UNIVERSITY OF MARYLAND

UMD DATA605 - Big Data Systems

12.2: Graph Data Management

- **Instructor**: Dr. GP Saggese, gsaggese@umd.edu

SCIENCE ACADEMY

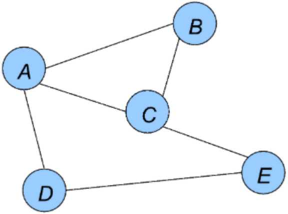
Overview

- Motivation
- Graph data models
  - RDF, Property Graph, XML
- Storing graph data
  - Neo4j
- Querying graph data
  - Cypher, SPARQL, Gremlin
- Typical graph analysis tasks
  - PageRank, clustering
- Executing graph analysis tasks
  - Google Pregel, Apache Giraph, Spark GraphX

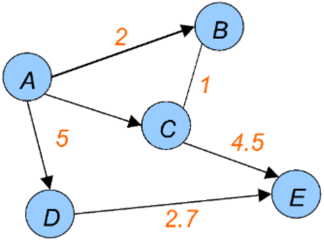
- ***Motivation***
- Graph data models
- Storing graph data
- Querying graph data
- Typical graph analysis tasks
- Executing graph analysis tasks

Graphs: Background

- A *graph (or network)* captures entities and interconnections
  - Entities represented by *vertices or nodes*
  - Interconnections called *edges* (or *links, arcs, relationships*)
- Graph theory and algorithms studied in Computer Science
  - Less work on managing graph-structured data



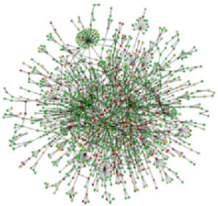
An undirected, unweighted graph



A directed, edge-weighted graph

Graph Data Structures: Motivation

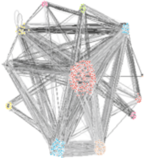
- Increasing interest in querying and reasoning about the *underlying graph structure* in a variety of disciplines



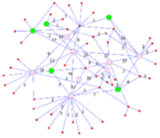
A protein-protein interaction network

Social networks

Supreme court citation network

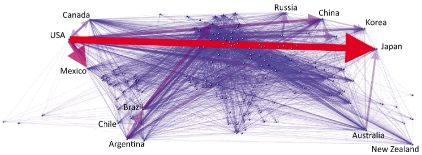


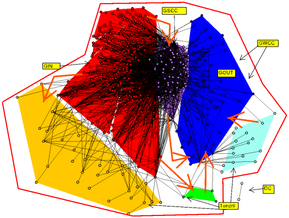
Financial transaction networks



Stock Trading Networks

Motivation





Global virtual water trade network

Federal funds networks

Citation networks

Parcel shipment networks

Collaboration networks

Knowledge Graph

Telecommunications networks

World Wide Web

Disease transmission networks

Motivation

- **Graph data structures have not changed that much over time**
  - Same problems in representing data in 1960s as today
- **What has changed in recent years**
  - Large data volumes and easier availability
  - Reasoning about graph structure provides useful insights
    - Lose information if graph structure ignored
  - Not easy to query with traditional tools (e.g., relational DBs)
    - Need specialized tools (e.g., Neo4j)
  - Hard to efficiently process graph-structured queries with existing tools
    - Dedicated solutions: Google Pregel / Apache Giraph, Spark GraphX
    - Problems worsen with increasingly large graphs in practice

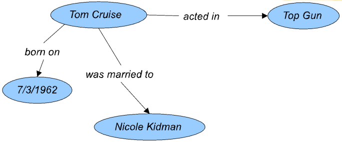
- Motivation
- ***Graph data models***
- Storing graph data
- Querying graph data
- Typical graph analysis tasks
- Executing graph analysis tasks

Knowledge Graphs

- **Representation of knowledge in the form of graphs**
  - Capture entities, relationships, properties
  - Provide structured view of real-world information
- Represent using RDF or Property Graph models
  - E.g., Google Knowledge Graph, DBpedia, Wikidata
- **Applications**
  - Enable machine understanding of complex domains
  - Support semantic search, recommendation, analytics
  - Used in industries for data integration, knowledge discovery, AI applications
- **Ontologies**
  - Provide formal representation of knowledge
  - Promote interoperability across knowledge bases

