1. SPARK GRAPHX

GraphX is Spark's built-in library for graph and graph-parallel analytics. It extends RDDs with a property-graph abstraction—directed multigraphs whose vertices and edges each carry user-defined attributes—and adds a small set of graph operators (e.g., subgraph, joinVertices, aggregateMessages) together with an optimized variant of the Pregel model for iterative, message-passing computations. This design lets we combine data-parallel and graph-parallel steps in the same Spark job: construct graphs from tables, transform their structure and properties, and run iterative algorithms without moving data into a separate system.

In practice, GraphX is used for workloads such as PageRank, Connected Components, Shortest Paths, community detection, influence propagation, and entity-resolution pipelines. Developers manipulate graphs through strongly typed Scala/Java APIs using VertexRDD, EdgeRDD, and triplets; under the hood, GraphX exploits partitioned RDDs, routing tables, and join rewrites to reduce shuffle and message traffic during iterations. Empirical results from the original paper show GraphX reaching performance comparable to specialized graph systems while outperforming them in end-to-end pipelines that include ETL and feature construction—precisely because data- and graph-parallel phases live in one engine.

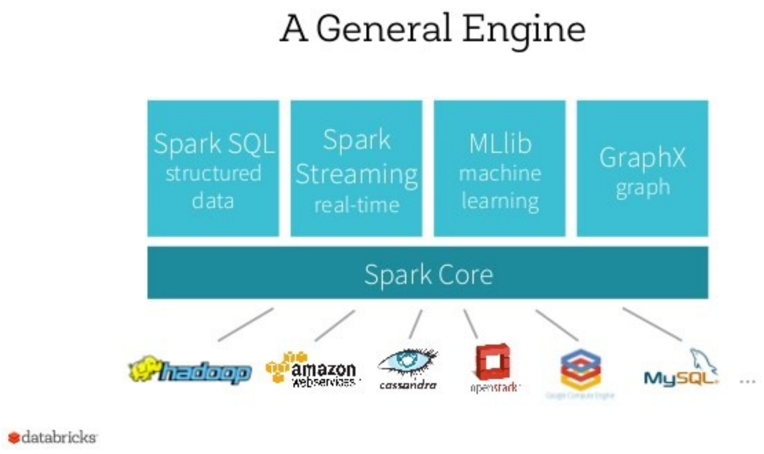


Figure 1: Components of Spark Core.

Programmatically, we can pick from high-level operators (filtering subgraphs, mapping vertex/edge properties) or express custom iterations with Pregel to send messages along edges and update vertex state until convergence. Because GraphX rides on Spark, it inherits cluster scalability, fault tolerance, and the ability to interleave SQL/DataFrame ETL with graph analytics in a single application.

GraphX's richest API surface is Scala/Java; teams needing a Python-friendly graph API on Spark often adopt GraphFrames, which provides a DataFrame-based interface and integrates tightly with Spark SQL while offering many of the same algorithms. In addition, GraphFrames has recently returned to active development, and some platforms (e.g., Databricks) document it as the recommended path when we want SQL/-DataFrame ergonomics or cross-language support.

2. TUTORIAL

This section is leveraged to help students get familiar with Spark GraphX component.
Config: Let's config Spark session with some prerequistes:

- io.graphframes:graphframes-spark4_2.13:0.9.3 in your SparkSession spark.jars.packages config.
- pip install graphframes-py

Code 1: Configure code.

