Big data: architectures and data analytics

Graph Analytics in Spark

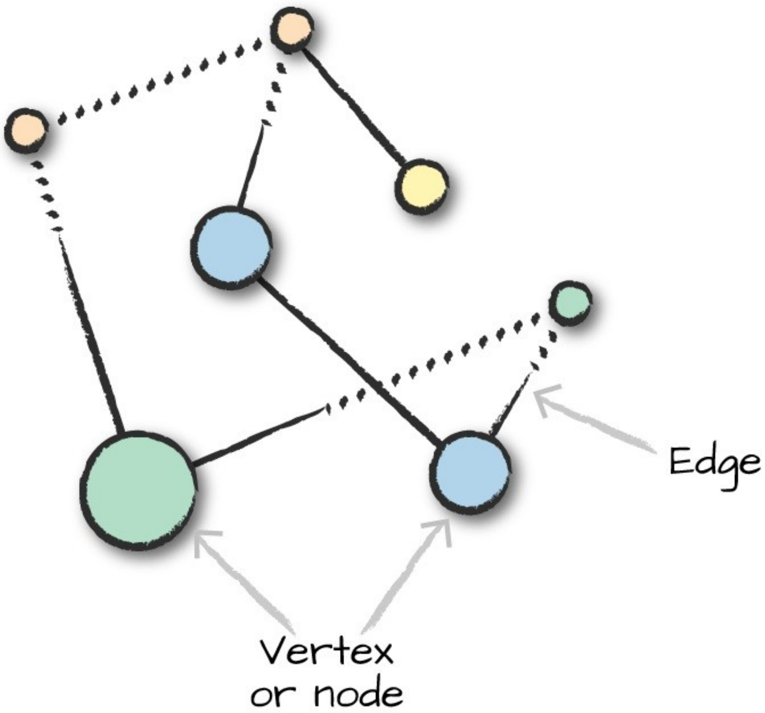
Part 1

Graphs: Introduction

Graph analytics

- Graphs are data structures composed of nodes and edges
- Nodes are denoted as $V=\{v_1,v_2,\dots,v_n\}$ and edges are denoted as $E=\{e_1,e_2,\dots,e_m\}$
- Graph analytics is the process of analyzing relationship between nodes and edges

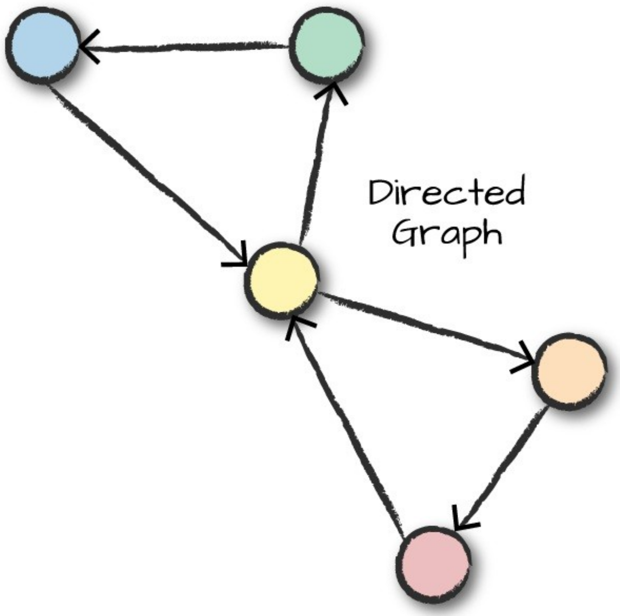
Graph analytics



Nodes, edges and weights

- Graphs are undirected if edges do not have a direction
- Otherwise they are called directed graphs
- Edges and nodes can have data associated with them → weight/label
- E.g., an edge weight may represent the strength of the relationship
- E.g., a node label may be the string associated with the name of the vertex

