B4M36DS2, BE4M36DS2: **Database Systems 2**

http://www.ksi.mff.cuni.cz/~svoboda/courses/211-B4M36DS2/

Lecture 13

Advanced Aspects

Lecturer: **Martin Svoboda**, author: **Irena Holubová**

martin.svoboda@fit.cvut.cz

13. 12. 2021

Charles University, Faculty of Mathematics and Physics

Czech Technical University in Prague, Faculty of Electrical Engineering

Graph Databases

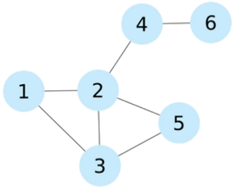
A bit of theory

- 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* x *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



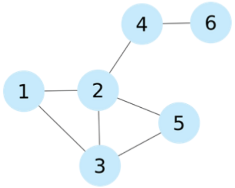
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

- Pros:
  - Adding/removing edges
  - Checking if two nodes are connected
- Cons:
  - Quadratic space with respect to *n*
    - We usually have sparse graphs → 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}

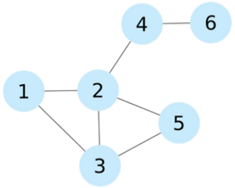
N6 → {N4}

- 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



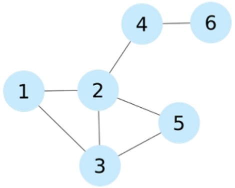
$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

- 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

Improving Data Locality

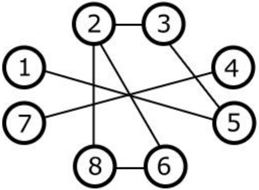
- Idea: take into account computer architecture in the data structures to reach a good performance
  - The way data is laid out physically in memory determines the locality to be obtained
  - Spatial locality = once a certain data item has been accessed, the nearby data items are likely to be accessed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

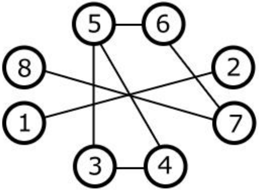
Breadth First Search Layout (BFSL)

- 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

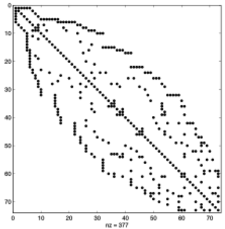




$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

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

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 <u>the smallest degree</u> 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 <u>the smallest degree</u>

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 distributes 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

	1	2	3	4	5	6	7	8	9	10	11	12
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

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

	1	2	3	4	5	6	7	8	9	10	11	12
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

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

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

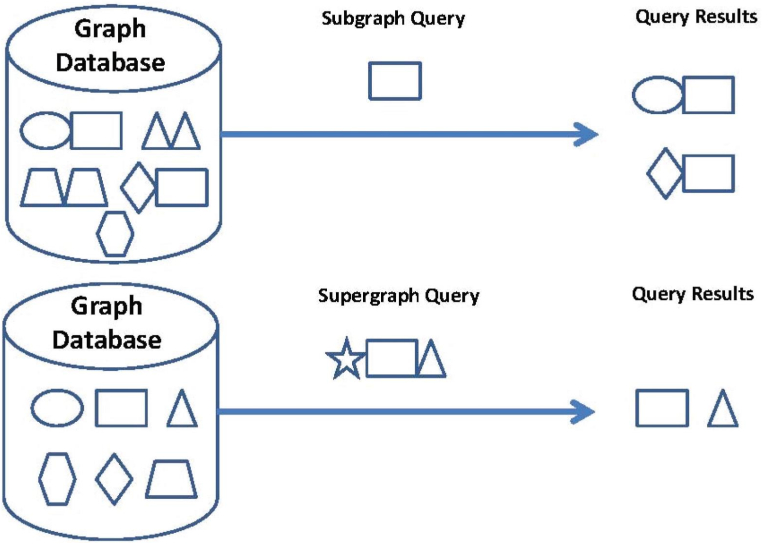
$$
\left[\begin{array}{c|c|c|c}
A^{(1)}_{1,1} & A^{(1)}_{1,2} & \cdots & A^{(1)}_{1,C} \\ \hline
A^{(1)}_{2,1} & A^{(1)}_{2,2} & \cdots & A^{(1)}_{2,C} \\ \hline
\vdots & \vdots & \ddots & \vdots \\ \hline
A^{(1)}_{R,1} & A^{(1)}_{R,2} & \cdots & A^{(1)}_{R,C} \\ \hline
 & & \vdots & \\
 & & \vdots &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