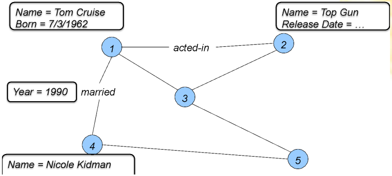
Graph Data Models: RDF

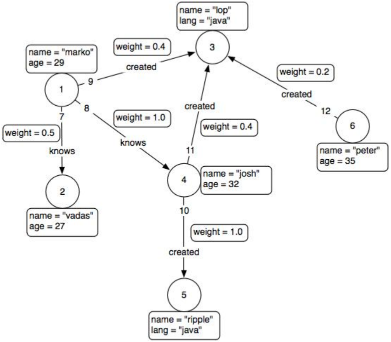
- Resource Description Framework
- RDF uses triples subject-predicate-object
  - Connects "subject" and "object" through a "predicate"
  - E.g., "TomCruise-acted-TopGun"
- **Used to represent knowledge bases**
  - Queried through SPARQL
- **Pros**
  - Standardization
    - W3C standard to model data
    - Subject and object can be URI in semantic web
  - Interoperability
    - Merge RDF data store
  - Extensibility
    - Add new nodes and relationships
    - Support ontologies



Graph Data Models: Property Graph

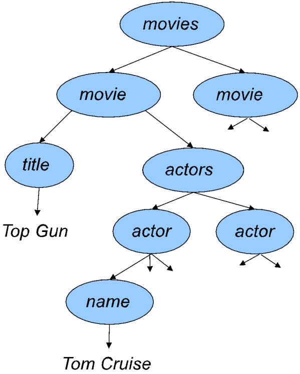
- Directed graph with nodes and edges having *properties* (*key-values*)
- Query languages
  - Cypher (e.g., Neo4j)
  - Gremlin (e.g., Apache TinkerPop)
- No universal standard
- Similar expressive power to RDFs, less "schema," harder to interoperate
- Used by many open-source graph data management tools





Graph Data Models: XML

- Common data model for flexible data representation
- Directed labeled tree
- Popular for non-tabular data exchange




```
<movies>
  <movie>
  <title>Top Gun</title>
  <actors>
    <actor>
     <name>Tom Cruise</name>
     <born>7/3/1962</born>
    </actor>
   <actor>
     ...
   </actor>
  </actors>
  </movie>
```

- Motivation
- Graph data models
- ***Storing graph data***
- Querying graph data
- Typical graph analysis tasks
- Executing graph analysis tasks

Storing Graph Data

- **File systems**
- Very simple
- No support for transactions, ACID
- Minimal functionality (e.g., must build the analysis/querying on top)
- **Relational database**
- Mature technology
- All the good stuff (SQL, transactions, ACID, toolchains)
- Minimal functionality
- **NoSQL key-value stores**
- Can handle very large datasets efficiently in a distributed fashion
- Minimal functionality
- **Graph database***
- Efficiently support for queries / tasks (e.g., graph traversals)
- Not as a mature as RDBMs
- Often no declarative language (similar to SQL)
    - You need to write programs

Graph Databases

- Many specialized graph database systems
  - E.g., Neo4j, Titan, OrientDB, AllegroGraph
- Key distinctions from relational databases
  - Manage and query graph-structured data
  - Store graph structure explicitly with pointers
    - Avoid joins, simplify graph traversals
    - Natural to write *queries* and *graph algorithms* (reachability, shortest paths)
  - Support graph query languages: SPARQL, Cypher, Gremlin
  - Rudimentary declarative interfaces
  - Applications often require programmatic interfaces
  - Provide programmatic API for arbitrary graph algorithms

- Motivation
- Graph data models
- Storing graph data
- ***Querying graph data***
- Typical graph analysis tasks
- Executing graph analysis tasks

Query Languages for Graph Databases

- Cypher
  - Designed for Property Graphs
  - Data: vertices and edges with key-value properties
  - Declarative
  - Subgraph pattern matching
  - Struggles with reachability queries
  - Native to Neo4j
- Gremlin
  - Works with RDF and Property Graphs
  - Imperative
  - Describes graph traversal
- SPARQL
  - Similar to Cypher
  - For RDF data
  - Standardized by W3C

```
MATCH (nicole:Actor
   {name:'Nicole Kidman'})-[[:ACTED_IN]->(movie
WHERE movie.year < 2007
RETURN movie
```

