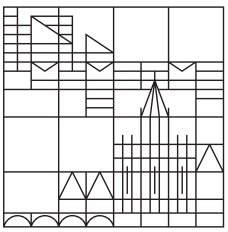
[Draft] Locality Optimization

for traversal-based queries on graph databases

A thesis submitted to the

Universität Konstanz



Department of Computer and Information Science

1st Reviewer:	Prof. Dr. Michael Grossniklaus
2nd Reviewer:	Dr. Michael Rupp

in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer and Information Science

by
Fabian Klopfer
Konstanz, 2020

Abstract:

Some New Abstract Text

Contents

1 Introduction

Essay-like Intro.

Organisation

Contributions Static/offline rearrangement scheme incidence list reordering block IO-based measurement

2 Graphs

In the following sections we discuss what data structures and algorithms are employed when computing with graph-based data. First we give a definition of graphs as discrete structures and concepts based upon that. Next we introduce and analyze possible data structures to represent graphs. Finally algorithms for traversals, path finding and partitioning or community detection are considered.

2.1 Definitions

Most of the definitions below follow the notations intorduced in [72, 32, 1, 16, 30]

A **graph** G is a tuple (V, E) where V is a non-empty set of vertices (also called nodes). E is a subset of cartesian product of the set vertices $E \subseteq V \times V$, called edges. A **subgraph** is a graph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$.

Two vertices are called **adjacent**, if there exists an edge between these vertices:

$$u, v \in V \text{ adjacent } \Leftrightarrow \exists e \in E : e = (u, v) \vee e = (v, u).$$

Given one vertex $v \in V$, the neighbourhood of v are all vertices that are adjacent to v:

$$N_v = u \in V | (v, u) \in E \vee (u, v) \in E.$$

A vertex and an edge are called incident, if the edge connects the vertex to another vertex (or itself):

$$v \in V, e \in E \text{ incident } \Leftrightarrow \exists u \in V : e = (u, v) \vee (v, u).$$

The number of neighbours a vertex has is called the **degree**:

$$v \in V : deg(v) = |N_v|.$$

The average degree of the graph G is defined by:

$$\deg(V) = \frac{1}{|V|} \sum_{v \in V} \deg(v)$$

The set of neighbours connected to a node by incoming edges is called $N_v|_{\text{in}}$. Analogously we define $N_v|_{\text{out}}$. One can model villages and roads using a graph. Given two villages that are connected by a road are adjacent. The road and one of the two cities are incident and all villages connected to one specific village by roads are the neighbourhood of this specific village.

A graph is **undirected**, if E is a symmetric relation, that is $(u, v) \in E \Rightarrow (v, u) \in E$. Otherwise the graph is called **directed**, that is the order within the tuple matters and E is not symmetric. Similar to the edges incident to a vertex we can define the incoming and outgoing edges by restricting which of the positions the vertex takes. The set of incoming edges is defined as:

$$v \in V : \text{In}_v = \{e \in E | u \in V : (u, v)\}.$$

Similarly the outgoing edges are defined as

$$v \in V : \text{Out}_v = \{e \in E | u \in V : (v, u)\}.$$

For example rivers or irrigation systems always have a flow, running exclusively in one direction. This behaviour can be modeled using a directed graph.

Weights can be assigned to both edges and vertices. The graph is called **weighted**, if either edges or nodes are assigned weights. Otherwise it's called unweighted. Similarly labels can be assigned to both nodes and edges. In some cases these labels may encode the type of the entity. Other arbitrary key-value pairs may be assigned to either the nodes or the edges, the so called properties. An example for a weighted graph is a road network: The vertices are crossings between roads, the roads are the edges and the edge weights represent the distances between the crossings that are connected by the road. To include labels, one could distinguish between highways and minor roads or simply assign the name of the road. The former would model the type of the road, while the latter would be an (potentially non-unique) identifier.

In case there may exist multiple edges between the same pair of nodes in the same direction, then the graph is called **multigraph**. That is, $E_M = (E, M)$ is a multiset, with $M : E \to \mathbb{N}$. Imagine one tries to model the transportation links between major cities. There are many possible means: Highways, railways, flights and for some sea routes. In particular, two cities may be connected by more than one mean of transportation.

A **walk** of length n is a sequence of edges, where the target and the source of consecutive edges are equal. Let $u, v, w \in V$. Then a trail is a sequence $(e_i)_{i\in\{0,\dots,n-1\}}$ where $e_i \in E$ and

$$\forall j \in \{0, \dots, n-2\} : e_j = (u, v) \Rightarrow e_{j+1} = (v, w)$$

A **trail** is a walk, where all edges are distinct. A **path** is a trail, where no vertex is visited twice. When planning a route from some point to another, one is interested in finding a path between these points. More explicitly, one wants to find the shortest possible path. Algorithms to solve this problem setting are given later in this chapter. A **cycle** is a trail, where the first visited vertex is also the last visited vertex. If you start your route from home, go to work and return home after closing time, your route is a cycle.

A graph is called **connected**, if for each pair of vertices there exists a path between those:

$$G \text{ connected } \Leftrightarrow \forall v_i, v_j \in V : \exists \text{ Path}(u, v).$$

A **tree** is a graph, which is connected and cycle-free. A **spanning tree** is a subgraph $G' = (V', E')$ of $G = (V, E)$, that is a tree and $V' = V$.

When partitioning a graph, one splits the vertices in disjoint subsets. Thus a **partition** of a graph is a set of subgraphs $i \in \{0, \dots, n-1\} : G_i = (V_i, E_i)$ of G, where

1. $\forall i, j \in \{0, \dots, n-1\}, i \neq j : V_i \cap V_j = \emptyset$.
2. $\bigcup_i V_i = V$.

2.2 Data Structures

When implementing graphs for computing machinery, there are some possibilities on how to represent the graph in memory. We only consider the costs of storing the structure of the graph, for the sake of succintness. Most of the following data structures can be extended to include labels and properties either by using additional fields or pointers. The definitions of the data types and parts of the complexity analysis are based upon [32, 1, 16, 30, 73]. Besides the ones elaborated on below there are the compressed sparse column and row (CSC/CSR) representations, which are used for sparse matrices in arithmetics-heavy applications, like in the library Eigen or Matlab. For more information on these the reader is reffered to [73, 21]. In 1 you can see a visualization of the graph that is used as an example troughout this section.

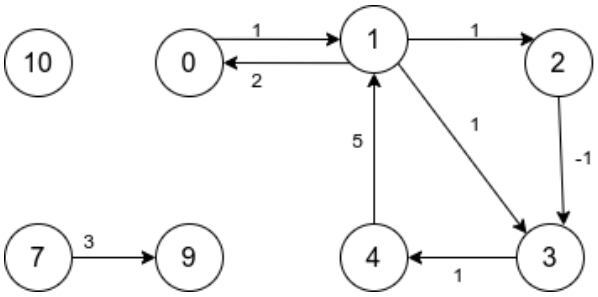


Figure 1 *An example graph used throughout this section.*

Unordered Edge List

The simplest representation uses an unordered list of edges. That is each element of the data structure carries the information of exactly one edge. For example in a directed, weighted graph, the indices of the source and target node and the weight of the edge are one entry. Additionally an edge list needs to store a list of vertex indices, in order to represent nodes with no edges. Overall this results in $|\mathcal{E}| + |\mathcal{V}|$ space complexity.

The number of nodes can be retreived in $\mathcal{O}(1)$ assuming that the list data structure stores its size as a filed. The same is true for edges. Finding a vertex requires to inspect the list of vertices, thus $\mathcal{O}(|V|)$. Assuming the list stores a pointer to its tail, vertex insertion's asymptotic runtime is $\mathcal{O}(1)$. Deleting a vertex requires a pass over all edges to remove the ones including the particular vertex, in total $\mathcal{O}(|E|)$. For edges, the basic operations find, and remove can be executed in linear runtime, i.e. $\mathcal{O}(|E|)$. Edge insertion's asymptotic runtime is $\mathcal{O}(1)$, again assuming the list stores a reference to its tail. Deciding wether two vertices are adjacent requires iterating over the list of edges, that is $\mathcal{O}(|E|)$ runtime. Finally, finding the neighbourhood N_v of a vertex requires a again a scan of all edges, i.e. an asymptotic runtime of $\mathcal{O}(|E|)$. The same is true for the incoming and outgoing sets of a vertex. An example of this data structure is shown in 2.

```
0 1 2 3 4 7 9 10

0 1 1
1 0 2
1 2 1
2 3 -1
1 3 1
3 4 1
4 1 5
7 9 3
```

```
0 1 1
1 0 2
1 2 1
2 3 -1
1 3 1
3 4 1
4 1 5
5 6 3
```

Figure 2 *An example of the edge list representation of a graph.The left handside uses a list to encode vertex indices, while the right handside assumes consecutive indexes.*

Adjacency Matrix

An adjacency matrix of a graph G is a $|V| \times |V|$ matrix where a non-zero entry corresponds to an edge with the weight beeing the value of that entry. Let $A \in |V| \times |V|.u, v \in \{0, \ldots, |V| - 1\}$ and $w_{u,v}$ the weight of the edge $e = (u, v) \in E$ then

$$a_{uv} = \begin{cases} w_{u,v} & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$$

Additionally in order to model non-consecutive indices one needs to store a mapping from the actual vertex index to the one used in the matrix — usually represented by a 2D array. It is also important to note, that adjacency matrix representations are not able to represent multi-graphs without further modification. The space complexity of an adjacency matrix is thus $\mathcal{O}(|V|^2+|V|)$.

The number of nodes can be retreived in $\mathcal{O}(1)$, as it's simply the size of the mapping that is stored. For the number of edges, one needs to iterate over all elements of the matrix and count the non-zero entries, which requires one to touch $\mathcal{O}(|V|^2)$ elements. Finding a vertex is just an array lookup, thus $\mathcal{O}(1)$. Insertion requires to add one row and one column to the matrix, as well as one entry to the mapping. This includes reallocating the matrix which is non-deterministic and independent of the matrix size. But it also requires copying all elements to the new matrix, such that we can estimate the overall asymptotic runtime of $\mathcal{O}(|V|^2)$. Deleting a vertex is similar: Either one leaves a gap that may be used on subsequent insertions and simply marks the true id in the mapping as deleted, which would be an $\mathcal{O}(1)$ operation. Alternatively one could imediately reallocate the matrix to free the extra row and column as well as the extra field in the mapping. This would again be non-deteministic, but can again be estimated by copying the elements from the former matrix $\mathcal{O}(|V-1|^2) = \mathcal{O}(|V|^2)$. For edges, the basic operations find, insert and remove can be executed in constant runtime, i.e. $\mathcal{O}(1)$, as a simple array access. Deciding wether two vertices are adjacent requires just reading what is in the particular array at the index of the two nodes, that is $\mathcal{O}(1)$ runtime. Finally, finding the neighbourhood N_v of a vertex requires a again a scan of a row and a column i.e. an asymptotic runtime of $\mathcal{O}(2|V|)$. For the incoming and outgoing sets of a vertex, one needs to access only either a row or a column resulting in $\mathcal{O}(|V|)$ steps per operation. An example of this data structure is shown in 3.

```
0 1 2 3 4 5 6 7
0 1 2 3 4 7 9 10

  0 1 0  0 0 0 0 0
  2 0 1  1 0 0 0 0
  0 0 0 -1 0 0 0 0
  0 0 0  0 1 0 0 0
  0 5 0  0 0 0 0 0
  0 0 0  0 0 0 3 0
  0 0 0  0 0 0 0 0
  0 0 0  0 0 0 0 0
```

Figure 3 *An example of the adjacency matrix representation of a graph.*

Incidence Matrix

An incidence matrix of a graph G is an $|V| \times |E|$ matrix, where each column corresponds to an edge. Each entry in a column is either the positive weight, if the node is the target of the edge or the negative weight, if the node is the source of the edge. Self-loops require a slight extension of this syntax, because here one node would be both source and target such that the entry is zero. One option is to just put the weight as entry of the node. Another problem is that incidence matrices can not represent negative weights without further extensions. Let $u, v \in \{0, |V|-1\}, j \in \{0, |E|-1\}, A \in |V| \times |E|$ and $a_{v,j}$ the entry at row v and column j of A. Let further w_j be the weight of the edge $e_j = (u, v) \in E$. Then

$$a_{vj} = \begin{cases} -w_{v,u} & \text{if } e_j = (v,u) \in E \\ w_{u,v} & \text{if } e_j = (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

As with adjacency matrices, in order to be able to represent non-consecutive indieces, we need to store a mapping from the true node indices to the ones used in the matrix. The space requirements are thus $\mathcal{O}(|V| \cdot |E| + |V|) = \mathcal{O}(|V| \cdot |E|)$.

The number of nodes can be retreived in $\mathcal{O}(1)$, as it's simply the size of the mapping that is stored. The number of edges can also be retreived in $\mathcal{O}(1)$ as it's the second dimension of the matrix. Finding a vertex is just an array lookup, thus $\mathcal{O}(1)$. Insertion requires to add one row and one column to the matrix, as well as one entry to the mapping, as with adjacency lists. Thus the complexity is again the cost of copying the whole matrix $\mathcal{O}(|V| \cdot |E|)$. The same is true for deleting a vertex. In order to find an edge, one needs to scan one row of either the source or the target node of the edge, which requires $\mathcal{O}(|E|)$ steps. Insertion and removal of edges correspond to the case of vertices: One would need to reallocate the matrix and copy all elements resulting in an asymptotic runtime complexity of $\mathcal{O}(|V| \cdot |E|)$. Deciding wether two vertices are adjacent requires reading one row and checking for each non-zero element, if the entry in the other nodes row is also non-zero, which has $\mathcal{O}(|E|)$ runtime. Finally, finding the neighbourhood N_v of a vertex requires a again a scan of a row and checking all non-zero entry columns for the neighbour i.e. an asymptotic runtime of $\mathcal{O}(|E|)$. For the incoming and outgoing sets the procedure is almost the same. The difference is, that only positive or negative non-zero columns — depending on wehter the incoming or outgoing neighbours shall be returned – have to be checked. An example of this data structure is shown in 4.

```
0 1 2 3 4 5 6 7
0 1 2 3 4 7 9 10

-1  2  0  0    0   0  0
1  -2 -1 -(-1) 0   5  0
0   0  1  0    0   0  0
0   0  0  (-1) 1   0  0
0   0  0  0    1  -5  0
0   0  0  0    0   0  0
0   0  0  0    0   0 -3
0   0  0  0    0   0  3
```

Figure 4 *An example of the incidence matrix representation of a graph.*

Adjacency List

In an adjacency list, there is an entry for each vertex in the graph. Each such entry stores the nodes that are adjacent to the vertex, i.e. its neighbourhood N_v. It is important to note, that in most implementations only $N_v|_{\text{out}}$ is the content of the adjacency list. When we sum $|N_v|_{\text{out}}|$ over all vertices of the graph, we count each edge once. The space complexity here is $\mathcal{O}(|V| + |V| \cdot \deg(V)) = \mathcal{O}(|V| + |E|)$, as we store each node once and then for each relationship one more node in the corresponding adjacency list containing $N_v|_{\text{out}}$.

