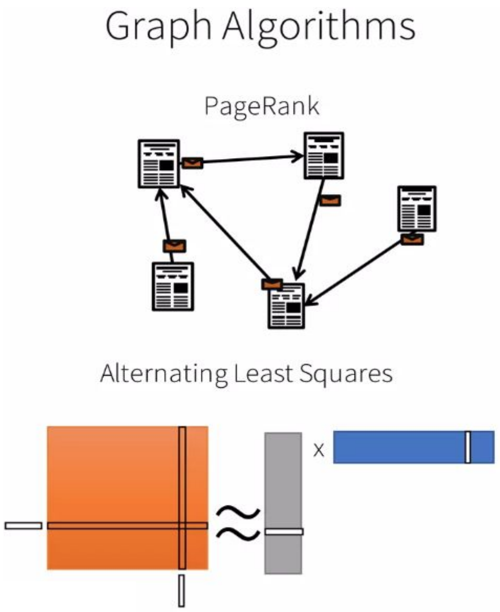
Limitations:

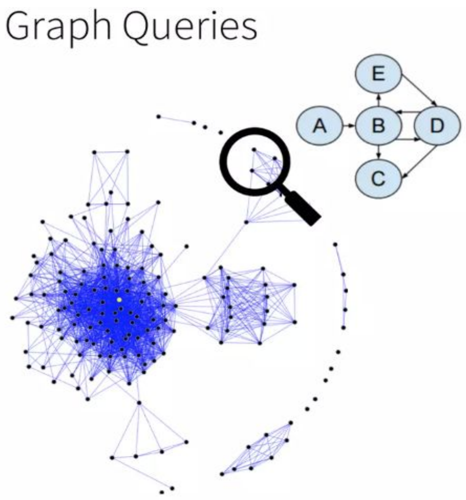
- No Java and Python APIs
- Lower-level RDD-based API (vs. DataFrames)
- Cannot use recent Spark optimizations: Catalyst query optimizer, Tungsten memory management
- *No support for graph queries/patterns*

Graph Algorithms vs. Graph Queries

Graph algorithms: complex computations, graph traversal
E.g: most influential people (Page Rank), shortest paths from Alice to Bob

Graph queries: identify an explicit **pattern** within the graph
E.g: common friends of users Alice and Bob,
all friends of Alice which are not friends of Bob





Graph Algorithms vs. Graph Queries

Graph Algorithm: Single Source Shortest Paths

```
val sssp= graph.pregel(Double.PositiveInfinity)(
  (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
  triplet => {  // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {
      Iterator.empty
    }
  },
  (a, b) => math.min(a, b) // Merge Message
 )
```

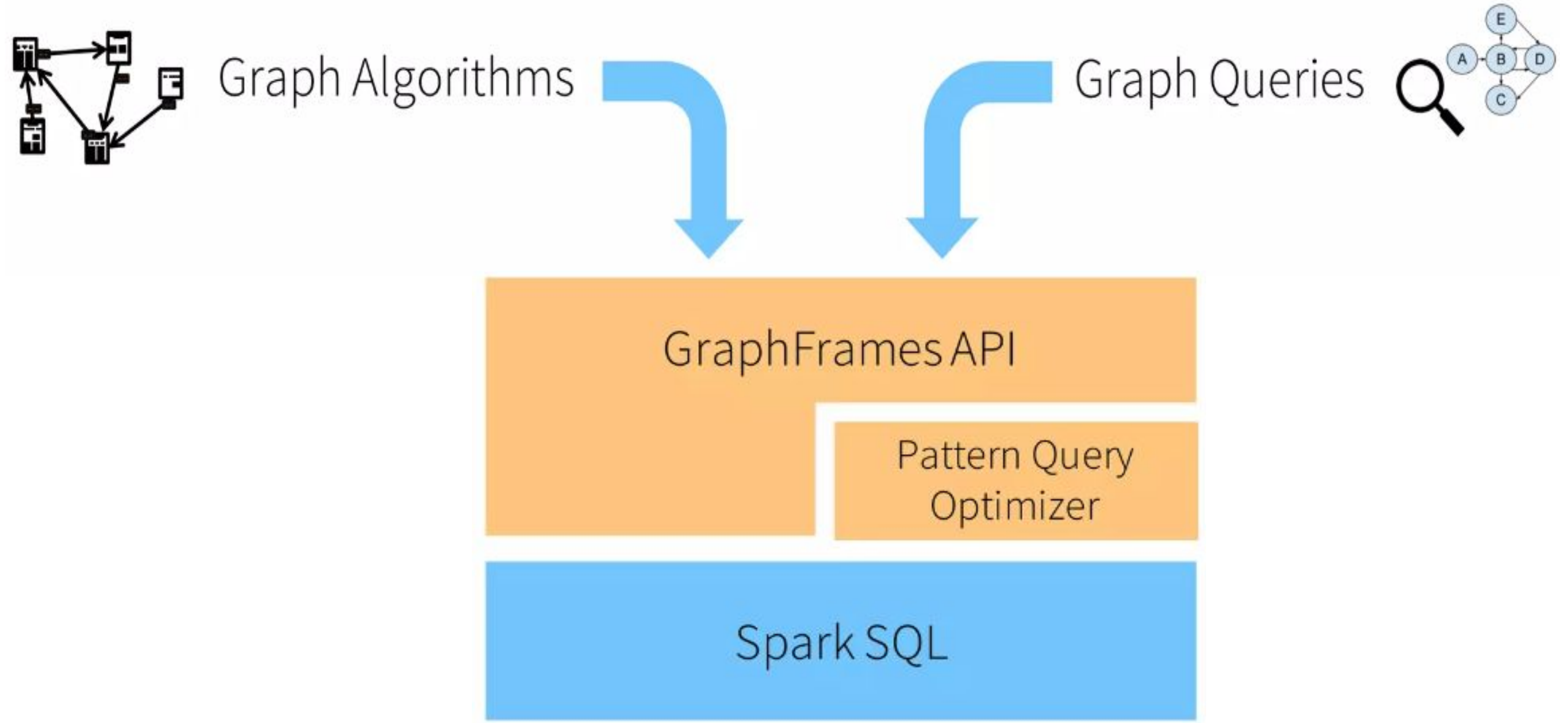
Graph Query: Select subgraph based on edges "e" of type "follow" pointing from a younger user "a" to an older user "b".

```
paths = g.find("(a)-[e]->(b)")\
 .filter("e.relationship = 'follow'")\
 .filter("a.age < b.age")
```

Separate Systems

Graph Algorithms	Graph Queries
GraphX	neo4j
APACHE GIRAPH	TITAN
GraphLab	OrientDB

Solution: GraphFrames



GraphFrames Spark package

- Spark package introduced in 2016
    - Collaboration between Databricks, UC Berkley and MIT

- DataFrames API for Spark
    - High-level API in Java, Python and Scala.
    - Simplifies interactive queries
    - Benefits from DataFrames optimizations
    - Integrates with the rest of Spark ecosystem

- GraphFrames are to DataFrames as GraphX are to RDDs
    - Expressive graph queries, pattern matching
    - Query plan optimizers from Spark SQL
    - Graph algorithms

- Not yet integrated into the Spark architecture

GraphFrames vs GraphX

	GraphFrames	GraphX
Core APIs	Scala, Java, Python	Scala only
Programming Abstraction	DataFrames	RDDs
Use Cases	Algorithms, Queries, Motif Finding	Algorithms
VertexIds	Any type (in Catalyst)	Long
Vertex/edge attributes	Any number of DataFrame columns	Any type (VD,ED)
Return Types	GraphFrames/DataFrames	Graph [VD,ED] or...

