Graph Frames

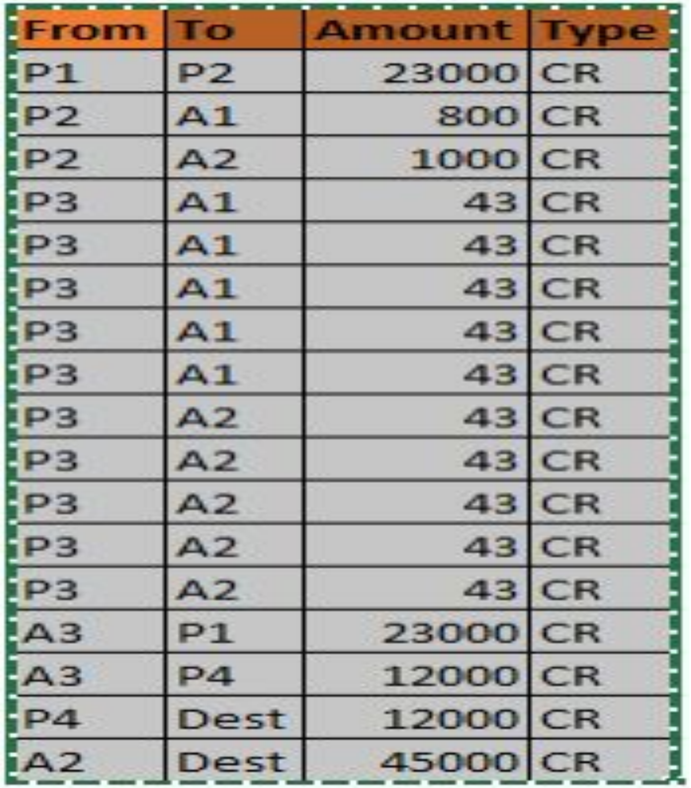
Ref: GraphFrames

Intro

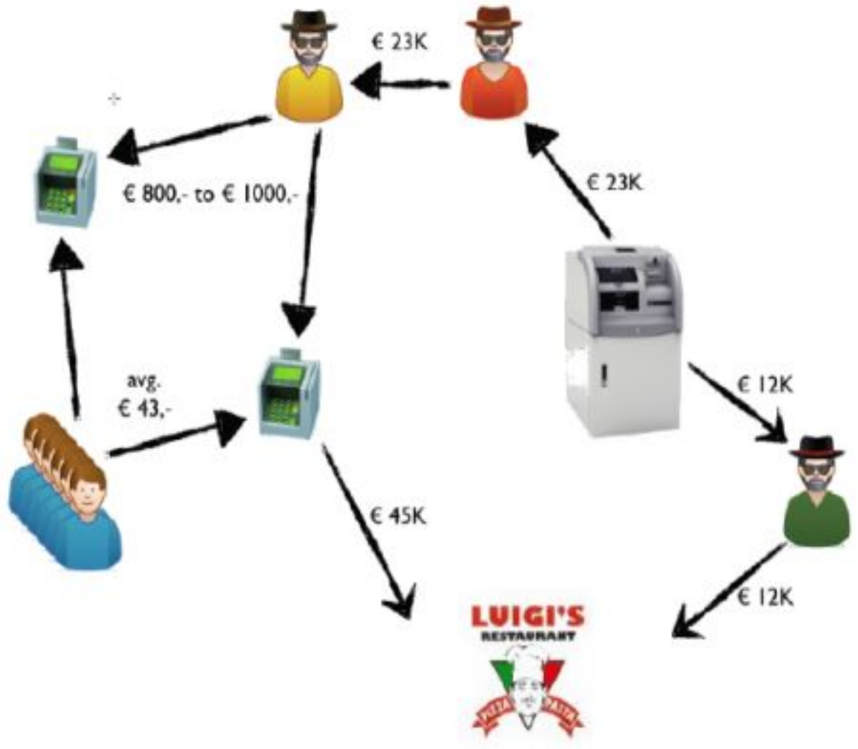
- A Graph Processing library for Apache Spark
- API are available for Java Python and Scala
- Built on top of the Spark Data Frames:
  - Powerful Queries
  - Save and Load Graphs
- Multiple Machine Learning Algorithms

Use Cases

- Fraud Detection & Analytics - Spot Fraud Rings in Their Tracks
- Motif finding.
- Determining importance of papers in bibliographic networks (i.e., which papers are most referenced).
- Ranking web pages, as Google famously used the PageRank algorithm to do.
- Identity & Access Management - Track Roles, Groups and Assets like Never Before.
- Knowledge Graph - Augment Your Knowledge Graph with Highly Contextual Search Results.
- Master Data Management - Graphs Provide a 360° View of Your Data etc ….