The number of nodes can be retreived in $\mathcal{O}(1)$, as it's the size of the list. For retrieving the number of edges, one needs to iterate over all elements of the node list and sum over their respective adjacency list. This requires $\mathcal{O}(|V| \cdot \deg(V)) = \mathcal{O}(|E|)$ operations. Finding a vertex is just a lookup, thus in $\mathcal{O}(1)$. Inserting a vertex means simply appending an element to a list which is in $\mathcal{O}(1)$. Deleting a vertex requires to iterate over all nodes and their adjacency list in order to remove the occurences as adjacent node and is in $\mathcal{O}(|V| \cdot \deg(V)) = \mathcal{O}(|E|)$. Finding an edge, can be done by checking the adjacency list of the source node, and requires to look at $\mathcal{O}(\deg(V))$ elements. For the insertion of an edge one needs to append one element to the end of the adjacency list of the source node, which can be done in $\mathcal{O}(1)$. Removing an edge again requires to iterate over the adjacency list of the source node and remove the corresponding

entry which is again in $\mathcal{O}(\deg(V))$. Deciding wether two vertices are adjacent can be checked by looking at the adjacency lists of two nodes, that is $\mathcal{O}(2 \cdot \deg(V)) = \mathcal{O}(\deg(V))$ runtime. Finally, the outgoing neighbourhood of a vertex, is already stored and can be returned in $\mathcal{O}(1)$. In contrast for the incoming neighbourhood one needs to access all vertices' adjacency list and see if the particular vertex is contained in it, resulting in $\mathcal{O}(|V| \cdot \deg(V)) = \mathcal{O}(|E|)$ operations. Finding the neighbourhood N_v of a vertex requires to do both of the above queries, that is $\mathcal{O}(|V| \cdot \deg(V) + 1) = \mathcal{O}(|V| \cdot \deg(V)) = \mathcal{O}(|E|)$ operations. Note that in undirected graphs, both directions of all edges exist, i.e. $N_v = N_v|_{\text{out}} = N_v|_{\text{in}}$. This means for undirected graphs all neighbourhood queries are in $\mathcal{O}(1)$. An example of this data structure is shown in 5.

```
0 -> (1, 1)
1 -> (0, 2) -> (2,1) -> (3, 1)
2 -> (3, -1)
3 -> (4, 1)
4 -> (1, 5)
7 -> (9, 3)
9
10
```

Figure 5 *An example of the adjacency list representation of a graph.*

Incidence List

This representation is also called incidence table in [32]. The incidence list of a graph G stores for each vertex $v \in V$ the list of edges it is conencted to. The space requirements are thus $\mathcal{O}(|V| + |V| \cdot \deg(V) + |E|) = \mathcal{O}(|V| + |E|)$. In contrast to adjacency lists, incidence lists do not only store the connected vertices but the edges. This comes with an additional cost of $|E|$ memory, but is beneficial when it comes to accessing information. Another benefit is that the additional costs can be mitigated by using references.

Most of the operations have the same complexity class as when using adjacency lists and the same operations are needed. Differences occur first when removing a vertex: Instead of having to iterate over all lists and check if the vertex is contained, it is sufficient to look the relevant lists up in the vertexes' list and delete them resulting in $\mathcal{O}(\deg(V))$ operations. Differences also occur, when accessing the neighbourhood. As all edges that are incident to a node are stored, finding all neighbours is an $\mathcal{O}(1)$ operation. Considering the incoming and outgoing neighbourhoods, one only needs to filter the list of incident edges accordingly, which has length $\mathcal{O}(\deg(V))$. An example of this data structure is shown in 6.

```
0 -> (0, 1, 1) -> (1, 0, 2)
1 -> (1, 0, 2) -> (1, 2, 1) -> (1, 3, 1) -> (4, 1, 5) -> (0, 1, 1)
2 -> (2, 3, -1) -> (1, 2, 1)
3 -> (3, 4, 1) -> (1, 3, 1) -> (2, 3, -1)
4 -> (4, 1, 5)
7 -> (7, 9, 3)
9 -> (7, 9, 3)
10
```

Figure 6 *An example of the incidence list representation of a graph.*

Summary

While edge lists are able to represent all variations of graphs, the asymptotic runtime for many operations is linear in the number of edges. These are inacceptable costs in many cases.

An adjacency matrix improves the performance for lookups and updates and is thus the standard data structure for many computation heavy tasks and widely used by libraries as Eigen, openBLAS and the Intel math kernel library (MKL) [56, 20, 55]. When dealing with multi-graphs, the adjacency matrix representation requires additional arrays (one per edge "type") or is not able to canonically represent them. The incidence matrix is not able to represent self-loops and negative weights without modification, but has some interesting relationships with other matrices. For example, if one multiplies the incidence matrix with its own transpose, one gets the sum of the adjacency matrix and the gradient matrix, i.e. the laplacian matrix [11]. Further it's useful in physical flow problems and simulations, e.g. when computing the current and resistances in a graph or when simulating micro-circuits [77]. Even tough the incidence matrix requires less space, both options are rather unfeasible when storing large graphs and the incidence matrix provides even worse access times than edge lists. As a side note: The compressed sparse row and compressed sparse column storage formats are very similar to adjacency lists. Insead of using lists, three arrays are used. The first one maps the node to the start index of its relationship in the other two arrays. The other two arrays store the adjacent nodes and the weight of the relationship respectively. CSR/CSC and adjacency lists share most of the algorithmic traits, while requiring least storage. These formats are used for sparse matrix arithmetics in some of the most popular matrix arithmetics libraries, like [56, 20, 55].

Finally the adjacency and incidence lists are quite similar in many respects: Both require linear storage space — which is optimal without further compression. Even tough not optimal for the operations find, insert and remove, both data structures provide access times that are asymptotically better than linear in most cases. If the edges are distributed uniformly, we have an average degree of

$$\deg(V) = \frac{2|E|}{|V|} \leq \frac{2|V|^2}{|V|} = 2|V|$$

In real networks, the distribution is often non-uniform but can be modeled using e.g. binomial, poisson or power law type [39]. A power law distribution would mean that there exist few nodes with a high degree and a lot of nodes with a rather low degree. What is also very applealing is the fact that the adjacency list and especially the incidence list enable one to return the neighbourhood of a vertex in constant or degree-based amount of time. When it comes to traversals of a graph, these are crucial operations as we will see in the next subsection.

In the table 1 we summarize the space and runtime complexities of the described data structures and the operations that act upon them.

	Edge List	Adjacency Matrix	Incidence Matrix	Adjacency List	Incidence List
Space Complexity	$\mathcal{O}(\|V\|+\|E\|)$	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(\|V\| \cdot \|E\|)$	$\mathcal{O}(\|V\|+\|E\|)$	$\mathcal{O}(\|V\|+\|E\|)$
Retrieve $\|V\|$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Retrieve $\|E\|$	$\mathcal{O}(1)$	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(1)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|E\|)$
Find $v \in V$	$\mathcal{O}(\|V\|)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Insert v to V	$\mathcal{O}(1)$	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(\|V\| \cdot \|E\|)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Remove v from V	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(\|V\| \cdot \|E\|)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\deg(V))$
Find $e \in E$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(1)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\deg(V))$	$\mathcal{O}(\deg(V))$
Insert e to E	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\|V\| \cdot \|E\|)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Remove e from E	$\mathcal{O}(\|E\|)$	$\mathcal{O}(1)$	$\mathcal{O}(\|V\| \cdot \|E\|)$	$\mathcal{O}(\deg(V))$	$\mathcal{O}(\deg(V))$
u, v adjacent?	$\mathcal{O}(\|E\|)$	$\mathcal{O}(1)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\deg(V))$	$\mathcal{O}(\deg(V))$
N_v	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|V\|)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(1)$
In_v	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|V\|)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\deg(V))$
Out_v	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|V\|)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(1)$	$\mathcal{O}(\deg(V))$

Table 1 *Space and runtime complexity comparison of the different data types.*

2.3 Algorithms

2.3.1 Traversal Algorithms

Visiting the nodes in a graph is known as graph traversal. The order in which the nodes are visited is given by the respective algorithm. The two most important graph traversal schemas are the breadth first search and the depth first search. Another such schema is the random walk, which is not quite a traversal but rather generates an arbitrary walk. However it defines, in which order nodes are visited even if it does not guarantee that all nodes are visited and that all nodes are visited exactly once.

Depth First Search

One description of the depth first search (DFS) was authored by Charles-Pierre Trémaux in 1818. Even though the work was published a lot later, this is the first appearence in modern citation history [54]. He used so called Trémaux trees to solve arbitrary mazes. Each result of a depth first search is such a Trémaux tree. These have the property that each two adjacent vertexes are in an ancestor-descendant relationship. Even though Trémaux trees themselves are interesting — for example all Hamiltonian paths are Trémaux trees — we focus on the description of the depth first search. Depth first search is very similar to backtracking: Chase one path until it proves to be a dead end. Go back to the to the point where you can take a different path, chose that path to chase and reapeat. In fact Donald Knuth considers depth first search and backtracking to be the same algorithm, as both are acting upon a graph. However when backtracking is used, the graph is often implicit [49]. Even though DFS is a general traversal schema – go deep first – it can also be used for other purposes like finding shortest paths, connected components, testing planarity and many more.

A pseudo-code description of it is given in 1. This version continues its search at the last found edge instead of the last visited edge. This is enable the usage of the implementation for other problems like finding spanning trees, cycles or paths. The runtime is again dependent on the data structure, that is used and again the runtime for querying the neighbourhood of a node is the varying term. Each node is visited exactly once and by the handshaking lemma [32] it should be clear that we visit each edge twice resulting in an overall worst-case runtime of $\mathcal{O}(|V| + |E|)$. Regarding space complexity, the worst case is that the node stack contains all

Algorithm 1: Pseudo-code for a depth first search on a graph G.

Input: Graph $G = (V, E)$, start vertex id v_id, direction d
Output: Search numbers DFS, predecessor edges parent

```
begin
    dfs ← array initialized to −1;
    parents ← array initialized to −1;
    node_stack ← create_stack();
    push(node_stack, v_id);
    while node_stack non-empty do
        node_id ← pop(node_stack);
        current_node_edges ← expand(G, v_id, d);
        for edge ∈ current_node_egdes do
            if dfs[edge.other_v_id] = −1 then
                dfs[edge.other_v_id] ← dfs[node_id] + 1;
                push(node_stack, edge.other_v_id);
                parent[edge.other_id] ← edge.id;
    return distances, parents;
```

nodes, i.e. that it is traversed without repetitions or — put differently — backtracking. The result is a worst-case space complexity of $\mathcal{O}(|V|)$.

Breadth First Search

The first description of beadth first search (BFS) in modern science was given by Konrad Zuse in the course of his Ph.D.thesis on the "Plankalkühl". He used it to find connected components [82]. The schema of the breadth first search was also used to find shortest path solutions for mazes and to wire placed electrical components on a printed circuit board (PCB). The general traversal scheme in breadth first search is to explore all next neighbours before continuing with those who are more steps away. Thus it first explores one "level" exhaustively, before continuing to the next one. In other words: The only difference between DFS and DFS is the data structure that is used: While DFS uses a stack and thus always inspects the element that was inserted last, BFS uses a queue. That means it inspects the element that was inserted first. An illustrative comparison of DFS and BFS is shown in figure 7.

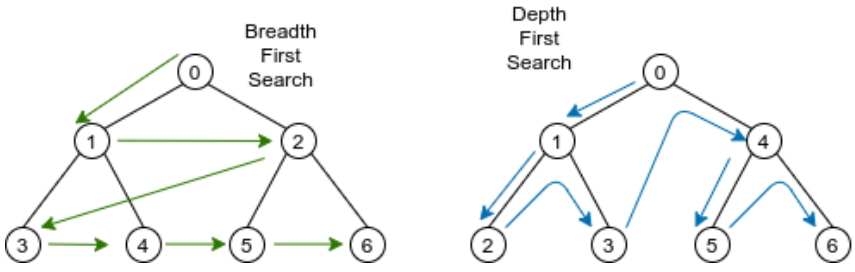


Figure 7 *Comparison between a DFS and a BFS traversal.*

Like the DFS, the BFS traversal schema can be used to find shortest paths, maximum flows and to test if a graph is bipartite. The space and runtime complexity of the BFS is similar to the complexities of DFS: $\mathcal{O}(|V|)$ space, when all nodes are stored in the queue at the same time. $\mathcal{O}(|V|+|E|)$, since all vertices are visited and each edge is visited twice by the handshake lemma. Pseudo-code for the BFS traversal-scheme is shown in 2. We again make the assumption, that

we are using an incidence list as data structure and that the expand operator is implemented accordingly.

Algorithm 2: Pseudo-code for a breadth first search on a graph G.

Input: Graph $G = (V, E)$, start vertex id v_id, direction d
Output: Search numbers bfs, predecessor edges parent

```
begin
    bfs ← array initialized to −1;
    parents ← array initialized to −1;
    node_queue ← create_queue();
    enqueue(node_queue, v_id);
    while node_queue non-empty do
        node_id ← dequeue(node_queue);
        current_node_edges ← expand(G, v_id, d);
        for edge ∈ current_node_egdes do
            if bfs[edge.other_v_id] = −1 then
                bfs[edge.other_v_id] ← bfs[node_id] + 1;
                enqueue(node_queue, edge.other_v_id);
                parent[edge.other_id] ← edge.id;
    return distances, parents;
```

Random Walk

A random walk is a stochastic process, originally defined by Karl Pearson posing the following probelm to the readers or the journal nature in 1905 [62]:

> A man starts from a point O and walks I yards in a straight line; he then turns through any angle whatever and walks another I yards in a second straight line. He repeats this process n times. I require the probability that after these n stretches he is at a distance between r and $r + \delta r$ from his starting point, O.

The problem has grathered wide interests and has many connections ranging from financial mathematics [7], over physics and biology (brownian motion[12]) to pure mathematics [78]. Random walks are modelled mathematically using markov chains. For further information on the pure mathematical theory the reader is reffered to a comprehensive survey of the topic [53]. Our focus will remain database oriented, that is traversal-based. In [24] the authors show, that the generated random walks can be used to compute the similarities of nodes in a graph. This insight is used by a method described in the next chapter.

Pseudo-code describing the algorithm can be found in listing 3. The code makes two assumptions: The function `random()` returns a unsigned integer by drawing from a uniform distribution. The function expand returns the edges with a certain direction of a vertex, given a graph, a vertex (id) and the respective direction.

The runtime complexity of a random walk can be estimated using the number of steps and the average degree of each node in the graph. In each of the n steps we have to construct the list of edges of the currently considered vertex, which has a length of $\mathcal{O}(\deg(V))$. How fast this construction is depends on the data structure that is used. The overall average runtime complexity is $\mathcal{O}(n \cdot \deg(V))$ for incidence lists. For the other data structures, one has to replace the $\deg(V)$ term with the respective runtime of retreiving the neighbourhood of vertex. The average space complexity of a random walk is $\mathcal{O}(\deg(V))$.

Algorithm 3: Pseudo-code for a random walk on a graph G.

Input: Graph $G = (V, E)$, number of steps n, start vertex id v_id, direction d
Output: A walk $(e_0, \ldots, e_{n-1})$

```
begin
    visited_edges ← edge_t[n];
    current_node_edges ← expand(G, v_id, d);
    for i from 0 to n − 1 do
        edge ← current_node_edges[random() % size(current_node_edges)];
        append(visited_edges, edge);
        current_node_edges ← expand(G, edge.other_v_id, d);
    return visited_edges;
```

2.3.2 Shortest Path Algorithms

The shortest path problem is defined by finding the shortest path between nodes efficiently. The algorithms below tackle two specific subproblems:

1. The Dijkstra algorithm finds shortest paths between a single source node and all other nodes in the graph.

2. The A* algorihtm finds a shortest path between a singe source and a single target node.

The difference in the problem setting allows for certain heuristic optimizations. While the Dijkstra's algorithm is slower, it returns more information. The A* algorithm's narrower focus allows it to inspect less elements based on a heuristic and is bound by the Dijksta's algorithm in the worst case.

Dijkstra

In the original formulation, Edsger Dijkstra formulated the algorithm as a solution to the problem of finding a shortest path between two nodes [18]. Many variants and extensions exist of which we are going to discuss two: A* and ALT [35, 29]. One slight variation makes it possible to find all shortest path from a given source node. The algorithm however imposes a restriction on the graph. Only positive weights are allowed as otherwise a negative cycle results in an infinite loop [16].

Conceptually Dijkstra's algorithm assigns each node in the graph a distance. The source node has the distance 0, while all other vertices have a distance of infinity at the beginning. Then it gradually considers the next "shortest" edge to take. That is the distance of the already taken path to a certain node (at the beginning it's zero) is added to an edge weight such that the sum of both is minimal. In order to efficiently compute the minimum a priority queue or a fibonacci heap is used to define the traversal order. An array stores the distances from the source to the already visited nodes, while the queue is filled with their neighbours along with the distance to them (i.e. the path to the predecessor plus the weight of the edge to be taken). The vertex with the shortest total distance is visited until all vertices are visited or the target vertex is reached.