GraphFrames API

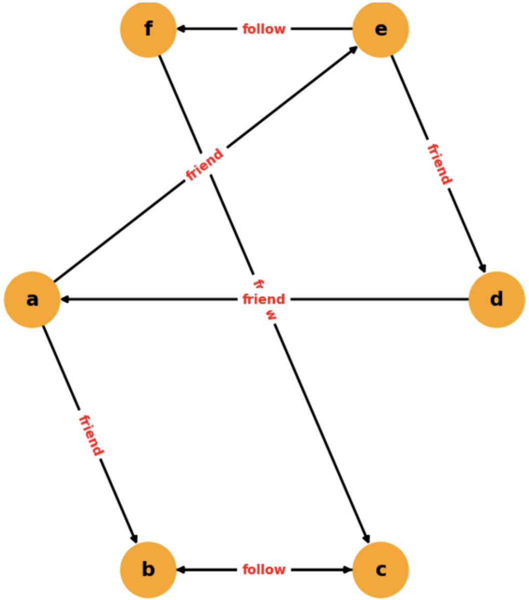
- Unifies graph algorithms, graph queries and DataFrames
- Available in Java, Scala and Python

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

        def find(pattern: String): DataFrame

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

Supported graph algorithms

- Breadth-first search (BFS)
- Connected components
    - Strongly connected components
- LPA: label propagation algorithm
- PageRank and Personalized PageRank
- Shortest paths
- Triangle count

Vertices DataFrame




```
g.vertices.show()
```

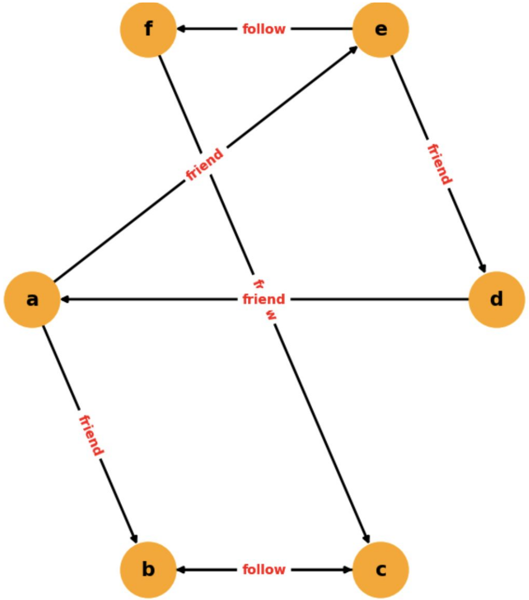
```
root
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- age: long (nullable = true)
```

id	name	age
a	Alice	34
b	Bob	36
c	Charlie	30
d	David	29
e	Esther	32
f	Fanny	36

Vertices DataFrame:

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

Edges DataFrame




```
root
 |-- src: string (nullable = true)
 |-- dst: string (nullable = true)
 |-- relationship: string (nullable = true)
```

`g.edges.show()`

src	dst	relationship
a	b	friend
b	c	follow
c	b	follow
f	c	follow
e	f	follow
e	d	friend
d	a	friend
a	e	friend

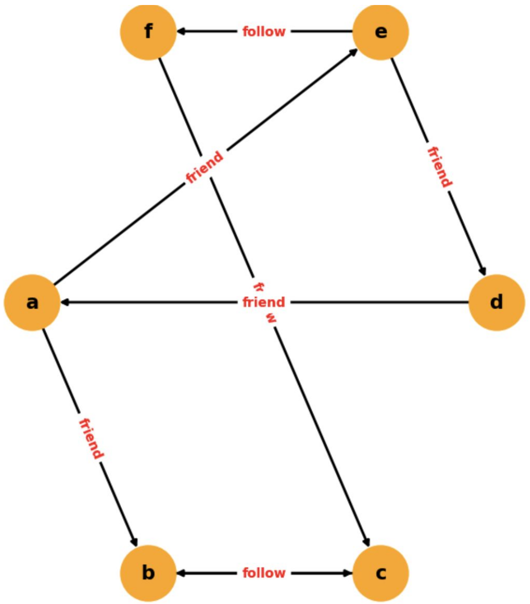
Edges DataFrame:

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

Extra columns store vertex of edge data (attributes or properties)

Triplets DataFrame

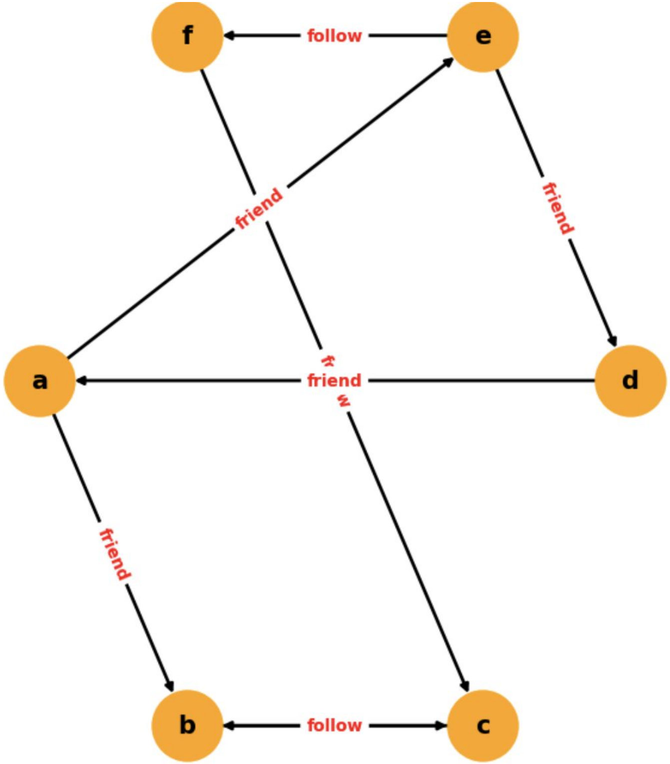
- Extends the information in edges with informations on the vertices
- Join vertices and edges to get (src, edge, dst)




```
g.triplets.show()
```

```
root
 |-- src: struct (nullable = false)
 |    |-- id: string (nullable = true)
 |    |-- name: string (nullable = true)
 |    |-- age: long (nullable = true)
 |-- edge: struct (nullable = false)
 |    |-- src: string (nullable = true)
 |    |-- dst: string (nullable = true)
 |    |-- relationship: string (nullable = true)
 |-- dst: struct (nullable = false)
 |    |-- id: string (nullable = true)
 |    |-- name: string (nullable = true)
 |    |-- age: long (nullable = true)

+----------------+--------------+----------------+
|             src|          edge|             dst|
+----------------+--------------+----------------+
|  {d, David, 29}|{d, a, friend}|  {a, Alice, 34}|
|  {a, Alice, 34}|{a, b, friend}|    {b, Bob, 36}|
|{c, Charlie, 30}|{c, b, follow}|    {b, Bob, 36}|
|    {b, Bob, 36}|{b, c, follow}|{c, Charlie, 30}|
|  {f, Fanny, 36}|{f, c, follow}|{c, Charlie, 30}|
| {e, Esther, 32}|{e, d, friend}|  {d, David, 29}|
|  {a, Alice, 34}|{a, e, friend}| {e, Esther, 32}|
| {e, Esther, 32}|{e, f, follow}|  {f, Fanny, 36}|
+----------------+--------------+----------------+
```