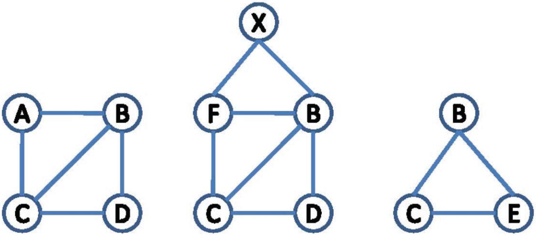
Partitioning of vertices:

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

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

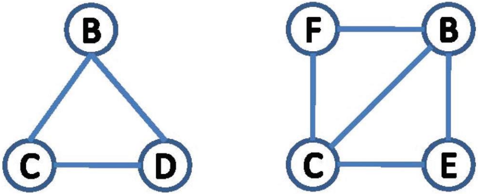

Graph Database
Subgraph Query
Query Results
Graph Database
Supergraph Query
Query Results



sub-graph:
q_1: g_1, g_2
q_2: $\varnothing$

g_1 g_2 g_3

super-graph:
q_1: $\varnothing$
q_2: g_3



q_1 q_2

Performance Tuning
Goals

Example from 2010: Tweets add up to 12 Terabytes per day. This amount of data needs around 48 hours to be written to a disk at a speed of about 80 Mbps.

- MapReduce creates a bottleneck-free way of scaling out
- To reduce latency
  - Latency:
    - Non-parallel systems: time taken to execute the entire program
    - Parallel systems: time taken to execute the smallest atomic sub-task
  - Strategies:
    - Reducing the <u>execution time</u> of a program
    - Choosing the most <u>optimal algorithms</u> for producing the output
    - <u>Parallelizing</u> the execution of sub-tasks
- To increase throughput
  - Throughput = the amount of input that can be manipulated to generate output within a process
  - Non-parallel systems:
    - Constrained by the available resources (amount of RAM, number of CPUs)
  - Parallel systems:
    - “No” constraints
    - Parallelization allows for any amount of commodity hardware

Performance Tuning

Linear Scalability

- Typical horizontally scaled MapReduce-based model: linear scalability
  - "One node of a cluster can process ***x*** MBs of data every second $\rightarrow$ ***n*** nodes can process ***x*** × ***n*** amounts of data every second."
    - Time taken to process *y* amounts of data on a single node = t seconds
    - Time taken to process *y* amounts of data on *n* nodes = *t* / *n* seconds
- Assumption: tasks can be parallelized into equally balanced units

Performance Tuning

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}.$$

Amdahl's Law

- Formula for finding the maximum improvement in performance of a system when a part is improved
  - *P* = the proportion of the program that is parallelized
  - 1 – *P* = the proportion of the program that cannot be parallelized
  - *N* = the times the parallelized part performs as compared to the non-parallelized one
    - i.e., how many times faster it is
      - e.g., the number of processors
    - Tends to infinity in the limit
- Example: a process that runs for 5 hours (300 minutes); all but a small part of the program that takes 25 minutes to run can be parallelized
  - Percentage of the overall program that can be parallelized: 91.6%
  - Percentage that cannot be parallelized: 8.4%
  - Maximum increase in speed: 1 / (1 – 0.916) = ~11.9 times faster
    - *N* tends to infinity

Performance Tuning

$L = kW$

Little's Law

- Origins in economics and queuing theory (mathematics)
- Analyzing the load on stable systems
  - Customer joins the queue and is served (in a finite time)
- "The average number of customers (*L*) in a stable system is the product of the average arrival rate (*k*) and the time each customer spends in the system (*W*)."
  - Intuitive but remarkable result
  - i.e., the relationship is not influenced by the arrival process distribution, the service distribution, the service order, or practically anything else
- Example: a gas station with cash-only payments over a single counter
  - 4 customers arrive every hour
  - Each customer spends about 15 minutes (0.25 hours) at the gas station
  - $\Rightarrow$ There should be on average 1 customer at any point in time
  - $\Rightarrow$ If more than 4 customers arrive at the same station, it would lead to a bottleneck

Performance Tuning

Message Cost Model

initialization

$C = a + bN$

linear dependence on size

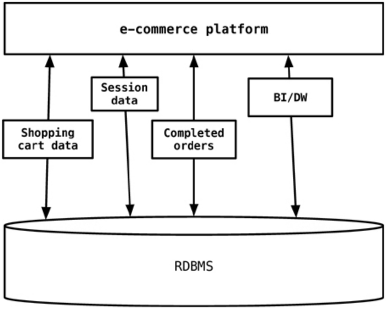
- Breaks down <u>the cost of sending a message</u> from one end to the other in terms of its fixed and variable costs
  - C = cost of sending the message from one end to the other
  - a = the upfront cost for sending the message
  - b = the cost per byte of the message
  - N = number of bytes of the message
- Example: gigabit Ethernet
  - a is about 300 microseconds = 0.3 milliseconds
  - b is 1 second per 125 MB
    - Implies a transmission rate of 125 MBps.
  - 100 messages of 10 KB => take 100 × (0.3 + 10/125) ms = 38 ms
  - 10 messages of 100 KB => take 10 × (0.3 + 100/125) ms = 11 ms
  - A way to optimize message cost is to send as big packet as possible each time

