UMD DATA605 - Big Data Systems

Graph Data Management

Neo4J

Dr. GP Saggese
gsaggese@umd.edu

with thanks to
Amol Deshpande, R. Licher (Technion),
S. Nagarajan (U. Texas)

Overview

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

Overview

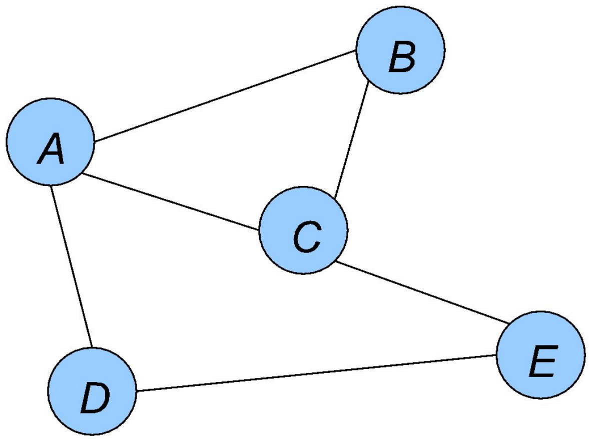
- Motivation
- Graph data models
  - E.g., RDF, Property Graph, XML
- Storing graph data
  - E.g., Neo4j
- Querying graph data
  - E.g., Cypher, SPARQL, Gremlin
- Typical graph analysis tasks
  - E.g., PageRank, clustering
- Executing graph analysis tasks
  - E.g., Google Pregel, Apache Giraph, Spark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

Overview

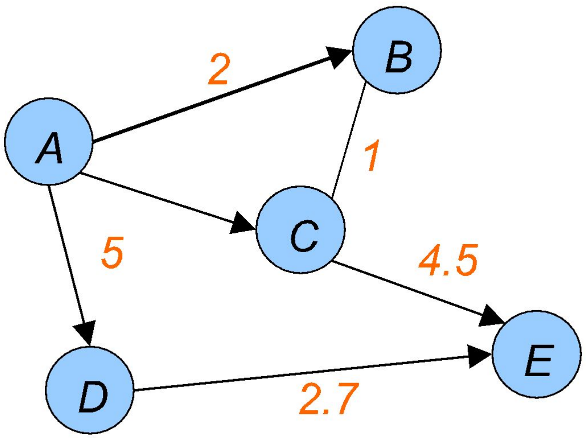
- **Motivation**
- Graph data models
- Storing graph data
- Querying graph data
- Typical graph analysis tasks
- Executing graph analysis tasks

Graphs: Background

- A *graph (or network)* captures a set of entities and interconnections between pairs of them
  - Entities / objects represented by *vertices or nodes*
  - Interconnections between pairs of vertices called *edges* (or *links, arcs, relationships)*
- Graph theory and algorithms widely studied in Computer Science
  - Not as much work on managing graph-structured data



