

[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	5
2	Preliminaries	6
2.1	Database Architecture	6
2.2	Graphs	8
2.2.1	Theory	8
2.2.2	Data Structures	11
2.2.3	Traversal-based Algorithms	17
2.3	The Property Graph Model	20
2.4	Example: Neo4J	21
3	Problem Definition	27
3.1	Locality	27
3.2	Problem Definition	27
3.3	Example: Vertex, Edge and Incidence List Order	27
4	Related Work	28
4.1	G-Store: Multilevel Partitioning	28
4.1.1	Coarsening	28
4.1.2	Turnaround	28
4.1.3	Uncoarsening	28
4.2	ICBL: Diffusion Set-based Clustering	28
4.2.1	Identify Diffusion Sets	28
4.2.2	Coarse Clustering	28
4.2.3	Block Formation	28
4.2.4	Layout	28
5	Methods	29
5.1	Louvain method	29
5.2	Partition order	29
5.3	Incidence List Rearrangement	29
6	Experimental Evaluation	30
6.1	Experimental Setup	30
6.1.1	Implementation	30
6.1.2	Environment	30
6.1.3	Data Sets and Queries	30
6.2	Results	30
7	Conclusion	31
7.1	Discussion	31
7.2	Future Work	31
7.3	Summary	31

1 Introduction

Essay-like Intro.

Organisation

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

2 Preliminaries

2.1 Database 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 incoming 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 1 — with a focus on the storage and loading elements.

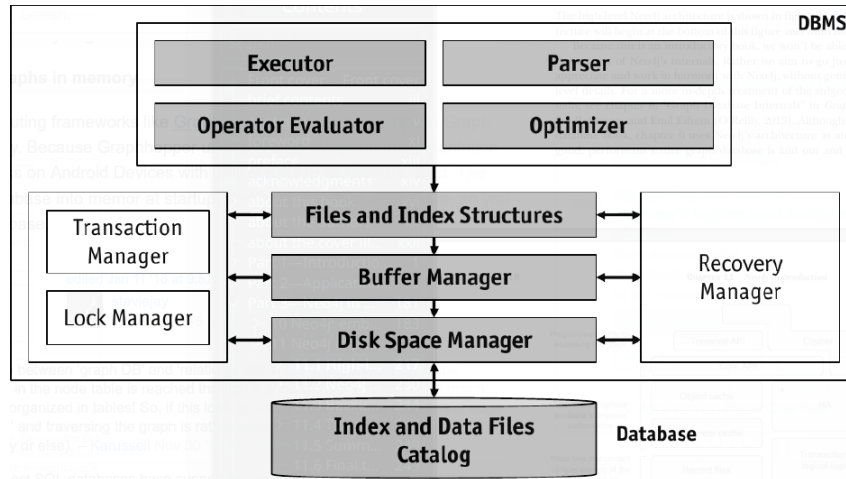


Figure 1 The typical structure of a relational database management system.

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

2.1. DATABASE ARCHITECTURE

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.

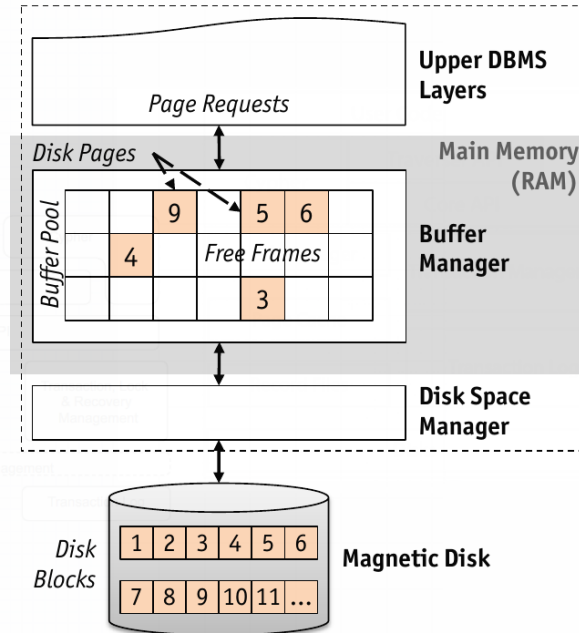


Figure 2 A visualization of the interaction of a database with memory.

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 3.

