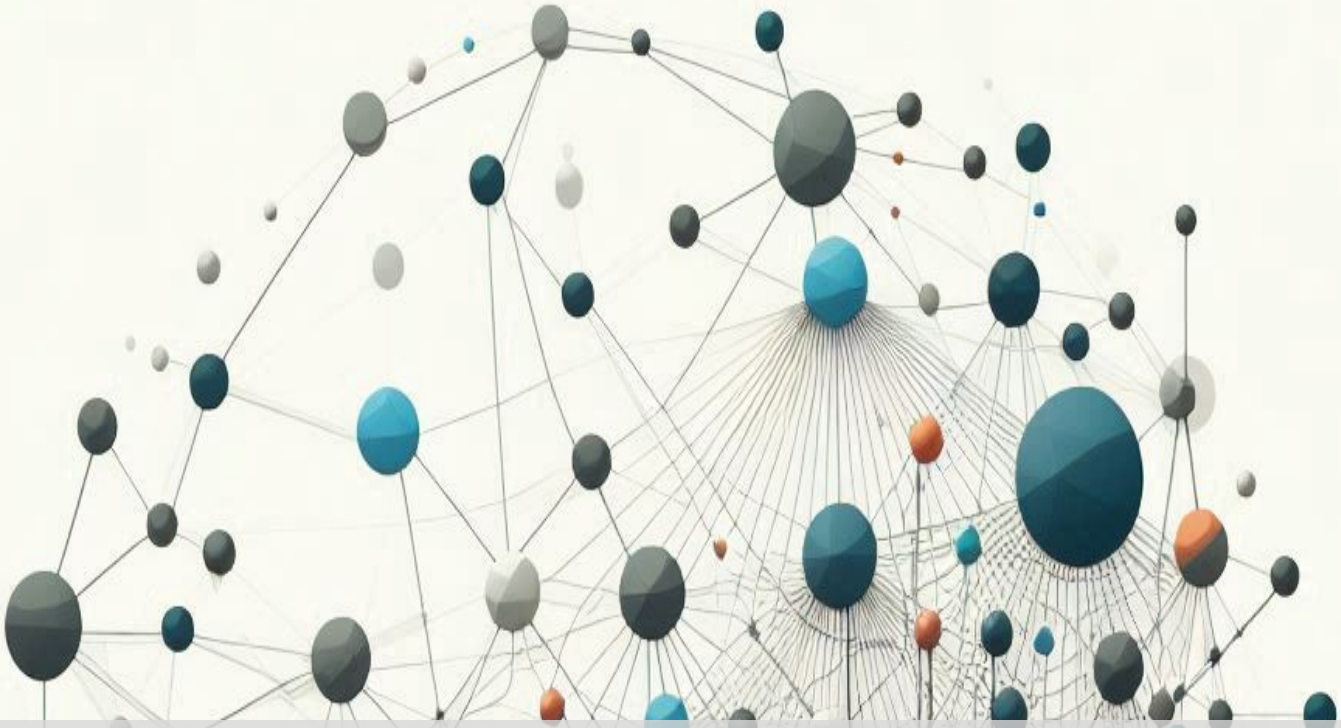
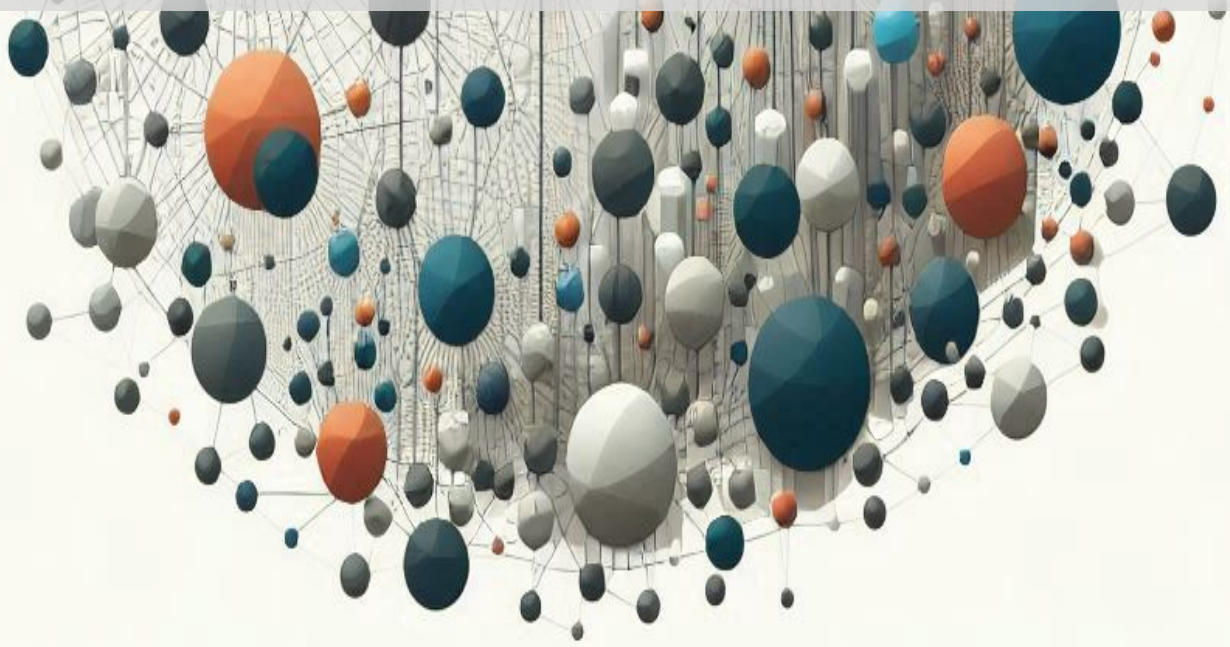


Graph Analytics

Luciano Imbimbo



PUBLISHED BY RAFAEL BRITO GOMES

BOOK-WEBSITE.COM

Contents

Graph Theory

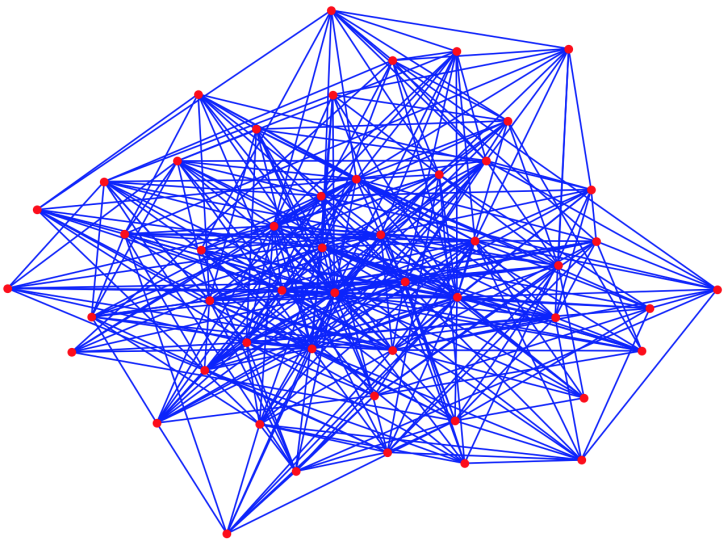
1. Introduction to graph theory

1.1 What is a graph?

[1] A graph is a collection of points (vertices) and lines connecting them (edges), used to model various real-life scenarios.

- Computer network
- Protein-protein interaction network
- Neuron network
- Underground Networks
- Disease Pathways

Graphs can be useful to see and analyze the interconnections among entities to evaluate the impact on the entire graph of a specific node, to identify nodes that might cause vulnerabilities, and to identify communities (subsets of nodes).

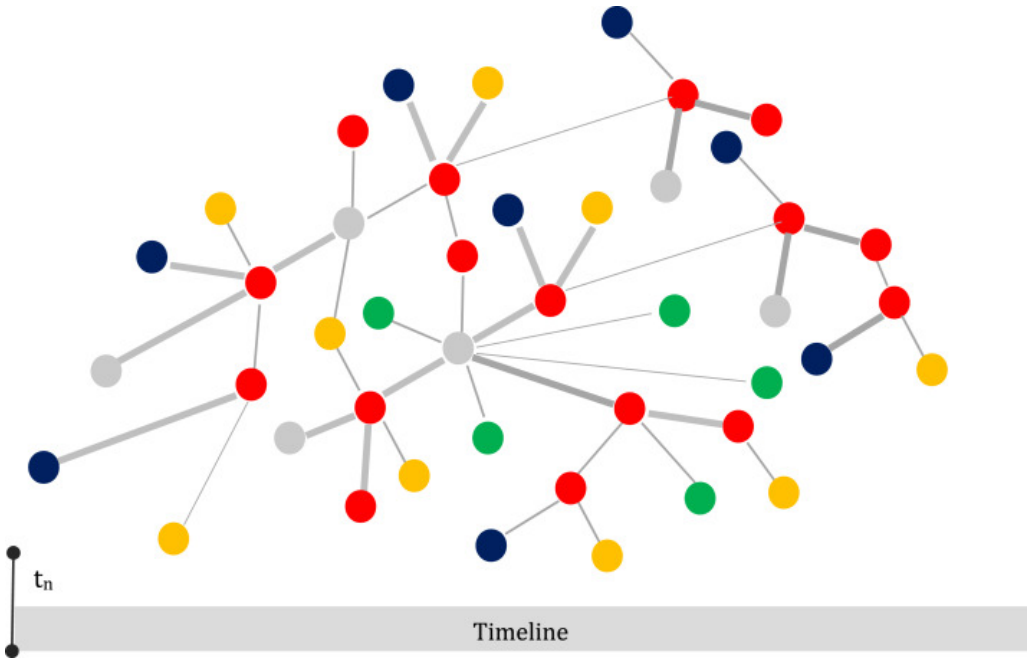


1.2 What is Graph Analytics?

Graph analytics provides algorithms that help data scientists and data-driven analysts answer questions or make predictions using graph data. The most common graph analytics use cases are the following:

- Community detection and influencer analysis
- Clustering for product recommendation
- Fraud and money laundering
- Path analysis and reachability
- Link prediction
- Pattern matching
- Medical research (understand the interactions between drugs or to track the spread of disease such as COVID-19)

Big data analytics is a method that focuses more on the relationships within the data, as compared to traditional analytics based on relational databases. With the increasing complexity of data, the use of big data is also growing and leveraging graph analytics is a key to business success in the future. In contrast, traditional analytics based on relational databases are more inclined towards individual data points and are not designed to accurately represent relationships (joins to explore them). To effectively manage the relationships between nodes, graphs have been defined, which enable direct queries. (direct queries). Graphs and Graph analytics are also widely used in machine learning as a powerful tool that can enable intuition and power a lot of useful features. Graph-based machine learning is becoming more common over time, transcending numerous traditional techniques.

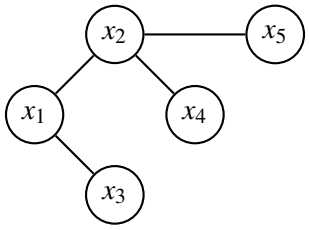


1.3 Graph Theory

The historical origin of graph theory dates back to 1736 when the mathematician Euler solved the problem of the bridges of Konigsberg. Konigsberg, a city divided by the river Pregel into four zones - two main areas (A and B) and two islands (C and D), was connected by seven bridges. The challenge was to find a way to cross all seven bridges without crossing any of them twice. Euler solved this problem, and in doing so, invented graph theory.

Figure 1.1: The bridges of Konigsberg

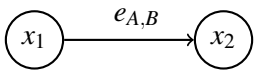
Definition 1.3.1 — Graph. A graph is a pair G = (V, E), where V is a collection of vertices $V = \{V_i, i = 1, \ldots, n\}$ and E is a collection of edges over V, $E_{ij} = \{(V_i, V_j), V_i \in V, V_j \in V\}$



Definition 1.3.2 — Vertices. A vertex with an associated value is called a **labeled** vertex, while a vertex with no associated value is called **unlabelled**.

Corollary 1.3.1 — Order of a graph. The number $|V|$ of vertices s contained in a graph is called the order of the graph

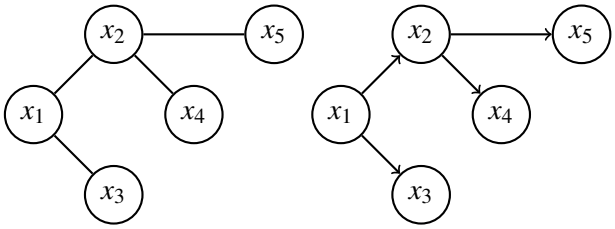
Definition 1.3.3 — Edges. An edge can connect any two vertices in a graph. The two vertices connected by an edge are called endpoints of that edge. By its definition, if an edge exists, then it has two endpoint. We call an edge going towards a vertex an **incoming edge**, while we call an edge originating from a vertex an **outgoing edge**



Corollary 1.3.2 — Size of a graph. The number of edges $|E|$ in a graph is a special parameter of that graph, called the size of the graph

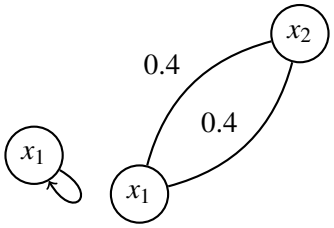
Graphs can be directed or undirected, depending on whether a direction of traversal is defined on the edge:

- In **directed graphs**, an edge E_{ij} can be traversed from V_i to V_j but not in the opposite direction; V_i is called the tail, or start node, and V_j is called the head, or end node
- In **undirected graphs** edge traversals in both directions are valid.



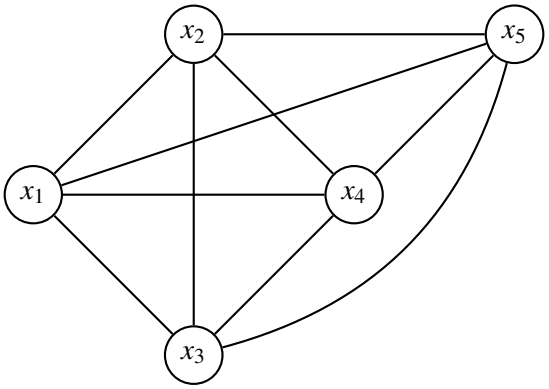
Undirected graph Directed graph

In graph can be also other type of edges like loops and multiple edges. There are many use cases when you want to permit multiple relationships between a pair of nodes; in that case, you would be dealing with a multigraph.



Corollary 1.3.3 — Adjacency or Neighbor. Two vertices x and y of G are defined as adjacent, or neighbors, if x,y is an edge of G. The edge E_{ij} connecting them is said to be incident on the two vertices V_i and V_j. Otherwise, two distinct edges e and f are adjacent if they have a vertex in common.

Definition 1.3.4 — Complete graphs. If all the vertices of G are pairwise adjacent, G is complete. In other words, a complete graph is a graph in which each vertex is connected to all the other vertices.



Definition 1.3.5 — Degree of a vertex. The degree of a vertex is defined as the total number of edges incident to that vertex, which is also equal to the number of neighbors of that vertex.

Corollary 1.3.4 — Indegree & Outdegree. In a directed graph, the degree of a vertex V_i is split into the in-degree of the vertex, defined as the number of edges for which V_i is their end node (the head of the arrow) and the out-degree of the vertex, which is the number of edges for

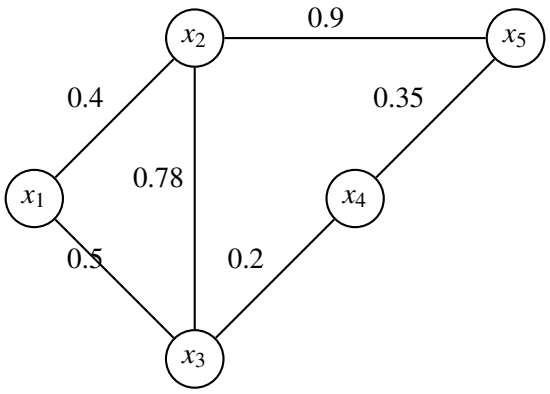
which V_i is their start node (the tail of the arrow)

Corollary 1.3.5 — Average degree of a graph. The average degree of a graph is computed as follows,

$$a = \frac{1}{N} \sum_{i=1,\ldots,N} degree(V_i) \tag{1.1}$$

1.4 Type of graphs

Definition 1.4.1 — Weighted graphs. Weighted graphs are graphs where the strength or the cost of traversing the relationships is stored as an attribute (usually a number).



Definition 1.4.2 — Monopartite graph. A monopartite graph describes a graph that consists of a single type or class of nodes.

Definition 1.4.3 — Multipartite graph. If there are multiple types of nodes present, you can describe the graph as multipartite. A graph that has exactly two types of nodes can also be defined as a **bipartite**.

1.5 Graph Paths

Definition 1.5.1 — Path. A sequence of vertices with the property that each consecutive pair in the sequence is connected by an edge is called a path. We can call paths that relate to sequences of directed edges **directed paths**.

Corollary 1.5.1 — Simple path. A path with no repeating vertices is called a simple path.

$$Simplepath = \{2,4,5\} \tag{1.2}$$

Corollary 1.5.2 — Cycle. A cycle is a path in which the first and the last vertex coincide.

$$Cycle = \{1,3,4,1\} \tag{1.3}$$

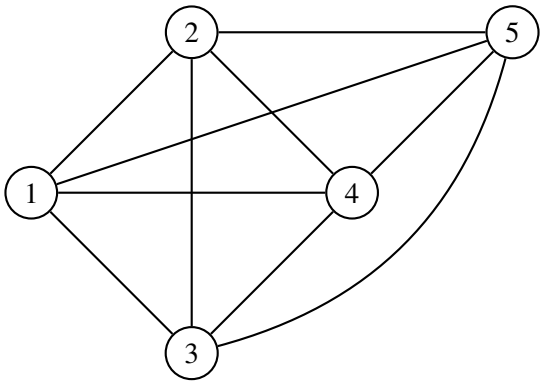
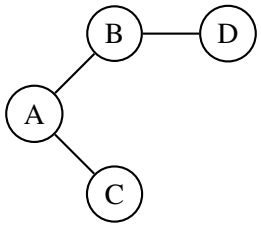


Figure 1.2: Path = $\{1,3,5\}$

2. Graph Storage

2.1 How to represent a Graph on a computer

How we can represent the structure of a graph (vertices and edges) and support several operations on that graph(add vertex, add edges, and find the nearest neighbors)? If we consider a simple graph like the figure below we have different options to represent it:



- **Edge List**: (A,B), (A,C), (B,D).
  To add a new vertex we add a new element at the end of the list. The main disadvantage of this representation is that we lost information about isolated nodes.
- **Adjacency Matrix**: matrix NxN where N is the number of vertex and entry is 1 if there is an edge and 0 if there is not

$$\begin{bmatrix} & A & B & C & D \\ A & 0 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 \\ C & 1 & 0 & 0 & 0 \\ D & 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.1}$$

In the case of an undirected graph where edges go both ways, the matrix will be symmetrical. If the graph has weights assigned to its edges, simply replace the 0/1 values in the matrix with the appropriate weight values.
To create a new connection between nodes, I can easily add a 1 to the appropriate cell in the

adjacency matrix. Instead to get neighbors of a node have to simply analyze the row and column of that node.

- **Adjacency List**: for each vertex, a list of adjacent vertices that allows to keep an order in the list.

Comparing these representations we discover that for many problems is useful to use an adjacency list because the average cost of operations is lower than the average of the other structures as shown in the figure below.

Op.	Is Edge?	List Edge	List Nbrs.
Adj. Matrix	$\Theta(1)$	$\Theta(\|V\|^2)$	$\Theta(\|V\|)$
Edge List	$\Theta(\|E\|)$	$\Theta(\|E\|)$	$\Theta(\|E\|)$
Adj. List	$\Theta(\deg)$	$\Theta(\|E\|)$	$\Theta(\deg)$

2.2 Graph Density

Another important metrics to define is the **density** that tells us how "full" a graph is, in terms of the number of edges that it possesses, and in relation to the number of its vertices.

Definition 2.2.1 — Density. D the density of a graph G(V, E) has the form of D(V, E).
For undirected graph D is:

$$D_U(V,E) = \frac{|E|}{Max_U(V)} = \frac{|E|}{\frac{|V|*(|V|-1)}{2}} = \frac{2*|E|}{|V|*(|V|-1)} \tag{2.2}$$

Instead for directed graphs, we can calculate their density as half that of the corresponding undirected graph:

$$D_D(V,E) = \frac{|E|}{Max_D(V)} = \frac{|E|}{|V|*(|V|-1)} = \frac{1}{2}*D_U(V,E) \tag{2.3}$$

Corollary 2.2.1 — Maximum number of edges. We can define the maximum number of edges in relation to the order |V| of an undirected graph, as:

$$Max_U(V) = \binom{|V|}{2} = \frac{|V|*(|V|-1)}{2} \tag{2.4}$$

We can extend this formula to directed graph.

$$Max_D(V) = 2*\binom{|V|}{2} = |V|*(|V|-1) = 2*Max_U(V) \tag{2.5}$$

Define the density can classify graphs in the following way:

- A **sparse graph** is a graph where $0 \leq D < \frac{1}{2}$. (Empty →Sparse)
- A **dense graph** is a graph where $\frac{1}{2} \leq D < 1$. (Complete →Dense)
- The graph for which $D = \frac{1}{2}$ are considered neither.

The density of a graph of order $|V| = 1$ is undefined, both for algebraic reasons and, intuitively, because it can either be seen as perfectly sparse or perfectly dense. Graphs tend to be very large data structures, it is needed to store the graph in the format that is more efficient, in relation to its density. As a general rule, the density of the graph dictates the preferable method for storage:

- If the graph is sparse, we should store it as a list of edges
- If the graph is dense, we should store it as an adjacency matrix

3. Graph Traversal

3.1 Walk, Trail and Path

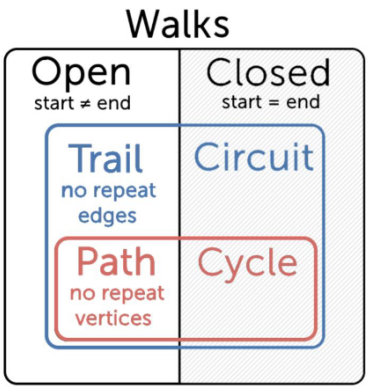
Definition 3.1.1 — Walk. A walk is the most general definition of how we can move around a graph. To start a walk simply move along vertices following edges. A walk is represent by a a sequence of vertices.

If a walk starts and ends at the same vertex then it is **closed**, and if it starts and ends at different vertices then it is **open**.

Corollary 3.1.1 — Trail. If a walk does not use the same edge more than once, then it is a trail. It is an open trail if it's open and a closed trail or **circuit** if it is closed.

Definition 3.1.2 — Path. If a walk does not use the same vertex more than once (other than at the start and end) we call it a path.

Corollary 3.1.2 — Cycle. When a path begins and ends at the same vertex it is a closed path, which is usually called a cycle.



Traversable is the word to describe graphs where you can start at a vertex and draw in all the edges without taking your pen off the page or retracing any edge. The two types of traversable graphs are the following:

- If a graph has a closed trail (it starts and finishes at the same vertex) that uses every edge, it is called **Eulerian** (Eulerian trail or Eulerian circuit).
- If a graph has an open trail (it starts and finishes at different vertices) that uses every edge, it is described as **semi-Eulerian** (semi-Eulerian trail).

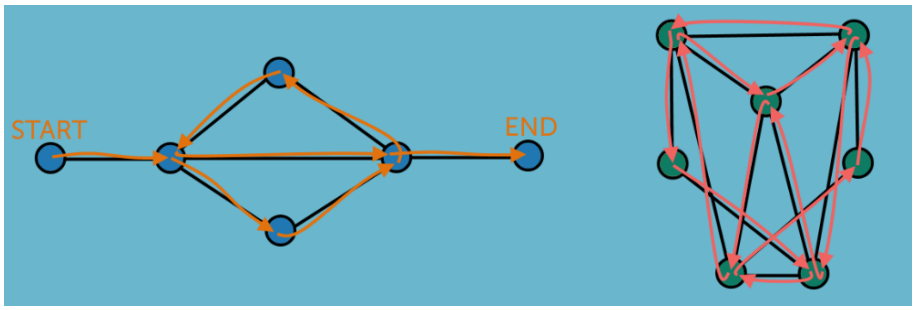
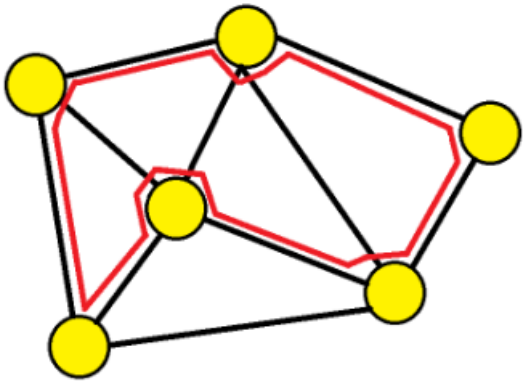


Figure 3.1: Eulerian and semi-Eulerian trail

Definition 3.1.3 — Hamilton Path. If there is a path in the connected graph that visits every vertex of the graph exactly once without repeating the edges then such a path is called Hamiltonian Path.



3.2 Graph Navigation

Given a node the main operations to perform are the following:

- Neighbors: obtain the list of all nodes connected to it.
- Degree: number of nodes connected when graph is undirected or In-degree /out-degree when is directed.
- Graph Traversal: start from a node, obtain the list of all reachable nodes.

Definition 3.2.1 — Reachability. Reachable nodes in a graph are nodes that can be reached from a given starting node by following a path along the edges of the graph. In other words, reachable nodes are those that are connected to the starting node by one or more paths.

Definition 3.2.2 — Graph Traversal. The traverse is the process of moving from vertex to edge or edge to vertex as we navigate through a graph. It is the way to perform query on the graph.

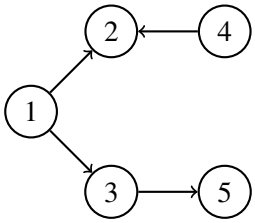


Figure 3.2: Reachable(1) = $\{2,3,5\}$

Exist main ways to perform traversal but the most commons are the following:

- **Depth First Search**: visit a neighbor of the last visited node. It uses the Stack data structure and performs two stages, first visited vertices are pushed into the stack, and second if there are no vertices then visited vertices are popped.

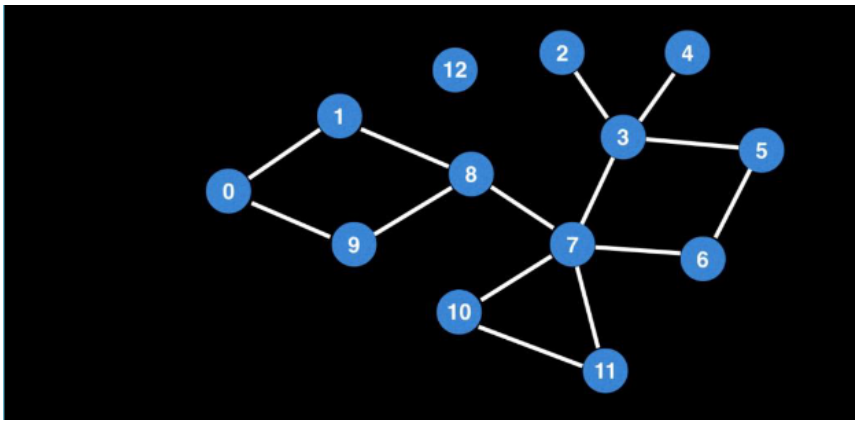


Figure 3.3: DFS: 0,9,8,7,10,11,7,6,5,3,2,4,1

- **Breadth-First Search**: visit first all the neighbors of a node before visiting the other.
  It uses a Queue data structure that follows FIFO policy. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue (It is slower than DFS).

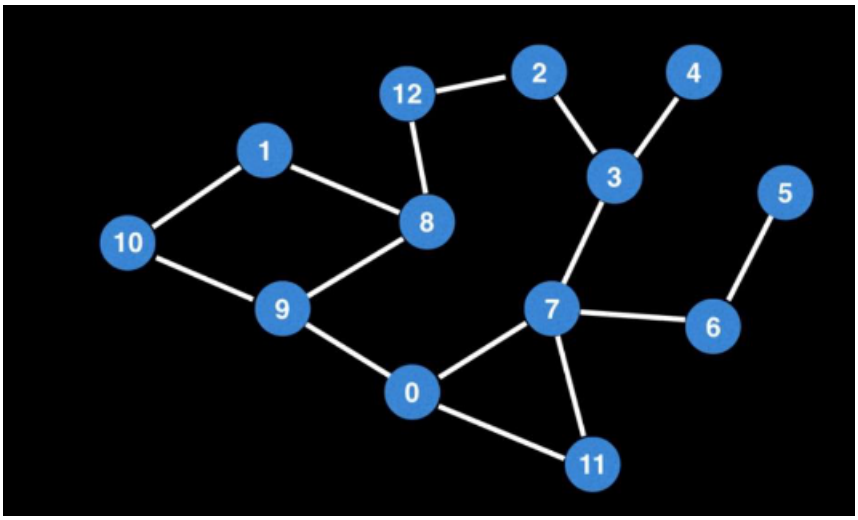


Figure 3.4: BFS: 0,11,7,9,6,3,8,10,5,4,2,1,12

Definition 3.2.3 — Shortest Path. In graph theory, the shortest path problem is the problem of finding the "quickest" path between two vertices (or nodes) in a graph.

If all edges cost the same (unweighted graph) the problem is to find the smaller number of edges (BFS can compute it). Otherwise if edges have different costs (weighted graph) the problem is to find the path such that the sum of the weights of its constituent edges is minimized. Can compute shortest path on different levels:

- **Single source**: given 1 source node find the "quickest" way to reach all other nodes
- **Single-Pair of nodes shortest path**: given a source and a destination, find the "quickest" way between the two (if exists).
- **All-pairs shortest path**: find the shortest paths between every pair of vertices.

The most important algorithms for solving this problem are the following:

- **Dijkstra's algorithm** solves the single-source shortest path problem with non-negative edge weight.
- **Bellman–Ford algorithm** solves the single-source problem if edge weights may be negative.
- **A*** search algorithm solves for single-pair shortest path using heuristics to try to speed up the search.
- **Floyd–Warshall** algorithm solves all pairs shortest paths.
- Johnson's algorithm solves all pairs shortest paths, and may be faster than Floyd–Warshall on sparse graphs.
- Viterbi algorithm solves the shortest stochastic path problem with an additional probabilistic weight on each node.

Shortest path algorithms are applied to automatically find directions between physical locations, such as driving directions on web mapping websites like Google Maps. For this application fast specialized algorithms are available. The algorithm with the fastest known query time is called **hub** labeling and is able to compute shortest path on the road networks of Europe or the US in a fraction of a microsecond.

3.3 Graph Statistics

Definition 3.3.1 — Connectivity. Connectivity means the minimum number of elements (nodes or edges) that need to be removed to separate the remaining nodes into two or more isolated subgraphs. It is an important measure of its resilience as a network.

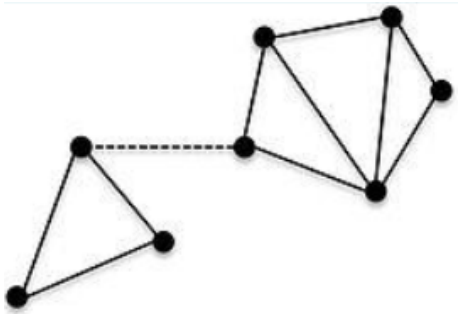


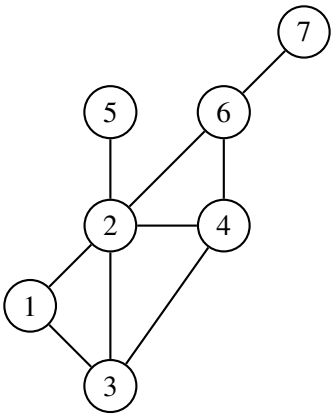
Figure 3.5: This graph becomes disconnected when the dashed edge is removed.

Definition 3.3.2 — Eccentricity of a node. Eccentricity of a node (v) in a connected graph G is the maximum graph distance between v and any other vertex u of G. In another way, the length of the longest shortest path starting at that node.

Corollary 3.3.1 — Radius of a graph. The minimum eccentricity of any node in the graph.

Corollary 3.3.2 — Diameter of a graph. The maximum eccentricity of any vertex in the graph.

If consider the graph shown as in the figure below the values of eccentricity of each node are the following: Eccentricity(1) = 3; Eccentricity(2) = 2; Eccentricity(3) = 3; Eccentricity(4) = 2; Eccentricity(5) = 2; Eccentricity(6) = 2; Eccentricity(7) = 3. The value of radius is 2 and of diameter is 3.



Definition 3.3.3 — Connected component. A connected component is a graph partition where each node can reach all other nodes in the same partition.

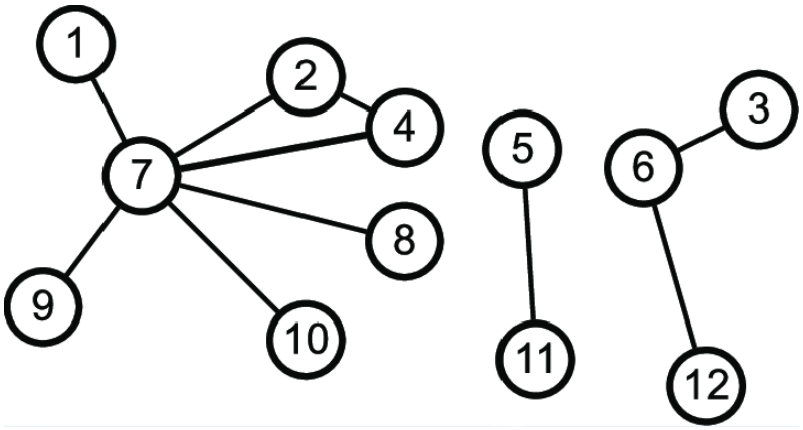
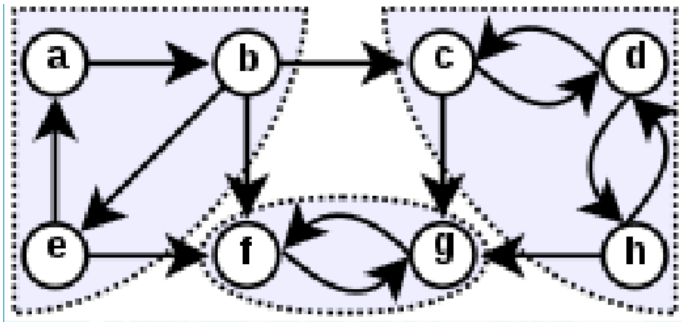


Figure 3.6: There are three connected component in this graph

In a directed graph we can have connected components, but if we follow directions, then it may happen that we cannot reach all nodes so we have to define another statistic.

Definition 3.3.4 — Strongly connected component. A strongly connected component is a portion of a directed graph where there is a directed path between any two nodes (forming a partition into subgraphs).



Corollary 3.3.3 A direct graph is said to be strongly connected if every vertex is reachable from every other vertex

Definition 3.3.5 — Conductance. Given a partition of a graph into two disjoint sets, G1 and G2, the conductance is the sum of the edges going from G1 to G2 divided by the minimum between the edges in G1 and in G2.

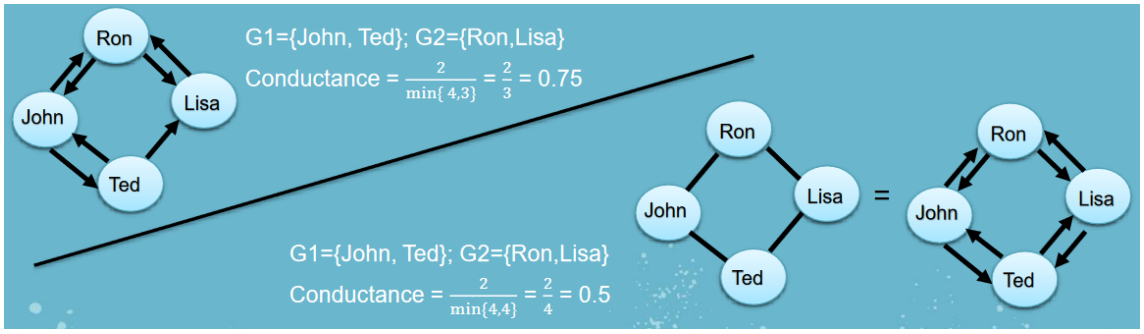


Figure 3.7: In an undirected graph we should consider both the directions for each edge

3.4 Graph Structure

What is the purpose of analyzing the structure of the graph and identifying cut points?

- Identifying areas in the graph (cliques)
- Identifying critical connections
- Clustering and Community Detection (portion of a graph with high internal connectivity)
- Storing the graph in a distributed manner
- Optimizing recommendation computation by focusing on specific areas only

Definition 3.4.1 — Cliques. Subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent

To identify these areas are used different algorithms. The most common is the **K-Core decomposition** is a graph algorithm that identifies the maximal subgraph of a graph where all nodes have at least degree k. It iteratively removes nodes with a degree less than k, and their incident edges, until no nodes with a degree less than k remain.
To analyze the structure of a graph, some measures are needed.

Definition 3.4.2 — Average Diameter L. It is the average length of the shortest paths connecting any two nodes. Give informations about average path in the graph.

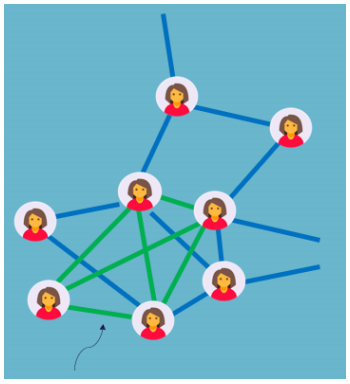


Figure 3.8: Clique

Definition 3.4.3 — Clustering Coefficient or Local Density. The clustering coefficient is Equivalent to the density of the subgraph when considering only the neighbors of n and ignoring n. The local clustering coefficient of a vertex (node) in a graph quantifies how close its neighbours are to being a clique (complete graph).

$$C_n = \frac{\text{edges between the neighbors of n}}{degree(n) * (degree(n) - 1)} \tag{3.1}$$

$$C_n = \frac{2 * \text{edges between the neighbors of n}}{degree(n) * (degree(n) - 1)} \tag{3.2}$$

Small World Graphs have relatively small L and a relatively large C (the distance between two nodes is relatively small).

Definition 3.4.4 — Average clustering coefficient. It is the Clustering Coefficient of the entire graph.

$$C = \frac{1}{|N|} \sum_{n \in N} C_n \tag{3.3}$$

Definition 3.4.5 — Wiener Index: Closeness of a graph. The Wiener index of a graph is a measure of its topological complexity, defined as the sum of all pairwise shortest path distances between nodes in the graph.

$$\sum_{(u,v) \in G} d(u,v) \tag{3.4}$$

3.4.1 Centrality Measures

How important is a node in a graph? Intuitively the importance of a node depends on its role in "keeping the graph connected".
In the case of connected graphs can define the following measures.

Definition 3.4.6 — Degree centrality. It is the number of neighbors of node v

Definition 3.4.7 — Closeness centrality. It is defined as the reciprocal of the total distance

from a node v to all the other nodes in a network.

$$C_c(v) = \frac{1}{\sum_{u \in V} \delta(u,v)} \tag{3.5}$$

$\delta(u,v)$ is the distance between node u and v.