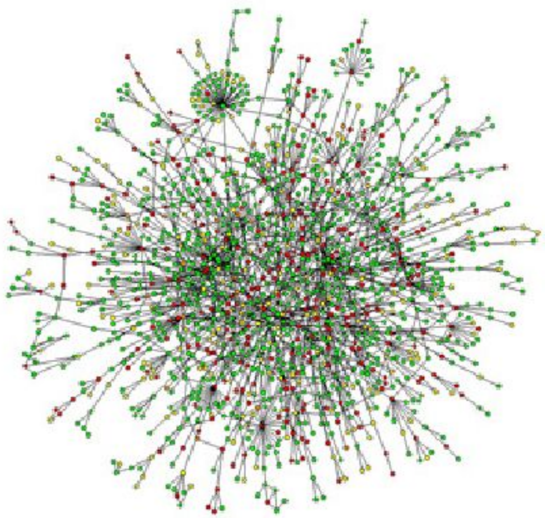
An undirected, unweighted graph



A directed, edge-weighted graph

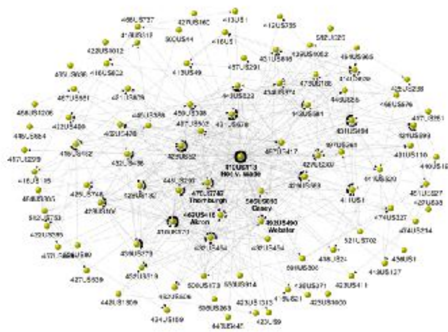
Graph Data Structures: Motivation

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

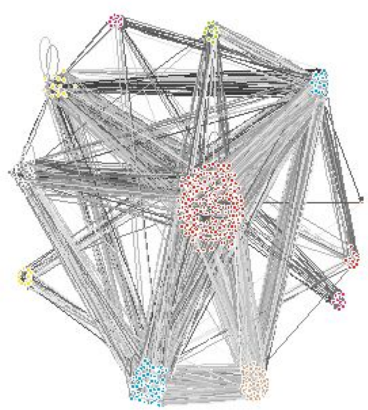


A protein-protein interaction network

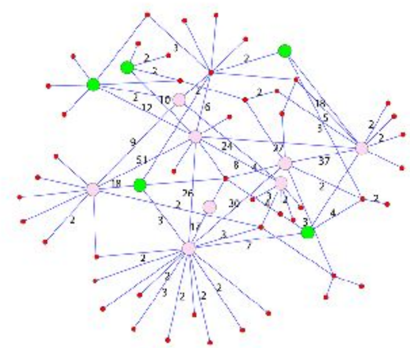
Social networks



Supreme court citation network

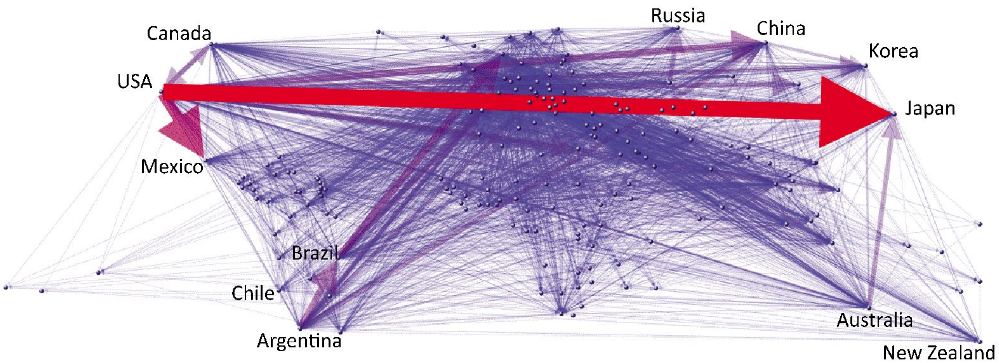


Financial transaction networks

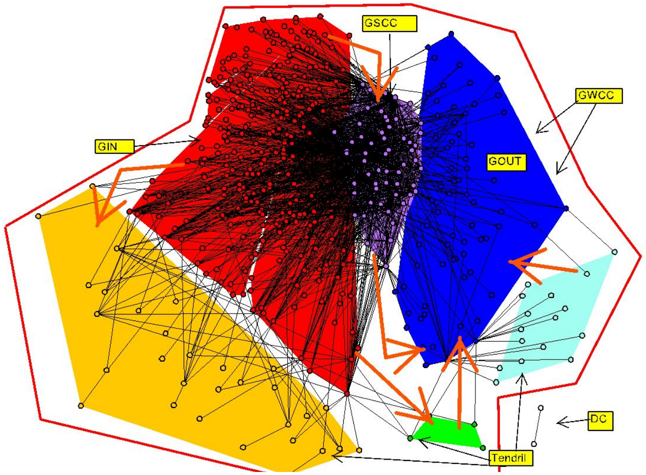


Stock Trading Networks

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 the data in 1960s than today
- **What has changed in recent years**
  - Large data volumes and easier availability
  - Reasoning about the graph structure can provide useful and actionable insights
    - Lose too much information if graph structure ignored
  - Not easy to query using traditional tools (e.g., relational DBs)
    - Need specialized tools (e.g., Neo4j)
  - Hard to efficiently process graph-structured queries using existing tools
    - Dedicated solutions: Google Pregel / Apache Giraph, Spark GraphX
    - Problems getting worse with increasingly large graphs seen in practice

Overview

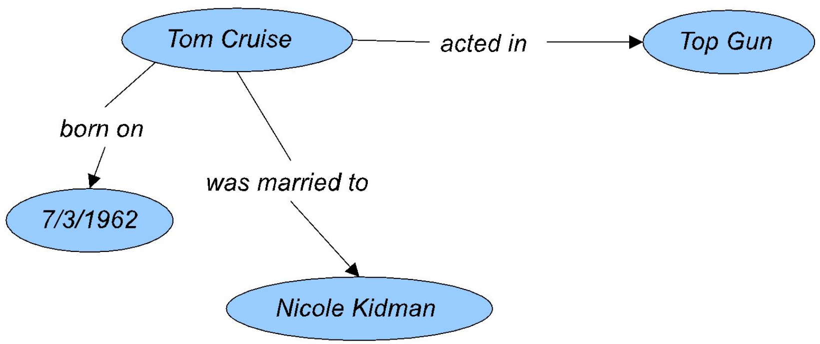
- Motivation
- **Graph data models**
- Storing graph data
- Querying graph data
- Typical graph analysis tasks
- Executing graph analysis tasks

