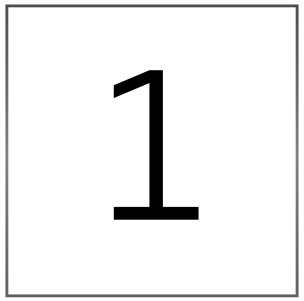
Graph Theory and Algorithms

Ph.D. Course – Marco Viviani

Graph Mining
(May 06, 2021 / 14:30-16:30)

TABLE OF CONTENTS



Basic Notions

Back to Data Mining and Intro
to Graph Mining

Data Mining

***Data mining**, also called **knowledge discovery in databases**, in computer science, the process of discovering interesting and useful patterns and relationships in large volumes of data.*

The field combines tools from statistics and artificial intelligence (such as neural networks and machine learning) with database management to analyze large digital collections, known as data sets.

Data mining is widely used in business (insurance, banking, retail), science research (astronomy, medicine), and government security (detection of criminals and terrorists).

Clifton, Christopher. "Data mining". Encyclopedia Britannica, 20 Dec. 2019, https://www.britannica.com/technology/data-mining. Accessed 9 May 2021.

Mining Frequent Patterns

- **Frequent Pattern**: a pattern (a set of items, subsequences, substructures, etc.) that occurs frequently in a data set.

- **Motivation**: Finding inherent regularities in data
  - What products were often purchased together?
  - What are the subsequent purchases after buying a PC?
  - What kinds of DNA are sensitive to this new drug?
  - Can we classify Web documents using frequent patterns?
  - …

Graph Mining: A Possible Definition

Graph mining *is the set of tools and techniques used to:*

(a) analyze the properties of real-world graphs,
(b) predict how the structure and properties of a given graph might affect some application, and
(c) develop models that can generate realistic graphs that match the patterns found in real-world graphs of interest.

Chakrabarti D. (2011) Graph Mining. In: Sammut C., Webb G.I. (eds) Encyclopedia of Machine Learning. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-30164-8_350

Graph Mining: Possible Research Areas

- **Cheminformatics (Chemical Informatics)**
- **Bioinformatics**
- **Computer Vision / Video Indexing**
- **Text Retrieval**
- **Web Analysis**
- **Social Network Analysis**
- **…**

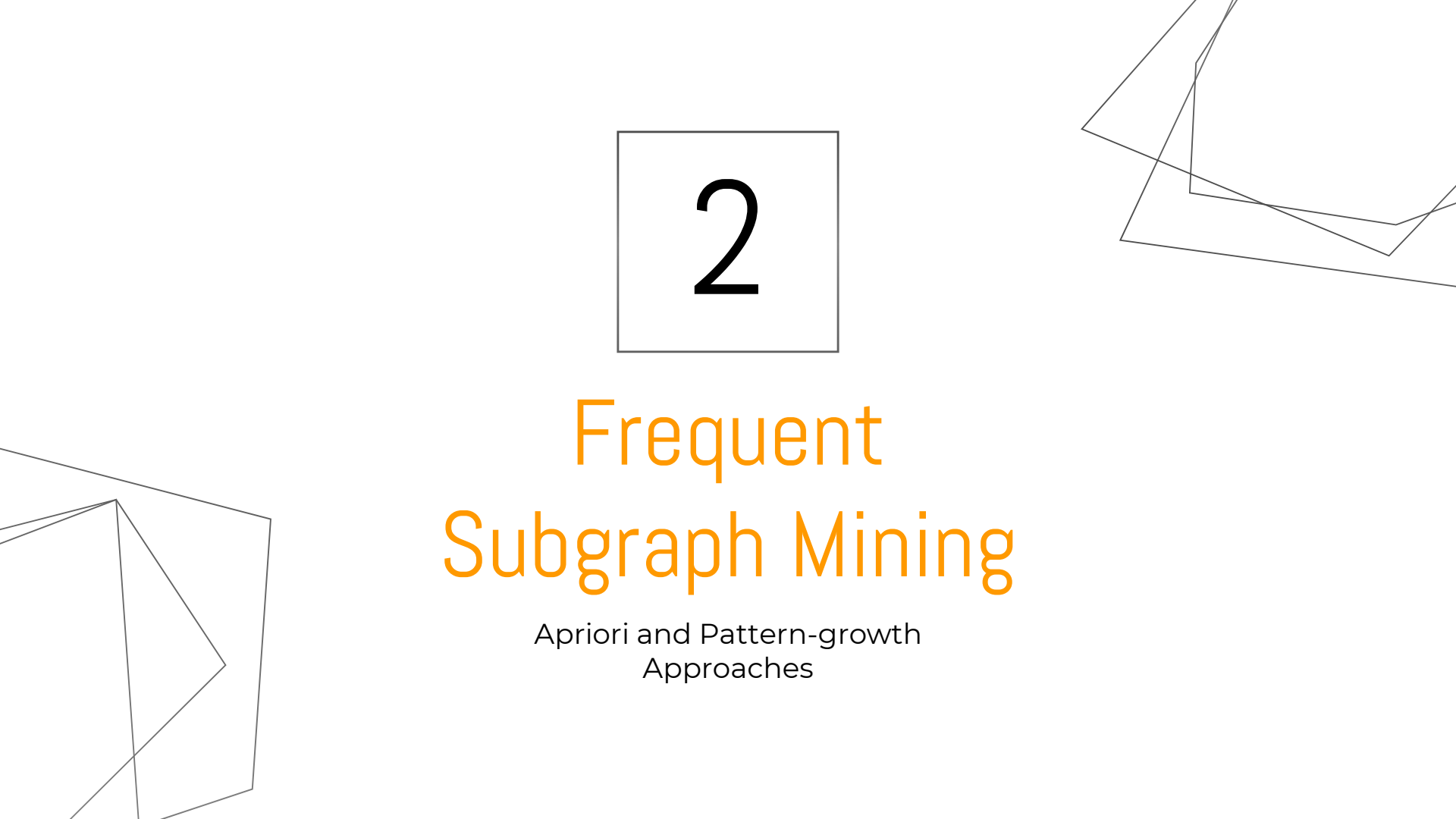
Graph Mining: Some Applications

- **Frequent Subgraph Mining** (This lecture)
- **Graph Indexing for Graph Search** (This lecture)
- **Graph Summarization** (Dedicated lecture)
- **Graph Classification** (High-level treatment in this lecture)
- **Graph Clustering / Partitioning / Community Detection** (Dedicated lecture)

Subgraph Matching (aka Isomorphism) Problem

- A graph g is a **sub-graph** of another graph g' if there exists a graph isomorphism* from g to g'.
- $support(g)$ (or, $frequency(g)$): number of graphs in $D = \{G_1, G_2, \dots, G_n\}$ where g is a sub-graph.

*For the definition of **graph isomorphism**, please refer to previous lectures.



Frequent Subgraph Mining

Apriori and Pattern-growth Approaches

Frequent Subgraphs

- A (sub)graph is **frequent** if its support in a given dataset is no less than a minimum support threshold.
- What is **support**? – intuitively the occurred frequency: the number of transactions containing a single occurrence.

Frequent Subgraphs … Cont'd

- Given a graph dataset $D = \{G_1, G_2, \dots, G_n\}$, find subgraph(s) G such that:

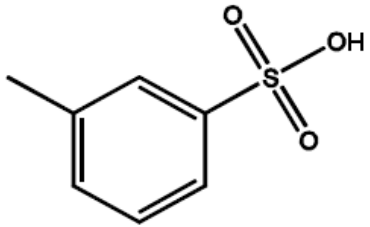
$$support(G) \geq minSup$$

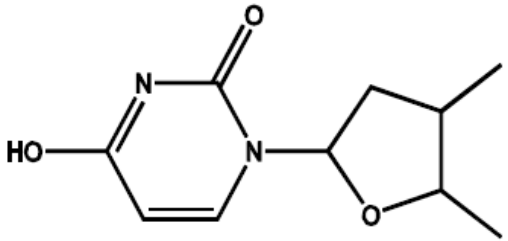
  where $support(G)$ is the frequency (or the percentage) of graphs in D containing G and $minSup$ is a selected threashold.

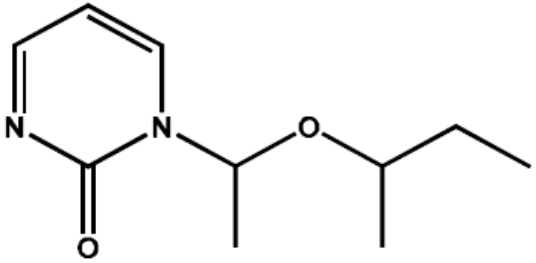
- **Frequent graph**: satisfies $minSup$ (a minimum support threshold).

Frequent Subgraphs (Example 1)

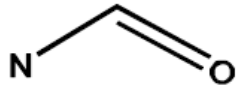
- Graph dataset (chemical compounds):

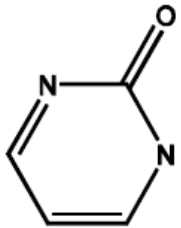






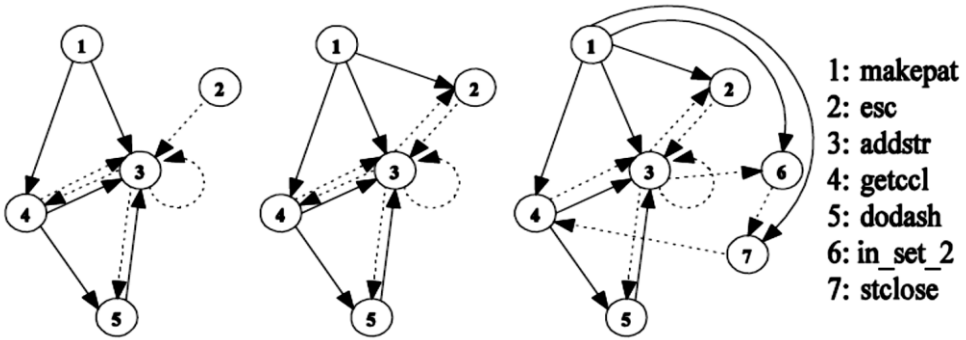
- Frequent subgraphs ($minSup = 2$):



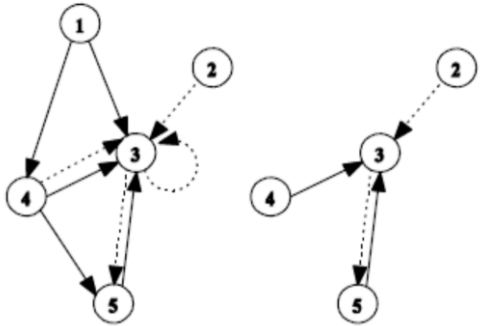


Frequent Subgraphs (Example 2)

- Graph dataset (Execution flow):



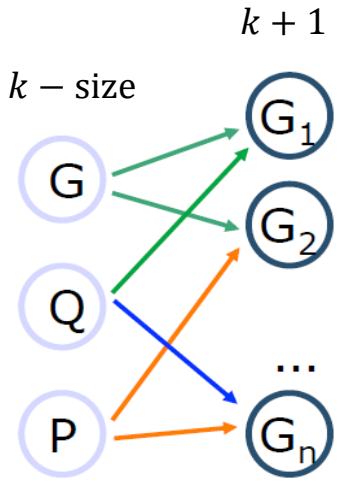
- Frequent subgraphs ($minSup = 2$):



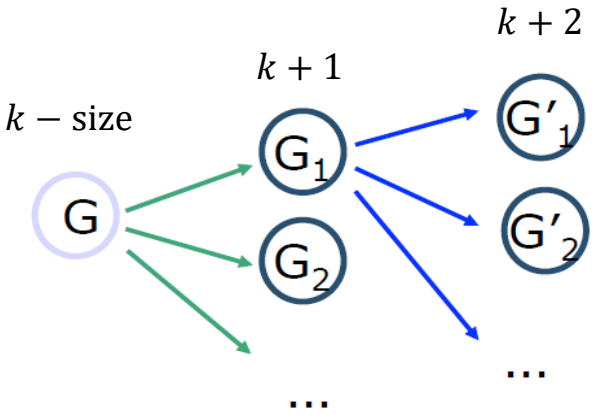
Some Approaches

- **Apriori-based approach**
  - Based on the merging (join) of sub-graphs.
- **Pattern-growth approach**
  - Based on the growth of sub-graphs.

