NDBI040: **Big Data Management and NoSQL Databases**

http://www.ksi.mff.cuni.cz/~svoboda/courses/171-NDBI040/

Lecture 11

Advanced Aspects

Lecturer: **Martin Svoboda**, author: **Irena Holubová**
svoboda@ksi.mff.cuni.cz

12. 12. 2017

Charles University in Prague, Faculty of Mathematics and Physics
Czech Technical University in Prague, Faculty of Electrical Engineering

Graph Databases

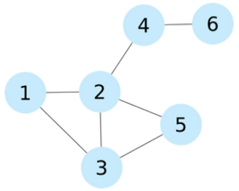
A bit of theory

- Data: a set of entities and their relationships
  - e.g., social networks, travelling routes, …
  - We need to efficiently represent graphs
- Basic operations: finding the neighbours of a node, checking if two nodes are connected by an edge, updating the graph structure, …
  - We need efficient graph operations
- $G = (V, E)$ is commonly modelled as
  - set of nodes (vertices) V
  - set of edges E
  - $n = |V|$, $m = |E|$
- Which data structure should be used?

Adjacency Matrix

- Bi-dimensional array A of $n \times n$ Boolean values
  - Indexes of the array = node identifiers of the graph
  - The Boolean junction A_{ij} of the two indices indicates whether the two nodes are connected
- Variants:
  - Directed graphs
  - Weighted graphs
  - …

Adjacency Matrix



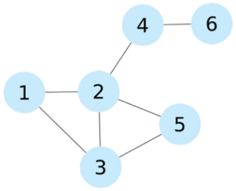
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

- Pros:
  - Adding/removing edges
  - Checking if two nodes are connected
- Cons:
  - Quadratic space with respect to *n*
    - We usually have sparse graphs $\rightarrow$ lots of 0 values
  - Addition of nodes is expensive
  - Retrieval of all the neighbouring nodes takes linear time with respect to *n*

Adjacency List

- A set of lists where each accounts for the neighbours of one node
  - A vector of *n* pointers to adjacency lists
- Undirected graph:
  - An edge connects nodes *i* and *j* => the list of neighbours of *i* contains the node *j* and vice versa
- Often compressed
  - Exploitation of regularities in graphs, difference from other nodes, …

Adjacency List



N1 → {N2, N3}

N2 → {N1, N3, N5}

N3 → {N1, N2, N5}

N4 → {N2, N6}

N5 → {N2, N3}

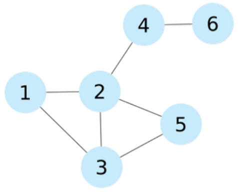
N6 → {N4}

- Pros:
  - Obtaining the neighbours of a node
  - Cheap addition of nodes to the structure
  - More compact representation of sparse matrices
- Cons:
  - Checking if there is an edge between two nodes
    - Optimization: sorted lists => logarithmic scan, but also logarithmic insertion

Incidence Matrix

- Bi-dimensional Boolean matrix of *n* rows and *m* columns
  - A column represents an edge
    - Nodes that are connected by a certain edge
  - A row represents a node
    - All edges that are connected to the node

Incidence Matrix



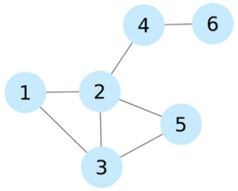
$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

- pros:
  - For representing hypergraphs, where one edge connects an arbitrary number of nodes
- Cons:
  - Requires *n* x *m* bits

Laplacian Matrix

- Bi-dimensional array of *n* x *n* integers
  - Diagonal of the Laplacian matrix indicates the degree of the node
  - The rest of positions are set to -1 if the two vertices are connected, 0 otherwise

Laplacian Matrix



$$\begin{pmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & -1 & 0 \\ -1 & -1 & 3 & 0 & -1 & 0 \\ 0 & -1 & 0 & 2 & 0 & -1 \\ 0 & -1 & -1 & 0 & 2 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$

- Pros:
  - Allows analyzing the graph structure by means of spectral analysis
    - Calculates the eigenvalues

Graph Traversals
Single Step

- Single step traversal from element i to element j, where $i, j \in (V \cup E)$
- Expose explicit adjacencies in the graph
  - e_{out} : traverse to the outgoing edges of the vertices
  - e_{in} : traverse to the incoming edges of the vertices
  - v_{out} : traverse to the outgoing vertices of the edges
  - v_{in} : traverse to the incoming vertices of the edges
  - e_{lab} : allow (or filter) all edges with the label
  - $\in$: get element property values for key r
  - e_p : allow (or filter) all elements with the property s for key r
  - $\in_=$: allow (or filter) all elements that are the provided element