Knowledge Graphs

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

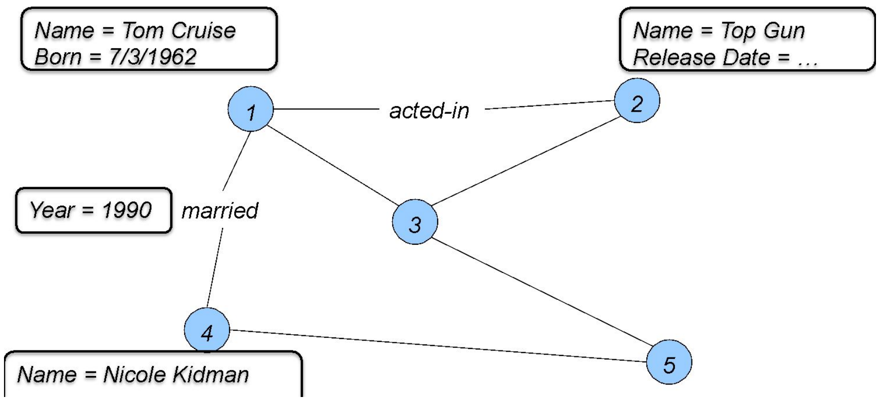
Graph Data Models: RDF

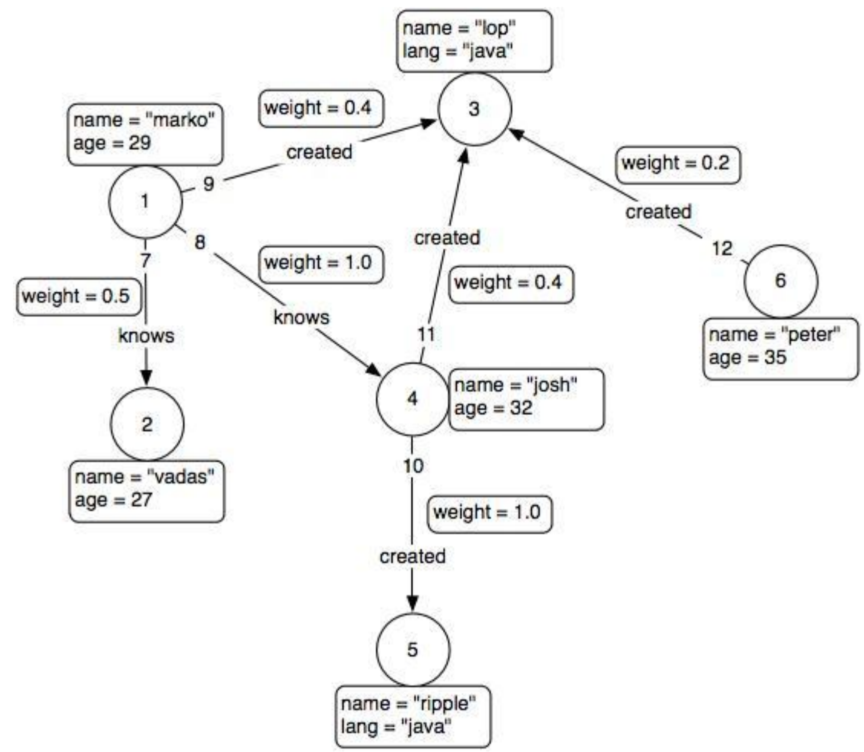
- [Resource Description Framework]()
- RDF uses triples subject-predicate-object
  - Each triple connects a "subject" and an "object" through a "predicate"
  - E.g., "TomCruise-acted-TopGun"
- **Used to represent *knowledge bases***
  - Typically queried through SPARQL
- **Pros**
  - Standardization
    - Standard W3C to model data
    - Subject and object can be URI (Uniform Resource Identifier) in semantic web
  - Interoperability
    - Can merge RDF data store
  - Extensibility
    - Can add new nodes and relationships
    - Support ontologies



Graph Data Models: Property Graph

- A directed graph where each node and each edge may be associated with a set of *properties* (*key-values*)
- Query languages
  - Cypher (e.g., Neo4j)
  - Gremlin (e.g., Apache TinkerPop)
- Lack universal standard
- Similar expressive power to RDFs but less “schema” so more difficult to interoperate
- Used by many open-source graph data management tools

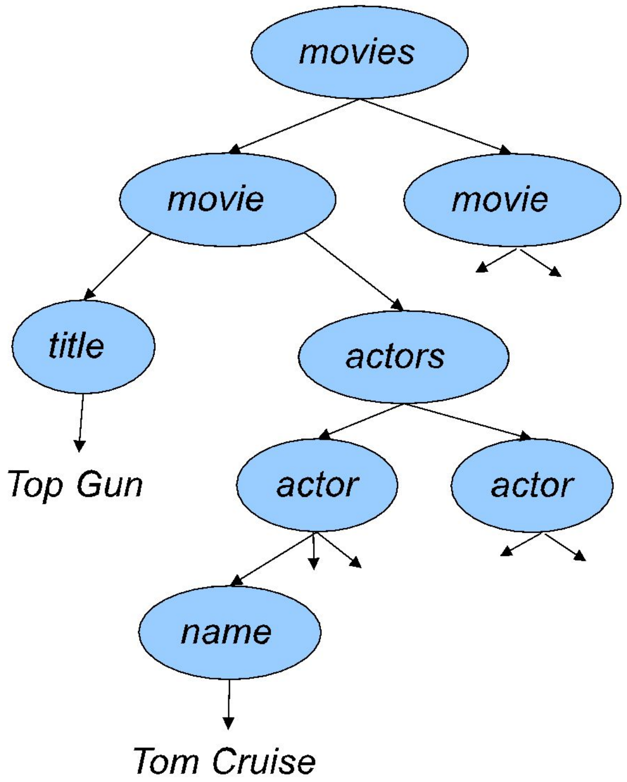




Graph Data Models: XML

- Commonly used data model for representing data without rigid structure
- It is a directed labeled tree
- Popular data exchange format for non-tabular data