Apriori vs Pattern-growth approach



join



grow

Apriori-Based Approach VS. Pattern-Growth Approach

Apriori-based Approach

- **Apriori graph**:
  - Level-wise mining method.
  - Size of new substructures is increased by 1.
  - Generated by joining two similar but slightly different frequent subgraphs.
  - Frequency is then checked.
- Start with a **graph of small size**.
- Generate candidates with **extra vertex/edge or path**.

Apriori-based Approach ... Cont'd

Algorithm: AprioriGraph. Apriori-based frequent substructure mining.

Input:

- D, a graph data set;
- *min_sup*, the minimum support threshold.

Output:

- S_k, the frequent substructure set.

Method:

$S_1 \leftarrow$ frequent single-elements in the data set;
Call AprioriGraph(D, *min_sup*, S_1);

Apriori-based Approach ... Cont'd

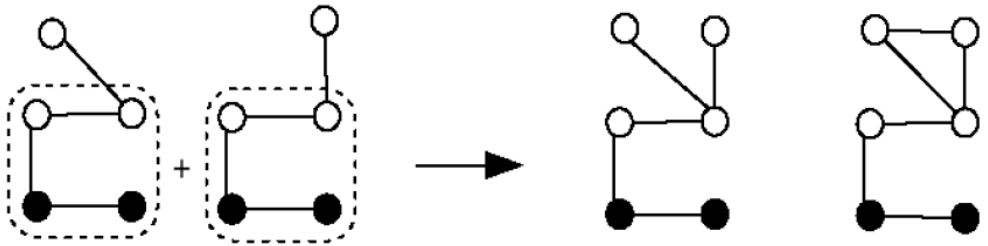
procedure AprioriGraph(D, *min_sup*, S_k)

(1) $S_{k+1} \leftarrow \varnothing$;
(2) **for each** frequent $g_i \in S_k$ **do**
(3) **for each** frequent $g_j \in S_k$ **do**
(4) **for each** size $(k+1)$ graph g formed by the merge of g_i and g_j **do**
(5) **if** g is frequent in D and $g \notin S_{k+1}$ **then**
(6) insert g into S_{k+1};
(7) **if** $s_{k+1} \neq \varnothing$ **then**
(8) *AprioriGraph*(D, *min_sup*, S_{k+1});
(9) **return**;

Vertex-based Candidate Generation

(AGM: Apriori-based Graph Mining)

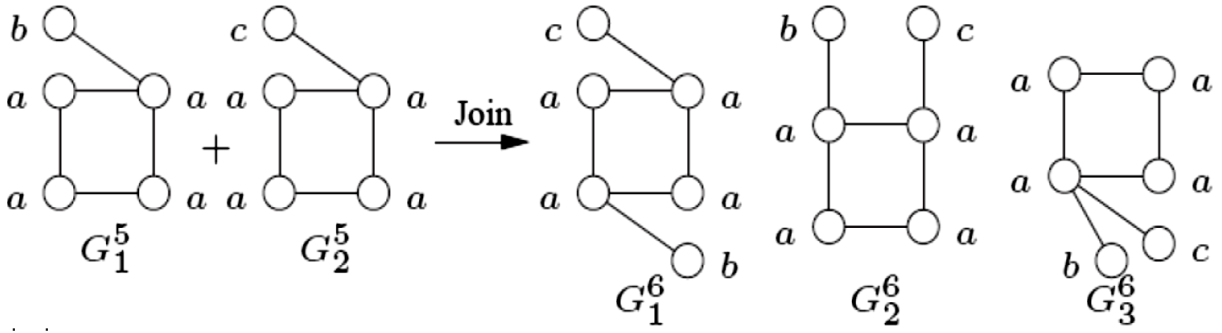
- Increases sub-structures size by **one vertex at each step**.
- Two frequent k-size graphs are merged only if they share the same $k-1$ subgraph (size: number of vertices).
- New candidate has $k-1$ sized component and the additional two vertices.
  - Two different sub-structures can be formed.



Edge-based Candidate Generation

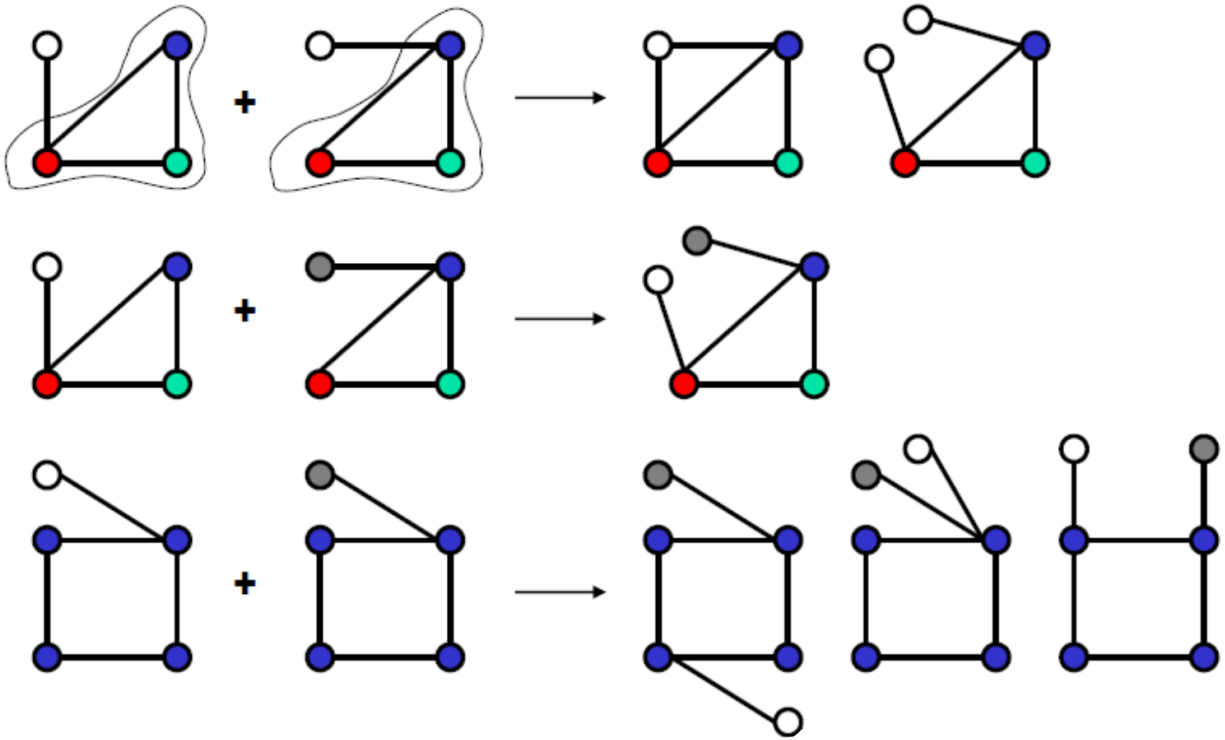
(FSG: Frequent Sub-Graph mining)

- Increases sub-structures size by **one-edge at each step**.
- Two frequent k-size graphs are merged only if they share the same subgraph having $k-1$ edges (core).
- New candidate – has core and the two additional edges.



Edge-based Candidate Generation … Cont'd

(FSG: Frequent Sub-Graph mining)



Edge Disjoint Path Method

- Classify graphs by number of **disjoint paths** they have.
- Two paths are edge-disjoint if they do not share any common edge.
- A substructure pattern with $k+1$ disjoint paths is generated by joining sub-structures with k disjoint paths.

A Parentheses: The Disjoint Paths Problem

- **Input**: A graph G with n vertices and m edges, k pairs of vertices $(s_1, t_1), (s_2, t_2), \dots, (s_k, t_k)$ in G.
- **Output**: (Vertex- or edge-) disjoint paths $p_1, p_2, \dots, p_k$ in G such that p_i joins s_i and t_i for $i = 1,2, \dots, k$.
- If k is a part of the input of the problem, this problem is known to be **NP-complete**.

Kawarabayashi, K. I. (2011, February). The disjoint paths problem: Algorithm and structure. In International Workshop on Algorithms and Computation (pp. 2-7). Springer, Berlin, Heidelberg

Disadvantages of Apriori Approaches

- **Overhead** when joining two sub-structures.
- Uses **BFS strategy**: level-wise candidate generation:
  - To check whether a $k+1$ graph is frequent, it must check all of its k-size subgraphs.
  - May consume more memory.

Pattern-growth Approach

- Uses **BFS** as well as **DFS**.
- A graph g can be **extended by adding a new edge** e. The newly formed graph is denoted by $g \diamond_x e$.
  - Edge e may or may not introduce a new vertex to g.
- **Pattern-growth approach**
  - For each discovered graph g, performs extensions recursively until all frequent graphs with g are found.
  - Simple but inefficient.
  - The same graph is discovered multiple times → duplicate graphs.

Pattern-growth Approach … Cont'd

Algorithm: PatternGrowthGraph. Simplistic pattern growth-based frequent substructure mining.

Input:

- g, a frequent graph;
- D, a graph data set;
- min_sup, minimum support threshold.

Output:

- The frequent graph set, S.

Method:

$S \leftarrow \varnothing$;
Call PatternGrowthGraph(g, D, min_sup, S);

Pattern-growth Approach … Cont'd

procedure PatternGrowthGraph(g, D, min_sup, S)

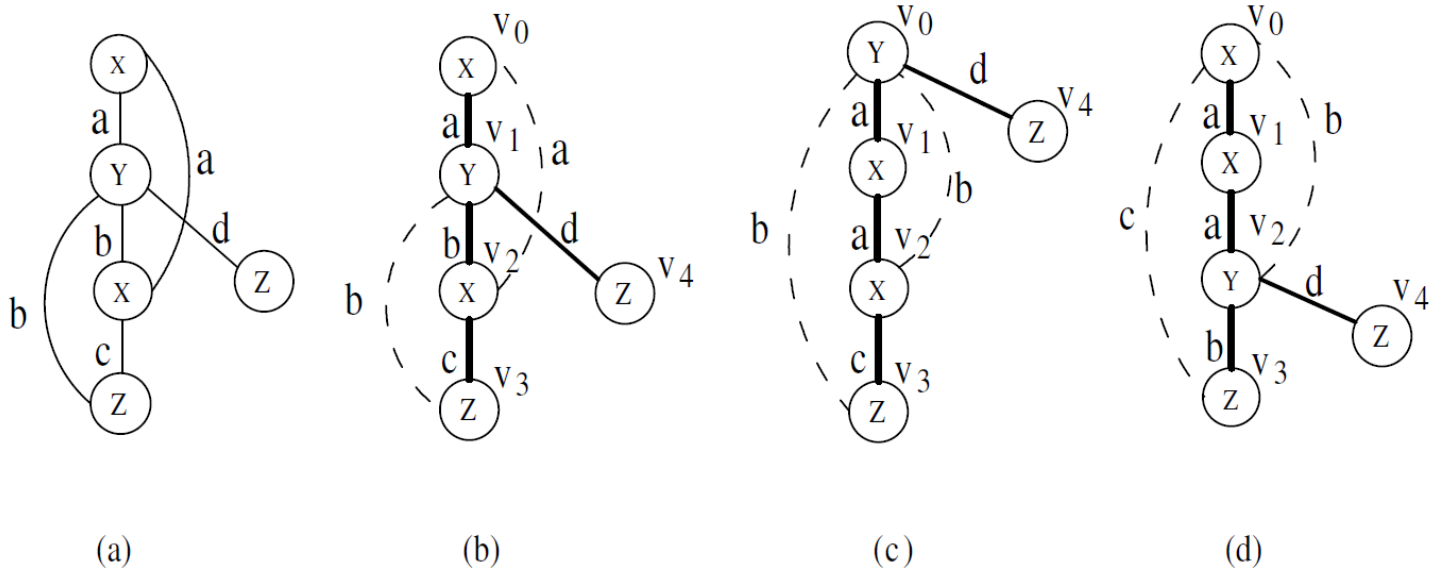
(1) **if** $g \in S$ **then return;**
(2) **else** insert g into S;
(3) scan D once, find all the edges e such that g can be extended to $g \diamond_x e$;
(4) **for each** frequent $g \diamond_x e$ **do**
(5) *PatternGrowthGraph*($g \diamond_x e$, D, min_sup, S);
(6) **return;**

gSpan

- **gSpan** adopts the **Depth-First Search (DFS)** strategy to mine frequent connected subgraphs efficiently.
- When performing a DFS in a graph, we construct a **DFS tree**.
- One graph can have **several different DFS trees**.