Nodes, edges and weights

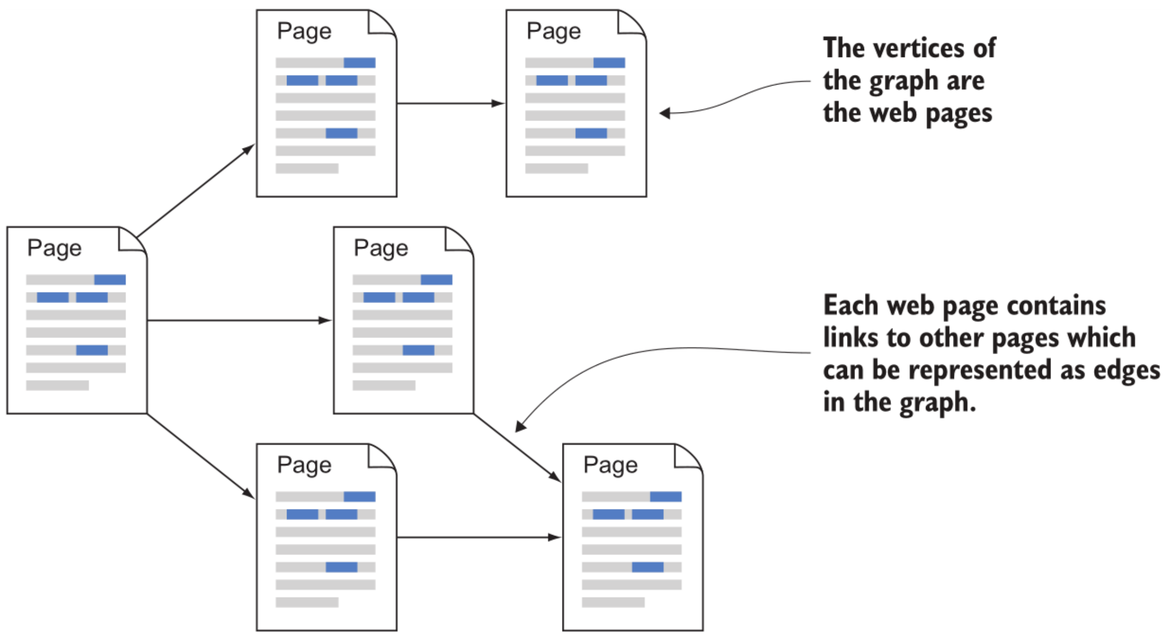


Why graph analytics?

- Graphs are natural way of describing relationship
- Practical example of analytics over graphs: ranking web pages (Google PageRank), detecting credit card fraud, determine importance of infrastructure in electrical networks,…

Graph structure in the web

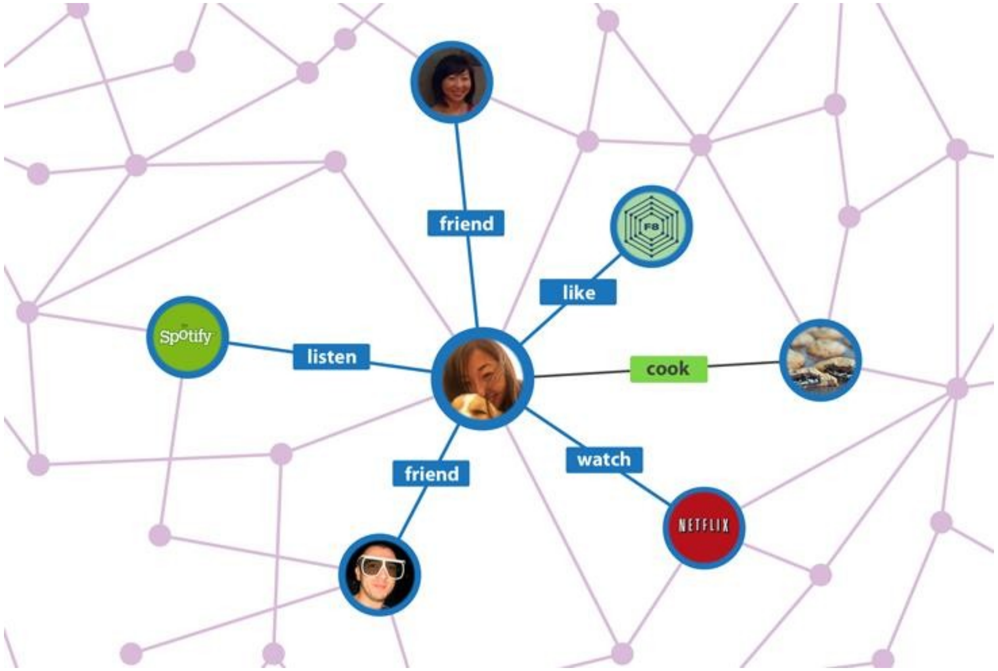
Importance and rank of web pages



Courtesy of “tutorial.razi.net.my”