Computation of vertex degrees




```
# Get a DataFrame with columns "id" and "inDegree" (in-degree)
vertexInDegrees = g.inDegrees
```

id	inDegree
b	2
c	2
f	1
d	1
a	1
e	1

Motif finding: Searching for structural patterns (1)

Notations for structural queries:

- () for vertices, e.g (a), ()-[]->() for edges, e.g (a)-[e]->(b)
- Anonymous vertices: (), anonymous edges: []
- Negation of an edge (the edge should *not* be present in the graph), e.g !(b)-[]->(a)

Pattern/Motif:

  - union of edges, e.g (a)-[e]->(b); (b)-[e2]->(c)
  - Same names for common elements, e.g (a)-[e]->(b); (b)-[e2]->(c)
  - names are used as column names in the result DataFrame

Motif finding: Searching for structural patterns (2)

Usage: graph.find("(a)-[e]->(b); (b)-[e2]->(a)")

- Result: a DataFrame with columns for each of the named elements (vertices or edges) in the motif, e.g a, b, e, e2

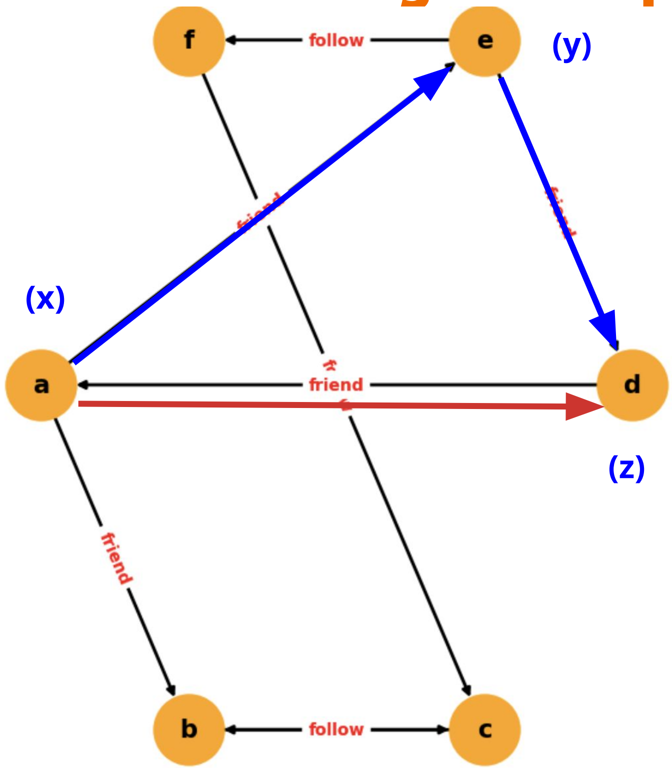
- columns “a” and “b” are StructType with sub-fields equivalent to the schema of **GraphFrame.vertices**.

- column “e” and “e2” are StructType with sub-fields to the schema of **GraphFrame.edges**

- Motifs are not allowed to contain:
  - edges without any named elements (e.g "()-[]->()" and "!()-[]->()" are prohibited)
  - named edges within negated terms since these named edges would never appear within results, e.g "!(a)-[ab]->(b)" is invalid, but "!(a)-[]->(b)" is valid.

- Can return duplicate rows, e.g "(u)-[]->()" will return a result for each matching edge

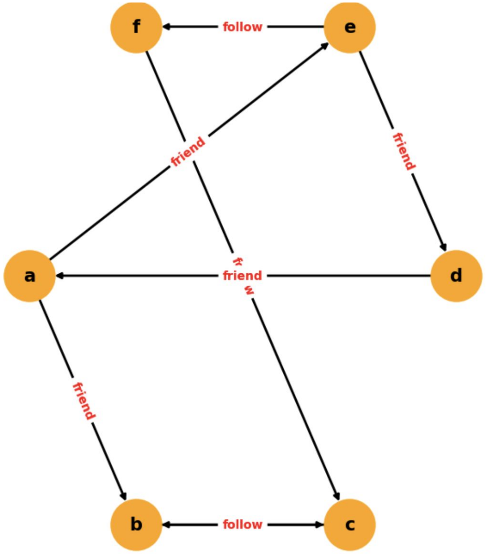
Motif finding: example




```
#Friends of friends
fof = g.find("(x)-[]->(y); (y)-[]->(z);
!(x)-[]->(z)").filter("x.id != z.id")

fof.select(col('x.id').alias('x'), col('y.id').alias('y'),
col('z.id').alias('z')).show()
```

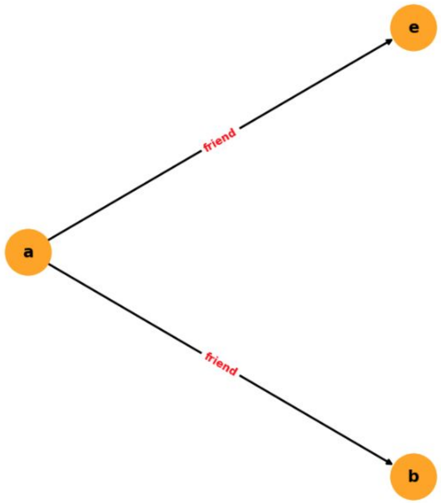
```
+---+---+---+
|  x|  y|  z|
+---+---+---+
|  a|  e|  d|
|  d|  a|  b|
|  d|  a|  e|
|  a|  b|  c|
|  e|  d|  a|
|  f|  c|  b|
|  e|  f|  c|
|  a|  e|  f|
+---+---+---+
```

Subgraph (1)