Definition 3.4.8 — Betweenness centrality. It is defined as the ratio of the number of shortest paths passing through a node v out of all shortest paths between all node pairs in a network.

$$C_B(v) = \sum_{s \neq t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{3.6}$$

σ_{st}: number of shortest paths between node s and t $\sigma_{st}(v)$: number of shortest paths passing on a node v

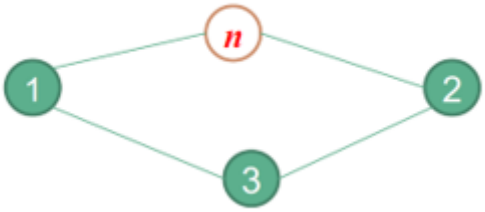
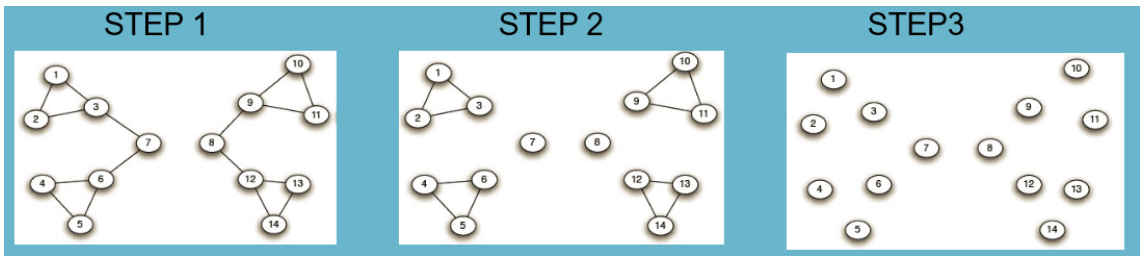


Figure 3.9: Betweeness $C_B(n) = \frac{5}{8}$

3.4.2 Community Detection

To identify cluster the most common algorithm is **Girvan-Newman Algorithm** based on the notion of edge betweenness. The Girvan-Newman algorithm for the detection and analysis of community structure relies on the iterative elimination of edges that have the highest number of shortest paths between nodes passing through them.

1. Calculate edge betweenness: find edges that are more "central" in the graph
2. Delete high-betweenness edges
3. Connected components are communities
4. Repeat until stopping condition



Another method to cut the graph to identify communities is using conductance. If we need to split the graph into 2 parts, we want to minimize the number of edges "break" and we also want the 2 parts to be "densely connected".

In all these algorithms is important to pay attention to overlapped nodes that are nodes with a high level of "importance" in the graph.

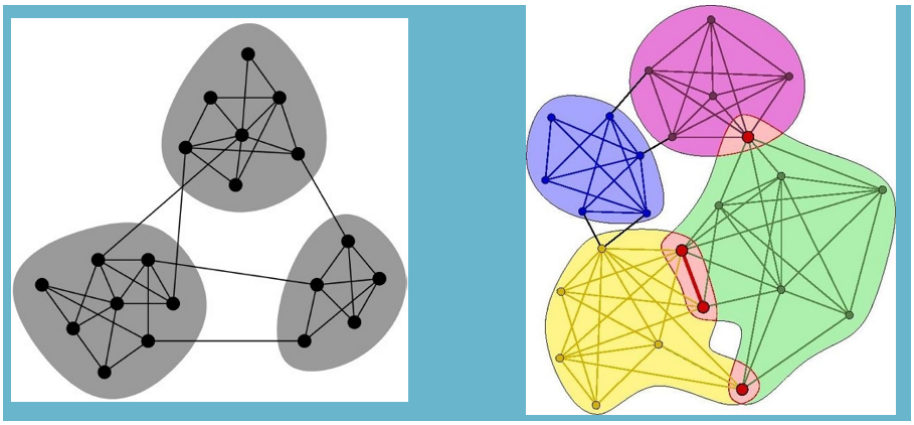


Figure 3.10: Non-overlapping vs. overlapping communities

3.4.3 Importance in Directed Graphs

In directed graphs can define other measures to understand the importance of a node.

Proximity Prestige

Prestige is a measure that consider the importance of the node base on the “in-links". It is usually computed for directed networks only, since for this measures the direction is important property of the relation.

Corollary 3.4.1 — In-Degree prestige. It is the number of incoming links of the node i.

Corollary 3.4.2 — Influence Domain. It is defined as the number of nodes that can reach node i $|I_i|$.

Corollary 3.4.3 — Proximity prestige. Proportional to the count of other nodes that can reach the node with a directed path and the distance of those paths.

$$PP(i) = \frac{|I_i|/(n-1)}{\sum_{j \in I_i} d(j,i)/|I_i|} \tag{3.7}$$

$d(j,i)$ is the length of the directed shortest path

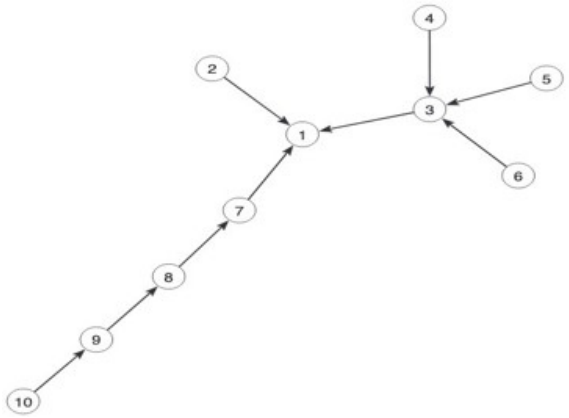


Figure 3.11: In-degree(8)=1, $|I_8| = 2$, $PP(8) = 0.148$

Measure Impact

Consider "importance" of an author as balance between the "productivity" and the "impact".

Corollary 3.4.4 — Productivity. It is the number of publications of an author.

Corollary 3.4.5 — Impact. It is defined as the number of citations of an author's work.

Definition 3.4.9 — H-Index. It is the maximum value of h such that the given author/journal has published h papers that have each been cited at least h times.

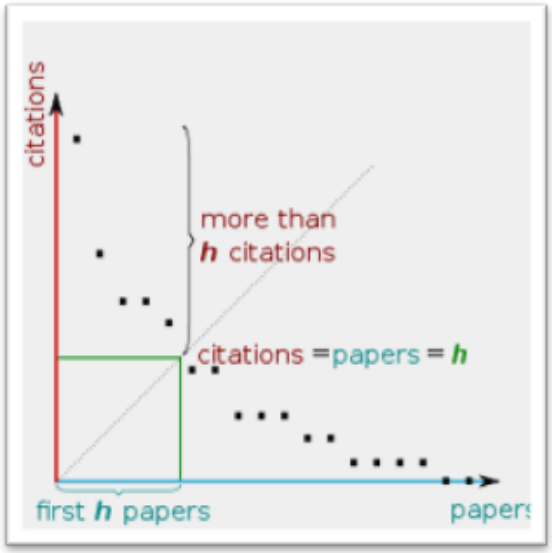


Figure 3.12: All the scientific production of an author can be described in this graph

4. Graph Importance

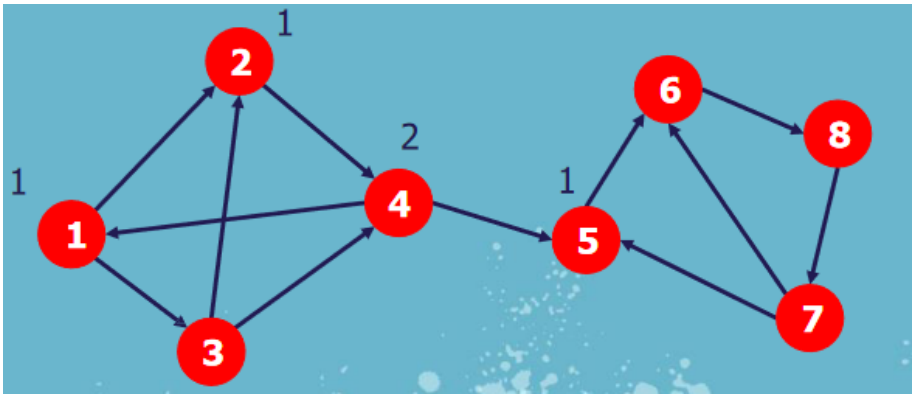
In this section we will discuss about advanced methods to understand the importance of a node.

4.1 Basic Concepts

4.1.1 Random Walk

Definition 4.1.1 — Random walk. Traversal of the graph by selecting neighbours at random. It is possible to visit the same edge/node multiple times. We keep note of the "frequency" with which each node is visited.

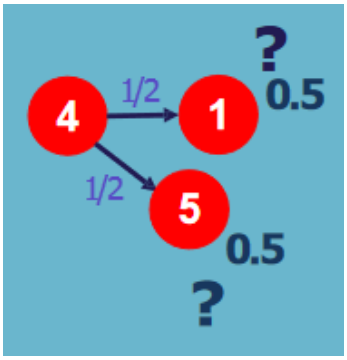
- Pick a node
- Select a neighbour at random: take a step
- Keep making steps until we are "tired
- Take note of the node where we stop and how often we visit each node



Defined the process we ask ourselves a question, given a node N_i ,assuming to pick a random step, what is the probability to end up in its neighbour N_j?

Definition 4.1.2 — Transition Probability. The probability, given a node N_i, to traverse its outgoing edge e_{ij} and reach neighbour node N_j.

If all edges are treated equally, the probability depends on the number of outgoing edges (uniform transition probability) and so for all the edges of a single node always sum to 1.



In the case of weighted edges, we can have non-uniform probabilities.
In case we want to know the probability of ending up in a specific node N_j, after K steps starting at node N_i we can simply compute the joint probability over a path.

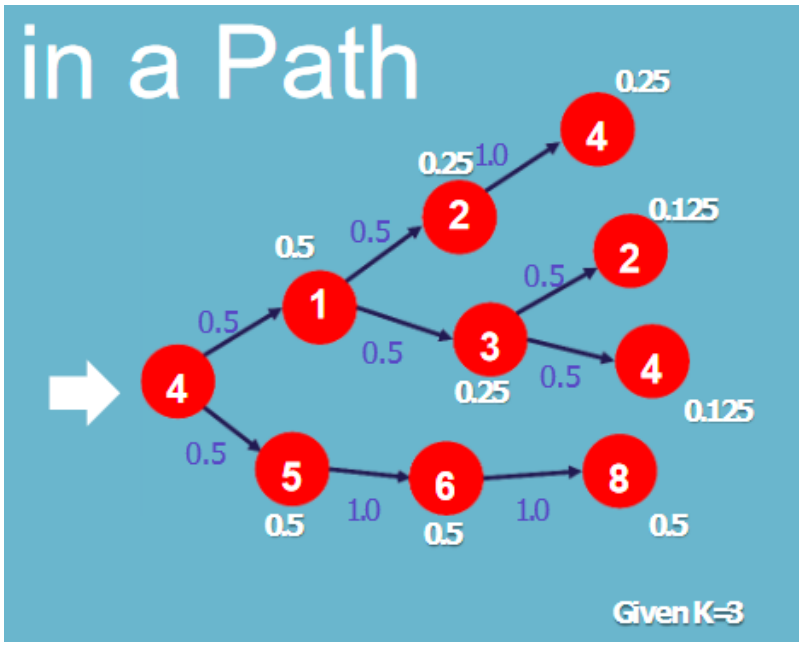


Figure 4.1: If the probability remains the same means that we are on a constraint path

A graph can also be explored by skipping some nodes such as social networks. So how we can model this use case? We model the probability of interrupting the walk with a parameter α, called teleport probability.

Definition 4.1.3 — Teleport Probability. It is defined as the probability of interrupting the random walk and of "jumping" to any other node at random.

$$\text{Transition probability} + \text{teleport probability} = 1 \tag{4.1}$$

A real use-case can be the **Random Surfer**, a web-user that starts from a page, follows links at random, after some clicks decides to start from a random new page

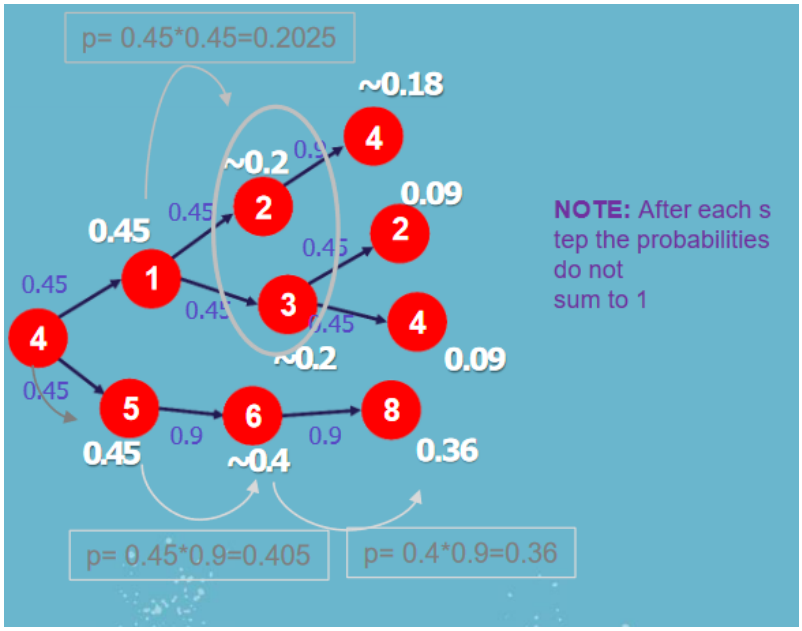
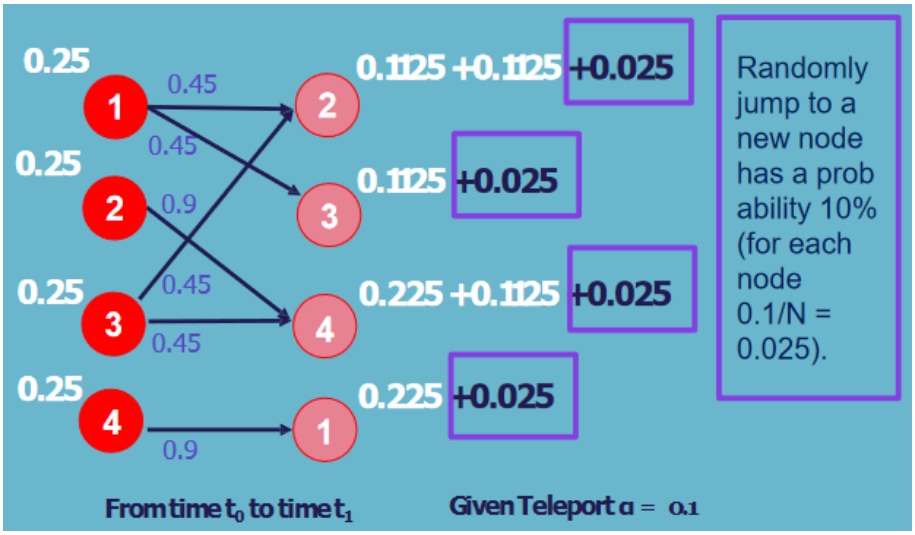


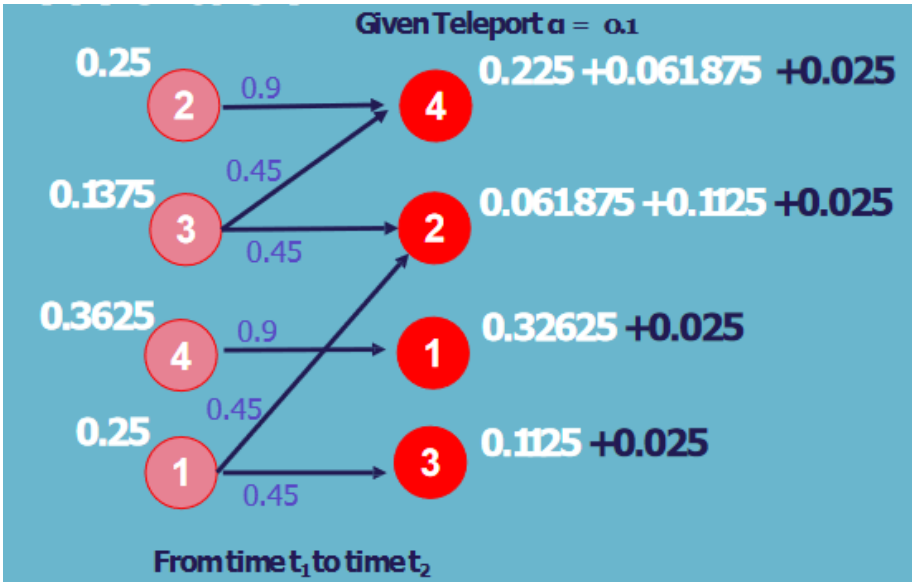
Figure 4.2: Probabilities given $\alpha = 0.1$. We can see that the final probability depends on the number of steps k

4.1.2 Markov Model

How do we model the random walk for each node?

- Each node is a starting point with equal probability
- Each node is connected to the neighbours
- We set a teleport probability to end in any other node
- Edges have uniform transition probabilities
- We sum the transition probabilities of each node and the teleport probability (uniformly distributed).





From this process we can extract an important property.

Corollary 4.1.1 The future is independent of the past, given the present. To compute the next probability we can just use the actual one.

In the **Markov chain**, each node in the graph is regarded as a state. An edge is a transition, which leads from one state to another state with a certain probability.

4.1.3 Algebraic representation

To describe in more formal way the markow model we need to define some measures.

Definition 4.1.4 — Adjacency Matrix. Adjecency Matrix is *NxN* matrix where entry is 1 if there is an edge between i and j and 0 if there is not.

$$\begin{bmatrix} & 1 & 2 & 3 & 4 \\ 1 & 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 1 & 0 & 1 \\ 4 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{4.2}$$

Definition 4.1.5 — Transition Probability Matrix. The transition probability matrix is derived from A by normalizing the rows and the entries are the transition probabilities from i to j (rows always sum to 1).

$$\begin{bmatrix} & 1 & 2 & 3 & 4 \\ 1 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 4 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{4.3}$$

So given the Transition probability matrix T, the initial vector of probabilities v of each node and the teleport probability α we can compute the probability to time t_{i+1} in the following way. We refer to this process as **Power Iteration**.

$$(1-\alpha)\cdot \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix}^T \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}_{t_i} + \alpha\cdot \begin{bmatrix} \frac{1}{n} \\ \vdots \\ \frac{1}{n} \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}_{t_{i+1}}$$

4.2 Application

One of the most common applications of power iteration is the **Citation network**. It is a directed acyclic graph representing scientific publications as nodes, an edge goes from A to B if A has a reference to B (A citation!).
On this type of graph we can perform citation analysis, a type of analysis that uses these relationships (links) to extract some information.

4.2.1 Co-Citation

An useful analysis is understand, given a citation network, what are the papers most cited.

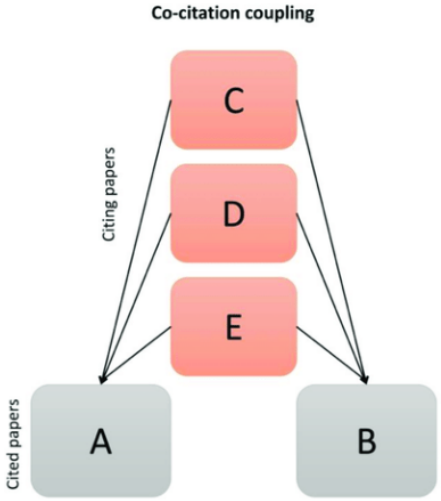


Figure 4.3: Papers A and B are associated because they are co-cited in the reference list of papers C, D and E

How we can model this type of network?

Definition 4.2.1 — Citation Matrix. Let L be the citation matrix where each cell of the matrix is defined as:

$$L_{ij} = \begin{cases} 1, & \text{if paper } i \text{ cites paper } j \\ 0, & \text{otherwise} \end{cases} \tag{4.4}$$

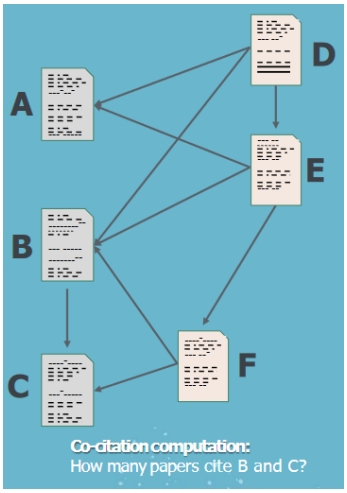
So we can define the **co-citation matrix** that is a similarity measure defined as the number of papers that co-cite i and j.

Definition 4.2.2 — Co-citation Matrix.

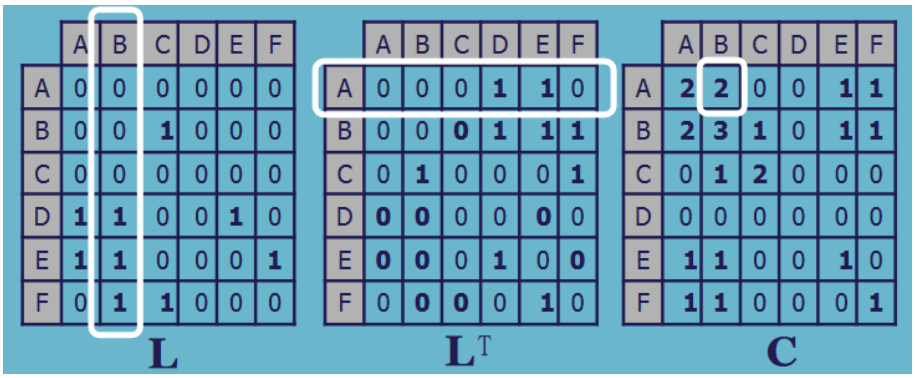
$$C_{ij} = \sum_{k=1}^{n} L_{ki} L_{kj} \tag{4.5}$$

C_{ii} is the number of papers that cite i.

To understand clearly the process we can see the following example.



We can compute citation matrix and then the co-citation matrix using the formula $C = L^T xL$.



4.2.2 Bibliographic Coupling

Bibliographic coupling occurs when two works reference a common third work in their bibliographies. It is an indication that a probability exists that the two works treat a related subject matter.

Bibliographic coupling operates on a similar principle to Co-citation linking papers that cite the same articles. Then we defined the **co-reference matrix** B_{ij} to represent the number of papers that are cited by both papers i and j.

Definition 4.2.3 — Co-reference matrix.

$$B_{ij} = \sum_{k=1}^{n} L_{ik} L_{jk} \tag{4.6}$$

B_{ii} is the number of references of paper i.

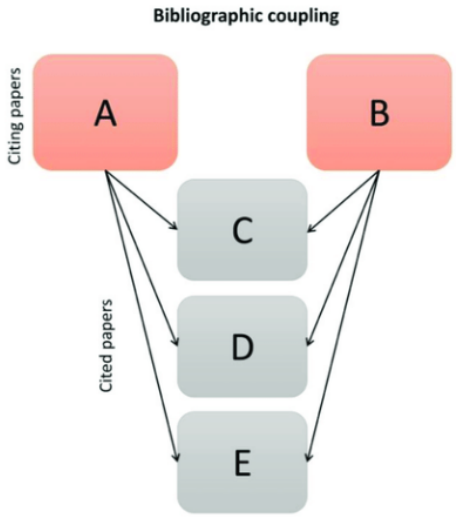
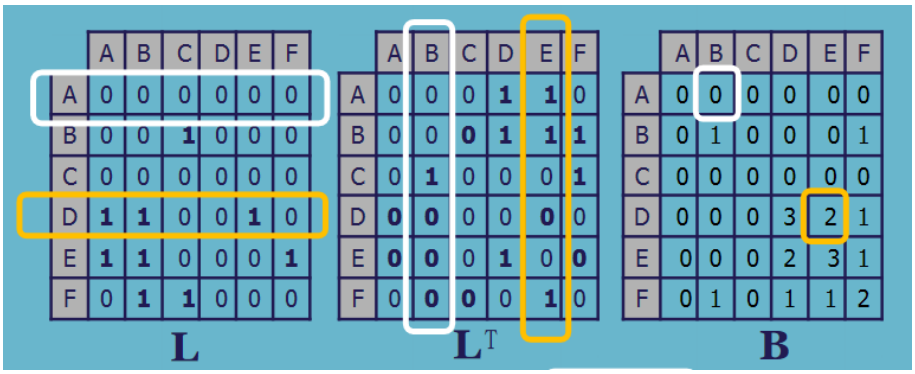


Figure 4.4: Papers A and B are bibliographically coupled because they have cited papers C,D and E in their reference list

To understand clearly the process we can compute the co-reference matrix of the previous example with the formula $B = LxL^T$



4.2.3 HITS

Hypertext Induced Topic Search is an analysis algorithm to understand links between web pages. It is composed of two main steps:

1. First expands the list of relevant pages returned by a search engine (it retrieves all the pages linked to and linked by the result pages)
2. Then produces two rankings of the expanded set of pages, authority ranking and hub ranking

Definition 4.2.4 — Authority. An authoritative page is a page with many in-links and many people trust it and link to it.

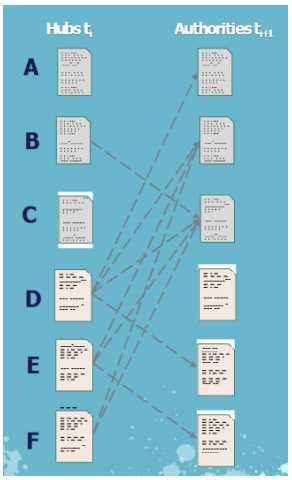
Definition 4.2.5 — Hub. A hub is a page with many out-links, that is a page serves as an organizer of the information on a particular topic.

Corollary 4.2.1 A good hub points to many good authorities, otherwise a good authority is pointed to by many good hubs.

Authorities and hubs have a mutual reinforcement relationship with scores that influences each

other. So to model this type of scenario we use a bipartite sub-graph, with nodes that have both an authority and hub score.
To compute scores we proceed iteratively:

1. First, at time t_0 , we assign an authority score and a hub score to all nodes (initial score $\frac{1}{\sqrt{N}}$.
2. Then (from time t_0 to time t_1) we accumulate the score from each authority to each hub and from each hub to each authority.
3. We repeat the process from each time t_i to time t_{i+1} updating both values at each step.



So more formally given N web pages, vector a = $(a_1,\ldots,a_n)$ of authority scores, vector h = $(h_1,\ldots,h_n)$ of hub scores and adjacency matrix between nodes, we can compute the next scores with the power iteration (to repeat until convergence):

- $h_i = \sum_j A_{ij} a_j = A * a$
- $a_i = \sum_j A_{ji} h_j = A^T * h$
- Normalize a and h

$$h_j^{(t+1)} = h_j^{(t+1)} / \sqrt{\sum (h_j^{(t+1)})^2} \tag{4.7}$$

$$a_i^{(t+1)} = a_i^{(t+1)} / \sqrt{\sum (a_i^{(t+1)})^2} \tag{4.8}$$

Under reasonable assumptions about A, HITS converges to vectors h* and a*.

4.2.4 Page Rank

Page rank is an algorithm used to estimate the "importance" of a web page. To model this scenario we use a graph where links represent a vote from page i to page j. To identify the most important nodes can we just count votes and select pages with the most votes? No, counting the votes is not enough because in this way people can just create empty pages that link to their page to increase the importance.
So to compute importance in an effective way each link's vote must be proportional to the importance of its source page and the sum of the votes on its in-links. To do that we need to define a **rank**.

Definition 4.2.6 — Rank. Given rank of the node i with a link toward j and the out-degree of node i, rank r_j for page j is defined as following:

$$r_j = \sum \frac{r_i}{d_i} \tag{4.9}$$

As the previous examples we can compute ranks with **power iteration**.

Definition 4.2.7 — Stochastic adjacency matrix M. Let M be the column stochastic adjacency matrix where each cell of the matrix is defined as:

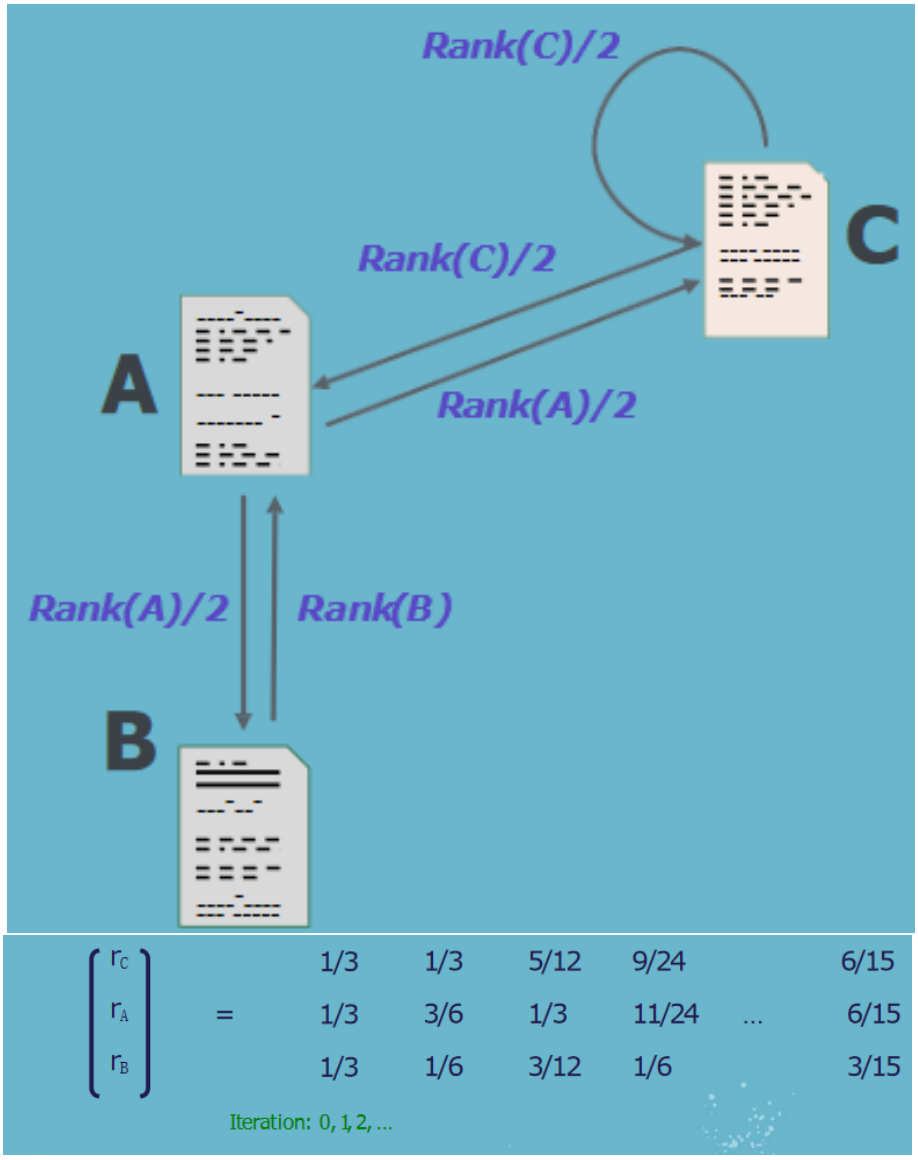
$$M_{ji} = \begin{cases} \frac{1}{d_i}, & \text{if there is a link } i \longrightarrow j \\ 0, & \text{otherwise} \end{cases} \tag{4.10}$$

Defined r as the rank vector we can update this vector with the following equations

$$r = M * r \tag{4.11}$$

So the simple iterative scheme of power iteration will be the following:

- Initialize the vector: $r^{(0)} = [1/N, \ldots, 1/N]^T$
- Iterate: $r^{(t+1)} = M * r^{(t)}$
- Stop when $|r^{(t+1)} - r^{(t)}|_1 < \varepsilon$ (norma 1)



The final result is the "stationary distribution", a probability distribution over pages. Not every power iteration convergence to a stationary distribution, there are cases where convergency can be an issue:

- **Dead-end**: a node with no out-links, a sink, the random walk reaching here will then stop and the "importance" will be lost in the next iteration.
- **Spider trap**: nodes in a cycle with no exit, a loop, the random walk will iterate continuously without reaching the stable state.

To avoid this type of issue we have to define the graph as the Markov Process and respect the following constraints.

> **Corollary 4.2.2 — Existence and Uniqueness of Stationary Distribution in Markov Processes.** The stationary distribution is unique and eventually will be reached no matter what the

initial probability distribution at time t = 0, for graphs that satisfy the following conditions:

1. The transition matrix is stochastic: all columns sum to 1 (add the teleport probability)
2. The matrix is irreducible so the graph is strongly connected
3. The matrix is aperiodic so it is possible to come back to each node in a fixed number of steps

An example of page rank modeled as Markov Process is Google's page rank. In fact in Google's solution at each step, random surfer has two options:

1. Follow a link at random with a probability β (damping factor)
2. Jump to some random page with probability $1 - \beta$,

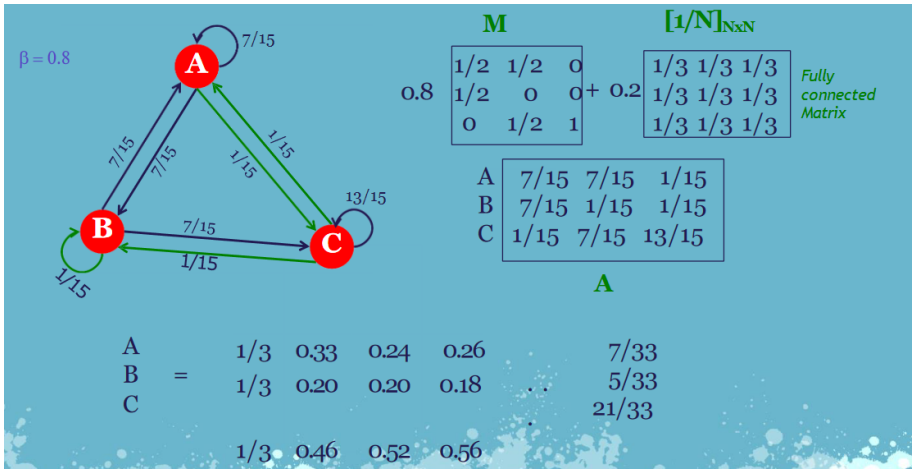


Figure 4.5: Example of page rank computation

4.2.5 Personalized Page Rank

Page Rank measures a "generic" popularity of a page, is no specific for a search query or a topic. The goal can be evaluate Web pages not just according to their popularity, but by how close they are to a particular topic. How can we do that?
Starting from a limited set of nodes, traversing randomly, restart point is one in the initial set. More formally:

1. When walker teleports, she pick a page from a set S
2. The set S contains only pages that are relevant to the topic
3. For each teleport set S, we get a different vector r_S

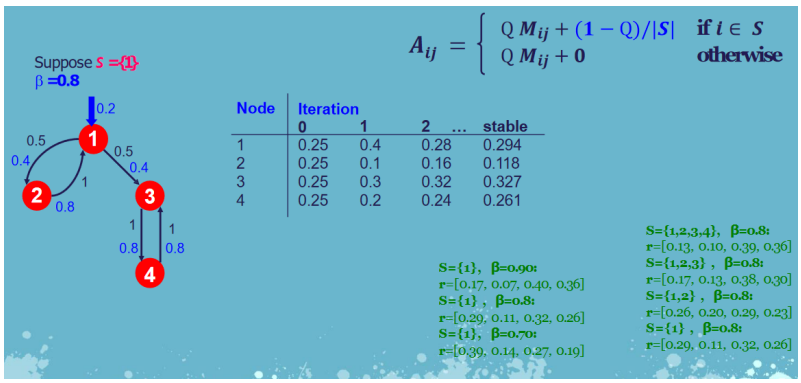


Figure 4.6: Example of personalized page rank computation

II Graph Database

5. Graph Database

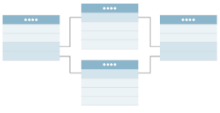
5.1 What is a graph database?

A graph database is a data-storage engine that combines the basic graph structures of vertices and edges with persistence technology and a traversal (query) language to create a database optimized for storage and fast retrieval of highly connected data.

Graph databases are less structurally rigid compared to relational databases, and they assign equal importance to entities and relationships. Relational databases use foreign keys as relationships, which are pointers to primary keys in other tables. These pointers are not easy to observe and manipulate. On the other hand, graph databases provide excellent tools for traversing relationships in the data. Graph databases make the connections or edges as important as the entities, and represent these connections as full-fledged constructs of the database that can be easily observed and manipulated.

Graph databases enhance developer productivity. Storing data to better represent its real-world counterpart can make it easier for developers to reason over and understand the domain in which they are working.

Graph database vs. relational database

	Graph database	Relational database
FORMAT	Nodes and edges	Tables with rows and columns
RELATIONSHIPS	Considered data, represented by edges between nodes	Related across tables, established using foreign keys between tables
COMPLEX QUERIES	Run quickly and do not require joins	Require complex joins between tables
TOP USE CASES	Relationship-heavy use cases, including fraud detection and recommendation engines	Transaction-focused use cases, including online transactions and accounting
EXAMPLE		

There are three areas where graph databases provide a simpler, more efficient solution than using a relational database:

1. **Recursive queries** (for example, an hierarchy)

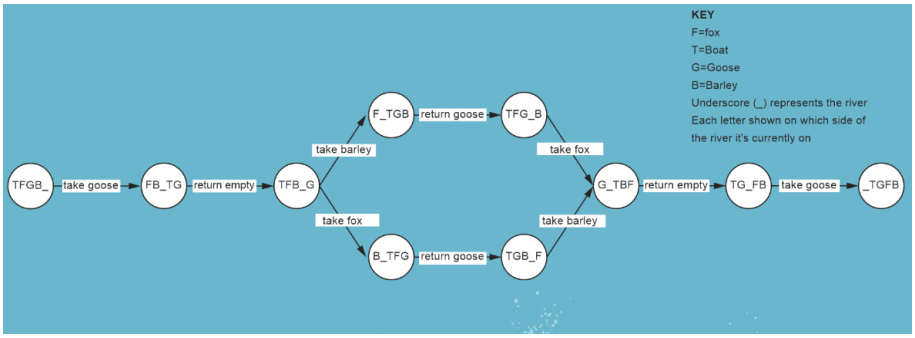
```
WITH RECURSIVE org AS (
    SELECT employee_id, manager_employee_id,
        employee_name, 1 AS level
    FROM org_chart
 UNION
    SELECT e.employee_id, e.manager_employee_id,
        e.employee_name, m.level + 1 AS level
    FROM org_chart AS e
     INNER JOIN org AS m
ON e.manager_employee_id = m.employee_id
 )

SELECT employee_id, manager_employee_id, employee_name
FROM org
ORDER BY level ASC;
```

```
g.V().
  repeat(
'works_for')
  ).path().next()
```

Figure 5.1: Relational vs Graph in Recursive query

2. **Different result types** (features of nodes are not mixed together).
3. **Paths** (we can model all the status of our system)



Obviuosly everything isn't roses because also graph DB suffer of some potential issues:

- **Security and privacy**: graph databases require more strict implementations of security and access measures.
- **Data integrity implications** — Graph databases simplify the ways in which information relates to other information (avoid conflicts). In doing so, shortening or condensing the relationship it's particularly vital that all data in a graph database is accurate.

