[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

2 Preliminaries

2.1 Graph Databases

In this section we first give a definition of graphs as discrete structures and related concepts. Next we introduce and analyze possible data structures and algorithms to represent and operate on graphs.

2.1.1 Graph Theory

Most of the definitions below follow the notations intorduced in [29, 18, 1, 10, 17]

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 neighbouhood 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 neighbouhood 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 $M : E \to \mathbb{N}$. A graph that is not a multi-graph and does not have self-loops is called **simple**.

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 $i \in \{0, \ldots, n-1\} : G_i = (V_i, E_i)$ of G, where

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

2. $\bigcup_i V_i = V$.

2.1.2 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 [18, 1, 10, 17, 30]. 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 [30, 13]

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 and a pass over the list of vertex indices to delete the vertex itself, in total $\mathcal{O}(|E| + |V|)$.
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 neighbouhood 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 **??**

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 $e = (e, v) \in E$ 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

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

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

Figure 1 *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.*

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-deterministic, 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 neighbouhood 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 **??**

Incidence Matrix

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, 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.

```
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 2 *An example of the adjacency matrix representation of a graph.*

The space complexity here is $\mathcal{O}(|V| + |E|)$, as we store each node once and then for each relationship one more node in the corresponding adjacency list containing $N_v|_{\mathrm{out}}$.

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 $\mathcal{O}(2\deg(V))$ 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 neighbouhood 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. $N_v = N_v|_{\mathrm{out}} = N_v|_{\mathrm{in}}$. This means for undirected graphs all neighbourhood queries are in $\mathcal{O}(1)$.

An example of this data structure is shown in **??**

Incidence List

Summary

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

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

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

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

---

[1]For directed graphs, for undirected ones it's $\mathcal{O}(1)$

2.1.3 Queries and Algorithms

2.1.3.1 Traversal-based

Breadth First Search & Dijkstra, Spreading Activation

A* Algorithm