Pseudo code for the variant that computes all shortest paths can be found in 4. The runtime complexity of Dijkstra's algorithm is $\mathcal{O}(|E| \cdot T_d + |V| \cdot T_m)$ where T_d, T_m stand for the complexities to update the distance of a path and to extract the minimum. Besides the new neccissity to select the element to inspect based on priorities and to maintain those, the runtime complexity is equivalent to what we had with BFS. We use a min-priority queue here, which is simpler to implement but yields a sub-optimal runtime: $T_d, T_m \in \log(|\mathcal{V}|)$. Overall the asymptotic runtime complexity using a min-priority queue is $\mathcal{O}((|V| + |E|) \log(|V|))$ [30]. One can also use plain

arrays, which requires a minimum search and no update of the priority. The minimum search is linear, i.e. $\mathcal{O}(|V|)$ and priorities can be updated in $\mathcal{O}(1)$. Overall we have $\mathcal{O}(|E| + |V|^2)$ [30]. Finally more advanced data structures can be used, like a Fibonacci-Heap [16]. These make it possible to update the priority in $\mathcal{O}(1)$ and still find the minimum in $\log(|\mathcal{V}|)$. This yields the optimal asymptotical runtime of $\mathcal{O}(|E| + |V| \log(|V|))$. The worst case space complexity is again $\mathcal{O}(|V|)$, that is when all nodes are stored in the queue at the same time. As there is only one shortest path (paths of equal size are discarded) to each node, the queue contains each node only once.

Algorithm 4: Pseudo-code of the Dijkstra's algorithm for finding shortest paths from a node v to all other nodes in a graph G.

Input: Graph $G = (V, E)$, source vertex id v_id, direction d
Output: Path distance distances, predecessor edges parent

```
begin
    distances ← array initialized to ∞;
    parents ← array initialized to −1;
    path_queue ← create_min_prio_queue();
    enqueue(path_queue, v_id);
    while path_queue non-empty do
        node_id ← dequeue(path_queue);
        current_node_edges ← expand(G, v_id, d);
        for edge ∈ current_node_egdes do
            if distances[edge.other_v_id] ≥ distances[node_id] + edge.weight then
                distances[edge.other_v_id] ← distances[node_id] + edge.weight;
                enqueue(path_queue, edge.other_v_id);
                parent[edge.other_id] ← edge.id;
    return distances, parents;
```

A*

The A* was originally invented in the late 60's to be used for path planning of a robot. It's an extension of Dijkstra's algorithm, that does not just use the distance as metric of priority, but adds a heuristic $h : V \to \mathbb{R}$ to the distance: $v \in V : f(v) = \text{distance}(v) + h(v)$, that has to fulfill certain conditions: With $u, v \in V$ and min distance(u) the minimal distance from the vertex u to the goal vertex

$$\forall u, v : h(u) \leq d(u, v) + h(v) \wedge h(u) \leq \text{min distance}(u).$$

The former condition is called consistency, the latter admissibility. As all consistent heuristics are admissible the first condition is sufficient. An example for graphs with an euclidean coordinate system is the euclidean distance [35].

5 shows pseudo code for the algorithm. The runtime is of course dependent on the complexity of the heuristics. Overall we have the same worst case complexity as with Dijkstra's algorithm for the constant heuristic: $\forall v \in V : h(v) = 0$. The best case of the A* algorithm is when the heuristic is equal to the distance from the current vertex to the goal vertex. Then exactly min distance nodes are visited and the algorithm is in $\mathcal{O}$) min distance), which is the global optimum for a single source shortest path problem.

Algorithm 5: Pseudo-code of the A* algorithm for finding shortest paths from a node v to a node u in a graph G.

Input: Graph $G = (V, E)$, heuristic h, source vertex id v_source, target node v_target, direction d

Output: Path p

```
begin
    parents ← array initialized to −1;
    path_queue ← create_min_prio_queue();
    enqueue(path_queue, v_id);
    while path_queue non-empty do
        node_id ← dequeue(path_queue);
        if node_id = v_target then
            return construct_path(parents);
        current_node_edges ← expand(G, v_id, d);
        for edge ∈ current_node_egdes do
            if distances[edge.other_v_id] ≥ distances[node_id] + edge.weight +
              h(edge.other_v_id) then
                distances[edge.other_v_id] ← distances[node_id] + edge.weight +
                  h(edge.other_v_id);
                enqueue(path_queue, edge.other_v_id);
                parent[edge.other_id] ← edge.id;
    return empty_path();
```

ALT

ALT stands for A*, landmarks, triangular inequality. It is an extension of A* which uses landmarks and the triangular inequality as a heuristic. A landmark is a vertex $v \in V$, which is used for orientation. With ALT we select a set of landmarks L and execute Dijkstra's algorithm on each of those, such that we have a set of distances per node and landmark. More explictly we use that $d(L_i, v) - d(L_i, w) \leq d(v, w)$ as a lower bound to the actual distance.

In the first step — the preprocessing step — of ALT we compute and store these values, giving a space overhead of $\mathcal{O}(|L| \cdot |V|)$. This is shown as pseudo code in 6. In the second step — the actual query — for every node we check which landmark gives the best lower bound of the actual distance. This is done by maximizing the following term per node and using it as heuristic h. With v_t beeing the target node:

$$h(v) = \max_i d(L_i, v) - d(L_i, v_t)$$

After that A* is executed as described in 7.

Besides the additional space that is used we also execute Dijkstra's algorithm $|L|$ times and have an asymptotic complexity of $\mathcal{O}(|L| \cdot (|E| + |V| \log |V|))$ using an incidence list to store the graph and a Fibonacci heap as data structure for the priority queue. Regarding space we need $\mathcal{O}(|V| \cdot (1 + |L|))$. For small values of $|L|$ we preserve the worst case complexity as average case complexity of ALT. What we gain by that is that the precomputations take the main runtime penalty while providing a reasonably good heuristic depending on the selection of the landmarks [29]. How to select the landmarks is discussed in [28].

Algorithm 6: Pseudo-code of the preprocessing stage of ALT.

Input: Graph $G = (V, E)$, direction d, number of landmarks nl
Output: Precomputed distance from each landmark to all other vertices landmarks[$|L|$][$|V|$]

```
begin
    /* Preprocessing stage.                                    */
    /* Done in advance and only once.                          */
    L ← select_landmarks(G, nl, d);
    for l_i ∈ L do
        for v_j ∈ V do
            landmark[i] = dijkstra(G, l_i, d).distances;
    return landmarks;
```

Algorithm 7: Pseudo-code of the query stage of the ALT algorithm for finding shortest paths from a node v to a node u in a graph G.

Input: Graph $G = (V, E)$, source vertex id v_source, target node v_target, direction d, landmarks[$|L|$][$|V|$]
Output: Path p

```
begin
    /* Query stage.                                            */
    /* Done for every shortest path query.                     */
    for v ∈ V \ {v_t} do
        h[v] ← max_i landmarks[i][v]−landmarks[i][v_target]
    return a-star(G, h v_source, v_target, d);
```

2.3.3 Partitioning Algorithms

Graph partitioning is the problem of separating the graph into disjoint subsets. The problem is NP-complete [4] and the algorithms below are thus heuristics: They do not provide the globally optimal partition with respect to some metric but approximate an optimal solution as close as possible. Besides the methods presented below, there are of course a lot of other methods. One method that is of considerable importance is spectral clustering, which we will not elaborate on here in depth. Briefly, the algorithm computes the laplacian matrix, extracts k eigenvectors, changes the basis of the matrix according the the k eigenvectors and clusters the nodes then using a simpler algorithm, like agglomerative cluistering [70] or k-means [52] The interested reader is referred to [19, 76, 58]. Apart from methods considering the graph structure as part of the algorithm itself, all other clustering algorithms can be applied, if suitable features are extracted based upon the graph structure. As we will see later, one of the methods that are related work extracts graph features by executing multiple random walks per node and aggregates them into a multi set in order to characterize the neighbourhood of a node. A comprehensive survey of non graph based algorithms can be found for example in [44, 9, 79, 34]. Other graph feature extraction methods are for example described in [57, 36, 37].

Kernighan-Lin Algorithm

The Kernighan-Lin (KL) algorithm [47] was invented by by Brain Kernighan — one of the creators of Unix and co-author of the de facto standard book "The C Programming Language" and Shen Lin. It was developed and is used for laying out digital circuits on a chips. It is also used by many other more complex graph partitionging algorithms as the multilevel partitioning algorithm implementation of Karypis and Kumar [46] described in the next part.

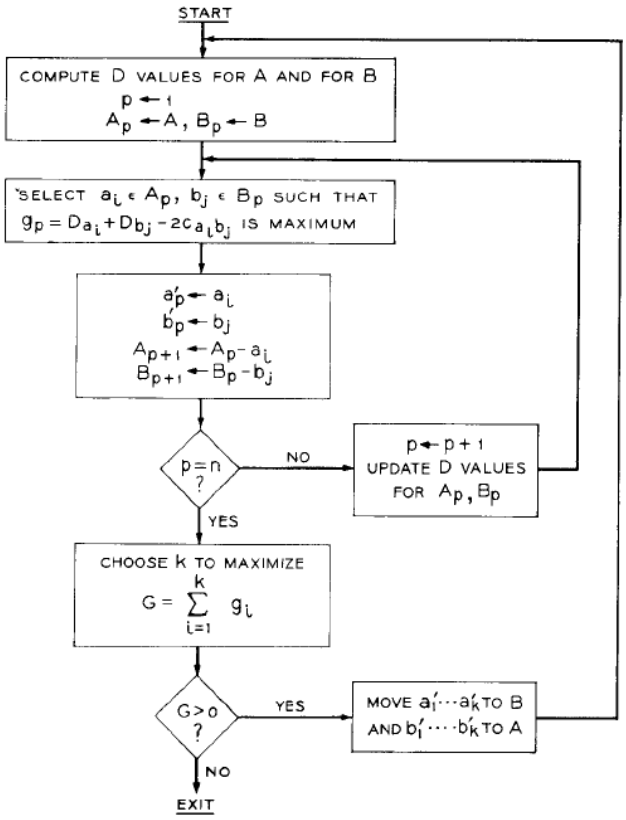


Figure 8 *A flow diagram of the KL algorithm [47].*

The KL algorithm in its most basic form finds a minimal cut of the graph into two dijoint partitions of the same size. It assumes an undirected graph and is extended to unequally sized partitions and k-way partitioning. To decide which node has to be in which partition a cost function $D(v)$ is used. Let $v \in V_i$ and $i \neq j$. $I(v)$ is called the internal cost of v, $E(v)$ is called the external cost of v with:

$$I(v) = \sum_{u \in V_i} w_{(u,v)}$$

$$E(v) = \sum_{s \in V_i} w_{(s,v)}$$

$$D(v) = E(v) - I(V)$$

The internal cost is thus the sum of all edge weights within a partition incident to a specific vertex, while the external cost is the sum of all edge weights to vertices in the other partition for a specifc vertex. The overall cost function is just the difference between external and internal cost. The gain of exchanging two vertices between partitions is proven [47] to be — with $v \in V_i, s \in V_j, i \neq j$:

$$g = D_v + D_s - 2w_{(v,s)}$$

The most basic form works as follows:

1. Split the set of vertices into two partitions arbitrarily.
2. $\forall v \in V$ compute $D(v)$.
3. Select $v \in V_i, s \in V_j, i \neq j$ such that g is maximal. Exclude them from their partitions.
4. Update the D values for all remaining ndoes in both partitions using

$$u \in V_i \setminus v : D'(u) = D(u) + 2w_{(u,v)} - 2w_{(u,s)}$$

$$x \in V_j \setminus s : D'(x) = D(x) + 2w_{(x,s)} - 2w_{(s,v)}$$

5. Goto 3 until the partitions are empty.
6. Choose k to maximize

$$G = \sum_{i=0}^{k} g_i$$

7. If $G > 0$ apply the swaps and goto 2. Else terminate

The steps are summarized in the flow diagram in 8. Regarding the extensions, an improvement schema is provided to avoid local optima: Apply the algorithm to the so created partitions and union the partitions alternatingly (i.e. $A_1 \cup B_2, A_2 \cup B_1$) and use these partitions as initialization of the algorithm. In order to create partitions of unequal size with n_1 being the minimal desired partition size and n_2 the maximal desired partition size, add dummy vertices, such that we have $2n_2$ vertices in total and restrict the number of changes that are allowed to n_1. For partitioning the graph into k-partitions, one needs to split the initial set of vertices into k partitions. Then apply the 2-way partitioning algorithm pairwise to all partitions until convergence.

One pass of the basic form of the algorithm is in $\mathcal{O}(|V|^2 \cdot \log(|V|))$. The k-way partitioning algorithm requires per iteration $\binom{k}{2} = \frac{k(k-1)}{2}$ executions of the algorithm, thus $\mathcal{O}(k^2 \cdot |V|^2 \cdot \log(|V|)$ in total, assuming that the number of iterations is small as shown empirically by the authors [47]. Further improvements were developed in the years after publication, for example a linear time implementation applicable to hypergraphs by Fiduccia and Mattheyses [22].

Multi-Level Partitioning

The multilevel partitioning algorithm was first described by Henderson and Leland [38]. Its steps are visualized schematically in 9 and given below:

1. Coarsen the graph by contracting edges between neighbouring vertices
2. Partition the graph unsing e.g. spectral clustering or the KL algorithm
3. Uncoarsen the graph, projecting the above derived partitioning downwards and refining it by applying th KL algorithm.

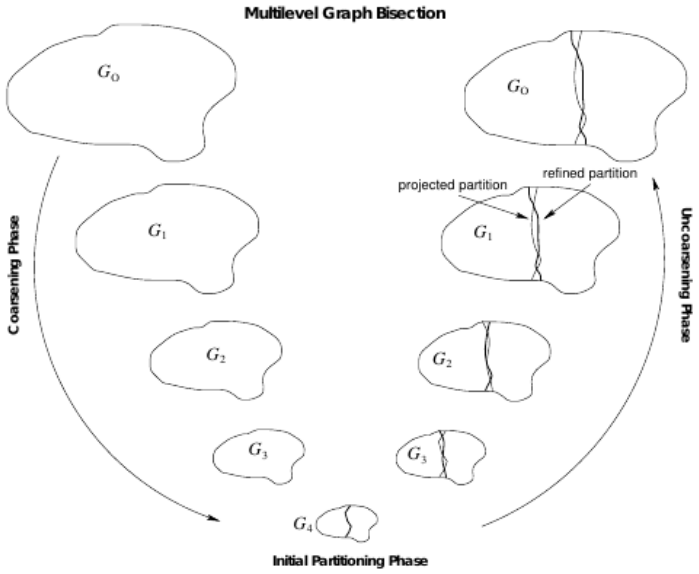


Figure 9 *A visualization of how the multilevel partitioning algorithm works [46].*

A particular implementation of Karypis and Kumar [46] introduces a couple of improvements: A matching scheme for the coarsening stage called heavy edge matching. It works by iterating over all vertices and matching each one with the vertex that is attached to the edge carrying the heaviest weight. This is done until all edges are matched. Afterwards, all pairs of vertices form a new vertex in a new graph, the edges of each vertex pair are aggregated and their weight is summed eventually, if both nodes had an edge to the same other vertex. Further the nodes are assigned a weight that corresponds to the number of vertices that are matched in one meta-vertex. The partitioning stage experiments with three different methods: The KL-algorithm [47], spectral clustering [19], the graph growing partitioning algorithm — which uses BFS with a limited number of steps from a set of arbitrarily chosen vertices. Finally it proposes an improved version of the refinement procedure, where only boundary vertices are considered for interchanges between partitions.