5.2 Types of graph databases

- **Property Graph Database**: it stores data as nodes and edges, and allows for properties to be assigned to both.
- **Resource Description Framework (RDF) Graph Database**: it uses the RDF model, which represents data as triples consisting of a subject , predicate, and object.
- **Hypergraph Database**: it stores nodes and hyperedges; hyperedges connect any number of nodes at once and are useful to connect subset of nodes
- **Spatial Graph Database**: it is designed for storing and querying spatial data, such asmaps, GPS coordinates, and geographical features. Nodes represent points/locations, and edges represent relationships between those points. They support spatial indexing and spatial querying.

- **Temporal Graph Database**: it is designed for storing and querying data with a temporal dimension, such as time series data or event logs. Nodes represent entities, and edges represent relationships between those entities at specific points in time.
- **Property-Graph/Triplestore Hybrid**: a graph database that combines features of both property graph and RDF triplestores.

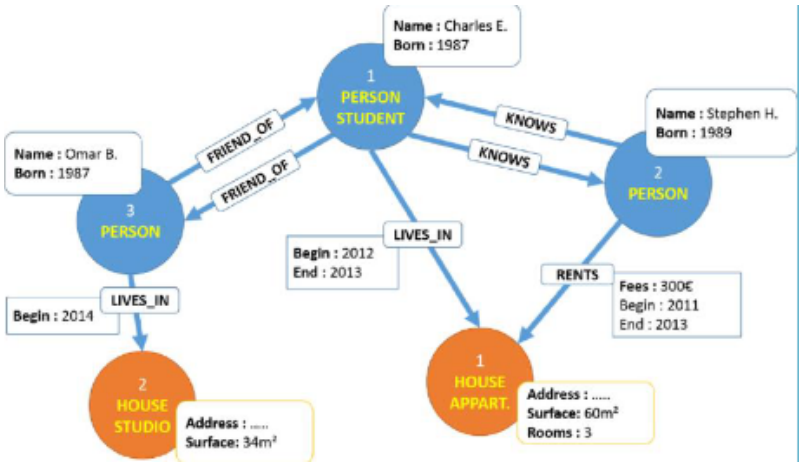


Figure 5.2: Property Graph DB

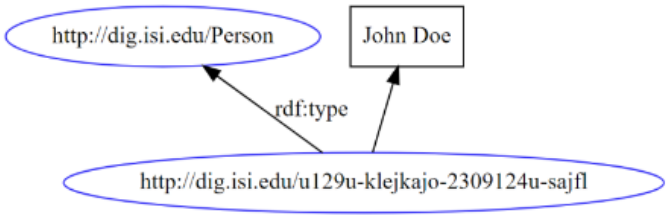


Figure 5.3: RDF Graph DB

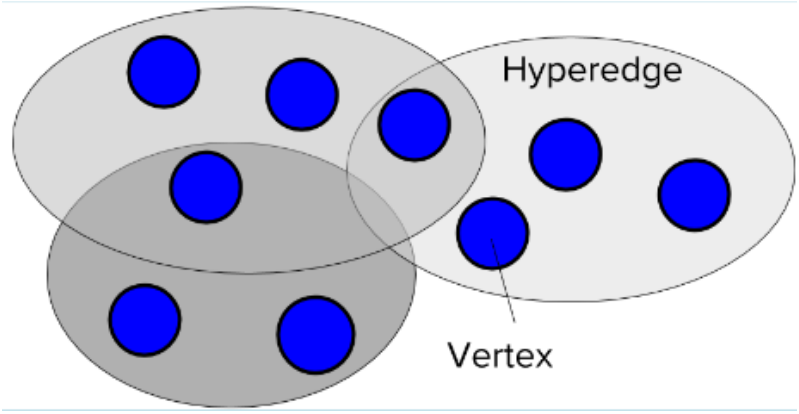


Figure 5.4: Hypergraph DB

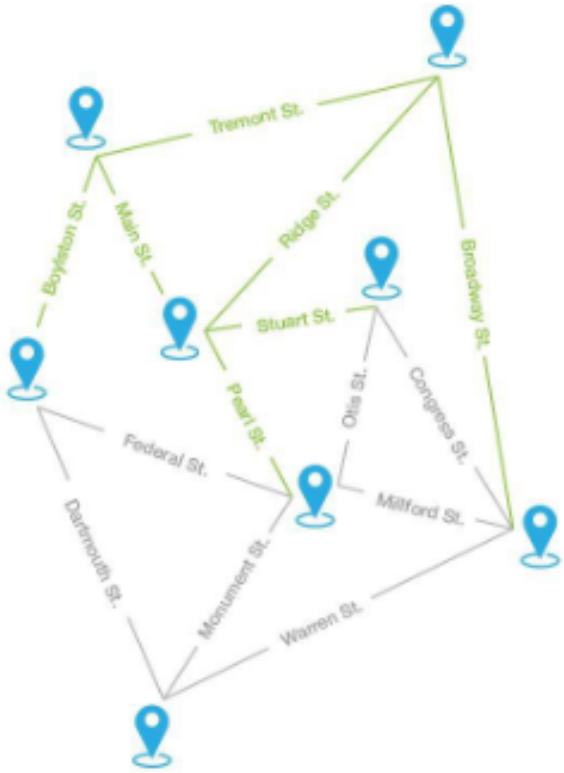


Figure 5.5: Spatial Graph DB

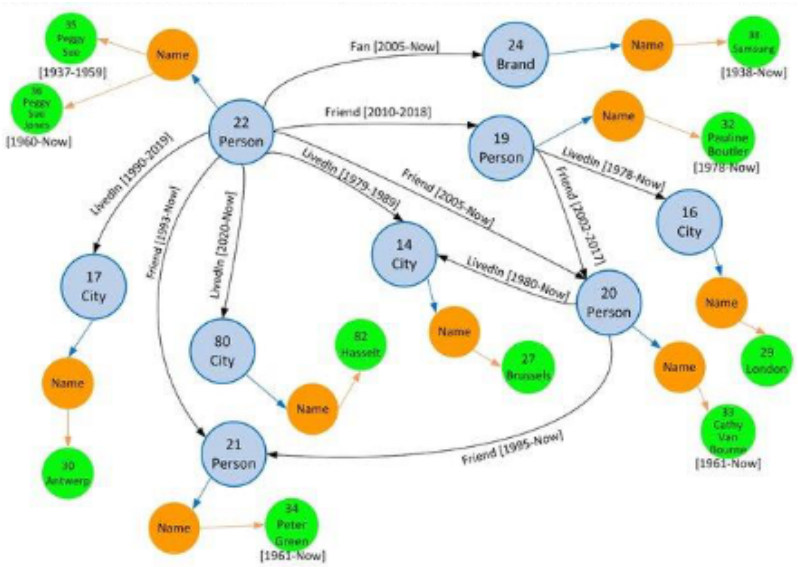


Figure 5.6: Temporal Graph DB

5.3 Tools

5.3.1 To model graph DB

- **Neo4j** is a graph database management system that is designed to store and process highly connected data. Neo4j uses a property graph data model, which means that each node and edge in the graph can have properties associated with it. It is optimized for high performance

and scalability, and it has a number of built-in features that make it easy to work with graph data.
- **Apache TinkerPop** is a graph computing framework that provides a set of interfaces and APIs for working with graph data in a vendor-agnostic way. It provides a standard set of operations and algorithms that can be used across different graph databases, and supports a variety of programming languages, including Java, Python, and .NET.

5.3.2 To query graph DB

- **GraphQL** is a declarative query language for APIs, can be used with a variety of database systems. It is designed to be highly flexible and adaptable to different types of applications.
- **Cypher** is a declarative query language that is designed specifically for Neo4j. It provides a simple, expressive syntax for querying and manipulating graph data, and supports a range of operations and functions that are optimized for Neo4j's storage.
- **Gremlin** is a functional, declarative graph traversal language that is also part of the TinkerPop framework. It provides a set of operators and functions for querying and manipulating graph data, and supports a range of graph algorithms and traversal strategies.
- **GQL** (Graph Query Language) is a proposed standard graph query language. (should officially be published in late April or early May of 2024!)

6. Graph modeling

Data modeling is the process of translating real-world entities and relationships into equivalent software representations. In the graph data modeling process we must shift from an “entity first” mindset to an “entity and relationship” mindset. ER model can be a good data model to use but the counterpart software DBMS use relationship model that is not good to represent relationship between entity.
The advantage of most graph databases is that are schemaless, there is no predefined schema that dictates how nodes and relationships should be defined; nodes and relationships can be added, modified, or removed on the fly (schema depends on the number of nodes). To model data in the right way we have to follow this four-step process:

1. Understand the problem.
2. Create a conceptual model
3. Create a logical data model
4. Test the model

6.1 Understand the problem

Specified the common terms and the core access patterns of users.

- Domain and scope: What will the app/the graph do for its users?
- Business entity: What are the fundamental building blocks of our application? How are these related to one another?
- Functionality: How are people going to use the system?

To understand better the process we can take an example of an app that provides users with personalized restaurant recommendations, called DiningByFriends.

6.1.1 Domain and Scope

When using DiningByFriends, users have three main needs that the application must satisfy:

1. Want to connect with friends who are also using the application
2. Want to create and look at reviews of restaurants and then get recommendations for a

restaurant based on these reviews Want to rate the reviews of restaurants to indicate whether the review was helpful or not

6.1.2 Business Entity

What are the fundamental building blocks of our application an how are related?

- People write reviews.
- Reviews discuss restaurants.
- Restaurants serve one or more types of food.
- A person is friends with another person.
- People rate reviews.

In this phase we have to define properties of our blocks. For example people have a first and last name, restaurant have a name, address and a types of foods served, exc.. .

6.1.3 Functionality

Understand how people will use the app means to answer to the following questions:

- Who are my friends?
- Who are the friends of my friends?
- How is user X associated with user Y ?
- What restaurant near me with a specific cuisine is the highest rated?
- Which restaurants are the 10 highest-rated restaurants near me?
- What are the newest reviews for this restaurant?

6.2 Develop the conceptual model

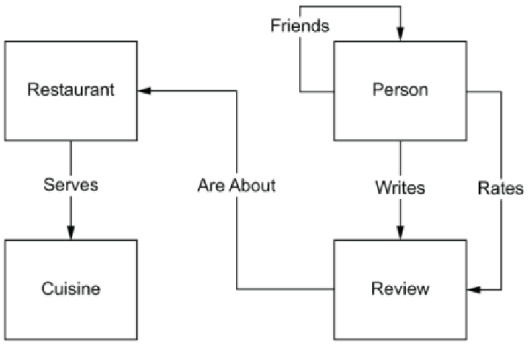
In this phase we have to identify and group entities and identify relationships between them. In the example our entities are the following:

- Restaurant, which includes name and location.
- Cuisine, describes the type of food served.
- Person, which includes the first and last name.
- Review, which includes the full review text and rating.

Between these entities we can identify the following relationship:

- Person-Friends-Person: this relationship helps construct the social network component of our app by allowing the application to answer user questions such as who are my friends or who are the friends of my friends.
- Person-Writes-Review: this relationship enables us to construct the recommendation engine for DiningByFriends as the reviews serve as the underlying data to provide recommendations.

The final conceptual model can be something describe by the image below.



6.3 Building a Graph data model

There are four steps to building a (logical) graph data model from a conceptual model:

1. Translate entities to vertices.
2. Translate relationships to edges.
3. Find and assign properties to vertices and edges.
4. Check your model.

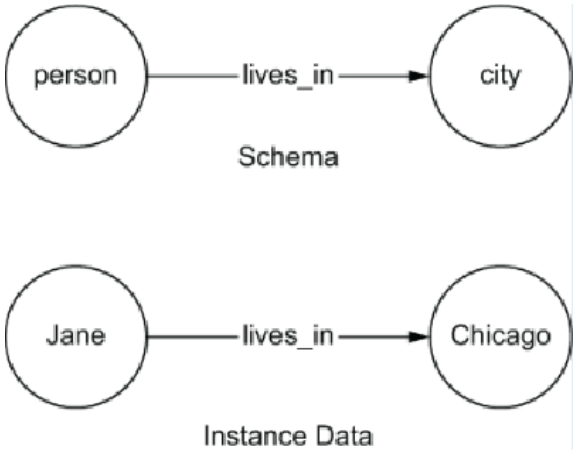
6.3.1 Translating entities to vertices

The entities in a conceptual data model map almost directly to the vertices in a logical graph model. The creation of the vertices in our graph model requires two things: identify all the relevant entities from our conceptual model and give the vertex a name in the form of a label to uniquely identify that type of entity in our graph model.
It is a best practice to make vertex labels singular because each vertex only refers to a single instance of an item.

6.3.2 Translating relationships to edges

Edges are based on the relationships from our conceptual model. Defining edges takes a little more effort than finding the vertices. The edges in graph databases include features like directionality and uniqueness,which do not have direct counterparts in relational databases.

- Identifying the relevant relationships from the conceptual data model. In our example are Person-Friends-Person and Person-Writes-Review.
- Naming the edge in the form of a label to uniquely identify that relationship in our graph data model. Apply the same best practice naming rules: be concise, descriptive, and generic.
- Giving the edge a direction. A good practice is that the vertex-edge-vertex combinations should be read as a sentence, for example "Bill friends Ted". The direction of an edge should complement the edge label to make a sentence that sounds natural (or mostly natural) and that fits the functional needs of the use case.



- Determining edge **uniqueness** is described as the number of times an instance of a vertex is related to another instance of a vertex with an edge having the same label. Describes what is the allowable number of edges of a given label between two vertices. For example in a movie app such as Netflix if we want to store how many times a user watches a certain film we have to use multiple uniqueness with more edges with the same label between two vertex.

Keep in mind that uniquenss is something different to cardinality and multiplicity.
Cardinality is the number of elements in a set (for example, has two Y edges between A and B).
Multiplicity is a specification of the minimum and maximum cardinality that a set can have while

data uniqueness describes the measure of duplication of identified data items within a data set. Uniqueness can be of two type:

- **Single uniqueness** refers to zero or one edge
- **Multiple uniqueness** means more than one possible edge

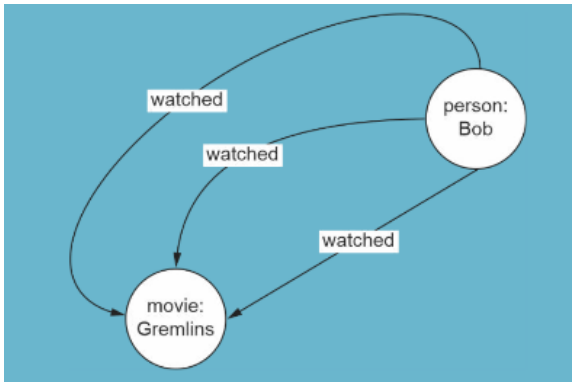


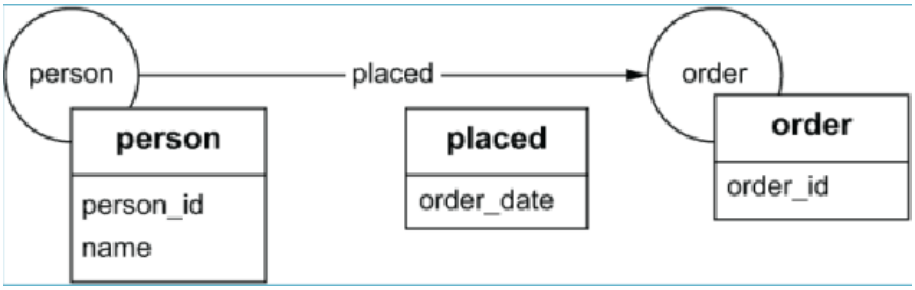
Figure 6.1: Multiple uniqueness example

Incorrect uniqueness can affect an application to occurs the following issues:

- Too little data returned: when we have an edge with single uniqueness that really should be multiple.
- Duplicated data returned: when we have multiple uniqueness edges but we should only have on
- Poor query performance: most often having a multiple uniqueness edge instead of a single uniqueness edge is what causes poor query performance, because the database has to do more work to return the data for a query with multiple edges.

6.3.3 Find and assign properties to vertices

Properties in a graph data model are key-value pairs that describe a specific attribute of a vertex or edge. To understand that we have to look to functionality's questions. Based on these questions, we can see that we need to store, our example, the first and last name for a person to identify who our friend is. We can also assume that we are going to need a unique identifier of each person to differentiate between people with the same name (unlike DBMS in graph DB don't exist the concept of primary key). Another difference with DBMS is that, unlike columns, an application does not insert default values nor null values into properties in graph databases; a null value means that the edge doesn't exist. Similarly, we can set properties on edges (weights).



6.3.4 Check your model

The last step in creating our logical data model is to validate that we can answer the questions for our social network use case and that the model we built follows best practices for graph data

modeling. In our example looking at our questions, can we answer these using our graph data model?

- We can answer to "Who are my friends?" question by starting at a specific person and traversing the friends edge to see all of their friends.
- We can answer to "Who are the friends of my friend?" question by starting at a specific person and traversing the friends edge, and then traversing the friends edge again to see all of the friends of my friends.

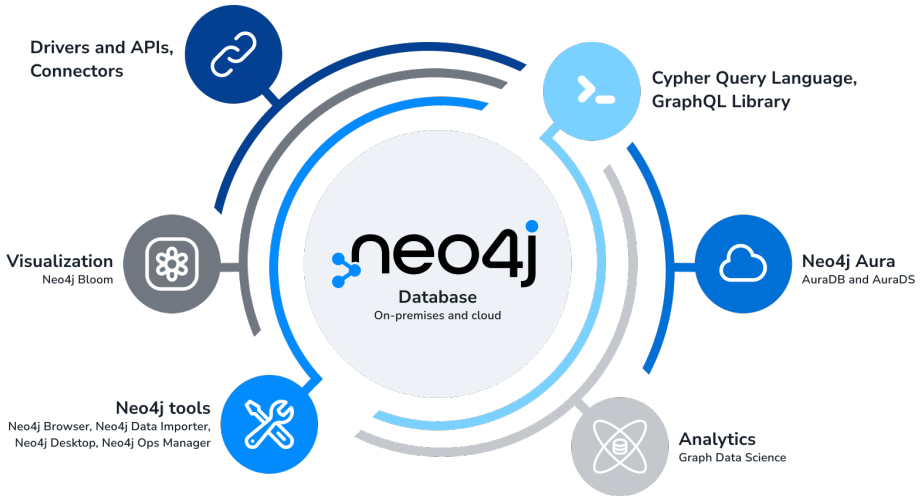
6.4 Test your model

Start using the model and see if modifications are needed (the schemaless property is an advantage).

7. Neo4j

7.1 Architecture

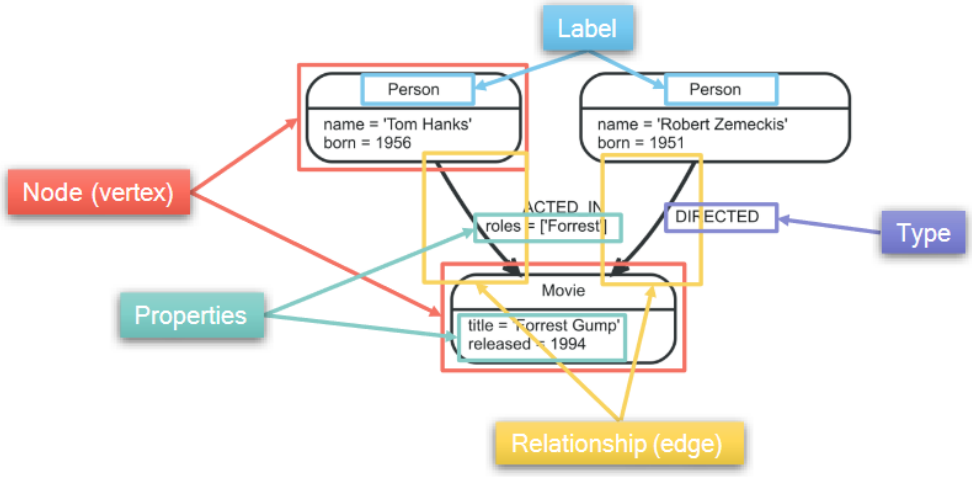
Neo4j is a graph database that manages data in a natural, connected state by using a property graph approach. This approach has benefits for both traversal performance and operations runtime. Neo4j has become a rich ecosystem with various tools, applications, and libraries that allow for seamless integration of graph technologies with your working environment.



7.2 Terminology and Syntax

As other graph DB languages, Neo4j define various components to model graphs.

- Label (name of a node)
- Relationship(edge) and his type
- Node
- Properties of a node or a relationship



7.2.1 MATCH-clause

Match is primary way of getting data from a Neo4j database specifing the patterns. Query can have multiple Match clauses.

```
MATCH (node)-[relationship]-(node)
```

```
MATCH (p:Person)-[:Likes]-(f:Person)
```

7.2.2 RETURN-clause

Allows projection to nodes, edges, and properties (only once per query).

```
MATCH (p:Person)-[:Likes]-(f:Person)
RETURN p.name,f.sex
```

```
MATCH (n)
RETURN n, "node " + id(n) +" is " +
    CASE WHEN n.title IS NOT NULL THEN "a Movie"
        WHEN EXISTS(n.name) THEN "a Person"
        ELSE "something unknown"
    END AS about
```

As other programming languages we can use aggegration function as COUNT,SUM,AVG, MIN, MAX, DISTINCT(removes duplicates in a group before the aggregation).

```
MATCH (me:Person {name:'An'})-->(friend:Person)-->(friend_of_friend:Person)
RETURN me.name, count(DISTINCT friend_of_friend), count(friend_of_friend)
```

7.2.3 Vertex pattern

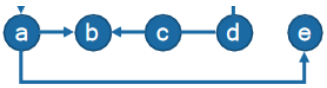
()	unidentified vertex
(matrix)	vertex identified by variable *matrix*
(:Movie)	unidentified vertex with label *Movie*
(matrix:Movie:Action)	vertex with labels *Movie* and *Action* identified by variable *matrix*
(matrix:Movie {title: "The Matrix"})	+ property *title* equal the string "The Matrix"
(matrix:Movie {title: "The Matrix", released: 1997})	+ property *released* equal the integer *1997*

7.2.4 Edge pattern

`-->`	unidentified edge
`-[role]->`	edge identified by variable *role*
`-[:ACTED_IN]->`	unidentified edge with label *ACTED_IN*
`-[role:ACTED_IN]->`	edge with label *ACTED_IN* identified by variable *role*
`-[role:ACTED_IN {roles: ["Neo"]}]->`	+ property *roles* contains the string "Neo"

7.2.5 Path pattern

String of alternating vertex pattern and edge pattern
Starting and ending with a vertex pattern
(a)-->(b)<--(c)--(d)-->(a)-->(e)



(keanu:Person:Actor {name: "Keanu Reeves"})-[role:ACTED_IN {roles: ["Neo"]}]-> (matrix:Movie {title: "The Matrix"})

We can also assign matched paths to variable.

```
p = ((a)-[*3..5]->(b))
```

Repetitive edge types can be expressed by specifying a length with lower and upper bounds

```
(a)-[:x*2]->(b)      is equal to          (a)-[:x]->()-[:x]->(b)
(a)-[*3..5]->(b)
(a)-[*3..]->(b)
(a)-[*..5]->(b)
(a)-[*]->(b)
```

```
MATCH (me)-[:KNOWS*1..2]-(remote_friend)
WHERE me.name = "Filipa"
RETURN remote_friend.name
```

Neo4j have some in-built function to compute path between two nodes with minimum number of edges (**shortestpath**). Additional filter predicates can be given with WHERE clause.

```
 MATCH (m { name:"Martin Sheen" }),(o { name:"Oliver Stone" }), p =
    shortestPath((m)-[*..15]-(o))
WHERE NONE(r IN rels(p) WHERE type(r)= "FATHER") RETURN p
```

```
MATCH (m { name:"Martin Sheen" }),(o { name:"Oliver Stone" }), p =
    shortestPath((m)-[*..15]-(o))
WHERE length(p) > 1 RETURN p
```

7.2.6 OPTIONAL MATCH-clause

Matches patterns against your graph database, just like MATCH

```
MATCH (a:Movie)
OPTIONAL MATCH (a)<-[:WROTE]-(x)
RETURN a.title, x.name
```

7.2.7 WHERE

The WHERE clause is not a clause in its own right, it is part of the MATCH, OPTIONAL MATCH, and WITH clauses. When used with MATCH and OPTIONAL MATCH, WHERE adds constraints to the patterns described. In the case of WITH, however, WHERE simply filters the results.

```
MATCH (n)
WHERE n.name = 'Peter' XOR (n.age < 30 AND n.name = 'Tobias')
    OR NOT (n.name ~= 'Tob.*' OR n.name CONTAINS 'ete')
RETURN n
```

7.2.8 WITH-clause

The WITH clause allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.

- Filter on aggregates

```
MATCH (p)-[:PLAYS]->(t) WITH t, AVG(p.age) AS a WHERE a < 25 RETURN t

```

- Aggregation of aggregates

```
MATCH (p)-[:PLAYS]->(t) WITH t, MIN(p.age) AS a RETURN AVG(a)

```

- Limit search space based on order of properties or aggregates

```
MATCH (p)-[f:FRIENDS]->(p2)
WITH f,p2 ORDER BYf.rating DESC LIMIT 5
MATCH (p2)-[f:FRIENDS]->(p3) RETURN DISTINCT p3

```

8. Gremlin

The Gremlin traversal language is the graph query language of the TinkerPop project. It supports both imperative and declarative syntax, but imperative syntax is the predominant approach and is used to build traversals step-by-step, where each step is a specific operation on the graph data. Instead declarative syntax is used to specify a pattern or condition to match, and the graph database will automatically find the relevant data that matches that pattern or condition. To sum up the imperative syntax is more flexible and allows for more fine-grained control over the traversal, while declarative syntax is often more concise and easier to read. The choice of which syntax to use depends on the specific use case and personal preference.

8.1 TinkerPop

TinkerPop is a top-level Apache Foundation project, which offers an open source and vendor-agnostic graph computing framework with both transactional (OLTP) and analytical (OLAP) capabilities. The goal of TinkerPop, as a Graph Computing Framework, is to make it easy for developers to create graph applications by providing APIs and tools that simplify their endeavors. **TinkerGraph** is an in-memory graph engine that supports both OLTP and OLAP workloads and is part of the TinkerPop Gremlin Server and Gremlin Console. It is a full-featured, open source implementation of TinkerPop. TinkerGraph is the core graph engine used in the various tools and software provided as part of TinkerPop. To start the engine we have the following steps:

- Get a Gremlin Server running and available to receive connections

```
$ cd apache-tinkerpop-gremlin-server-3.4.6
$ bin\gremlin-server.bat
```

- Connect a Gremlin Console to your Gremlin Server with a session

```
bin\gremlin.bat -i "file"
```

- Load test data into the serve

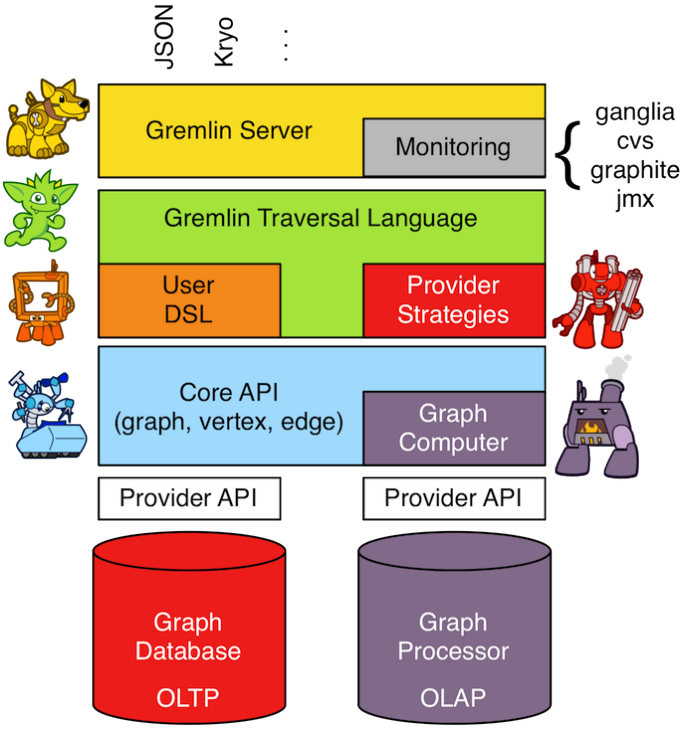


Figure 8.1: Tinkerpop Architecture

8.2 Traversing a graph

Before starting with the exercises we have to introduce some fundamental concepts:

Definition 8.2.1 — Traverse. The process of moving from vertex to edge or edge to vertex as we navigate through a graph

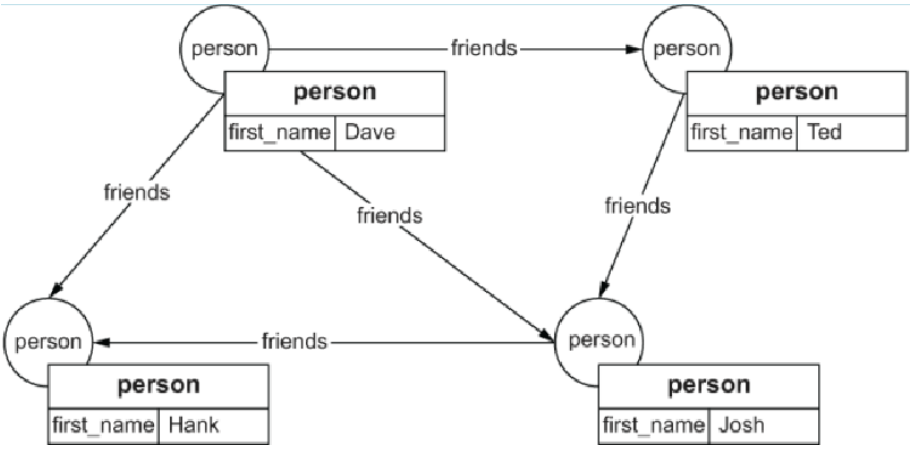
Definition 8.2.2 — Traversal. A specification of one or more steps or actions to perform on a graph, which either returns data or makes changes, or in some cases, does both. In the graph world, this is the set of operations, called steps, that are sent to the server to be executed.

Corollary 8.2.1 — Traversal source. The traversal source is a concept specific to TinkerPop. It represents the base or starting point from which steps traverse the graph. By convention, this is usually represented with the variable g and is required to begin any traversal.

Definition 8.2.3 — Traverser. The computing process associated with a specific branch of a traversal's execution. A unique traverser represents each branch through the data. For example to explore two outgoing edges of a node are needed two traverser, one for each edge.

To understand how traversing a graph consider the example shown in the figure below. What we need to do to answer the question, "Who are Ted's friends?"

1. The first step is to establish a starting point in the graph: we need to find the Ted vertex
2. Next step is to find Ted's friends. Looking at our graph, we notice a descriptively named friends edge connected to our Ted vertex. Let's traverse the edge between Ted and Josh *
3. Complete the traversing of the edge to the person vertex at the other end



The process of traversing a graph can be broken down into these basic operations:

- find a starting vertex (traversals require knowing our location in a graph)
- identify an edge to traverse
- traverse that edge
- complete the traversal by arriving at the destination vertex.

These steps can also include various operations for manipulating the graph data, such as filtering data, as well as the traversing of edges.

A metaphor of a graph can be a series of rooms connected by hallways. Imagine that you're a little Gremlin, sitting on a vertex. What we actually see through the eyes of the Gremlin is a closed room. Our perception is limited to what's visible within the room. As we look around the room, we observe the following:

- A chest of drawers with a label on each drawer. Each drawer is a vertex property, and the label is the property name.
- A series of doors, also with labels and with IN and OUT plaques. Each door is an incident edge, and the plaques represent the edge direction.
- The doors themselves also have drawers. These drawers are the edge properties.

In Gremlin we have the ability to control whether we traverse only incoming, outgoing, or both edge directions provides us with a powerful tool to customize our traversals.

From graph escape room mental model we can retrieve an important consequence: traversals don't have history. As we traverse the graph, we only have knowledge of where we currently are, not where we've previously been.

8.3 First Example

```
g.V().has('person', 'first_name', 'Ted').out('friends').
    values('first_name')
```

8.3.1 Traversal source

The g step is always the first step in every Gremlin traversal and it represents the traversal source for our graph and is the base on which all traversals are written.

8.3.2 Second step

The second step in our traversals is the V() step. The V() step returns an iterator that contains every vertex in the graph. It's one of two global graph steps. The other global graph step is E(), which returns an iterator that contains every edge in the graph.

```
g.V()
g.E()
```

8.3.3 Filtering

This is one of the most common Gremlin steps because it only passes through any vertex or edge that matches the label specified, if a label is specified or has a key-value pair that matches the specified key-value pair. The most commonly filter used are the following:

- hasLabel(label): yields all vertices or edges of the specified label type

```
g.V().has('person','first_name', 'Ted')
```

- has(key, value): yields all vertices and edges with a property matching the specified key and value
- has(label, key, value): yields all vertices and edges with both the specified label and with a property matching the specified key and value. The has() step, as with most of the Gremlin steps, can be chained together to perform more complex filtering operations.

```
g.V().hasLabel('person').has('first_name', 'Ted')
```

8.3.4 Traversal step

The **out(label)** step traverses all outgoing edges to the incident vertex with the specified label, if a label is provided. If a label isn't provided, then it just traverses all outgoing edges. This is one of the two most common traversal steps we use to navigate from one vertex to another. The other common traversal step is **in(label)**, which traverses all incoming edges to the incident vertex with the specified label, if a label is provided.
This flexibility to traverse relationships in either direction is a fundamental capability of graph databases. This directionality filters our traversals, which aids in both readability and performance, but it carries limitations as well. We might not know the direction of the edges we want to traverse, or we might not care in what direction we traverse.
What if we want to find two sets of people at the same time? To answer this, we traverse the friends edge in both the incoming and outgoing directions simultaneously. Let's introduce another Gremlin step: **both(label)**. This step traverses from a vertex to the adjacent vertex along edges with the given label.

8.3.5 Retrieving properties with values steps

The final step in our traversal is the **values(keys...)** step, which returns the values of the element's properties. A separate line displays each resulting value. This is one of several different ways to return the property values of an element in our graph. The other commonly used step is **valueMap(keys...)**, in which both the keys and values for the properties matchin

8.4 Recursive traversal

We use recursive traversals for problems where some portion of the traversal needs to be executed multiple times in succession. Which problems require recursive traversals? For example, if we want to know who are the friends of friends of Ted we have to use recursive logic. Without recursive traversal, we can accomplish the traversal in the following way.

```
g.V().has('person', 'first_name', 'Ted').out('friends').out('friends').
    values('first_name')
```

In recursive case, we should loop two times through the out('friends') step from our earlier traversal. To accomplish this in Gremlin, we need to introduce a few new steps:

- **repeat(traversal)**: repeatedly loops through the steps until instructed to stop.
- **times(integer)**: the integer parameter represents the number of operations for the loop to execute.

```
g.V().has('person', 'first_name', 'Ted').repeat(out('friends')).
    times(2).values('first_name')
```

But what if we don't know how many times we need to repeat our traversal to find Hank from Ted? Imagine we want to continue looping until we find an element that matches a specific set of criteria. For the situations where we don't know the number of times we need to recurse, we use the until() step. The until() step allows us to loop continuously until a specified condition is met. If the condition is never met, then it continues until it exhausts very potential path in the graph. This scenario is known as an **unbounded traversal**. When using the until() step, we recommend providing a maximum number of iterations using the times() step or using a time limit with the timeLimit() step. If the until() step comes before the repeat() step, then the loop operates as a while-do loop. If it appears after the repeat(), then it functions as a do-while loop.

```
g.V().has('person','first_name','Ted').repeat(out()).
    until(has('person','first_name','Dave')).values('first_name')
```

Figure 8.2: While loop

```
g.V().has('person', 'first_name','Ted').
    until(has('person','first_name', 'Hank')).repeat(out('friends')).values
    ('first_name')
```

Figure 8.3: Do-while loop

8.4.1 Intermediate steps

What if we want to see how two nodes are connected? To determine the intermediate steps, we need to introduce a modifier step to the repeat() step, known as emit(). The emit() step informs the repeat() step to emit the value at the current location in the loop.

```
g.V().has('person', 'first_name', 'Ted').until(has('person', 'first_name',
    'Hank')).
    repeat(out('friends')).emit().values('first_name')
```

The emit() step is similar to the until() step. If the emit() is placed before the repeat(), then it will include the starting vertex. If it's placed after the repeat(), then it will only emit the vertices traversed as part of the loop.

8.5 Mutating a graph

Mutating simply means changing the graph by adding, modifying, or deleting vertices, edges, and/or properties.

8.5.1 Creating vertices

We can think of the process for adding a vertex as:

- Given a traversal source g.
- Add a new vertex of type person with the method **addV(label)** that returns a reference to the newly added vertex.

- Add a property to that vertex.

```
g.addV('person').property('first_name', 'Dave')
```

8.5.2 Adding edges

The process for adding this edge is the following:

- Given a traversal source g.
- Add a new edge with a label friends with method **addE(label)**.
- Assign the outbound vertex of the edge to the vertex with the method **from(vertex)**.
- Assign the inbound vertex of the edge to the vertex with method **to(vertex)**.

```
g.addE('friends').
    from(V().has('person', 'first_name', 'Ted')).
    to(V().has('person', 'first_name', 'Hank'))
```

8.5.3 Removing data from the graph

To remove a vertex/edge the process is the following:

- Given a traversal source g
- Find the vertex/edge with a specific ID
- Remove (or drop) that vertex/edge.

```
g.V(15).drop()
g.E(13L).drop()
```

8.5.4 Updating a graph

What if you accidently misspelled "Dave" as "Dav" when adding a vertex? We can update the graph in the following way:

- Given a traversal source g
- Find the vertex with the error
- Update the property to that vertex

```
 g.V().has('person', 'first_name', 'Dav').property('first_name', 'Dave')
```

8.5.5 Variables

Variables are another source of variance within the graph world. To store value in variable we can use the following methods:

- **iterate()**: doesn't return a result.
- **next(id)**: returns the result of the traversal.

```
g.V().drop().iterate()
dave = g.addV('person').property('first_name', 'Dave').next()
```

8.5.6 Chaining mutations

In graph databases, mutations can be chained together to perform multiple changes simultaneously. It is possible to add all of the edges with a single traversal by chaining together multiple mutations.

```
g.addE('friends').from(dave).to(josh).
    addE('friends').from(dave).to(hank).
    addE('friends').from(josh).to(hank).
    addE('friends').from(ted).to(josh).iterate()
```

8.6 Path

Paths offer a description of the series of steps that a traverser takes to get from the start vertex to the end vertex. For example if we we want to find out which set of friends Ted needs to go through to get introduced to Denise and the path between these two nodes we have to use the method **path()** that returns the history of the vertices (and optionally edges) a specific traverser visits.

```
g.V().has('person', 'first_name', 'Ted').
    until(has('person', 'first_name', 'Denise')).
    repeat(both('friends')).path()
```

A common issue in graph are cycles. We can avoid that using simple path; it is a path that doesn't repeat any vertices, meaning that we only get results that are not cyclical.

```
g.V().has('person', 'first_name', 'Ted').until(has('person', 'first_name',
    'Denise')).
    repeat(both('friends').simplePath()).path()
```

Why do we add the simplePath() within the repeat() step instead of at the end? To see if we're in a cycle, we evaluate both our current position in the graph as well as our historical path through the graph at the end of each loop's iteration.