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
spark = (SparkSession.builder.master("spark://bd-1:7077")\
.config("spark.jars.packages", "io.graphframes:graphframes-spark4_2.13:0.9.3,org.apache.spark:spark-sql-kafka-0-10_2.13:4.0.1,org.apache.kafka:kafka
      -clients:3.9.1,org.apache.spark:spark-streaming-kafka-0-10_2.13:4.0.1,org.apache.hadoop:hadoop-aws:3.4.1")\
.config("spark.hadoop.fs.s3a.endpoint", "http://10.1.11.5:9000")
.config("spark.hadoop.fs.s3a.access.key", "test")
.config("spark.hadoop.fs.s3a.secret.key", "hcmuthpcc")
.config("spark.hadoop.fs.s3a.path.style.access", "true")
.config("spark.hadoop.fs.s3a.aws.credentials.provider", "org.apache.hadoop.fs.s3a.SimpleAWSCredentialsProvider")
.config("spark.hadoop.fs.s3a.impl", "org.apache.hadoop.fs.s3a.S3AFileSystem")
.config("spark.hadoop.fs.s3a.committer.name", "magic")
.config("spark.hadoop.fs.s3a.committer.magic.partitioned.enabled", "true")
.config("spark.hadoop.fs.s3a.fast.upload", "true")
.config("spark.hadoop.fs.s3a.fast.upload.buffer", "disk")
.config("spark.hadoop.fs.s3a.committer.staging.conflict-mode", "replace")
.config("spark.hadoop.fs.s3a.committer.staging.abort.pending.uploads", "true")
.appName("Lab3").getOrCreate())

spark.sparkContext.setCheckpointDir("s3a://big-data/test/checkpoint/") # Set checkpoint directory
```

Data generation: Let's generate simulated transaction, user and item data and their corresponding DataFrames.

Code 2: Data generation.

```python
from pyspark.sql.types import *
from pyspark.sql.functions import *
# Transaction Data - 10 carefully crafted rows
transaction_data = [
    ("2024-01-01 10:00", "alice", "apple", 2, "sess_1"),
    ("2024-01-01 10:05", "alice", "banana", 1, "sess_1"),  # Same session co-purchase
    ("2024-01-01 11:00", "bob", "orange", 3, "sess_2"),
    ("2024-01-01 14:00", "charlie", "apple", 1, "sess_3"),
    ("2024-01-01 14:10", "charlie", "banana", 2, "sess_3"), # Same session co-purchase
    ("2024-01-02 09:00", "diana", "banana", 1, "sess_4"),   # Different day
    ("2024-01-02 15:00", "bob", "apple", 1, "sess_5"),     # Bob buys apple later (influence?)
    ("2024-01-02 16:00", "eve", "orange", 2, "sess_6"),
    ("2024-01-02 16:30", "alice", "orange", 1, "sess_7"),  # Alice tries new item
    ("2024-01-03 12:00", "diana", "apple", 1, "sess_8")    # Diana influenced by others?
]

user_data = [
    ("alice", "young", "NYC"),
    ("bob", "middle", "NYC"),
    ("charlie", "young", "LA"),
    ("diana", "middle", "LA"),
    ("eve", "senior", "NYC")
]

item_data = [
    ("apple", 1.20, "fruit"),
    ("banana", 0.80, "fruit"),
    ("orange", 1.50, "fruit")
]

# Create DataFrames
transaction_schema = StructType([
    StructField("timestamp", StringType(), True),
    StructField("user_id", StringType(), True),
    StructField("item", StringType(), True),
    StructField("quantity", IntegerType(), True),
    StructField("session_id", StringType(), True)
])

transactions_df = spark.createDataFrame(transaction_data, transaction_schema)
users_df = spark.createDataFrame(user_data, ["user_id", "age_group", "location"])
items_df = spark.createDataFrame(item_data, ["item", "price", "category"])