Graph structure in the web

Social network structure and web usage



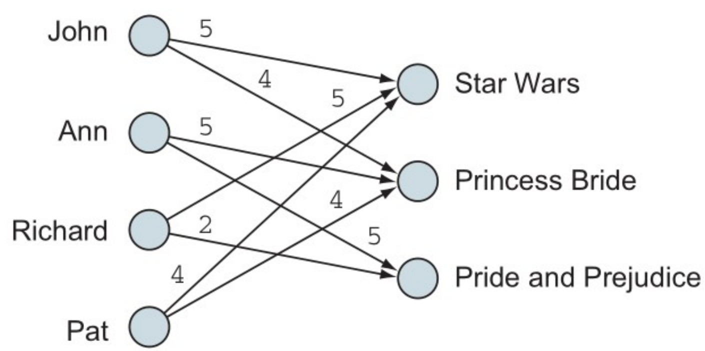
Courtesy of “tutorial.razi.net.my”

Graph structure in the web

Movies seen by users

Sparse matrix

$$
\begin{array}{r|ccc|}
 & \text{Star Wars} & \text{Princess Bride} & \text{Pride and Prejudice} \\
\text{John} & 5 & 4 & \\
\text{Ann} & & 5 & 5 \\
\text{Richard} & 5 & & 2 \\
\text{Pat} & 4 & 4 &
\end{array}
$$

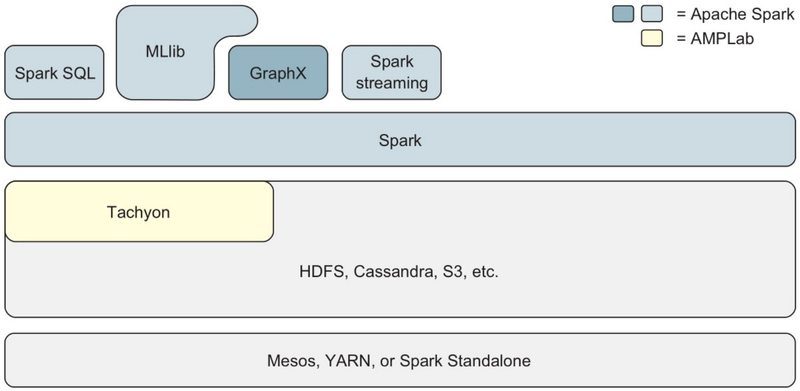


Courtesy of “tutorial.razi.net.my”