From	To	Amount	Type
P1	P2	23000	CR
P2	A1	800	CR
P2	A2	1000	CR
P3	A1	43	CR
P3	A1	43	CR
P3	A1	43	CR
P3	A1	43	CR
P3	A1	43	CR
P3	A2	43	CR
P3	A2	43	CR
P3	A2	43	CR
P3	A2	43	CR
P3	A2	43	CR
A3	P1	23000	CR
A3	P4	12000	CR
P4	Dest	12000	CR
A2	Dest	45000	CR

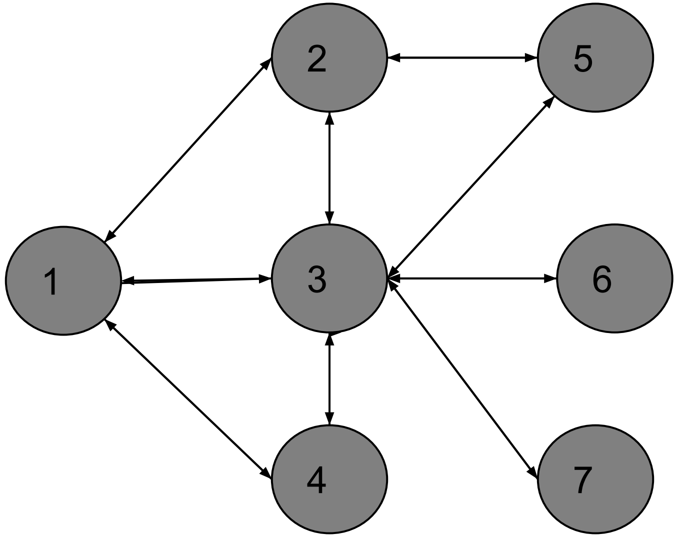


Creating Graph Frame

- Graph Frame can be created from the vertex and Edge Data frames.
  - The Vertex Data Frame should contain a special column called “id” and it can contain the information related to the “id”
  - The edges Data Frame should contain two special columns called the “src” and “dst” which lays an edge between the two vertices and other info that connects that two edges.

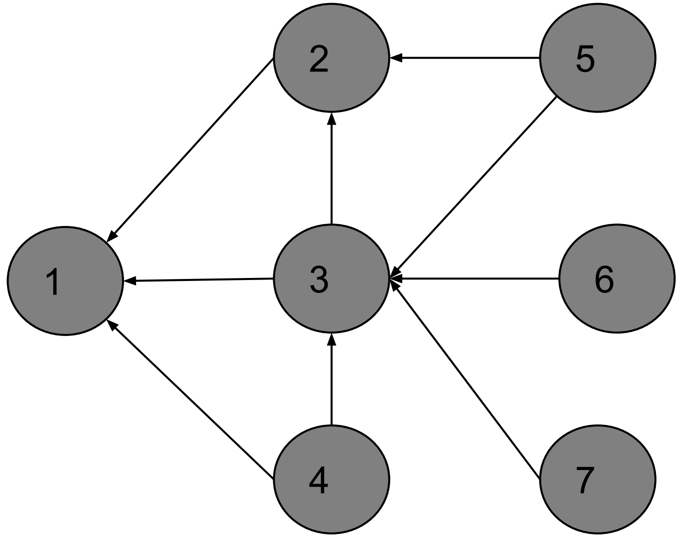
Creating a Mini Social Graph

```
vertices = spark.createDataFrame([
    ("1","Aryan",23,"M","NIT"),
    ("2","Ram",28,"M","BITS"),
    ("3","Samera",25,"F","IIT"),
    ("4","Sachin",27,"M","NIT"),
    ("5","Manoj",27,"M","NIT"),
    ("6","Mytri",27,"F","BITS"),
("7","Shiva",27,"M","IIT")],
["id","name","age","gender","university"])
```



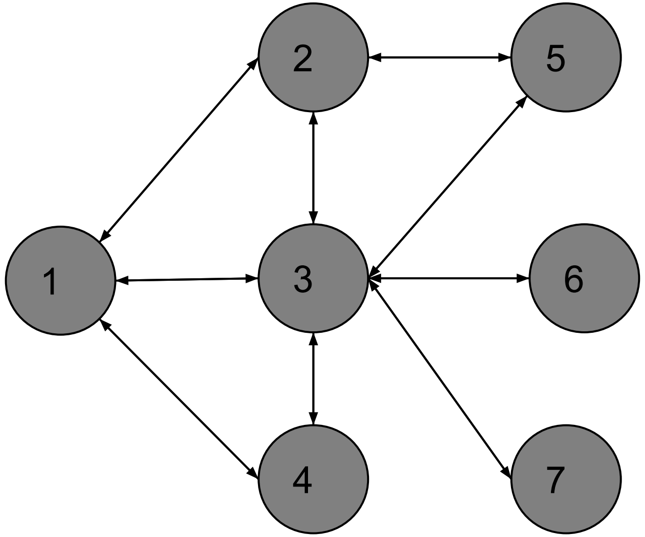
contd..

```
edges = spark.createDataFrame([
    ("1", "2", "friend"), ("2", "1", "friend"),
    ("1", "3", "friend"), ("3", "1", "friend"),
    ("1", "4", "friend"), ("4", "1", "friend"),
    ("2", "3", "friend"), ("3", "2", "friend"),
    ("3", "4", "friend"),  ("4", "3", "friend"),
    ("3", "5", "friend"), ("5", "3", "friend"),
    ("3", "6", "friend"), ("6", "3", "friend"),
    ("3", "7", "friend"), ("7", "3", "friend"),
], ["src", "dst", "relationship"])
```



Contd..

```
from graphframes import GraphFrame

g  =  GraphFrame(vertices,edges)
```



Basics

- Graph Frames which works on the top of the dataframe API are also lazy. The action the graph frame like a query will trigger the physical plan.
- Let us ask a basic query on the mini social graph we created like

  "*How many users in our mini social network has age > 25* "

- If we refer to the graph the age information is with the vertices dataframe and we can query on vertices

contd..