transactions_df.show()
users_df.show()
items_df.show()
```

Now, we will resole some problem by Spark GraphX.

Problem 1: User-Item Purchase Network Analysis.

In particular, let's find users who bought items that are 'unique' to their location - items that no other user in the same city purchased.

Let's resolve it by original **DataFrame**.

- Getting all user-item-location combinations.
- Counting users per item per location.
- Finding items with only 1 user per location.
- Joining back to find which users bought these unique items.

Code 3: Example solution by using DataFrame.

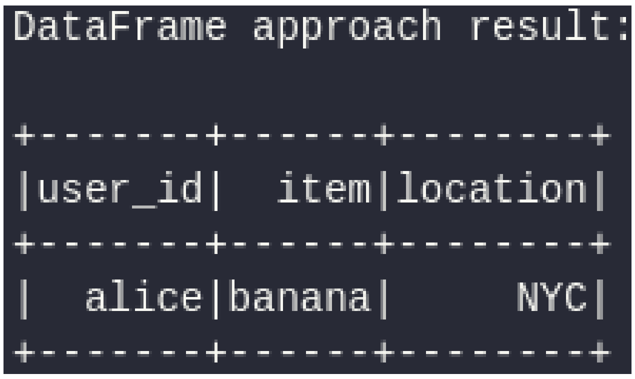
```
# Step 1: Get all user-item-location combinations
user_items = transactions_df.join(users_df, "user_id") \
    .select("user_id", "item", "location").distinct()

# Step 2: Count users per item per location
item_location_counts = user_items.groupBy("item", "location") \
    .agg({"user_id": "count"}) \
    .withColumnRenamed("count(user_id)", "user_count")

# Step 3: Find items with only 1 user per location
unique_items = item_location_counts.filter("user_count = 1") \
    .select("item", "location")

# Step 4: Join back to find which users bought these unique items
result_df = user_items.join(unique_items, ["item", "location"]) \
    .select("user_id", "item", "location")

print("DataFrame approach result:")
result_df.show()
```

Let's resolve it by original **Spark GraphX**.

Graph convert.

Which to be used as id/type/property?

- id: Unique identifier for vertices (e.g., user_id, item_id).
- type: Type of the vertex (e.g., user, item).
- property: Additional attributes or features of the vertex (e.g., location, category).



```
DataFrame approach result:

+-------+------+--------+
|user_id|  item|location|
+-------+------+--------+
|  alice|banana|     NYC|
+-------+------+--------+
```

Figure 2: DataFrame results.

Solution: Creating User-Item bipartite graph. Vertices are defined as users + items with their properties.

Code 4: Definition of vertices.

```
user_vertices = users_df.select(
    col("user_id").alias("id"),
    lit("user").alias("type"),
    col("location").alias("property")
)

item_vertices = items_df.select(
    col("item").alias("id"),
    lit("item").alias("type"),
    col("category").alias("property")
)

vertices = user_vertices.union(item_vertices)
```

Which to be used as src/dst/relationship?

- src: Source vertex id (e.g., user_id).
- dst: Destination vertex id (e.g., item_id).
- relationship: Type of relationship (e.g., "purchased", "viewed").

In this case, we want to create edges based on user-item purchases $\longrightarrow$ “purchased".

Code 5: Definition of edges.

```
# Edges: user -> item purchases
edges = transactions_df.select(
    col("user_id").alias("src"),
    col("item").alias("dst"),
    lit("purchased").alias("relationship")
).distinct()
```

Let’s creating GraphFrame.

Code 6: GraphFrame definition.

```
# In this graphframe, vertices represent users and items, while edges represent user-item interactions (purchases).
graph = GraphFrame(vertices, edges)

print("Graph created:")
print(f"Vertices: {graph.vertices.count()}")
print(f"Edges: {graph.edges.count()}")
```

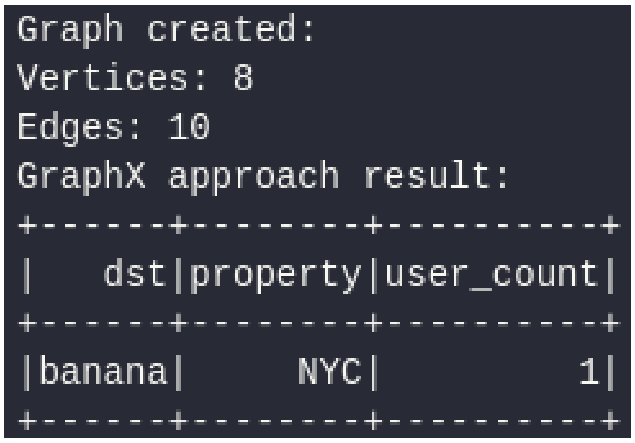
Finally, let’s resolve the problem.

- Finding items unique to each location using graph operations.

- Counting users per item per location using graph structure.

Code 7: GraphFrame operations.