8.6.1 Traversing and filtering edge

To filter edge during the traversal we can use the following methods:

- **inE(label)**: traverses from the current vertex onto the incoming incident edges. If a label is specified, then filters to only traverse to edges of that type.
- **outE(label)**: traverses from the current vertex onto the outgoing incident edges. If a label is specified, then filters to only traverse to edges of that type.
- **bothE(label)**: traverses from the current vertex onto the incident edges, regardless of direction. If a label is specified, then filters to only traverse to edges of that type.

```
g.V().has('person','first_name','Dave').
    bothE('works_with').has('end_year',lte(2018)).
    otherV().values('first_name')
```

After outE(), inE(), bothE() how do we get back to the vertex? Gremlin provides companion V steps to accompany the E steps:

- **inV()**: traverses from the current edge to the incoming vertex. It's commonly paired with the outE() step.
- **outV()**: traverses from the current edge to the outgoing vertex. It's commonly paired with the inE() step.
- **otherV()**: traverses to the vertex that isn't the vertex that's used to traverse onto the edge. It's commonly paired with the bothE() step.
- **bothV()**: traverses from the current edge to both of the incident vertices. In the case of bothE(), we might be tempted to use bothV(), but that would be a mistake. If we used a bothV(), we would end up with two traversers: one on the start vertex and one on the end vertex.

8.6.2 Use cases

Why do we use the E and V steps (that Apache TinkerPop calls vertex steps) ? We find that there are three common use cases for the E and V steps:

1. **Filtering on edge properties using the has() filtering method**

```
g.V().has('person','first_name','Dave').
    bothE('works_with').has('end_year',lte(2018)).
    otherV().values('first_name')
```

2. **Including edges in path() results**

```
g.V().has('person', 'first_name', 'Ted').
    until(has('person', 'first_name', 'Denise')).
    repeat(bothE('works_with').otherV().simplePath()).path()
```

3. **Performant edge counts and denormalization**
   We emphasize that vertex properties, edges, and edge properties are essentially local to a vertex, so the cost to use these is basically free. But everything outside of this room can only be accessed by traversing an edge to get to another vertex. Due to this additional cost, when possible, don't traverse an edge to the other vertex.

```
g.V().bothE().count()
g.V().both().count()

```

   In the first case, we count the edge "doors" that we see from our vertex "room". Instead, in the second case, we go through each door and count the rooms (vertices) on the other side. Because there's a one-to-one correspondence between the doors (edges) and the other rooms counting the doors returns the same value as counting the rooms, but the second one costs much more.
   Denormalization in the graph is a matter of copying an often-accessed vertex property onto an adjacent edge. It avoids taking the cost of a full traversal when reading that property; it can be helpful for certain types of read-intensive activity (there's overhead in maintaining two copies of a property value).

8.7 Formatting results

To formatting results of a traversal query we can use two main methods:

- **values()**: returns the values of the properties

```
g.V().has('person','first_name', 'Ted').values()
```

- **valueMap()**: returns a map, which is a collection of key-value pairs of the specified properties

```
g.V().has('person','first_name', 'Ted').valueMap()
```

In a graph database, only the values of the current vertices or edges are retrieved. For example if we consider the following query:

```
g.V().hasLabel('order').out('contains')
```

It returns information about products but we aren't able to understand at which orders are related. How to retrieve orders' value? We can solve using **alias**.
An alias in a graph database is a labeled reference to a specific output of a step, either a vertex or an edge, that can be referenced by later steps.

```
g.V().hasLabel('order').as('o').out('contains').as('p')
```

To lock back in the traversal and retrieve values we can use the following methods:

- **select(string[])**: selects aliased elements from earlier in the traversal. This step always looks back to previous steps in the traversal to find the aliases.
- **by(key)** or **by(values(key))**: specifies the key of the property to return the value from the corresponding aliased element.
- **by(traversal)**: specifies the traversal to perform on the corresponding aliased element.

Each aliased element we specify in a select() statement should have a corresponding by() statement to indicate the operations to perform on it.

```
g.V().hasLabel('order').as('o').out('contains').as('p')
    .select('o', 'p').by('number').by('name')
```

8.7.1 Projecting results

Sometimes, instead of looking back in the traversal for earlier results, it is preferable to project results forward from the current elements.
Projection is the process of working with vertices, properties, or additional traversal expressions to create results from the input to the current step. Projection always moves forward, taking the incoming data as the starting point. It is generally used to group or aggregate data starting from the current location in the graph. In our order-processing graph, let's answer "For each of the products in order ABC123, how many times has that product been ordered?"

```
g.V().has('order','number','ABC123').out('contains')
    .project('name','c').by('name').by(inE().count())
```

8.7.2 Selection vs Projection

Selection uses the select() step to create a result set based on previously traversed elements of a graph. To use the select() step, we alias each of the elements with the as() step for later use. Instead projection uses the project() step to branch from the current location within the graph and creates new objects. In our present example, we had one element remain static, the person's name, but we needed the other elements to be calculated through further traversing of the graph to return the number of friends.

8.7.3 Manipulate results

Graph databases don't guarantee the order of the results by default. This leads us to the three most common requirements for organizing our result data:

- **Ordering the results**: using **order()** we collect all objects up to this point of the traversal into a list, which is ordered according to the accompanying by() modulator

```
g.V().hasLabel('person').values('first_name').
    order().by() // or by(desc) or by(shuffle)

```

- **Grouping the results**: **group()**: groups the results based on the specified by() modulator. Data is grouped by using either one or two by() modulators. The first one specifies the keys for the grouping. The second one, if present, specifies the values. If not present, the incoming data is collected as a list of the values associated with the grouping key.

```
g.V().has('person', 'first_name', 'Dave').both().both().
    group().by('first_name').by(count()).unfold()

//using groupCount()
g.V().has('person', 'first_name', 'Dave').both().both().
    groupCount().by('first_name').unfold()
```

- **Limiting the size of the results**
  1. limit(number): returns the first number of results

```
g.V().hasLabel('person').values('first_name').order().
    by().limit(3)
```

  2. tail(number): returns the last number of results

```
g.V().hasLabel('person').values('first_name').order().by().tail(3)
```

  3. range(startNumber, endNumber): returns the results from startNumber (inclusive, zero-based) to endNumber (not inclusive).

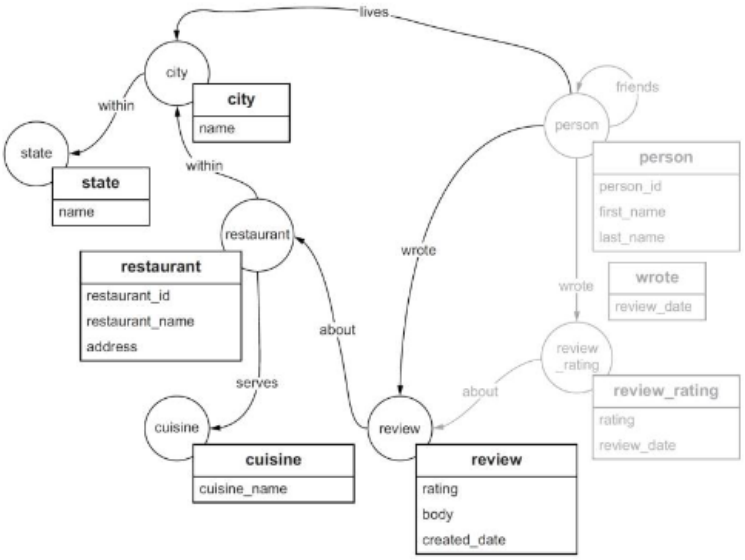
```
g.V().hasLabel('person').values('first_name').order().
    by().range(2,4)
```

8.8 Advanced Graph Traversal

To building traversals using known walks we need to identify the required elements:

1. Examine each requirement and break it into the components needed to answer the question.
2. Identify the required vertex labels.
3. Identify the required edge labels.

Take DiningByFriends as example and try to answer to question "What restaurant near me with a specific cuisine is the highest rated?" In order to do that we have to break problem in small parts:



- Locates the restaurants in a geographic area "near me"
- Filters by cuisine to find a specific cuisine
- Calculates the average rating for each restaurant to find the highest rated

To identify the vertices we have to look at our data model, and see which vertex labels provide the information needed to satisfy requirements:

- restaurant: the core piece of information to return
- city: to find restaurants "near me"
- cuisine: to filter by cuisine
- review: to calculate the average rating of a restaurant
- person: to find a restaurant "near me"

Instead to identify the edges is very simple because, for our vertex labels (restaurant, city, cuisine, review, person), we are limited in the edges we can use. We could just use the edge labels connecting those vertex labels.

8.8.1 Selecting a starting place

A crucial decision in traversal is where do we begin our development work? Two reasonable approaches:

1. One approach is to pick what we think is the most challenging problem and start there. This approach allows us to fail fast and is the right choice when a quick decision of some sort is needed—perhaps to make a "go/no-go" decision or to determine whether the technology is the right choice for the problem.
2. Another approach is to start with the most straightforward or the least complicated question and use it as a building block for the rest of our work. This path allows for the progressive development of the code base. The idea here is to get a quick win with a smaller, simpler problem before tackling more complex issues.

The second approach is the most common and are usually accomplished following a certain process:

1. Identify the vertex labels and edge labels required to answer the question.
2. Find the starting location for the traversal.
3. Find the end location for the traversal.
4. Write out the steps in plain English (or in your preferred language).
5. Code each step, one at a time, with Gremlin and verify the results against the test data after each step

For example if we consider the question "What are the newest reviews for this restaurant?" the steps to follow are the following:

1. Get the restaurant based on the restaurantid.

```
g.V().has('restaurant','restaurant_id',rid)
```

2. Traverse the about edges to the review vertices

```
g.V().has('restaurant','restaurant_id',rid).in('about').valueMap('
    created_date','body')
```

3. Sort review vertices by created date in descending order

```
g.V().has('restaurant','restaurant_id',rid).in('about').order().
    by('created_date',desc).valueMap('created_date', 'body')
```

4. Limit the results to the first three returned

```
g.V().has('restaurant','restaurant_id',rid).in('about').order().
    by('created_date',desc).limit(3).valueMap('created_date', 'body')
```

III Graph Powered ML

9. Machine Learning and Graph

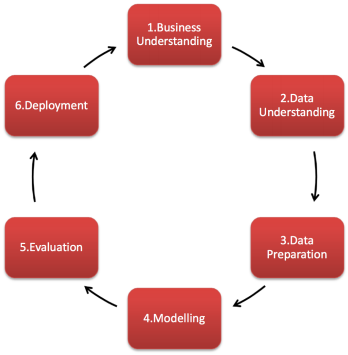
9.1 ML project life cycle

Delivering ML solutions is a complex process that requires more than selecting the right algorithm(s):

- Selecting the data sources and gathering data
- Understanding the data
- Cleaning and transforming the data
- Processing the data to create ML models
- Evaluating the results
- Deploying

After deployment, it is necessary to monitor the application and tune it. The entire process involves multiple tools, a lot of data, and different people.

One of the most commonly used processes for data mining and ML projects is the Cross-Industry Standard Process for Data Mining, or **CRISP-DM**. Key features of this process that make it attractive as part of the base workflow model are that it isn't proprietary, it is application, industry, and tool neutral and it explicitly views the data analytics process from both an application-focused and a technical perspective. As shown in the figure above the CRISP-DM process is composed of



the following phases:

- **Business Understanding**: The first stage of the CRISP-DM process is to understand what you want to accomplish from a business perspective:
  - Define requirements, assumptions and constraints
  - Understand business and domain
  - Set objectives
  - Produce raw version of the project plan
  - List the risks or events that might delay the project or cause it to fail.
- **Data Understanding**: The second stage of the CRISP-DM process requires you to acquire the data listed in the project resources:
  - Identify data sources
  - Verify data quality
  - Describe and explore data to understand relationships and subsets
- **Data Preparation**: This is the stage of the project where you decide on the data that you're going to use for analysis
  - Gather data from multiple sources
  - Merge and clean data
  - Identify the DBMS
  - Organize data in algorithm-specific structures
- **Modelling**:
  - Select and apply different algorithms
  - Tune algorithm parameters to find optimal values
  - Build a range of prediction models
- **Evaluation**:During this step you'll assesses the degree to which the model meets your business objectives and seek to determine if there is some business reason why this model is deficient:
  - Evaluate model using testing data
  - Check if the business objectives are satisfied
  - Define performance measures for future development
  - Get authorization to move into production
- **Deployment**: In the deployment stage you'll take your evaluation results and determine a strategy for their deployment:
  - Define deployment plan and build production environment
  - Monitor performance
  - Define maintenance plan
  - Produce and pubblish documentation

9.1.1 The first big challenge: Data

In every ML project, the first big challenge to overcome is related to data:

- **Insufficient quantity of data**: ML requires a lot of training data to work properly. Even for simple use cases, it needs thousands of examples, and for complex problems such as deep learning or for nonlinear algorithms, you may need millions of examples.
- **Poor quality of data**: Data sources are always full of errors, outliers, and noise. Poor data quality directly affects the quality of the results of the ML process because it is hard for many algorithms to discard wrong (incorrect, unrelated, or irrelevant) values and to then detect underlying patterns.
- **Non representative data**: ML is a process of induction where the model makes inferences from what it has observed and will likely not support cases that your training data does not include. If the training data is too noisy or is related only to a subset of possible cases, the

learner might generate bias or overfit the training data and will not be able to generalize to all the possible cases.
- **Irrelevant features**: The algorithm will learn in the right way if the data contains a good set of relevant features and not too many irrelevant features. Feature selection and feature extraction represent two important tasks to avoid overfitting.
- **Managing big data**: Gathering data from multiple data sources and merging it into a unified source of truth will generate a huge dataset not always easy to use.
- **Designing a flexible schema**: Try to create a schema model that provides the capabilities to merge multiple heterogeneous schemas into a unified and homogeneous data structure that satisfies the informational and navigational requirements.
- **Developing efficient access patterns**: Fast data reads boost the performance, in terms of processing time, of the training process.

9.1.2 Performance

Another complex topic, in ML, is performance. Performance are very difficult to achieve because can be related to multiple factors:
- **Predictive accuracy**, which can be evaluated by using different performance measures. Data plays a primary role in guaranteeing a proper level of accuracy.
- **Training performance**, which refers to the time required to compute the model. The amount of data to be processed and the type of algorithm used determine the processing time and the storage required for computing the prediction model.
- **Prediction performance**, which refers to the response time required to provide predictions. The output of the machine learning project could be a static one-time report to help managers make strategic decisions or an online service for end users. In some case prediction speed does matter, because it affects the user experience and the efficacy of the prediction.

In this context, graphs could provide the proper storage mechanism for both source and model data, reducing the access time required to read data as well as offering multiple algorithmic techniques for improving the accuracy of the predictions.

9.1.3 Store the model

Another aspect to consider is how storing the model. In the model-based learner approach, the output of the training phase is a model that will be used for making predictions. This model requires time to be computed and has to be stored in a persistence layer to avoid recomputation at every system restart. We have two main approcess to this problem:
- **Item-to-item similarities** for a recommendation engine that uses a nearest-neighbor approach
- An **item-cluster mapping** that expresses how the elements are grouped in clusters

The sizes of the two models differ enormously. Consider a system that contains 100 items. The item-to-item similarities would require 100 x 100 entries to be stored. Taking advantage of optimizations, this number can be reduced to consider only the top k similar items so we have 100 x k entries. Instead, an item-cluster mapping requires only 100 entries.
Hence, the space required to store the model in memory or on disk could be huge or relatively modest. For these reasons, model storage management represents a significant challenge in machine learning.

9.1.4 Another challenge: Time

Machine learning is being used more and more frequently to deliver real-time services to use recommendation system. For example, a simple recommendation engines that react to the user's last clicks or self-driving cars that have been instructed not to injure a pedestrian crossing the street. In all these system the goals are the following:

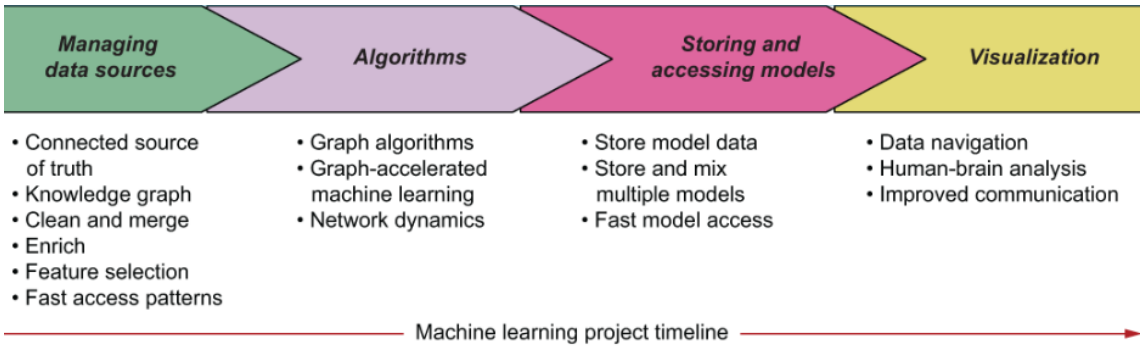
- **Learn fast**: The online learner should be able to update the model as soon as new data is available. This capability will reduce the time gap between events or generic feedback; the more the model is aligned to the latest events, the more it is able to meet the current needs of the user.
- **Predict fast**: When the model is updated, the prediction should be fast as possible because the user may navigate away from the current page or even change their opinion quite rapidly.

9.2 The role of graphs in machine learning

Building a graph-powered machine learning platform has numerous benefits, because graphs can be valuable tools not only for overcoming the previously described challenges, but also for delivering more advanced features that are impossible to implement without graph support. Graph features are grouped into three main areas:

1. **Data management**: This area contains the features provided by graphs that help machine learning projects deal with the data.
2. **Data analysis**: This area contains graph features and algorithms useful for learning and predicting.
3. **Data visualization**: This area highlights the utility of graphs as a visual tool that helps people communicate, interact with data, and discover insights by using the human brain.

The role of graphs depends on each specific case. Not in all machine learning projects we use graphs for all data management, data analysis and visualization. We can use graphs in different ways in each of the four stages of a machine learning workflow:



9.2.1 Data management

Graphs allow the learning system to explore more of the data, to access it faster, to clean and enrich it easily.

Graph-powered data management features include:

- **Connected sources of truth**: graphs allow to merge multiple data sources into a single connected data set ready for the training phase. It reduces data sparsity, increases the amount of data available and simplify data integration.
- **Knowledge graphs**: provide a homogeneous data structure for combining not only data sources, but also prediction models, manually provided data, and external sources of knowledge.
- **Fast data access**: provide a single access pattern related to row and column filters. Graphs provide multiple access points to the same set of data. This feature improves performance by reducing the amount of data to be accessed for a specific set of needs.

- **Data enrichment**: make it easy, thanks to schemaless nature, to extend existing data with external sources.
- **Feature selection**: by providing fast access to data and multiple query patterns, graphs speed feature identification and extraction.

9.2.2 Data analysis

Graphs can be used to model and analyze the relationships between entities as well as their properties. The schema flexibility provided by graphs also allows different models to coexist in the same dataset. Graph-powered data analysis features include:

- **Graph algorithms**: several types of graph algorithms, such as clustering, page ranking, and link analysis algorithms.
- **Graph-accelerated machine learning**: the graph-powered feature extraction is an example of how graphs can speed or improve the quality of the learning system. Graphs can help in filtering, cleaning, enriching, and merging data before or during training phases.
- **Network dynamics**: awareness of the surrounding contexts and related forces that act on networks allows not only to understand network dynamics, but also to use them to improve the quality of the predictions.
- **Mixing models**: multiple models can coexist in the same graph, taking advantage of flexible and fast access patterns.
- **Fast model access**: real-time use requires fast predictions, which implies a model that can be accessed as fast as possible. Graphs provide the right patterns for these scopes.

9.2.3 Data visualization

Graphs have high communication power, and they can display multiple types of information at the same time in a way the human brain can easily understand. This feature is greatly important in a machine learning project for sharing results, analyzing the model and for helping people navigate the data. Graph-powered data visualization features include:

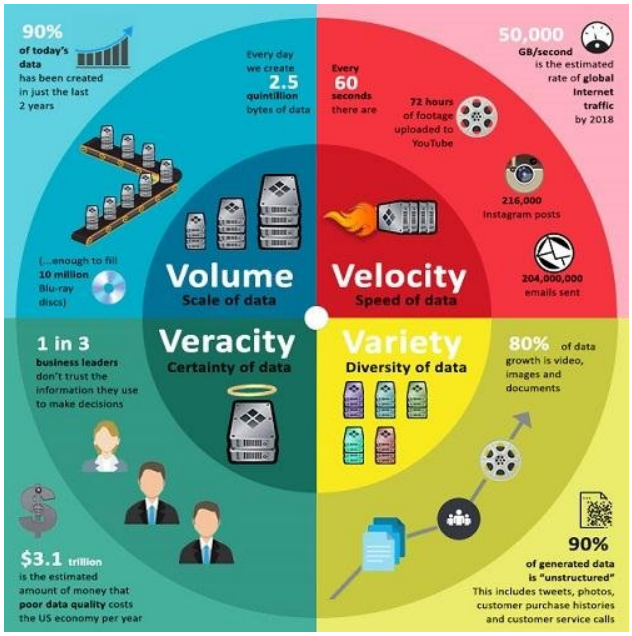
- **Data navigation**: networks are useful for displaying data by highlighting connections between elements.
- **Human-brain analysis**: displaying data in the form of a graph unleashes the power of machine learning by combining it with the power of the human brain enabling efficient, data processing and pattern recognition.
- **Improved communication**: graphs are **"whiteboard friendly"** which means they are conceptually represented on a board as they are stored in the database. This feature reduces the gap between the technicalities of a complex model and the way in which it is communicated to the domain experts or stakeholders.

10. Graph Data Engineering

What are the main issues working with Big Data?

- Collect and gathered data from multiple data sources.
- Store data in a proper way to make easy-to-access and ready for the next phases.
- The data should be merged, cleaned, and (whenever possible) normalized by using a unified and homogeneous schema.
- Multiple views or access patterns should be provided to simplify and speed access to the dataset that will be used for training purposes.

10.1 The four V's of Big Data



10.1.1 Volume

The solutions to this challenge fall into two main categories:

- Scalable storage: scaling storage generally refers to adding more machines and distributing the load (reads, writes, or both) over them. This process is known as scaling horizontally. Also query or access mechanisms that provide multiple access points to a subset of the full data store, without the need to go over the entire dataset by using filters or index lookups.
- Scalable processing: the horizontal scaling of processing doesn't only mean having multiple machines executing tasks in parallel; it also requires a distributed approach to querying, a protocol for effective communication over the network, orchestration, monitoring, and a specific paradigm for distributed processing.

S A graph-based model enables data from multiple data sources to be stored in a single, highly connected, and homogeneous source of truth that offers multiple fast access patterns. Specifically, in a big data platform, graphs can help address volume issues by playing two roles:

1. **Main data source**: in this case, the graph contains all data with the lowest granularity. The learning algorithms access the graph directly to perform their analyses (indexing structure to support random access and and an access pattern for accessing only a small portion of the graph.
2. **Materialized views**: in this case, the graph represents a subset of the main dataset or an aggregated version of data in it, and is useful for analysis, visualization, or results communication.

10.1.2 Velocity

Velocity refers to how rapidly data is generated, accumulated, or processed. Some applications have strict time constraints for data analysis, including stock trading, online fraud detection, and real-time applications generally. For example, in self-driving car the system should be able not only to process this data at speed, but also to generate a prediction as fast as possible to avoid hitting a pedestrian crossing the street.

10.1.3 Variety

Variety has to do with the different types and nature of the data that is analyze. Because it is collected from different and varied sources, big data comes in multiple shapes (structured, unstructured, or semistructured),size and formats and rarely does it present itself in a form that's perfectly ordered and ready for processing. Hence, any big data platform needs to be flexible enough to handle such variety, especially considering the data's potentially unpredictable evolution.
Graph databases with the simple model based on nodes and edges provides great flexibility in terms of data representation. Furthermore, new types of nodes and edges can appear later in the design process without affecting the previously defined model, giving graphs a high level of extensibility.

10.1.4 Veracity

Veracity is related to the quality and/or trustworthiness of the data collected. Data-driven applications can reap the benefits of big data only when the data is correct, meaningful, and accurate. Data rarely comes wholly accurate and complete, which is why a cleaning process is necessary. This task can be accomplished with graph approaches. Specifically, graph access patterns make it easy to spot issues based on the relationships among elements. Moreover, by combining multiple sources in a single connected source of truth, it is possible to merge information, reducing data sparsity.
The approaches that deal with these challenges can be grouped into two categories:

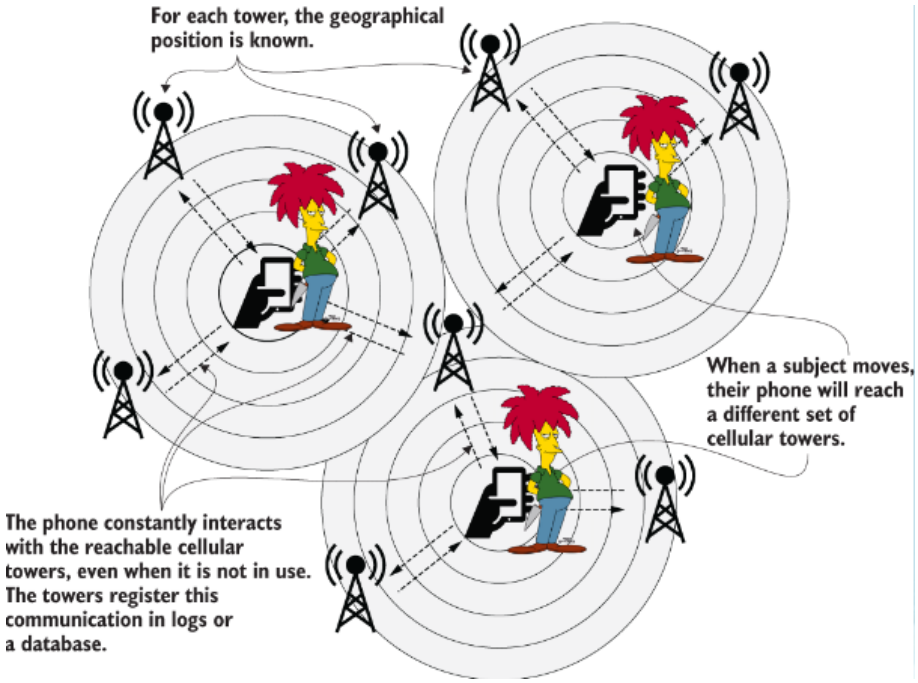
- Methodological approaches include all the design decisions that involve the architecture, algorithms, storage schema, and cleaning methods

- Technological approaches include the design aspects related to the DBMS to use, the cluster configuration to adopt, and the reliability of the solution to deliver

To get the best from both, you should use them in combination, harmonically.

10.2 Graphs for Big Data

To understand better the usage of graphs in Big data, we take the following use case as example: "You are a police officer. How can you track a suspect by using cellular tower data collected from the continuous monitoring of signals every phone sends to (or receives from) all towers it can reach?"
The goal of our example scenario is to use such monitoring data to create a predictive model that identifies location clusters relevant for the subject's life and that predicts and anticipates subsequent movements according to the subject's current location.

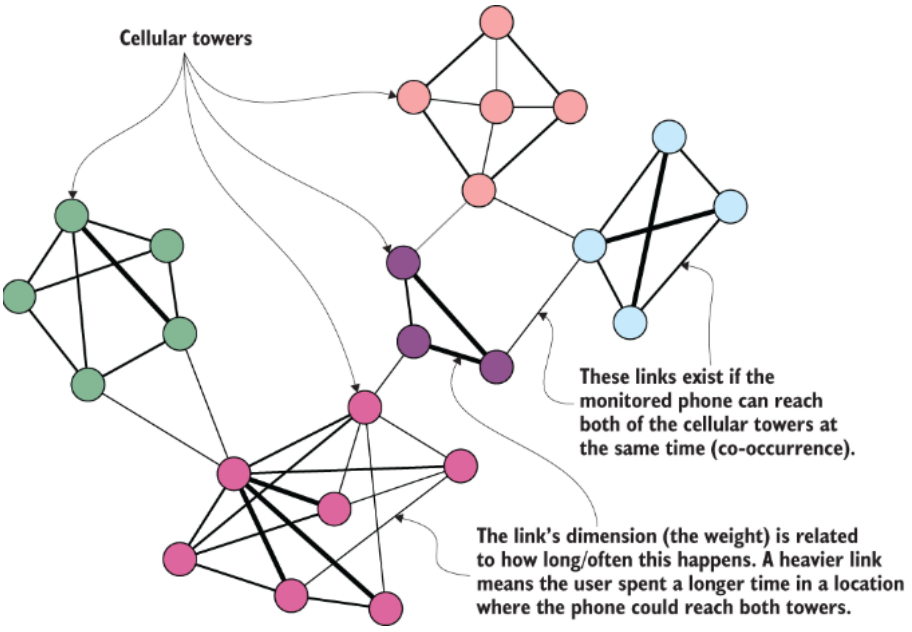


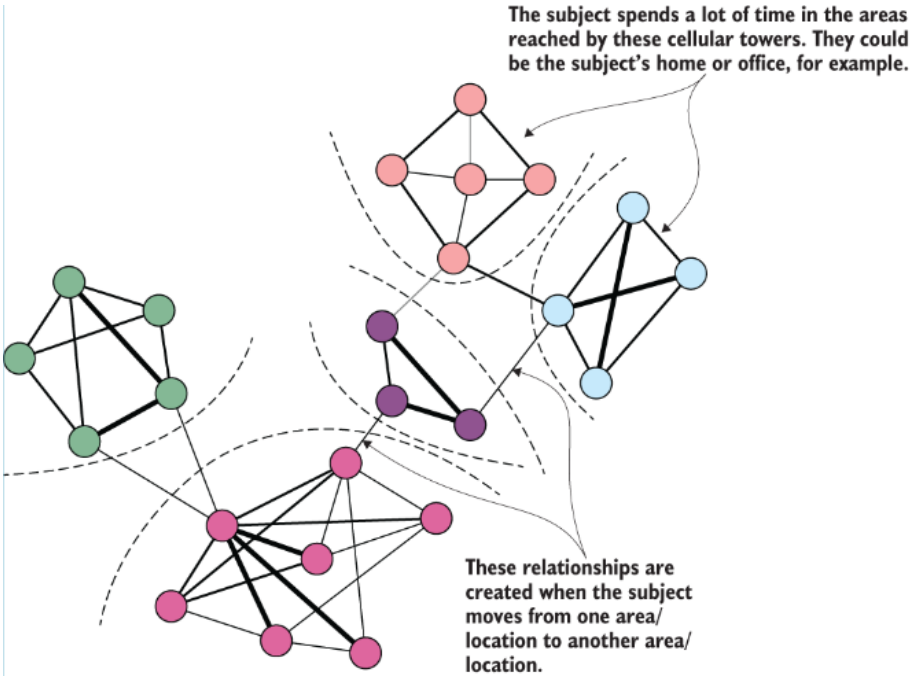
We will use a graph as a way to collapse and organize the data available from cellular towers and creates a graph-based materialized view of the subject's movements. Then the resulting graph is analyzed with a graph algorithm that identifies clusters of positions and build the position prediction model. Every mobile phone in use today has continuous access to information about the nearby cellular towers, so studying these data streams can provide valuable insight into a user's movements and behavior. To build graph-based materialized view of the subject's movement we have to go through the following phases:

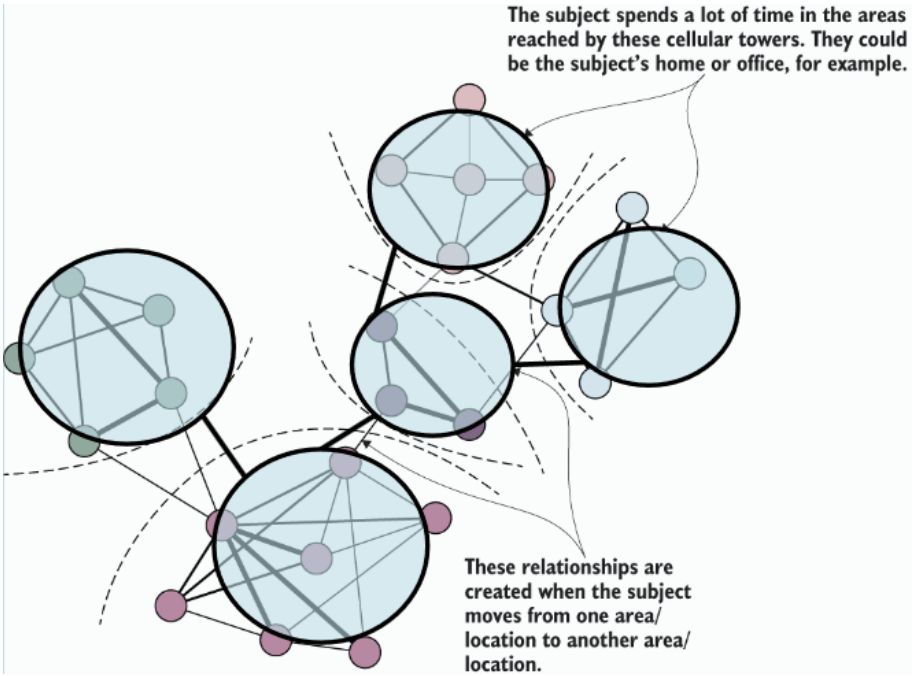
1. Each subject's phone records the four nearest towers (the towers with the strongest signal) at 30-second intervals. Once collected, this data can be represented as a **cellular tower network (CTN)**, in which the nodes are unique cellular towers and an edge exists between each pair of nodes that co-occur(phone can reach both of the cellular towers at the same time) in the same record, and each edge is weighted according to the total amount of time the pair co-occurred over all records (a CTN is generated for each subject).
2. A node's strength is determined by summing the weights of all its edges. The nodes with the highest total edge weights identify the towers that are most often close to the subject's phone. Groups of highly weighted nodes, therefore, should correspond to locations where the subject

spends a significant amount of time. Building on this idea, the graph can be segmented into clusters by means of different clustering algorithms.

3. The clusters of towers identified can be converted to the states of a dynamic model. Given a sequence of locations visited by a subject, it is possible to learn patterns in their behavior and calculate the probability that they will move to various locations in the future. When the prediction model is computed, it can be used to make a prediction about the future movements of a subject.





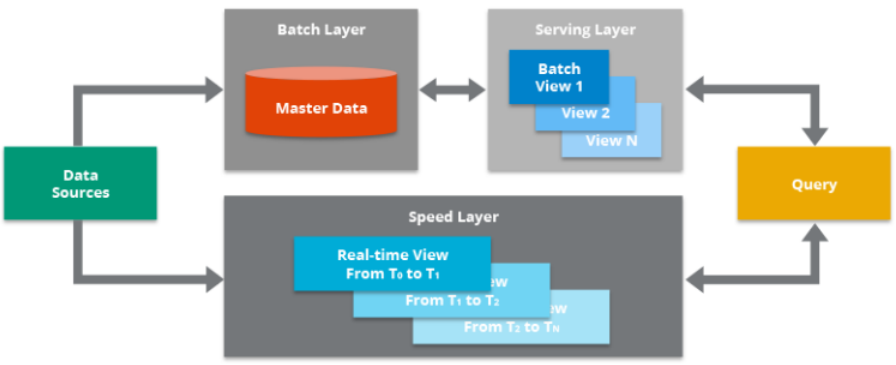


There are several problems to overcome:

- There is a lot of data in the form of events. The events logged by the phones or the cellular towers are raw, immutable, and true so it is necessary to store them one time.
- The data is distributed across multiple data sources. Hence, the data needs to be aggregated and organized in a form that simplifies further processes and analysis.
- From the first aggregation format, multiple views to manage are created.
- The view-building process generally operates on the entire set of data, and this process can take time, especially when it operates on a large amount of data. The time required to process the data creates a gap between the view of current events and previous events
- To have a real-time view of the data, it is necessary to fill this gap. The real-time view requires a kind of streaming process that reads the events and appends information to the views.

10.2.1 Lambda Architecture

The architectural problems can be addressed by the Lambda Architecture which conceives the big data system as a series of three layers: batch,serving, and speed. The Lambda Architecture is a

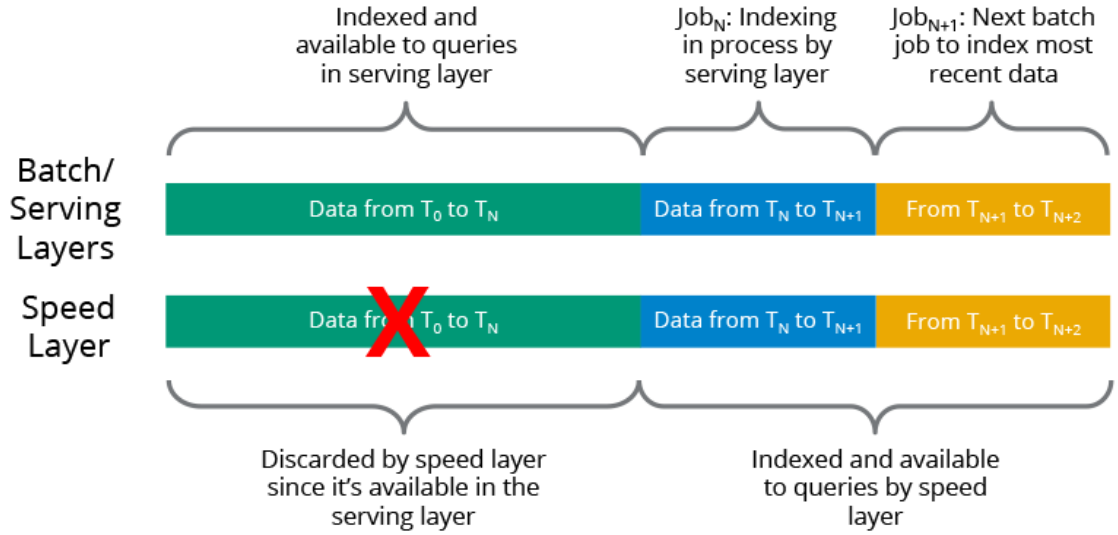


deployment model for data processing that organizations use to combine a traditional batch pipeline

with a fast real-time stream pipeline for data access. In the diagram above, you can see the main components of the Lambda Architecture:

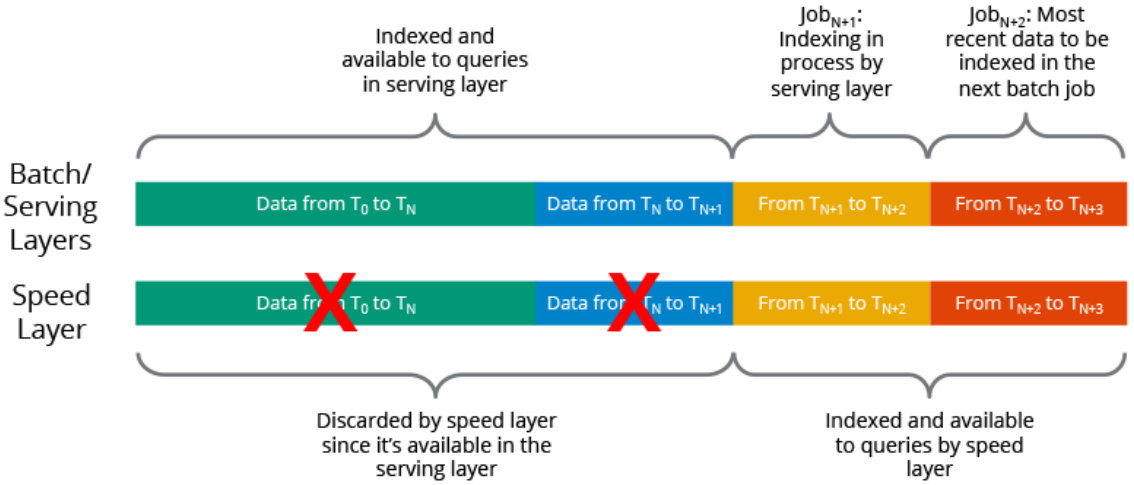
- **Data Sources.** Data can be obtained from a variety of sources, which can then be included in the Lambda Architecture for analysis. This component is oftentimes a streaming source which is not the original data source but is an intermediary store that can hold data in order to serve both the batch layer and the speed layer of the Lambda Architecture to enable a parallel indexing effort.
- **Batch Layer**. This component saves all data coming into the system as batch views in preparation for indexing. The input data is saved in a model that looks like a series of changes/updates that were made to a system of record. Oftentimes this is simply a file in the comma-separated values (CSV) format. The data is treated as immutable and append-only to ensure a trusted historical record of all incoming data.
- **Serving Layer**. This layer incrementally indexes the latest batch views to make it queryable by end users. This layer can also reindex all data to fix a coding bug or to create different indexes for different use cases. The processing is done in an extremely parallelized way to minimize the time to index the data set.
- **Speed Layer**. This layer complements the serving layer by indexing the most recently added data not yet fully indexed by the serving layer.
- **Query**. This component is responsible for submitting end user queries to both the serving layer and the speed layer and consolidating the results. This gives end users a complete query on all data, including the most recently added data, to provide a near real-time analytics system.