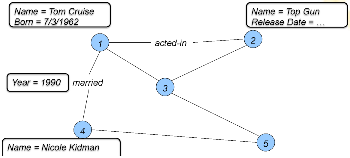
```
// calculate basic collaborative filtering for vertex 1
m = [:]
g.v(1).out('likes').in('likes').out('likes').groupCount(m)
m.sort{-it.value}

// calculate the primary eigenvector (eigenvector centrality) of a graph
m = [:]; c = 0;
g.V.as('x').out.groupCount(m).loop('x'){c++ < 1000}
m.sort{-it.value}
```

```
PREFIX  foaf: <[\textcolor{blue}{\underline{htt
SELECT ?name
       ?email
WHERE
  {
    ?person  a          foaf:Person.
    ?person  foaf:name  ?name .
    ?person  foaf:mbox  ?email .
  }
```

SCIENCE ACADEMY

Neo4j

- Graph DB storing data as Property Graph
  - Nodes, edges hold data as key-value pairs
- Focus is
  - On relationships between values
- Two querying languages
  - Cypher, Gremlin
- GUI or REST API
- Full ACID-compliant transactions
- High-availability clustering
- Incremental backups
- Run in small application or large server clusters



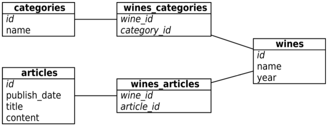
SCIENCE ACADEMY

Graph DB: Example

Specs

- Create a wine suggestion engine
- Wines categorized by
  - Varieties (e.g., Chardonnay, Pinot Noir)
  - Regions (e.g., Bordeaux, Napa, Tuscany)
  - Vintage (year grapes harvested)
- Track articles describing wines by authors
- Users track favorite wines

Relational model



- The important relationships are `produced`, `reported_on`, `grape_type`
- Create various tables
  - `wines`: (id, name, year)
  - `wines_categories` (wine_id, category_id)
  - `category` table (id, name)
  - `wines_articles` (wine_id, article_id)
  - `articles` (id, publish_date, title, content)

Labeled Property Graphs in Neo4j

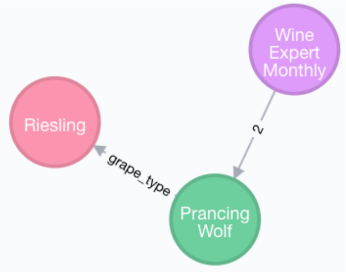
- **Nodes**
  - Main data elements
  - Connected via *relationships*
  - Have *properties* (key/value pairs)
- **Relationships**
  - Connect two *nodes*
  - Directional
  - Multiple relationships per node
  - Have *properties* (key/value pairs)
- **Properties**
  - Named values (key is a string)
  - Indexed and constrained
  - Composite indexes from multiple properties
- **Labels**
  - Group nodes into sets
  - Nodes may have multiple labels
  - Labels indexed for faster node retrieval
  - Native label indexes optimized for performance

Cypher Example

```
CREATE (w:Wine
    {name: "Prancing Wolf",
      style: "ice wine",
      vintage: 2015})
MATCH (n)
  RETURN n;
CREATE (p:Publication
    {name: "Wine Expert Monthly"})
MATCH (p:Publication
    {name: "Wine Expert Monthly"}),
    (w:Wine {name: "Prancing Wolf",
    vintage: 2015})
    CREATE (p)-[r:reported_on]->(w)
```

Cypher Example

```
MATCH (p:Publication {name: "Wine Expert Monthly"}),
    (w:Wine {name: "Prancing Wolf"})
    CREATE (p)-[r:reported_on {rating: 2}]->(w)

CREATE (g:GrapeType {name: "Riesling"})
MATCH (w:Wine {name: "Prancing Wolf"}),
  (g:GrapeType {name: "Riesling"})
  CREATE (w)-[r:grape_type]->(g)
```

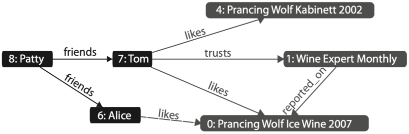


Cypher Example

```
CREATE (wr:Winery {name: "Prancing Wolf Winery"})
MATCH (w:Wine {name: "Prancing Wolf"}),
    (wr:Winery {name: "Prancing Wolf Winery"})
    CREATE (wr)-[r:produced]->(w)
CREATE (w:Wine
    {name:"Prancing Wolf", style: "Kabinett", vintage: 2002})
CREATE (w:Wine
    {name: "Prancing Wolf", style: "Spätlese", vintage: 2010})
MATCH (wr:Winery
    {name: "Prancing Wolf Winery"}),(w:Wine {name: "Prancing Wolf"})
    CREATE (wr)-[r:produced]->(w)
MATCH (w:Wine), (g:GrapeType {name: "Riesling"})
    CREATE (w)-[r:grape_type]->(g)
```