Yan, X., & Han, J. (2002, December). gspan: Graph-based substructure pattern mining. In 2002 IEEE International Conference on Data Mining, 2002. Proceedings. (pp. 721-724). IEEE.

gSpan ... Cont'd



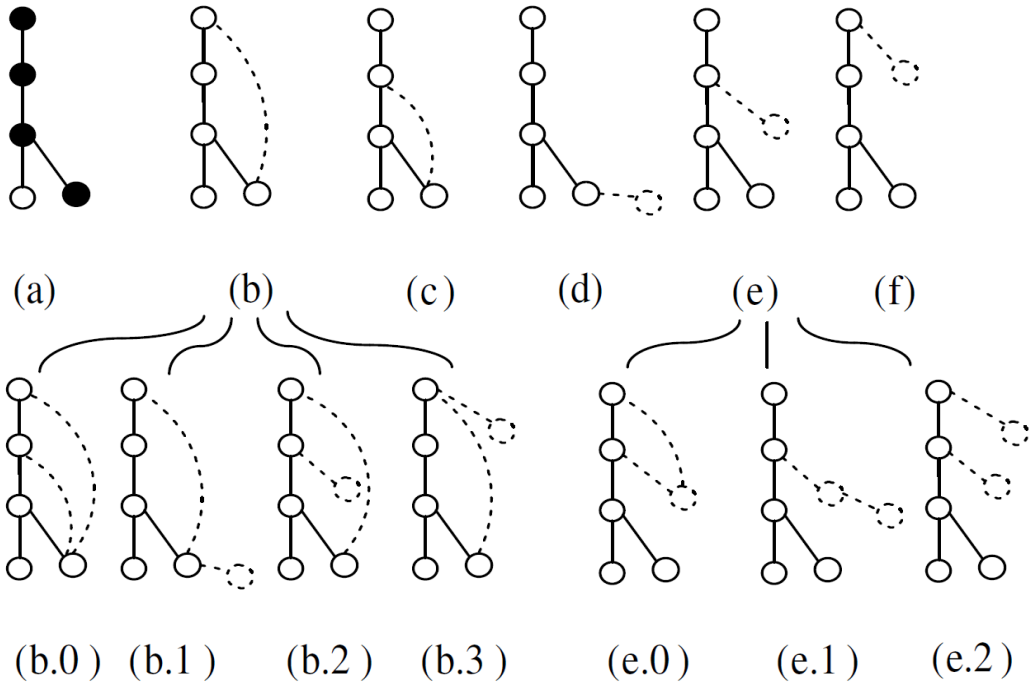
bold line - forward edge
dashed line - backward edge

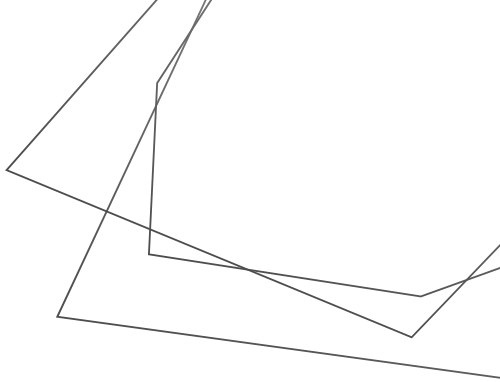
gSpan … Cont'd

gSpan **restricts the extension method**:

- either a new edge can be added between the right-most vertex and another vertex on the right-most path (backward extension);
- or it can introduce a new vertex and connect to a vertex on the right-most path (forward extension).
- Right-most extension, denoted by $g \diamond_r e$.

gSpan ... Cont'd

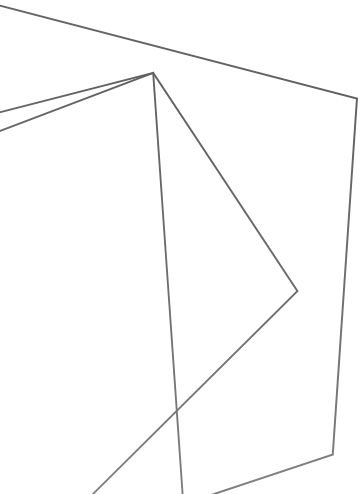




3

Graph Indexing

From IR to Graph Search
(Sub-/Super-/Similarity-graph Search)

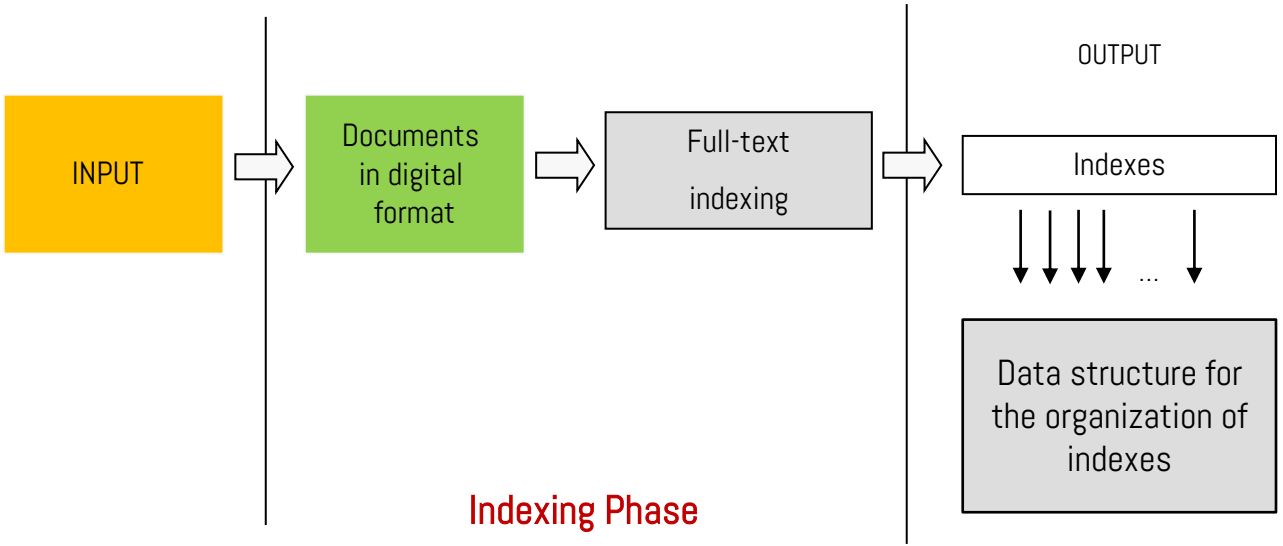


Indexing and IR

- **Indexing** is essential for efficient search and query processing.
- Fundamental in **Information Retrieval (IR)**.
- The purpose of storing an index in IR is to **optimize speed and performance** in finding relevant documents for a search query.
  - Without an index, the search engine would scan every document in the corpus, which would require considerable time and computing power.
- Indexing is an "**off-line**" phase.

Indexing and IR … Cont'd

- Scheme of automatic indexing process:



Indexing and IR: The Inverted File

Motivations

- The organization of the indexes in a “static” data structure, for example a matrix of document/term occurrences would be inefficient (forward index).
  - Since the matrix is sparse, space would be wasted to store also the “non-occurrence” of the terms (e.g., Term3 in Doc1, next slide).
  - Getting the list of documents that contain a specific term would be burdensome.

Indexing and IR: The Inverted File … Cont'd

DocID	Term1	Term2	Term3	Term4	Term5	Term6	Term7
Doc1	3	4	-	-	-	-	-
Doc2	-	-	1	2	6	5	8
Doc3	-	2	-	1	3	-	-
Doc4	-	-	-	-	4	7	6

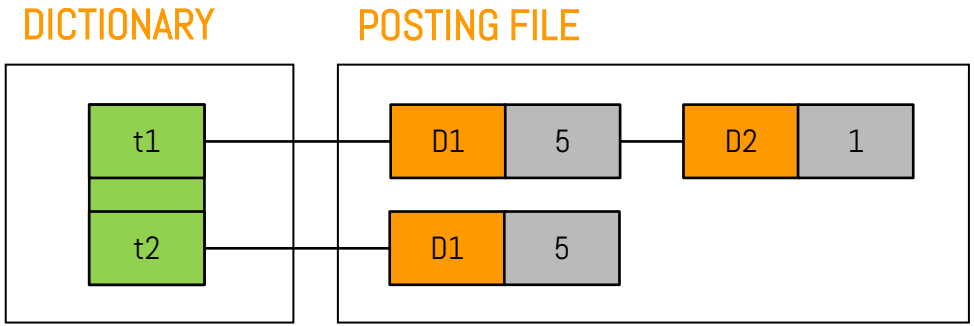
Indexing and IR: The Inverted File … Cont'd

- The table of occurrences must be inverted (term/document) and organized in a dynamic data structure.

Term	Doc1	Doc2	Doc3	Doc4
Term1	3	-	-	-
Term2	4	-	2	-
Term3	-	1	-	-
Term4	-	2	1	-
Term5	-	6	3	4
Term6	-	5	-	7
Term7	-	8	-	6

Indexing and IR: The Inverted File … Cont'd

- The index terms are organized in a dictionary. Each term "points" to a list containing the references to documents of which the term is an index.
- Use of two files: dictionary and posting file (it contains the posting lists of all index terms).



Graph Indexing

- Traditional approaches are not feasible for graphs.
- Graphs are complex structures.
- Queries in graph search are constituted by other graphs or patterns.
- There is the need of **suitable indexing techniques**.

Graph Indexing – Techniques

- Indexing based on nodes / edges / paths / sub-graphs.

- **Path-based Indexing Approach**
  - Enumerate all the paths in a database up to max length and index them
  - Index is used to identify all graphs with the paths in query
  - Not suitable for complex graph queries
  - Structural information is lost when a query graph is broken apart
  - Many false positives maybe returned

Graph Indexing – Techniques … Cont'd

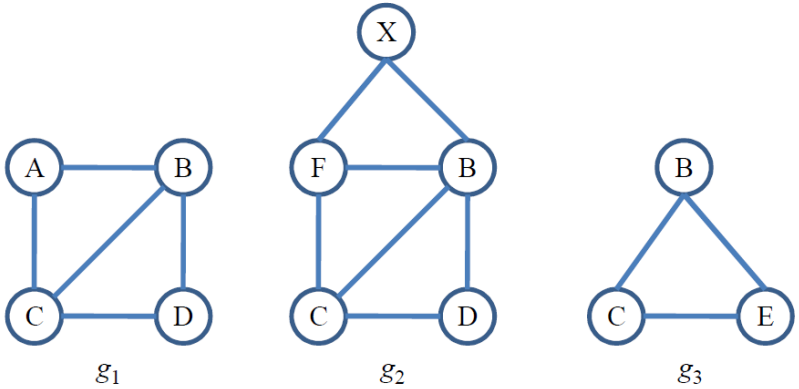
- **gIndex**: considers frequent and discriminative substructures as index features
  - A frequent substructure is discriminative if its support cannot be approximated by the intersection of the graph sets.
  - Achieves good performance at less cost.