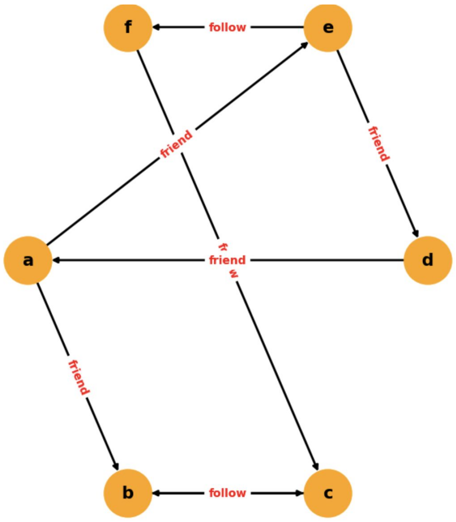



```
+---+------+---+
| id|  name|age|
+---+------+---+
|  a| Alice| 34|
|  b|   Bob| 36|
|  e|Esther| 32|
|  f| Fanny| 36|
+---+------+---+
```

```
# Select subgraph of users older than 30, and
relationships of type "friend".
g1 = g.filterVertices("age > 30")
.filterEdges("relationship = 'friend'")
.dropIsolatedVertices()
```

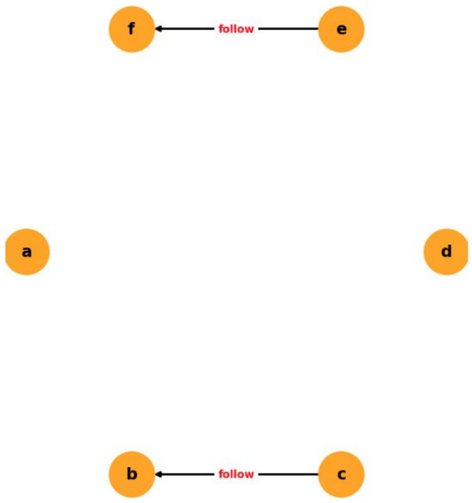


Subgraph (2)




```
g.triplets.filter("src.age < dst.age").show()
+----------------+--------------+--------------+
|             src|          edge|           dst|
+----------------+--------------+--------------+
|  {d, David, 29}|{d, a, friend}|{a, Alice, 34}|
|  {a, Alice, 34}|{a, b, friend}|  {b, Bob, 36}|
|{c, Charlie, 30}|{c, b, follow}|  {b, Bob, 36}|
| {e, Esther, 32}|{e, f, follow}|{f, Fanny, 36}|
+----------------+--------------+--------------+
```

```
# Select subgraph based on edges "e" of type "follow"
# pointing from a younger user "a" to an older user "b".
paths = g.find("(a)-[e]->(b)")\
 .filter("e.relationship = 'follow'")\
 .filter("a.age < b.age")
e2 = paths.select("e.src", "e.dst", "e.relationship")
# Construct the subgraph
g2 = GraphFrame(g.vertices, e2)
```



Graph Algorithms

- `bfs`(*fromExpr*, *toExpr*, *edgeFilter=None*, *maxPathLength=10*)
  - **Returns:** DataFrame with one Row for each shortest path between matching vertices.

- `connectedComponents`(*algorithm='graphframes'*, *checkpointInterval=2*, *broadcastThreshold=1000000*)
  - **algorithm** – connected components algorithm to use (default: “graphframes”) Supported algorithms are “graphframes” and “graphx”.
  - **checkpointInterval** – checkpoint interval in terms of number of iterations (default: 2)
  - **broadcastThreshold** – broadcast threshold in propagating component assignments (default: 1000000)
  - **Returns:** DataFrame with new vertices column “component”

- `stronglyConnectedComponents`(*maxIter*): Runs the strongly connected components algorithm on this graph. Based on Pregel ()
- `labelPropagation`(*maxIter*): Runs static label propagation for detecting communities in networks. Based on Pregel, messages sent on edges in both directions.
  - **maxIter** – the number of iterations to run
  - **Returns**: DataFrame with new vertex column “component”

Graph Algorithms

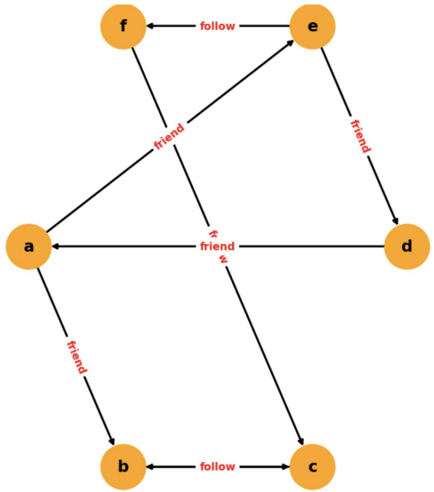
- `shortestPaths`(*landmarks*): Runs the shortest path algorithm from a set of landmark vertices in the graph. Takes edge direction into account.
    - **landmarks** – a set of one or more landmarks
    - **Returns:** DataFrame with new column “distances”

- `triangleCount`(): Counts the number of triangles passing through each vertex in this graph.
    - **Returns:** DataFrame with new vertex column “count”

- `pageRank`(*resetProbability=0.15*, *sourceId=None*, *maxIter=None*, *tol=None*)
    - **resetProbability** – Probability of resetting to a random vertex.
    - **sourceId** – (optional) the source vertex for a personalized PageRank.
    - **maxIter** – If set, the algorithm is run for a fixed number of iterations. This may not be set if the tol parameter is set.
    - **tol** – If set, the algorithm is run until the given tolerance. This may not be set if the numIter parameter is set. (**Exactly one of fixed_num_iter or tolerance must be set.**)
    - **Returns:** GraphFrame with new vertices column “pagerank” and new edges column “weight”

Graph Algorithms: BFS

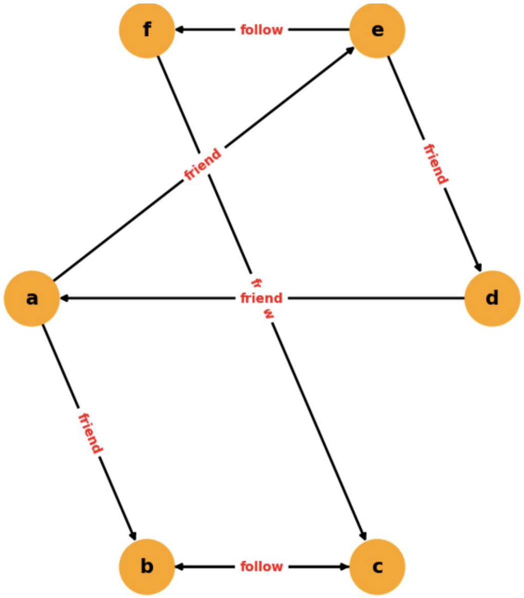
```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
+---+-------+---+
```




```
# Search from "Esther" for users of age < 32.
g.bfs("name = 'Esther'", "age < 32",\
 edgeFilter="relationship != 'friend'", maxPathLength=3).show()
```

```
+--------------+-------------+-------------+-------------+---------------+
|          from|           e0|           v1|           e1|             to|
+--------------+-------------+-------------+-------------+---------------+
|{e, Esther, 32}|{e, f, follow}|{f, Fanny, 36}|{f, c, follow}|{c, Charlie, 30}|
+--------------+-------------+-------------+-------------+---------------+
```