- g.vertices.filter(“age>25”) will be a transformation when a show() is called upon the transformation will be actually executed where by we get the physical plan.
- Similar to the Data Frame queries Graph Frame queries also get optimized by the Catalyst Optimizer which will generate the Parsed Logical Plan,Analyzed Logical Plan,Optimized Logical Plan and Physical Plan

contd..

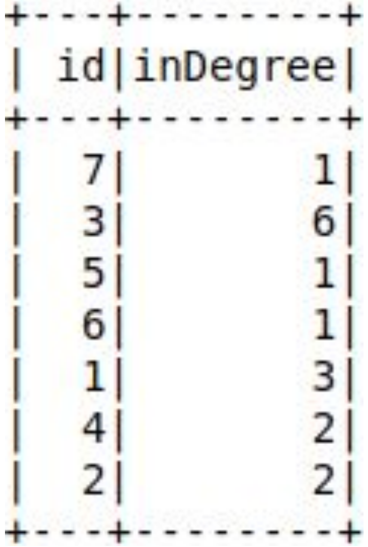
-
```
== Parsed Logical Plan ==

'Filter ('age > 25)
+- AnalysisBarrier
      +- LogicalRDD [id#0, name#1, age#2L, gender#3, university#4], false

== Analyzed Logical Plan ==
id: string, name: string, age: bigint, gender: string, university: string
Filter (age#2L > cast(25 as bigint))
+- LogicalRDD [id#0, name#1, age#2L, gender#3, university#4], false

== Optimized Logical Plan ==
Filter (isnotnull(age#2L) && (age#2L > 25))
+- LogicalRDD [id#0, name#1, age#2L, gender#3, university#4], false

== Physical Plan ==
*(1) Filter (isnotnull(age#2L) && (age#2L > 25))
+- Scan ExistingRDD[id#0,name#1,age#2L,gender#3,university#4]
```

Contd..

- "*How many users have friends greater than 2*"
- This query is on the relationship between the vertices.
- This means that the query should be on the edges.
- The friends of the user is a edge from the other users to him/her that means it is the count of the edges that connects the user from other users which is otherwise called the inDegree.

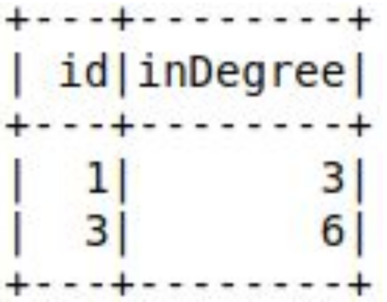
Contd..

g.inDegrees.show()



```
+---+--------+
| id|inDegree|
+---+--------+
|  7|       1|
|  3|       6|
|  5|       1|
|  6|       1|
|  1|       3|
|  4|       2|
|  2|       2|
+---+--------+
```

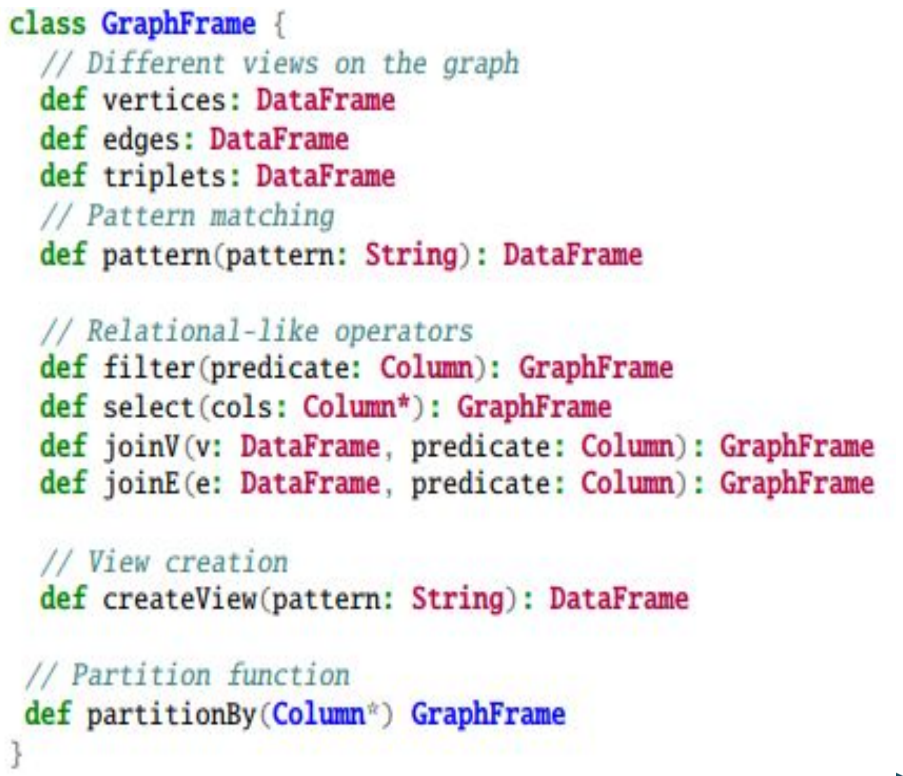
g.inDegrees.
filter("inDegree >2").
sort("id").
show()



```
+---+--------+
| id|inDegree|
+---+--------+
|  1|       3|
|  3|       6|
+---+--------+
```

Operations on Graph Frame

- Different operations that are available on the graph frame class are :