Graph Queries

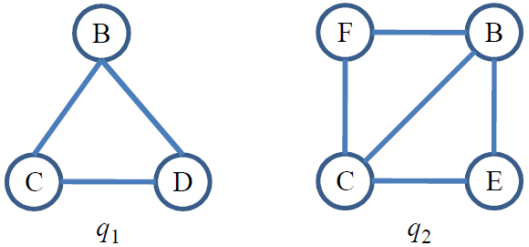
In principle, queries in graph datasets can be broadly classified into the following **three main categories**:

- **Subgraph queries**
  a. This category searches for a specific pattern in the graph dataset.
  b. The pattern can be either a small graph or a graph where some parts of it are uncertain.

Example

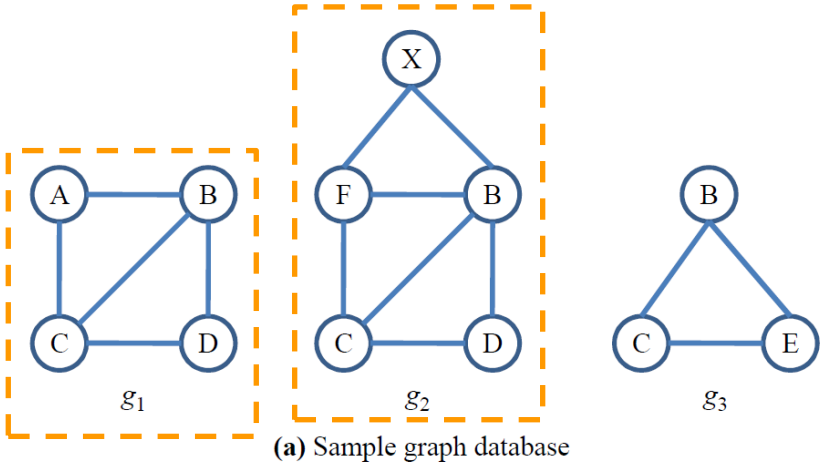


(a) Sample graph database

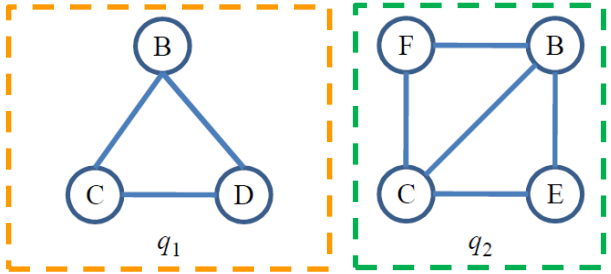


(b) Graph queries

Subgraph Queries



(a) Sample graph database

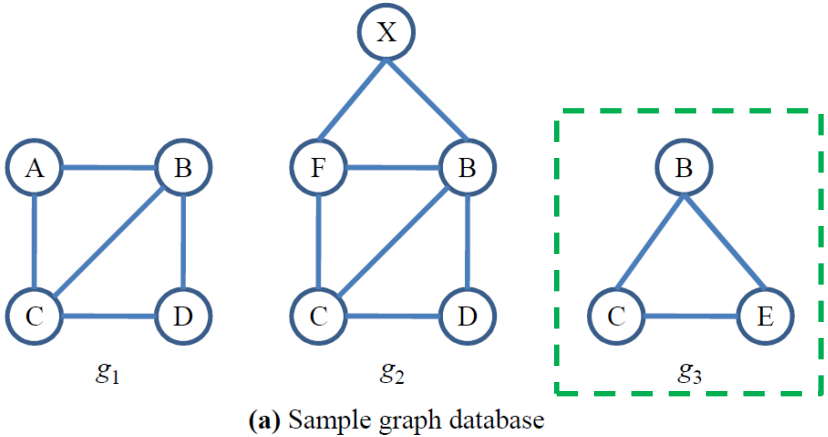


(b) Graph queries

Graph Queries … Cont'd

- **Supergraph queries**
  - This category searches for the graph database members of which their whole structures are contained in the input query.

Supergraph Queries



(a) Sample graph database

(b) Graph queries

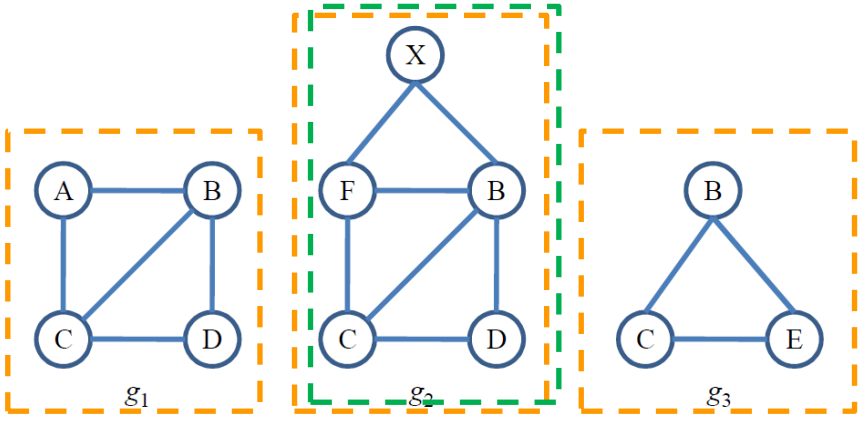
Graph Queries … Cont'd

- **Similarity (approximate matching) queries**
  - This category finds graphs which are similar, but not necessarily isomorphic to a given query graph.

- This category of queries can be further classified to the following two kinds of queries:
  - Substructure similarity search. These queries are used to discover all graphs that approximately contain the query graph.
  - Reverse similarity search. These queries are used to discover all graphs that are approximately contained by the query graph.

Similarity Queries

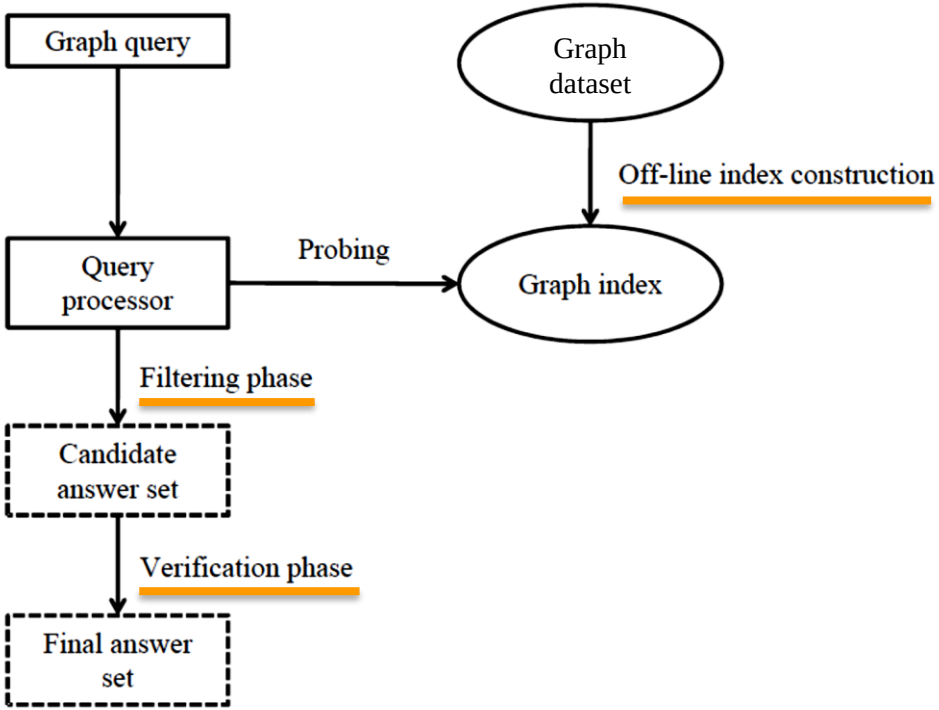
(Example with Substructure similarity search)



(a) Sample graph database

(b) Graph queries

Subgraph Query Processing



GraphGrep: A Path-based Indexing Approach

Three basic components:

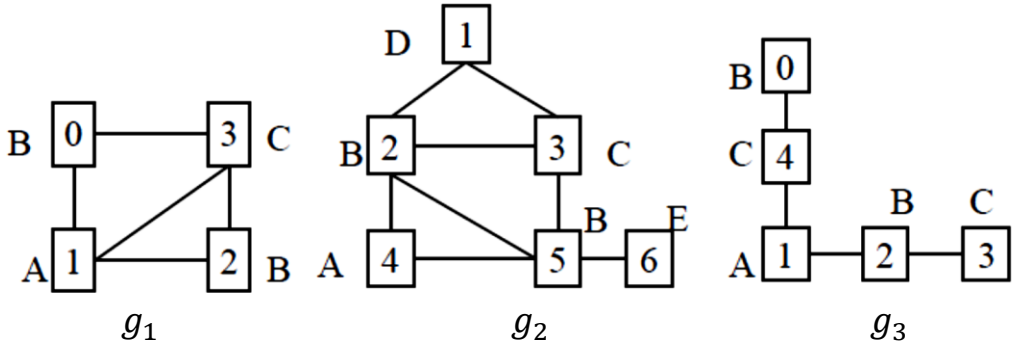
- **Build the index** to represent the dataset of graphs as sets of paths (this step is done only once).
- **Filter the dataset** based on the submitted query and the index to reduce the search space.
- Perform **exact matching**.

Shasha, D., Wang, J. T., & Giugno, R. (2002, June). Algorithmics and applications of tree and graph searching. In Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (pp. 39-52).

GraphGrep: Index Construction

- For each graph and for each node, find all paths that start at this node and have lenght one (single node) up to a (small) constant value.
- Because several paths may contain the same label sequence, we group the **id-paths** associated with the same label-path in a set.
- The «**path-representation**» of a graph is the set of **labeh-paths** in the graph, where each label-path has a set of id-paths.
- The **fingerprint** of the dataset is a table where each row contains the number of id-paths associated with a «key» in each graph.

GraphGrep: Example



g_1 g_2 g_3

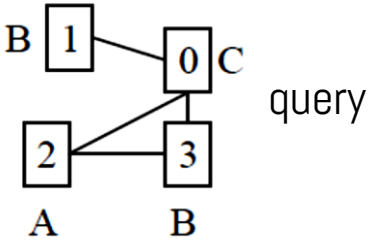
A={(1)} AB={(1, 0), (1,2)} AC ={(1, 3)} ACBA={…}
ABCA={(1 ,0 ,3 ,1),(1, 2, 3, 1)} CB={(3,0),(3,2)} C={(3)}
CBAB={((3, 0, 1 ,2),(3 ,2 ,1 ,0)} B={(0),(2)} BA={(0,1),(2,1)}
BAB={(0,1,2), (2,1,0)} ABC ={(1, 3, 0), (1,3,2)} ACB={…}
ABCB={…} BC={…} BAC={…} BCB={…} CBA={…}
BABC={…}CBAC={…} CABC={…} CAB={(3,1,0), (3,1,2)}
BACB={…}BCBA={…}BCAB={…} BCA={…} CA={(3,1)}

Key	g_1	g_2	g_3
h(CA)	1	0	1
……			
h(ABCB)	2	2	0

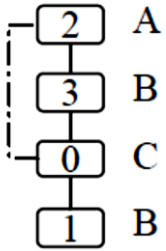
GraphGrep: Filtering the Dataset

- The **query graph** is parsed to build its fingerprint.
- The graph dataset is **filtered** by comparing the fingerprint of the query with the fingerprint of the dataset.
- A graph, for which at least one value in its fingerprint is less than the corresponding value in the fingerprint of the query, is discarded.