Spark GraphX and GraphFrames

GraphX

- Library **RDD-based** for performing graph processing
- Core part of Spark



GraphX

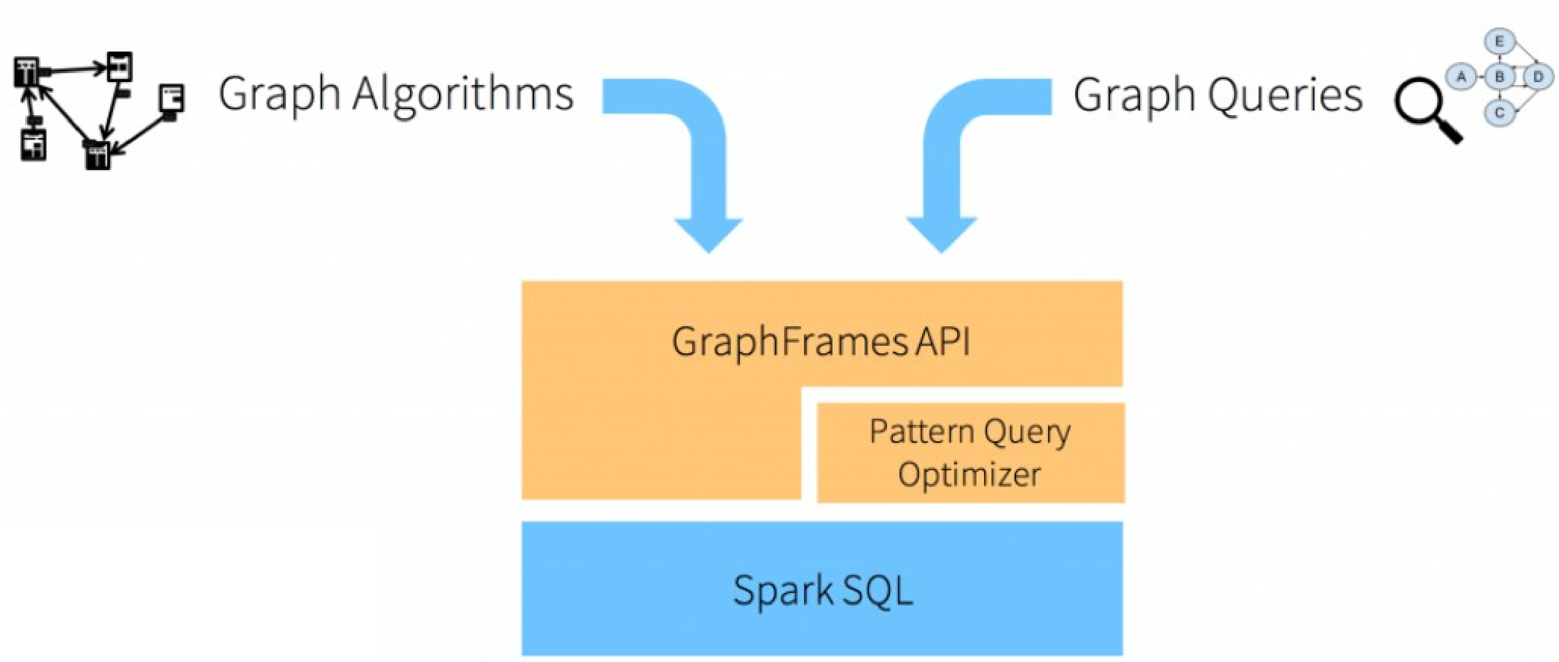
- Low level interface with RDD
- Very powerful → many application and libraries built on top of it
- However, not easy to use or optimize (as RDD in general)

GraphFrames

- Library **DataFrame**-based for performing graph processing
- Spark external package built on top of GraphX
- May be merged with core of Spark in the future

https://spark-packages.org/package/graphframes/graphframes
https://graphframes.github.io/graphframes/docs/_site/index.html

GraphFrames



https://spark-packages.org/package/graphframes/graphframes
https://graphframes.github.io/graphframes/docs/_site/index.html

GraphFrames vs. Graph databases

- Graph databases are part of the NoSQL databases
  https://en.wikipedia.org/wiki/Graph_database
- Spark is a distributed computation engine, not a database (no long-term data storage or transactions)
- GraphFrames can scale to much larger workloads than many graph databases and performs well for analytics but it does not support transactional processing and serving

Building and querying graphs with GraphFrames

Building a graph

- Define nodes and edges
- They are DataFrames with some **specifically named** columns
- By default, graphs in graphframes are **directed**

Building a graph

- Use **GraphFrame** method from **graphframes** module
- DataFrame of nodes must contain a column named "id" that stores unique vertex IDs
- DataFrame of edges must contain two columns named "src" and "dst" storing source vertex IDs and destination vertex IDs of edges, respectively.

Building a graph

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Building a graph

```
from graphframes import GraphFrame

# 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"])

# 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"])

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

Querying the graph

- GraphFrames provides simple access to vertics and edges as DataFrames
- **g.vertices** returns the DataFrame with the nodes
- **g.edges** returns the DataFrame with the edges

Querying the graph

- All DataFrame transformations/ actions are available on the edges and nodes tables
- For example, number of nodes and number of edges can be returned with the **count()** action of DataFrame

Filtering the graph

- A GraphFrame itself can’t be filtered with filter() since it is not a DataFrame
- DataFrames deducted from a Graph can be filtered
- DataFrame **filter** function (or any other function) can be used
- SQL-like: the whole condition should be quoted
- Other options in slide 46

Querying graphs: Example

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

1) Count how many nodes and edges has the graph

2) Find the youngest user's age in the graph

3) Count the number of edge "follows" in the graph.

Querying graphs: Example

```
# Count how many nodes and edges has the graph
print("Number of nodes: ",g.vertices.count())
print("Number of edges: ",g.edges.count())


# Find the youngest user's age in the graph.
# This queries the vertex DataFrame.
#option 1
g.vertices.groupBy().min("age").show()
# option 2
g.vertices.agg({"age":"min"}).show()