How does the Lambda Architecture work? The batch/serving layers continue to index incoming data in batches. Since the batch indexing takes time, the speed layer complements the batch/serving layers by indexing all the new, unindexed data in near real-time. This gives you a large and consistent view of data in the batch/serving layers that can be recreated at any time, along with a smaller index that contains the most recent data.



Once a batch indexing job completes, the newly batch-indexed data is available for querying, so the speed layer's copy of the same data/indexes is no longer needed and is therefore deleted from the speed layer. The serving layer then begins indexing the latest data in the system that had not yet been indexed by this layer, which has already been indexed by the speed layer (so it is available for

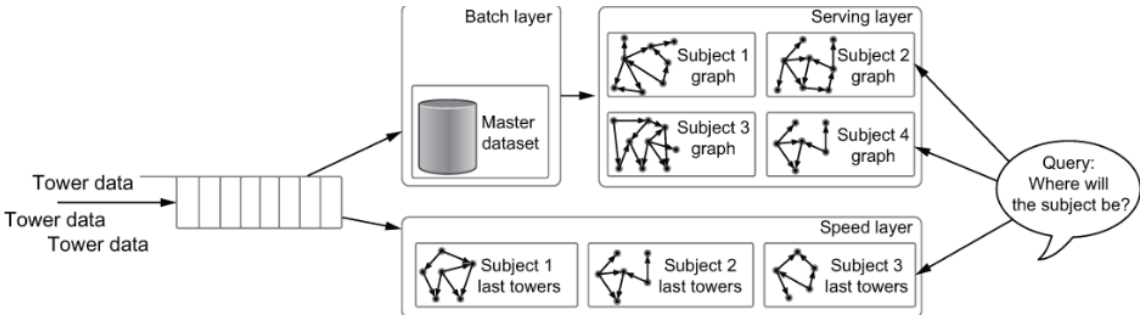
querying at the speed layer). This ongoing hand-off between the speed layer and the batch/serving layers ensures that all data is ready for querying and that the latency for data availability is low.



Advantages of Lambda Architecture are the following:

- Parallelization helps to reduce the latency that is inherent in the batch/serving layers.
- One key idea behind the Lambda Architecture is that it eliminates the risk of data inconsistency that is often seen in distributed systems. Since the data is processed sequentially (and not in parallel with overlap, which may be the case for operations on a distributed database), the indexing process can ensure the data reflects the latest state in both the batch and speed layers.
- The Lambda Architecture does not specify the exact technologies to use, but is based on distributed, scale-out technologies that can be expanded by simply adding more nodes.
- As above, the Lambda Architecture is based on distributed systems that support fault tolerance, so should a hardware failure occur, other nodes are available to continue the workload.
- Since raw data is saved for indexing, it acts as a system of record for your analyzable data, and all indexes can be recreated from this data set. This means that if there are any bugs in the indexing code or any omissions, the code can be updated and then rerun to reindex all data.

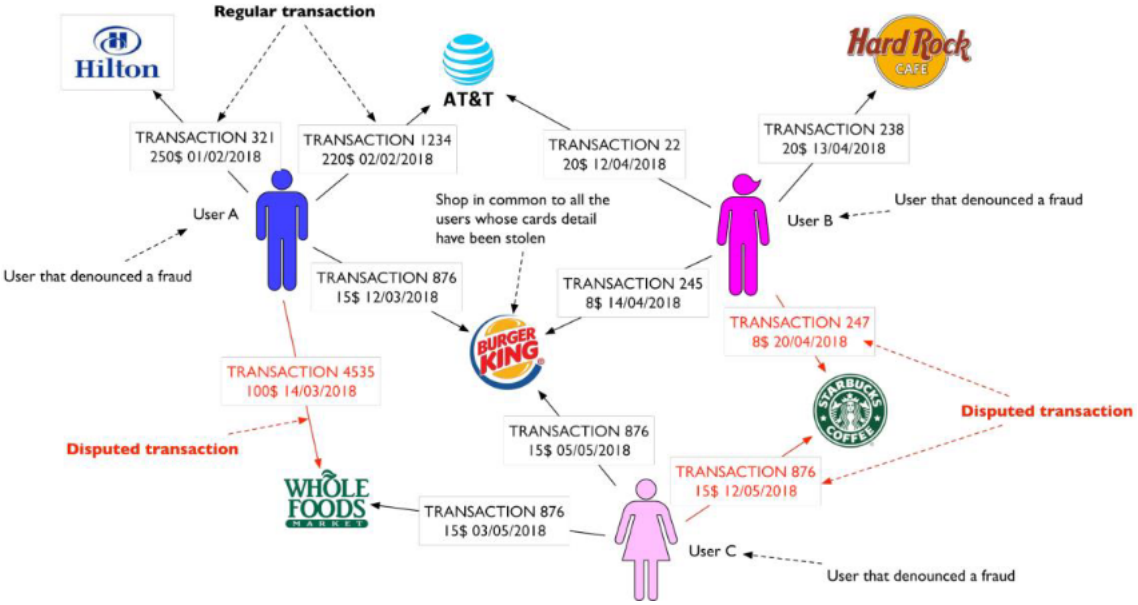
In the cellular tower scenario, a first batch view is the CTN, which is a graph. The function that operates on such a view is the graph cluster algorithm, which produces another view: the clustered network. In the CTN scenario, both the batch views and the real-time views are modeled as graphs. This graph representation not only allows us to reply faster to queries, but also facilitates analysis.



The Lambda Architecture has sometimes been criticized as being overly complex. Each pipeline requires its own code base, and the code bases must be kept in sync to ensure consistent, accurate results when queries touch both pipelines.

10.3 Graphs for master data management

Other types of analysis cannot be performed on an aggregate version of the data. Such algorithms require more detailed information to be effective; they need access to the fine-grained version of the data to accomplish their job. This type of analysis can also use the graph model as a way of representing connections and exploiting insights from the data. To understand it consider the following use case: "You would like to create a simple but effective fraud detection platform for banks, in particular for credit card theft"
Starting from this transaction dataset, let's define a graph model. Each transaction is an edge involves two nodes: a person (the customer or user) and a merchant. Furthermore, each transaction has a date and a status: undisputed for legitimate transactions and disputed for reported fraudulent transactions.



Starting from the graph, the analysis steps to spot the source of fraud are the following:

- Filter the fraudulent transactions. Identify the people and the cards involved in the attack.
- Spot the point of origin of the fraud. Search for all the transactions before the beginning of the fraud.
- Isolate the thieves. Identify some common pattern, such as a merchant in common, that could be the origin of the fraud.

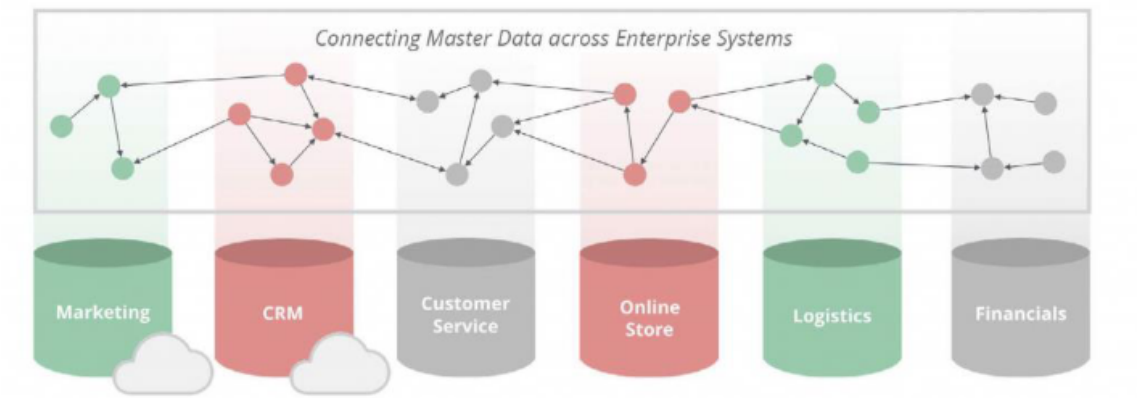
In this scenario, the graph is used to store the single source of truth on which analysis is performed by using graph queries. Furthermore, the data can be visualized in the form of a graph for further analysis and investigation. The advantages brought by use of graphs are the following:

- Multiple data sources (GPS information, social network data, user personal profiles, family data) can be merged in a single connected source of truth.
- Existing data can be easily extended with external sources of knowledge.
- The same data model can support several analysis techniques
- Data can be visualized as a graph to speed the manual analysis.

- The structure simplifies the merging and cleaning operation, thanks to the flexible access pattern provided by the graph model.

The graph plays the role of master dataset and is the foundation for **master data management (MDM)** that is the practice of identifying, cleaning, storing, and governing data. The key concerns of MDM include:

- Managing changes over time as organizational structures change, businesses merge, and business rules evolve
- Incorporating new sources of data
- Supplementing existing data with external data sources
- Addressing the needs of reporting, compliance, and business intelligence consumers
- Versioning data as its values and schema changes



MDM is not an alternative or modern version of data warehousing (DW), although the two practices have a lot in common. DW relates to the storage of historical data, whereas MDM deals with current data. An MDM solution contains the current and complete information for all business entities within a company.

1. **Different Goals**: DW analyze data in a multidimensional fashion, otherwise MDM create and maintain a single source of truth for a particular dimension within the organization.
2. **Different Types of Data**: MDM is applied to entities and not transactional data, instead DW includes data that are both transactional and non-transaction
3. **Different Reporting Needs**: DW deliver to end users the proper types of reports using the proper type of reporting, meanwhile MDM far more important to be able to provide reports on data governance, data quality, and compliance, rather than reports based on analytical needs.
4. **Where Data Is Used**: in DW, applications access the data warehouse directly and original data sources are not affected. In MDM, we often need to have a strategy to get a copy of the master data back to the source system.

MDM has numerous advantages:

- Streamlining data sharing among personnel and departments
- Facilitating computing in multiple system architectures, platforms, and applications
- Removing inconsistencies and duplications from data
- Reducing unnecessary frustration when searching for information
- Simplifying business procedures
- Improving communication throughout the organization

Using graphs combined with MDM we can unlock other several advantages:

- **Flexibility**: the data captured can be easily changed to include additional attributes and

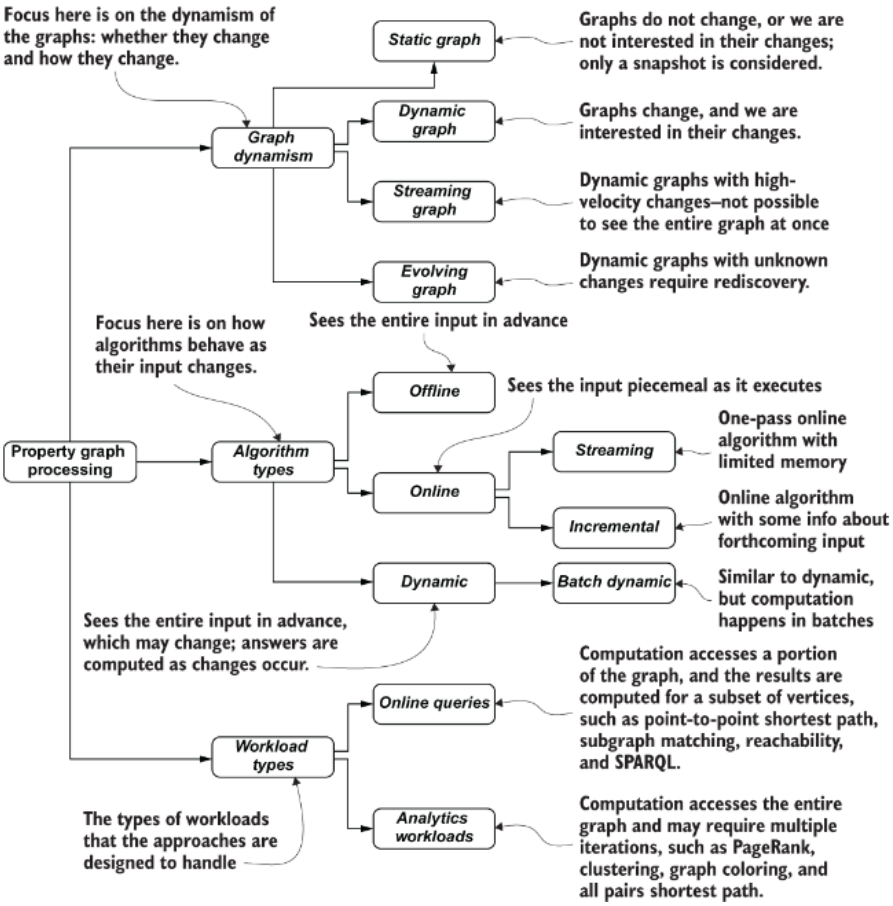
objects.

- **Extensibility**: the model allows the rapid evolution of the master data model in line with changing business needs.
- **Search capability**: each node, each relationship, and all their related properties are search entry points
- **Indexing capability**: graph databases are naturally indexed by both relationships and nodes, providing faster access compared to relational data.

10.4 Graphs data management

Technological aspects related to graph management are relevant in a machine learning project's life cycle, during which it is necessary to manipulate, store, and access real data. Furthermore, working with big data, scalability is a crucial aspect. The graph database management is a technical approach to manage the following phases:

- **Modeling**: same aspect of reality can be mapped in multiple ways in a graph model and "schemaless" nature of graphs can affect performance.
- **Storage**: persistent data storage, memory use, caching, physical/logical data independence, query languages, data integrity and consistency.
- **Processing**: frameworks for processing and analysis of graphs.



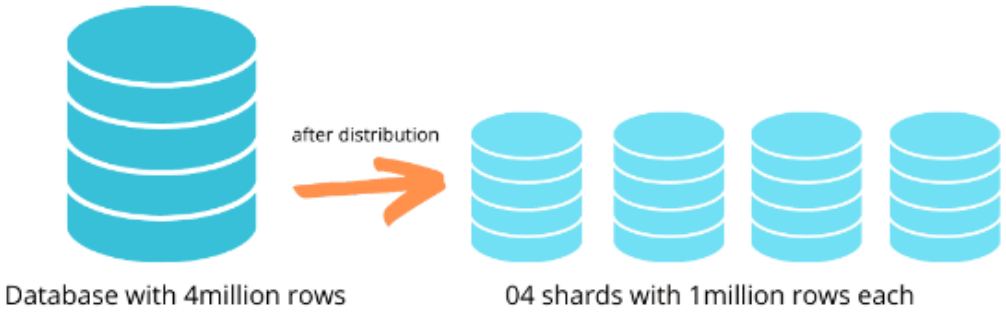
To enable scalability we have several option:

- Sharding
- Replication

- Native and non native graph DB

10.4.1 Sharding

Looking at big data applications from a pure data storage perspective, the main Vs challenges are the volume of the data involved and the velocity to perform. A possible solution can be **shard** the dataset that means that a large dataset is split, and subsets are distributed across several shards on different servers. We have different sharding strategy that determines which data partitions should be sent to which shards. A possible strategy can be to co-locate related nodes and, hence,



Sharding example

the related edges, boosting graph traversal performance. But having too many connected nodes on the same database shard would make it heavily loaded, because a lot of data will be on the same shard, making it unbalanced. Furthermore cross-shard traversals are quite expensive because it requires many network hops, resulting in substantially increased query times. In such a scenario, performance degrades quickly compared with the case in which everything happens on the same shard.

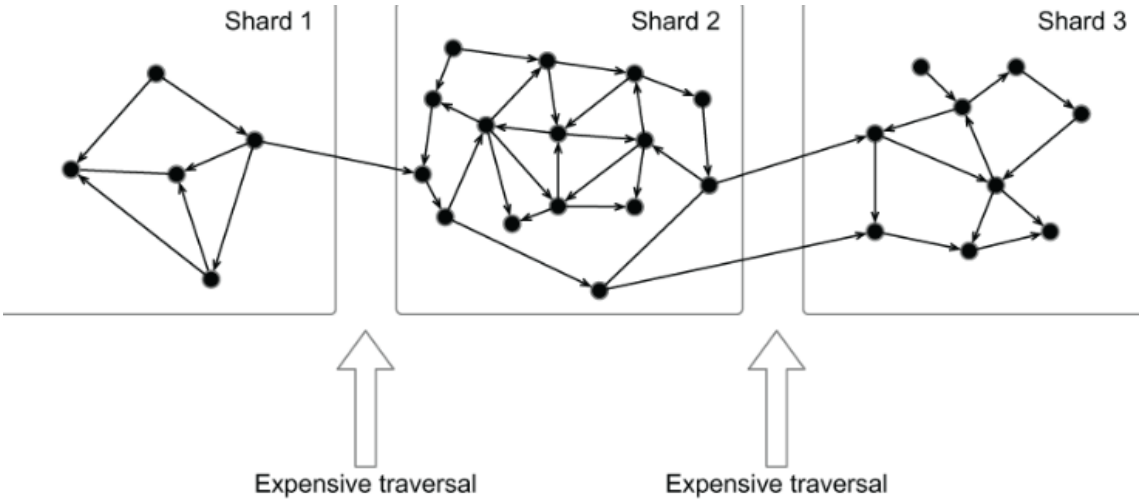


Figure 10.1: Caption

Other techniques are possible for scaling a graph database:

1. **Application-level sharding**
2. **Increasing the RAM or using cache sharding**
3. **Replication**

Application-level sharding

Sharding of the data is accomplished on the application side by using domain-specific knowledge. Each shard contains all the data required to execute the algorithm, and some nodes can be replicated across the shards.

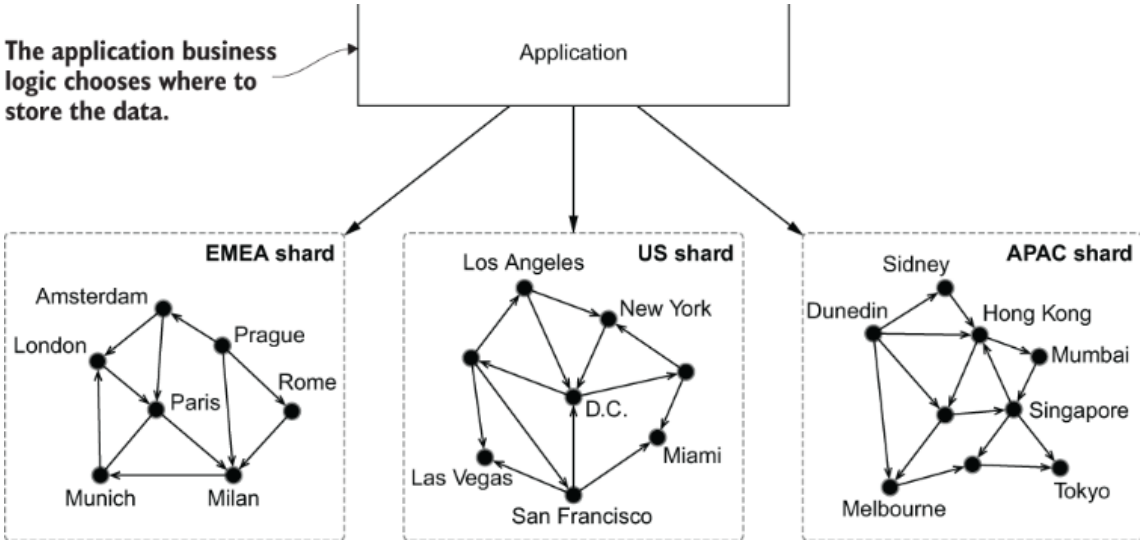


Figure 10.2: Sharding based on country; very efficient because inter-sharding traversal are not used

Increasing the RAM or using cache sharding

It is possible to scale the server vertically, adding more RAM so that the entire database fits in memory. This solution makes graph traversal extremely fast but is both unreasonable and unfeasible for large databases. For these datasets a common technique is **cache sharding** that maintain high performance with a dataset whose size far exceeds the main memory space. To implement cache sharing, we partition the workload undertaken by each database instance to increase the likelihood of hitting a warm cache for a given request.

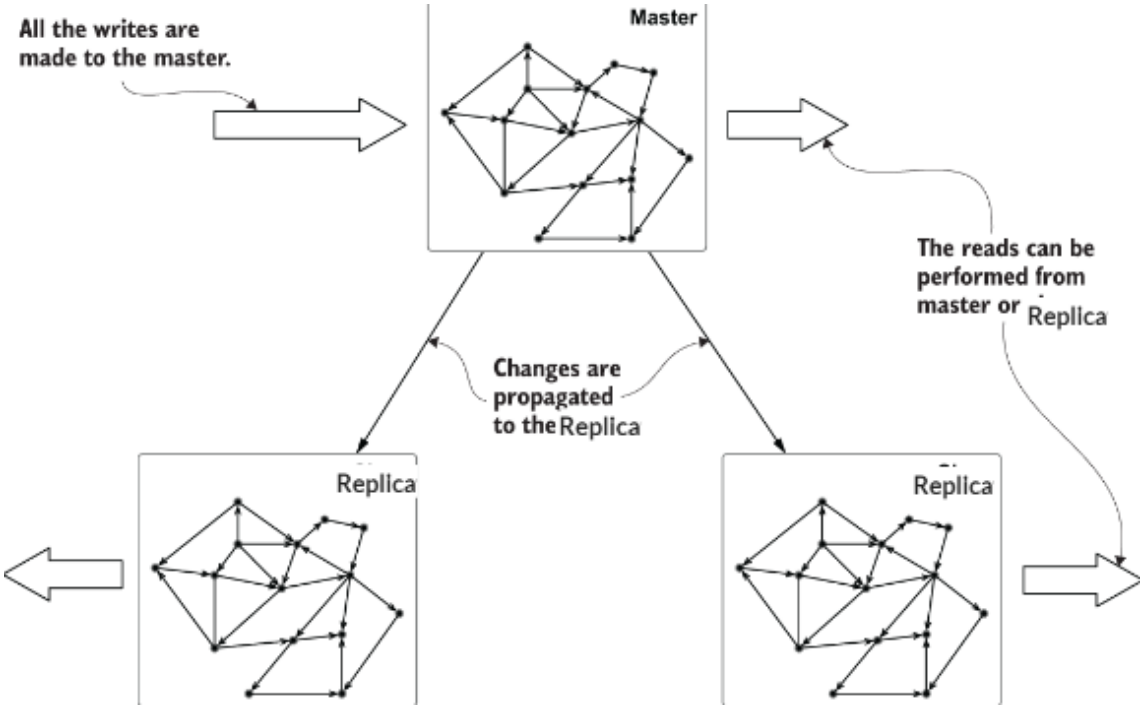
10.4.2 Replication

It is possible to achieve scaling of the database by adding more (identical) copies of the database that act as followers with read-only access. Replication has several purposes:

- System availability: Replication removes single points of failure from distributed DBMSs by making data items accessible from multiple sites.
- Performance: Replication enables us to reduce latency by locating the data closer to its access points.
- Scalability: Replication allows systems to grow, both geographically and in terms of the number of access requests, while maintaining acceptable response times.
- Application requirements: As part of their operational specifications, applications may require multiple copies of the data to be maintained.

On the other hand, keeping the different copies synchronized is a challenge. We have two main techniques to manage it:

1. **Centralized** if they first perform updates on a master copy. Centralized techniques can be further identified as single master when there is only one master database copy for all data items in the system or primary copy when there can be a single master copy for each data item or set of data items.
2. **Distributed** if they allow updates to any replica.



Which are the use cases for this technique? In the cellular tower monitoring example, a graph is created for each monitored subject, so the machine learning model produces multiple independent graphs that will be accessed in isolation. In this case, application-level sharding is an easy task, because all the graphs are isolated. Instead in the fraud detection use case, sharding would be tricky because in theory, all the nodes can be connected. Some heuristics can be applied to reduce cross-shard traversals or to keep nodes that are frequently accessed together on the same shard, but the graph cannot be divided into multiple isolated graphs, as in the preceding use case. In such cases, another option is to use replication to scale read performance and speed analysis time.

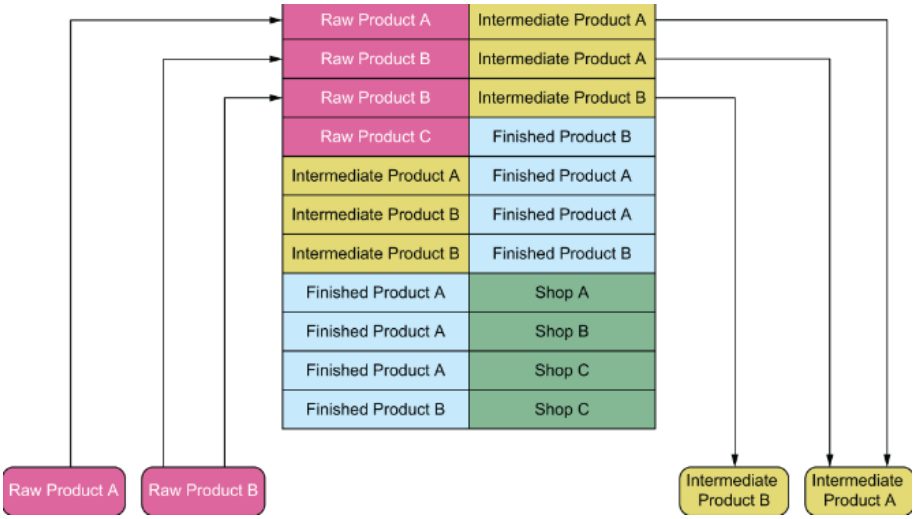
subsectionNative vs. non-native graph databases There are numerous ways to encode and represent graphs in the different database engines.

- **Native graph databases**: designed to use the filesystem in a way that not only understands but also supports graphs, which means that they are both highly performant and safe for graph workloads. In more detail, a native graph DBMS exhibits a property called index-free adjacency, which means that each node maintains direct references to its adjacent nodes (as an adjacency list)
- **Non-native graph database**: is optimized for an alternative storage model, such as columnar, relational, document, or key/value data, so when dealing with graphs, the DBMS has to perform costly translations to and from the primary model of the database. Implementers can try to optimize these translations through radical denormalization, but this approach typically leads to high latency when querying graphs. In other words, a nonnative graph database will realistically never be as performant as a native graph database, for the simple reason that a translation process will need to occur. Non-native are useful for hybrid graphs where data are not all stored as nodes and edges. Non-native graph databases can be divided into two categories:
  1. Those that layer a graph API on top of an existing different data structure, such as key/value, relational, document, or column-based store.
  2. Those that claim multimodel semantics, in which one system can support several data

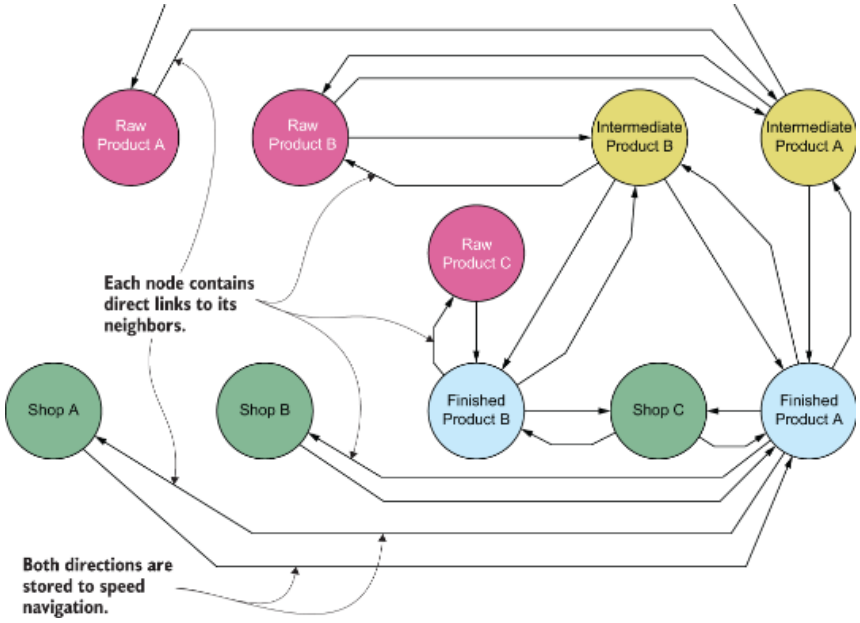
models

To understand better the differences consider the following example: "You have to implement a supply chain management system that analyzes the entire chain to predict stock inventory issues in the future or spot bottlenecks in the network". We have several option to model this use case:

- **A tabular model for supply chain network**.
  These indexes add a layer of indirection to each traversal, thereby incurring greater computational cost. To find where Finished Product B will be delivered after it is produced, we first have to perform an index lookup, at cost O(log n), and then get the list of next nodes in the chain. This approach may be acceptable for occasional or shallow lookups, but it quickly becomes intolerably expensive when we reverse the direction of the traversal.



- **A graph-based model for storing the supply chain**.
  In this representation, the cost of traversing a relationship when you have the first node is O(1), and it points directly to the next node. Performing the same traversal required before now costs only O(m). Not only is the graph engine faster, but also, the cost is related only to the number of hops (m), not to the total number of relationships (n).



HOW CAN WE IDENTIFY BOTTLENECKS IN THE SUPPLY CHAIN?
One common method for spotting bottlenecks in a network is betweenness centrality, which is a measure of centrality (importance) in a graph based on calculating the shortest paths between nodes.

Advantage of native graph architecture are the following:

- Native graph databases handle connected data queries far faster than nonnative graph databases.
- Native graph databases can deliver constant-time traversals with index-free adjacency without complex schema design and query optimizations.
- To improve performance in a nonnative graph, it is possible to denormalize indexes or create new indexes, or a combination of both, affecting the amount of space required to store the same amount of information.
- Index denormalization also has an effect on write performance because all those additional index structures need to be updated as well.

Label property graphs

Graph database management system providers introduced the label property graph model to tie a set of attributes to graph structures (nodes and relationships) and add classes or types to nodes and relationships. This data model allows a more complex set of query features typical of any DBMS, such a s projection, filtering, grouping, and counting.

Definition 10.4.1 — Label property graph. A label property graph is defined as "a directed, vertex-labeled, edge-labeled multigraph with self-edges, where edges have their own identity." In other words, in a property graph, we use node to denote a vertex and relationship to denote an edge.

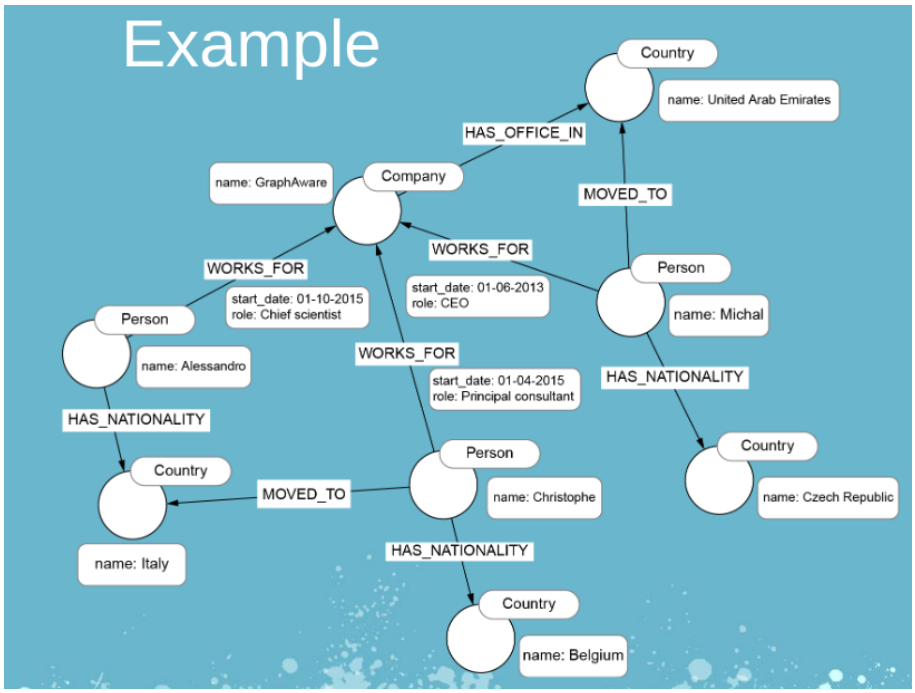


Figure 10.3: Communication capability is greater than before

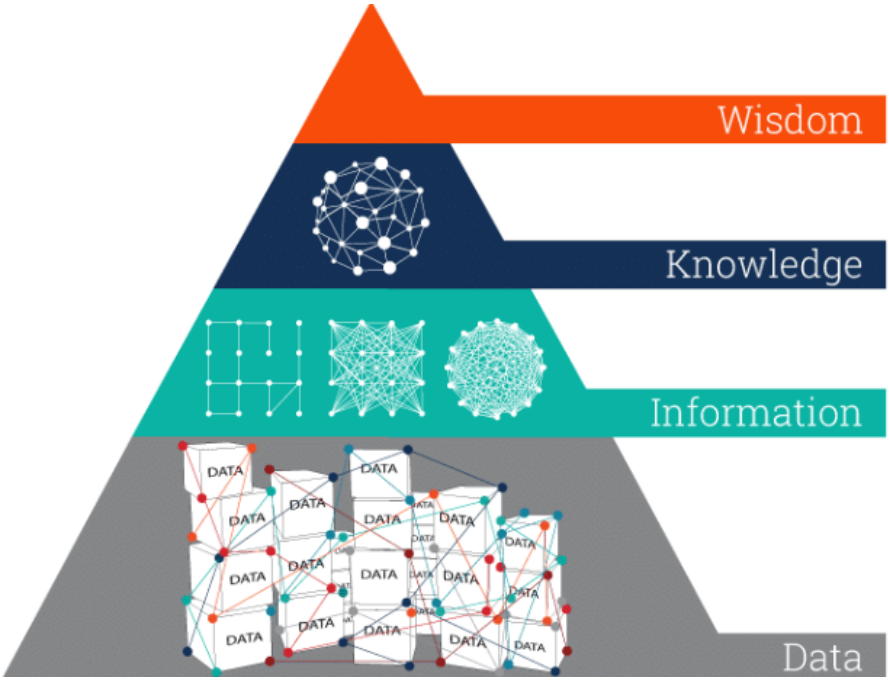
More in detail, a property graph has the following properties:

- The graph consists of a set of entities. An entity represents either a node or a relationship.

- Each entity has an identifier that uniquely identifies it across the entire graph.
- Each relationship has a direction, a name that identifies the type of the relations hip, a start node, and an end node.
- An entity can have a set of properties, which are typically represented as key/value pairs.
- Nodes can be tagged with one or more labels, which group nodes and indicate the roles they play within the dataset.

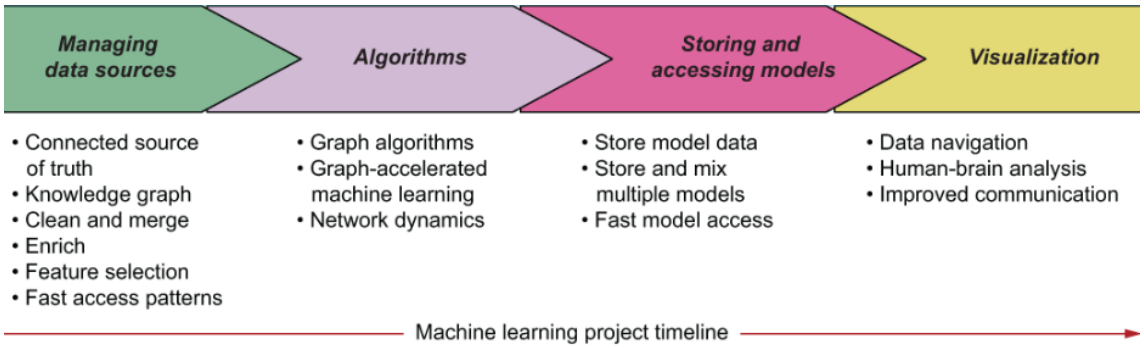
11. Graphs in ML applications

11.1 Learning path



The Data-Information-Knowledge-Wisdom (DIKW) pyramid illustrates the progression of raw data to valuable insights. It gives you a framework to discuss the level of meaning and utility within data. Each building block is a step towards an higher level and each step answers different questions about the initial data and adds value to it.
In graph data and information are gathered from one source or several sources. This data is the training data, on top of which any learning will happen, and it is managed in the form of a graph When the data is organized in the form of knowledge and represented by a proper graph, machine

learning algorithms can extract and build insights and wisdom on top of it. The prediction models that are created as the result of the training of a machine learning algorithm on the knowledge are stored back in the graph, making the wisdom inferred permanent and usable. Finally, the visualization shows the data in a way that the human brain can easily understand, making the derived knowledge, insights, and wisdom accessible.