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



Overview

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

Storing Graph Data

1. **File systems**
   + Very simple
   - No support for transactions, ACID
   - Minimal functionality (e.g., must build the analysis/querying on top)
2. **Relational database**
   + Mature technology
   + All the good stuff (SQL, transactions, ACID, toolchains)
   - Minimal functionality
3. **NoSQL key-value stores**
   + Can handle very large datasets efficiently in a distributed fashion
   - Minimal functionality
4. **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 in recent years
  - E.g., Neo4j, Titan, OrientDB, AllegroGraph
- A few key distinctions from relational databases
  - Built to manage and query graph-structured data
  - Store the graph structure explicitly using data structures with pointers
    - Avoid the need for joins, making graph traversals easier
    - More natural to write *queries* and *graph algorithms* (reachability or shortest paths)
  - Support graph query languages like SPARQL, Cypher, Gremlin
  - Fairly rudimentary declarative interfaces
  - Most applications need to be written using programmatic interfaces
  - Expose a programmatic API to write arbitrary graph algorithms

Overview

- 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 annotated with key-value properties
  - Declarative
  - Subgraph pattern matching
  - Can't easily handle queries like reachability
  - Native query language for Neo4j
- Gremlin
  - Works with both RDF and Property Graphs
  - Imperative
  - Allow to describe graph traversal
- SPARQL
  - Similar to Cypher
  - Query language for RDF data
  - Standardized by W3C

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

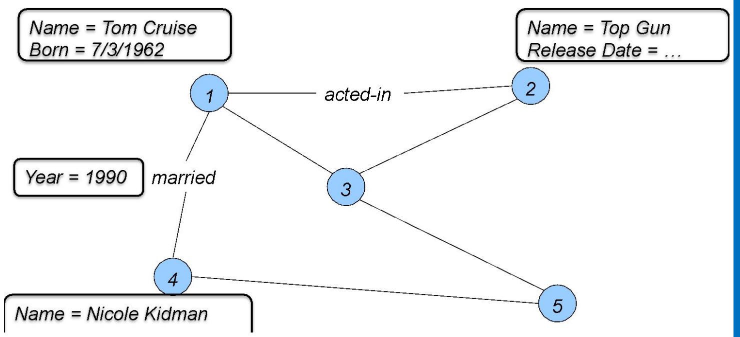
```
// calculate basic collaborative filtering for vertex 1
m = [:]
g.v(1).out('likes').in('likes').out('likes').groupCount(m)
m.sort{-it.value}

// 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: <http://xmlns.com/foaf/0.1/>
SELECT ?name
       ?email
WHERE
  {
    ?person  a          foaf:Person .
    ?person  foaf:name  ?name .
    ?person  foaf:mbox  ?email .
  }
```

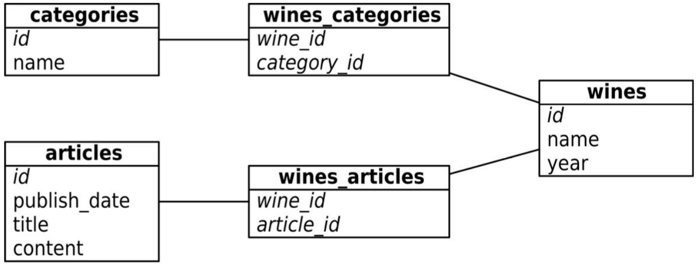
Neo4j

- Graph DB storing data as Property Graph
  - Nodes, edges hold data as key-value pairs (like non-relational DBs)
- Focus is
  - On relationships between values
  - Instead of commonalities among sets of values (e.g., tables of rows for RDBMs and collection of documents)
- Two querying languages
  - Cypher, Gremlin
- GUI or REST API
- Full ACID-compliant transactions (atomicity, consistency, isolation, durability)
- High-availability clustering
- Incremental backups
- Can run in small application or run on large clusters of servers



Graph DB: Example

- **Specs**
  - Create a wine suggestion engine
  - Wines are categorized by different
    - Varieties (e.g., Chardonnay, Pinot Noir)
    - Regions (e.g., Bordeaux, Napa, Tuscany)
    - Vintage (year in which the grapes were harvested)
  - Keep track of articles describing wines by various authors
  - Users can track their 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)
- **Problem with relational approach**
  - There isn't much of a schema
  - Lots of incomplete data
  - An old saying in relational DB world: "*On a long enough timeline all fields become optional*"
- **Graph DB approach**
  - Provide values and structure only where necessary



Labeled Property Graphs in Neo4j

- **Nodes**
  - Main data elements
  - Connected to other nodes via *relationships*
  - Can have one or more *properties* (stored as key/value pairs)
- **Relationships**
  - Connect two *nodes*
  - Are directional
  - *Nodes* can have multiple relationships
  - Can have one or more *properties* (stored as key/value pairs)
- **Properties**
  - Named values where the name (or key) is a string
  - Can be indexed and constrained
  - Composite indexes can be created from multiple properties
- **Labels**
  - Used to group nodes into sets
  - A node may have multiple labels
  - Labels indexed to accelerate finding nodes in the graph
  - Native label indexes optimized for performance