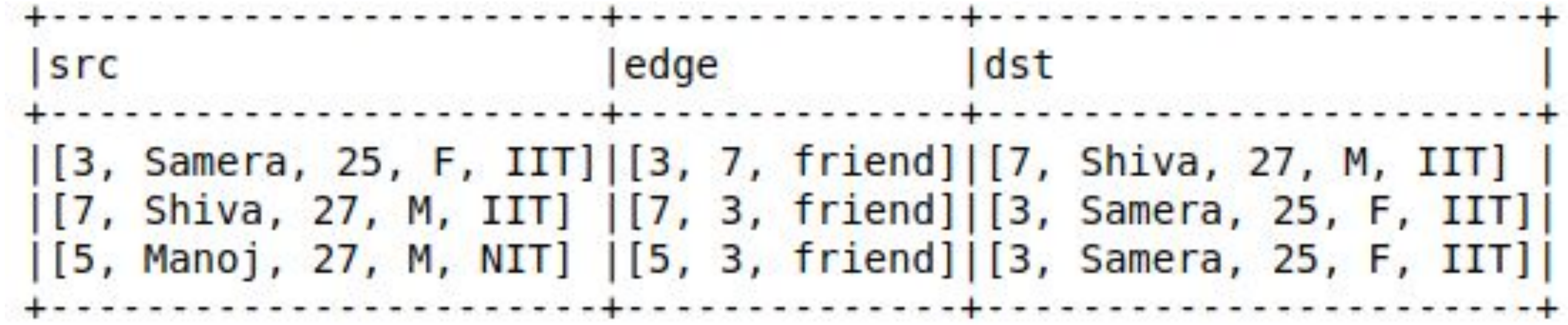


```
class GraphFrame {
  // Different views on the graph
  def vertices: DataFrame
  def edges: DataFrame
  def triplets: DataFrame
  // Pattern matching
  def pattern(pattern: String): DataFrame

  // Relational-like operators
  def filter(predicate: Column): GraphFrame
  def select(cols: Column*): GraphFrame
  def joinV(v: DataFrame, predicate: Column): GraphFrame
  def joinE(e: DataFrame, predicate: Column): GraphFrame

  // View creation
  def createView(pattern: String): DataFrame

 // Partition function
 def partitionBy(Column*) GraphFrame
}
```

contd..

Triplets:

A triplet is a data frame which has the columns of the source vertex , destination vertex and the edge.



```
+------------------------+---------------+------------------------+
|src                     |edge           |dst                     |
+------------------------+---------------+------------------------+
|[3, Samera, 25, F, IIT]|[3, 7, friend]|[7, Shiva, 27, M, IIT] |
|[7, Shiva, 27, M, IIT] |[7, 3, friend]|[3, Samera, 25, F, IIT]|
|[5, Manoj, 27, M, NIT] |[5, 3, friend]|[3, Samera, 25, F, IIT]|
+------------------------+---------------+------------------------+
```

Contd.

Patterns :

- Graphs support a pattern view which is similar to Cypher pattern Language of Neo4j.
- Typical graph patterns consist of two nodes connected by a directed edge relationship, which is represented in the format ()-[]->().
- Nodes are specified using parentheses (), and relationships are specified using square brackets, [].

Contd..

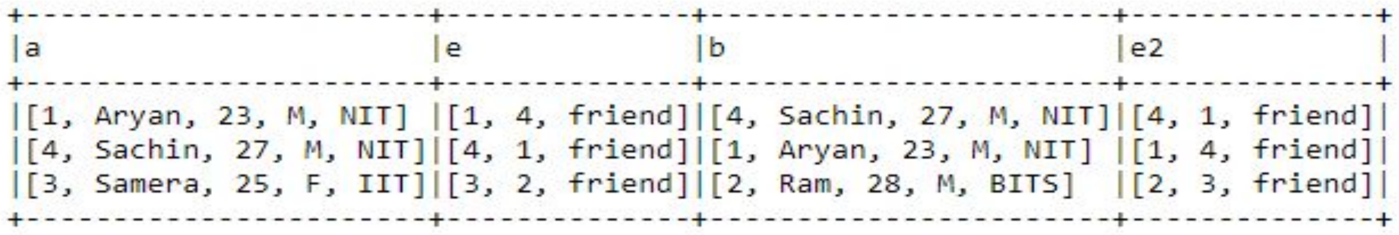
- Nodes and relationships are linked using an arrow-like syntax to express edge direction.
- The same node may be referenced in multiple relationships, allowing relationships to be composed into complex patterns.

Motif - Motif Finding

- Motif : The subgraph or pattern that repeats themselves in a graph are called Motifs.
- Motif Finding : The search for the pattern that repeats in the Graph is called Motif Finding.
- Motif Finding can be done with the help of patterns that graph Class provides us.

Contd..

- g.find("(a)-[e]->(b); (b)-[e2]->(a)").show(3)

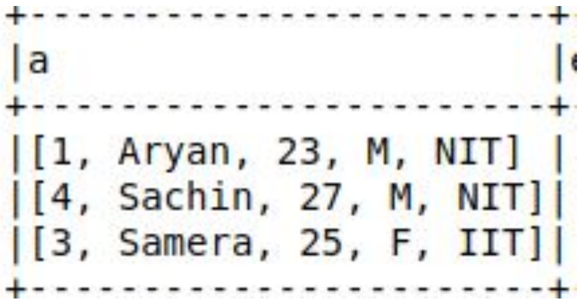


```
+----------------------+--------------+----------------------+-------------+
|a                     |e             |b                     |e2           |
+----------------------+--------------+----------------------+-------------+
|[1, Aryan, 23, M, NIT] |[1, 4, friend]|[4, Sachin, 27, M, NIT]|[4, 1, friend]|
|[4, Sachin, 27, M, NIT]|[4, 1, friend]|[1, Aryan, 23, M, NIT] |[1, 4, friend]|
|[3, Samera, 25, F, IIT]|[3, 2, friend]|[2, Ram, 28, M, BITS]  |[2, 3, friend]|
+----------------------+--------------+----------------------+-------------+
```

- It gives the list of all the bidirectional edges in the graph.
- The count of the bidirectional edges in the graph are
- g.find(("(a)-[e]->(b);(b)-[e2]->(a)")).count() → 18

Contd..