11.1.1 Managing data sources

The first step is to create a graph model from the original data. The transformation from the original data shape to a graph could be done through multiple techniques that can be classified in two groups:

- **Graph modeling** where data is converted to some graph representation by means of a modeling pattern.
  1. Identify the data sources available for algorithm training purposes
  2. Analyze the data available and evaluate the content, in terms of quality and quantity.
  3. Design the graph data model, identifing the meaningful information to be extracted from the data sources and considering the data available, access patterns, and extensibility.
  4. Define the data flow
  5. Import data into the graph
  6. Perform pre and post processing tasks (data cleaning, data enrichment and data merging).
- **Graph construction** where a new graph is created, starting from the data available. The resulting graph contains more information than before.

For example "HOW CAN WE REPRESENT CREDIT CARD TRANSACTIONS IN A GRAPH?

1. As a directed edge between the two entities involved in the transaction.
2. As a node that contains all the relevant information about the event and is connected via edges to the related nodes.

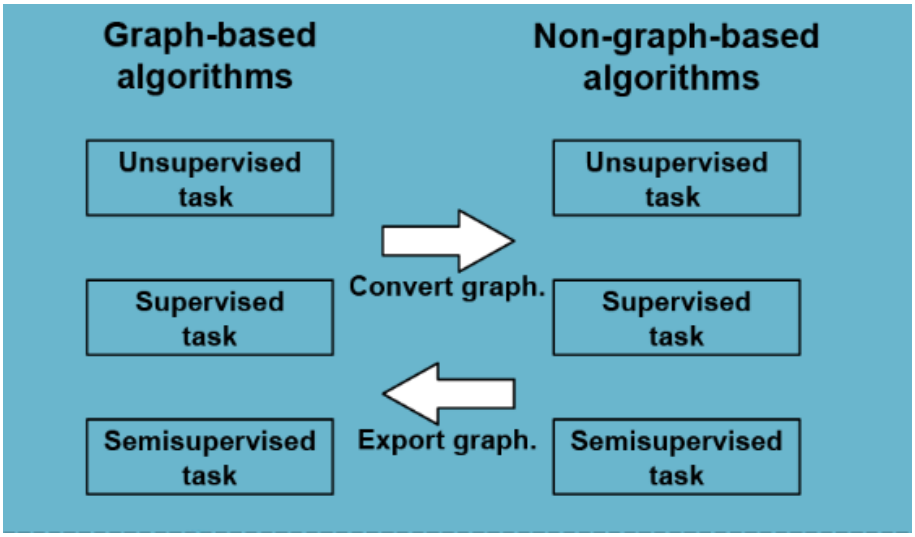
11.1.2 Algorithms

After storing graph in a proper way we can apply, as shown in the figure below, various algorithms,to it to extract main information.

Famous algorithms that belong to graph machine algorithms are the following:

- **PageRank**: This algorithm works by counting the number and quality of edges to a node to arrive at a rough estimate of the node's importance.
- **Betweenness centrality**: This algorithm measures the importance of a node by considering how often it lies on the shortest paths between other nodes

The graph representation is helpful for the following tasks:



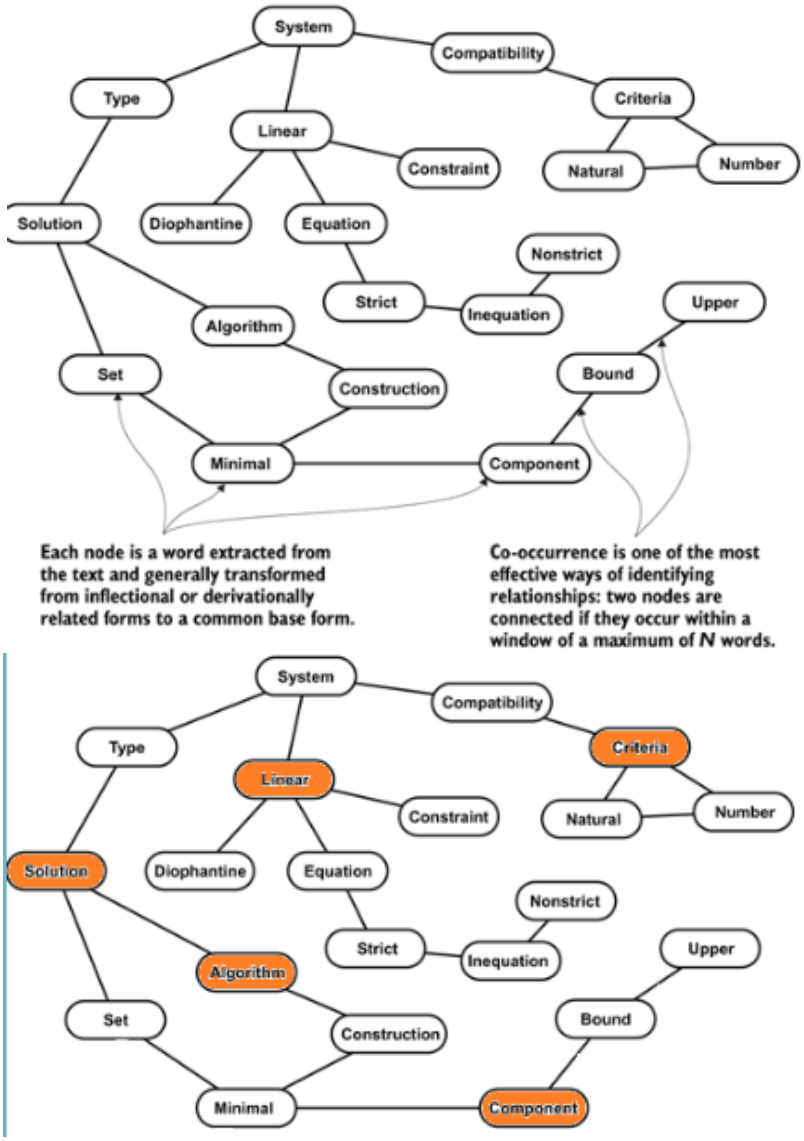
- **Feature selection**: A graph is easy to query and can merge data from multiple sources, so finding and extracting the list of variables to use for training is made simpler by the graph approach.
- **Data filtering**: The easy-to-navigate relationships among objects make it easy to filter out useless data.
- **Data preparation** Graphs make it easy to clean the data, removing spurious entries, and to merge data from multiple sources.
- **Data enrichment**: Extending the data with external sources of knowledge
- **Data formatting**: Export the data in whichever format is necessary.

Other fields where graph combined with machine learning algorithms are useful are the following:

- **Keyword extraction** can also be used to build an automatic index for a document collection, to construct domain-specific dictionaries, or to perform text classification or summarization. The simplest approach is to use a relative-frequency criterion to select the important keywords in a document (this method lacks sophistication and typically leads to poor results). Instead using supervised learning methods can train an accurate model.Graphs provide a mechanism to extract keywords or sentences from the text in an unsupervised manner by using a graph representation of the data and a graph algorithm like **TextRank** (graph-based ranking model).
  1. Identify text units relevant to the task at hand, and add them to the graph as nodes.
  2. Identify relations that connect the text units.
  3. Iterate the graph-based ranking algorithm until convergence or until the maximum number of iterations is reached.
  4. Sort the nodes based on their final scores, use these scores for ranking/selection decisions, and eventually merge two or more text units in a single (phrase) keyword.
- **Monitor a subject** thanks to graph clustering algorithms that is an unsupervised learning method that aims to group the nodes of the graph in clusters. When the graph is organized into multiple subgraphs that identify locations, the next step is using this information to build a predictive model that is able to indicate where the subject is likely to go next, based on their current position. The clusters of towers identified previously can be incorporated as states of a dynamic model.

11.1.3 Models storage

The output of the learning phase is a model that contains the result of the inference process and allows us to make predictions about unseen instances. The model has to be stored in permanent



storage or in memory so that it can be accessed whenever a new prediction is required. Depend on use case we have different method to store the model:

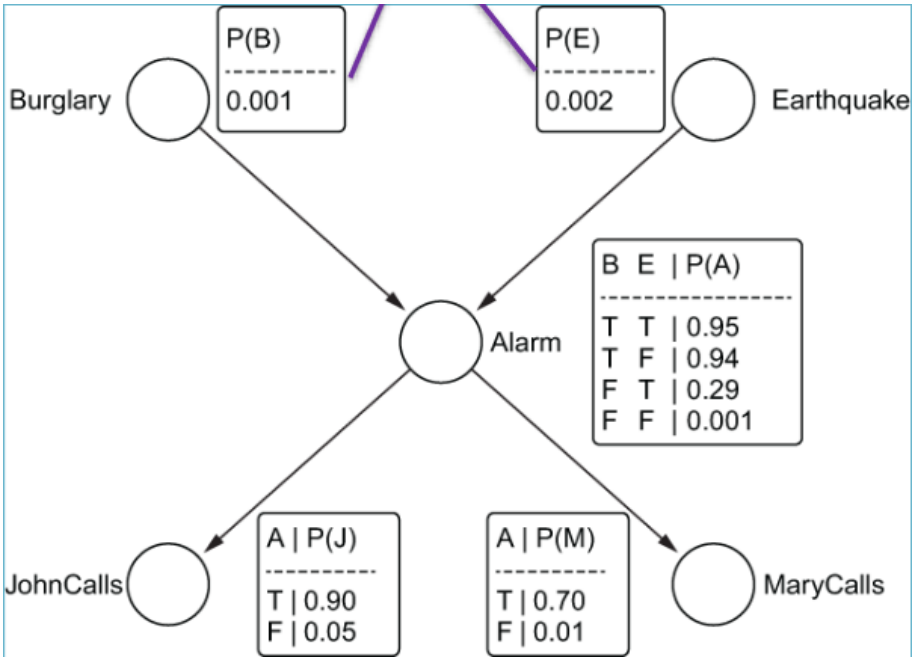
- In the item-based approach to collaborative filtering, the result of the learning phase, an Item-Item matrix that contains the similarity between each pair of items, can be stored in the graph easily. Starting from the bipartite graph created for storing the User-Item matrix, storing this matrix is a matter of adding new relationships that connect items to other items (so the graph will not be bipartite anymore). The weight of the relationship is the value of the similarity, between 0 and 1.
  To reduce storing and computations only the top K relationships for each node are stored.
- In the monitoring a subject use case we can use dynamic models, such as a **dynamic Bayesian network**, to build a predictive model for subject location.

Definition 11.1.1 — Bayesian network. A Bayesian network is a directed graph in which each node is annotated with quantitative probability information. It represents a mix of probability theory and graph theory in which dependencies between variables are expressed graphically.

Each node corresponds to a random variable that may be observable quantities, latent variables,

unknown parameters, or hypotheses. Instead edges represent conditional dependencies; so if there is an edge from node X to node Y, X is said to be a parent of Y. More in detail, each node X_i has a conditional probability distribution P(X_i, Parents(X_i)) that quantifies the effect of the parents on the node. In other words, each node is associated with a probability function that takes, as input, a particular set of values for the node's parent variables and gives, as output, the probability of the variable represented by the node.

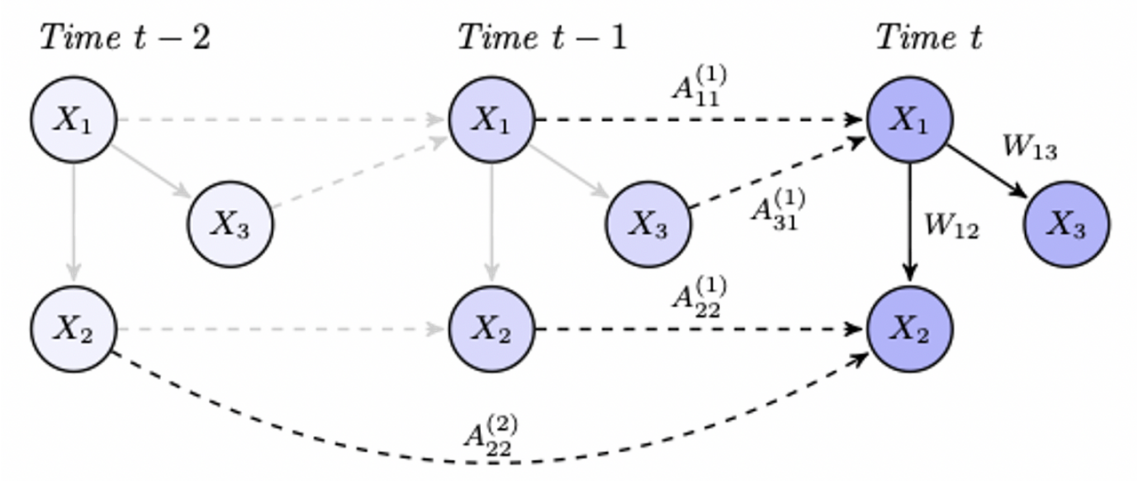
To understand it better consider the following use case: "You have a new burglar alarm installed at home. It is fairly reliable at detecting a burglary, but also responds on occasion to minor earthquakes. You have two neighbors, John and Mary, who have promised to call you at work when they hear the alarm. John nearly always calls when he hears the alarm, but sometimes confuses the telephone ringing with the alarm and calls then, too (false positive). Mary, on the other hand, likes rather loud music and often misses the alarm altogether (false negative). Given the evidence of who has or has not called, we would like to estimate the probability of a burglary.



Definition 11.1.2 — Dynamic Bayesian network. A dynamic Bayesian network (DBN) is a special type of Bayesian network that relates variables over adjacent time steps.

The simplest version of a DBN that can be used for performing a location prediction is a Markov chain, where nodes in this case represent the status at point t, and the weights of the relationships represent the probability of a status transition at time t + 1.

An extended model based is the **contextual Markov chain (CMC)**, in which the probability of the subject being in a location is also dependent on the hour of the day and the day of the week (which represent the context).



11.1.4 Graph visualization

Visualization is presented at the end of the process, because visualizing data after initial processing has been performed is much better than visualizing raw data, but data visualization can happen at any point in the workflow. Data visualization helps experts in analyzing complex data, identifying patterns,extracting valuable insights and enables decision-makers to make informed and effective decisions quickly and accurately. Why does visualizing graph data make it easier to analyze?

- Humans are naturally **visual creatures**. Our eyes are our most powerful sensory receptors, and presenting data through information visualizations makes the most of our perceptual abilities
- Many datasets today are too large to be inspected without computational tools that facilitate processing and interaction. If we explore the data in the form of a **graph**, it's **easier** to spot patterns, outliers, and gaps.
- The graph model exposes **relationships that may be hidden** in other views of the same data and helps us pick out the important details.

Choosing an effective visualization can be a challenge, because different forms have different strengths and weaknesses. It depends a lot on the use case analyzed.

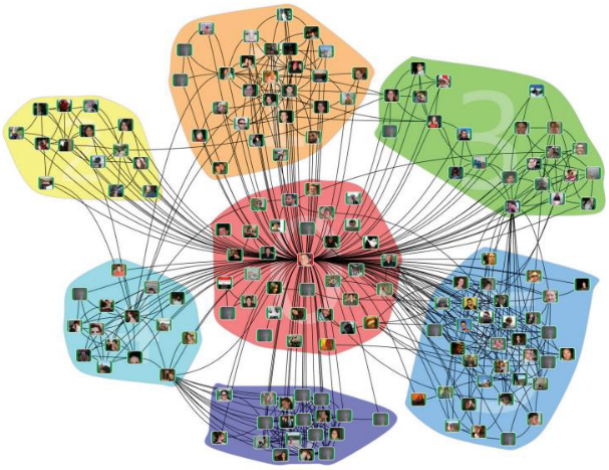


Figure 11.1: Visualization of one user's Facebook social network: we can notice that there are different cluster that help us to identify close friend and the typology of friends

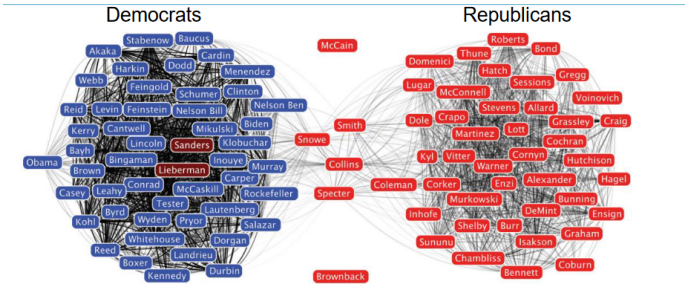


Figure 11.2: Voting patterns of U. S. senators during 2007: Democrats voted more similarly than republicans. We can also see that some republicans voted more similar to democrats that their colleagues. Democrats more united that republicans

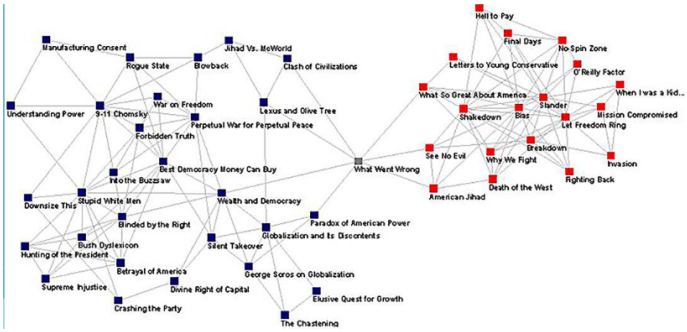
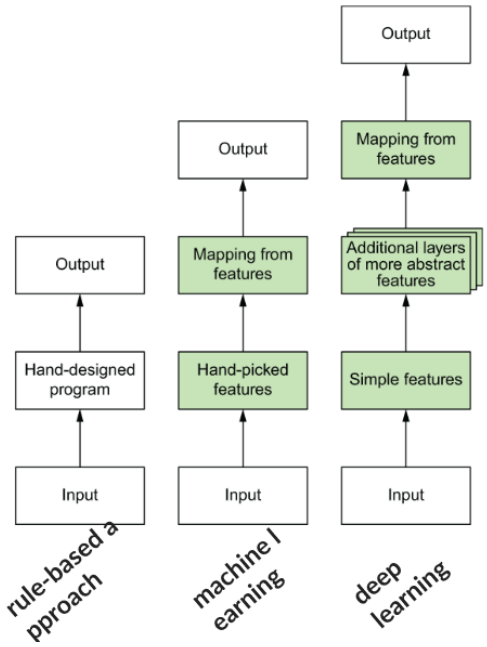


Figure 11.3: Understanding political opinions thanks to reading preferences

11.2 Deep learning and graph neural networks

For many tasks, it is not so simple to identify features to be extracted for training a model; one possible solution to this representation problem is to use machine learning to discover not only the mapping from representation to output, but also the representation itself. This approach is known as **representation learning**. An example of it is deep learning.



12. Recommendation Systems

Modelling a complex system such as recommendation system require two models:

1. The **descriptive model** is a simplified representation of reality (the training dataset) created to serve a specific learning purpose. The simplification is based on some assumptions about what is and is not relevant for the specific purpose.
2. The **predictive model** is a formula for estimating the unknown value of interest: the target. Such a formula could be mathematical, a query on a data structure, a logical statement such as a rule, or any combination of these. It represents, in an efficient format, the result of the learning process on the training dataset, and it is accessed to perform the actual prediction.

Definition 12.0.1 — Recommendation system. The term recommender system (RS) refers to all software tools and techniques that, by using the knowledge they can gather about the users and items (what the system recommends) in question, suggest items that are likely to be of interest to a particular user.

An RS normally focuses on a specific type or class of items, such as books to buy, news articles to read, or hotels to book. The overall design and the techniques used to generate the recommendations are customized to provide useful and relevant suggestions for that specific type of item.

Why are recommendation systems so commonly used?

- Increasing the number of items sold, offering any kind of recommendation
- Selling more diverse items, allowing the user to select items that might be hard to find without a precise recommendation
- Increasing user satisfaction: if the user finds the recommendations interesting, relevant, they will enjoy using the system
- Increasing user loyalty: customer-centric applications appreciate loyalty by recognizing returning customers and treating them as valued visitors. Tracking returning users is a common requirement for RSs because the algorithms use the information acquired from users during previous interactions
- Getting a better understanding of what the user wants: RS creates a model for the user's (profiling) preferences, which are collected explicitly or predicted by the system itself.

12.1 Content-based Recommendations

Suppose you would like to build a movie recommender system for your local video rental store. This scenario inherently has a lot in common with more-complex online recommender systems:

- A small user community
- A limited set of well-curated items that can have a lot of associated details. For movies, these details might include plot description, keywords, genres, and actors
- Knowledge of user preferences: The owner or shop assistant knows the preferences of almost all the customers, even if they've rented only a few movies or games.

CBRSs rely on item and user descriptions (content) to build item representations (or item profiles) and user profiles to suggest items similar to those a target user liked in the past. These types of recommender systems are also known as semantic-aware CBRSs. This approach allows the system to provide recommendations even when the amount of data available is quite small (that is, a limited number of users, items, or interactions).

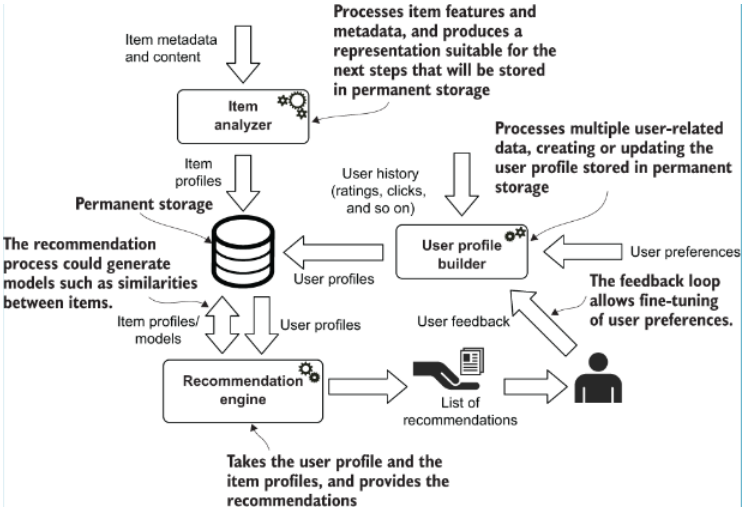
The basic process to do that is the following:

- matching the attributes of the target user profile, with attributes of the items to find items similar to what the user liked in the past
- the result is a relevance score that predicts the target user's level of interest in those items
- CBRSs use these content-rich items for making comparisons or inferring a user's interests based on the list of items they've interacted with.

Usually, attributes for describing an item are features extracted from metadata associated with that item or textual features somehow related to the item (actors, plot, comments from other users, keywords, and so on).

Usually CBRS are composed of three main components:

1. **Item analyser**: extract or identify relevant features and represent the items. It takes as input the item content (contents of a book) and meta information (book's author) from information sources and converts them to an item model that is used later to provide recommendations.
2. **User profile builder**: this process collects data representative of user preferences and infers user profiles, sking users about their interests or implicit feedback collected by observing and storing user behavior. The result is a model—specifically, a graph model—that represents user interest in some item (or item feature).
3. **Recommendation engine**: exploits the user profiles and item representations to suggest relevant items by matching user interests with item features. In this phase, a prediction model is built and used to create a relevancy score for each item for each user. This score is used to rank and order the items to suggest to the user.



Taking the final feedback of users about recommended items there is the possibility to create a continuing loop to improve constantly the model.

> **Corollary 12.1.1 — Features.** Features are defined generally as meta information because they are not actually the content of the item. Unfortunately, there are classes of items for which it is not so easy to find or identify features, such as document collections, email messages, news articles, and images.

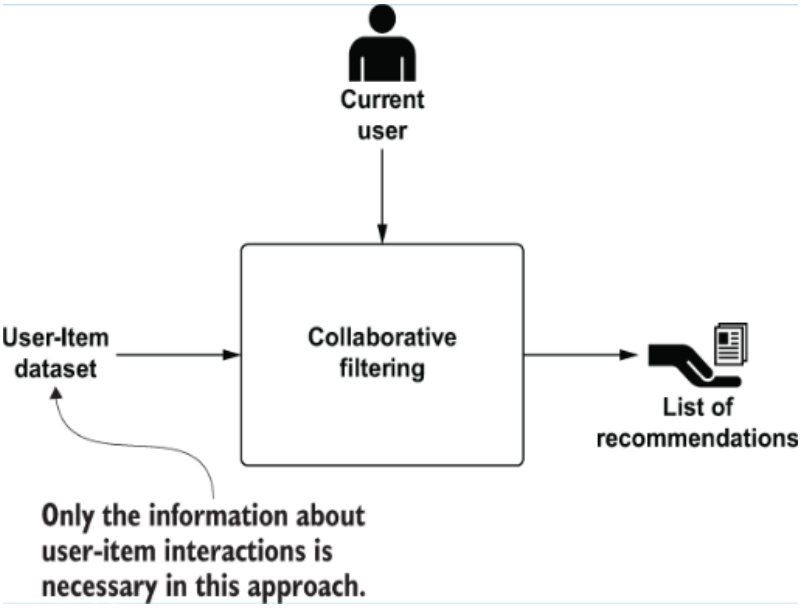
For example text-based items do not tend to have readily available sets of features. Nonetheless, their content can be represented by identifying a set of features that describe them. A common approach is the identification of words that characterize the topic. Different techniques exist for accomplishing this task where the result is a list of features (keywords, tags, relevant words) that describe the content of the item.

12.1.1 User profiling

The purpose of the user profile and the defined model is to help the recommendation engine assign a score to each item or item feature. The score helps rank the items suggested to the specific user, ordered from high to low. In a CBRS, several methods exist for gathering and modeling user profiles. A straightforward way to collect user preferences is to ask the user. The user might express interest in specific genres or keywords, or in particular actors or directors.

12.2 Collaborative Filtering Recommendations

An alternative to content filtering relies only on past user behavior, such as previous transactions, item rating or the opinions of an existing user community, to predict which items the users will most probably like or be interested in, without requiring the creation of explicit profiles for both items and users based on item features. This approach is known as collaborative filtering that analyzes relationships between users and interdependencies among items to predict new user-item associations.



The main advantages of this approach are the following:

- It is domain free (model doesn't depend on type of product)
- It doesn't require any detail about the items

- It can be applied to a vast variety of use cases and scenarios, and it can address data aspects that are often elusive and difficult to profile by using content filtering.

Collaborative Filtering RS is not without its challenges, one of which is known as **the cold-start problem**. This refers to the difficulty in providing accurate recommendations for new items or users, or when there is limited interaction data available. Despite this, there are various mechanisms in place that can help mitigate the effects of the cold-start problem. For instance, different algorithms like the graph approach can be used, and other sources of knowledge such as social networks can also prove useful.

Collaborative Filtering RS can be implemented with two main approaches:

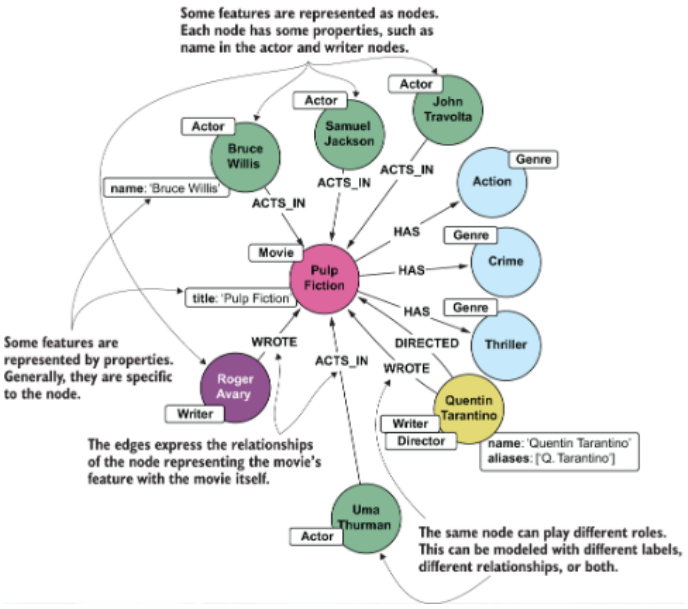
1. **Memory-based**: also called neighborhood-based methods this approach for a particular user's rating for Saving Private Ryan, would look for the movie's nearest neighbors that this user rated. Alternatively, the algorithm can look for similar users based on the set of films they watched and suggest something that the current user hasn't watched yet. In these methods, the User-Item dataset that is stored is used directly to predict ratings for items.
2. **Model-based**: create models for users and items that describe their behavior via a set of factors or features and the weight these features have for each item and each user. In the movie example, the discovered factors might measure obvious dimensions such as genre (comedy, drama, action) or orientation to children; less well-defined dimensions such as depth of character development or quirkiness; or uninterpretable dimensions. For users, each factor expresses how much the user likes movies that score high on the corresponding factor. In these methods, the raw data (the User-Item dataset) is first processed offline, with the information on ratings or previous purchases used to create this predictive model. At runtime, only the precomputed or learned model is required to make predictions during the recommendation process. **Latent factor models** represent the most common approaches in this class. They try to explain the ratings by characterizing both items and users on, say, 20 to 100 factors inferred from the rating patterns. In a sense, such factors comprise a computerized alternative to the human-created features encountered in the content-based recommendation systems

13. Movie Recommendation System

In order to build a recommendation system with a content-based approach, the first thing to do is define the features of our items. How can we represent movies and which are their features? Each movie can be described by using genres or categories, plot description, actors, tags or keywords manually (or automatically) assigned to the movie, year of production, director, writers and producers.

13.1 Item Modelling

How can we represent movies in a Graph DB? A good representation could be the one shown in the figure below.



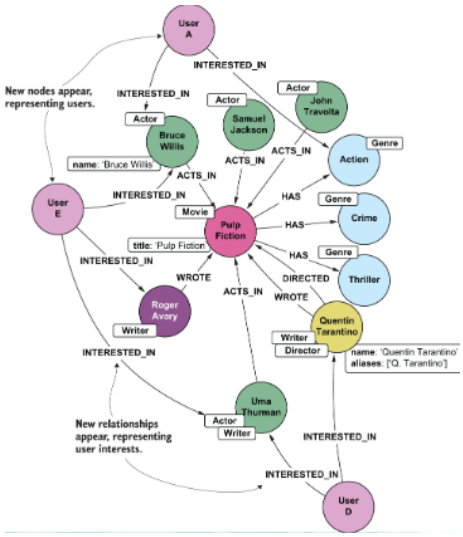
This model takes the following advantages:

- No data duplication.
- Preventing data duplication guarantees better tolerance to errors in the values.
- Easy to extend/enrich.
- Easy to navigate enables multiple and more efficient access patterns to the data.

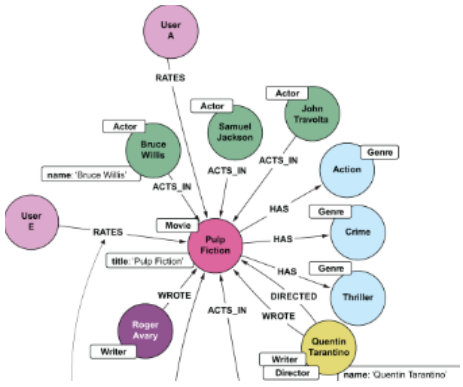
13.2 User modeling

In a CBRS, several methods exist for gathering and modeling user profiles. The selected design model will vary according to how the preferences are collected and the type of filtering strategy or recommendation approach. A straightforward way to collect user preferences is to ask the user. The user might express interest in specific genres or keywords, or in particular actors or directors. From a high-level perspective, the purpose of the user profile and the defined model is to help the recommendation engine assign a score to each item or item feature. The score helps rank the items suggested to the specific user, ordered from high to low. For this reason, recommendation systems belong to the area of machine learning called **learning to rank**. As said we have different options to model user:

1. We can add preferences or interests to the model we are designing by adding nodes for users and connecting them to the features of interest.



2. The system can explicitly ask the user to rate some items. The ratings are stored on the edges as a property.



3. Infer the users' interests, tastes, and preferences implicitly by considering the interactions each user has with items.

13.3 Providing Recommendations

CBRS uses user profiles to match users with the items most likely to interest them. Depending on the information available and the models defined for both users and items, different algorithms or techniques can be used for this purpose:

13.3.1 1st Approach: Graph model with user interests pointing to meta information

This approach is applicable when the items are represented by a list of features and the user profiles are represented by connecting the users to features that are of interest to them. These connections are described in a binary form: like and don't like.

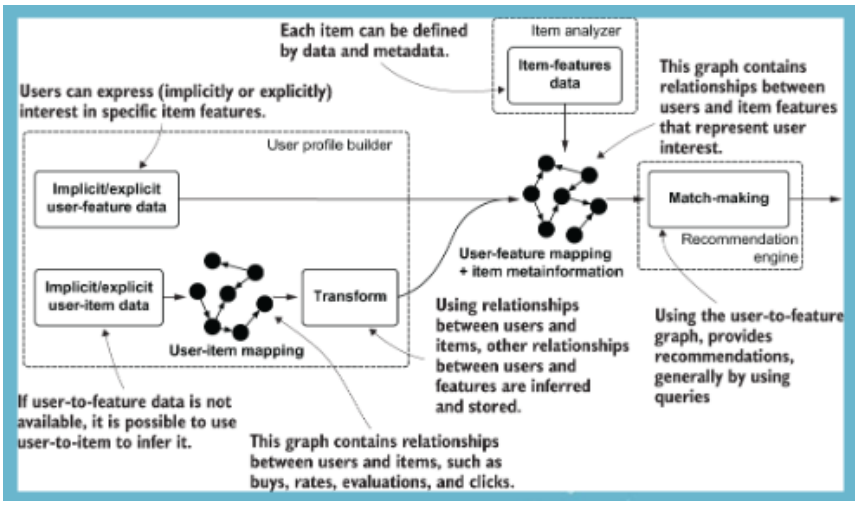


Figure 13.1: No need to prebuild any model because the description and prediction models overlap.

This pure graph-based approach is simple but has a lot of advantages:

- The quality of the recommendations is quite high, considering the limited effort required by this method.
- It's simple because it doesn't require complex computations or complex code that reads and preprocesses the data before providing recommendations and all information are stored in

one graph.
- It's extensible, the graph can contain other information that can be useful for refining the results according to other data sources or contextual information.

The limitations of this approach are the following:
- In the user profile, interest in an item feature is represented by a Boolean value. This value is binary, representing only the fact that the user is interested in the feature. It doesn't ascribe any weight to this relationship.
- We need a function that computes the similarity or commonalities between user interests and items.

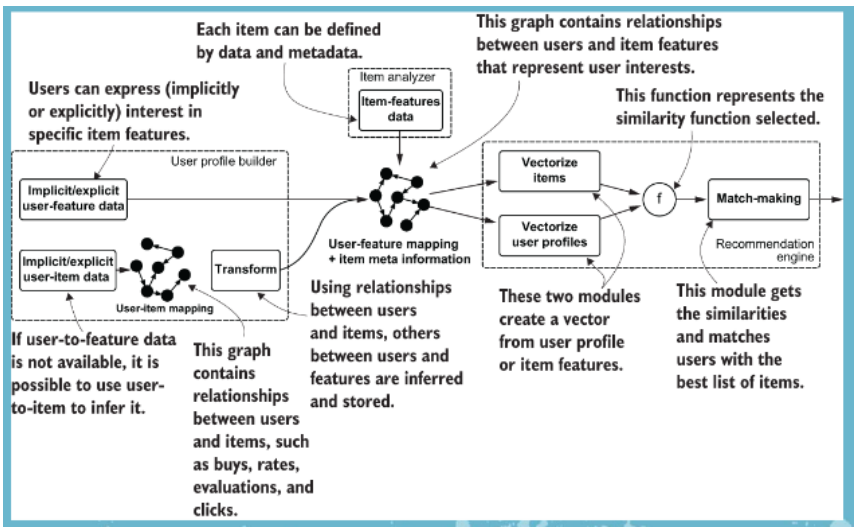
13.3.2 2nd Approach

The second approach extends the previous one by considering two main aspects:

1. A function that measures the similarity. Different functions are available. One of the most accurate functions is **cosine similarity**:

$$sim(\overrightarrow{a}, \overrightarrow{b}) = cos(\overrightarrow{a}, \overrightarrow{b}) = \frac{\overrightarrow{a} \cdot \overrightarrow{b}}{|\overrightarrow{a}|x|\overrightarrow{b}|} \tag{13.1}$$

2. A common representation for both items and user profiles so that the similarity is measurable



In our scenario we have to convert items to vectors before compute recommendations.

	Action	Drama	Crime	Thriller	Adventure	Quentin Tarantino	Jonathan Hensleigh
Pulp Fiction	1	0	1	1	0	1	0
The Punisher	1	1	1	1	1	0	1
Kill Bill: Vol I	1	0	1	1	0	1	0

We can also add all the features including those that have numerical values, such as the average ratings in our movie scenario but to compute the cosine distance between vectors, we should

consider some appropriate scaling of the non-Boolean components so that they neither dominate the calculation nor are irrelevant.

$$Vector(PulpFiction) = [1,0,1,1,0,1,0,4\alpha] \tag{13.2}$$

Because the vector space has the feature values as dimensions, the first step in the projection is to migrate the user-item matrix to the user-feature space. The simplest technique is aggregating by counting the occurrences of each feature in a user's list of previously liked items. Another option is to compute the average values for numerical features.

	Action	Drama	Crime	Thriller	Adventure	Quentin T arantino	Jonathan Hensleigh	Total
User A	3	1	4	5	1	3	1	9
User B	0	10	1	2	3	0	1	15
User C	1	0	3	1	0	1	0	5

We can also normalize each value with the total number of movies watched. Sometimes normalization is useless because the update of total number of movies watched change all the normalized values in the vectors. As a difference with the first approach, the recommendation task requires complex operations that cannot be accomplished by a query because they require complex computations, looping, transformation, and so on.

13.4 3rd Approach

The third approach we will consider for content-based recommendations can be described as "Recommend items that are similar to those the user liked in the past". This approach works well in general and is the only option when it is possible to compute relevant similarities between items but difficult or not relevant to represent user profiles in the same way. This technique, known as the **similarity-based retrieval approach**, is a valuable approach to cover for several reasons:

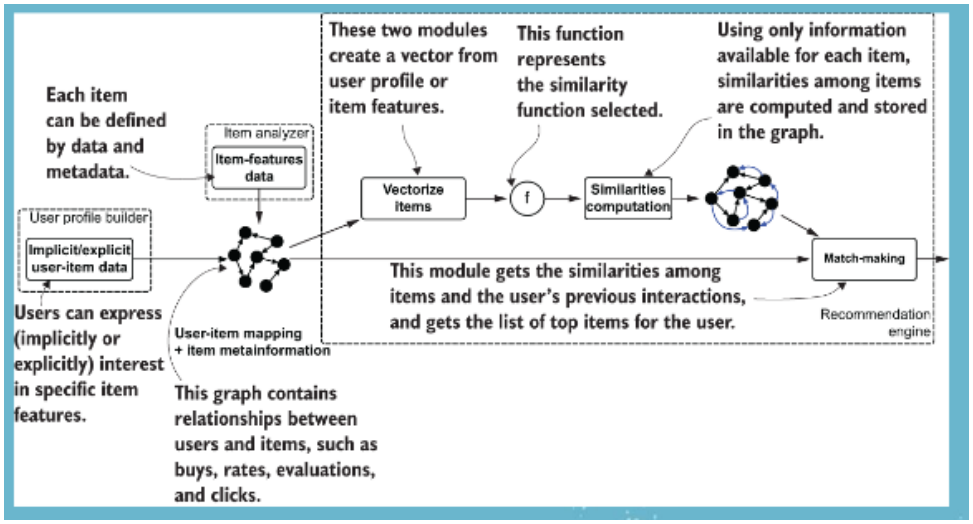
- Similarities are easy to store back in the graph as relationships between items.
- It is one of the most common and powerful approaches to CBRSs.
- It is flexible and general enough to be used in many scenarios (it doesn't depend on the number and the type of data/information avalaible).