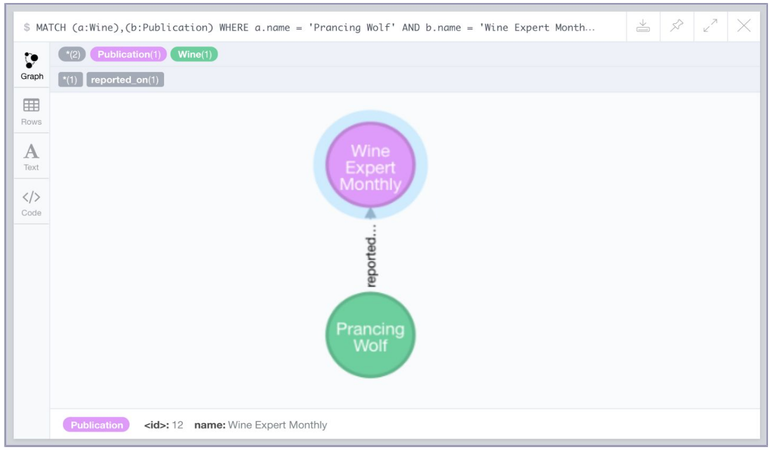
Cypher Example

```
# Create a wine node with attributes.
$ CREATE (w:Wine
    {name:"Prancing Wolf",
       style: "ice wine",
    vintage: 2015})

# Return the entire graph.
$ MATCH (n)
  RETURN n;

# Create a publication node.
$ CREATE (p:Publication
    {name: "Wine Expert Monthly"})

# Create a relation “reported_on”.
$ MATCH (p:Publication
    {name: "Wine Expert Monthly"}),
    (w:Wine {name: "Prancing Wolf",
     vintage: 2015})
    CREATE (p)-[r:reported_on]->(w)
```

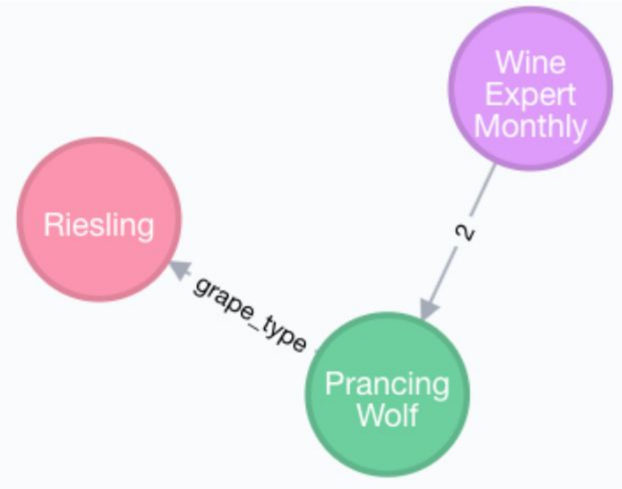


Cypher Example

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

# Add a “grape_type” relationship.
$ CREATE (g:GrapeType {name: "Riesling"})

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



Cypher Example

```
# Add winery.
$ CREATE (wr:Winery {name: "Prancing Wolf Winery"})

# Add “produced” relationship.
$ 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"}),(w:Wine {name: "Prancing Wolf"})
    CREATE (wr)-[r:produced]->(w)

# Add “grape type” relationship.
$ MATCH (w:Wine), (g:GrapeType {name: "Riesling"})
    CREATE (w)-[r:grape_type]->(g)
```

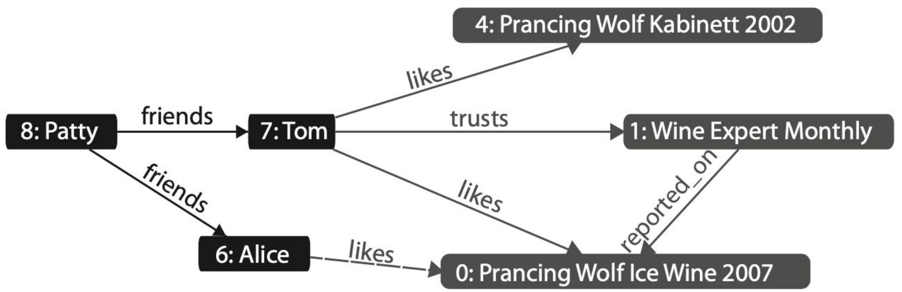
Cypher Example

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

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

# Patty and Tom are friends.
$ 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

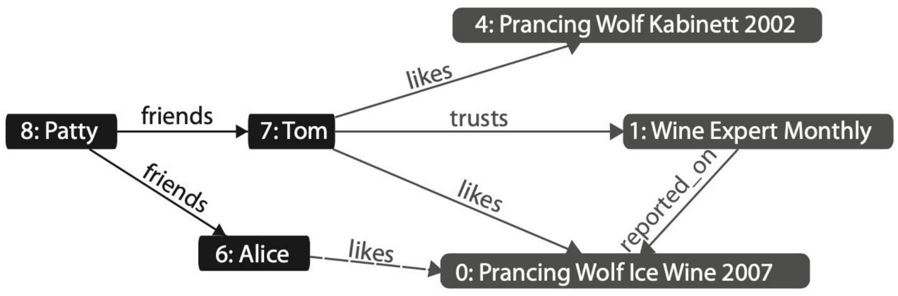


Cypher Example

```
# See all nodes associated with Alice.
$ MATCH (p:Person
  {name: "Alice"})-->(n)
  RETURN n;

# Find all of the people that Alice is
friends with, returning only the name
property of those nodes
$ MATCH (p:Person
  {name: "Alice"})-->(other: Person)
  RETURN other.name;

# Find friends of friends of Alice.
$ MATCH
(fof:Person)-[:friends]-(f:Person)-[:f
riends]-(p:Person {name: "Patty"})
  RETURN fof.name;
```