- Internally the motif of (a)-[ ]->(b) will call nestAsCol(df,colName) method to use data frame g.vertices and create a one_column_vertices_df in the pattern and converts that into a column called “a”. Similarly for “b”. It will create a join with the Current result and iterate over it.



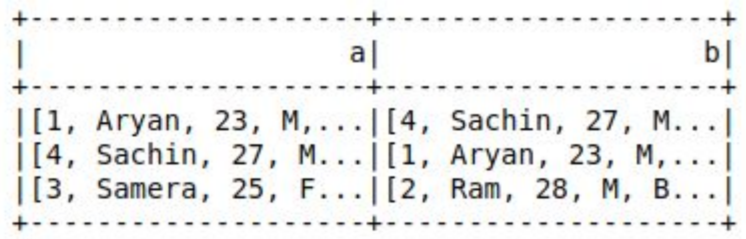
```
+---------------------+
|a                    |
+---------------------+
|[1, Aryan, 23, M, NIT] |
|[4, Sachin, 27, M, NIT]|
|[3, Samera, 25, F, IIT]|
+---------------------+
```

Contd..

- Pattern is expressed as union of edges and the different patterns are joined by semicolon.
- The [e] will create a column with the name e and has the vertices that connects the edges and the common properties that connect them.--> Named Edge.
- We can create anonymous edges also like [ ] which do not have a edge name. The data frame that will be created do not have the edge name.

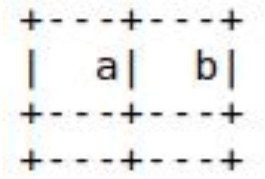
contd..

- g.find(("(a)-[]->(b);(b)-[]->(a)")).show(3)



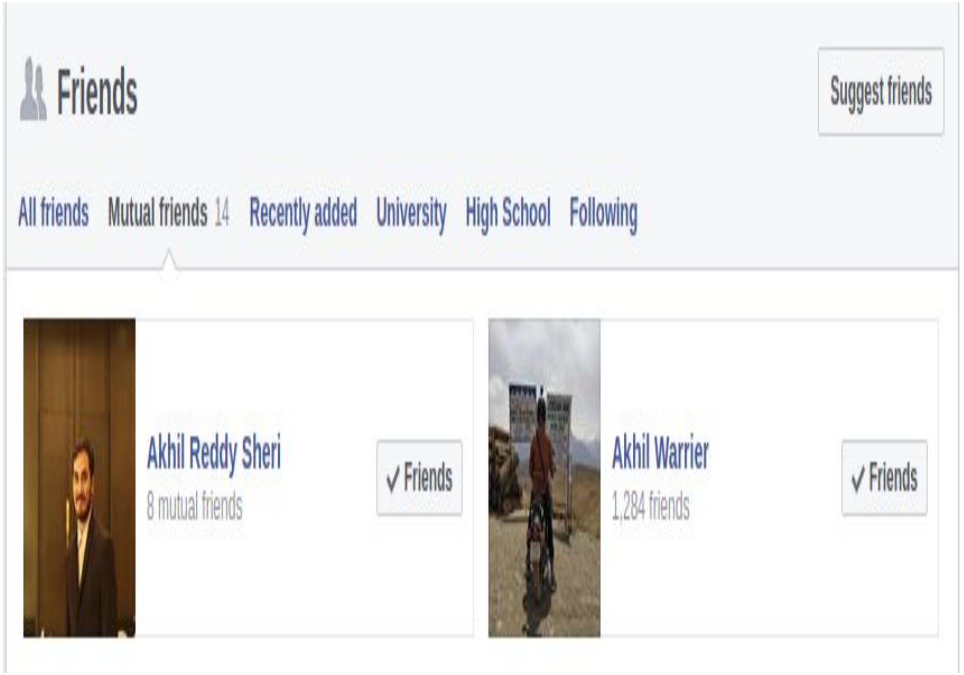
```
+--------------------+--------------------+
|                   a|                   b|
+--------------------+--------------------+
|[1, Aryan, 23, M,...|[4, Sachin, 27, M...|
|[4, Sachin, 27, M...|[1, Aryan, 23, M,...|
|[3, Samera, 25, F...|[2, Ram, 28, M, B...|
+--------------------+--------------------+
```

- We can also check for the complex queries like g.find(("(a)-[]->(b);!(b)-[]->(a)")).show(3)



```
+---+---+
|  a|  b|
+---+---+
+---+---+
```