Three key elements are necessary in this scenario:

- User profile, represented by modeling the interactions the user has with items, such as rated, bought, or watched. (represented as relationships between the nodes user the nodes items).
- Item representation/description, to compute similarities between items.
- Similarity function. Another typical similarity metric, which is suitable for multivalued characteristics, is the **Dice Coefficient**, that just compute how two items are similar basing on number of keywords/feature in common.

$$Dice - coefficient(I_i, I_j) = \frac{2x|keywords(I_i) \cap keywords(I_j)|}{|keywords(I_i)| + |keywords(I_j)|} \tag{13.3}$$

It is necessary to compute all similarity values but, although, it's not necessary to store the neighbor relationships between each pair of nodes. Generally, only a small number is stored. You can define a minimum similarity threshold, or you can define a k value and keep only the k topmost similar items (**K-nearest neighbor (k-NN)**)



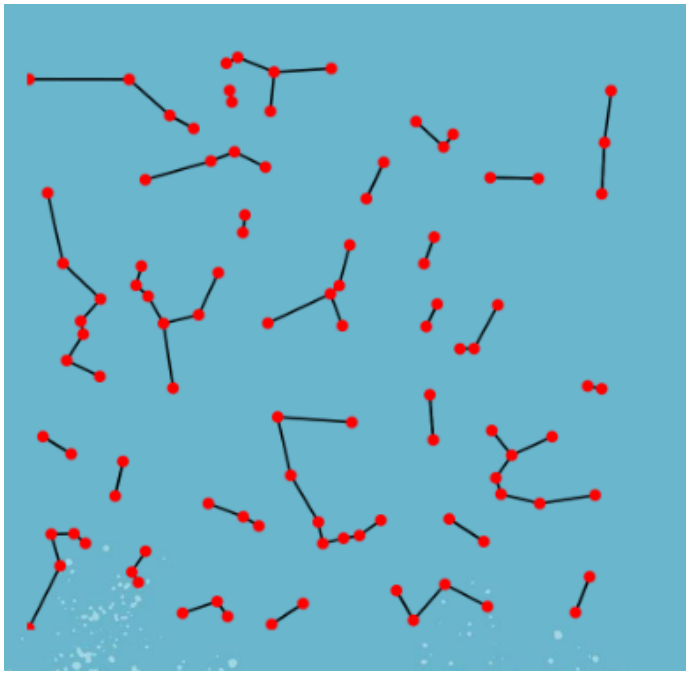


Figure 13.2: KNN is not a connected graph because we don't store all similarities

Unlike in the first and second approaches, in which the recommendation process uses the data as it is, the recommendation process here requires an intermediate step: this k-NN computation and storing. In this case, the descriptive model and the prediction model don't match.

The last step in the recommendation process for this third approach consists of making the recommendations, which we do by drawing on the k-NN network and the user's implicit/explicit preferences for items. This task can be accomplished in different ways:

- In the simplest approach, the prediction for a not-yet-seen item d for a user u is based on a voting mechanism considering the k most similar items to the item d. For example, if the current user liked 4 out of k = 5 of the items most similar to d, the system may guess that the chance that the user will also like d.
- Another, more-accurate approach is inspired by collaborative filtering, involves predicting the interest of a user in a specific item by considering the sum of all the similarities of the

target item to the other items the user interacted with before:

$$interest(u,p) = \sum_{i \in Items(u)} sim(i,p) \tag{13.4}$$

Items(u) returns all the items the user has interacted with (liked, watched, bought, clicked). The returned value can be used to rank all the not-yet-seen items and return the top k to the user as recommendations.

Advantages took by the graph approach are the following:

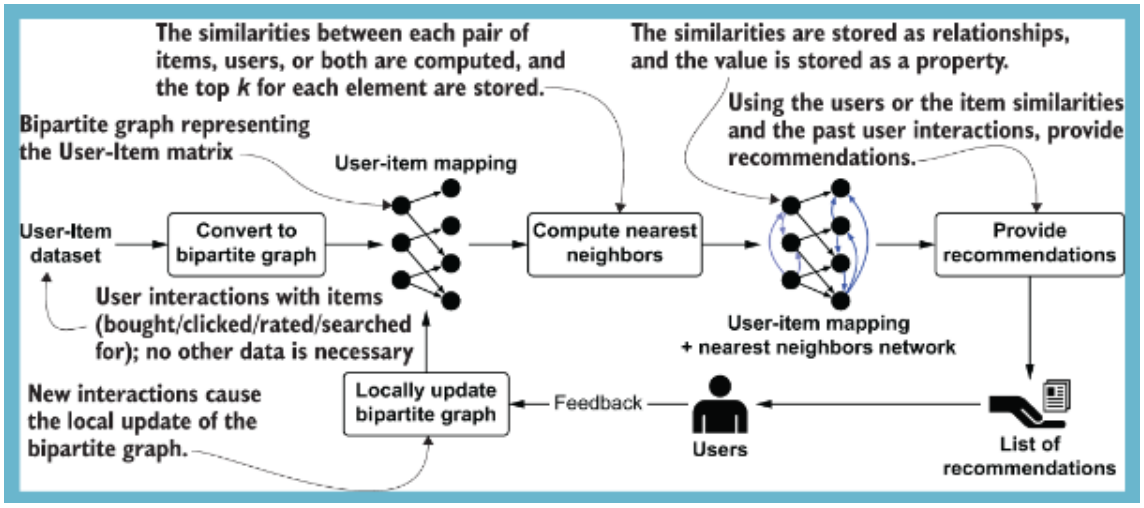
- Meaningful information must be stored as unique node entities in the graph so that these entities can be shared across items and users
- Converting user-item data to user-feature data is a trivial task when the meta information is available and is meaningful
- It is possible to extract several vector representations for both items and user profiles from the same graph model.
- It is possible to store different similarity values computed with different functions and use them in combination.

14. E-commerce Recommendation System

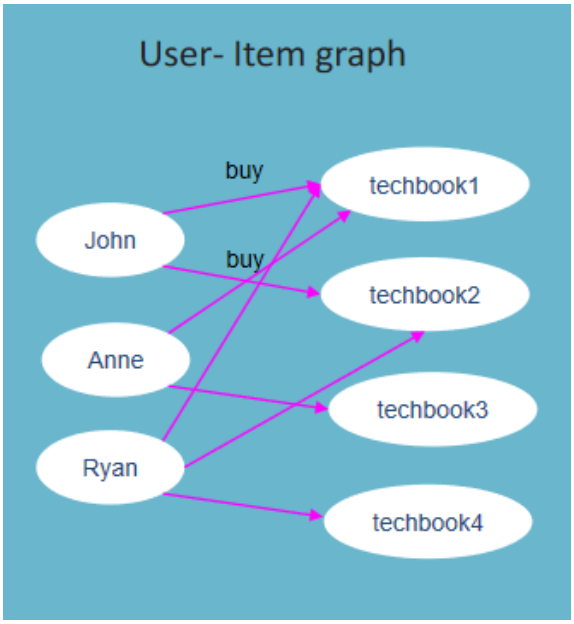
We want to design graph models and the algorithms on top to implement a recommender system for an e-commerce site that uses a collaborative filtering approach to gently suggest to users items that could be of interest to them. The site sell many types of items, such as books, computers, watches, and clothing. The details about each item are not curated; some items have only a title, one or more pictures, and a small and useless description.
Users' activity is tracked so it is possible to collect and store a huge variety of interactions between users and items (purchases, clicks, searches, and ratings). We can already see that this approach has various advantages:

- Less information required
- It's not required merge and integration between heterogeneous sources of data.
- Recommendation quick and immediate (no cold-start problem)

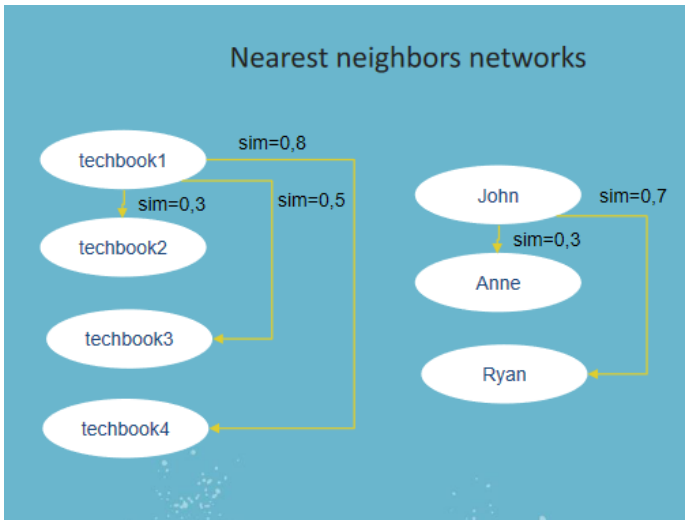
The most common graph-based approach is shown in the figure below. The first is to build a user-



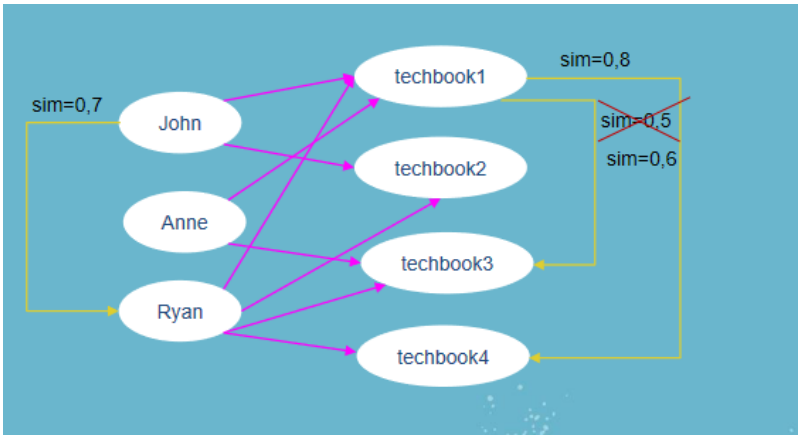
item bipartite graph that models users' activity. In this graph, we will have only nodes belonging to two classes (item and user) and edges that connect only two different classes. Notice that a bipartite graph can model only one type of interaction, hence, in the case of multiple interactions we should build on a bipartite graph for each interaction.



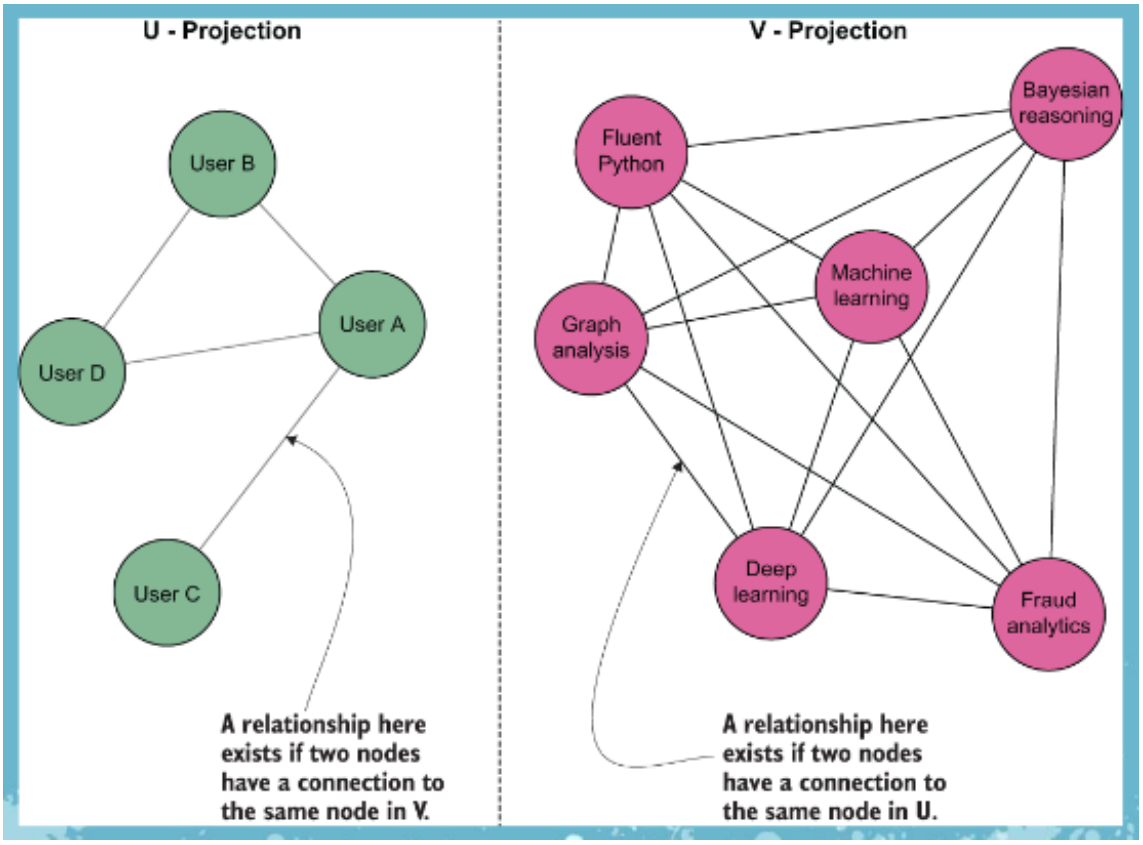
After we compute the similarity between items and users and we build the **Nearest neighbors networks**. Similarity between items can be the number of times that two items are bought together; instead similarity between users depends on their activity.



At the end will have a final graph (user-item mapping + nearest-neighbors network) that help us to provide recommendations to final users.



In our e-commerce scenario is possible to infer connections between nodes of the same type, creating a one-mode projection. Thanks to these representations we can have a clear picture of similarities between items and users but we lost general information (who bought two items together).



Furthermore we can add some weights on the edges to understand what is the best recommendation choice. To recap advantages of a bipartite graph representation are the following:

- It represents the data in a compact and intuitive way. The User-Item dataset is sparse by nature; generally, it has a lot of users and a lot of items, and users interact with a small portion of the items available.
- Projections derived from the bipartite graph are information rich and allow different types of

analyses both graphical and via algorithms.

- The representation can be extended by modeling multiple bipartite graphs that have the same set of vertices but use different edges. Engines use multiple types of interactions to provide better recommendations.
- This approach simplifies the recommendation phase by providing access to a single data structure that contains both user preferences and the nearest neighbors network.

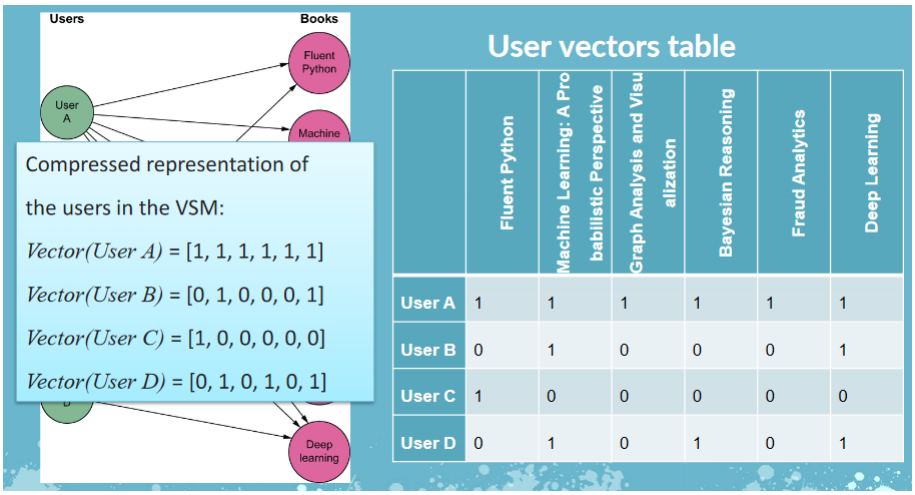
14.1 Computing the nearest-neighbor network

There are two possible approaches to memory-based recommendation for collaborative filtering:

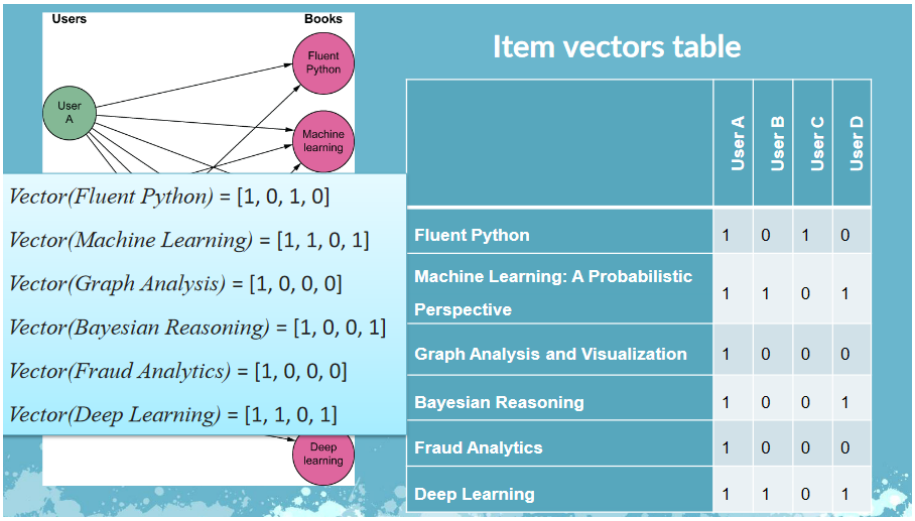
- **Item-based**: the similarities are computed between items based on the users who interact with them (rating, buying, clicking, and so on).
- **User-based**: the similarities are computed between users based on the list of items they interact with.

The procedure for the similarity computation is the same as that for the content-based approach:

- Identify/select a similarity function that allows you to compute distances between homogeneous elements in the graph.
- Represent each element in a way that's suitable for the selected similarity function
- Compute similarities, and store them in the graph.



	Fluent Python	Machine Learning: A Probabilistic Perspective	Graph Analysis and Visualization	Bayesian Reasoning	Fraud Analytics	Deep Learning
User A	1	1	1	1	1	1
User B	0	1	0	0	0	1
User C	1	0	0	0	0	0
User D	0	1	0	1	0	1

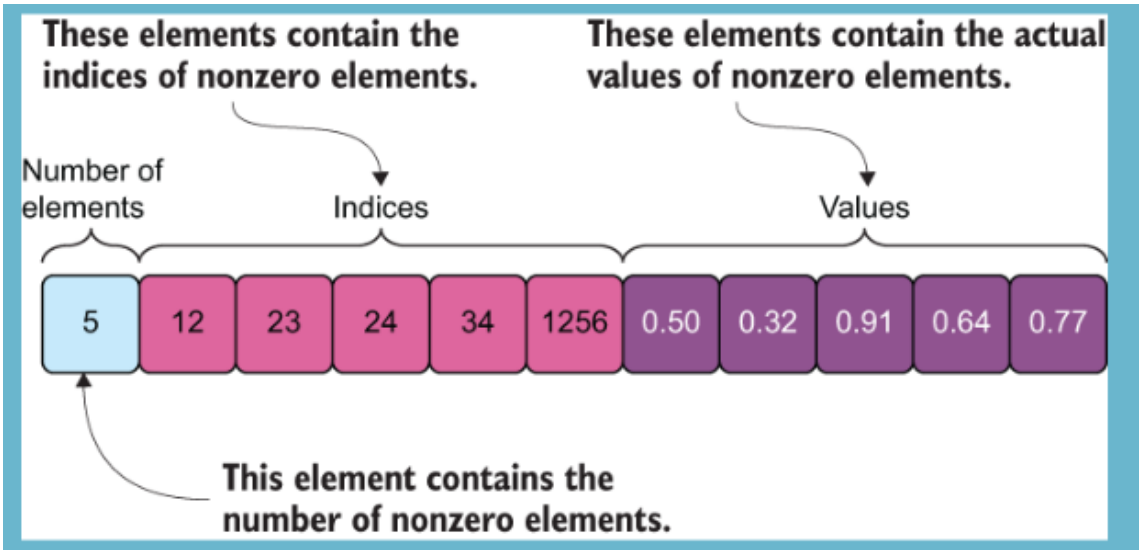


	User A	User B	User C	User D
Fluent Python	1	0	1	0
Machine Learning: A Probabilistic Perspective	1	1	0	1
Graph Analysis and Visualization	1	0	0	0
Bayesian Reasoning	1	0	0	1
Fraud Analytics	1	0	0	0
Deep Learning	1	1	0	1

During this process the sparseness or denseness of a vector's data is the most important consideration. In fact, representing vectors in a memory-efficient way and accessing their values efficiently are important machine learning tasks.

- A vector with relatively many nonzero values in relationship to its size is called **dense**.
- A vector, in which the total percentage of nonzero values is small, is called **sparse**.

Representing sparse vectors with an array is not only a waste of memory, but also makes any computation expensive. It is possible to represent a sparse vector in different ways to optimize memory and simplify manipulation.



If density is greater than sparsity it's useful to compute the compact vector. Usually this computation is done in collaborative-filtering approach because we have big and sparse vector. Instead, in the content-based compact vector is useless, unless we have an high number of features.

14.2 Computing similarities

The next step is computing and storing the similarities. For each user or item, we have to do the following steps:

- Compute the similarities with all the other elements
- Order the similarities in a descending order.
- Keep only the top k, where the value of k is predefined. Alternatively, fix a threshold or minimum similarity value, and keep only the similarities above it.
- Store the top k similar elements in the graph as new relationships between users or items.

The process distills new knowledge from the bipartite graph and stores it back in a way that can serve various purposes in addition to recommendations

- **Clustering items**: by applying some graph clustering algorithms on the items' nearest neighbor network, it is possible to recognize (for example) groups of products that are generally bought together or movies that are watched by the same group of users (knn + bipartite).
- **Segmenting users**: the same clustering algorithms can be applied on the users' nearest neighbor network, and the result will be a group (segment) of users who generally buy the same products or see the same movies (knn + bipartite).
- **Finding similar products**: the items' nearest neighbor network itself is useful. If a user is looking at a specific product, by querying the graph it is possible to show a list of similar

products based on the SIMILARITY relationship, and this operation will be fast (knn).

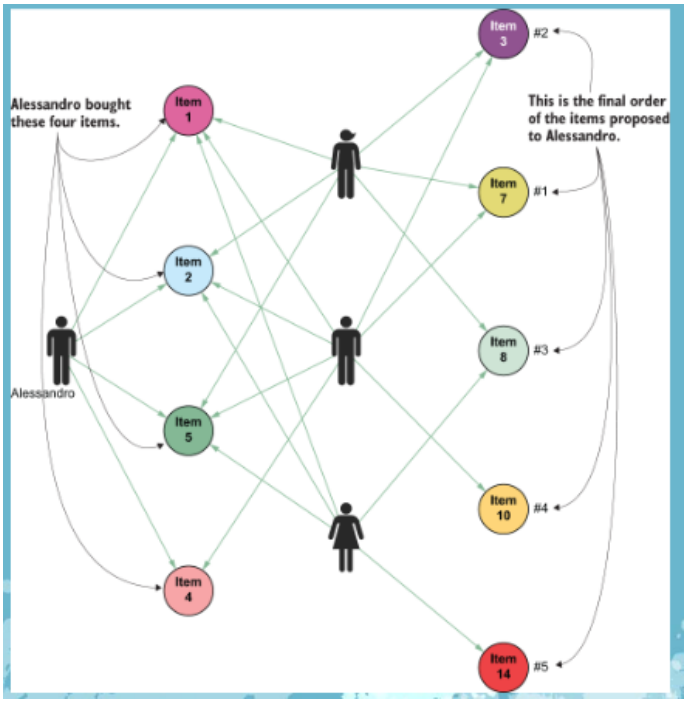
14.3 Providing recommendations

At a high level, the recommendation process produces the following types of output:

- **Relevance scores**: numerical predictions indicating the degree to which the current user likes or dislikes a certain item.
- **Recommendations**: a list of N recommended items.

14.3.1 User-based

The basic idea in the user-based approach is that given the current user as input, the bipartite graph representing the interaction database, and the nearest neighbors network, for every product p that the user has not seen or bought, a prediction is computed based on the ratings for p made by the peer users (the users in that user's nearest neighbors network). Two cases can occur:



1. In one case, the interactions between users and items contain no weight, it is called the binary or **Boolean** model. The Boolean case is a little trickier, because in the literature, no single method is recognized as the best approach.

$$score(a,p) = \frac{1}{|KNN(a)|} \times \sum_{b \in KNN(a)} r_{b,p} \tag{14.1}$$

   $r_{b,p}$ will be 1 if the user b purchased the product p and 0 otherwise, so the sum will return how many of the nearest neighbors of a bought the product p. This value is normalized with the number of nearest neighbors for a (the value of |KNN(a)|)

2. In the second case, there is a weight to the interactions (such as ratings).

$$pred(a,p) = \frac{\sum_{b \in KNN(a)} sim(a,b) \times r_{b,p}}{\sum_{b \in KNN(a)} sim(a,b)} \tag{14.2}$$

   KNN(a) represents the k-nearest neighbors of the user a, and $r_{b,p}$ represents the rating assigned by the user b to the product p. This rating can be 0 if user b doesn't rate item.

14.3.2 Item-based

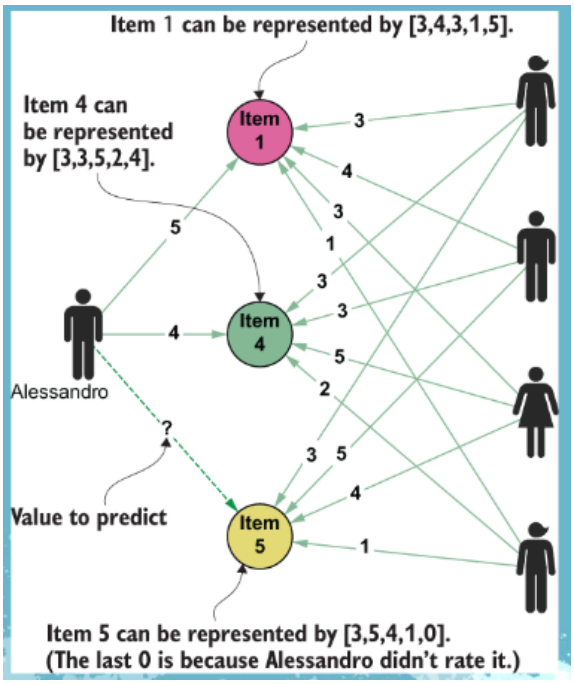
The main idea of item-based algorithms is to compute predictions by using the similarity between items, not users.

- If the original User-Item dataset contains rating values, the formula to predict the rating for a not-yet-seen product in the dataset is the following:

$$pred(a,p) = \frac{\sum_{q \in ratedItem(a)} (sim(p,q) \times r_{a,q} \times |KNN(q) \cap p|}{\sum_{q \in ratedItem(a)} (sim(p,q) \times |KNN(q) \cap p|} \tag{14.3}$$

  - $q \in ratedItem(a)$ considers all the products rated by user a.
  - $|KNN(q) \cap p|$, which is 1 if p is in the set of nearest neighbors of q and 0 otherwise. It is possible to consider only the nearest neighbors of q and not all the similarities.
  - The denominator normalizes the value to not exceed the max value of the rating
- As with the user-based approach, for the Boolean case there is no accepted standard approach for computing the scores.

Consider the example shown in the figure below. The similarities among all the three items are



Item 1-Item 4: 0.943, Item 1-Item 5: 0.759, Item 4-Item 5: 0.811. The $|KNN(q)c \cap p|$ is always 1. The user Alessandro rated Item 1 with 5 stars and Item 4 with 4 stars. The formula for predicting Alessandro's interest/stars for Item 5 is

$$pred(a,5) = \frac{0.759 \times 5 + 0.811 \times 4}{0.759 + 0.811} = \frac{7.039}{1.57} = 4.48 \tag{14.4}$$

14.3.3 Cold-start problem

In real-world e-commerce applications, User-Item matrices tend to be sparse, as customers typically have bought only a small fraction of the products in the catalog. The problem is even worse for new users or new items that have had no or few interactions. This problem is referred to as the cold-start problem, and it further illustrates the importance of addressing the issue of data sparsity.
Which approach can be applied to solve the cold start problem?

1. To exploit additional information about the users, such as gender, age, education, interests, or any other available data that can help classify a user.

2. Creates hybrid recommender systems that merge multiple approaches in a single prediction mechanism. These approaches are no longer purely collaborative, and new questions arise as to how to acquire the additional information and combine the different classifiers

A graph-based approach to cold start problem is the **Transitive associations method**. The main idea of this approach is to exploit the supposed transitivity of customer tastes when they share items. Users' preferences are represented by the items and their interactions with the items. In

	Item 1	Item 2	Item 3	Item 4
User 1	0	1	0	1
User 2	0	1	1	1
User 3	1	0	1	0

the transitive associations method, the recommendation approach can be easily implemented in a graph-based model by computing the associations between item nodes and user nodes. The recommendations are determined by determining paths between users and item; the higher the number of unique paths connecting an item node to a user node, the stronger the association is between those two nodes.

An extended version of this method consider also longer paths, the are called indirect associations, to compute recommendations.

It is necessary to define a scoring mechanism to order the recommendations list. In this case, recommendations are made based on the associations computed for pairs of user nodes and item nodes. Given a user node User t and an item node Item j, the association between them, score(User t, Item j), is defined as the sum of the weights of all distinctive paths connecting User t and Item j.

$$score(user, item) = \sum_{p \in pathsBetween(user,item,M)} \alpha^{length(p)} \tag{14.5}$$

In this formula, only paths whose length is less than or equal to the maximum defined length M will be considered. The limit M is a parameter that the designer of the recommender system can control. Alpha is parameter that control weight of undirected paths.

Advantages of this approach are the following:

- Generates high-quality recommendations even when a small amount of information is available.
- Uses the same User-Item dataset used for the creation of the bi-graph.
- It is purely graph-based, and it uses the same bi-graph model.

A comparison with the standard user-based and item-based algorithms shows that the quality of the recommendations can be significantly improved by the proposed technique based on indirect relationships, in particular when the ratings matrix is sparse.

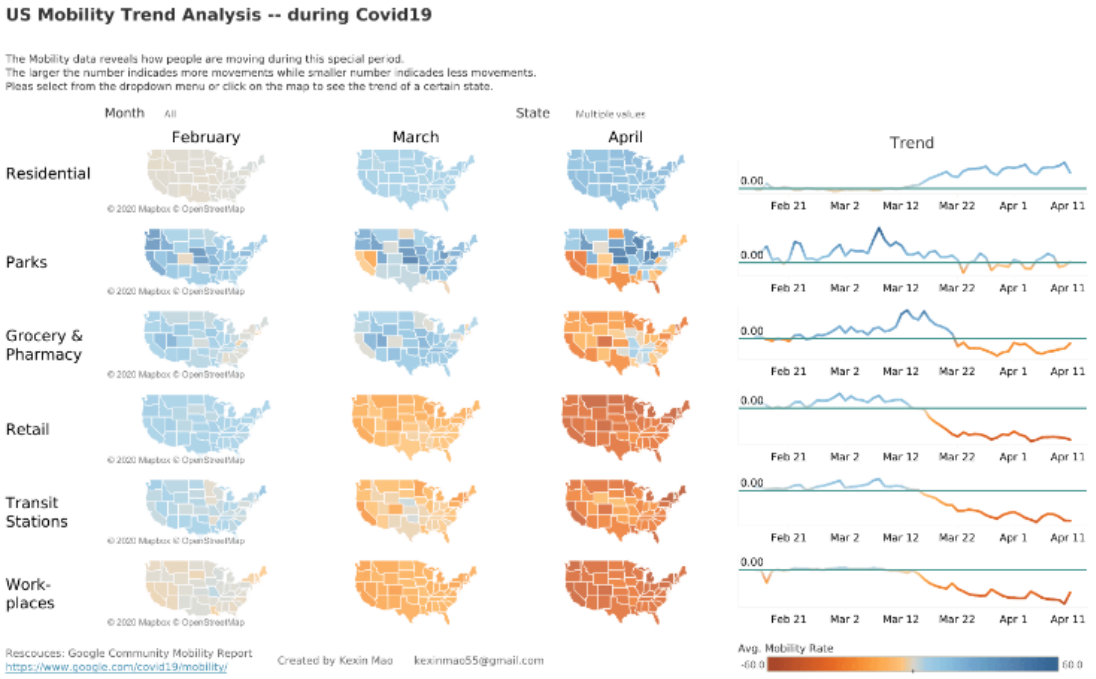
15. Graphs for mobility

What are the key Challenges in Urban Mobility?

1. **Traffic Congestion**: Dense urban centers face significant traffic congestion, leading to delays.
2. **Air Pollution**
3. **Greenhouse Gas Emissions**: The transportation sector significantly contributes to greenhouse gas emissions, influencing climate change.
4. **Road Safety**: Road accidents are a constant concern, necessitating improved infrastructure and safe driving practices.
5. **Accessibility**: Certain communities, especially rural or underserved areas, struggle with access to reliable public transportation.
6. **Public Transport Needs**: Cities must provide effective, efficient, and accessible public transport to reduce reliance on private vehicles.
7. **Inclusion**: Ensuring transportation systems are accessible to all, including the disabled, elderly, and low-income individuals.
8. **Emerging Technologies**: Adoption of technologies like autonomous vehicles and electric transport poses regulatory, infrastructure, and public acceptance challenges.

15.1 Types of Mobility Data

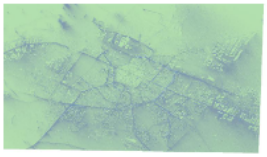
- **Static Data**: Only spatial dimension (transport network, traffic averages).
- **Dynamic Data:**: Includes both spatial and temporal dimensions (real-time traffic, movement trajectories).
  - Fixed space and time varies
  - Fixed time and space not aggregated
  - Space and time not aggregated



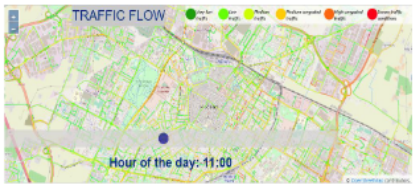
MAPPA AGGREGATA

MAPPA ISTANTANEA

MAPPA REAL-TIME O DINAMICA



MAPPA TEMPORALE



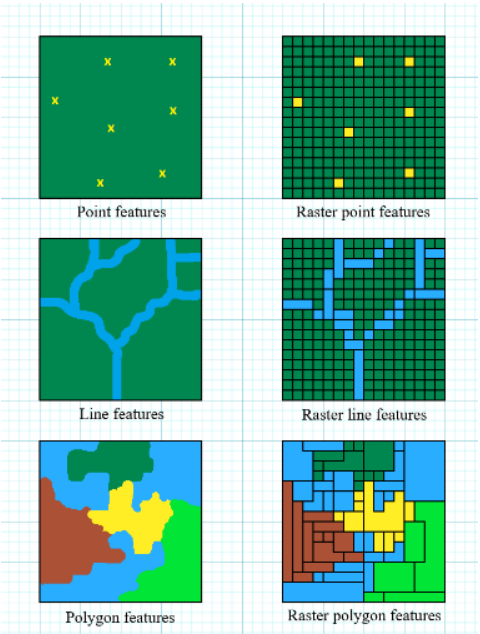
15.2 Geospatial Data

Geodata are information that have an explicit or implicit association with a location on the earth. Geodata depends on the coordinate system used. We can have two main CS:

- Geographic Coordinate System (GCS): where the point is positioned on the surface of the earth.
- Projected Coordinate System (PCS): for mapping locations (depend on how is the projection onto a plane)
  - An example can be the **SRS** that use a coordinate system combined with a datum, a set of parameters that define the position of the origin, the scale and the orientation of the coordinate system.

Geodata can belong to different types:

- Vector data (points, lines, polygons)
  - **Well Known Text(WKT)**
  - **Well Known Binary (WKB)**
- Raster data (images with spatial coordinates)



To store this type of data we have different options:

- RDBMS such as PostgreSQL, Timescale or PostGIS
- NoSQL databases like MongoDB + GeoJSON and Neo4j + Spatial Plugin (add r-tree index, data from OSM, spatial function)

There are two main techniques to index geospatial databases:

- **R-TREE**: divides data into rectangles, sub-rectangles, sub-sub-rectangles, etc. It is a dynamic indexing structure for spatial search. First, search inside the rectangle. If not found, search in the rectangle below. If still not found, search in the rectangle next to it and find it.
- **GiST**: Generalized search trees. Divides data into objects in one position, overlapping objects and objects inside. First, search all points in the green circle and find nothing. Then search the yellow circle that overlaps the green one and all points inside the yellow circle and find it.

Graph Database bring both advantages and disadvantages:

- Performance (quick and scalable) that depend on model built.
- Good representation of geodata.
- We need to manage temporal series
- Model depends a lot on the aim.

15.3 Road Network

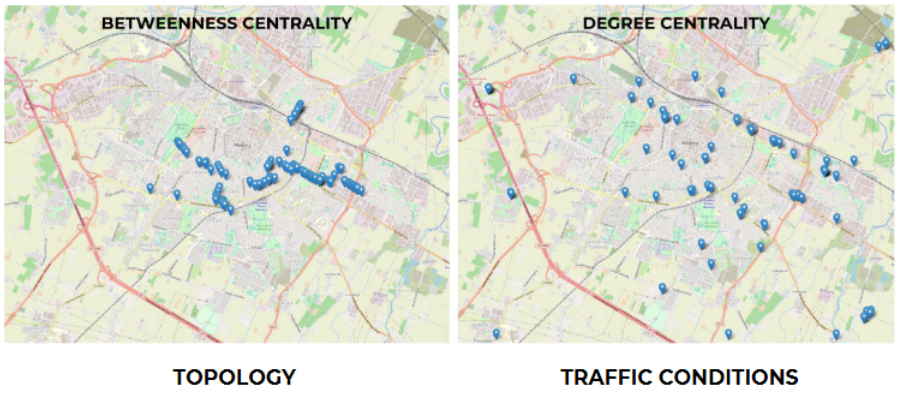
How can we model the road network composed of roads and intersections? We have two main approach:

- **Primal Graph**: perfect for routing.
- **Dual Graph**: less memory required (smaller graph).

15.3.1 Primal Graph

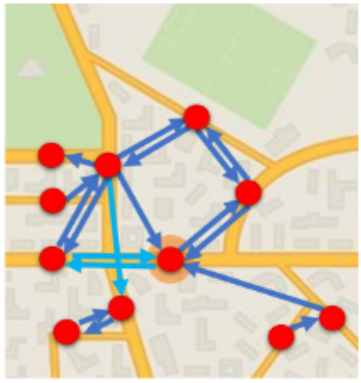
Nodes represent intersections and edges represent roads.We can also add status of road and geometrical info (lat and lon).

In these type of graph betweenness and degree centrality help identify key junctions and congested roads.