The runtime for heavy edge matching is $\mathcal{O}(|E|)$ for each level and we have at least $\mathcal{O}(\log(\frac{|V|}{|V_c|}))$ steps, where V_c is the set of vertices in the coarsest graph. Assuming $|V_c|$ is a small constant (in the worst case for refinement $|V_c| = 1$), we get overall $\mathcal{O}(|E|\log(|V|))$ for the refinement procedure. The partitioning step is dependent on the choice of the algorithm. Using the KL k-way algorithm we have $\mathcal{O}(k^2 \cdot |V_c|^2 \cdot \log(|V_c|))$. As $|V_c|$ is small in comparison to the original graph, this results in a reduced overall runtime complexity. Finally the uncoarsening including projection and refinement take $\mathcal{O}(\log(|V|) \cdot (|V_i| + \mathcal{O}(k^2 \cdot |V_i|^2 \cdot \log(|V_i|)))$ with $|V_i|$ the number of vertices in the i times coarsened graph. As the projection should already induce a reasonable initial partitioning the algorithm should converge quickly [38]. The overall runime depends

heavily on $|V_c|$. The refinement phase is the most costly step as it needs to be done for the last level too, resulting in an asymptotic complexity of $\mathcal{O}(\mathcal{O}(k^2 \cdot |V|^2 \cdot \log^2(|V|)))$.

Louvain Method

The Louvain method [10] is an algorithm for community detection. The authors define community detection as a graph partitioning problem, where nodes within a partition shall be densely connected, while nodes belonging to different partitions shall be sparsely connected [10]. Measuring the quality of such partitioning can be done using the modularity of the partition as introducted by Newman and Girvan [26]:

$$Q = \frac{1}{2m} \sum_{u,v \in V} \left(w_{(u,v)} - \frac{k_u k_v}{2m} \right) \cdot \delta(c_u, c_v)$$

$w_{(u,v)}$ is the weight of the edge between u and v (or 0 if the edge does not exist). c_u and c_v are the communities the respective nodes are assigned to, and δ is the Kronecker delta function, i.e. 1 if $c_u = c_v$ and 0 otherwise. k_u is the sum of the edge weight connected to u and m is the total edge weight $m = \frac{1}{2} \sum_{e \in E} w_e$. Overall the modularity is in the range $[-1, 1]$ and can be interpreted as measure for the edge (weight) density inside of a partition in contrast to the edge density in the network overall.

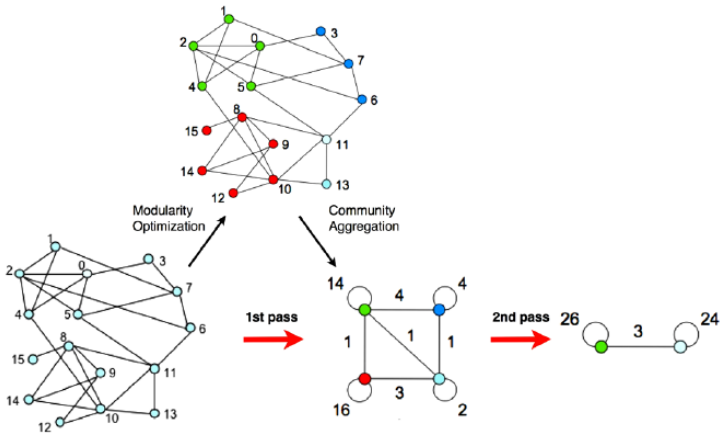


Figure 10 *The steps that the lovain method comprises [10].*

Optimizing the modularity exactly is computationally hard, so as soon as the graph size increases, approximate optimization is required. The Louvain method is such an approximation. It's broad control flow comprises the following steps and is visualized in 10:

1. Initialize the initial partition with each node in an own community
2. $\forall v \in V \forall u \in N_v$: Compute the gain in modularity $\Delta Q(u, v)$ when merging v into the community of u. Place v into the community of the neighbouring node u where $\Delta Q(u, v)$ is maximal, if the gain is positive.
3. Goto step 2 until the improvement is below a threshold.
4. Construct a new graph G_i with the communities of the previous graph being the vertices of the new one and Aggregate the edge weights. Links between nodes of the same communities are aggregated to self-loops.
5. if the previous graph and the new graph differ, goto step 2 using the G_i as input graph, else terminate.

The gain in modularity is computed using a specific formulation, that eases computation:

$$\Delta Q(v, C) = \left(\frac{I_c + 2w_{v,C}}{2m} - \left(\frac{w_C + w_v}{2m} \right)^2 \right) - \left(\frac{I_c}{2m} - \left(\frac{w_C}{2m} \right)^2 - \left(\frac{w_v}{2m} \right)^2 \right)$$

with I_C is the sum of edge weights within the community C, w_C is the sum of the edge weights incident to nodes in C, $w_{v,C}$ is the sum of edge weights between v and nodes in C, w_v is the sum of all edge weights incident to v and m is again the sum of all edge weights.

Many of these quantities can be precomputed and easily updated: m is constant, w_v is constant, I_C can be updated by adding $w_{v,C}$, w_C can be updated by adding w_v, only $w_{v,C}$ needs to be recomputed for every community to consider. It requires to iterate over all edges of v and check if the other incident node is in the community, thus $\deg(V)$ The complexity of the algorithm is determined by computing the gain in modularity for each node and its neighbourhood: $\mathcal{O}(|V| \cdot \deg(V)^2)$. Notice, that this algorithm is similar to the coarsening phase of the multilevel algorithm: The contraction is done by calculating the modularity gain. However, not only two vertices are matched during an iteration but arbitrarily many until convergence. Both partitioning and refinement are not done, as the special contraction mechanism already partitions the graph. That is, opposed to the top-down partitioning of the multilevel partitioning algorithm, the louvain method works bottom-up. Both algorithms however generate a hierarchy.

One problem of the louvain method is its resolution limit: For very large and dense graphs, the total edge weight grows extremely large and thus the modularity is always small. This problem is approached by two different ways: Conde-Céspedes et al. [14] experiment with different other modularity measures, that are provably stable with respect to the graph size. However those measures that provide good results require user defined parameters, which in turn requires either domain specific knowledge of the data or parameter optimization, which implies multiple executions of the algorithm. The Leiden method [75] also introduces an additional parameter by changing the quality function, but also an additional step. It adds a refinement stage after the first step is finished, where the nodes in each partition are again instantiated as a single refined partition and merged in a different way. Empirically, the Leiden algorithm converges faster with higher modularity score, whith appropriate values for γ.

3 Databases

First we are going to elaborate on the storage-related architecture of databases in general. Then we introduce a popular data model that is employed by many current graph databases [27, 6, 3, 64]. Finally, at the end of this chapter the implementation of this model in a native graph database called Neo4J is described.

3.1 Architecture

The reason why we use databases is twofold: First, every computer is equipped with different kinds of memory, which differ in size, capacity, speed and price per byte. This induces the so called memory hierarchy, the principle, that few fast, expensive, low capacity memory is used close to the central processing unit, that gets layerwise augmented with inccreasingly slower, less expensive, higher capacity memory. The last layer, which has the highest capacity, defines the overall capacity, while the smallest one is a crucial factor for performance. Thus what is shown as secondary storage in 11 is orders of magnitude slower in terms of both latency and throughput. But it is also able to store orders of magnitude more data. In order to mitigate

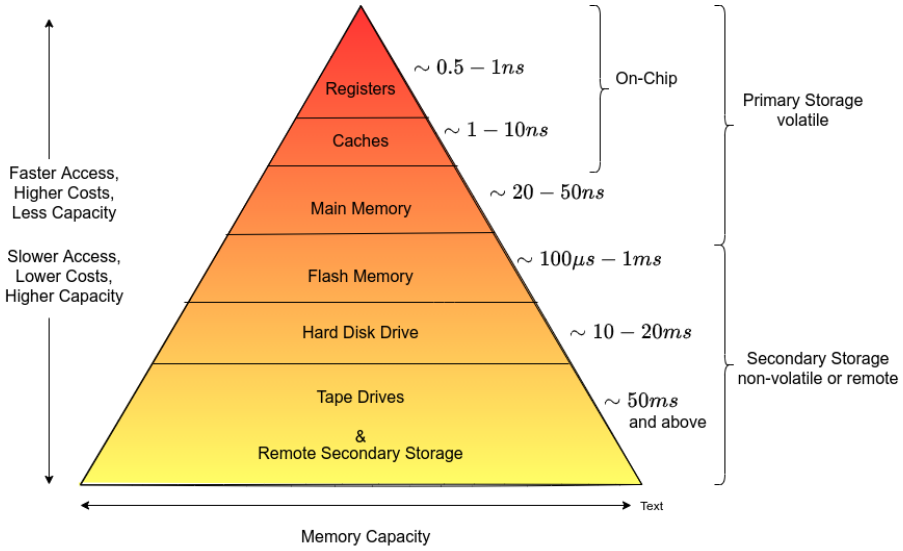


Figure 11 *The memory hierarchy used in today's computing systems.*

the effects of this, the accesses between primary and secondary memory need to be handled very carefully for data intensive — also called IO bound — applications. Second, the operating system (OS) actually handles the first reason. However application specific payloads enable further optimizations when it comes to how data is stored and accessed. Put differently, the operating system is not able to infer certain information, as it does not constrain how data is stored, and as it does not profile in what patterns data is referenced or queried. Databases take care of these issues by different mechanisms, which will be lined out from a high level perspective, in order to understand how a database works on its architectural lower levels. Put differently, we are not going to discuss query processing, transactions, concurrency related components and recovery facilities. Most of the information below is outlined comprehensively in [63, 69]

Let us consider the high level architecture of a general database management systems as shown in figure 12 — with a focus on the storage and access elements.

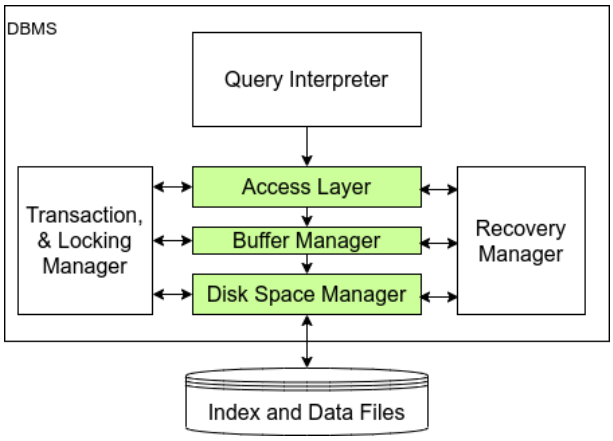


Figure 12 *The typical structure of a relational database management system [63].*

The disk space manager, sometimes also called storage manager, handles de-/allocations, reads & writes and provides the concept of a block: One or many physical disk blocks are grouped to a logical disk block. These logical disk blocks are again grouped and brought into main memory (RAM) — these groups are called a page. Optimally both a disk block and a page are of the same size or at least a multiple of each other. Further the database needs to keep track of free blocks in the file: A linked list or a directory must record free blocks and some structure needs to keep track of the free slots either globally or per block. Data locality is a concept that we examine closely in an extra chapter later on. To summarize the two most important objectives of a storage manager are to

1. take care of (de-)allocations of disk space,
2. abstract storing data on a physical device using the operating system: Files, split into logical disk blocks, accessed using OS facilities, and
3. provide data structures in order to maintain records within a file, blocks.

A buffer manager is used to mediate between external storage and main memory. It provides the concept of a page and maintains a designated pre-allocated area of main memory — called the buffer pool — to load, cache and evict pages into or from main memory [63]. A conceptual illustration of this is shown in 13. It's objective is to minimize the number of disk reads to be executed by caching, pre-fetching and the usage of suitable replacement policies. It also needs to take care of allocating a certain fraction of pages to each transaction.

The final component that is crucial to the storage of data the of a database management system is the file and record layout, along with possible index structures — also called the access layer. In order to store data a DBMS may either use one single or multiple files to maintain records.

A file consists of a set of blocks split into slots. A slot stores one record with each record containing a set of fields. Records can be layout in row or column major order. That is one can store sequences of tuples or sequences of fields. The former is beneficial if a lot of update, insert or delete operations are committed to the database, while the latter optimizes the performance when scans and aggregations are the most typical queries to the system. Records may be of fixed or of variable size, depending on the types of their fields. Another option is to store the structure of the records along with pointers to the values of their fields in one files and the actual values in one or multiple separate files. Also distinct types of tables can be stored in different

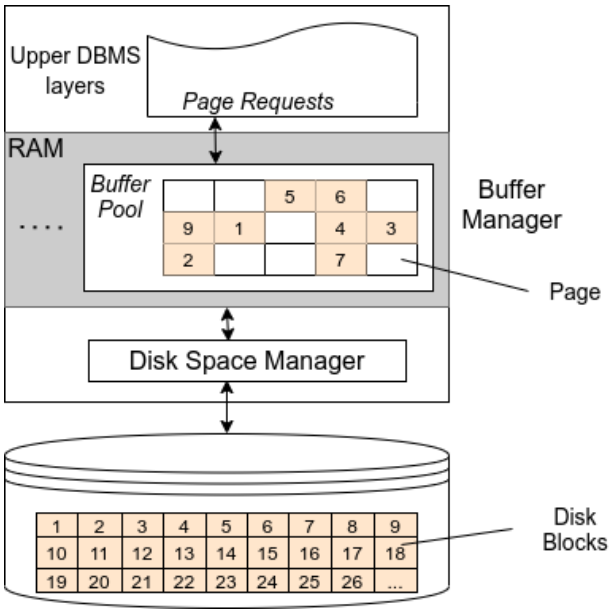


Figure 13 *A visualization of the interaction of a database with memory [63].*

files. For example entities and relations can be stored in different files with references to each other, thus enabling the layout of these two to be specialized to their structure and usage in queries.

Files may either organize their records in random order (heap file), sorted or using a hash function on one or more fields. All of these approaches have upsides and downsides when it comes to scans, searches, insertions, deletions and updates.

To mitigate the effect that result from selecting one file organization or another, one record organization or another, the concept of an index has been introduced. Indexes are auxiliary structures to speed up certain operations or queries that depend on one field. Indexes may be clustered or unclustered. An index over field F is called clustered if the underlying data file is sorted according to the values of F. Otherwise the index is called unclustered In a similar way indexes can be sparse or dense. A sparse index has less index entries than records, mostly one index entry per block. This can of course only be done for clustered indexes as the sorting of the data file keeps the elements between index keys in order. An index is dense if there is a one to one correspondence between records and index entries. All unclustered indexes are dense indexes. There are different variants of storing index entries which have again certain implications on the compactness of the index and the underlying design decisions. Finally, there are operators that act upon and use the above structures and mechanisms. Logical operators define an algebraic operation used to process a query. Physical operators implement the operation described by logical operators. For each logical operator there may exist multiple different physical implementations using different access methods.

All these considerations make choosing different file splits, layouts, orderings, addressing schemes, management structures, de-/allocation schemes and indexes a complex set of dependent choices. These depend mainly on the structure of the data to be stored and the queries to be run.

3.2 Graph Databases

Relational databases store data in tables. The links considered in this category of DBMS are mostly used to stitch together the fields of a record stored in different tables into one row again,

after it has been split to satisfy a certain normal form. Of course one may also store tables where one table stores nodes and the other table's fields are node IDs to represent relationships.

However, in order to traverse the graph, one has either to do a lot of rather expensive look ups or store auxiliary structures to speed up the look up process. In particular when using B-trees as index structure, each look up takes $\mathcal{O}(\log(n))$ steps to locate a specific edge. Alternatively one could store an additional table that holds incidence lists such that the look up of outgoing or incomming edges is only $\mathcal{O}(\log(n))$ which would speed up breadth first traversals, thus duplicate data. But still one has to compute joins in order to continue the traversal in terms of depth. Another way to speed things up is to use a hash-based index, but this also has a certain overhead aside from the joins.

In contrast to relational databases, native graph databases use structures specialised for these kinds of queries.

The Property Graph Model

The property graph model is a widely adopted data model to represent graphs in databases. It is not only able to represent the structure of directed or undirected, weighted or unweighted, but also of typed graphs having additional properties.