```
# Find items unique to each location using graph operations
# In a graph mindset, we're trying to find items (dst) that are connected to users (src) from only one location (property).
user_item_edges = graph.edges.join(
    graph.vertices.filter("type = 'user'").select("id", "property"),
    graph.edges.src == col("id")
).select("dst", "property")


# Count users per item per location using graph structure
item_location_counts = user_item_edges.groupBy("dst", "property") \
    .count().withColumnRenamed("count", "user_count")

unique_items_graph = item_location_counts.filter("user_count = 1")

print("GraphX approach result:")
unique_items_graph.show()
```



```
Graph created:
Vertices: 8
Edges: 10
GraphX approach result:
+------+--------+----------+
|   dst|property|user_count|
+------+--------+----------+
|banana|     NYC|         1|
+------+--------+----------+
```

Figure 3: Spark GraphX results.

Problem 2: Purchase Influence Network.

Build a user similarity network and find communities of users with similar purchasing behaviors

Resolving this problem by original DataFrame is extremely complex. Let's try by yourselves and get a bonus point.

Let's resolve the problem by ***Spark GraphX***.

- Creating user-user similarity edges based on shared purchases.

```
user_purchases = transactions_df.groupBy("user_id") \
    .agg(collect_set("item").alias("items"))
```

- Creating user similarity edges (users who share 2+ items).

```
user_similarities = (user_purchases.alias("u1")     # You need alias to self-join
    .crossJoin(user_purchases.alias("u2"))          # Cartesian product - every row from u1 paired with every row from u2
    .filter(col("u1.user_id") != col("u2.user_id")) # Removes self-comparisons (alice compared to alice)
    .withColumn("shared_items", size(array_intersect("u1.items", "u2.items"))) # array_intersect("u1.items", "u2.items"): Finds common
  items between two users and counts them
    .filter("shared_items >= 2")                    # keeps only user pairs who share at least 2 items, won't be connected in the graph
   otherwise
    .select(
        col("u1.user_id").alias("src"),
        col("u2.user_id").alias("dst"),
        lit("similar").alias("relationship")        # "Draw an edge between user1 and user2, named 'similar'"
    ))
```

- Creating user-only graph for community detection.

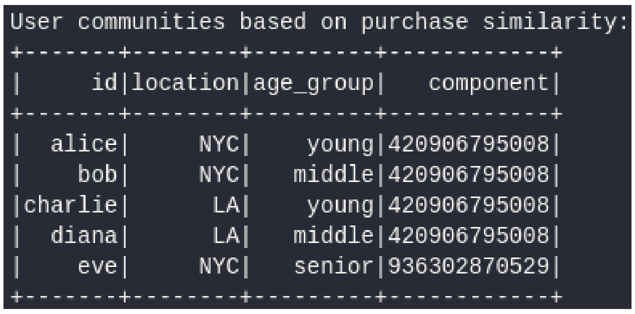
```
user_graph_vertices = users_df.select(
    col("user_id").alias("id"),
    col("location"),
    col("age_group")
)
```

- Declaring GraphFrame.

```
user_similarity_graph = GraphFrame(user_graph_vertices, user_similarities)
```

- Finding connected components (communities).

```
communities = user_similarity_graph.connectedComponents()

print("User communities based on purchase similarity:")
communities.select("id", "location", "age_group", "component").show()
```



```
User communities based on purchase similarity:
+-------+--------+---------+------------+
|     id|location|age_group|   component|
+-------+--------+---------+------------+
|  alice|     NYC|    young|420906795008|
|    bob|     NYC|   middle|420906795008|
|charlie|      LA|    young|420906795008|
|  diana|      LA|   middle|420906795008|
|    eve|     NYC|   senior|936302870529|
+-------+--------+---------+------------+
```

Figure 4: Spark GraphX results.

Problem 3: Item Co-Purchase Network & Importance Ranking.

Let's answering a question: *Which items are most 'central' to the purchasing ecosystem?* Find items that influence other item purchases.

Resolving this problem by original DataFrame is extremely complex. Let's try by yourselves and get a bonus point.

Let's resolve the problem by ***Spark GraphX***.

- Creating item co-purchase network.