Graph Traversals

Composition

- Single step traversals can compose complex traversals of arbitrary length
  - e.g., find all friends of Alberto
  - „Traverse to the outgoing edges of vertex *i* (representing Alberto), then only allow those edges with the label *friend*, then traverse to the incoming (i.e. head) vertices on those *friend*-labeled edges. Finally, of those vertices, return their *name* property.“

$$f(i) = (\in^{name} \circ v_{in} \circ e_{lab}^{friend} \circ e_{out})(i)$$

Improving Data Locality

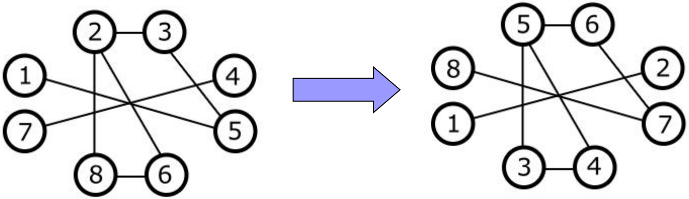
- Idea: take into account computer architecture in the data structures to reach a good performance
  - The way data is laid out physically in memory determines the locality to be obtained
  - Spatial locality = once a certain data item has been accessed, the nearby data items are likely to be accessed in the following computations
    - e.g., graph traversal
- Strategy: in graph adjacency matrix representation, exchange rows and columns to improve the cache hit ratio

Breadth First Search Layout (BFSL)

- Trivial algorithm
- Input: sequence of vertices of a graph
- Output: a permutation of the vertices which obtains better cache performance for graph traversals
- BFSL algorithm:
  1. Selects a node (at random) that is the origin of the traversal
  2. Traverses the graph following a breadth first search algorithm, generating a list of vertex identifiers in the order they are visited
  3. Takes the generated list and assigns the node identifiers sequentially
- Pros: optimal when starting from the selected node
- Cons: starting from other nodes

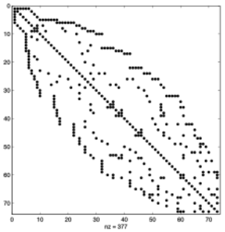
Bandwidth of a Matrix

- Graphs $\leftrightarrow$ matrices
- Locality problem = minimum bandwidth problem
  - Bandwidth of a row in a matrix = the maximum distance between nonzero elements, with the condition that one is on the left of the diagonal and the other on the right of the diagonal
  - Bandwidth of a matrix = maximum of the bandwidth of its rows
- Matrices with low bandwidths are more cache friendly
  - Non zero elements (edges) are clustered across the diagonal
- Bandwidth minimization problem (BMP) is NP hard
  - For large matrices (graphs) the solutions are only approximated



$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \Longrightarrow \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Cuthill-McKee (1969)



- Popular bandwidth minimization technique for sparse matrices
- Re-labels the vertices of a matrix according to a sequence, with the aim of a heuristically guided traversal
- Algorithm:
  1. Node with the first identifier (where the traversal starts) is the node with the smallest degree in the whole graph
  2. Other nodes are labeled sequentially as they are visited by BFS traversal
     - In addition, the heuristic prefers those nodes that have the smallest degree

Graph Partitioning

- Some graphs are too large to be fully loaded into the main memory of a single computer
  - Usage of secondary storage degrades the performance of graph applications
  - Scalable solution <u>distributes</u> the graph on multiple computers
- We need to partition the graph reasonably
  - Usually for particular (set of) operation(s)
  - The shortest path, finding frequent patterns, BFS, spanning tree search, …

One and Two Dimensional Graph Partitioning

- Aim: partitioning the graph to solve BFS more efficiently
  - Distributed into shared-nothing parallel system
  - Partitioning of the adjacency matrix
- 1D partitioning
  - Matrix rows are randomly assigned to the *P* nodes (processors) in the system
  - Each vertex and the edges emanating from it are owned by one processor

$$
\begin{array}{c|cccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
\hline
4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
\hline
7 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
8 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
10 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
11 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
12 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}
$$