Mutual Friends


Friends

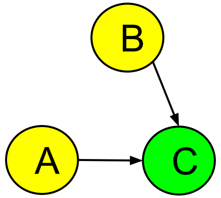
Suggest friends

All friends Mutual friends 14 Recently added University High School Following

Akhil Reddy Sheri
8 mutual friends
✓ Friends

Akhil Warrier
1,284 friends
✓ Friends

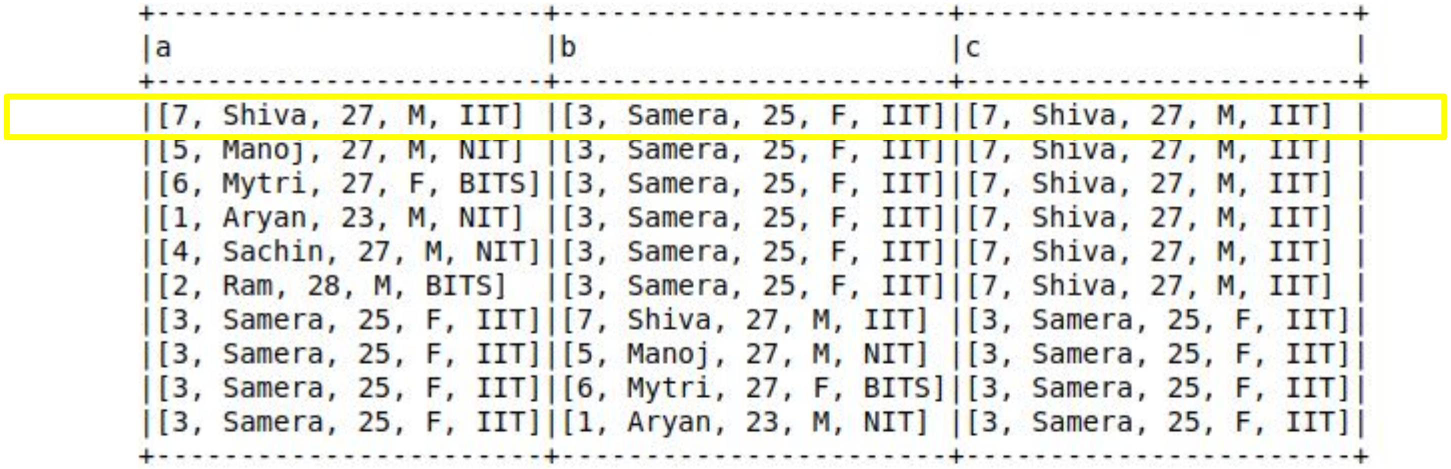
Mutual Friends can be found from the motif findings.


$$\begin{tikzcd} & B \arrow[d] \\ A \arrow[r] & C \end{tikzcd}$$

The mutual friend is like the common destiny for the vertices and we have to find the patterns for the common destinies for each vertex and filter them on one user to know his/her mutual friends.

Contd..

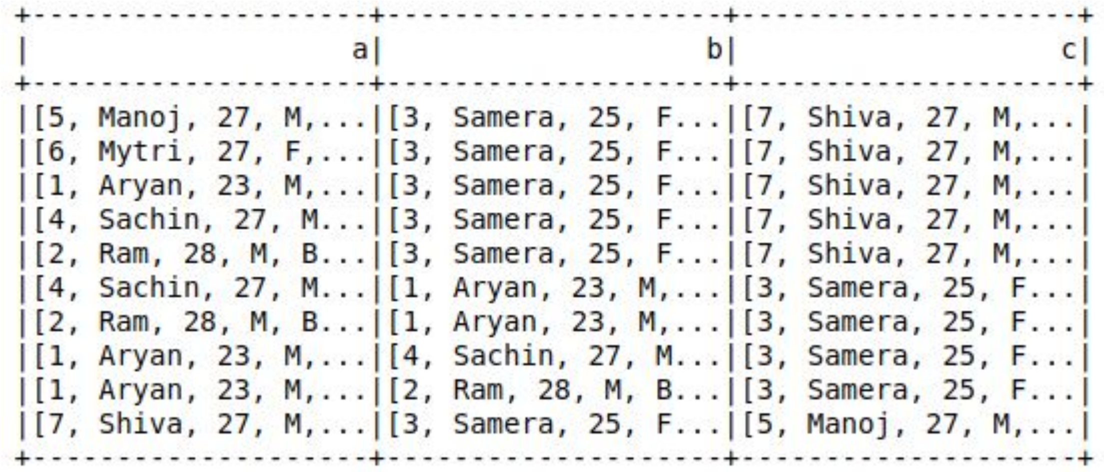
- The pattern for finding the mutual friends will be …
- g.find("(a)-[]->(b);(b)-[]->(c)").show()
- The problem with the above pattern is we are not recommending the Shiva to become friend of the other



```
+--------------------+--------------------+--------------------+
|a                   |b                   |c                   |
+--------------------+--------------------+--------------------+
|[7, Shiva, 27, M, IIT] |[3, Samera, 25, F, IIT]|[7, Shiva, 27, M, IIT] |
|[5, Manoj, 27, M, NIT] |[3, Samera, 25, F, IIT]|[7, Shiva, 27, M, IIT] |
|[6, Mytri, 27, F, BITS]|[3, Samera, 25, F, IIT]|[7, Shiva, 27, M, IIT] |
|[1, Aryan, 23, M, NIT] |[3, Samera, 25, F, IIT]|[7, Shiva, 27, M, IIT] |
|[4, Sachin, 27, M, NIT]|[3, Samera, 25, F, IIT]|[7, Shiva, 27, M, IIT] |
|[2, Ram, 28, M, BITS]  |[3, Samera, 25, F, IIT]|[7, Shiva, 27, M, IIT] |
|[3, Samera, 25, F, IIT]|[7, Shiva, 27, M, IIT] |[3, Samera, 25, F, IIT]|
|[3, Samera, 25, F, IIT]|[5, Manoj, 27, M, NIT] |[3, Samera, 25, F, IIT]|
|[3, Samera, 25, F, IIT]|[6, Mytri, 27, F, BITS]|[3, Samera, 25, F, IIT]|
|[3, Samera, 25, F, IIT]|[1, Aryan, 23, M, NIT] |[3, Samera, 25, F, IIT]|
+--------------------+--------------------+--------------------+
```

Contd..