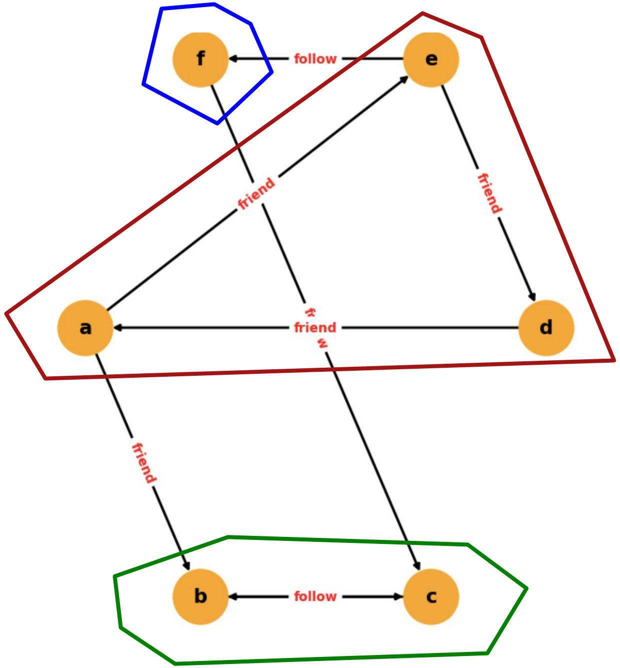
Graph Algorithms: Connected components




```
g.connectedComponents().orderBy("component").show()
```

```
+---+-------+---+---------+
| id|   name|age|component|
+---+-------+---+---------+
|  a|  Alice| 34|        0|
|  b|    Bob| 36|        0|
|  c|Charlie| 30|        0|
|  d|  David| 29|        0|
|  e| Esther| 32|        0|
|  f|  Fanny| 36|        0|
+---+-------+---+---------+
```

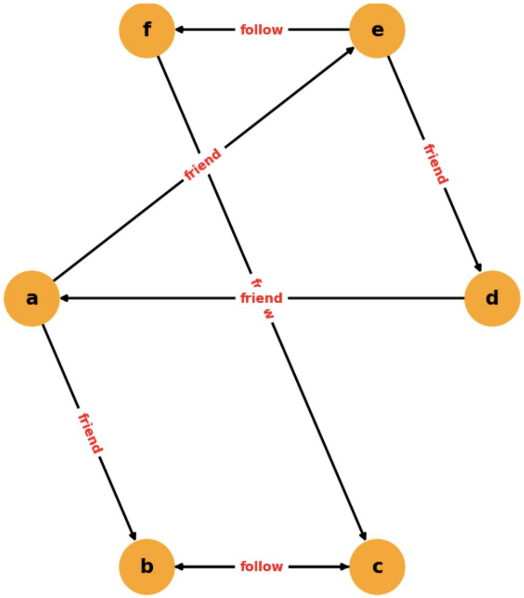
Graph Algorithms: Strongly connected components




```
g.stronglyConnectedComponents(maxIter=10).orderBy("component").show()
```

```
+---+-------+---+---------+
| id|   name|age|component|
+---+-------+---+---------+
|  a|  Alice| 34|        0|
|  d|  David| 29|        0|
|  e| Esther| 32|        0|
|  b|    Bob| 36|        1|
|  c|Charlie| 30|        1|
|  f|  Fanny| 36|        5|
+---+-------+---+---------+
```

Graph Algorithms: Page Rank



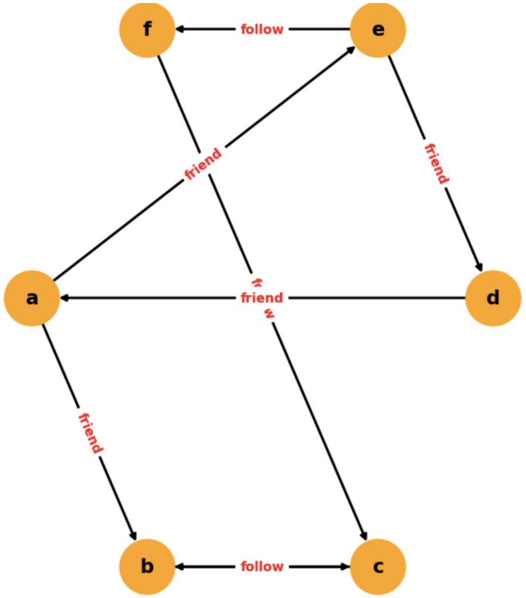

```
# Run PageRank until convergence to tolerance "tol".
results = g.pageRank(resetProbability=0.15, tol=0.01)

results.vertices.select("id", "pagerank").show()
results.edges.select("src", "dst", "weight").show()
```

src	dst	weight
a	b	0.5
a	e	0.5
b	c	1.0
c	b	1.0
d	a	1.0
e	d	0.5
e	f	0.5
f	c	1.0

id	pagerank
a	0.3951071796531403 5
b	2.336217781395228
c	2.364653632110854 4
d	0.2888796063698876 3
e	0.3262621941010017
f	0.2888796063698876 3

```
# Run PageRank personalized for vertex ["a", "b", "c", "d"] in parallel
results = g.parallelPersonalizedPageRank(resetProbability=0.15, sourceIds=["a", "b", "c", "d"], maxIter=10)
```

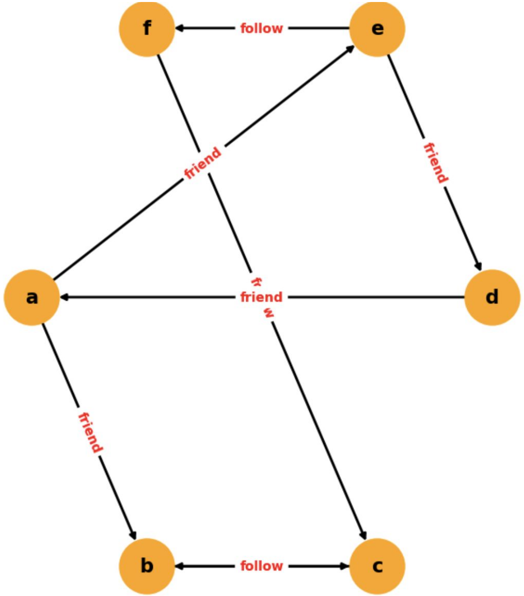
Graph Algorithms: Shortest Paths




```
g.shortestPaths(landmarks=[ "a", "d"])
  .select("id", "distances").show()
```

```
+---+----------------+
| id|       distances|
+---+----------------+
|  a|{d -> 2, a -> 0}|
|  b|              {}|
|  c|              {}|
|  d|{d -> 0, a -> 1}|
|  e|{d -> 1, a -> 2}|
|  f|              {}|
+---+----------------+
```