Depth First Search

Random Walk

#### 2.1.3.2 Partitioning, Clustering or Community Detection

2.1.4 Architecture

Memory hierarchy Relational databases store tables of data. The links considered in this category of DBMS are mostly used to stitch together the fields of an entry 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 edge lists such that the look up of outgoing or incomming edges is only $\mathcal{O}(\log(n))$ which would speed up breadth first traversals. 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 data base management systems, native graph databases use structures specialised for this kind of queries. In the following sections we discuss what structures and mechanisms graph databases employ in order to achieve this superior performance in the domain of graphs using the popular graph database Neo4J as example.

First of all, let us consider the high level architecture of a general database management systems as shown in figure **??** — with a focus on the storage and loading elements.
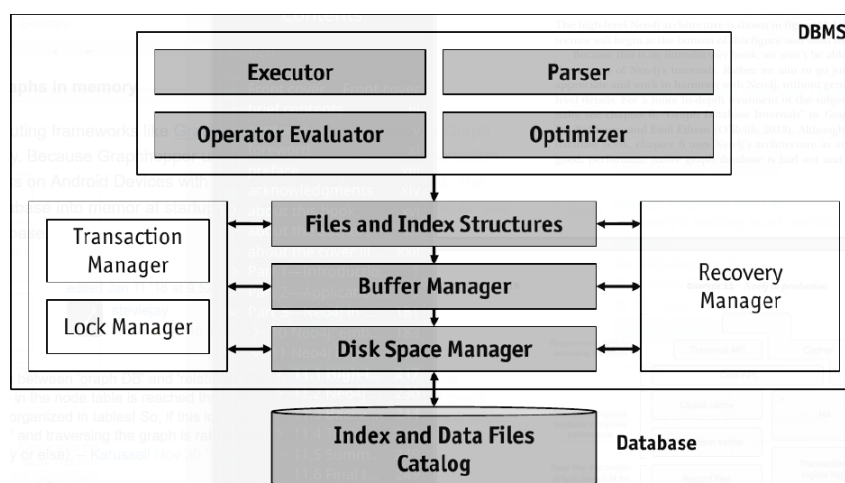

Figure 4 *The typical structure of a relational database management system.*

Here The disk space manager, sometimes also called storage manager, handles de-/allocations, reads & writes and provides the concept of a page: A disk block brought into memory. For that

it needs to keep track of free blocks in the allocated file. Optimally both a disk block and a page are of the same size. One crucial task of a disk space manager is to store sequences of pages into continuous memory blocks in order to optimize data locality. Data locality has the upside, that one needs only one I/O operation to load multiple pages. To summarize the two most important objectives of a storage manager are to provide a locality-preserving mapping from pages to blocks based upon the information in the DBMS and to abstract physical storage to pages, taking care of allocation and access.

A buffer manager is used to mediate between external storage and main memory. It maintains a designated pre-allocated area of main memory — called the buffer pool — to load, cache and evict pages into or from main memory. 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.
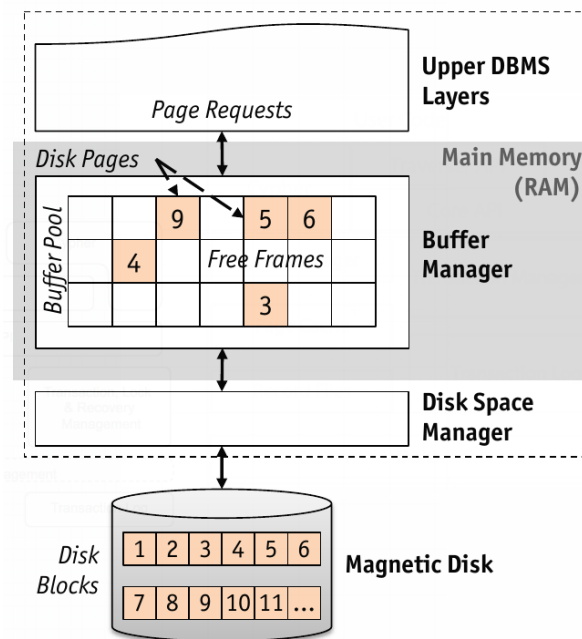
Figure 5 *A visualization of the interaction of a database with memory.*

The final memory and storage model relevant component of the of a database management system is the file layout and possible index structures. In order to store data a DBMS may either store one single or multiple files to maintain records.

A file may consist of a set of pages containing a set of slots. A slot stores one record with each record containing a set of fields. Further the database needs to keep track of free space in the file: A linked list or a directory must record free pages and some structure needs to keep track of the free slots either globally or per page. Records may be of fixed or of variable size, depending on the types of their 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. 