```
# Items bought in same session are connected
co_purchases = (transactions_df.alias("t1") # Use alias to self-join later
    .join(transactions_df.alias("t2"), "session_id") # self-join in session_id, pairs up all transactions that happened in the same
  shopping session -> finds co-purchased items
    .filter(col("t1.item") != col("t2.item")) # Remove self-pairs (apple paired with apple,...)
    .select(                                   # Transform into edge format for GraphX
        col("t1.item").alias("src"),           # First item in the pair
        col("t2.item").alias("dst"),           # Second item in the pair
        lit("co_purchased").alias("relationship") # Edge labeled "co_purchased"
    ).distinct())                  # Remove duplicate edges (apple->banana and banana->apple are the same in undirected graph)
```

- Creating item graph.

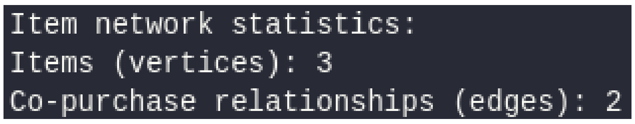
```
item_graph_vertices = items_df.select(
    col("item").alias("id"),
    col("price"),
    col("category")
)

item_graph = GraphFrame(item_graph_vertices, co_purchases)

print("Item co-purchase network:")
item_graph.edges.show()
```

- Applying PageRank to find most influential items.

```
pagerank_results = item_graph.pageRank(resetProbability=0.15, maxIter=5)

print("Item importance ranking (PageRank):")
pagerank_results.vertices \
    .select("id", "price", "pagerank") \
    .orderBy(desc("pagerank")) \
    .show()
```

- Additional graph metrics.

```
print("Item network statistics:")
print(f"Items (vertices): {item_graph.vertices.count()}")
print(f"Co-purchase relationships (edges): {item_graph.edges.count()}")
```

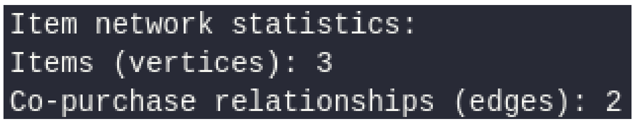


```
Item network statistics:
Items (vertices): 3
Co-purchase relationships (edges): 2
```

Figure 5: Outputs.

- In-degree (how often item is co-purchased with others).

```
indegrees = item_graph.inDegrees
print("Items ranked by co-purchase frequency:")
indegrees.orderBy(desc("inDegree")).show()
```

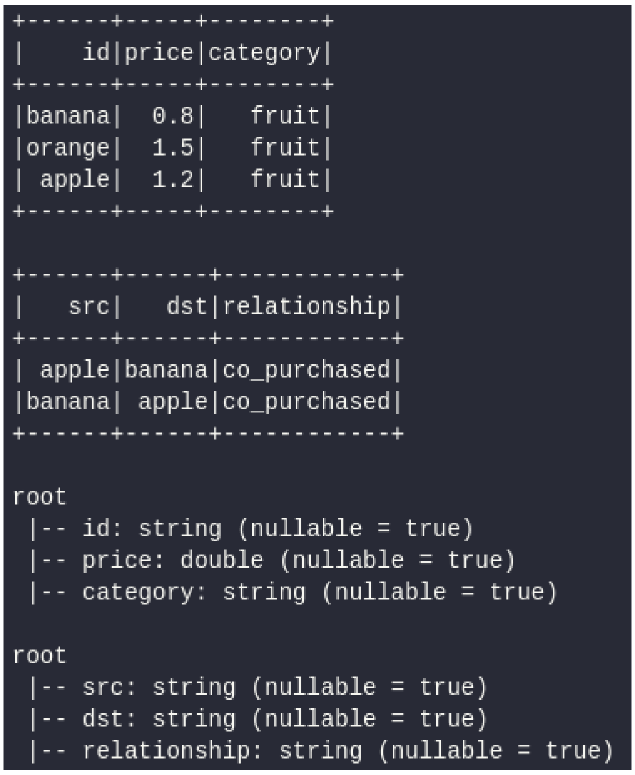