GraphGrep: Exact Matching



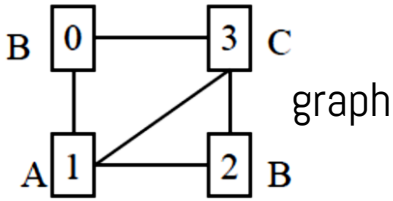
query



(b)

query DFS

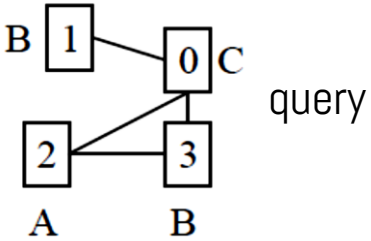
query fragments: ABCA - CB



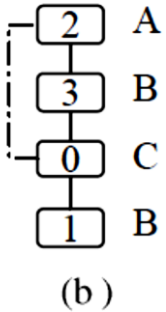
graph

- Select the set of paths in the graph matching the patterns of the query: ABCA = {(1,0,3,1), (1,2,3,1)}, CB = {(3,0),(3,2)}
- Combine any list from ABCA with any list of CB: ABCACB = {[(1,0,3,1),(3,0)], [(1,0,3,1),(3,2)], [(1,2,3,1),(3,0)], [(1,2,3,1),(3,2)]}

GraphGrep: Exact Matching … Cont'd

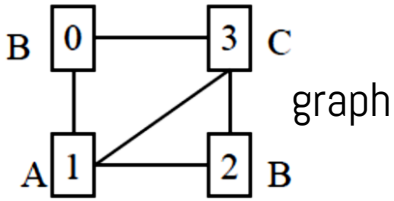


query



query DFS

query fragments: ABCA - CB



graph

- Remove lists from ABCACB if they contain equal id-nodes in non-overlapping positions. Query = ABCA - CB

ABCACB = [(1,0,3,1),(3,0)] ← NO
ABCACB = [(1,0,3,1),(3,2)] ← OK
ABCACB = [(1,2,3,1),(3,0)] ← OK
ABCACB = [(1,2,3,1),(3,2)] ← NO

Path-based Indexing Approaches: Drawbacks

- Not suitable for complex graph queries.
- Structural information is lost when a query graph is broken apart.
- Many false positives maybe returned.

gIndex

- Identify **frequent structures** in the dataset.
  - The frequent structures are subgraphs that appear often in the graph dataset.
- **Prune redundant frequent structures** to maintain a small set of discriminative structures.
- **Create an inverted index** between discriminative frequent structures and graphs in the dataset.

Yan, X., Yu, P. S., & Han, J. (2004, June). Graph indexing: A frequent structure-based approach. In Proceedings of the 2004 ACM SIGMOD international conference on Management of data (pp. 335-346).

gIndex: Frequent Fragments

- The number of graph structure is large.
  - Index only frequent subgraphs.
- $support(g)$
  - The number of graphs in D (graph dataset), where g is a subgraph.
- $minSup$
  - Minimum support threshold.
  - Index a fragment g only if $support(g) \geq minSup$.
- **Size-increasing support**
  - Low $minSup$ for small fragments, high $minSup$ for large fragments.

gIndex: Size-increasing Support

- **Size-increasing support**
  - Low $minSup$ for small fragments, high $minSup$ for large fragments.
- **Example**: a completely connected graph with 10 vertices, each of which has a distinct label. There are 45 1-edge subgraphs, 360 2-edge ones, and more than 1,814,400 8-edge ones.
- By enforcing the size-increasing support constraint, we bias the feature selection to small fragments with low minimum support and large fragments with high minimum support.

gIndex:
Redundant and Discriminative Fragments

- **Redundant fragment**
  - The indexed graphs by a fragment are also indexed by its subgraphs.
  - We don’t need to include redundant fragments.

- **Discriminative fragment**
  - Fragments which are not redundant.

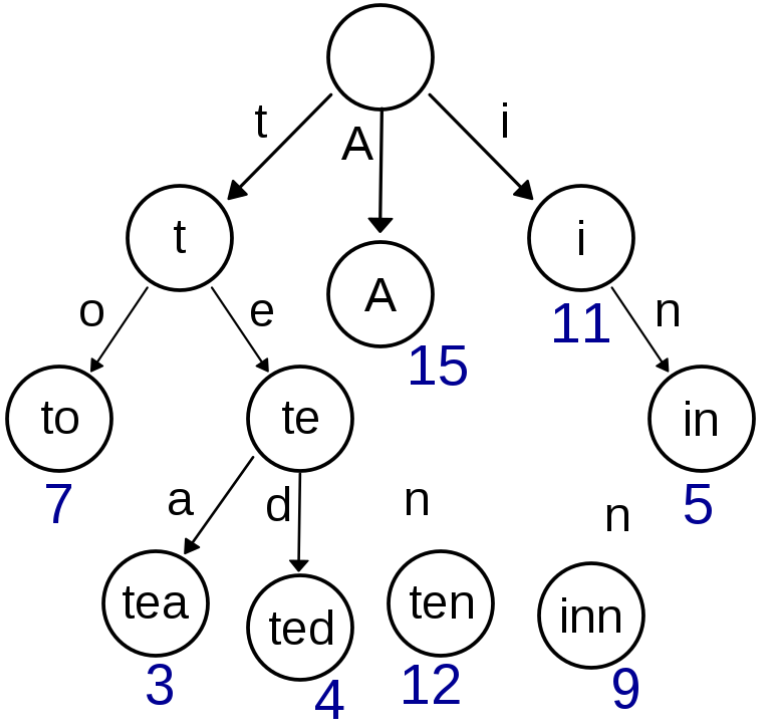
- **Details** on how to identify redundant and discriminative fragments are illustrated in: «Yan, X., Yu, P. S., & Han, J. (2004, June). Graph indexing: A frequent structure-based approach. In Proceedings of the 2004 ACM SIGMOD international conference on Management of data (pp. 335-346).»

gIndex: gIndex Tree

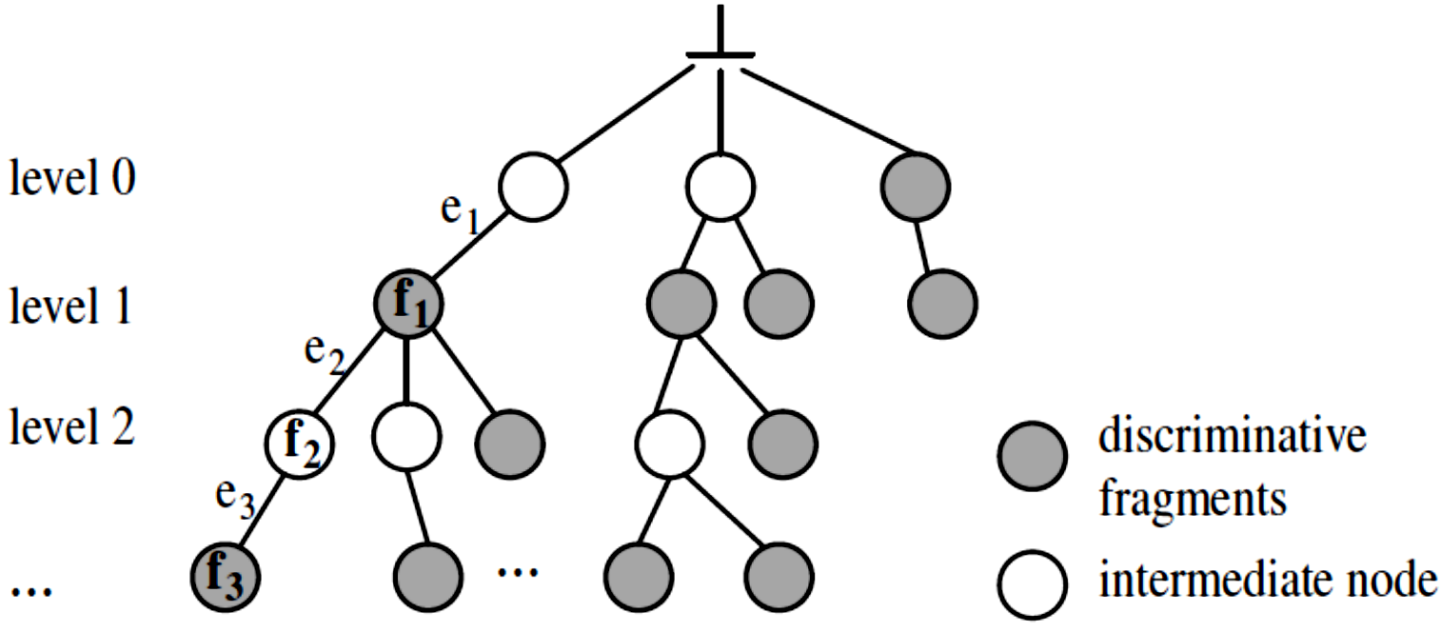
- Once discriminative fragments are selected, gIndex has efficient **data structures** to store and retrieve them.
- It translates fragments into sequences and holds them in a **prefix tree**.
- **Note**: In computer science, a **trie**, also called digital tree or prefix tree, is a type of search tree, a tree data structure used for locating specific keys from within a set.

Bodon, F., & Rónyai, L. (2003). Trie: an alternative data structure for data mining algorithms. Mathematical and Computer Modelling, 38(7-9), 739-751.

Example of a Trie



gIndex: gIndex Tree … Cont'd



gIndex: Searching

- Given a query q, gIndex enumerates all its fragments up to a maximum size and locates them in the index.

Algorithm 2 Search

Input: Graph database D, Feature set F, Query q, and Maximum fragment size $maxL$.
Output: Candidate answer set C_q.

1: let $C_q = D$;
2: **for each** fragment $x \subseteq q$ and $len(x) \leq maxL$ **do**
3: **if** $x \in F$ **then**
4: $C_q = C_q \cap D_x$;
5: **return** C_q;

Supergraph Query Processing

- Less investigated in the literature.
- It will not be detailed in this course.
- **Possibility of investigating this problem as a possible assignement.**

Similarity Query Processing

- A key question in graph similarity queries is on how to measure the **similarity** between a target graph member of the database and the query graph.
- It is difficult to give a precise definition of graph similarity.
- These approaches can allow for **node mismatches**, **node gaps** (gap node is a node in the query that cannot be mapped to any node in the target graph) as well as **graph structural differences**.
  - Such techniques are employed in the case of noisy or incomplete graphs.

Closure Tree

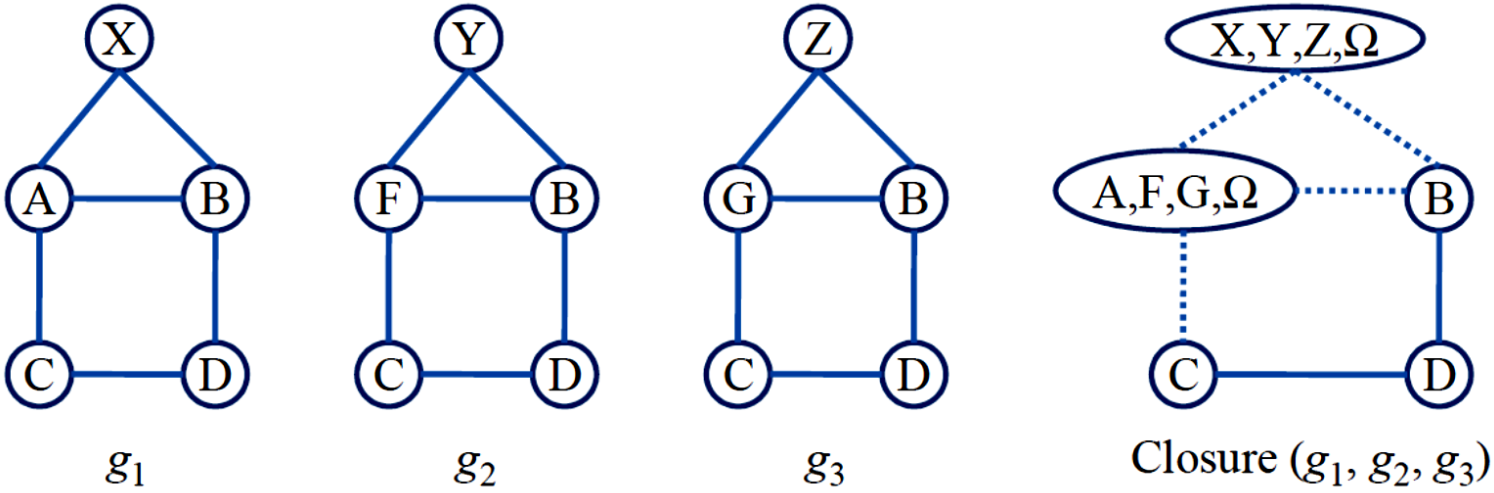
- A tree-based index structure named **CTree** (closure-tree).
- In this index structure, each node in the tree contains discriminative information about its descendants.
- The **closure of a set of vertices** is defined as a generalized vertex whose attribute is the union of the attribute values of the vertices.
- The **closure of a set of edges** is defined as a generalized edge whose attribute is the union of the attribute values of the edges.