15.3.2 Dual graph

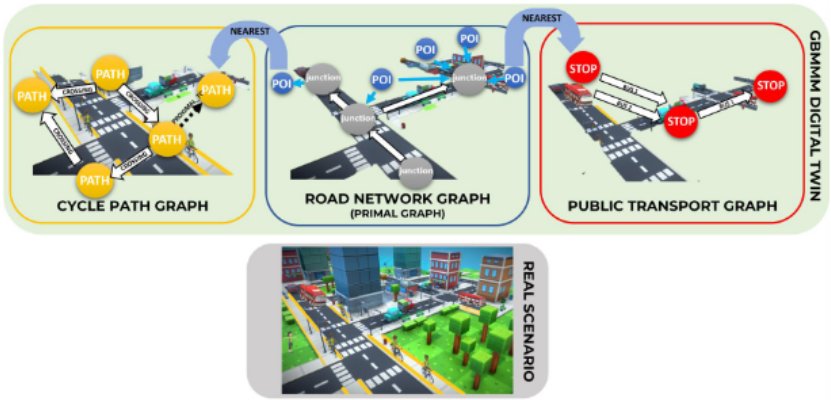
Nodes represent roads and edges represent connections at intersections.



Dual graph can be used in combination with primal one to identify congestion and important road. Using page rank on dual graph can help us:

- Connection of roads
- Identify congested junctions

Additional layers can be added to recreate real scenario.



16. Fraud Detection

16.1 Fraud and Anomaly Detection

Some of the techniques for fraud detection we will see are borrowed from the more generic field of anomaly or outlier detection. An anomaly or outlier refers to a data point that is significantly different from the others. In the context of fraud, this looks like behaviors (such as transactions) that diverge from an individual's usual behavior, which can be an indicator of fraudulent activity.

In addition to revealing suspicious or abnormal behavior in financial contexts, anomaly detection is vital for spotting rare events such as rare disease outbreaks or side effects in the medical domain, with vital applications in medical diagnosis. Examples of fraud can be the following:

- **Banking**: Credit Card Fraud, Identity Theft, Wire Transfer Fraud
- **E-commerce**: Payment Fraud, Account Takeover, Return Fraud
- **Insurance**: False Claims, Premium Diversion, Staged Accidents

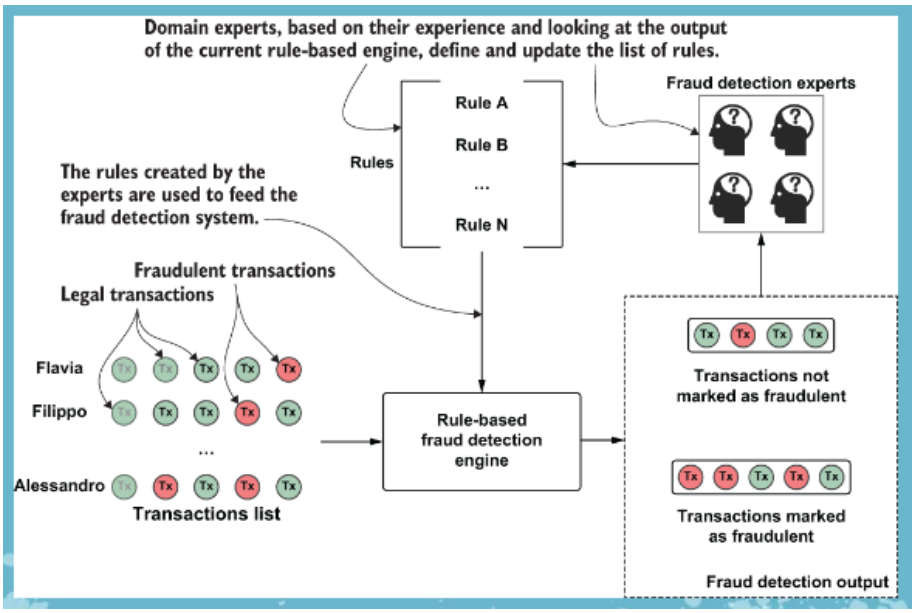
16.2 Graph Model for Fraud Detection

Fraud is an uncommon, well-considered, time-evolving, carefully organized and imperceptibly concealed crime that appears in many different types and forms. Fraud can be authorized (spoofing) or not authorized (stolen card).

Graphs provide a powerful modeling and analysis tool for capturing long-range correlations among interdependent data objects, which makes them well suited to the fraud-fighting scenario. Our bank account and credit card transactions follow a logic related to what we do, where we live, what we like to buy, and so on.

- **Fraud prevention** refers to measures that can be taken to prevent or reduce fraud. Each of prevention methods has drawbacks in terms of vulnerability, effectiveness, cost, and/or inconvenience for customers. A trade-off between pros and cons needs to be found.
- **Fraud detection** refers to the ability to recognize or discover fraud. It comes into play when fraud prevention has failed, but because it's not always obvious when that happens, fraud detection measures must be used all the time.

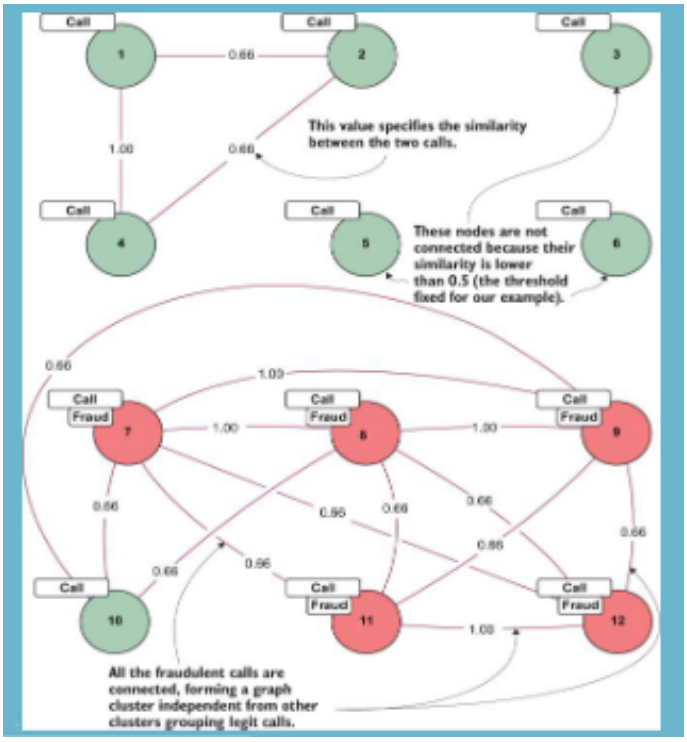
The simplest approach to fraud detection is build a rule-based engine. Domain experts, based on their experience and looking at the output, define and update the list of rules.



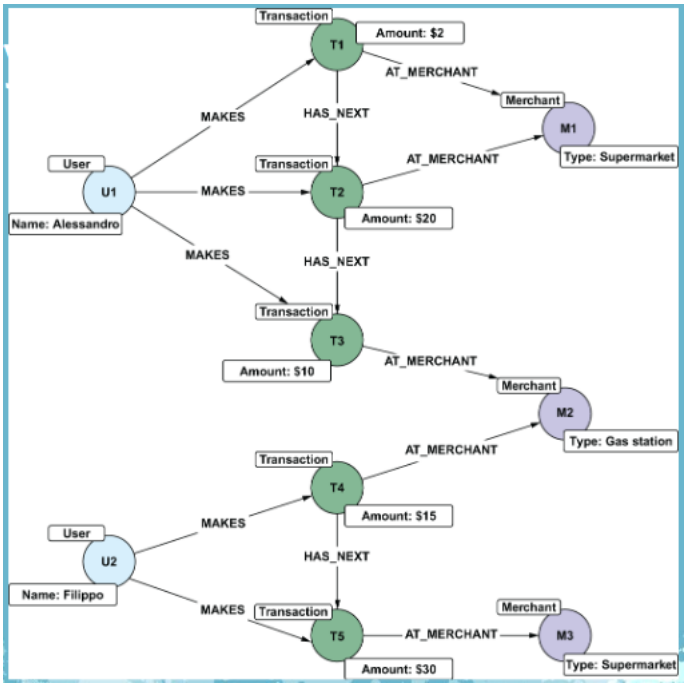
The main disadvantages of this approach are that it's not a scalable and dynamic solution; experts need time to change rules or discover new fraud.

Instead, graphs provide a powerful modeling and analysis tool for capturing long-range correlations among interdependent data objects, which makes them well suited to the fraud-fighting scenario.

Another use case can be anomaly detection for telecommunication, where is useful understand fraudulent calls. The fraudulent calls are closer to other fraudulent calls than they are to normal calls. Graph clustering algorithms such as community detection can be used to group fraudulent calls and then use such a model to classify new calls as fraudulent or not.



How can we use a graph model to discover the transactions where the fraud originated?



A more sophisticated approach will consider two sets of transactions:

- The global set containing data on all transactions, not only the transactions of people who

were the victim of fraud. This dataset is our **background set**. We will get statistics about the occurrences of merchants in this set and use that information in our formula.

- The set of recent transactions by people complaining about a fraudulent charge. This dataset is our **foreground set**, representing our real target. Our goal can be stated as follows: find merchants that are uncommonly common in the foreground dataset compared with the background dataset

$$score(merchantX) = \frac{foregroundPercentage(merchantX)^2}{backgroundPercentage(merchantX)} - foregroundPercentage(merchantX) \tag{16.1}$$

16.3 Fraud Ring

A common fraud is the fraud ring where accounts, their details, and the money create a circle around the same set of people. The steps of the particular fraud scheme are summarized in the following list:

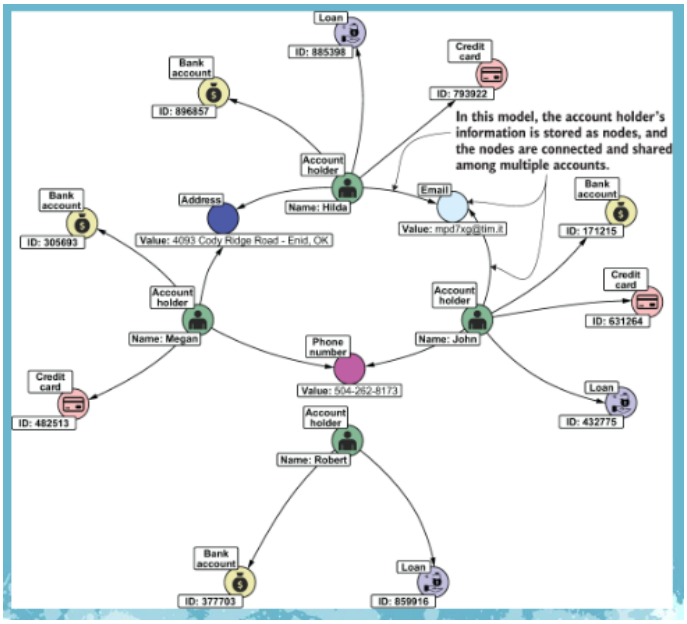
1. A large number of synthetic customer accounts is created at a financial institution (real accounts created for the purpose of committing the fraud).
2. For a long time, the accounts act like normal customers.
3. Over time, they request higher levels of credit, which they pay back on time, allowing them to gain credibility and trust with the bank.
4. In reality, the money is moved between the same set of accounts, using multiple and various hops to avoid recognition.
5. At some point, they all request the maximum credit they can get, take out the money, and disappear.

A graph to model this scenario is shown in the figure below.

Account ID	Username	Email	Phone number	Full name	Address
49295987202	alenegro	mpd7xg@tim.it	580-548-1149	Hilda J Womack	4093 Cody Ridge Road - Enid, OK
45322860293	jimjam	jam@mail.com	504-262-8173	Megan S Blubaugh	4093 Cody Ridge Road - Enid, OK
45059804875	drwho	mpd7xg@tim.it	504-262-8173	John V Danielson	4985 Rose Avenue - Mount Hope, WI
41098759500	robbob	bob@google.com	352-588-9221	Robert C Antunez	2041 Bagwell Avenue - San Antonio, FL

Before we can execute any graph algorithms, we first have to project an in-memory graph. The in-memory graph does not have to be an exact copy of the stored graph in the database. We have the ability to select only a subset of graph, or to project virtual relationships that are not stored in the database. After the in-memory graph is projected, we have can execute how many graph algorithms we want, and then either stream the results directly to the user, or write them back to the database. It's worth mentioning that the ring-detection approach can be used in contexts other than financial fraud, such as to identify multiple accounts belonging to the same user on a website. This problem is a common one in diverse scenarios, such as the following:

- Banned users trying to get new accounts.
- On an auction site, the same user bidding with multiple accounts to increase the price of an item.
- In a poker room, the same user playing with multiple accounts on the same table against other players.



- A merchant on a marketplace such as Amazon paying people or other companies for posting fraudulent reviews to increase their product's credibility.

17. Graphs for Text Mining

Search engines, recommendation systems, and vocal assistants use text for building a knowledge base:

- Recommendation engines use item descriptions to create the recommendation model.
- Search engines preprocess the documents via indexing.
- Chat bots and conversational agents create the knowledge base using unstructured data.

Interaction with the user, which in most cases happens via natural language.

17.1 Simplest Approach

The simplest approach to implement a tool that supports message writing, suggesting the next word while you are typing is the following:

1. Split the input into words by use a delimiter
2. **Tokenize** the word, so converting the sequence of text into smaller parts, known as tokens.
3. Store the extracted words in a way that keeps track of their order in the original text.

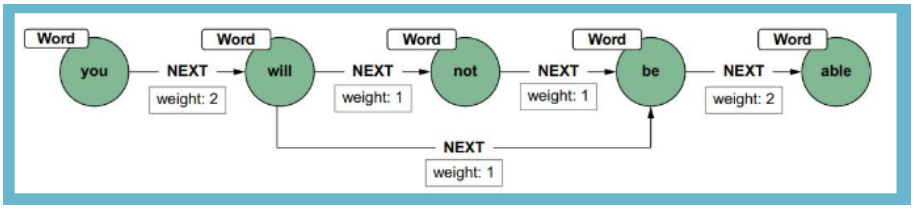


Figure 17.1: Weights are the number of times that a word follows the second one

Limitation of this approach is that the suggestions are based only on the current word because connections among words are lost and we don't have the context.

17.2 N-token Approach

1. Take last two/three words

2. Search in database for all sentences in which they appear in the same order.
3. List all the possible next words
4. Group words and count frequency for each word.
5. Order by frequency (descending)
6. Take the top three

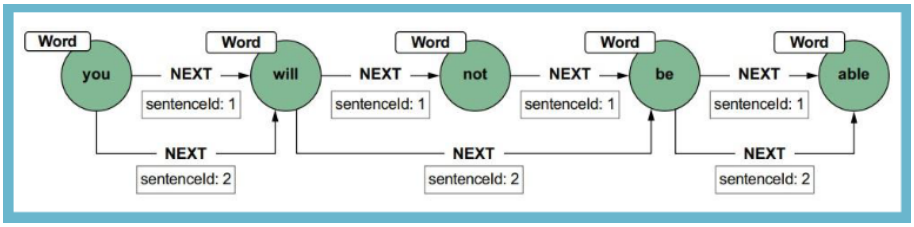


Figure 17.2: We have an id for the sentence that help us to understand the context

Also this approach is very simple and have to deal with a lot of trade-off:

- The words are not normalized to their base form.
- Dependencies among words (such as the connection between adjectives and nouns) are not taken into account.
- Some words make more sense together because they represent a single entity (Mario Rossi)
- The stop words are not properly identified.
- Splitting by using only whitespace typically is not good enough. Punctuation might be attached to the last word in a sentence.

17.3 Natural Language Processing (NLP)

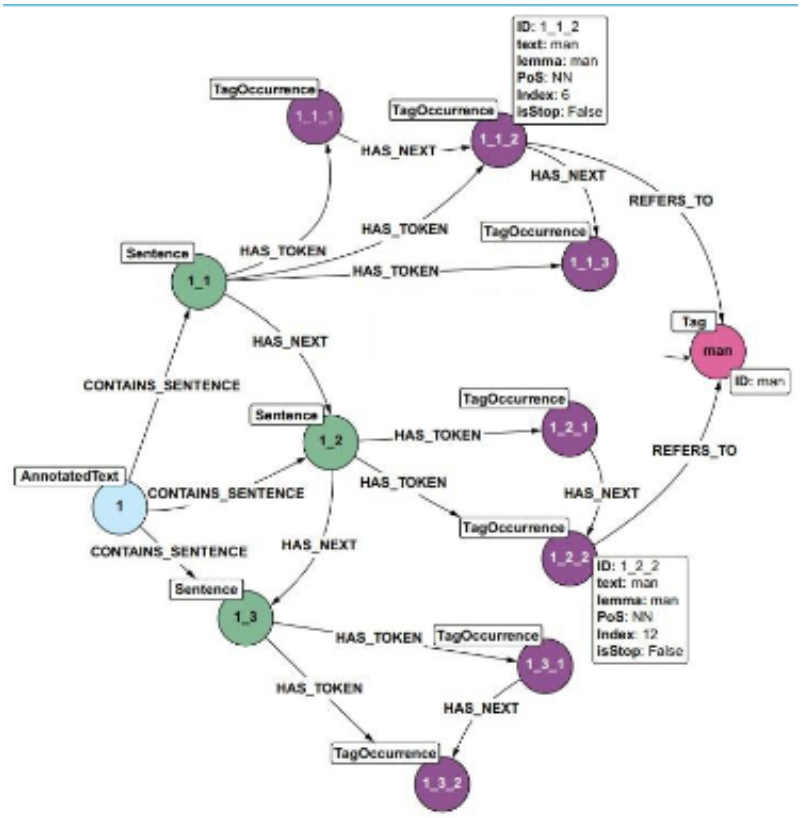
NLP is an interdisciplinary field that uses concepts from computer science, artificial intelligence, and linguistics with the aim of processing and analyzing large amounts of natural language data.

Following are some of the main tasks:

- Text classification: classify documents in different categories.
- Sentiment analysis: understand the judgement of a comment.
- Machine translation (google translate).
- Named entity recognition
- Topic modeling: understand topic of an article
- Text generation
- Summarization
- Question answering: extract answer from a text
- POS Tagging: grammatical analysis to extrapolate context from a phrase.

The first step of almost any IE process consists of breaking the content into small, usable chunks of text, called tokens. Then we need to remove the stop words as the,it,at,etc.. The last step is to do stemming (extrapolate origin of a word) and lemmatization (reduce words to canonic form). All this process can be modeled in a graph.

1. The **AnnotatedText** node represents the entry point. It acts as a proxy for the document, aggregating all the sentences from that document.
2. **CONTAINS_SENTENCE** relationships connect the AnnotatedText node to all the sentences in the document.
3. The **Sentence** nodes contain a reference to the sentence. This ID is obtained from the document ID and an incrementing number separated by "_".
4. **HAS_NEXT** relationships between sentences help us extract the order of sentences in the original text.



5. **HAS_TOKEN** relationships connect sentences to all the tokens they contain.
6. **TagOccurrence** nodes contain the tokens and all the information related to them (PoS, index in the document, and so on). The ID is obtained from the ID of the previous node in the hierarchy and a counter separated by "_".
7. **HAS_NEXT** relationships between TagOccurrence nodes help us extract the order of tokens in the original text, which is useful for identifying patterns in the phrases.
8. **Tag** nodes represent a lemmatized version of the token and are unique by lemma.
9. We can also memorize different word meanings or synonyms through a process of disambiguation.

Data produced by NLP tasks have a highly connected nature. Storing their results in a graph model appears to be a logical, rational choice.

- In some cases, as with syntactic dependencies, the relationships are generated as output of the NLP task, and the graph only has to store them.
- In other cases, the model has been designed to serve multiple scopes at the same time, providing easy-to-navigate data structures.

The graph model proposed here not only stores the main data and relationships extracted during the IE process, but also allows further extension by adding new information computed in a post-processing phase: similarity computation, sentiment extraction, and so on.

IV

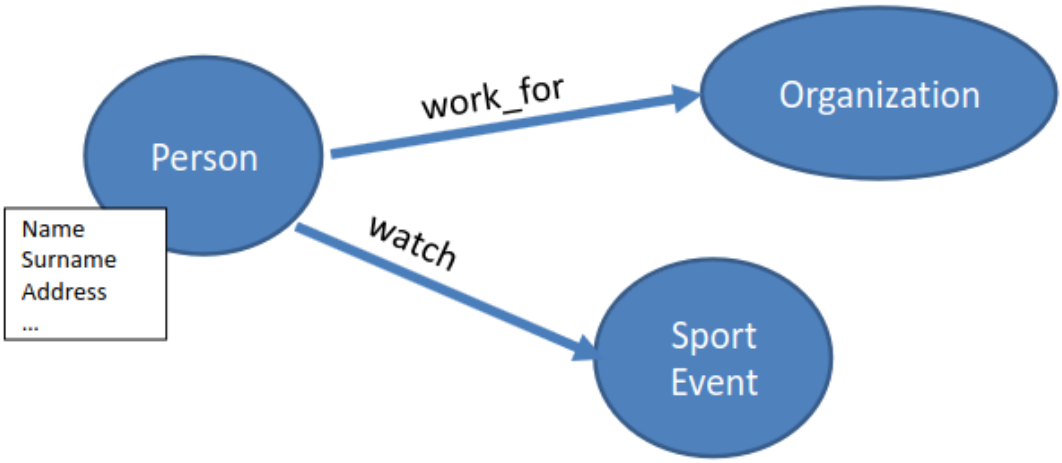
Knowledge graph

18. Knowledge Graph

18.1 Knowledge graph

Definition 18.1.1 — Knowledge graph. Knowledge graph is a knowledge base that uses a graph-structured data model or topology to integrate data. Knowledge graphs are often used to store interlinked descriptions of entities (objects, events, situations or abstract concepts), while also encoding the semantics underlying the used terminology.

What makes a normal graph a knowledge graph? "A knowledge graph consists of a set of interconnected typed entities and their attributes."



The main features of knowledge graph are the following:

- Consisting of concepts, classes, properties, relationships,and entity descriptions.
- Based on formal knowledge representations (RDF)

- Data can be open, private (sensitive information) or closed
- There are 3 type of data:
  1. Instance data
  2. Schema data (vocabularies, ontologies, knowledge)
  3. Metadata (provenance, versioning, licensing)
- Taxonomies are used to categorize entities
- Links exist between internal and external data

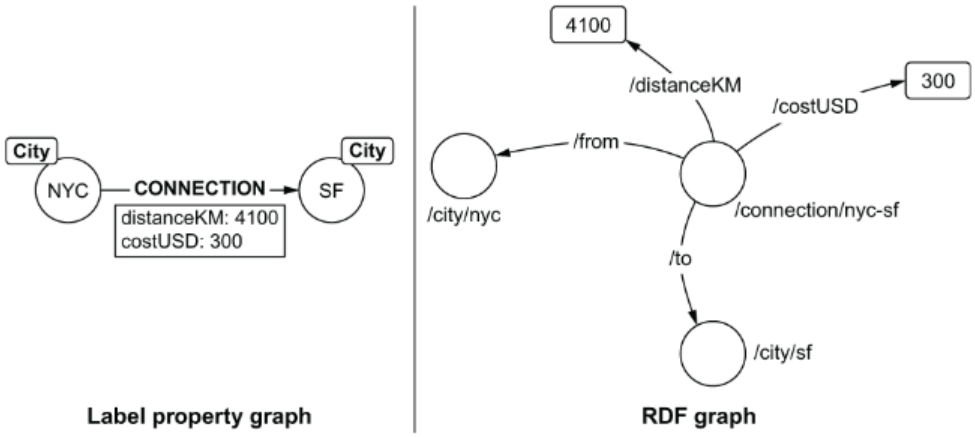
Compared with other knowledge-oriented information systems, knowledge graphs are distinguished by their particular combination of:

- Knowledge representation structures and reasoning, such as languages, schemas, standard vocabularies, and hierarchies among concepts
- Information management processes
- Accessing and processing patterns, such as querying mechanisms, search algorithms, and pre and postprocessing techniques

Knowledge graphs generally are represented with a Resource Description Framework (RDF) data model.

Definition 18.1.2 — RDF. RDF is a W3C standard for data exchange on the web, designed as a language for representing information about web resources. The underlying structure of any expression in RDF is a collection of triples, each consisting of a subject, a predicate, and an object. Each triple can be represented as a node-arc-node link, also called an RDF graph.

As shown in the example below differences between label property graphs and RDF Graph are all in how data are represented.



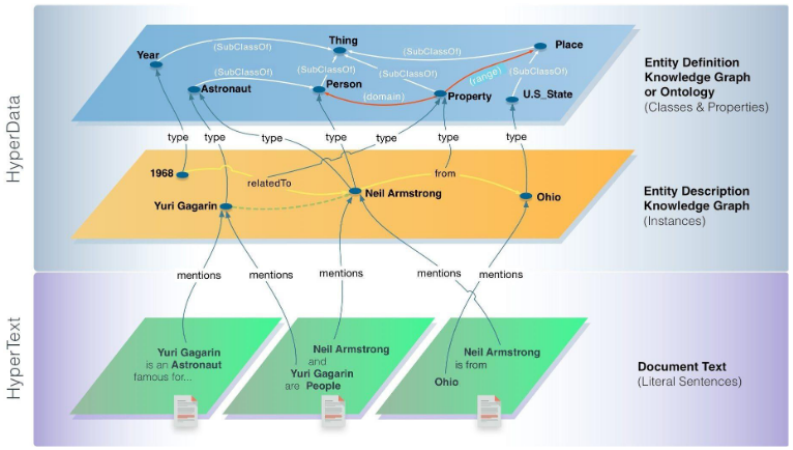
Continuing with Knowledge Graph we can add other information on it:

- It is structured (in the form of a specific data structure)
- It is normalised (consisting of small units, such as vertices and edges)

- It is connected
- It is explicit so it's created purposefully with an intended meaning
- It is annotated so enriched with contextual information to record additional details and meta-data
- It is a non-hierarchical structure

Thanks to figure below we can see that a knowledge graph is composed of two levels of astraction:

- **Level of Entity Definition** where ontology/semantics are defined.
- **Level of Instances**

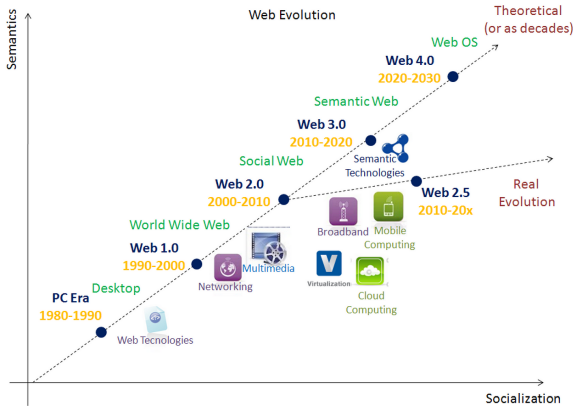


Through knowledge graphs we can connect concepts with and instances (in the same graph) and thanks to these links we can create the knowledge base.

18.2 History of Knowledge Graphs

How did we get to knowledge graphs?

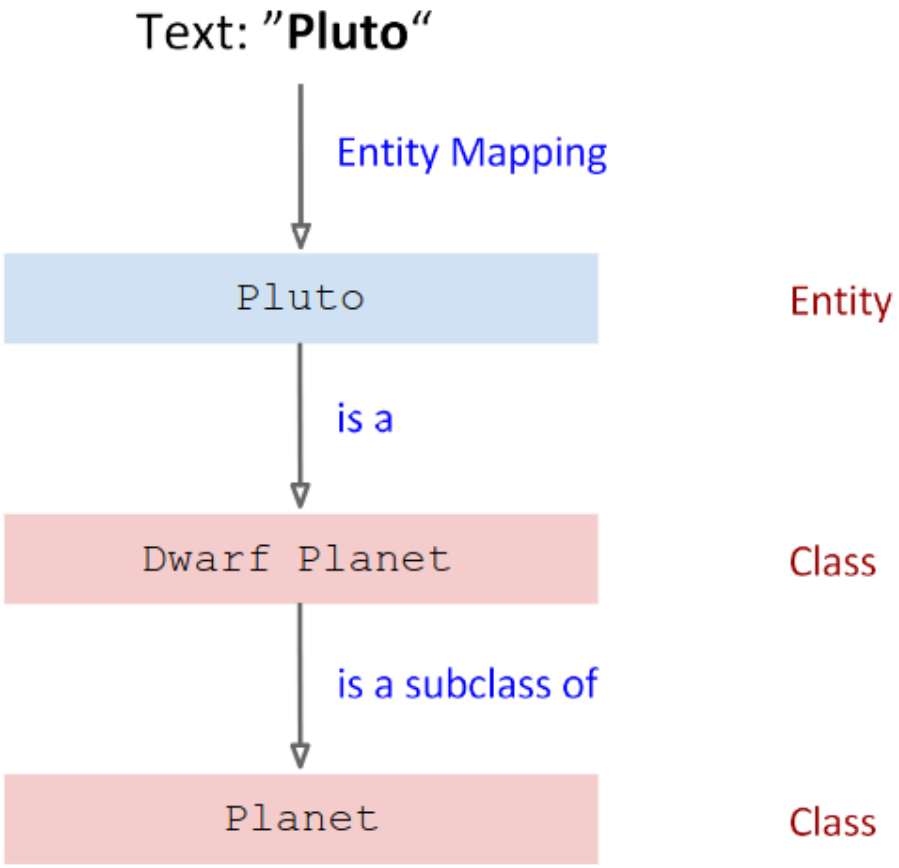
1. **1989-Web 1.0** refers to the first stage in the World Wide Web, which was entirely made up of web pages connected by hyperlinks
2. **1999-Web 2.0** is focused on the ability for people to collaborate and share information online via social media, wikis, blogging and Web-based communities.
3. **2009-Web 3.0**, the semantic web, exploits semantic Web technologies, distributed databases, natural language processing, machine learning, machine reasoning, and autonomous agents.
4. **2020-Web 4.0** connects all devices in the real and virtual world in real-time. It is more connected, open, and intelligent (Emotional Web).



Pay attention to not confuse NLP with Semantic Web. The first one is a technology that using

statistical models and machine learning, does Information Retrieval (It tries to understand automatically the semantic). Instead, Semantic Web uses content that is explicitly annotated with semantic data (linked data with URI). To understand better the concept consider the following example "On August 24th, 2006, the International Astronomical Union voted Pulto off the planetary list due to it's small size and irregular orbit". How can we connect to semantic data?

- **Disambiguation**: solution of linguistic ambiguities
- **Entity Mapping**: connect words to semantics
- **Logical Inference**: extract new knowledge from entity

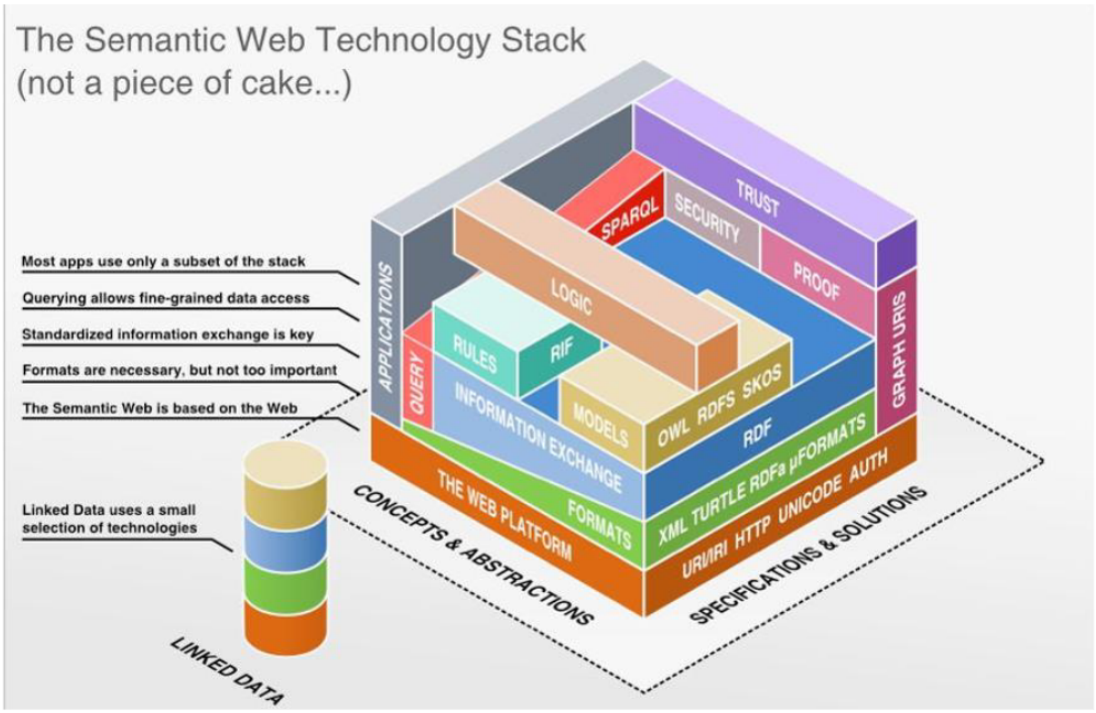


Thanks to this operation born Semantic Web or Web of data defined as an umbrella term for all the public data stored in databases on the Web that are linked together in some manner. To be included in the Semantic Web, a dataset must be publish according to the Linked Data principles, have resolvable URI, be publish in one of the popular RDF formats and links to a dataset that is already in the diagram.

18.3 Ontology

"Ontology is the philosophical study of the nature of being, existence, or reality, as well as the basic categories of being and their relations."

Definition 18.3.1 — Ontology. An ontology is an explicit, formal specification of a shared conceptualization. The term is borrowed from philosophy, where an ontology is a systematic account of existence. For AI systems, what 'exists' is that which can be represented.

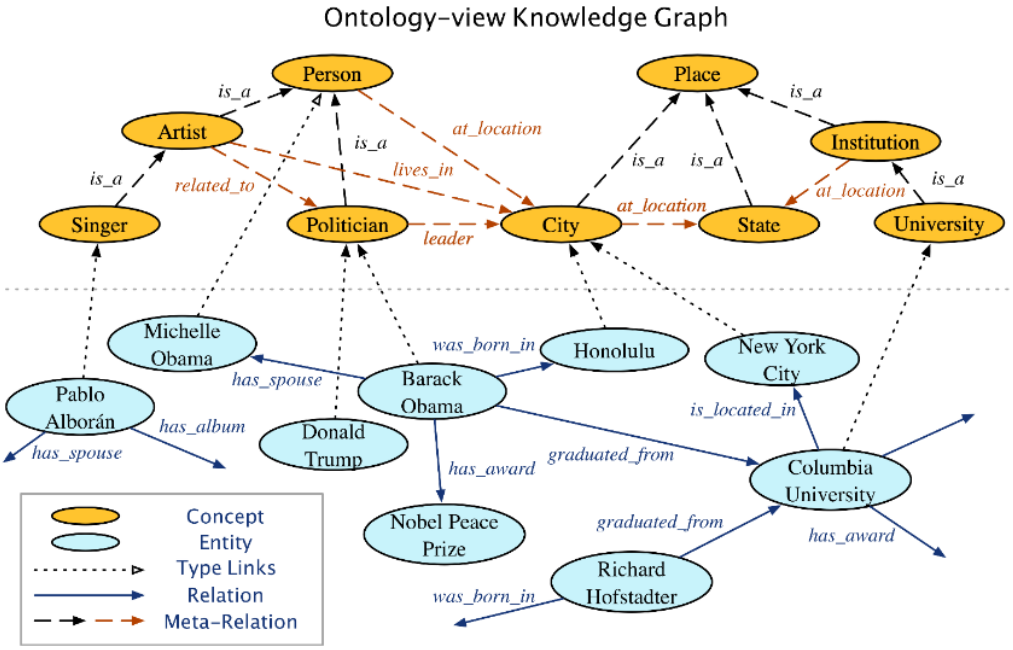


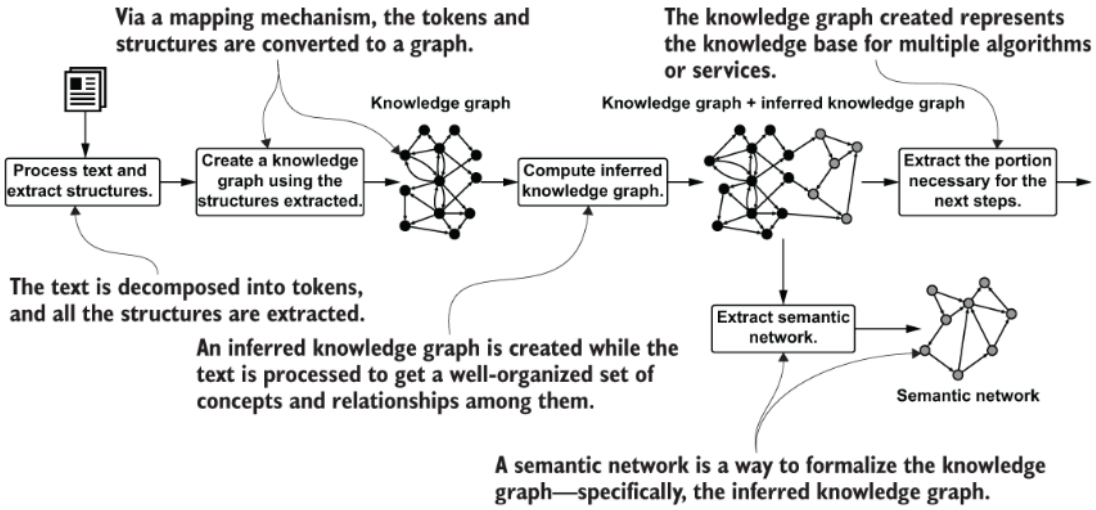
Corollary 18.3.1 Ontology is a conceptualization so defines an abstract model that must be explicit (meaning of all concepts must be defined), formal (machine understandable), shared.

Ontologies can be represented via Classes, Relations and Instances.

- Classes are abstract groups, sets, or collections of objects and represent ontology concepts. Classes are characterised via attributes that are name-value pairs.
- Classes can be related to other classes thanks to relations, special attributes, whose values are objects of (other) classes. For Relations and Attributes Rules (Constraints) can be defined that determine allowed/valid values.
- Instances describe individuals of an ontology.

To build a knowledge graph from a text, we need to infer a generic model that represents entities and relathionships in documents belonging to a domain-specific corpus, abstracting from the instances appearing in the text. This model is known as an inferred knowledge graph.


Ontology-view Knowledge Graph
Person
Artist
Singer
Politician
Place
City
State
Institution
University
is_a
at_location
lives_in
related_to
leader
Michelle Obama
Barack Obama
Honolulu
New York City
Pablo Alborán
Donald Trump
Columbia University
Nobel Peace Prize
Richard Hofstadter
was_born_in
has_spouse
has_album
has_award
graduated_from
is_located_in
Concept
Entity
Type Links
Relation
Meta-Relation
Instance-view Knowledge Graph


Via a mapping mechanism, the tokens and structures are converted to a graph.
The knowledge graph created represents the knowledge base for multiple algorithms or services.
Knowledge graph
Knowledge graph + inferred knowledge graph
Process text and extract structures.
Create a knowledge graph using the structures extracted.
Compute inferred knowledge graph.
Extract the portion necessary for the next steps.
The text is decomposed into tokens, and all the structures are extracted.
An inferred knowledge graph is created while the text is processed to get a well-organized set of concepts and relationships among them.
Extract semantic network.
Semantic network
A semantic network is a way to formalize the knowledge graph—specifically, the inferred knowledge graph.

V

Bibliography

Bibliography

References

[Po24] Laura Po. *Slides of Graph Analytics*. Unimore, 2024 (cited on page 11).