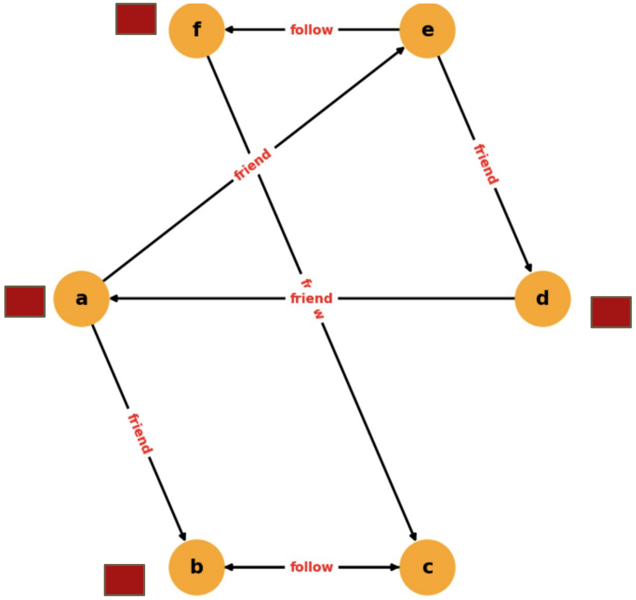
Graph Algorithms: Triangle count




```
g.triangleCount().select("id", "count").show()
```

```
+---+-----+
| id|count|
+---+-----+
|  a|    1|
|  b|    0|
|  c|    0|
|  d|    1|
|  e|    1|
|  f|    0|
+---+-----+
```

Graph Algorithms: LPA




```
g.labelPropagation(maxIter=5).show()
```

```
+---+-------+---+-----+
| id|   name|age|label|
+---+-------+---+-----+
|  a|  Alice| 34|    2|
|  b|    Bob| 36|    2|
|  c|Charlie| 30|    1|
|  d|  David| 29|    2|
|  e| Esther| 32|    5|
|  f|  Fanny| 36|    2|
+---+-------+---+-----+
```

No unique solution

API aggregateMessages

`aggregateMessages`(*aggCol*, *sendToSrc=None*, *sendToDst=None*)

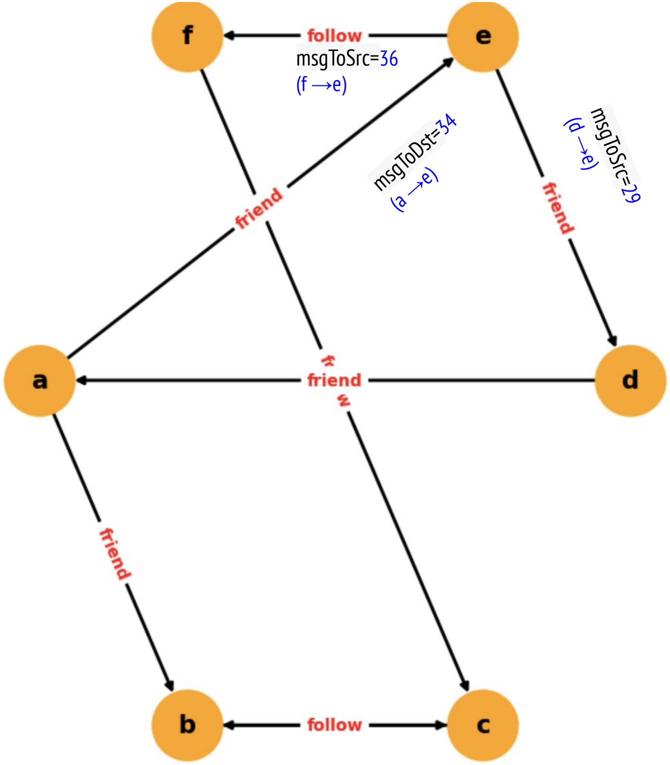
- primitive for developing graph algorithms
- send messages between vertices and aggregate messages for each vertex.
- When specifying the messages and aggregation function, the user may reference columns using the static methods in **`graphframes.lib.AggregateMessages`**.

Parameters:

- **aggCol** – the requested aggregation output either as **`pyspark.sql.Column`** or SQL expression string
- **sendToSrc** – message sent to the source vertex of each triplet either as **`pyspark.sql.Column`** or SQL expression string (default: None)
- **sendToDst** – message sent to the destination vertex of each triplet either as **`pyspark.sql.Column`** or SQL expression string (default: None)

Returns: DataFrame with columns for the vertex ID and the resulting aggregated message

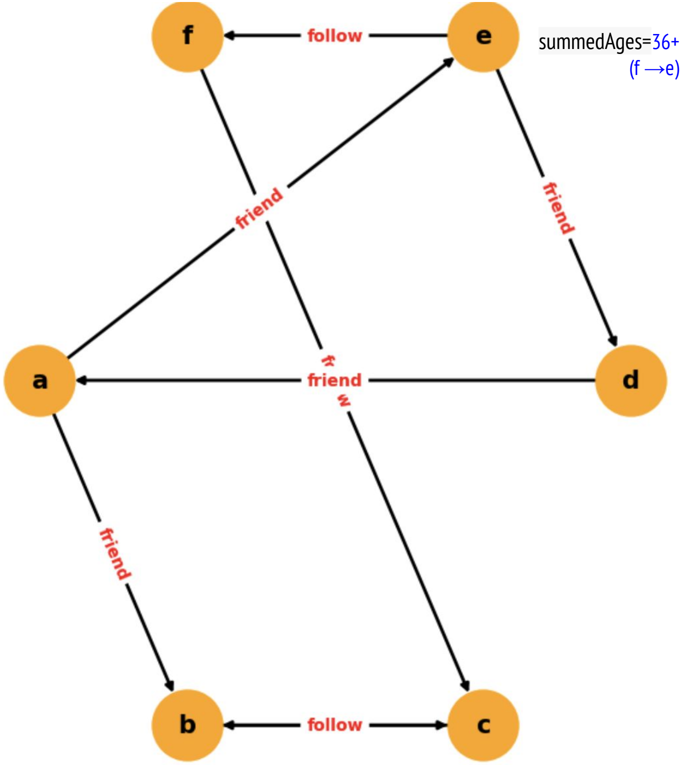
Example: aggregateMessages




```
from pyspark.sql.functions import sum
from graphframes.lib import AggregateMessages as AM
# For each user, sum the ages of the adjacent users.
msgToSrc = AM.dst["age"]
msgToDst = AM.src["age"]
agg = g.aggregateMessages(
    sum(AM.msg).alias("summedAges"),
    sendToSrc=msgToSrc,
    sendToDst=msgToDst)
```