He, H., & Singh, A. K. (2006, April). Closure-tree: An index structure for graph queries. In 22nd International Conference on Data Engineering (ICDE'06) (pp. 38-38). IEEE.

Closure Tree … Cont'd

- The **closure** of two graphs g_1 and g_2 under a mapping M is defined as a generalized graph $G_c = (V, E)$ where V is the set of vertex closures of the corresponding vertices and E is the set of edge closures of the corresponding edges. $G_c = closure(g_1, g_2)$.
- Hence, a graph closure has the same characteristics of a graph.
- The only difference is that the graph dataset member has singleton labels on vertices and edges while the graph closure can have multiple labels.
- For similarity queries, CTree defines graph similarity based on **edit distance**

Closure Tree: Example



g_1 g_2 g_3 Closure (g_1, g_2, g_3)

SAGA

- **SAGA: Substructure index-based Approximate Graph Alignment**

- The distance model contains **three components**:
  - The StructDist component measures the structural differences for the matching node pairs in the two graphs.
  - The NodeMismatches component is the penalty associated with matching two nodes with different labels.
  - The NodeGaps component is used to measure the penalty for the gap nodes in the query graph.

Tian, Y., Mceachin, R. C., Santos, C., States, D. J., & Patel, J. M. (2007). SAGA: a subgraph matching tool for biological graphs. Bioinformatics, 23(2), 232-239.

SAGA … Cont'd

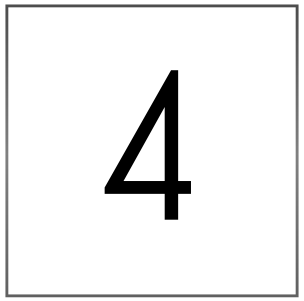
- **SAGA index** is built on small substructures of graphs in the dataset (fragment index).
- Each fragment is a set of k nodes from the graphs in the dataset. The index does not enumerate all possible k-node sets.
- The fragments in SAGA do not always correspond to connected subgraphs. The reason behind this is to allow node gaps in the matching process.
- A **DistanceIndex** is also maintained. This index is used to look up the precomputed distance between any pair of nodes in a graph.

SAGA … Cont'd

Graph matching process

- The query is broken into small fragments and the fragment index is probed to find database fragments that are similar to the query fragments.

- Also in this case, further details can be found in: “Tian, Y., Mceachin, R. C., Santos, C., States, D. J., & Patel, J. M. (2007). SAGA: a subgraph matching tool for biological graphs. Bioinformatics, 23(2), 232-239”.



Graph Databases

Basic Notions and
Representations

Basic Notions

- A **graph database** (GDB) is a database that uses graph structures for semantic queries with nodes, edges, and properties to represent and store data.

- The graph relates the data items in the store to a collection of nodes and edges, the edges representing the relationships between the nodes.

Basic Notions … Cont'd

- Graph databases **are a type of NoSQL database**, created to address the limitations of relational databases.
- While the graph model explicitly lays out the dependencies between nodes of data, the relational model and other NoSQL database models link the data by **implicit connections**.
- IMPORTANT: **Relationships** are a first-class citizen in a graph database and can be labelled, directed, and given properties.
  - This is not the case in other database management systems, where we have to infer connections between entities using things like foreign keys.

Basic Notions ... Cont'd

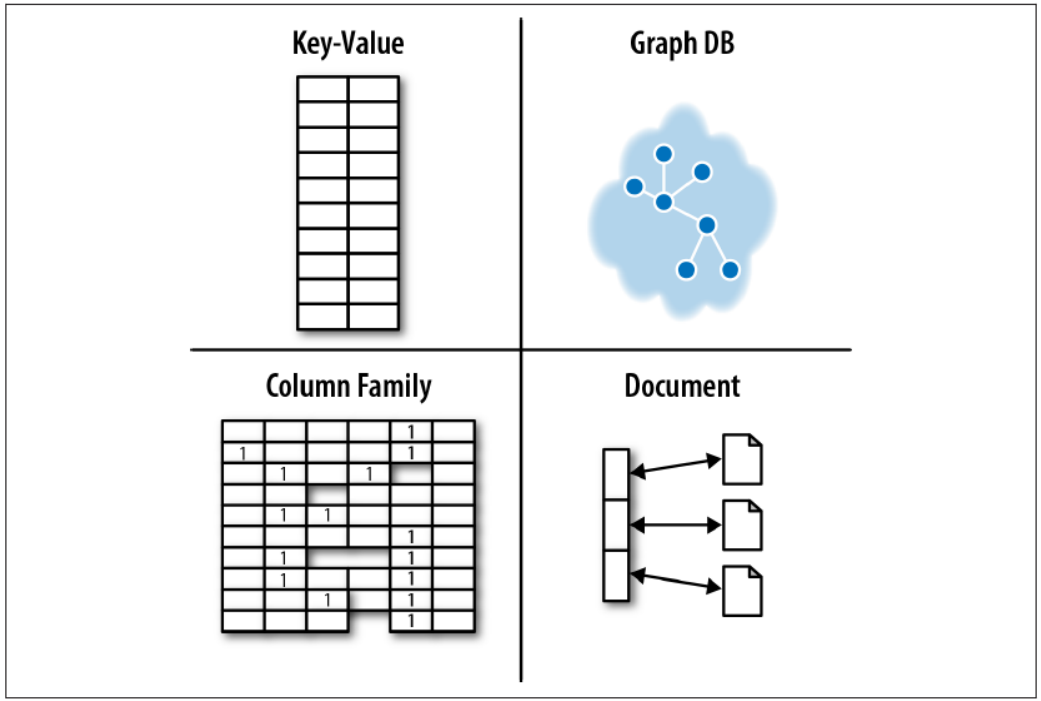
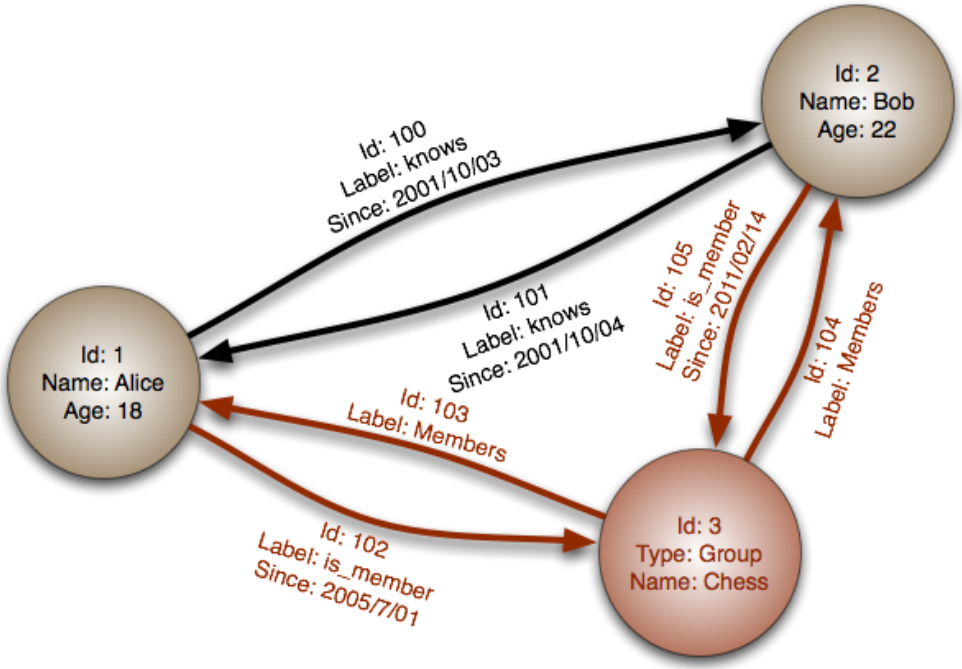


Figure A-1. The NOSQL store quadrants

Basic Notions: Example



Graph Database Management Systems

- A **graph database management system** (henceforth, a graph database) is an online database management system with Create, Read, Update, and Delete (CRUD) methods that expose a **graph data model**.

Graph Database Management Systems … Cont'd

There are **two properties** of graph databases we should consider when investigating graph database technologies:

- **The underlying storage**

  Some graph databases use native graph storage that is optimized and designed for storing and managing graphs. Not all graph database technologies use native graph storage, however.

- **The processing engine**

  Some definitions require that a graph database use index-free adjacency, meaning that connected nodes physically "point" to each other in the database.

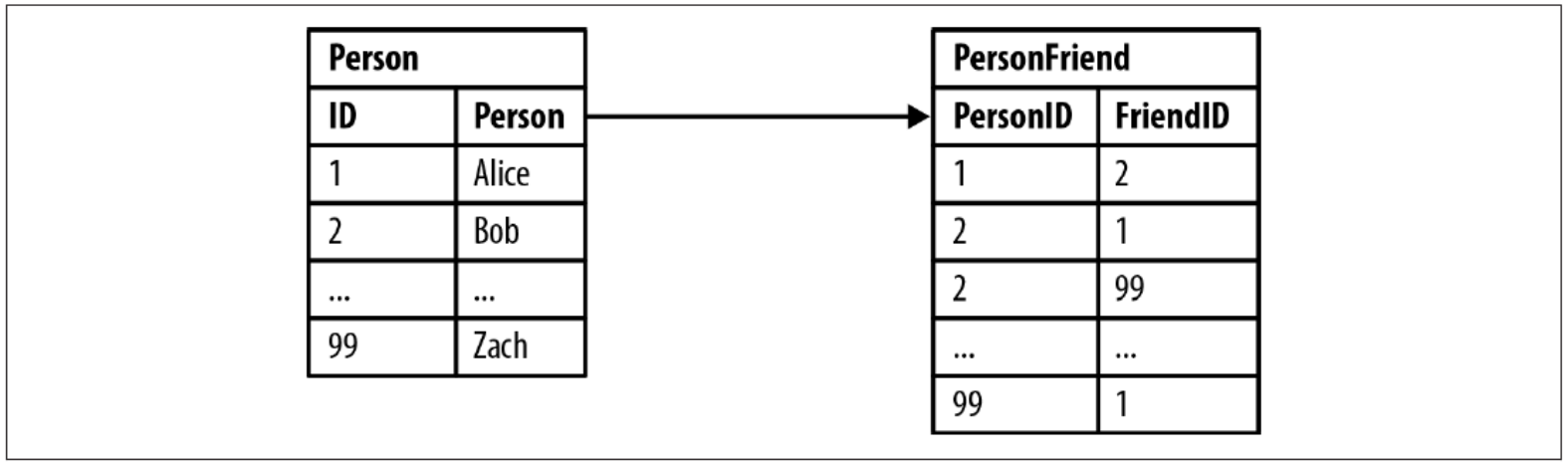
  A slightly broader view: any database that from the user's perspective behaves like a graph database (i.e., exposes a graph data model) qualifies as a graph database.

The Underlying Storage

- The **underlying storage mechanism** of graph databases can vary.
- Some depend on a relational engine and "store" the graph data in a **table**.
  - Although a table is a logical element, therefore this approach imposes another level of abstraction between the graph database, the graph database management system and the physical devices where the data is actually stored.
- Others use a **key–value store** or **document-oriented database** for storage, making them inherently NoSQL structures.

Relational Databases Lack Relationships

Figure 2-2 shows a simple join-table arrangement for recording friendships.