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 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, the concept of indexes have been introduced. Indexes are auxiliary structures to speed up certain operations 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. An unclustered index over field G is one where the file is not sorted according to G. In a similar way indexes can be sparse or dense. A sparse index has less index entries than records, mostly one index entry per file. 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. There are different variants of storing index entries which have again certain implications on the compactness of the index and the underlying design decisions.

In another view of database management systems architectures, this boils down to the design decisions one has to make when implementing the storage layer and the access layer as shown in figure **??**.
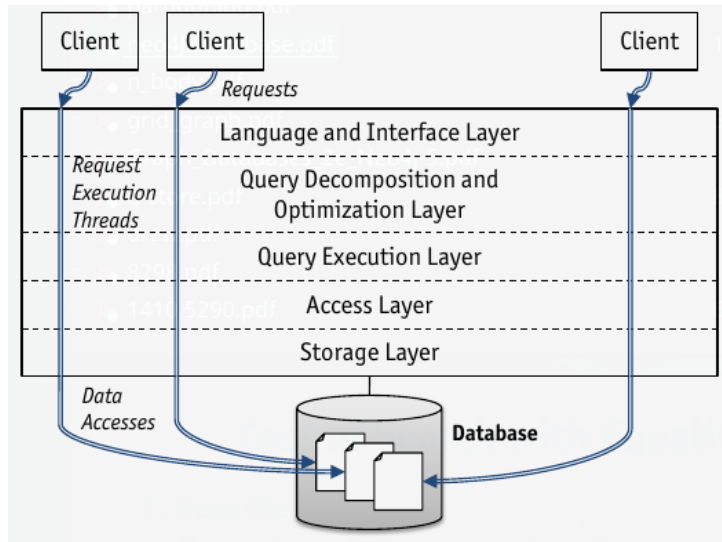
Figure 6 *The architecture of a database management system from another point of view.*

Here the storage layer is in close correspondence to the disk space manager in combination with the buffer manager, while the files and index structures provide the access layer.

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.

2.1.5 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 carrying additional information.

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 function assigning a pair of nodes to an edge.

- L a set of strings used as labels.

- P a set of key-value pairs called properties.

- T a set of strings used as relationship types.

- $f_P : V \cup E \to 2^P$ a function that assigns a set of properties to a node or relationship.

- $f_T : E \to T$ a function that assigns a type to a relationship.

- $f_L : V \to 2^L$ a function 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 key-value pairs [3].

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. This means they either need to be addressed explicitly or one needs to store all information, including the properties, consecutively. The latter approach has the downside, that when the type or a property contain a variable length string, the relationships have to be variable lenght records. This would cause the lookup of a relationship's property to be not only indirected via a node index, but also requires an additional mechanism to be able to tell the beginning and length of the record.

An illustration of this model is shown in Figure 7 . Neo4j is a graph database employing the
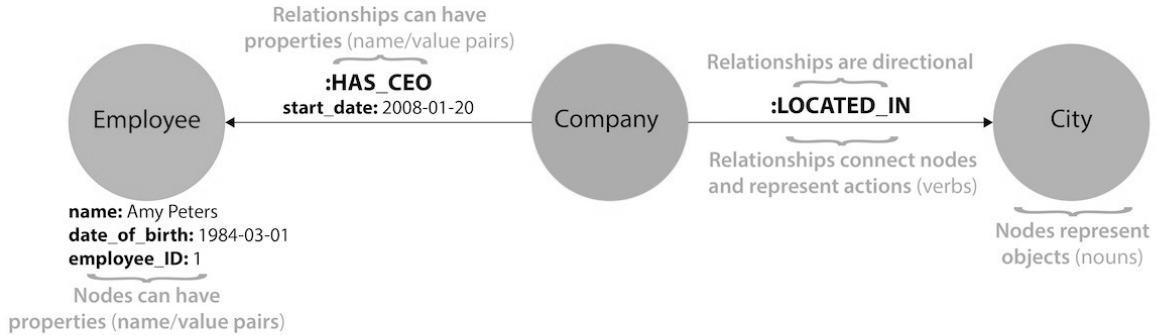
Figure 7 *A visualization of the property graph model*

property graph model [**neo4j__book**], which is used in the evaluation part of this thesis.

2.1.6 Example: Neo4J

When restricting to graph structures where nodes and relationships are allowed to have properties and labels and types respectively, this allows one to narrow down some of the design decisions. In particular the example of a popular graph native database — Neo4J — is what we discuss in this subsection.

To get an overview of the architecture let us consider figure **??**.