A general query structure

```
MATCH [Nodes and relationships]
WHERE [Boolean filter statement]
RETURN [DISTINCT] [statements [AS alias]]
ORDER BY [Properties] [ASC\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)

Functions

- On paths:
  - MATCH shortestPath((a)-[*]-(b))
  - MATCH allShorestPath((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 the result sequence before it is passed on to the following query parts.
- Usage of WITH:
  - Limit the number of entries that are then passed on to other MATCH clauses.
  - Introduce aggregates which can then be used in predicates in WHERE.
  - Separate reading from updating of the graph. Every part of a query must be either 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

Overview

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

Queries vs Analysis Tasks

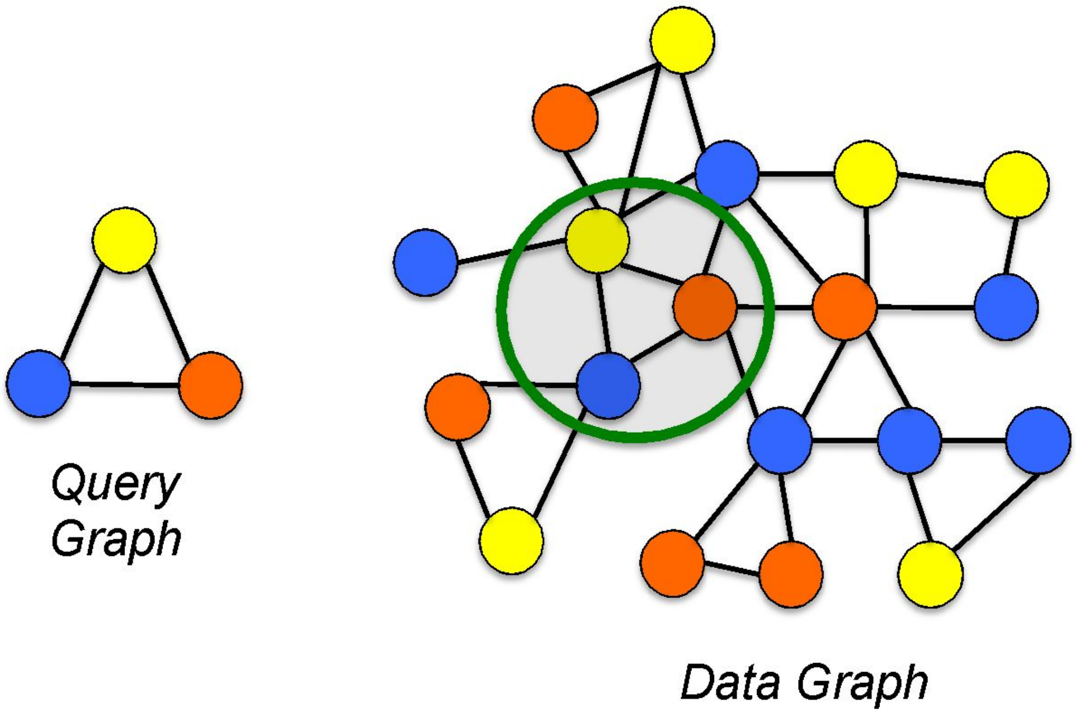
- **Queries**
  - Focused exploration of the data
  - Result is typically a small portion of the graph (often just a node)
  - Challenges
    - Minimize the portion of the graph that is explored
    - Use of indexes (auxiliary data structures)
- **Analysis tasks**
  - Typically require processing the entire graph
  - Challenges
    - How to handle the large volume of data efficiently
    - How to parallelize if data doesn't fit in memory / disk

Examples of Graph Queries / Tasks

- Subgraph pattern matching
  - Find matching instances of a given small graph in a large graph
  - Although technically NP-hard, usually the patterns are small
- Shortest path queries
  - Find the shortest path between two given nodes
  - E.g., in road networks
- Reachability
  - Given two nodes, is there an undirected or directed path between them?
  - Sometimes with constraints on the types of edges that can be used
- Keyword search
  - Find the smallest subgraph that contains all the specified keywords
- Historical queries