Person	
ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Figure 2-2. Modeling friends and friends-of-friends in a relational database

Relational Databases Lack Relationships – Example

Example 2-1. Bob's friends

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
  ON PersonFriend.FriendID = p1.ID
JOIN Person p2
  ON PersonFriend.PersonID = p2.ID
WHERE p2.Person = 'Bob'
```

Example 2-3. Alice's friends-of-friends

```
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN Person p1
  ON pf1.PersonID = p1.ID
JOIN PersonFriend pf2
  ON pf2.PersonID = pf1.FriendID
JOIN Person p2
  ON pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```

Relational Databases Lack Relationships … Cont'd

- Join tables add accidental complexity.
- Foreign key constraints add additional development and maintenance overhead just to make the database work.
- Several expensive joins are needed just to discover what a customer bought.
- Reciprocal queries are even more costly. "What products did a customer buy?" is relatively cheap compared to "which customers bought this product?" which is the basis of Recommender Systems.
- **We could introduce an index**, but even with an index, recursive questions such as "which customers buying this product also bought that product?" quickly become prohibitively expensive as the degree of recursion increases.

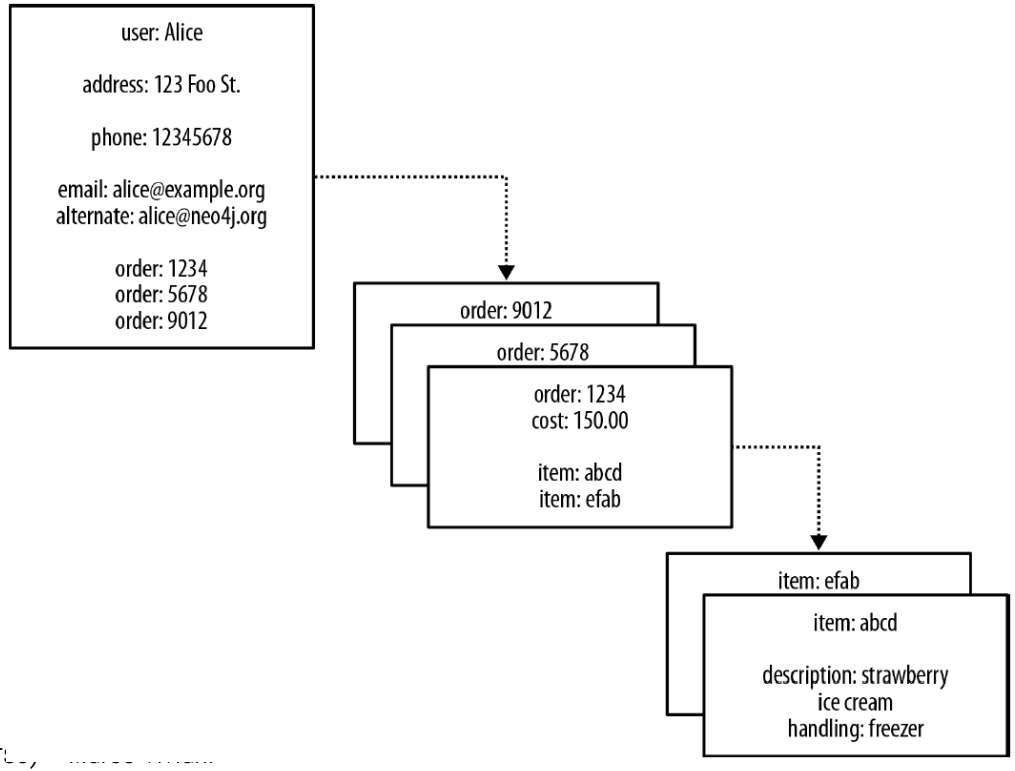
NOSQL Databases Also Lack Relationships

- Most NOSQL databases—whether key-value-, document-, or column-oriented—store sets of disconnected documents/values/columns.
- This makes it difficult to use them for connected data and graphs.

NOSQL Databases Also Lack Relationships … Cont'd

- One well-known strategy for adding relationships to such stores is to embed an **aggregate's identifier** inside the field belonging to another aggregate—effectively introducing foreign keys.
- But this requires joining aggregates at the application level, which quickly becomes prohibitively expensive.

NOSQL Databases Also Lack Relationships ... Example



Graph Databases Embrace Relationships

- In the graph world, connected data **is stored as connected data**.
- **Relationships** in a graph naturally form **paths**.
- Querying—or **traversing**—the graph involves following paths.
- Because of the fundamentally path-oriented nature of the data model, the majority of path-based graph database operations are highly aligned with the way in which the data is laid out, making them extremely efficient.

Graph Databases Embrace Relationships … Cont'd

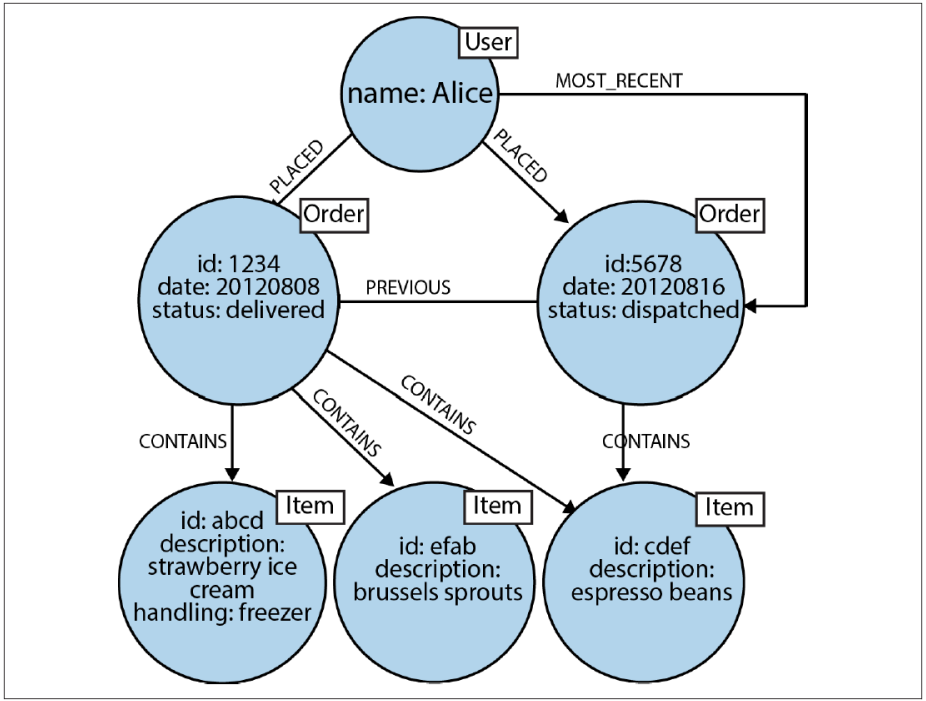


Figure 2-6. Modeling a user's order history in a graph

The Labeled Property Graph Model

- A *labeled property graph* is made up of *nodes*, *relationships*, *properties*, and *labels*.
- Nodes contain properties. Think of nodes as documents that store properties in the form of arbitrary key-value pairs. In Neo4j, the keys are strings and the values are the Java string and primitive data types, plus arrays of these types.
- Nodes can be tagged with one or more labels. Labels group nodes together, and indicate the roles they play within the dataset.

The Labeled Property Graph Model … Cont'd

- Relationships connect nodes and structure the graph. A relationship always has a direction, a single name, and a *start node* and an *end node*—there are no dangling relationships. Together, a relationship's direction and name add semantic clarity to the structuring of nodes.

- Like nodes, relationships can also have properties. The ability to add properties to relationships is particularly useful for providing additional metadata for graph algorithms, adding additional semantics to relationships (including quality and weight), and for constraining queries at runtime.

Querying Graphs

- Colloquially, we ask the database to “find things like this.”

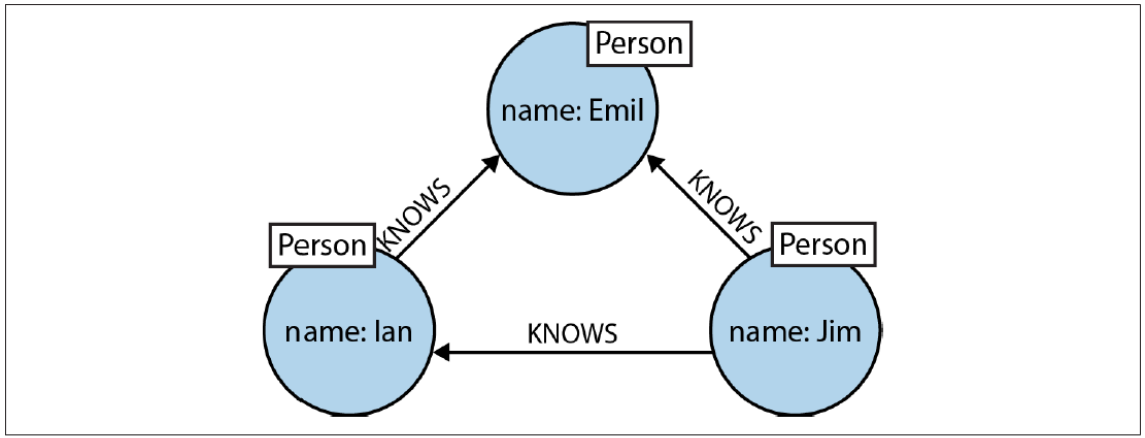


Figure 3-1. A simple graph pattern, expressed using a diagram

From: https://neo4j.com/blog/why-database-query-language-matters/

Querying Graphs – Example (Cypher)

- If we want to express the pattern of this basic graph in Cypher, we would write:

```
(emil)<-[:KNOWS]-(jim)-[:KNOWS]->(johan)-[:KNOWS]->(emil)
```

- This Cypher statement describes a path which forms a triangle that connects an node we call `jim` to the two nodes we call `johan` and `emil`, and which also connects the `johan` node to the `emil` node. As you can see, Cypher naturally follows the way we draw graphs on the whiteboard.

- Now, while this Cypher pattern describes a simple graph structure it doesn't yet refer to any particular data in the database. To bind the pattern to specific nodes and relationships in an existing dataset we first need to specify some property values and node labels that help locate the relevant elements in the dataset.

https://neo4j.com/developer/cypher/

From: https://neo4j.com/blog/why-database-query-language-matters/

Querying Graphs – Example … Cont'd

- Here's our more fleshed-out query:

```
(emil:Person {name:'Emil'})
     <-[:KNOWS]-(jim:Person {name:'Jim'})
     -[:KNOWS]->(johan:Person {name:'Johan'})
     -[:KNOWS]->(emil)
```

- Here we've bound each node to its identifier using its `name` property and `Person` label. The `emil` identifier, for example, is bound to a node in the dataset with a label `Person` and a `name` property whose value is `Emil`. Anchoring parts of the pattern to real data in this way is normal Cypher practice.

From: https://neo4j.com/blog/why-database-query-language-matters/

Querying Graphs – Example … Cont'd

- The simplest queries consist of a `MATCH` clause followed by a `RETURN` clause. Here's an example of a Cypher query that uses these three clauses to find the mutual friends of a user named `Jim`:

```
MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b)-[:KNOWS]->(c),
      (a)-[:KNOWS]->(c)
RETURN b, c
```

The Processing Engine

- **Native approaches:**
  - Index-free adjacency.
- **Non-native approaches**
  - Any database that from the user's perspective behaves like a graph database (i.e., exposes a graph data model) qualifies as a graph database.

Index-free Adjacency

- A graph database has native processing capabilities if it exhibits a property called **index-free adjacency**.
- A database engine that utilizes index-free adjacency is one in which each node maintains **direct references to its adjacent nodes**.
- Each node, therefore, acts as a **micro-index** of other nearby nodes, which is much cheaper than using global indexes.

Global Indexing

- A non-native graph database engine, in contrast, uses **(global) indexes** to link nodes together.