Here we can see that the previous schema is not exactly straight forward to apply, mainly due to a lack of concise documentation. The diagram was taken from the only publication that elaborates on the internals of Neo4J aside from the code of course.
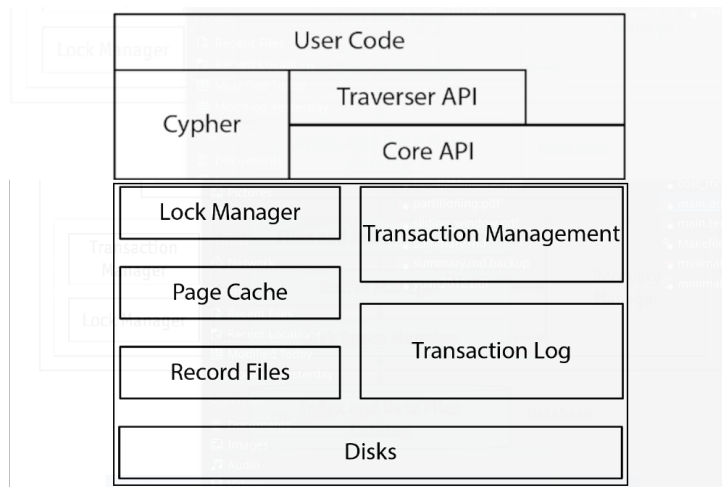
Figure 8 *The high level architecture of Neo4J according to Emil Efrim, the co-founder of Neo technologies.*
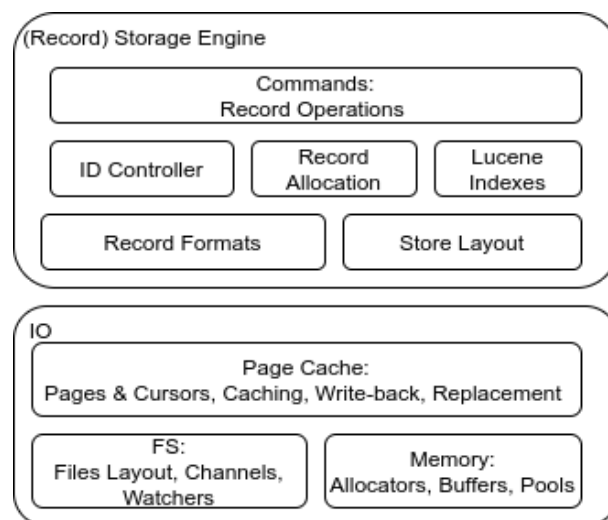
Figure 9 *A visualization of the broad the storage and memory organization of Neo4J.*

Here The storage manager makes mostly use of the Java NIO package with some additional usage of operating system native calls to allocate memory for the page cache and network buffers. A more detailed view on the high level architecture of the disk space and buffer manager and the files and index structures was deduced by the author from the source code and the non-public JavaDocs. This is shown in figure **??**.
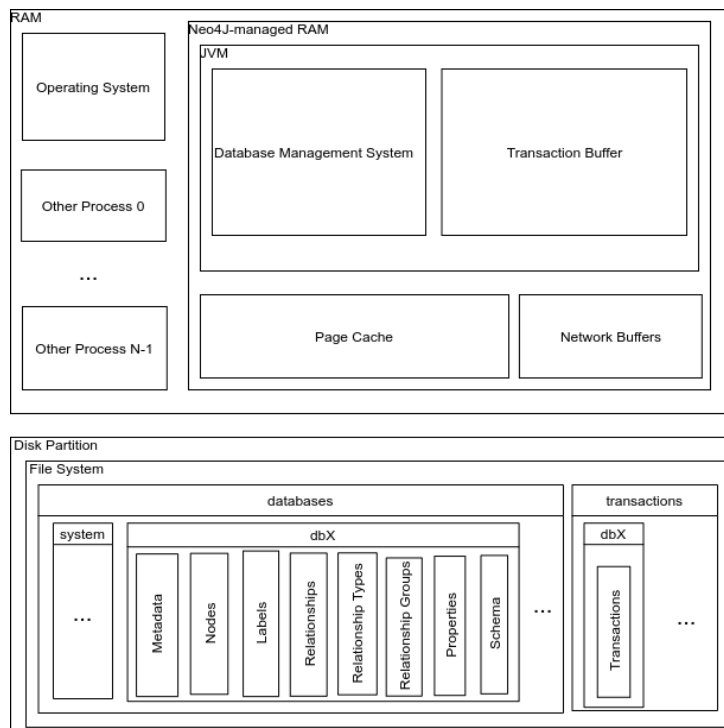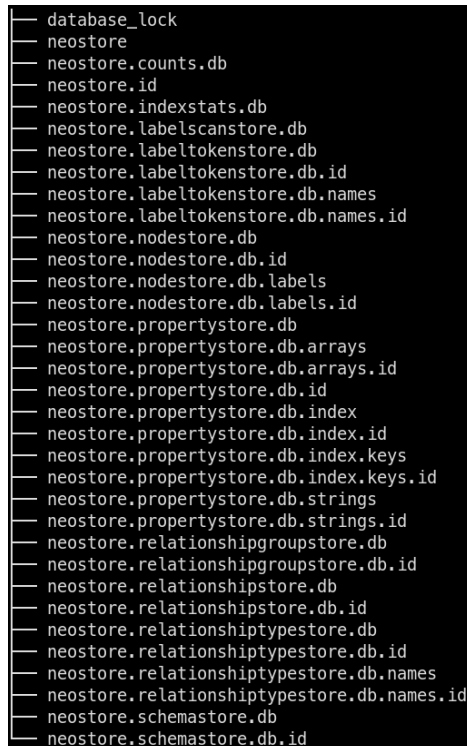

Figure 10 *A sketch of how Neo4J occupies memory.*

The overall memory and storage state of a Neo4J instance and its environment may thus be visualized like this figure **??**.

2.1.6.1 IO & File Layout

The files that Neo4J uses to store data are categorizable into 5 classes based upon the records they contain:

- Node-related files: These files start with the prefixes `neostore.nodestore*` and `neostore.label*`. The contents of these files contain the node structure and labels.

- Relationship-related files: The prefix `neostore.relationship*` is used for all files containing records related to the graph's relationship structure, types and possibly groups.

- Property-related files: Containing the properties of both nodes and relationship, this is the part that is least structured and rather unspecialized with respect to graphs.

- Schema: Files starting with `neostore.schemastore*` contain information about the schema of the graph or rather the schematic information on the node, relationship and property labels, types and constraints.

- Metadata: Files only starting with `neostore` and none of the above prefixes contain meta information about the graph.

- Finally, a lock file.

Figure 11 *A visualization of the files layout of Neo4j.*

Most importantly the files ending with `.db` contain all fixed size records representing the structure of the graph and small amounts of data. Files ending with `.id` are used for record allocation and contain unused IDs. Files having one of the suffixes `names`, `arrays` and `strings` are so called dynamic stores and contain dynamic sized records. A dynamic store is always started with a header block containing extra information and the offset to the first block in the store files. Files ending with `.index` contain indexes either generated by Neo4J (e.g. the keys of properties are referenced like this to avoid storing key names multiple times) or by the Apache Lucene indexing library.

2.1.6.2 Record Structures

Address Translation Neo4j defines the following constants as the size of the addresses used. These are represented by the datatype **long** when loaded into main memory. On disk these