id	name	age
a	Alice	34
b	Bob	36
c	Charlie	30
d	David	29
e	Esther	32
f	Fanny	36

Example: aggregateMessages



summedAges=36+ 29+ 34
(f →e) (d →e) (a →e)

```
from pyspark.sql.functions import sum
from graphframes.lib import AggregateMessages as AM
# For each user, sum the ages of the adjacent users.
msgToSrc = AM.dst["age"]
msgToDst = AM.src["age"]
agg = g.aggregateMessages(
    sum(AM.msg).alias("summedAges"),
    sendToSrc=msgToSrc,
    sendToDst=msgToDst)
```

id	summedAges
f	62
b	94
a	97
c	108
d	66
e	99

id	name	age
a	Alice	34
b	Bob	36
c	Charlie	30
d	David	29
e	Esther	32
f	Fanny	36

API Pregel

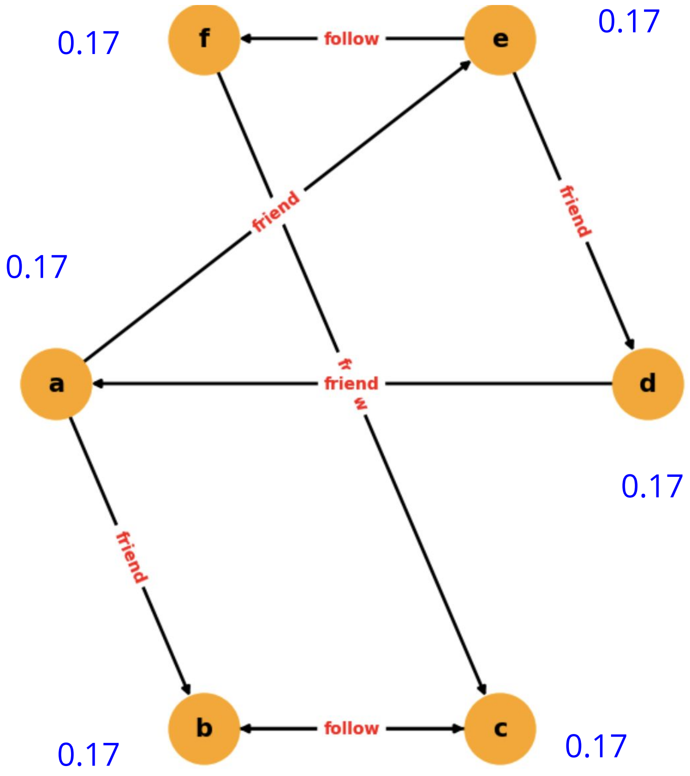
class `graphframes.lib.`**`Pregel`**(*graph*)

- implements a Pregel-like bulk-synchronous message-passing (**BSP**) API based on DataFrame operations.
- **Iterative** algorithm that computes the properties of vertices based on the properties of their neighbours
- returns a DataFrame of vertices from the last iteration.
- When a run starts, it expands the vertices DataFrame using column expressions defined by **`withVertexColumn()`**.
- *Three phases for each iteration*:
  - For each edge triplet generate messages and specify target vertices to send, described by **`sendMsgToDst()`** and **`sendMsgToSrc()`**.
  - Aggregate messages by target vertex IDs, described by **`aggMsgs()`**
  - Update additional vertex properties based on aggregated messages and states from previous iteration, described by **`withVertexColumn()`**.
- *End of computation*: when there are no more messages (at each stage, vertices that have not received a message do not send messages) or the maximum number of iterations has been reached
  - **`setMaxIter()`**: control the number of iterations

Example: PageRank computation

```
alpha = 0.15
numVertices = g.vertices.count()

ranks = g.pregel \
    .setMaxIter(5) \
   .withVertexColumn("rank", lit(1.0 / numVertices), \
        coalesce(Pregel.msg(), lit( 0.0)) * lit(1.0 - alpha) + lit(alpha /
numVertices)) \
    .sendMsgToDst(Pregel.src("rank") / Pregel.src("outDegree")) \
    .aggMsgs(sum(Pregel.msg())) \
    .run()
```

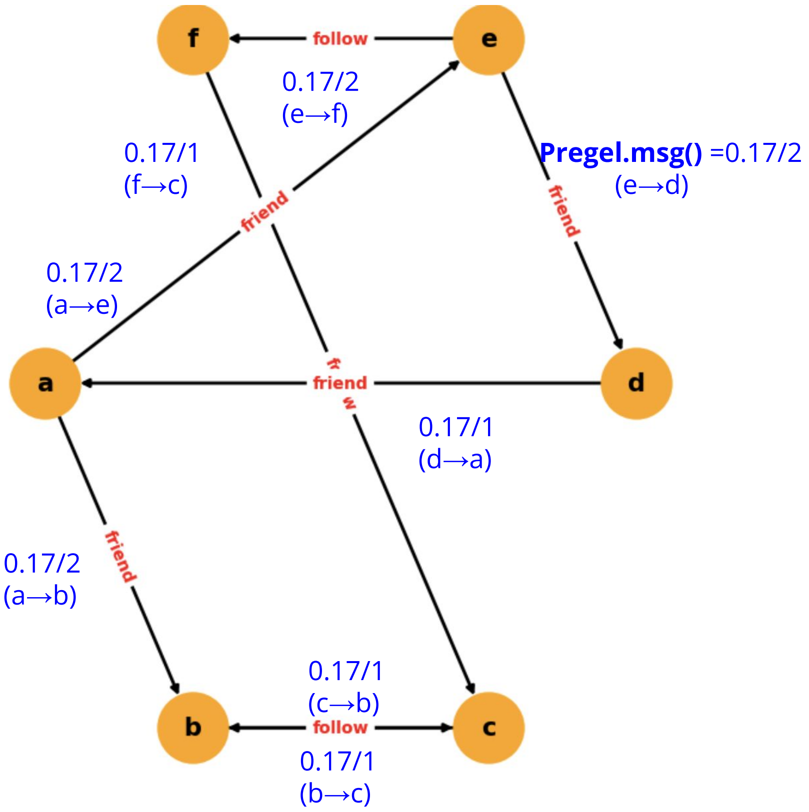
Pagerank: initialization




```
alpha = 0.15
numVertices = g.vertices.count()
ranks = g.pregel \
  .setMaxIter(5) \
  .withVertexColumn("rank", lit(1.0 / numVertices), \
      coalesce(Pregel.msg(), lit(0.0)) * lit(1.0 - alpha) + lit(alpha / numVertices)) \
  .sendMsgToDst(Pregel.src("rank") / Pregel.src("outDegree")) \
  .aggMsgs(sum(Pregel.msg())) \
  .run()
```

id	name	age	outDegree	rank
a	Alice	34	2	0.17
b	Bob	36	1	0.17
c	Charlie	30	1	0.17
d	David	29	1	0.17
e	Esther	32	2	0.17
f	Fanny	36	1	0.17