A **Property Graph** is a 9-Tuple $G = (V, E, \lambda, P, T, L, f_P, f_T, f_L)$ with

- V the set of vertices.
- E the set of edges.
- $\lambda : (V \times V) \to E$ a binary relation assigning a pair of nodes to an edge.
- P a set of key-value pairs called properties.
- T a set of strings used as relationship types.
- L a set of strings used as labels.
- $f_P : V \cup E \to 2^P$ a binary relation that assigns a set of properties to a node or relationship.
- $f_T : E \to T$ a binary relation that assigns a type to a relationship.
- $f_L : V \to 2^L$ a binary relation that assigns a node a set of labels.

The property graph model reflects a directed, node-labeled and relationship-typed multigraph G, where each node and relationship can hold a set of properties [5, 66, 67]. In a graph the edges are normally defined as $E \subseteq (V \times V)$, but in the property graph model edges have sets of properties and a type, which makes them records on their own. An illustration of this model is shown in 14.

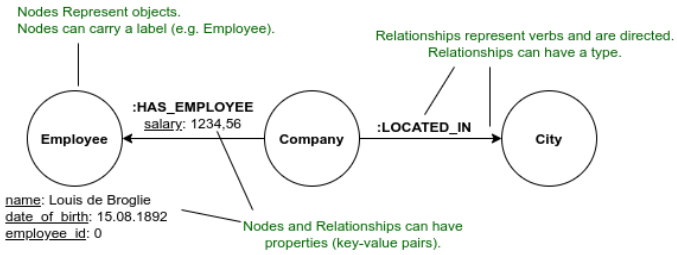


Figure 14 *A schematic visualization of the property graph model.*

Neo4j is a graph database employing the property graph model [65]. The logical operators of this model are described in [40]. The *get_nodes*-operator returns all nodes of the graph. This

means that however the nodes are stored, the whole file (portion) needs to be scanned. Further the *expand*-operator returns the incident edges of a node depending on the direction. Expand only considers a part of the set of all edges, so it does not do full scans but rather smaller reads. Finally the *filter*-operand selects certain nodes or relationships based on properties, labels or relationship type.

In the next section 3.2 we are going to discuss how Neo4J implements the property graph model, with our focus on the structure of the graph and the low level storage scheme.

Example: Neo4J

Neo4J is a native graph database using the property graph model. The source code of the community edition is available at GitHub [27]. We look at some implementation details of the storage and buffer manager, as well as the record structure. We are not going to take properties, relationship types, labels and concepts related to those into account.

High-level Architecture

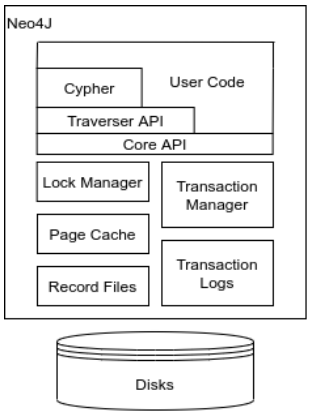


Figure 15 *The high level architecture of Neo4J [65].*

To get an overview of the architecture let us consider figure 15. This description was outlined by the co-founder of Neo4J Emil Effrem, the chief science officer Jim Webber and Ian Robinson who was an engineer at that time at Neo4J in their book on graph databases [65]. Here we can see that the architectural schema outlined in 3.1 and especially 12 was not quite applied.

Still the components are very similar: The "Page Cache" is equivalent to the buffer manager, the record files are what is managed by the disk space manager, mechanisms to deal with free slots [59] and (de-)allocations [60] are also part of the software stack, as are the record formats [61] and indexes, corresponding to the access layer. The corresponding components are just put together in a slightly different manner.

The detailed composition is shown in 16. The IO package contains the page cache, which is basically the buffer manager. It also contains facilities to create, grow and shrink files using the `java.nio` library and wrappers arround platform dependent allocation facilities. Thus the (de-)allocation part of the disk space manager resides in the IO package, too. The record storage engine defines the record format and the file layout, as well as means to create and maintain indices, thus it is similar to the access layer. It also handles the management of free slots something usually done by the disk space manager. To summarize: The buffer manager and the access layer correspond closely to these two packages, while the disk space manager is distributed mainly over these two packages.

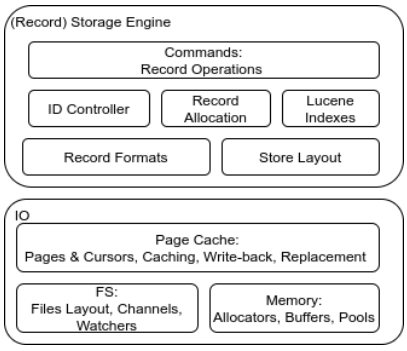


Figure 16 *A visualization of the broad the storage and memory organization of Neo4J.*

Record and File Structures

Neo4J uses several different record types. They can be split broadly in the following categories:

- Variable size records: Strings, Arrays
- Fixed size records:
  - Graph structure related records: Nodes, relationships, relationship groups
  - Properties, labels, relationship types

Each record type is stored in an own file per database in the database management system. An additional system database keeps track of the existence and metadata of the other ones storing user data. This is visualized in 17.

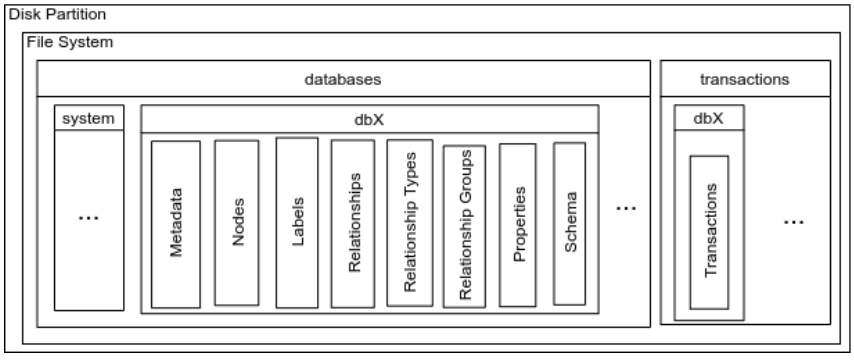


Figure 17 *A visualization of how the files are arranged of Neo4J.*

The records are ordered simply by their insertion order, i.e. the files storing the records are heap files. While variable length records store strings and arrays, labels for example store a pointer to the actual string of the label to be fixed size and thus efficiently retreived. The same is true for relationship types and property keys and values that are stings or arrays. This is done to avoid duplications of strings e.g. of each label. As mentioned before for the sake of succintness we are just going to elaborate on the elements that represent the graph structure. Only one thing is to be mentioned: Properties are stored as a linked list for each nodes and relationships.

Nodes

The record format of nodes consist of a 15 byte structure. The IDs of nodes are stored implicitly as their address. If a node has ID 100 we know that its record starts at offset 15 Bytes$\cdot$100 = 1500 from the beginning of the file. The struct of a record looks like this:

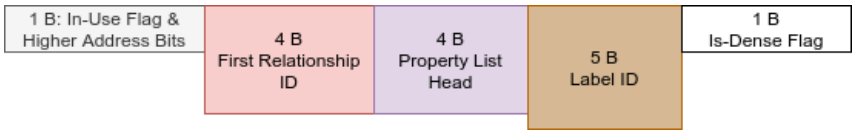


Figure 18 *A visualization of the record structure of a node record [23].*

1. Byte 1: One bit for the in-use flag. The additional bits are used to compress the node struct by using the other 7 bits to store the most significant bits of the first relationship ID and the first property ID

2. Bytes 2 — 5: The next 4 Bytes represent the relationship ID of the head in the linked list of relationships of the considered node.

3. Bytes 6 — 9: Again 4 bytes encode the property ID 0of the head in the linked list of properties of the node.

4. Bytes 10 — 14: This 5 byte section points to the labels of this node.

5. Byte 15: The last byte stores if the node is dense, i.e. has an aweful lot of relationships, such that it needs special treatment in order to remain efficient to traverse over. That is a relationships are grouped by type and direction for this node.

To summarize: The records on disk are stored as in the enumeration above and as shown in 18. In the database all IDs get mapped to longs and their respective space is larger than the space representable by 35 bit — what is perfectly fine.

On disk 4 byte integers are used to store the 32 lowest bits of the respective addresses and the higher bits are stored in the first byte that also carries the in-use bit.

Relationships

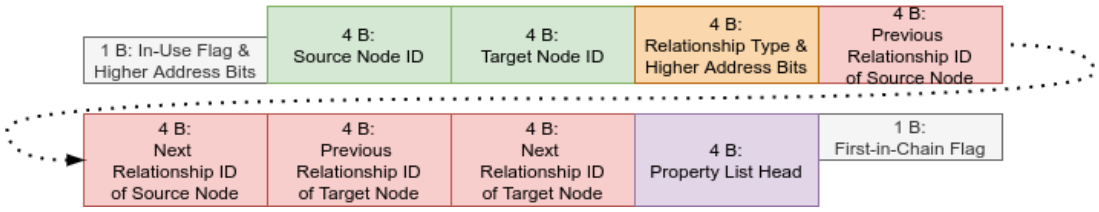


Figure 19 *A visualization of the record structure of a relationship in Neo4J.*

Relationship records are stored with implicit IDs too. Their fixed size records contain 34 bytes. Besides an in-use flag, the source and target node IDs, and the relationship type, the record also contains two doubly linked list: One for the incident edegs of the first node and one for the incident edges of the second node. Next a link to the head of the linked list of properties for this relationship is stored. Finally, the last byte contains a marker if this relationship is the first element in the incidence list of one of the nodes.

1. Byte 1: In-use bit, first node high order bits (3 bits), first property high order bits (4 bits)

2. Bytes 2 — 5: first node ID

3. Bytes 6 — 9: second node ID

4. Bytes 10 — 13: relationship type (16 bit), second node high order bits (3 bits), relationship previous and next ID higher bits for first and second node ($4 \cdot 3 = 12$ bits), one unused bit.
5. Bytes 14 — 17: previous relationship ID for first node
6. Bytes 18 — 21: next relationship ID for first node
7. Bytes 22 — 25: previous relationship ID for second node
8. Bytes 26 — 29: next relationship ID for second node
9. Bytes 30 — 33: link to the first property of the relationship
10. Bytes 34: A marker if this relation is the first element in the relationship linked list of one of the nodes stored in the lowest two bits of the byte. The other 6 bits are unused.

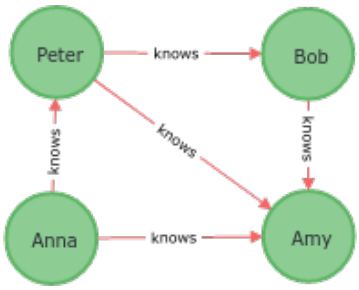


Figure 20 *An example graph.*

The relationship structure is a key element of the layout and reveals the actual data type that the database is using: Nodes and relationships are both stored once only (i.e. nothing is duplicated). Without taking the fields into account, this is an unordered edge list. When taking the linked lists of relationships into account, it turns out that the underlying data structure is that of an indicence list, with a couple of additional properties.

First, as already mentioned, the edges are not physically duplicated but only referenced. The records are fixed sized, so addressing them is done by multiplying the index by the size of an entry, meaning one does not need to store primary keys explicitly and adress translation can be done using a simple multiplication. Theoretically one could align the record size to a power of two to turn the multiplication into a bit shift. Next, as doubly linked lists are used, the deletion of an edge is in $\mathcal{O}(1)$ if the ID is known. If this was not the case, the incidence list would need to be traversed in order to find the previous element. Also, the incidence list is stored in the relationships. Thus in order to traverse from one node's incidence list to another, there is no need to load the node record itself. It suffices to just dereference the next element in the incidence list stored by the relationships, along with storing the ID of the edge that started the traversal. This makes the assumption, that the doubly linked incidence lists is circular, i.e. the head's pervious element is the tail and reciprocally.

To conclude this example, we briefly visualize the just described storage schema. The high-level graph is shown in 20. It contains four nodes and five edges. The underlying instantiated data structures are shown in 21. The light red arcs represent edges, the light green circles nodes. The colored boxes on the edges and nodes represent the data structures. The brighter red edges represent the doubly linked incidence lists. Notice, that the heads and tails of the doubly linked incidence lists are marked by "X" to avoid to draw additional edges. The brighter green arrows represent the source and target nodes, as stored by the edges.

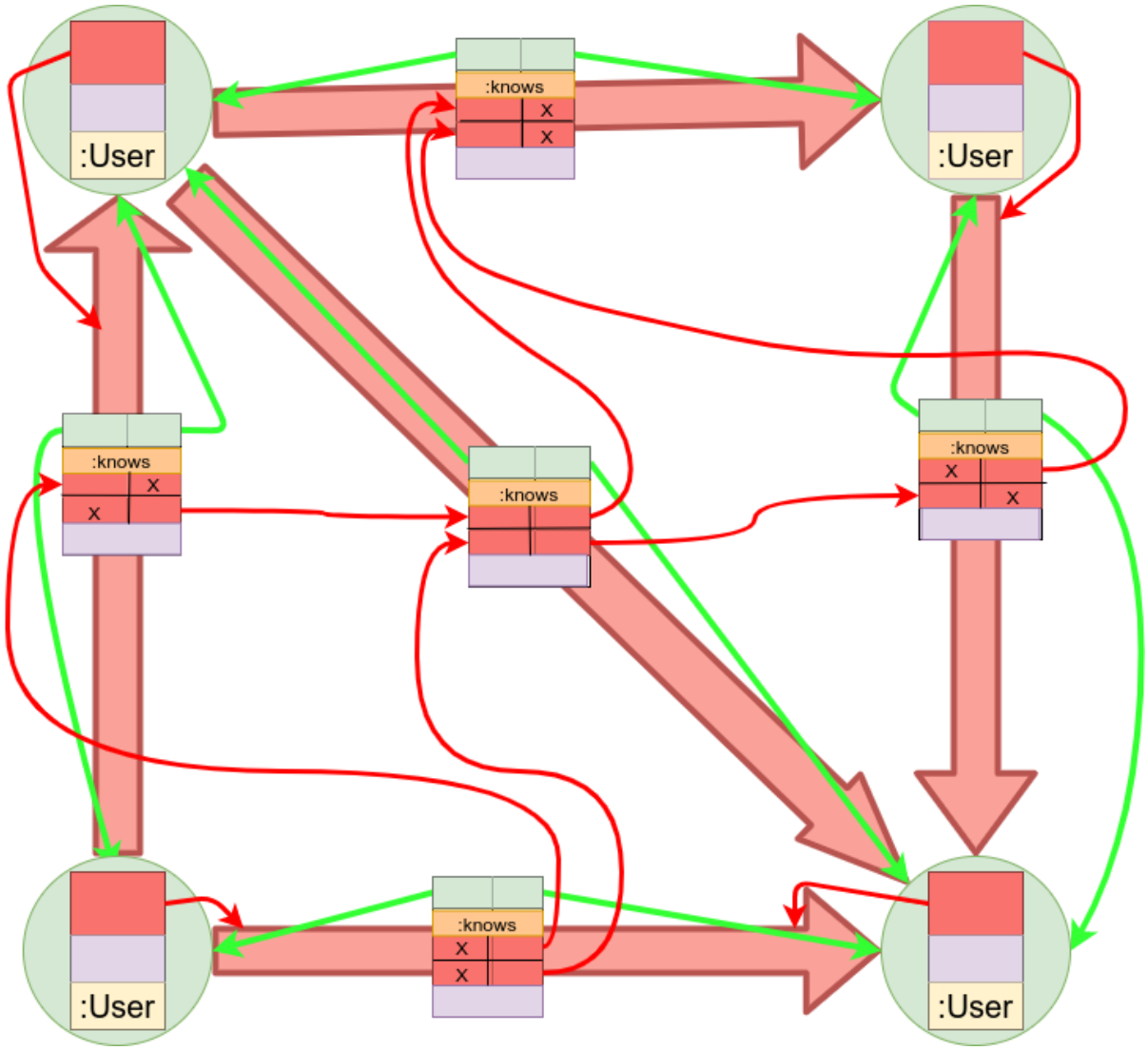


Figure 21 *Visualization of the data structures, as initialized by the example graph shown in 20.*

4 Problem Definition

4.1 Locality

The memory hierarchy was introduced in 3.1 and 11. In summary it tries to unify the strengths of fast, low capacity memory — caches (SRAM) —, with slower but larger memory — main memory (DRAM), with orders of magnitude slower but orders of magnitude larger memory — disks (HDD) and more recently flash memory (SSD, SD-Cards). But how can this actually work? Given that only a tiny fraction of fast memory is available to hold the necessarry parts, while additional loads of data are transfered in time — "desirably fast enough".