Cypher Example

- Add a social component to the wine graph
  - People preference for wine
  - Relationships with one another



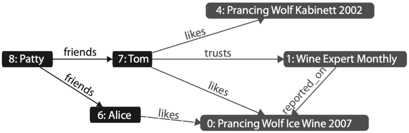

```
CREATE (p:Person {name: "Alice"})

MATCH (p:Person {name: "Alice"}),
    (w:Wine {name: "Prancing Wolf",
    style: "ice wine"})
    CREATE (p)-[r:likes]->(w)

CREATE (p:Person {name: "Patty"})

MATCH (p1:Person {name: "Patty"}),
    (p2:Person {name: "Tom"})
    CREATE (p1)-[r:friends]->(p2)
```

- The changes were made “superimposing” new relationships without changing the previous data

SCIENCE ACADEMY

Cypher Example

```
MATCH (p:Person
  {name: "Alice"})-->(n)
  RETURN n;
```




```
MATCH (p:Person
  {name: "Alice"})-->(other: Person)
  RETURN other.name;
```

```
MATCH (fof:Person)-[:friends]-(f:Person)-[:friends]-(p:Person {name
  RETURN fof.name;
```

A general query structure

```
MATCH [Nodes and relationships]
WHERE [Boolean filter statement]
RETURN [DISTINCT] [statements [AS alias]]
ORDER BY [Properties] [ASC or DESC]
SKIP [Number] LIMIT [Number]
```

Simple query

- Get all nodes of type `Program` that have the name `Hello World!`

```
MATCH (a : Program)
WHERE a.name = 'Hello World!'
RETURN a
```

Query relationships

- Get all relationships of type `Author` connecting `Programmers` and `Programs`:


```
MATCH (a:Programmer)-[r:Author]->(b:Program)

RETURN r
```

Matching nodes and relationships

- Nodes
  - (a), (), (:Ntype), (a:Ntype),
  - (a { prop:‘value’ }) ,
  - (a:Ntype { prop:‘value’ })
- Relationships
  - (a)–(b)
  - (a)–>(b), (a)<–(b),
  - (a)–>(), (a)-[r]->(b),
  - (a)-[:Rtype]->(b), (a)-[:R1|:R2]->(b),
  - (a)-[r:Rtype]->(b)
- May have more than 2 nodes
  - (a)–>(b)<–(c), (a)–>(b)–>(c)
- Path
  - p = (a)–>(b)

More options

- Relationship distance:
  - `(a)-[:Rtype*2]->(b)`: 2 hops of type Rtype
  - `(a)-[:Rtype*]->(b)`: any number of hops of type Rtype
  - `(a)-[:Rtype*2..10]->(b)`: 2-10 hops of Rtype
  - `(a)-[:Rtype*..10]->(b)`: 1-10 hops of Rtype
  - `(a)-[:Rtype*2..]->(b)`: at least 2 hops of Rtype
- Could be used also as:
  - `(a)-[r*2]->(b)` r gets a sequence of relationships
  - `(a)-[*{prop:val}]->(b)`

Operators

- Mathematical
  - +, -, *, /,%, ^ (power, not XOR)
- Comparison
  - =,<>,<,>,>=,<=, =~ (Regex), IS NULL, IS NOT NULL
- Boolean
  - AND, OR, XOR, NOT
- String
  - Concatenation through +
- Collection
  - Concatenation through +
  - IN to check if an element exists in a collection

More WHERE options

- WHERE others.name IN [‘Andres’, ‘Peter’]
- WHERE user.age IN range (18,30)
- WHERE n.name =~ 'Tob.*‘
- WHERE n.name =~ '(?i)ANDR.*‘ - (case insensitive)
- WHERE (tobias)–>()
- WHERE NOT (tobias)–>()
- WHERE has(b.name)
- WHERE b.name? = ‘Bob’ (Returns all nodes where name = ‘Bob’ plus all nodes without a name property)