  - Given a node, find other nodes that evolved most similarly in the past
- Graph algorithms
  - Network flows
  - Spanning trees

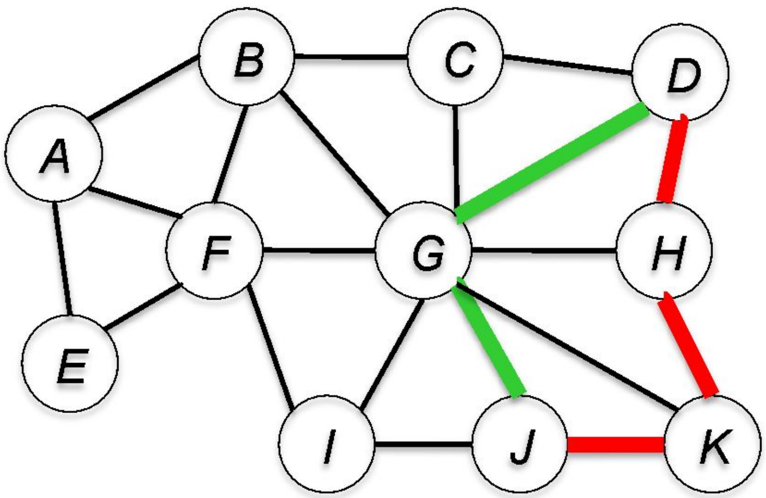
Queries: Subgraph Matching

- 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 two (or more) nodes in it, find a small subgraph that best captures the relationship between the nodes
- How to define "best captures"?
  - E.g., "shortest path": but that may not be most informative

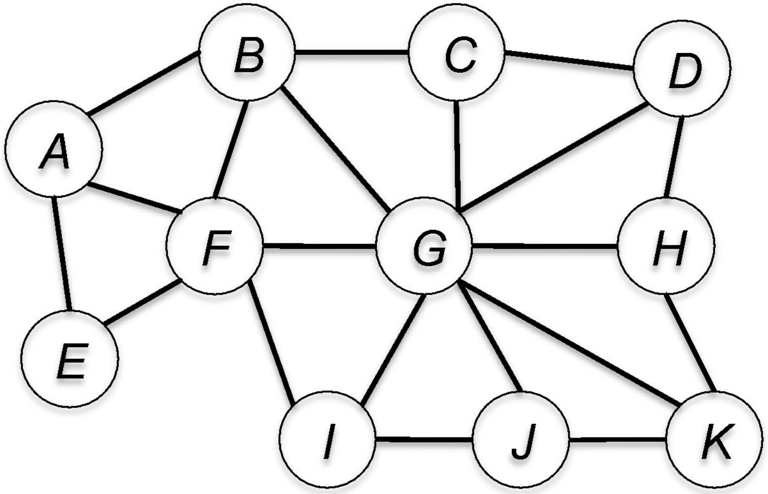


The "red" path between D and J maybe more informative than the "green" path

Graph Analysis: Centrality Measures

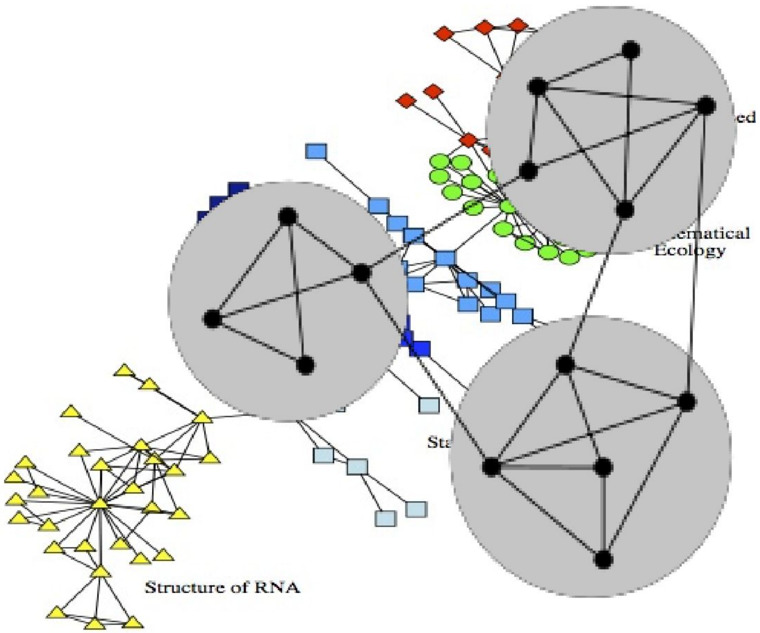
- Centrality measure: a measure of the relative importance of a vertex within a graph
- Many different definition of centrality measures
  - Can give fairly different results

- **Degree centrality of a node *u***
  - *Number of edges incident on **u***
- **Betweenness centrality of a node *u***
  - *Number of shortest paths between pairs of vertices that go through **u***
- **Pagerank of a node *u*:**
  - *Probability that a random surfer (who is following links randomly) ends up at node **u***



Graph Analysis: Community Detection

- Goal: partitioning the vertices into (potentially overlapping) groups based on the interconnections between them
  - Basic intuition: More connections within a community than across communities
  - Provide insights into how networks function; identify functional modules; improve performance of Web services; etc.
- Numerous techniques proposed for community detection over the years
  - Graph partitioning-based methods
  - Maximizing some “goodness” function
  - Recursively removing high centrality edges
  - And so on …



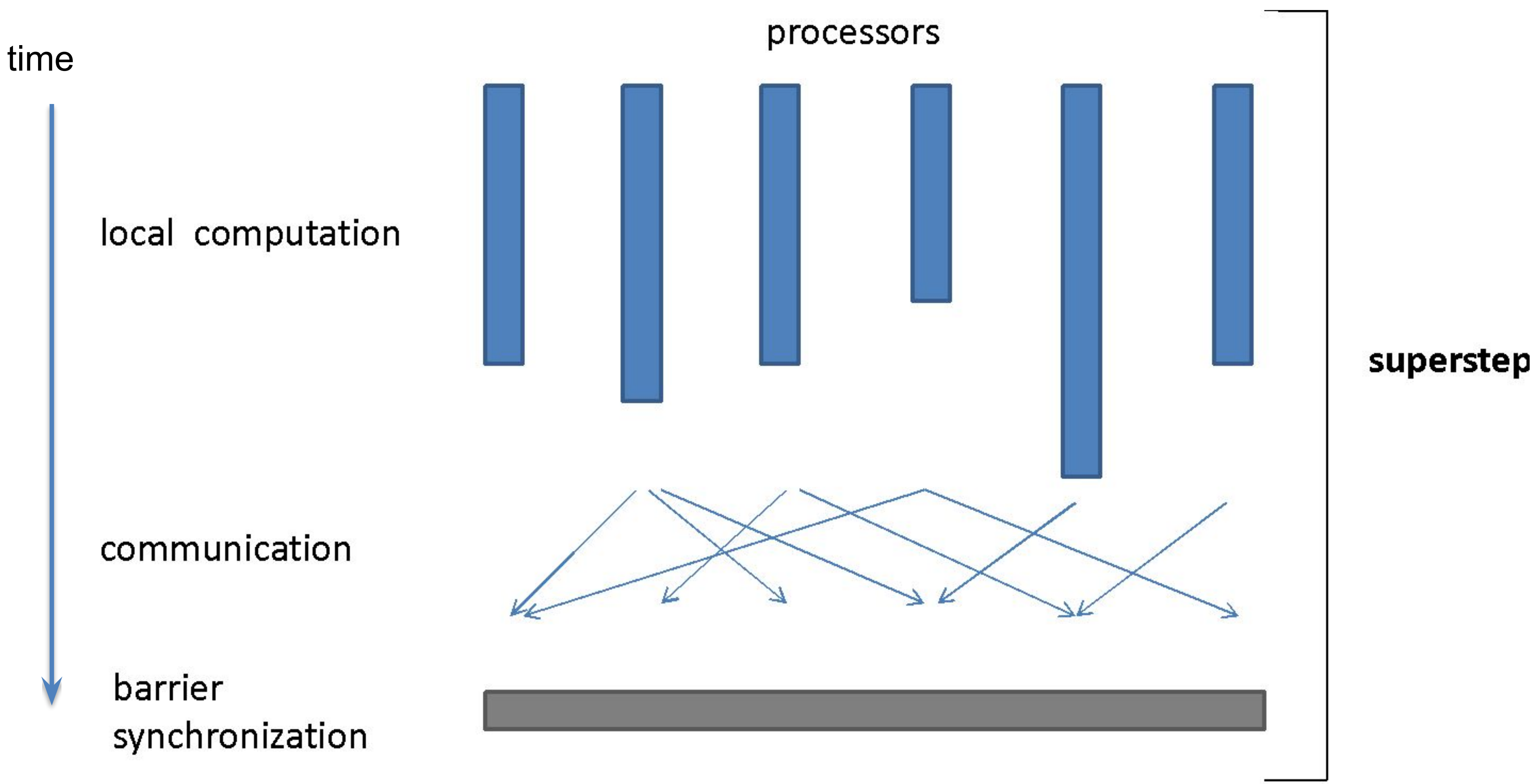
Overview

- 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 used to design parallel algorithms for distributed systems
- Computation is divided into a series of *supersteps*, each consisting of three main phases