```java
public static final int PROPERTY_TOKEN_MAXIMUM_ID_BITS = 24;
       static final int NODE_MAXIMUM_ID_BITS = 35;
       static final int RELATIONSHIP_MAXIMUM_ID_BITS = 35;
       static final int PROPERTY_MAXIMUM_ID_BITS = 36;
       public static final int DYNAMIC_MAXIMUM_ID_BITS = 36;
       public static final int LABEL_TOKEN_MAXIMUM_ID_BITS = 32;
       public static final int RELATIONSHIP_TYPE_TOKEN_MAXIMUM_ID_BITS = 16;
       static final int RELATIONSHIP_GROUP_MAXIMUM_ID_BITS = 35;
       public static final int SCHEMA_RECORD_ID_BITS = 32;
```

values are stored as 32-bit `Integer`s with additional modifiers, that is the highest bits that do not fit into an int are stored to fields with additional space.

For example the in-use bit consumes one byte of memory and the additional 7 bytes are used to stroe the highest bits that do not fit into an integer of another field.

Finally the base and the modifer are aggregated into a long by shifting both appropriately, casting them to longs and applying the logical or | operation.

Thus when refering to high bits and ID below, high bits do always mean the remainer of the bits that did not fit into the field that is said to be holding the ID. The latter is actually only holding the lowest 32 bits of an ID/address. The *actual ID* is always high bits and ID field, shifted appropriately and connected using the logical or operator.

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 · 100 = 1500 from the beginning of the file. The struct of a record looks like this:

1. Byte 1: The first byte contains one bit for the in-use flag. The additional 7 bits are used to store the 3 highest bits of the relationship ID and the 4 highest bits of a property ID.

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

3. Bytes 6 — 9: Again 4 bytes encode the ID to the first property 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. one node has an aweful lot of relationships and is treated a bit differently. That is a relationships are stored by type and direction for this node into groups, see **??**.
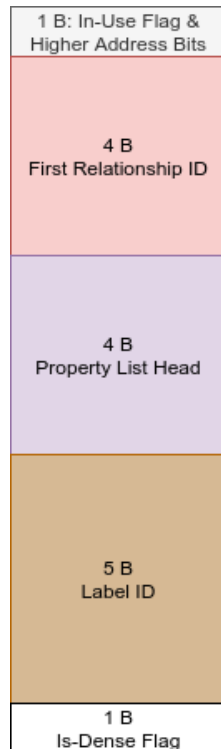
Figure 12 *A visualization of the record structure of a node.*

To summarize: The records on disk are stored as in the enumeration above. 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.
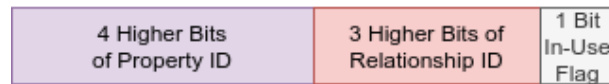
Figure 13 *A visualization of the information stored in the first byte of a node record.*

Node Labels Each node label record consists of an in use bit and a name ID, which in turn points to an entry in a seperate file storing label names as dynamic records (see **??**), storing the label names. This is done in order to assure that the records are of fixed length.

1. Byte 1: In-use flag

2. Bytes 2 — 5: Pointer to the label string entry
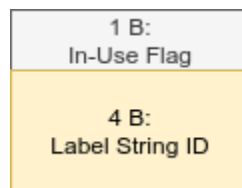
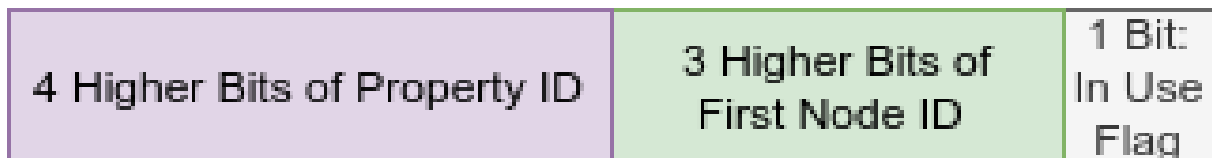**Figure 14** *A visualization of the structure of a label record.*