The key principle for the memory hierarchy to work is what is called *locality of reference* in the literature [43, 74]. This principle expresses, that most programs do not access their address space uniformly or randomly, but rather tend to access small subsets of all addresses in certain time intervals, depending on the program state. Locality comes in two flavors [17]:

- *Temporal locality* refers to the number of other references between two accesses of the same memory location.
- *Spatial locality* refers to the number of accesses and the radius of the neighbourhood that is accessed in a number of steps.

If the same location is accessed multiple times in a short amount of time, the temporal locality is high. Thus temporal locality can be measured using reference frequencies. From a bayesian point of view, one can say that temporal locality is the probability of an object being rereferenced after the first usage [33].

$$P(X_{t+\Delta} = A | X_t = A)$$

X_t is the reference at time step t, A is an address and Δ is a parameter, which depends not only on the system specifications (like the CPU and memory clock), but also on the program and the scale of interest.

If a small range of addresses is accessed very often then spatial locality is high. If the range is limited to one address, then spatial locality is equivalent to temporal locality. Thus temporal locality is a special case of temporal locality [33]. With ε a radius we can characterize spatial locality by:

$$P(X_{t+\Delta} = A \pm \varepsilon | X_t = A)$$

Spatial locality is thus a function of time Δ and neighbourhood range ε.

In order to leverage these concepts, several components profile the memory usage. In the memory hierarchy, all on and off chip caches (i.e. SRAM) are handled by hardware [43].

At the level of main memory (DRAM), the operating system manages what is fetched, buffered and evicted from disk to main memory. The optimal buffering strategy is to load what is needed before its usage and evict the objects whos usage is furthest in the future [74].

When it comes to eviction the best approximation to the optimal strategy is the least recently used algorithm. It aims to keep things in memory, that have the highest chance to exhibit temporal locality. That is, the things in memory, that have not been referenced for the longest time, have a lower chance to be temporally local in the future [68]. Put differently *caches and buffers exploit temporal locality.*

As this information is not available in general, objects are loaded when they are referenced, often with additional addresses which are hoped to be needed, too — this is called prefetch

or predicitve fetching [71, 43]. Prefetch tries to exploit spatial locality. There are several components that try to exploit this:

- Compiler generated prefetches: The compiler knows what addresses the program accesses in which sequence and tries to minimize the time that is spent waiting for IO. This is called instruction scheduling [2]. Other compiler generated heuristics are applied e.g. in domain specific compilers, like in the TVM compiler for neural networks [13].

- The operating system may use specialized data structures and algorithms to estimate, if prefetching should be done, based on the previous accesses. An example is the "spatial look-ahead" algorithm by Baier and Sager [43, 8], but there exist many more e.g. [45, 31, 50, 15]. Most of these methods are capable to find correlations between addresses and their neighbourhood, file accesses and pointed-to objects.

- A special role in the context of prefetches and spatial locality take databases. As these are not only able to predict content-based correlations, e.g. by knowing what table is queried in the case of relational databases, but also can augment data by using auxiliary data structures like indices. The most remarkable capability in this context is to be able to reorganize data, based upon how it is queried. Relational databases store data in tables and often sort these tables based upon either a certain field (like the primary key) or a set of fields. This in combination with being able to analyze the query before executing it allows reordering the memory accesses, such that as many accesses as possible are sequential [63, 69].

Spatial locality depends on how data is ordered: If semantically closely coupled data is spread out as wide as possible, the program or file of interest will harldy exhibit locality. As an example consider a program with n instructions, with logical addresses from $0, \ldots, n-1$. An inversion is a change of position of two lines l_1, l_2, such that the line that gets executed earlier l_1 has a higher address than one that gets executed later l_2. Such a program can maximally have $\frac{n(n-1)}{2}$ inversions. If it has that many inversion, the program is layed out in the opposite direction and the spatial locality would be similar to the original program. Thus lets assume only every second instruction is misplaced. In effect, to execute the program two pages must always remain in memory instead of one and the radius of the neighbourhood doubles.

In short: The layout of the data or records in the address space — on file or in memory — is crucial to the concept of spatial locality. Achieving optimal temporal locality is a matter of grouping and ordering data such that what is referenced together is in a neighbourhood in terms of addressing.

4.2 Problem Definition

In order to optimize spatial locality for traversal-based queries, the graph needs to be grouped and ordered. Ultimately disk storage is block-based and disk access is page-based. That is the vertices and edges must be grouped into blocks.

Assumptions: In the remainder of this thesis we are assuming that the graph is represented in the property graph model 3.2 and uses incidence lists 2.2 as the storage schema. We are not taking properties, labels and relationship types into account. We are focusing on spatial locality here, that is the page replacement algorithm is fixed but arbitrary. Finally in the remainder of the thesis when talking about traversal-based queries, we mean all queries described in 2.3, but the random walk.

Problem Definition: Given a graph G, logical block size b, page size p.
Desired is

1. A partition of G into blocks of vertex records V_i and E_i relationship records,
2. orderings or permutations π_v, π_e of the blocks of vertex and edge records V_i, E_i,
3. a reordering of the incidence list pointers

such that spatial locality is as high as possible for traversal-based queries.

As partitioning a graph optimally [4], as well as finding an optimal linear arrangement [25] are both NP-complete problems [51], we use the formulation "as high as possible" instead of optimal or maximal.

In order to measure the spatial locality we introduce two measures that are used in the evaluation chapter:

1. Number of block accesses.
2. Number of non-consecutive block accesses.

The first measure is to take the neighbourhood within a block into account: If vertices and edges that are accessed together are stored in the same block, this measure should be as small as possible. The second measure takes the order of the blocks into account: If vertices and edges that are connected or "close" to each other are stored in adjacent blocks, they can be loaded with one sequential read. But the second measure also takes into account how the traversal is executed in terms of pointer chasing with respect to the incidence list.

4.3 Example: Vertex, Edge and Incidence List Order

Why are these three criteria neccessary? Why are there only two measurements for three criteria? This is what shall be explained in an example. Something that is of importance for the traversal — but not as straight forward to see as node and edge grouping and order — is the order of the pointers in the incidence list, as we are going to see.

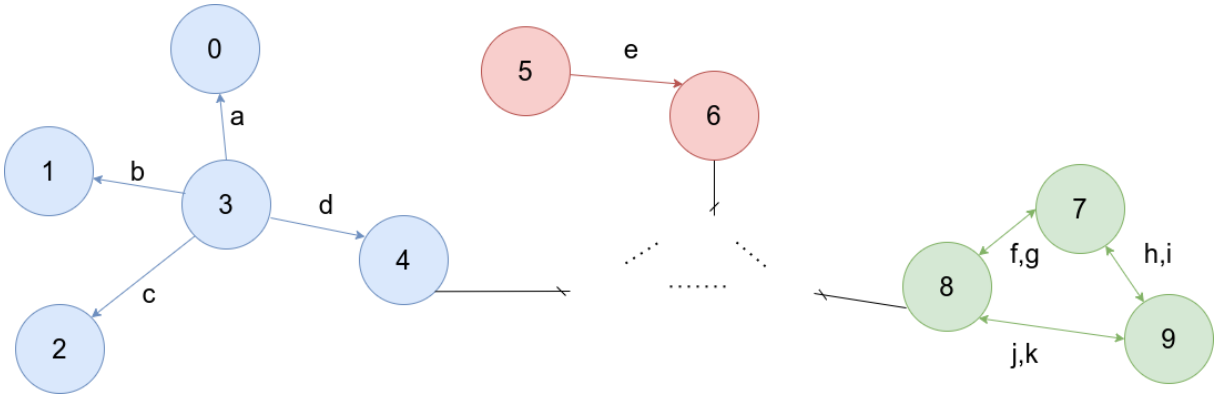


Figure 22 *Parts of a graph that is used in subsequent examples. Cut through edges mean edges to any non-visualized component of the graph. The dotted lines indicate, that other nodes and edges are between the three shown components.*

The graph used in the below example looks as shown in 22. We use a storage schema that is motivated by the one of Neo4J, that is nodes and relationships are stored in separated files, the incidence list is stored in the records of the edges and the nodes conatain a pointer to the head relationship of their incidence list each. Further we assume that we can only read

sequentially, if the blocks are directly adjacent. As this is a rather small example for the sake of succintness, things are just shown on a conceptual level. We make the assumption that 3 nodes or 2 relationships fit onto a disk block. Realistically 8 to 16 nodes and 5 to 8 relationships fit on a 512 byte disk block. An average graph in the stanford network analysis platform graph dataset collection has thousands of nodes and edges. Taking the californian road network graph as an example, the whole graph would take

$$1965206 \text{ nodes} \cdot 35 \text{ bytes per node} \cdot 512^{-1} \text{ bytes per block} = 134341 \text{ blocks}$$

to store all nodes and

$$2766607 \text{ relationships} \cdot 72 \text{ bytes per relationship} \cdot 512^{-1} \text{ bytes per block} = 389055 \text{ blocks}$$

blocks to store the relationships in Neo4J. To summarize, the principles shown below scale with the graph size and for realistic assumptions, these conditions emerge.

First consider the split of vertices and edges into blocks in the upper table in 2. None of the vertices in the blocks are neighbouring to each other. Thus when traversing the graph, each step requires to load a block. The same is true for the edges: None of the edges in the same block are connected to the same vertex. Each edge causes a page fault and a load of another block(s). This may happen in current state of the art graph databases like Neo4J. The placment into blocks is currently by insertion order, thus depends on the ordering of the input dataset. On the other handside in the lower table in 2, the vertices are grouped into blocks according to their neighbourhood and the edges are grouped by the vertices they are connected to.

node.db	0, 5, 7	1, 4, 9	2, 6, 8	3		
edge.db	a, f	b, g	c, h	d, i	e, j	k

node.db	7, 8, 9	0, 1, 3	6	4, 2, 5,		
edge.db	f, h	g, k	i, j	a, b	e	c, d

Table 2 *An example of suboptimal and improved record placement into blocks. The block size is assumed to be only 3 vertex records and 2 node records respectively. For larger block size, the same principle applies.*

Next 3 shows two different orderings of the blocks, this time with a focus on the edges only. In the upper table, an edge is stored in the neighbourhood of its source node, but appart from its target node. Thus two single reads are required to go from one vertex over an edge to another vertex and retreive it's incident edges. In the lower table, the blocks are adjacent and one sequential read is enough to go from source to target and fetch the target't incidence list.

edge.db	f, h	a, b	i, j	e	c, d	g, k

edge.db	f, h	g, k	i, j	a, b	c, d	e

Table 3 *Suboptimal and improved block order.*

Finally: Consider the visualization of the incidence list of the node 3, given the page placement in the upper part of 2 in 4. Theoretically the only difference is the order in which the list points to the relationships. In terms of traversed blocks, we need to do four single reads when using the order given in the upper table. If we rearrange the pointers according to the lower table, the list may be loaded sequentially with one read operation instead of four single reads. This phenomenon may also appear when the blocks are formed and ordered in an improved way, but

only when scaling up to a graph with larger neighbourhoods. Given that an edge can be placed either near the other edges of the source or the target node, the impact when jumping back and forth will grow along with the gaps between blocks in which the edges are stored. Finally if the blocks are cached, this induces a higher load on the buffer manager, as more pages need to reside in memory at the same time and as they are frequently rereferenced instead of being read once and evicted then. The higher the degree — i.e. the length of the incidence list — of the node, the more severe the effect.

incidence list of node 3	c	a	d	b

incidence list for node 3	a	b	c	d

Table 4 *Suboptimal and improved incidence list order.*

5 Related Work

5.1 G-Store: Multilevel Partitioning

G-Store is a disk-based storage manager for graph data implemented and published by Steinhaus et al. [73]. To the best of the authors knowledge, this is the first structured approach to improve locality in graph databases by altering the placement of records into blocks. More specifically, in order to maximize performance, they try to place adjacent nodes close to each other, such that they can be read sequentially. The rearrangement of the records is done when importing a new data set and is static after insertion. G-Store uses adjacency lists as data structure and does not store these in an own file but directly next to the vertex in the very same file. The broad schema of the placement method developed by Steinhaus et al. [73] is derived from multilevel partitioning methods, that is described in 2.3.3. Briefly, the multilevel partitioning algorithm works in three steps: Coarsening, Turn-around and uncoarsening. The coarsening phase tries to reduce the original graph to a smaller one that braodly preserves the structure of the underlying graph. A more expensive partitioning algorithm can then be applied to this smaller graph to solve the actual problem approximately and fast. During the uncoarsening, the approximate solution is refined and the coarser graph is projected back until the original graph is mapped and restored. Finally in a last step the partitions are mapped to actual blocks. Note how similar the procedure is to the actual multilevel partitioning algorithm. Thus we are more interested here in the differences from the reference algorithm. A broad overview of the method is shown in 23.

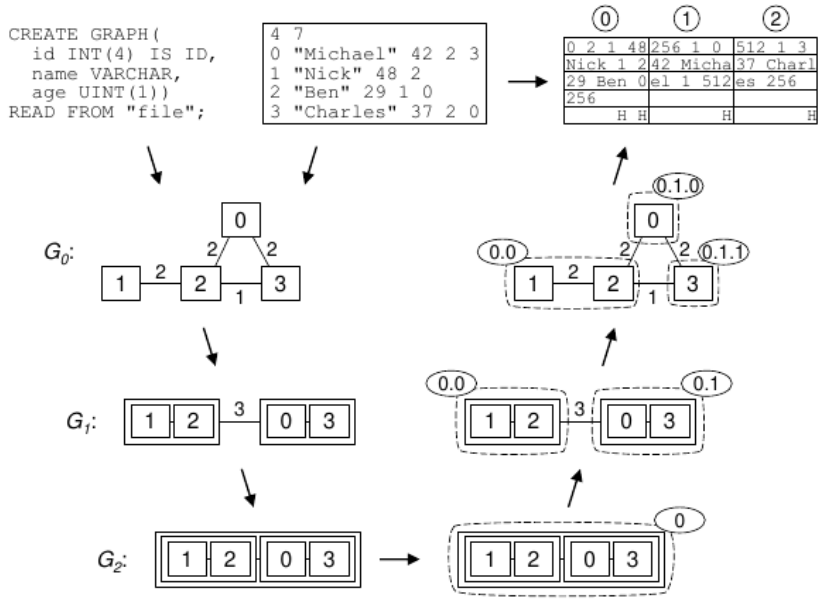


Figure 23 *A broad overview of the multilevel partitioning method applied by G-Store [73].*

In the next part of this section we are going to use the notions of a finer and a coarser graph a lot. Let therefore $G = G_0 = (V_0, E_0)$ the original graph, and $G_i = (V_i, E_i)$ the graph that was coarsened i times.

Coarsening

The coarsening phase of G-Stores partitioning algorithm is what is called heavy edge matching (HEM) in Karypis formulation of the algorithm as already mentioned. Thus the algorithm takes G_i as parameter and returns G_{i+1} along with a projection Z_{i+1} specifying which finer nodes maps to which coarser node. Coarsening proceeds in the original algorithm until a certain lower limit of vertices is reached. In the METIS implementation this is hard-coded to $\max(20k, 40\log_2 k)$ where k is a user defined parameter specifying the number of partitions. The algorithm used in G-Store keeps coarsening until there are no edges, i.e. only one vertex. It can thus be seen as a form of hierarchical agglomerative clustering [70] with the inverse edge weight acting as a distance function.