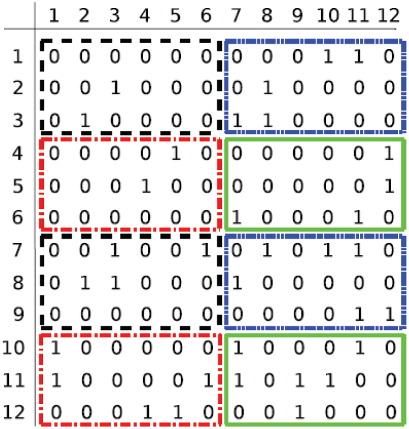
One and Two Dimensional Graph Partitioning

- BFS with 1D partitioning
  - Input: starting node *s* having level 0
  - Output: every vertex *v* becomes labeled with its level, denoting its distance from the starting node

1. Each processor has a set of frontier vertices *F*
   - At the beginning it is node *s* where the BFS starts
2. The edge lists of the vertices in *F* are merged to form a set of neighbouring vertices *N*
   - Some owned by the current processor, some by others
3. Messages are sent to all other processors to (potentially) add these vertices to their frontier set *F* for the next level
   - A processor may have marked some vertices in a previous iteration => ignores messages regarding them

One and Two Dimensional Graph Partitioning

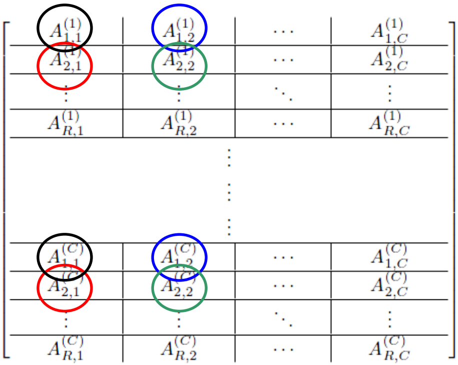
- 2D partitioning
  - Processors are logically arranged in an R x C processor mesh
  - Adjacency matrix is divided C block columns and R x C block rows
  - Each processor owns *C* blocks
- Note: 1D partitioning = 2D partitioning with *C = 1* (or *R = 1*)
- Consequence: each node communicates with at most *R + C* nodes instead of all *P* nodes
  - In step 2 a message is sent to all processors in the same row
  - In step 3 a message is sent to all processors in the same column



$$
\begin{array}{c|cccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
7 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
8 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
10 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
11 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
12 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}
$$

$A^{(*)}_{i,j}$

= block owned by processor *(i,j)*



$$
\left[\begin{array}{c|c|c|c}
A^{(1)}_{1,1} & A^{(1)}_{1,2} & \cdots & A^{(1)}_{1,C} \\
\hline
A^{(1)}_{2,1} & A^{(1)}_{2,2} & \cdots & A^{(1)}_{2,C} \\
\hline
\vdots & \vdots & \ddots & \vdots \\
\hline
A^{(1)}_{R,1} & A^{(1)}_{R,2} & \cdots & A^{(1)}_{R,C} \\
\hline
 & \vdots & & \\
 & \vdots & & \\
 & \vdots & & \\
\hline
A^{(C)}_{1,1} & A^{(C)}_{1,2} & \cdots & A^{(C)}_{1,C} \\
\hline
A^{(C)}_{2,1} & A^{(C)}_{2,2} & \cdots & A^{(C)}_{2,C} \\
\hline
\vdots & \vdots & \ddots & \vdots \\
\hline
A^{(C)}_{R,1} & A^{(C)}_{R,2} & \cdots & A^{(C)}_{R,C}
\end{array}\right]
$$

Partitioning of vertices:

Processor *(i, j)* owns vertices corresponding to block row *(j−1) x R + i*

Types of Graphs

- Single-relational
  - Edges are homogeneous in meaning
    - e.g., all edges represent friendship
- Multi-relational (property) graphs
  - Edges are typed or labeled
    - e.g., friendship, business, communication
  - Vertices and edges in a property graph maintain a set of key/value pairs
    - Representation of non-graphical data (properties)
    - e.g., name of a vertex, the weight of an edge