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

Querying graphs: Example

```
# Count how many nodes and edges has the graph
print("Number of nodes: ",g.vertices.count())
print("Number of edges: ",g.edges.count())


# Find the youngest user's age in the graph.
# This queries the vertex DataFrame.
g.vertices.groupBy().min("age").show()


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

Output:

```
Number of nodes:  7
Number of edges:  8
+--------+
|min(age)|
+--------+
|      29|
+--------+

Number of edges of type 'follows':  4
```

Analyze graphs with GraphFrames

Motif finding

- Motif finding refers to searching for structural patterns in a graph
- It uses the **find()** method of a GraphFrame
- GraphFrame motif finding uses a simple Domain-Specific Language (DSL) for expressing structural queries

DSL for Motif finding

- The basic unit of a pattern is an edge
- Vertices are denoted by parentheses (a)
- Edges are denoted by square brackets [e]
- For example, "(a)-[e]->(b)" expresses an edge e from a vertex a to a vertex b

DSL for Motif finding

- It is acceptable to omit names for vertices or edges in motifs when not needed
- E.g., "(a)-[]->(b)" expresses an edge between vertices a and b but does not assign a name to the edge
- These are called anonymous vertices and edges

DSL for Motif finding

- An edge can be negated to indicate that the edge should not be present in the graph.
- E.g., "(a)-[]->(b); !(b)-[]->(a)" finds edges from a to b for which there is no edge from b to a.

DSL for Motif finding

- **find(motif)** method of a GraphFrame will return a DataFrame of all such structures in the graph
- The DataFrame will have a column for each of the **named** elements (vertices or edges) in the motif
- More complex queries can be expressed by applying filters to the result DataFrame.
- Find can return duplicate rows

Motif finding: example 1

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Find all users that follow each other

Motif finding: example 1

```
# Search for pairs of vertices with edges in both directions between
    them.
motifs = g.find("(a)-[e]->(b); (b)-[e2]->(a)")

motifs.show()

motifs.select("a").show()
```

Motif finding: example 1

```
# Search for pairs of vertices with edges in both directions between
    them.
motifs = g.find("(a)-[e]->(b); (b)-[e2]->(a)")

motifs.show()

mot
```

Find two users that are both connected with each other.
The results is a DataFrame

Motif finding: example 1

```
# Search for pairs of vertices with edges in both directions between
    them.
motifs = g.find("(a)-[e]->(b); (b)-[e2]->(a)")

motifs.show()

motifs.select("a").show()
```

Output:

```
+---------------+--------------+---------------+--------------+
|              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]|
+---------------+--------------+---------------+--------------+
```

```
+---------------+
|              a|
+---------------+
|[c, Charlie, 30]|
|    [b, Bob, 36]|
+---------------+
```

Motif finding: example 2

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Identify chains of 4 vertices (users) such that at least 2 of the 3 edges are “friend” relationships.

Motif finding: example 2

```
from pyspark.sql.types import BooleanType

chain4 = g.find("(v1)-[e1]->(v2); (v2)-[e2]->(v3); (v3)-[e3]->(v4)")

def condition(e1,e2,e3):
    first=(e1["relationship"]== "friend")
    second=(e2["relationship"]== "friend")
    third=(e3["relationship"]== "friend")
    return (int(first)+int(second)+int(third)>=2)

from pyspark.sql.functions import udf
conditionUDF = udf(condition,BooleanType())

chainWith2Friends =
    chain4.filter(conditionUDF(chain4.e1,chain4.e2,chain4.e3))
chainWith2Friends.show()
```

Motif finding: example 2

```
from pyspark.sql.types import BooleanType

chain4 = g.find("(v1)-[e1]->(v2); (v2)-[e2]->(v3); (v3)-[e3]->(v4)")

def condition(e1,e2,e3):
    first=(e1["relationship"]== "friend")
    se
    third=(e3["relationship"]== "friend")
    return (int(first)+int(second)+int(third)>=2)

from pyspark.sql.functions import udf
conditionUDF = udf(condition,BooleanType())

chainWith2Friends =
    chain4.filter(conditionUDF(chain4.e1,chain4.e2,chain4.e3))
chainWith2Friends.show()
```

Find the chains of 4 users. The result is a DataFrame

Motif finding: example 2