- To continue with the pattern we need to filter those that have same columns in the a and c.
- g.find("(a)-[]->(b);(b)-[]->(c)").filter("a.id!=c.id").show()

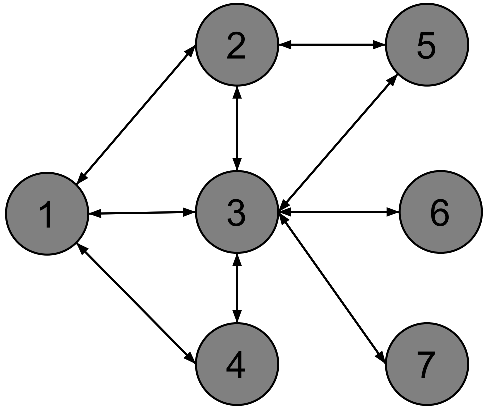


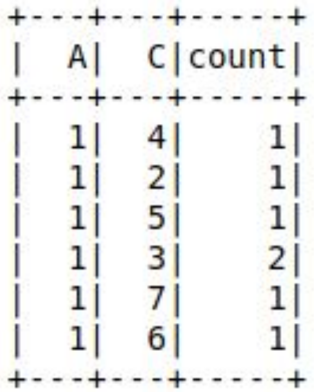
```
+--------------------+--------------------+--------------------+
|                   a|                   b|                   c|
+--------------------+--------------------+--------------------+
|[5, Manoj, 27, M,...|[3, Samera, 25, F...|[7, Shiva, 27, M,...|
|[6, Mytri, 27, F,...|[3, Samera, 25, F...|[7, Shiva, 27, M,...|
|[1, Aryan, 23, M,...|[3, Samera, 25, F...|[7, Shiva, 27, M,...|
|[4, Sachin, 27, M...|[3, Samera, 25, F...|[7, Shiva, 27, M,...|
|[2, Ram, 28, M, B...|[3, Samera, 25, F...|[7, Shiva, 27, M,...|
|[4, Sachin, 27, M...|[1, Aryan, 23, M,...|[3, Samera, 25, F...|
|[2, Ram, 28, M, B...|[1, Aryan, 23, M,...|[3, Samera, 25, F...|
|[1, Aryan, 23, M,...|[4, Sachin, 27, M...|[3, Samera, 25, F...|
|[1, Aryan, 23, M,...|[2, Ram, 28, M, B...|[3, Samera, 25, F...|
|[7, Shiva, 27, M,...|[3, Samera, 25, F...|[5, Manoj, 27, M,...|
+--------------------+--------------------+--------------------+
```

Count of the Mutual Friends

- We now take pattern choose only those columns that have "id" and count on the id's filtered for a particular "id".

```
motifs = g.find("(a)-[]->(b);(b)-[]->(c)").filter("a.id!=c.id")
AC = motifs.selectExpr("A.id as A", "C.id as C")
AC.groupBy("A", "C").count().filter("A = 1").show()
```





```
+---+---+-----+
|  A|  C|count|
+---+---+-----+
|  1|  4|    1|
|  1|  2|    1|
|  1|  5|    1|
|  1|  3|    2|
|  1|  7|    1|
|  1|  6|    1|
+---+---+-----+
```

Motif Finding -- Under the Hood

- Depends on the type of Pattern
- Depends on whether the currentResult already exists containing the pattern that has the column names mentioned in the pattern.

Contd . ..

Type of Patterns:

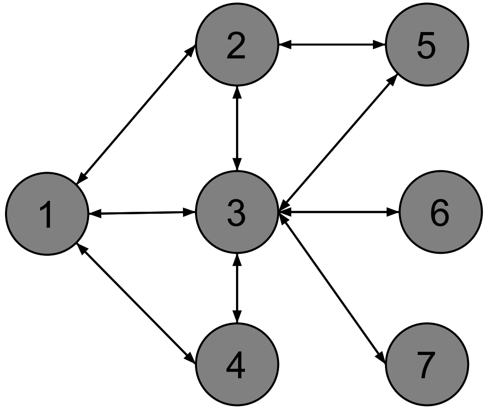
- Named Vertex (Vertex Name)
- Anonymous Vertex ( )
- Named Edge

  src: NamedVertex(srcVertexName) | AnonymousVertex,

  dst: NamedVertex(dstVertexName) | AnonymousVertex)
- Anonymous Edge

  src: NamedVertex(srcVertexName) | AnonymousVertex, dst: NamedVertex(dstVertexName) | AnonymousVertex)
- Negation