  - **Local computation phase**
    - Each processing unit performs calculations independently and concurrently, without any interaction
  - **Communication phase**
    - The processing units exchange information with each other by sending and receiving messages
    - These messages can be exchanged asynchronously without waiting for a response
  - **Synchronization phase**
    - Aka barrier
    - Ensures that all processing units have completed their local computations and communication before proceeding to the next superstep
    - This guarantees that all messages from the previous superstep have been received and processed
- Suitable for iterative graph algorithms
  - E.g., PageRank and Shortest Path

Bulk Synchronous Parallel (BSP)



Pregel System

- Large-scale graph processing system developed by Google
  - Pregel paper, 2010
- Inspired by the Bulk Synchronous Parallel (BSP) model
  - Vertex-centric programming model
  - Asynchronous message passing between vertices
- Fault-tolerant using checkpointing mechanism
- Scalable and distributed architecture
- Designed for processing large graphs with billions of vertices and edges
- Handles graph mutations and updates during computation
- Not open-source, used internally at Google

Apache Giraph

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

Apache Spark GraphX

- Apache Spark GraphX
- Graph processing library for Apache Spark
- Built on top of Spark's RDD (Resilient Distributed Dataset) model
- Supports both directed and undirected graphs
- Provides a flexible graph computation API
- Optimized for iterative graph computations
- Scalable and fault-tolerant architecture
- Supports in-memory graph processing for improved performance
- Suitable for large-scale graph analytics and machine learning tasks
- Implements various graph algorithms
  - E.g., PageRank, Connected Components, and Shortest Path