0,08

0,8

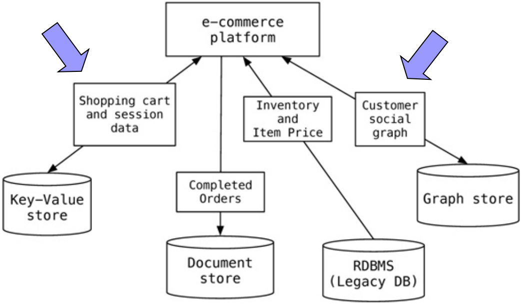
Polyglot Persistence

- Different databases are designed to solve different kinds of problems
- Using a single database engine for all of the requirements usually leads to partially non-performant solutions
- Example: e-commerce
  - Many types of data
    - Business transactions, session management data, reporting, data warehousing, logging information, …
  - Do not need the same properties of availability, consistency, or backup requirements



Polyglot Persistence

- Polyglot programming (2006)
  - Applications should be written in a mix of languages
  - Different languages are suitable for tackling different problems
- Polyglot persistence
  - Hybrid approach to persistence
  - e.g., a data store for the shopping cart which is highly available vs. finding products bought by the customers' friends

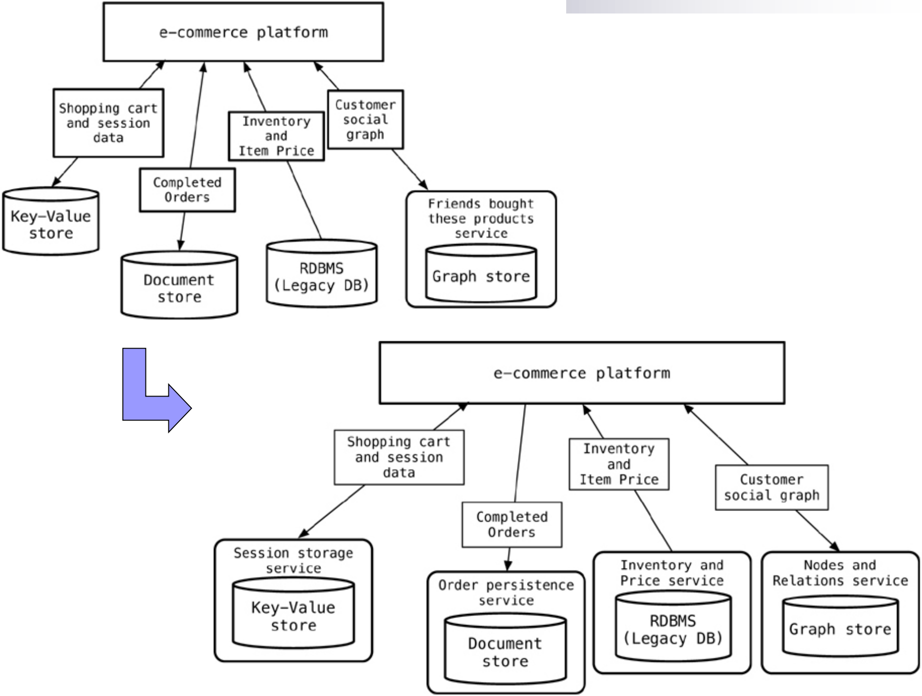


Polyglot Persistence

- There may be other applications in the enterprise
  - e.g., the graph data store can serve data to applications that need to understand which products are being bought by a certain segment of the customer base

$\Rightarrow$ Instead of each application talking independently to the graph database, we can wrap the graph database into a service

  - Assumption:
    - Nodes can be saved in one place
    - Queried by all the applications
  - Allows for the databases inside the services to evolve without having to change the dependent applications


e-commerce platform
Shopping cart and session data
Completed Orders
Inventory and Item Price
Customer social graph
Key-Value store
Document store
RDBMS (Legacy DB)
Friends bought these products service
Graph store
e-commerce platform
Shopping cart and session data
Completed Orders
Inventory and Item Price
Customer social graph
Session storage service
Key-Value store
Order persistence service
Document store
Inventory and Price service
RDBMS (Legacy DB)
Nodes and Relations service
Graph store