Relationships Relationship records are stored with implicit IDs too. Their fixed size records contain 34 bytes. Besides an in-use flag and the node IDs that are connected, and the relationship type, the record also contains two doubly linked list: One for the relationships of the first node and one for the relationship of the second node. Finally a link to the head of the properties linked list of this relationship and a marker if this relationship is the first element in the relationships linked list of one of the nodes.
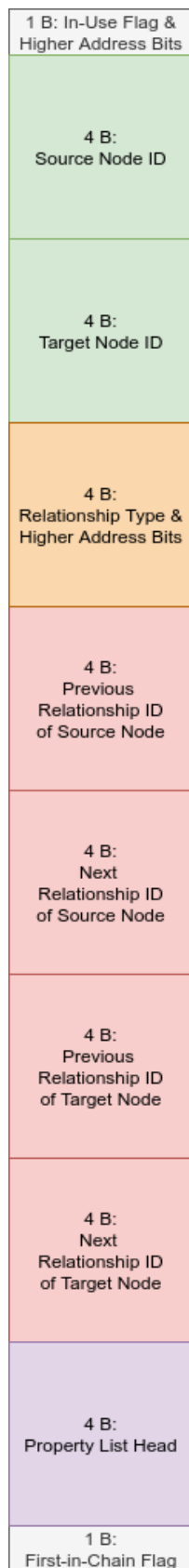
Figure 15 *A visualization of the record structure of a relationship in Neo4J.*

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 16 *A visualization of how information is stored in the first byte of a relationship record.*
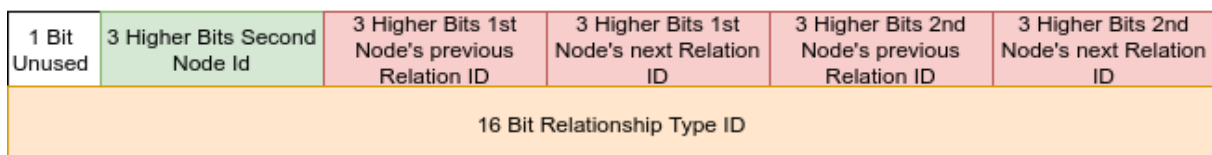
Figure 17 *The structure of the bytes that are used to store the type of a relationship and high bits of a node and a relationship IDs.*

Relationship Types   Similarly to the node labels, the relationship type records posses an in-use flag and a type ID that points to a record in a file containing strings in the dynamic record format.

1. Byte 1: In-use flag

2. Bytes 2 — 5: Pointer to the type string entry

All further record structures like properties, arrays and strings are omitted for the sake of succintness.
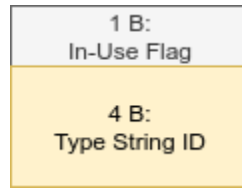
Figure 18 *A visualization of the record structure of a relationship type.*

2.1.6.3 Caching & Indexes

2.1.6.4 Queries & Transactions

2.1.7 Locality

2.2 Problem Definition

2.2.1 Metrics

3 Methods

3.1 G-Store: Multilevel Partitioning

3.1.1 Coarsening

3.1.2 Turnaround

3.1.3 Uncoarsening

3.1.3.1 Projection

3.1.3.2 Reordering

3.1.3.3 Refinement

3.2 ICBL: Diffusion Set-based Clustering

3.2.1 Identify Diffusion Sets

3.2.2 Coarse Clustering

3.2.3 Block Formation

3.2.4 Layout

3.3 Our Approach: Community Detection

3.3.1 Louvain method/Leiden

3.3.2 Kernighan-Lin Algorithm/N-Body Simulation/X?

3.3.3 Adjacency List Rearrangement

4 Experimental Evaluation

4.1 Experimental Setup

4.1.1 Implementation

4.1.2 Environment

4.1.3 Data Sets and Queries

4.2 Results

4.2.1 Modularity

4.2.2 Conductance and Cohesiveness

4.2.3 Tension

4.2.4 Block-based IOs

5 Conclustion

5.1 Discussion

Remention contributions

5.2 Future Work

5.3 Summary

Bibliography

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

[2] *An overview of Neo4j Internals.* Dec. 9, 2020. URL: https://www.slideshare.net/thobe/an-overview-of-neo4j-internals (visited on 12/09/2020).

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

[4] L. Belady. "A Study of Replacement Algorithms for Virtual-Storage Computer". In: *IBM Syst. J.* 5 (1966), pp. 78–101.

[5] Michael R Berthold et al. "Pure spreading activation is pointless". In: *Proceedings of the 18th ACM conference on Information and knowledge management.* 2009, pp. 1915–1918.

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