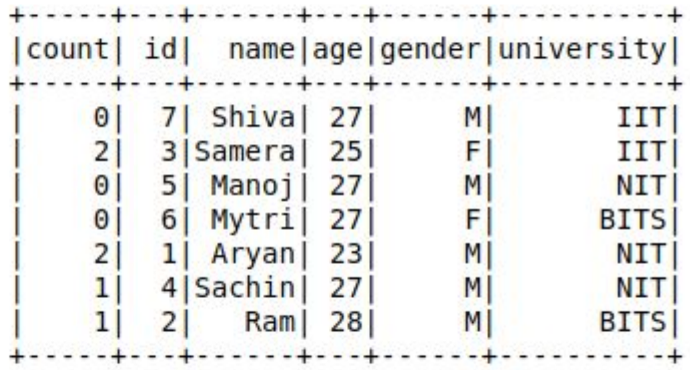
Triangle Count

- Triangle - a set of 3 vertices, provided there is an edge between any 2 of them
- For the mini Social graph we have the count as follows
- 

1 - 2 triangles
2 - 2 triangles
3 - 3 triangles
4 - 1 triangles
5 - 1 triangles
6 - 0 triangles
7 - 0 triangles

Contd ..

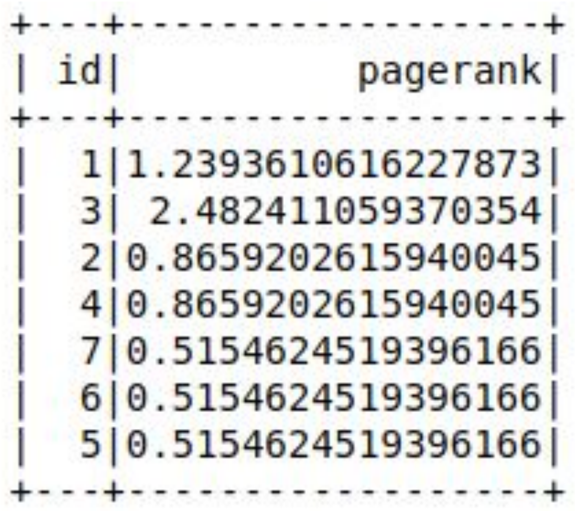
- One of the applications is that it helps in finding the fake user of the social networks
- Clustering Coefficients etc.,
- g.triangleCount( ).show()



```
+-----+---+------+---+------+----------+
|count| id|  name|age|gender|university|
+-----+---+------+---+------+----------+
|    0|  7| Shiva| 27|     M|       IIT|
|    2|  3|Samera| 25|     F|       IIT|
|    0|  5| Manoj| 27|     M|       NIT|
|    0|  6| Mytri| 27|     F|      BITS|
|    2|  1| Aryan| 23|     M|       NIT|
|    1|  4|Sachin| 27|     M|       NIT|
|    1|  2|   Ram| 28|     M|      BITS|
+-----+---+------+---+------+----------+
```

Page Rank

```
results = g.pageRank(resetProbability=0.15, tol=0.01)
results.show()
```



```
+---+------------------+
| id|          pagerank|
+---+------------------+
|  1|1.2393610616227873|
|  3| 2.482411059370354|
|  2|0.8659202615940045|
|  4|0.8659202615940045|
|  7|0.5154624519396166|
|  6|0.5154624519396166|
|  5|0.5154624519396166|
+---+------------------+
```

BFS

- 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).
- The beginning and end vertices are specified as Spark DataFrame expressions.

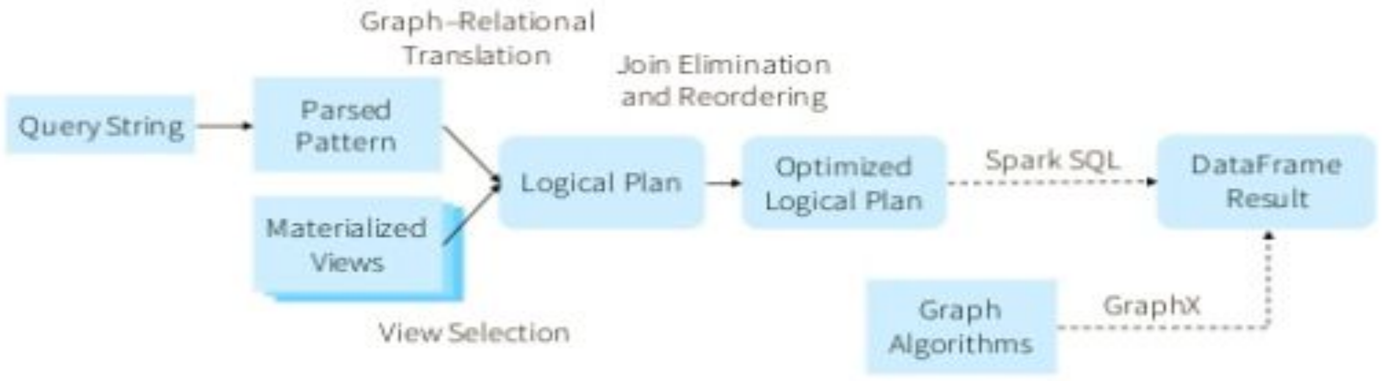
Saving and loading GraphFrames

- g.vertices.write.parquet(“/user/mahidharv/vertices”)
- g.edges.write.parquet(“/user/mahidhar/edges”)

- sameV = spark.read.parquet("/user/mahidharv/vertices")

  sameE = spark.read.parquet("/user/mahidharv/edges")

Implementation


Graph-Relational Translation
Join Elimination and Reordering
Query String
Parsed Pattern
Logical Plan
Optimized Logical Plan
Spark SQL
DataFrame Result
Materialized Views
View Selection
Graph Algorithms
GraphX