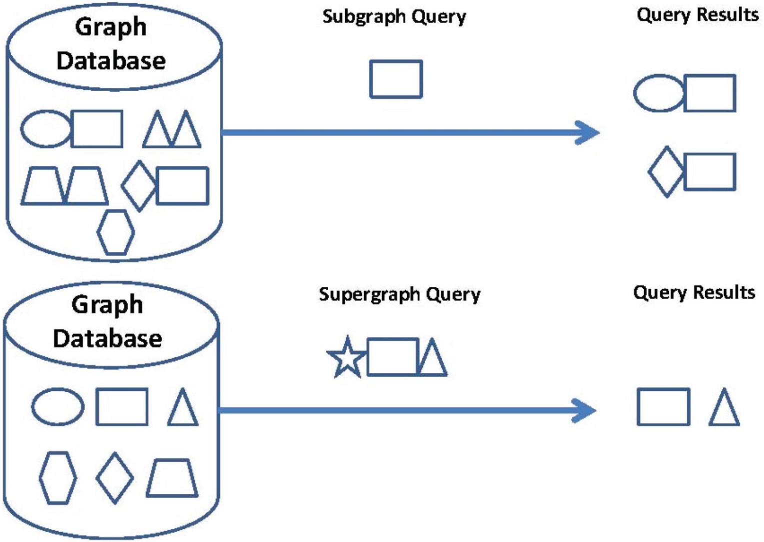
Graph Databases

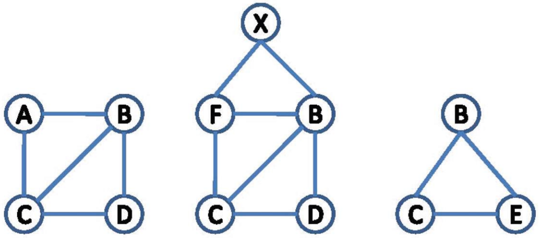
- A graph database = a set of graphs
- Types of graphs:
  - Directed-labeled graphs
    - e.g., XML, RDF, traffic networks
  - Undirected-labeled graphs
    - e.g., social networks, chemical compounds
- Types of graph databases:
  - Non-transactional = few numbers of very large graphs
    - e.g., Web graph, social networks, …
  - Transactional = large set of small graphs
    - e.g., chemical compounds, biological pathways, linguistic trees each representing the structure of a sentence…

Transactional Graph Databases

Types of Queries

- Sub-graph queries
  - Searches for a specific pattern in the graph database
  - A small graph or a graph, where some parts are uncertain
    - e.g., vertices with wildcard labels
  - More general type: sub-graph isomorphism
- Super-graph queries
  - Searches for the graph database members of which their whole structures are <u>contained</u> in the input query
- Similarity (approximate matching) queries
  - Finds graphs which are <u>similar</u>, but not necessarily isomorphic to a given query graph
  - Key question: how to measure the similarity

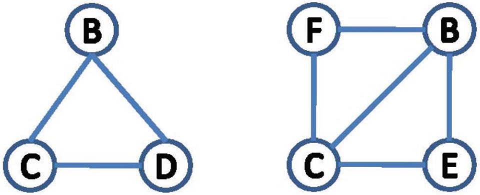

Graph Database
Subgraph Query
Query Results
Graph Database
Supergraph Query
Query Results



sub-graph:
q_1: g_1, g_2
q_2: ∅

g_1 g_2 g_3

super-graph:
q_1: ∅
q_2: g_3



q_1 q_2

Sub-graph Query Processing

Mining-Based Graph Indexing Techniques

- Idea: if features of query graph *q* do not exist in data graph *G*, then *G* cannot contain *q* as its sub-graph
- Graph-mining methods extract selected features (sub-structures) from the graph database members
  - An <u>inverted index</u> is created for each feature
- Answering a sub-graph query *q*:
  1. Identifying the set of features of *q*
  2. Using the inverted index to retrieve all graphs that contain the same features of *q*
- Cons:
  - Effectiveness depends on the quality of mining techniques to effectively identify the set of features
  - Quality of the selected features may degrade over time (after lots of insertions and deletions)
    - Re-identification and re-indexing must be done

Sub-graph Query Processing
Non Mining-Based Graph Indexing Techniques

- Focus on indexing whole constructs of the graph database
  - Instead of indexing only some selected features
- Cons:
  - Can be less effective in their pruning (filtering) power
  - May need to conduct expensive structure comparisons in the filtering process
- Pros:
  - Can handle graph updates with less cost
    - Do not rely on the effectiveness of the selected features
    - Do not need to rebuild whole indexes

Graph Similarity Queries

- Find sub-graphs in the database that are similar to query *q*
  - Allows for node mismatches, node gaps, structural differences, …
- Usage: when graph databases are noisy or incomplete
  - Approximate graph matching query-processing techniques can be more useful and effective than exact matching
- Key question: how to measure the similarity?