SCIENCE ACADEMY

Functions

- On paths:
  - MATCH shortestPath((a)-[*]-(b))
  - MATCH allShortestPath((a)-[*]-(b))
  - Length(path) – The path length or 0 if not exists.
  - RETURN relationships(p) - Returns all relationships in a path.
- On collections:
  - RETURN a.array, filter(x IN a.array WHERE length(x)= 3) FILTER - returns the elements in a collection that comply to a predicate.
  - WHERE ANY (x IN a.array WHERE x = "one") – at least one
  - WHERE ALL (x IN nodes(p) WHERE x.age > 30) – all elements
  - WHERE SINGLE (x IN nodes(p) WHERE var.eyes = "blue") – Only one
- nodes(p) – nodes of the path p

With

- Manipulate result sequence before passing to following query parts
- Usage of WITH:
  - Limit entries passed to other MATCH clauses
  - Introduce aggregates for predicates in WHERE
  - Separate reading from updating the graph. Each query part must be read-only or write-only

Data access is programmatic

- REST API
- Through the Java APIs
  - JVM languages have bindings to the same APIs
    - JRuby, Jython, Clojure, Scala...
- Managing nodes and relationships
- Indexing
- Traversing
- Path finding
- Pattern matching

- Motivation
- Graph data models
- Storing graph data
- Querying graph data
- ***Typical graph analysis tasks***
- Executing graph analysis tasks

Queries vs Analysis Tasks

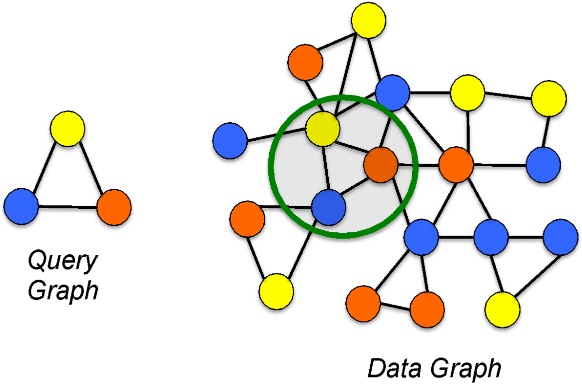
- **Queries**
  - Explore data
  - Result: small graph portion (often a node)
  - Challenges
    - Minimize explored graph portion
    - Use indexes (auxiliary data structures)
- **Analysis tasks**
  - Process entire graph
  - Challenges
    - Handle large data efficiently
    - Parallelize if data doesn't fit in memory/disk

Examples of Graph Tasks

- Subgraph pattern matching
  - Find matching instances of a small graph in a large graph
  - Patterns are usually small
- Shortest path queries
  - Find shortest path between two nodes
  - E.g., road networks
- Reachability
  - Determine if a path exists between two nodes
  - May include edge constraints
- Keyword search
  - Find smallest subgraph containing all specified keywords
- Historical queries
  - Find nodes with similar evolution to a given node
- Graph algorithms
  - Network flows
  - Spanning trees

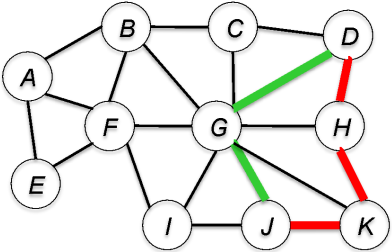
Queries: Subgraph Matching

- Given a “query” graph, find where it occurs in a given “data” graph
  - Query graph can specify restrictions on the graph structure, on values of node attributes, and so on
  - An important variation: *approximate* matching



Queries: Connection Subgraphs

- Given a data graph and nodes, find a subgraph that captures the relationship
- Define "best captures"
  - E.g., "shortest path": may not be most informative

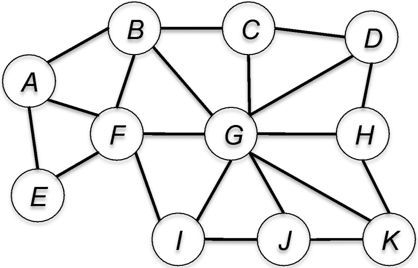


The "red" path between D and J maybe more informative than the "green" path

Graph Analysis: Centrality Measures

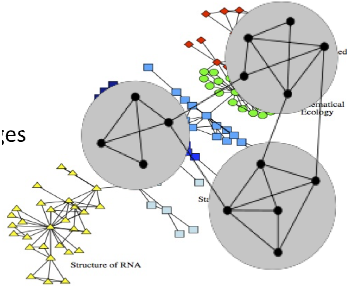
- Centrality measure: relative importance of a vertex in a graph
- Different centrality measures
  - Yield different results