```
from pyspark.sql.types import BooleanType

chain4 = g.find("(v1)-[e1]->(v2); (v2)-[e2]->(v3); (v3)-[e3]->(v4)")

def condition(e1,e2,e3):
    first=(e1["relationship"]== "friend")
    second=(e2["relationship"]== "friend")
    third=(e3["relationship"]== "friend")
    return (int(first)+int(second)+int(third)>=2)

from pyspark.sql.functions import udf
co

chainWith2Friends =
    chain4.filter(conditionUDF(chain4.e1,chain4.e2,chain4.e3))
chainWith2Friends.show()
```

We define a user defined function to compute how many “friend” edges there are in the chain

Motif finding: example 2

```
from pyspark.sql.types import BooleanType

chain4 = g.find("(v1)-[e1]->(v2); (v2)-[e2]->(v3); (v3)-[e3]->(v4)")

def condition(e1,e2,e3):
    first=(e1["relationship"]== "friend")
    second=(e2["relationship"]== "friend")
    third=(e3["relationship"]== "friend")
    return (int(first)+int(second)+int(third)>=2)

from pyspark.sql.functions import udf
conditionUDF = udf(condition,BooleanType())

chainWith2Friends =
    chain4.filter(conditionUDF(chain4.e1,chain4.e2,chain4.e3))
chainWith2Friends.show()
```

We filter the DataFrame

Motif finding: example 2

Identify chains of 4 user/vertices such that at least 2 of the 3 edges are “friend” relationships.

Output:

v1	e1	v2	e2	v3	e3	v4
[d, David, 29]	[d, a, friend]	[a, Alice, 34]	[a, e, friend]	[e, Esther, 32]	[e, f, follow]	[f, Fanny, 36]
[e, Esther, 32]	[e, d, friend]	[d, David, 29]	[d, a, friend]	[a, Alice, 34]	[a, e, friend]	[e, Esther, 32]
[d, David, 29]	[d, a, friend]	[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, b, friend]	[b, Bob, 36]	[b, c, follow]	[c, Charlie, 30]
[e, Esther, 32]	[e, d, friend]	[d, David, 29]	[d, a, friend]	[a, Alice, 34]	[a, b, friend]	[b, Bob, 36]
[a, Alice, 34]	[a, e, friend]	[e, Esther, 32]	[e, d, friend]	[d, David, 29]	[d, a, friend]	[a, Alice, 34]

Subgraphs

- Subgraphs are just smaller graphs within the larger one
- Option 1) Three direct methods for subgraph selection:
  - **filterVertices**(condition)
  - **filterEdges**(condition)
  - **DropIsolatedVertices**()
- Option 2) Process edges/nodes DataFrames and then re-create a new graph with **GraphFrame**

Subgraphs

- Option 1) These 3 methods return a new GraphFrame
- **filterEdges(condition)** filters the edges based on expression, but keep all vertices
- **filterVertices(condition)** filters the vertices based on expression, and remove edges containing any dropped vertices
- **DropIsolatedVertices()** drops vertices that are not contained in any edges of the graph

Subgraph: example 1

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Select subgraph of users older than 30 with relationships of type "friend"

Subgraph: example 1

```
# Select subgraph of users older than 30
g1 = g.filterVertices("age > 30")

# With relationships of type "friend".
g2=g1.filterEdges("relationship = 'friend'")

# Drop isolated users which are in any edges relationships
g3=g2.dropIsolatedVertices()

g3.edges.show()
g3.vertices.show()
```

Subgraph: example 1

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Select subgraph of users older than 30 with relationships of type "friend"

Output Nodes DataFrame

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

Output Edge DataFrame

```
+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  a|  e|      friend|
|  a|  b|      friend|
+---+---+------------+
```

Subgraph: example 2

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Select subgraph of users that follow people that are older than them

Subgraph: example 2

```
# 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")