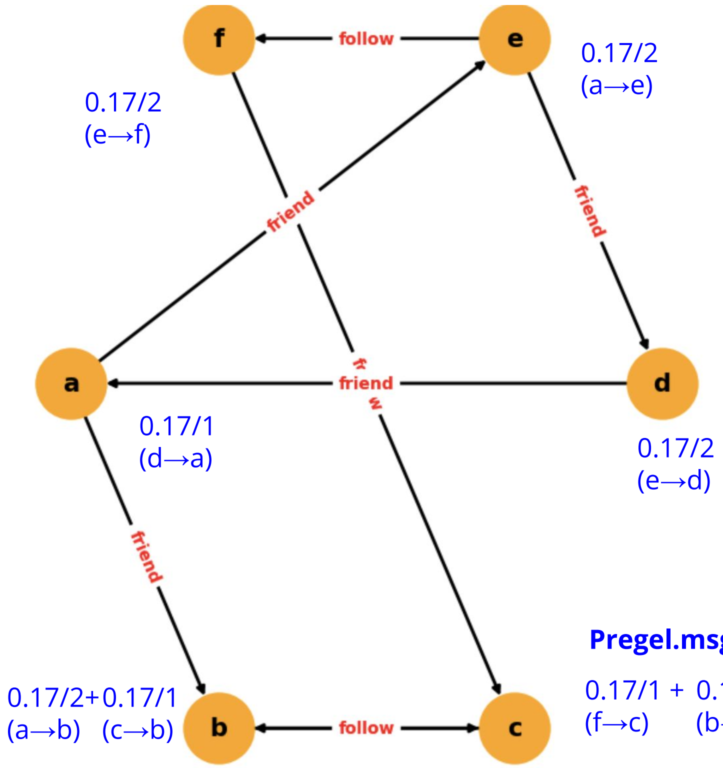
Pagerank: First Iteration - send messages




```
alpha = 0.15
numVertices = g.vertices.count()
ranks = g.pregel \
  .setMaxIter(5) \
  .withVertexColumn("rank", lit(1.0 / numVertices), \
     coalesce(Pregel.msg(), lit(0.0)) * lit(1.0 - alpha) + lit(alpha / numVertices)) \
  .sendMsgToDst(Pregel.src("rank") / Pregel.src("outDegree")) \
  .aggMsgs(sum(Pregel.msg())) \
  .run()
```

id	name	age	outDegree	rank
a	Alice	34	2	0.17
b	Bob	36	1	0.17
c	Charlie	30	1	0.17
d	David	29	1	0.17
e	Esther	32	2	0.17
f	Fanny	36	1	0.17

Pagerank: First Iteration - aggregate messages



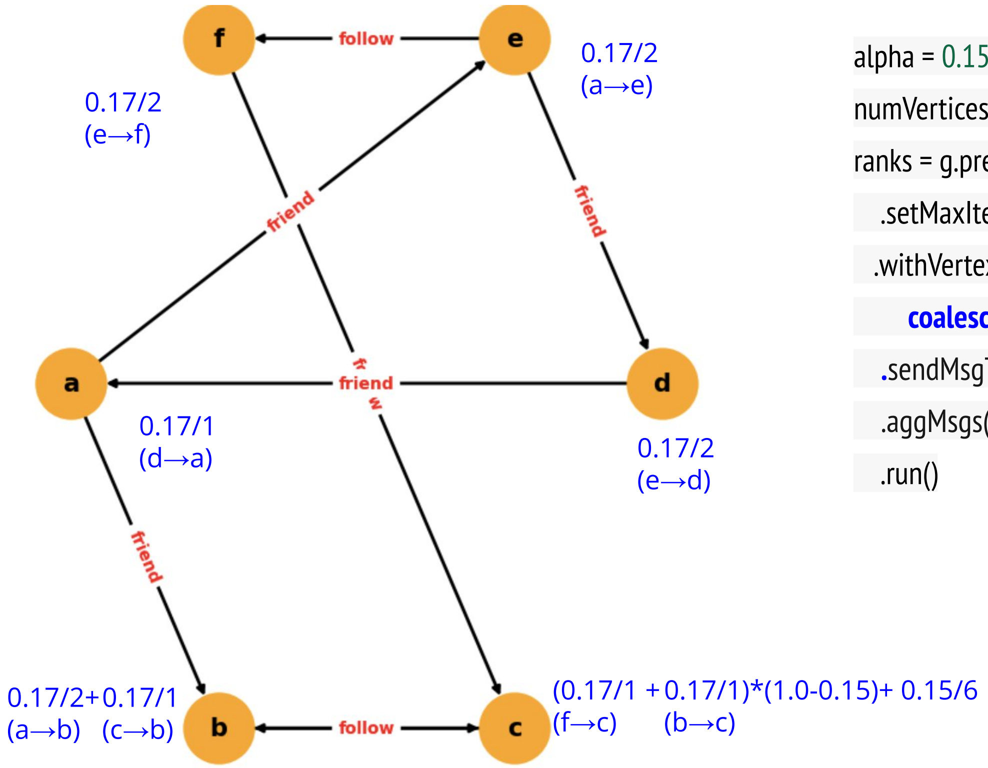
Pregel.msg() =

0.17/1 + 0.17/1
(f→c) (b→c)

```
alpha = 0.15
numVertices = g.vertices.count()
ranks = g.pregel \
  .setMaxIter(5) \
  .withVertexColumn("rank", lit(1.0 / numVertices), \
      coalesce(Pregel.msg(), lit(0.0)) * lit(1.0 - alpha) + lit(alpha / numVertices)) \
  .sendMsgToDst(Pregel.src("rank") / Pregel.src("outDegree")) \
  .aggMsgs(sum(Pregel.msg())) \
  .run()
```

id	name	age	outDegree	rank
a	Alice	34	2	0.17
b	Bob	36	1	0.17
c	Charlie	30	1	0.17
d	David	29	1	0.17
e	Esther	32	2	0.17
f	Fanny	36	1	0.17

Pagerank: First Iteration - update ranks




```
alpha = 0.15
numVertices = g.vertices.count()
ranks = g.pregel \
  .setMaxIter(5) \
  .withVertexColumn("rank", lit(1.0 / numVertices), \
      coalesce(Pregel.msg(), lit(0.0)) * lit(1.0 - alpha) + lit(alpha / numVertices)) \
  .sendMsgToDst(Pregel.src("rank") / Pregel.src("outDegree")) \
  .aggMsgs(sum(Pregel.msg())) \
  .run()
```

id	name	age	outDegree	rank
a	Alice	34	2	0.17
b	Bob	36	1	0.24
c	Charlie	30	1	0.31
d	David	29	1	0.1
e	Esther	32	2	0.1
f	Fanny	36	1	0.1