[7] Marek Ciglan and Kjetil Nørvåg. "SGDB–Simple graph database optimized for activation spreading computation". In: *International Conference on Database Systems for Advanced Applications.* Springer. 2010, pp. 45–56.

[8] Allan M Collins and Elizabeth F Loftus. "A spreading-activation theory of semantic processing." In: *Psychological review* 82.6 (1975), p. 407.

[9] 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.

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

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

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

[13] 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.

[14] Bura Gedik and Rajesh Bordawekar. "Disk-based management of interaction graphs". In: *IEEE Transactions on Knowledge and Data Engineering* 26.11 (2014), pp. 2689–2702.

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

[16] Andrew V Goldberg and Chris Harrelson. "Computing the shortest path: A search meets graph theory." In: In SODA (Vol. 5, pp. 156-165).

[17] M. Goodrich and R. Tamassia. "Algorithm Design and Applications". In: 2014.

[18] J. Gross and Jay Yellen. "Graph Theory and Its Applications". In: 1998.

[19] 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.

[20] Jürgen Hölsch and Michael Grossniklaus. "An algebra and equivalences to transform graph patterns in neo4j". In: *EDBT/ICDT 2016 Workshops: EDBT Workshop on Querying Graph Structured Data (GraphQ)*. 2016.

[21] Sven Kosub. "A note on the triangle inequality for the Jaccard distance". In: *CoRR* abs/1612.02696 (2016). arXiv: `1612.02696`. URL: `http://arxiv.org/abs/1612.02696`.

[22] *Neo4j architecture | Key-value Stories*. Feb. 29, 2020. URL: `http://key-value-stories.blogspot.com/2015/02/neo4j-architecture.html` (visited on 12/09/2020).

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

[24] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw-Hill, 2000.

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

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

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

[28] Robert Soulé and Bura Gedik. "RailwayDB: adaptive storage of interaction graphs". In: *The VLDB Journal* 25.2 (2016), pp. 151–169.

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

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

[31] *The Neo4j Operations Manual v4.2 - Operations Manual*. Dec. 4, 2020. URL: `https://neo4j.com/docs/operations-manual/current/` (visited on 12/09/2020).

[32] 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.

[33] Abdurrahman Yaar. "Scalable layout of large graphs on disk". PhD thesis. Bilkent University, 2015.

[34] 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.

[35] Liang Yuan, Chen Ding, Yunquan Zhang, et al. "Modeling the locality in graph traversals". In: *2012 41st International Conference on Parallel Processing*. IEEE. 2012, pp. 138–147.

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