Another modification is that in contrast to METIS [46], not only two edges are matched at a time but possibly many and that there is an upper limit to the vertex weights. This depends on the coarsening factor $c(i) = \frac{|V_i|-|V_{i+1}|}{|V_i|-|\{v \in V | N_v = \emptyset\}|}$, with i the level of coarsening, where $i = 0$ it the original graph. The counter is the number of vertices that were reduced and the denominator is the number of nodes in the larger graph minus the irreducible nodes. Initially the allowed vertex weight θ is the size of a block. If this factor $c(i) < 0.3$ then the node weight θ is doubled as long as $\theta \leq 32$. Otherwise the number of nodes that are allowed to be matched is incremented.

Turn-Around

In G-Store the turn-around simply assigns every vertex in the coarsest graph, whose weight is larger than the size of a block, a distinct partition number. The other nodes are added up until their weight reaches the size of a block and are assigned a partition number together. Thus the algorithm accepts a fully coarsened graph G_i and returns the partition numbers for this graph ϕ_i.

This step is completely different than in the reference multilevel partitioning algorithm.

Uncoarsening

The uncoarsening phase consists of three different steps that are performed per level: Projection, reordering and refinement. Projection constructs a first mapping, reordering swaps partitions and refinement exchanges nodes between partitions. Each iteration of the full uncoarsening procedure takes the coarser graph G_{i+1} as an argument along with its the partition numbers ϕ_{i+1} and the mapping Z_i and returns the one level uncoarsened graph G_i, along with the respective partition numbers ϕ_i. The algorithm also defines a weight threshold per level. With $\overline{c}$ the average coarsening factor

$$\chi_i = \lfloor \frac{\text{block size}}{(1-\overline{c})^i} \rfloor.$$

Further it defines three objective functions:
The first function is closesly related to the minimal linear arrangement problem [51] and expresses that nodes that share an edge shall be minimally far appart from each other.

$$\min C_1 = \min \sum_{(u,v)\in E} |\phi(u) - \phi(v)|$$

The second objective function aims to reduce the overall number of edges between the partitions.

$$\min C_2 = \min \sum_{(u,v)\in E)} \begin{cases} 1 & \phi(u) \neq \phi(v) \\ 0 & \text{otherwise} \end{cases}$$

The last objective function penalizes the number of blocks that are linked. That is in order to reduce the number of overall linked blocks according to [73].

$$\min C_3 = \min \sum_{i \leq j} \begin{cases} 1 & \phi(u) = i \wedge \phi(v) = j,\ (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

Finally there are two functions that are similar to the first objective function that are used in the projection step called tension and modified tension: The tension is the weighted distance between the block of the vertices that share and edge. Let $v \in V_i$.

$$t(v) = \sum_{u \in N_v} w_{(u,v)} \phi_i(v) - \phi_i(u)$$

The modified tension is just the same, but instead of using the actual graph, one uses the coarser graph G_{i+1} and the projection Z_i to estimate the tension:

$$t'(v) = \sum_{u \in N_v} w_{(u,v)} \phi_{i+1}(Z(v)) - \phi_{i+1}(Z(u))$$

Projection

This part of the algorithm constructs a first version of the finer-grained partition numbers ϕ_i from the coarser ones ϕ_{i+1}. The enumeration follows a dewey numbering scheme. That is per level one place in the partition label is added. If the vertex $\phi_{i+1}(Z(v)) = 2$ and vertex $\phi_i(v) = 1$, then the overall partition label of the vertex is 2.1.

Per partition in the coarser level graph, the algorithm either simply assigns the same partition number to all nodes $v_i \in Z(\phi i + 1, j)$ in the finer graph G_i that were clustered into the recpective coarser nodes, if the overall weight of the partition in the coarser graph is smaller than the weight threshold: $w_{\phi i+1,j} \leq \chi_i$. Otherwise the modified tension is computed is computed for all nodes of the partition in the finer graph $\forall v_i \in Z(\phi i + 1, j) : t'(v_i)$. The minimal tension is extracted, placed to the leftmost free position in the partition and the tensions of the neighbours are updated. This is done repeatedly until all nodes are placed. During this step, each partition is marked with a flag, that is true, when it was created from nodes in the right half of the coarser partition.

The projection differs vastly from the reference algorithm that just assigns the coarser partition number to all nodes in the finer grained graph.

Reordering

This step is non-existent in the multilevel partitioning algorithm. When swapping the above created partitions, only C_1 changes, but not C_2 or C_3. Using the just created flags, groups are identified which span two half coarser partitions $\phi_{i+1,j}, \phi_{i+1,j+1}$. Per group the finer partitions are swapped as long as the overall absolute tension (C_1) decreases by some swap. In effect, this is a fix-point computation.

Refinement

Finally the refinement step tries to optimize a weighted sum of all the objective functions by moving vertices to other partitions. In the reference algorithm this step simply uses the gain as defined in 2.3.3. Three used-defined parameters α, β, γ control the weighting, another one specifies the number of iterations that shall be executed r. In each iteration, the algorithm steps over the partitions $\phi_{i,j}$ and creates a two dimensional array A of dimension $|\phi_{i,j}| \times |P_{i,j}|$ with $P_{i,j} = \{\phi_i(u) | v \in \phi_{i,j}, u \in N_n\} \setminus \phi_{i,j}$. Each entry is defined with v the vertex that is to be moved to partition k:

$$a_{v,k} = \alpha C_1 + \beta C_2 + \gamma C_3 + \lambda$$

Where λ penalizes overfull blocks or rewards the filling of less filled blocks.

5.2 ICBL: Diffusion Set-based Clustering

Yaçar and Gedik propose another method to form and order blocks [80, 81].

They define one metric for each of the task: *Block locality* is defined by the means of conductance and cohesiveness. Conductance is defined as the ratio of edge cuts to total edges in a block:

$$C_d(B) = \frac{|\{(u,v) \in E : |\{u,v\} \cap V_B| = 1\}|}{|\{(u,v) \in E : |\{u,v\} \cap V_B| > 0\}|}$$

Cohesiveness is the number of nodes in the same blocks that are connected by an edge divided by the number of theoretically possible edges, i.e. $|V|^2$ in a directed graph. The authors assume an undirected self-loop free graph, thus the number of possible edges is $\frac{|V|(|V|-1)}{2}$.

$$C_h(B) = \frac{|\{(u,v) \in E : u,v \in V_B\}|}{|V|^2}$$

As conductance takes edges between blocks into account and cohesiveness meausures the edges within a block, they are complementary [80]. Thus the locality of a block us defined as the geometric mean of the measures above, where the conductance is subtracted from one:

$$L(B) = \sqrt{C_h(B) \cdot (1 - C_d(B))}$$

Ranking locality is related to what we called tension before. Here we do not measure it between partitions of the node but directly to the position in the order. Let $v \in V$ a node and $r(v)$ a function that assigns a natural number in the range of $\{0, \ldots, |V| - 1\}$ to each vertex.

$$R(v) = \sum_{u \in N_v} r(v) - r(u)$$

The locality of a block is then defined by one minus the normalized average distance for all vertices in the block:

$$R(B) = 1 - \frac{1}{(|V| - 1)} \sum_{v \in V_B} \frac{R(v)}{|N_v|}$$

ICBL is an acronym for the single steps performed by this algorithm. It is designed to be implemented and executed using the map-reduce model. First ICBL extracts so called diffusion sets as features. Then it clusters the vertices based upon these features, to split the graph into subgraphs. After that it hierarchically clusters the subgraphs obtained in the previous step in order to form blocks. Finally the just formed blocks are ranked and layed out on disk. A brief visualization of the method is shown in 24.

Identify Diffusion Sets

The diffusion set $\mathcal{D}_v$ of a vertex $v \in V$ is a characterization of the surrounding of a vertex. A surrounding means other vertices that are reachable in a certain number of steps. To identify the diffusion set, ICBL carries out t random walks (2.3.1, 3) of length l. The authors characterize a random walk using the vertices so from out definition we need to extract the multiset of vertices from the walk.

For choosing the parameters t and l Yaçar and Gedik propose heuristics: For t, construct the cumulative degree distribution $f(d) \mapsto P(x \leq d)$ and chose the minimal degree value such that the derivative of the cumulative degree distribution $f'(d) = 1$. The value of the derivative was derived empirically. Regarding l the authors make the assumotion that the network exihibits the small world phenomenon [48]. In a network that has that property, the probability is high

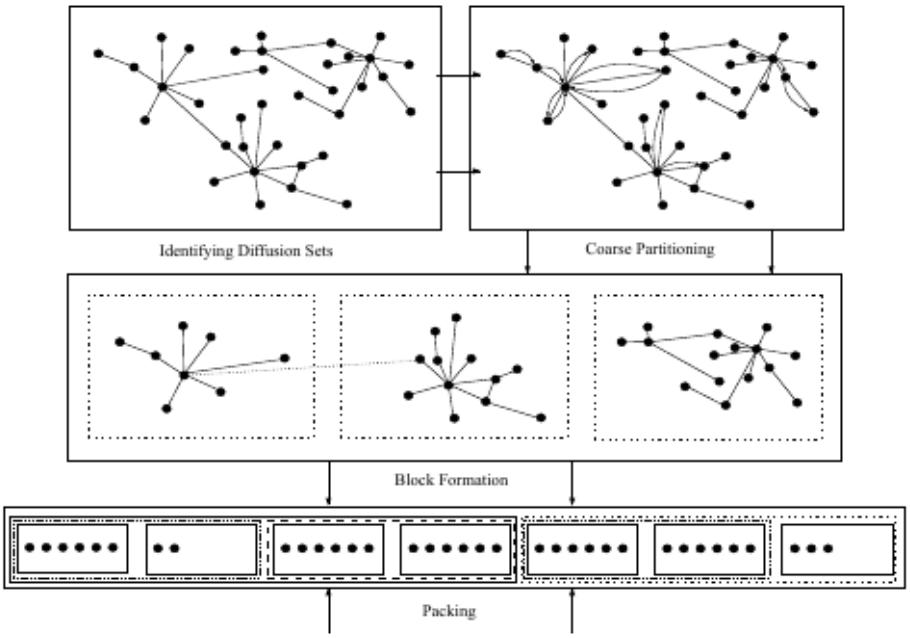


Figure 24 *A broad overview of the ICBL method by Yaçar and Gedik [80].*

that there exists a path between any two nodes $u, v \in V$ of length $\ln |V|$. In order to construct diffusion sets that are characterizing, the length of the random walk should not be too long as all nodes might be visitable then. Thus the heuristic for choosing l is $1 + \lceil \frac{\ln |V|}{k} \rceil$ where k is the number of clusters in the next step.

Coarse Clustering

After generating the diffusion sets, the graph is clustered using a variation of the k-Means algorithm [52]. It is used to partition the graph into k smaller subgraphs, such that the computationally more expensive agglomerative hierarchical clustering [70] algorithm that is used in block formation can be executed in parallel. Instead of using a geometric distance function (like the manhatten or the euclidean distance between points on a plane), an alternated version of the Jaccard distance function [42] is used. The Jaccard function $J(u, v) = 1 - \frac{\mathcal{D}_u \cap \mathcal{D}_v}{\mathcal{D}_u \cup \mathcal{D}_v}$, i.e. the distance is the number of common elements divided by the set of all elements in both sets. The altered distance is adjusted for multisets:

$$J_w(u, v) = 1 - \frac{\sum_{x \in \mathcal{D}_v \cap \mathcal{D}_u} \min(w_{\mathcal{D}_v}, w_{\mathcal{D}_u})}{\sum_{x \in \mathcal{D}_v \cup \mathcal{D}_u} \max(w_{\mathcal{D}_v}, w_{\mathcal{D}_u})}$$

First k initial centers are chosen based upon the node degree and the distance to the already chosen centers. Then all nodes get assigned to the closest center. After that the centers are updated, by building the union of all diffusion sets and use the vertex with the highest weight as new center. This is done until the centers do not change further. In order to determine the number of clusters, the authors propose a heuristic that is based on the available memory M and the size of a vertex and the average diffusion set $s = \text{sizeof}(v) + \overline{\text{sizeof}(\mathcal{D})}$:

$$k = \lceil \frac{s \cdot |V|}{\sqrt{0.8 \cdot M}} \rceil$$

Block Formation

For each subgraph, agglomerative hierarchical clustering [70] is used to form blocks and label them for the ranking process. Each vertex starts in an own partition. In every step the two closest partitions are merged. The distance function here is the minimum of the previously defined weighted Jaccard distance of all nodes in the partition:

$$J_P(P_i, P_j) = \underset{u \in P_i, v \in P_j}{\arg\min}\ J_w(\mathcal{D}_u, \mathcal{D}_v)$$

Each partition maintains a label, that is used subsequently. In the beginning the node id is used as label. When a potential merge would cause the so formed partition to exceed the block size, without one of the child partitions being already marked as block, the partition is marked as a block. Additionally the label is adjusted by appending a dot and a counter. If only one of the child partitions formed a block, the label of that partition is used. Finally when both children have formed a block before, their label is merged and a double colon is inserted in the middle. The algorithm terminates when all partitions are assigned to a block. To keep track of the uncaptured nodes in a partition where one child formed a block before an additional field is neccessary.

Layout

Finally, the tree is traversed to extract the so formed blocks and these blocks are sorted according to the label. Finally each of the subgraphs is treated as vertex and the distance between them is the inverse of the number of edges between the subgraphs. Those with the lowest distance get merged and the whole graph is layed out to disk.

5.3 Bondhu

Bondhu [41] is a data layout technique for online social network data. The authors define the cost of a placement similar to the ranking locality above, with r defined as before:

$$\text{cost} = \sum_{(u,v) \in E} |r(v) - r(u)|$$

Without citing G-Store, Hoque and Gupta propose to use the multilevel partitioning algorithm implemented in METIS [46] to partition the graph. Additionally the authors propose to use the Louvain method [10] to find communities, but they don't provide results on this method. The louvain method is described in more detail in the next section.

The authors propose an incremental placement schema within a block: Place the node with the highest degree in the middle of the block, then select the node with the heaviest edge connected to the first node and place it next to the first node. After these two placements, a new graph is created, where the two already placed nodes are merged and their relationships are aggregated. The node with the next heaviest edge is selected and placed alternatingly. The previous two steps are repeated until all nodes are placed within blocks. This is done for every community. In the final part of the method, the intercommunity layout is derived. Here each community is a vertex and the edges between those are just the accumulated edges of the underlying graph. After creating the graph the previous alternating placement schema is applied.

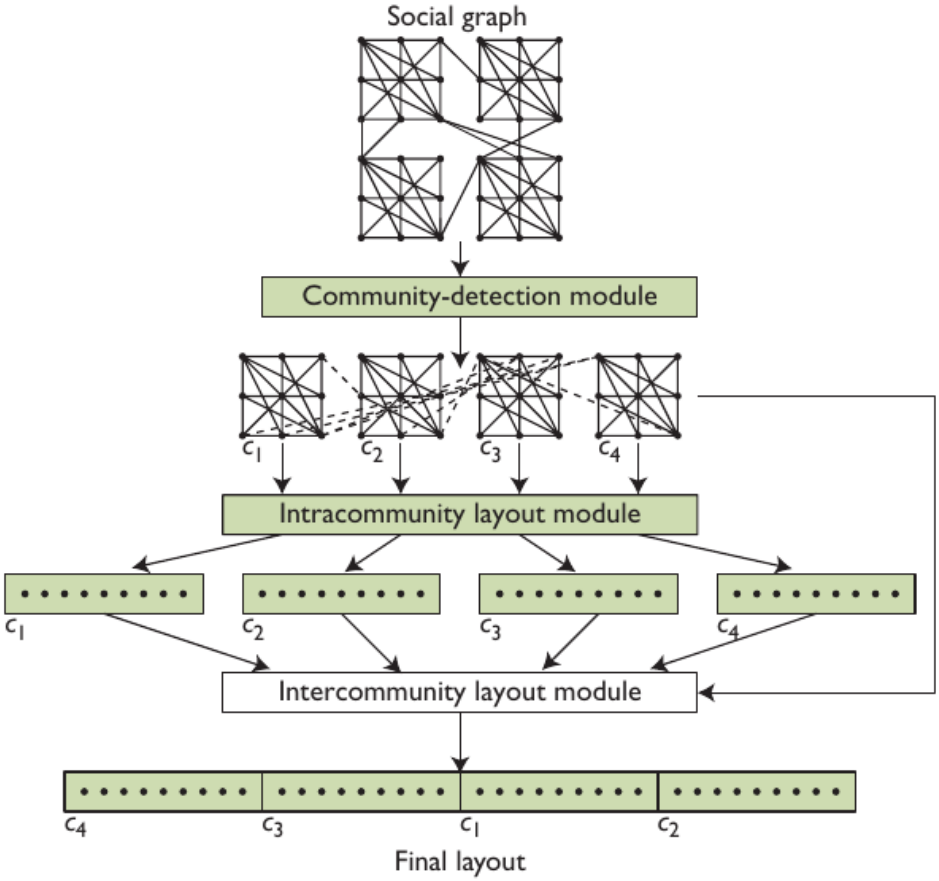


Figure 25 *Broad steps performed by the bondhu algorithm [41].*

6 Method

In relational databases, the records are sorted to achieve locality. Block formation is less of an issue there, as the sorting order of the records yields both, the formation and the order of the blocks. In contrast, graphs need to be partitioned into blocks and if this is done, the sorting order is far from trivial. Both partitioning and linear arrangement are NP-complete problems.