# "paths" contains vertex info. Extract the edges.
e2 = paths.select("e.src", "e.dst", "e.relationship")

# Construct the subgraph
g2 = GraphFrame(g.vertices, e2).dropIsolatedVertices()
```

Subgraph: example 2

Select subgraph of users that follow people that are older than them

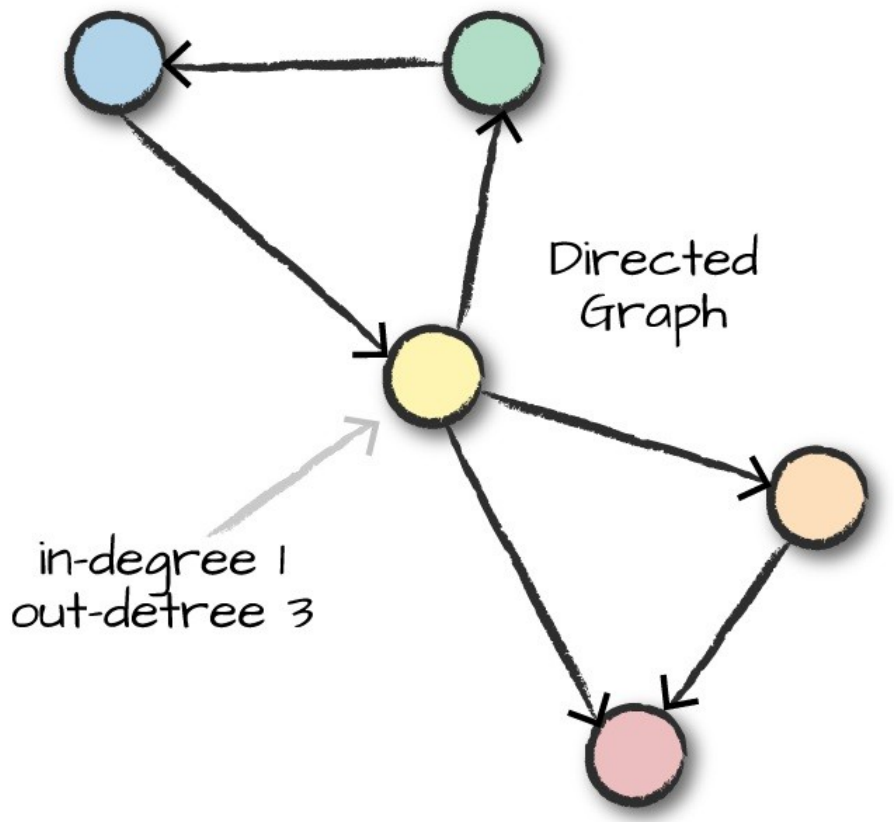
Output graph nodes

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  f|  Fanny| 36|
|  e| Esther| 32|
|  c|Charlie| 30|
|  b|    Bob| 36|
+---+-------+---+
```

Output graph edges

```
+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  e|  f|      follow|
|  c|  b|      follow|
+---+---+------------+
```

Degrees



Degrees

- **degrees** computes number of edges connected with each vertex in the graph, returned as a DataFrame with two columns:
  - "id": the ID of the vertex
  - 'degree' (integer) the degree of the vertex
- Vertices with 0 edges are not returned in the result
- **inDegrees** and **outDegrees** compute number of edges starting and ending to a vertex, respectively

Degrees: example

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

1) Return the nodes ordered by total degree (in descending order)

2) Return the nodes with outdegree higher than 1

Degrees: example

```
# Check the number of edges of each vertex
gDeg=g.degrees

# Sort the returned dataFrame
gDegSorted=gDeg.sort("degree", ascending=False)

# Check the number of outgoing edges of each vertex
gOut=g.outDegrees

# filter the returned dataFrame
gOut.filter("outDegree > 1")
```

Degrees: example

1) Return the nodes ordered by total degree descending

```
+---+------+
| id|degree|
+---+------+
|  e|     3|
|  c|     3|
|  a|     3|
|  b|     3|
|  d|     2|
|  f|     2|
+---+------+
```

2) Return the nodes with outdegree higher than 1

```
+---+---------+
| id|outDegree|
+---+---------+
|  e|        2|
|  a|        2|
+---+---------+
```

Directed vs undirected edges

- In undirected graphs the edges indicate a two-way relationship (each edge can be traversed in both directions)
- In GraphX you could use to_undirected() to create an undirected copy of the Graph, unfortunately GraphFrames does not support it (yet)
- You can convert your graph by mapping a function over the edges DataFrame that creates symmetric edges and then create a new GraphFrame

Cache graphs

- As with RDD and DataFrame, you can cache the graph in GraphFrame
- Convenient if the same (complex) graph result of (multiple) transformations is used multiple times in the same script/notebook
- Simply use the method **cache()** applied to a **GraphFrame**
- It persists the dataframe representation of vertices and edges of the graph with the default storage level