```
Item network statistics:
Items (vertices): 3
Co-purchase relationships (edges): 2
```

Figure 6: Outputs.

Let's check data persistence.

```
# Graph persistence: made of vertices and edges, store them and you store the graph
# In a production scenario, you might store these in a graph database or as Parquet files
item_graph.vertices.write.mode("overwrite").parquet("s3a://big-data/test/graphs/item_graph/vertices")
item_graph.edges.write.mode("overwrite").parquet("s3a://big-data/test/graphs/item_graph/edges")

# Re-read and verify what I said above!
v_df = spark.read.parquet("s3a://big-data/test/graphs/item_graph/vertices")
e_df = spark.read.parquet("s3a://big-data/test/graphs/item_graph/edges")
v_df.show()
e_df.show()
v_df.printSchema()
e_df.printSchema()
```



```
+------+-----+--------+
|    id|price|category|
+------+-----+--------+
|banana|  0.8|   fruit|
|orange|  1.5|   fruit|
| apple|  1.2|   fruit|
+------+-----+--------+

+------+------+------------+
|   src|   dst|relationship|
+------+------+------------+
| apple|banana|co_purchased|
|banana| apple|co_purchased|
+------+------+------------+

root
 |-- id: string (nullable = true)
 |-- price: double (nullable = true)
 |-- category: string (nullable = true)

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

Figure 7: Outputs.

3. EXERCISES

This exercises is leveraged to help students understand Spark's GraphFrame and practice them to solve problems. The requirements of exercise are shown as the followings.

Exercise 0: Prepare movie data. This is mandatory to do any of the exercises.

- Reading data from 3 topics, including:
  - Movies (with "Lab1_movies" as topic name (*i.e.*, subscribe))
  - Ratings (with "Lab1_ratings" as topic name (*i.e.*, subscribe))
  - Tags similar to ratings (with "Lab1_tags" as topic name (*i.e.*, subscribe))

  Kafka addresses are 10.1.1.10:30090, 10.1.1.27:30091, 10.1.1.203:30092.

- Defining schema for movies, ratings and tags. Subsequently, converting data type of the DataFrame to its corresponding schema.

Exercise 1: Check for popularity bias
Popularity bias occurs when a popular item is disproportionately regarded as high quality. Think of J97's songs earning millions of views despite that they all suck; this is due to view farming from his loyal fans.

- Compute the in-degree of movies (how many distinct raters) and weighted in-degree (sum of rating weights).
- Output:
  - Top-20 movies by in-degree and weighted in-degree.
  - Three single-sentence insights about the results.

Exercise 2: Get the 20 most relevant movies using PageRank.

- Run global PageRank on user -> movie graph.
- Output: Top-20 movies by global PageRank, with their genres, and PageRank score.

Exercise 3 (bonus): Motifs for polarization (disagreement)

- Find movies that are polarizing (causing disagreement/controversy). Rank them by polarized pair count (Polarized pair means two users rated the same movie with an absolute difference of at least 3.0.)
- Output:
  - A DataFrame of the top 10 most polarizing movies, must have movieId, title, and number of polarized pairs.
  - Three single-sentence insights about the results.

Submission: The works must be done in a group of at most 3.

- Compress all necessary files (sources and a report) as **CO3137_BigDataLab_Lab03_<Student ID-1>_<Student ID-2>_<Student ID-3>.zip**.
- Students must submit and demonstrate their results before the end of the class.
- Early submitters (within the lab class) get bonus points. The rest can still get a 10 if they deserve it comprehensively.