In the following section we try to summarize and abstract the methods and steps that are used to generate an address space layout of the disk that optimizes spatial locality.

6.1 Block Formation & Order

To summarize, previous methods first partitioned the graph by using an adapted version multi-level partitioning algorithm, combining feature extraction with traditional clustering algorithms, the louvain method or the METIS implementation of the multilevel partitioning algorithm. Then based on the partitioning the blocks were formed and ordered. In G-Store this is done in the uncoarsening phase. In ICBL, hierarchical agglomerative clustering in combination with a labelling scheme is used. Bondhu uses a scheme where the vertex with the highest partition is placed in the middle and then iteratively the neighbours with the highest edge weight is then placed next to it and the two nodes are merged in the graph.

G-Store uses adjacency lists as data structure. Thus the edges are placed directly next to the vertices in the very same file. For ICBL, the very same is true: They represent the graph unsing an adjacency list. In the evaluation part of Yaçar's and Gedik's work, the authors apply their order to Neo4J's incidence list structure. As already discussed in 3.2, the incidence list is implemented using an edge list with the incidence list included in the edge's record structure. To adapt ICBL's adjacency list to Neo4J, they insert the relations in the order of the adjacency list and store the nodes in order of their appearance in the edge list. Regarding Bondhu, the relationship arrangement is not mentioned in the paper.

6.2 Block Order

6.3 Incidence List Rearrangement

7 Experimental Evaluation

7.1 Experimental Setup

7.1.1 Implementation

7.1.2 Environment

7.1.3 Data Sets and Queries

7.2 Results

8 Conclusion

8.1 Discussion

Remention contributions

8.2 Future Work

Dynamic rearrangement (Partition order)

8.3 Summary

Bibliography

[1] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. "Compilers, principles, techniques". In: *Addison wesley* 7.8 (1986), p. 9.

[3] *Amazon Neptune*. Feb. 23, 2021. URL: `https://aws.amazon.com/de/neptune/` (visited on 03/26/2021).

[4] Konstantin Andreev and Harald Racke. "Balanced graph partitioning". In: *Theory of Computing Systems* 39.6 (2006), pp. 929–939.

[5] Renzo Angles. "The Property Graph Database Model." In: *AMW*. 2018.

[6] *ArangoDB, the multi-model database for graph and beyond*. Mar. 26, 2021. URL: `https://www.arangodb.com/` (visited on 03/26/2021).

[7] Louis Bachelier. "Théorie de la spéculation". In: *Annales scientifiques de l'École normale supérieure*. Vol. 17. 1900, pp. 21–86.

[8] J-L Baier and Gary R. Sager. "Dynamic improvement of locality in virtual memory systems". In: *IEEE Transactions on Software Engineering* 1 (1976), pp. 54–62.

[9] Pavel Berkhin. "A survey of clustering data mining techniques". In: *Grouping multidimensional data*. Springer, 2006, pp. 25–71.

[10] Vincent D Blondel et al. "Fast unfolding of communities in large networks". In: *Journal of statistical mechanics: theory and experiment* 2008.10 (2008), P10008.

[11] Andries E Brouwer and Willem H Haemers. *Spectra of graphs*. Springer Science & Business Media, 2011.

[12] Robert Brown. "XXVII. A brief account of microscopical observations made in the months of June, July and August 1827, on the particles contained in the pollen of plants; and on the general existence of active molecules in organic and inorganic bodies". In: *The philosophical magazine* 4.21 (1828), pp. 161–173.

[13] Tianqi Chen et al. "{TVM}: An automated end-to-end optimizing compiler for deep learning". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 578–594.

[14] Patricia Conde-Céspedes, Jean-François Marcotorchino, and Emmanuel Viennet. "Comparison of linear modularization criteria using the relational formalism, an approach to easily identify resolution limit". In: *Advances in Knowledge Discovery and Management*. Springer, 2017, pp. 101–120.

[15] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. "A stateless, content-directed data prefetching mechanism". In: *ACM SIGPLAN Notices* 37.10 (2002), pp. 279–290.

[16] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[17] Peter J Denning. "The locality principle". In: *Communication Networks And Computer Systems: A Tribute to Professor Erol Gelenbe*. World Scientific, 2006, pp. 43–67.

[18] Edsger W Dijkstra et al. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[19] W.E. Donath and A.J. Hoffman. “Algorithms for partitioning graphs and computer logic based on eigenvectors of connection matrices”. In: *IBM Technical Disclosure Bulletin* 15.3 (1972), pp. 938–944.

[20] *Eigen: The Matrix class*. Dec. 5, 2020. URL: `https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html` (visited on 03/11/2021).

[21] S. C. Eisenstat et al. “Yale sparse matrix package I: The symmetric codes”. In: *International Journal for Numerical Methods in Engineering* 18 (1982), pp. 1145–1151.

[22] Charles M Fiduccia and Robert M Mattheyses. “A linear-time heuristic for improving network partitions”. In: *19th design automation conference*. IEEE. 1982, pp. 175–181.

[23] *File NodeRecordFormat.java - Neo4J - GitHub*. Mar. 18, 2021. URL: `https://github.com/neo4j/neo4j/blob/4cd5556a5356c8d0d9efe9c8fb6a8b87865c48ed/community/record-storage-engine/src/main/java/org/neo4j/kernel/impl/store/format/standard/NodeRecordFormat.java` (visited on 03/18/2021).

[24] Francois Fouss et al. “Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation”. In: *IEEE Transactions on knowledge and data engineering* 19.3 (2007), pp. 355–369.

[25] Michael R Garey, David S Johnson, and Larry Stockmeyer. “Some simplified NP-complete problems”. In: *Proceedings of the sixth annual ACM symposium on Theory of computing*. 1974, pp. 47–63.

[26] Michelle Girvan and Mark EJ Newman. “Community structure in social and biological networks”. In: *Proceedings of the national academy of sciences* 99.12 (2002), pp. 7821–7826.

[27] *GitHub - neo4j/neo4j: Graphs for Everyone*. Dec. 9, 2020. URL: `https://github.com/neo4j/neo4j` (visited on 12/09/2020).

[28] A. Goldberg and Renato F. Werneck. “Computing Point-to-Point Shortest Paths from External Memory”. In: *ALENEX/ANALCO*. 2005.

[29] Andrew V Goldberg and Chris Harrelson. “Computing the shortest path: A search meets graph theory.” In: *SODA*. Vol. 5. Citeseer. 2005, pp. 156–165.

[30] M. Goodrich and R. Tamassia. “Algorithm Design and Applications”. In: 2014.

[31] Jim Griffioen and Randy Appleton. “Reducing File System Latency using a Predictive Approach.” In: *USENIX summer*. 1994, pp. 197–207.

[32] J. Gross and Jay Yellen. “Graph Theory and Its Applications”. In: 1998.

[33] Saurabh Gupta et al. “Locality principle revisited: A probability-based quantitative approach”. In: *Journal of Parallel and Distributed Computing* 73.7 (2013), pp. 1011–1027.

[34] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[35] Peter E Hart, Nils J Nilsson, and Bertram Raphael. “A formal basis for the heuristic determination of minimum cost paths”. In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[36] Keith Henderson et al. “It’s who you know: graph mining using recursive structural features”. In: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2011, pp. 663–671.

[37] Keith Henderson et al. “Rolx: structural role extraction & mining in large graphs”. In: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2012, pp. 1231–1239.

[38] Bruce Hendrickson and Robert W Leland. “A Multi-Level Algorithm For Partitioning Graphs.” In: *SC* 95.28 (1995), pp. 1–14.

[39] Petter Holme. “Rare and everywhere: Perspectives on scale-free networks”. In: *Nature communications* 10.1 (2019), pp. 1–3.

[40] Jürgen Hölsch and Michael Grossniklaus. “An Algebra and Equivalences to Transform Graph Patterns in Neo4j”. In: *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference (EDBT/ICDT 2016)*. Ed. by Themis Palpanas and Kostas Stefanidis. CEUR Workshop Proceedings 1558. 2016. URL: http://ceur-ws.org/Vol-1558/paper24.pdf.

[41] Imranul Hoque and Indranil Gupta. “Disk layout techniques for online social network data”. In: *IEEE Internet Computing* 16.3 (2012), pp. 24–36.

[42] Paul Jaccard. “The distribution of the flora in the alpine zone. 1”. In: *New phytologist* 11.2 (1912), pp. 37–50.

[43] Bruce Jacob, David Wang, and Spencer Ng. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[44] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. “Data clustering: a review”. In: *ACM computing surveys (CSUR)* 31.3 (1999), pp. 264–323.

[45] Doug Joseph and Dirk Grunwald. “Prefetching using markov predictors”. In: *IEEE transactions on computers* 48.2 (1999), pp. 121–133.

[46] George Karypis and Vipin Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs”. In: *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 359–392. DOI: 10.1137/S1064827595287997. eprint: https://doi.org/10.1137/S1064827595287997. URL: https://doi.org/10.1137/S1064827595287997.

[47] Brian W Kernighan and Shen Lin. “An efficient heuristic procedure for partitioning graphs”. In: *The Bell system technical journal* 49.2 (1970), pp. 291–307.

[48] Jon Kleinberg. “The small-world phenomenon: An algorithmic perspective”. In: *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. 2000, pp. 163–170.

[49] D. Knuth. “Dancing links”. In: 2000.

[50] Tom M Kroeger, Darrell DE Long, Jeffrey C Mogul, et al. “Exploring the Bounds of Web Latency Reduction from Caching and Prefetching.” In: *USENIX Symposium on Internet Technologies and Systems*. 1997, pp. 13–22.

[51] Harry R Lewis. *Computers and intractability. A guide to the theory of NP-completeness*. 1983.

[52] Stuart Lloyd. “Least squares quantization in PCM”. In: *IEEE transactions on information theory* 28.2 (1982), pp. 129–137.

[53] László Lovász et al. “Random walks on graphs: A survey”. In: *Combinatorics, Paul erdos is eighty* 2.1 (1993), pp. 1–46.

[54] Édouard Lucas. *Récréations mathématiques: Les traversees. Les ponts. Les labyrinthes. Les reines. Le solitaire. la numération. Le baguenaudier. Le taquin*. Vol. 1. Gauthier-Villars et fils, 1891.

[55] *Matrix Storage Schemes*. Oct. 1, 1999. URL: http://www.netlib.org/lapack/lug/node121.html (visited on 03/11/2021).

[56] *Matrix Storage Schemes*. Mar. 5, 2021. URL: https://software.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/blas-and-sparse-blas-routines/blas-routines/matrix-storage-schemes-for-blas-routines.html (visited on 03/11/2021).

[57] Thomas Neumann and Guido Moerkotte. “Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins”. In: *2011 IEEE 27th International Conference on Data Engineering*. IEEE. 2011, pp. 984–994.

[58] Andrew Y Ng, Michael I Jordan, Yair Weiss, et al. “On spectral clustering: Analysis and an algorithm”. In: *Advances in neural information processing systems* 2 (2002), pp. 849–856.

[59] *Package id-generator - Neo4J - GitHub*. Mar. 18, 2021. URL: `https://github.com/neo4j/neo4j/tree/4cd5556a5356c8d0d9efe9c8fb6a8b87865c48ed/community/id-generator` (visited on 03/18/2021).

[60] *Package io - Neo4J - GitHub*. Mar. 18, 2021. URL: `https://github.com/neo4j/neo4j/tree/4cd5556a5356c8d0d9efe9c8fb6a8b87865c48ed/community/io` (visited on 03/18/2021).

[61] *Package record-storage-engine - Neo4J - GitHub*. Mar. 18, 2021. URL: `https://github.com/neo4j/neo4j/tree/4cd5556a5356c8d0d9efe9c8fb6a8b87865c48ed/community/record-storage-engine` (visited on 03/18/2021).

[62] Karl Pearson. “The problem of the random walk”. In: *Nature* 72.1867 (1905), pp. 342–342.

[63] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2000.

[64] *RedisGraph - a graph database module for Redis*. Mar. 19, 2021. URL: `https://oss.redislabs.com/redisgraph/` (visited on 03/26/2021).

[65] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases: new opportunities for connected data*. "O’Reilly Media, Inc.", 2015.

[66] Marko A Rodriguez and Peter Neubauer. “The graph traversal pattern”. In: *Graph Data Management: Techniques and Applications*. IGI Global, 2012, pp. 29–46.

[67] Michael A. Rodriguez and P. Neubauer. “Constructions from Dots and Lines”. In: *ArXiv* abs/1006.2361 (2010).

[68] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating system concepts*. John Wiley & Sons, 2006.

[69] Abraham Silberschatz, Henry F Korth, Shashank Sudarshan, et al. *Database system concepts*. Vol. 4. Mcgraw-hill New York, 1997.

[70] P. H. A. Sneath. “The Application of Computers to Taxonomy”. In: *Microbiology* 17.1 (1957), pp. 201–226. ISSN: 1350-0872. DOI: `https://doi.org/10.1099/00221287-17-1-201`. URL: `https://www.microbiologyresearch.org/content/journal/micro/10.1099/00221287-17-1-201`.

[71] William Stallings. *Operating systems: internals and design principles*. Boston: Prentice Hall, 2012.

[72] Angelika Steger. *Diskrete Strukturen: Band 1: Kombinatorik, Graphentheorie, Algebra*. Springer-Verlag, 2007.

[73] Robin Steinhaus, Dan Olteanu, and Tim Furche. “G-Store: a storage manager for graph data”. PhD thesis. University of Oxford, 2010.

[74] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[75] Vincent A Traag, Ludo Waltman, and Nees Jan Van Eck. “From Louvain to Leiden: guaranteeing well-connected communities”. In: *Scientific reports* 9.1 (2019), pp. 1–12.

[76] Ulrike Von Luxburg. “A tutorial on spectral clustering”. In: *Statistics and computing* 17.4 (2007), pp. 395–416.

[77] Louis Weinberg. “Kirchhoff’s’ third and fourth laws’”. In: *IRE Transactions on Circuit Theory* 5.1 (1958), pp. 8–30.

[78] Norbert Wiener. *Collected Works, Vol. 1*. 1976.

[79] Rui Xu and Donald C Wunsch. “Survey of clustering algorithms”. In: (2005).

[80] Abdurrahman Yaar. “Scalable layout of large graphs on disk”. PhD thesis. Bilkent University, 2015.

[81] Abdurrahman Yaar, Bura Gedik, and Hakan Ferhatosmanolu. “Distributed block formation and layout for disk-based management of large-scale graphs”. In: *Distributed and Parallel Databases* 35.1 (2017), pp. 23–53.

[82] Konrad Zuse. “Über den allgemeinen Plankalkül als Mittel zur Formulierung schematisch-kombinativer Aufgaben”. In: *Archiv der Mathematik* 1.6 (1948), pp. 441–449.