Graph Database Internals: Example

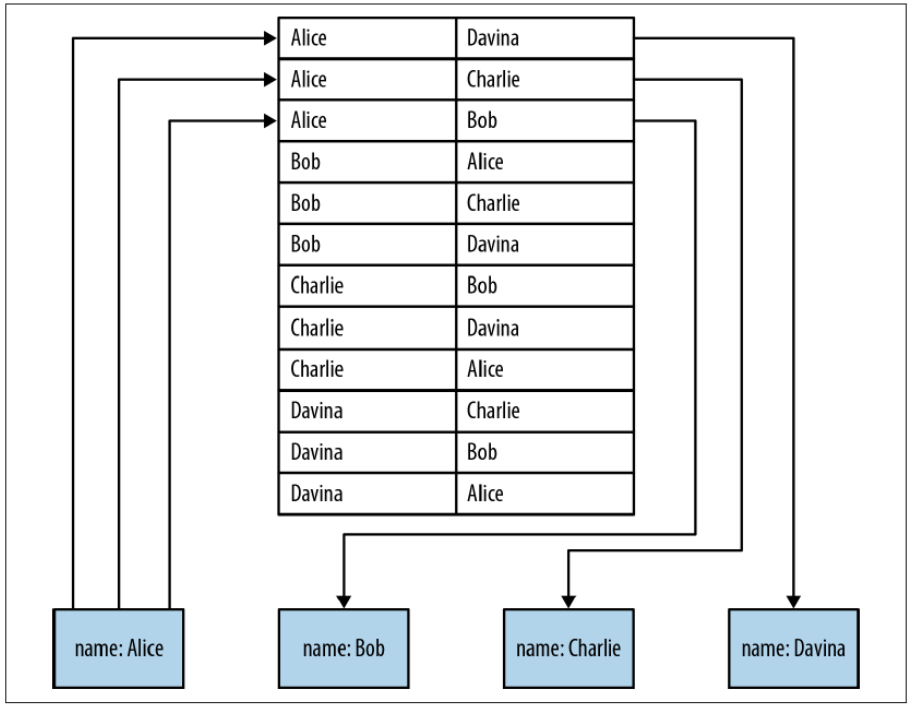


Figure 6-1. Nonnative graph processing engines use indexing to traverse between nodes

Index-free Adjacency VS (Global) Indexing

- With **index-free adjacency**, bidirectional joins are effectively precomputed and stored in the database as relationships.
- In contrast, when using **(global) indexes** to simulate connections between records, there is no actual relationship stored in the database.

Graph Database Internals: Neo4j

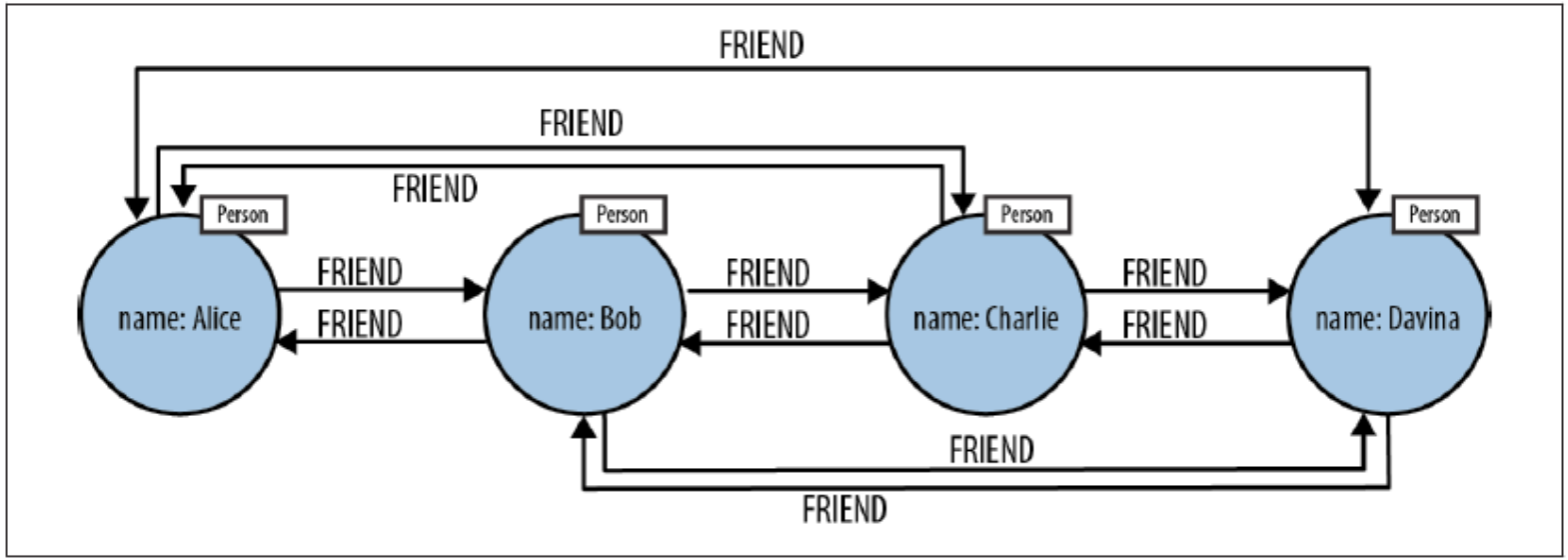


Figure 6-2. Neo4j uses relationships, not indexes, for fast traversals

Native Graph Storage: Neo4j

- If index-free adjacency is the key to high-performance traversals, queries, and writes, then one key aspect of the design of a graph database is the way in which graphs are stored.
- An efficient, **native graph storage format** supports extremely rapid traversals for arbitrary graph algorithms.
- **Neo4j** stores graph data in several different store files.
  - Each store file contains the data for a specific part of the graph (e.g., there are separate stores for nodes, relationships, labels, and properties).
  - The division of storage responsibilities—particularly the separation of graph structure from property data—facilitates performant graph traversals, even though it means the user's view of their graph and the actual records on disk are structurally dissimilar.

Native Graph Storage: Neo4j … Cont'd

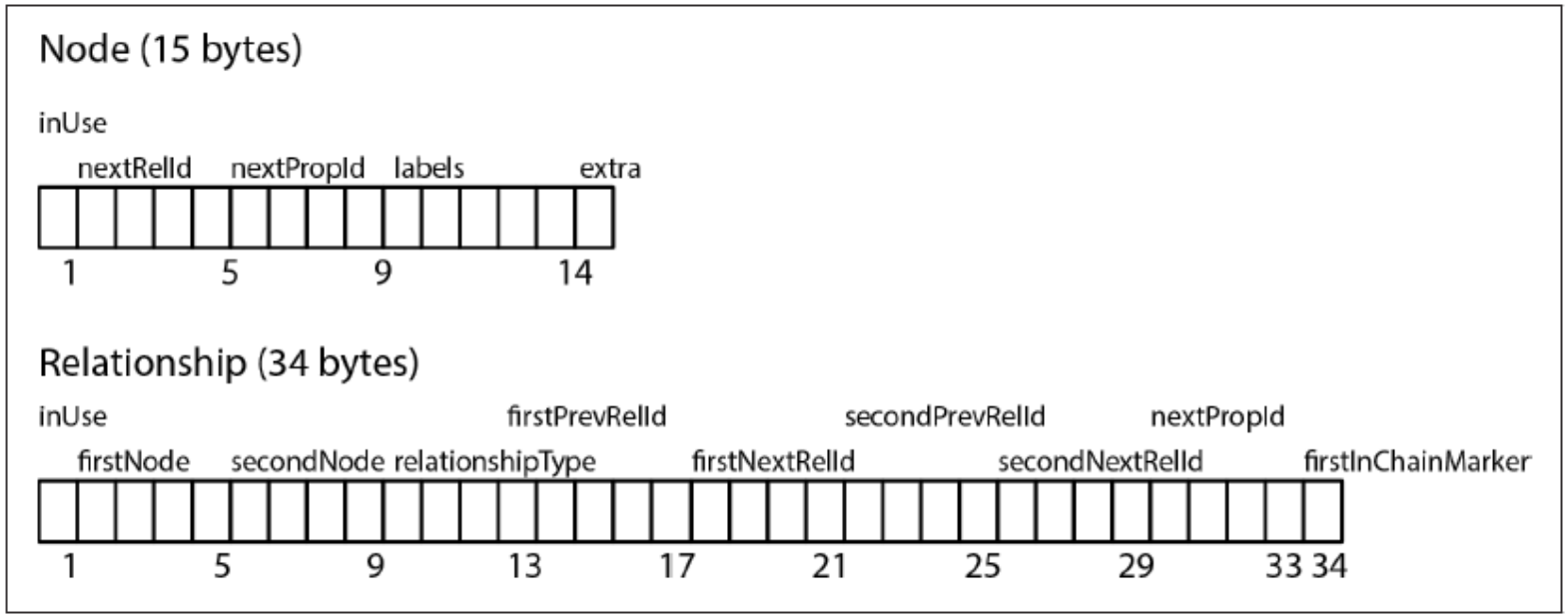
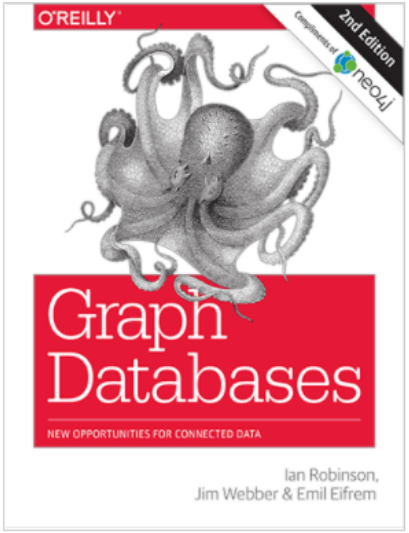


Figure 6-4. Neo4j node and relationship store file record structure

A Reference Book



"This book significantly helps in understanding what graph databases are and how to use them properly...
I really liked reading it!"

- Krzysztof Ropiak, Customer

Print Length: 224 Pages
Available Device Formats: PDF, Kindle, iBooks
Publisher: O'Reilly Media

https://tinyurl.com/rxecdexk

5

Graph Classification

Basic Notions and Ideas

Basic notions

- **Mine frequent graph patterns**
  - Features that are frequent in one class but less in another: discriminative features.
  - Model construction.
- Can adjust frequency, connectivity thresholds, …
- **SVM** and other **supervised machine learning approaches** are used.

Basic Idea

- Mine the frequent sub-graphs, call them **terms**.
- Use “**TF-IDF-like**” **measures** for assigning the most characteristic terms to “documents”.
- Based on such measures and labelled graphs w.r.t. distinct categories, it is possible to perform classification.

Subgraph Classification Rate

- **Basic assumption**: Classification-relevant sub-graphs are more frequent in a specific category than in other categories.
- **Subgraph Classification Rate (SCR)**
  - $SCF(g(c_i))$: Subgraph Class Frequency of subgraph g in category c_i
  - $ISF(g(c_i))$: Inverse Subgraph Frequency of subgraph g in category c_i

$$SCR = SCF(g(c_i)) \times ISF(g(c_i))$$

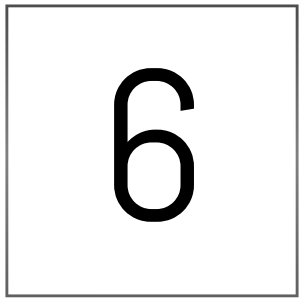
Inverse Subgraph Frequency

$$ISF\left(g(c_i)\right) = \begin{cases} \log_2 \dfrac{\sum N(c_j)}{\sum F\left(g(c_j)\right)} & \text{if } \sum F\left(g(c_j)\right) > 0 \\ \log_2 \left(2 * \sum N(c_j)\right) & \text{if } \sum F\left(g(c_j)\right) = 0 \end{cases}$$

- $ISF(g(c_i))$: measure for the Inverse Frequency of subgraph g in category c_i.
- $N(c_j)$: number of graphs in category c_j.
- $F\left(g(c_j)\right)$: number of graphs containing g in category c_j.

Possible Assignment

- Investigate a specific approach to perform graph classification.
- Investigate and detail (with comparisons of pros and cons) possible "tf-idf-like" measures for performing graph classification.



Graph Partitioning (and Clustering)

Dedicated Lecture

Date to be announced