- **Degree centrality of a node u**
  - Number of edges incident on u
- **Betweenness centrality of a node u**
  - Number of shortest paths between vertex pairs through u
- **Page Rank of a node u**
  - Probability a random surfer ends up at node u



Graph Analysis: Community Detection

- Goal: partition vertices into (potentially overlapping) groups based on interconnections
  - More connections within a community than across communities
  - Insights into network function; identify functional modules; improve Web services
- Techniques for community detection
  - Graph partitioning-based methods
  - Maximizing “goodness” function
  - Recursively removing high centrality edges

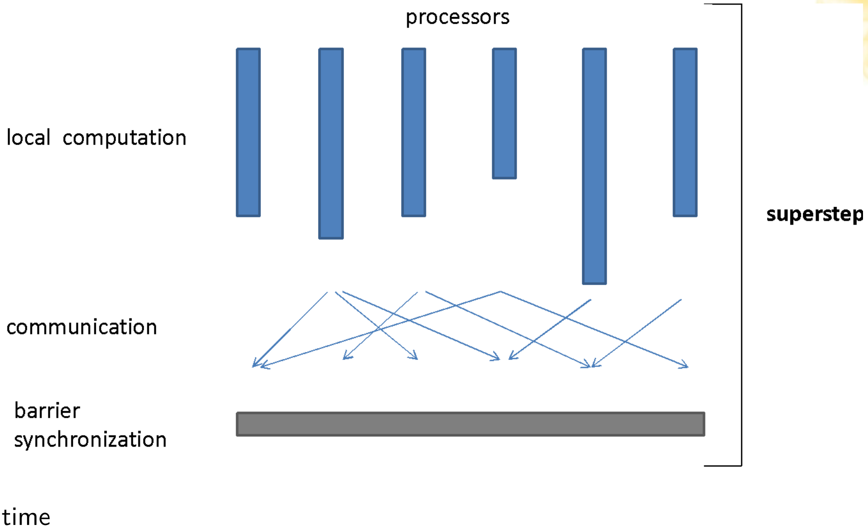


- Motivation
- Graph data models
- Storing graph data
- Querying graph data
- Typical graph analysis tasks
- ***Executing graph analysis tasks***

Bulk Synchronous Parallel (BSP)

- BSP model is a computational model for designing parallel algorithms for distributed systems
- Computation is divided into *supersteps* with three phases
  - **Local computation phase**
    - Processing units perform calculations independently and concurrently
  - **Communication phase**
    - Processing units exchange information by sending and receiving messages asynchronously
  - **Synchronization phase**
    - Aka barrier
    - Ensures all units complete computations and communication before the next superstep
    - Guarantees all messages from the previous superstep are processed
- Suitable for iterative graph algorithms
  - E.g., pageRank and Shortest Path

Bulk Synchronous Parallel (BSP)



Pregel System

- Large-scale graph processing system by Google
    - Pregel paper, 2010
- Inspired by Bulk Synchronous Parallel (BSP) model
    - Vertex-centric programming
    - Asynchronous message passing
- Fault-tolerant with checkpointing
- Scalable, distributed architecture
- Processes large graphs with billions of vertices, edges
- Handles graph mutations, updates during computation
- Not open-source, internal to Google

Apache Giraph

- Apache Giraph
  - Open-source graph processing framework, inspired by Google's Pregel
  - Implemented by Facebook, then open-sourced
  - Built on Apache Hadoop
  - Fault-tolerant with Hadoop checkpointing
  - Scalable, distributed architecture
  - Suitable for large-scale graph analytics, machine learning algorithms
  - Actively maintained, widely adopted in open-source community

Apache Spark GraphX

- Apache Spark GraphX
  - Graph processing library for Apache Spark
  - Built on Spark's RDD model
  - Supports directed and undirected graphs
  - Flexible graph computation API
  - Optimized for iterative graph computations
  - Scalable, fault-tolerant architecture
  - In-memory graph processing for improved performance
  - Suitable for large-scale graph analytics, machine learning tasks
  - Implements various graph algorithms
    - E.g., pageRank, Connected Components, Shortest Path