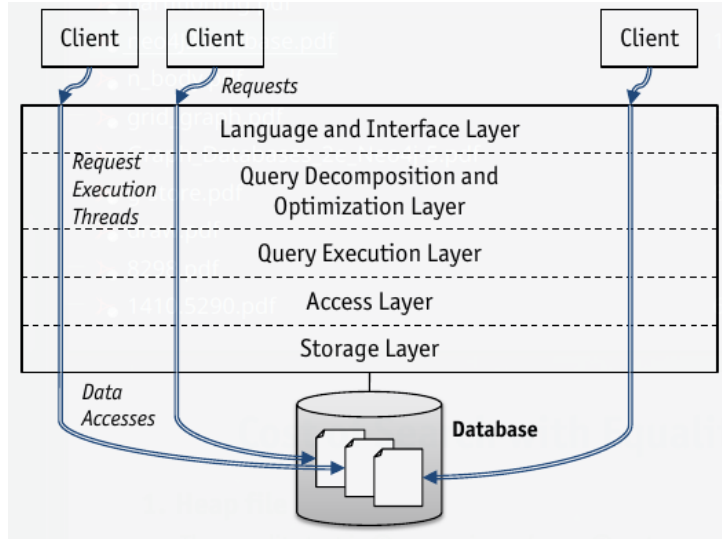


Figure 3 *The architecture of a database management system from another point of view.*

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.

2.2 Graphs

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.

2.2.1 Theory

Most of the definitions below follow the notations introduced in [24, 13, 1, 6, 12]

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 $E' \subseteq E$.

2.2. GRAPHS

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 :

$$N_v = \{u \in V \mid (v, u) \in E \vee (u, v) \in E\}.$$

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:

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

Similarly the outgoing edges are defined as

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

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 of an unweighted and undirected, a directed and a weighted graph is shown in figure 4

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,

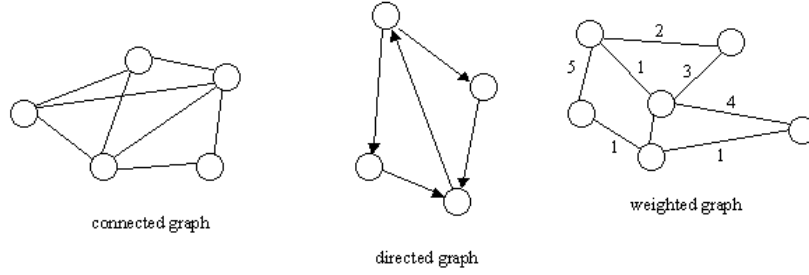


Figure 4 An undirected, a directed and a weighted graph.

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 $M : E \rightarrow \mathbb{N}$. 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 $i \in \{0, \dots, n-1\} : G_i = (V_i, E_i)$ of G , where

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

2.2.2 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 succinctness. 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 [13, 1, 6, 12, 25]. 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 referred to [25, 9]

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 retrieved in $\mathcal{O}(1)$ assuming that the list data structure stores its size as a field. 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 whether 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 5.

0	1	2	3	4	7	9	10		0	1	1
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 5 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 being the value of that entry. Let $A \in |V| \times |V|$, $u, v \in \{0, \dots, |V| - 1\}$ 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 retrieved 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 immediately 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 whether 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 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 6.

```

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 6 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

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

2.2. GRAPHS

As with adjacency matrices, in order to be able to represent non-consecutive indices, 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 retrieved in $\mathcal{O}(1)$, as it's simply the size of the mapping that is stored. The number of edges can also be retrieved 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 whether 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 whether the incoming or outgoing neighbours shall be returned — have to be checked. An example of this data structure is shown in 7.

0	1	2	3	4	5	6	7
0	1	2	3	4	7	9	10
-1	2	0	0	0	0	0	0
1	-2	-1	$-(-1)$	0	5	0	0
0	0	1	0	0	0	0	0
0	0	0	(-1)	1	0	0	0
0	0	0	0	1	-5	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	-3
0	0	0	0	0	0	0	3

Figure 7 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 $N_{v|out}$ is the content of the adjacency list. When we sum $|N_{v|out}|$ 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 $N_{v|out}$.

The number of nodes can be retrieved 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 occurrences 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 whether 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 8.

```

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 8 An example of the adjacency list representation of a graph.

Incidence List

This representation is also called incidence table in [13]. The incidence list of a graph G stores for each vertex $v \in V$ the list of edges it is connected 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 thing that is beneficial, 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 9.

```

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 9 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 unacceptable 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) [20, 8, 19]. 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 [4]. Further it’s useful in physical flow problems & simulations, e.g. when computing the current and resistances in a graph or when simulating micro-circuits [26]. Even though 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. Instead 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 [20, 8, 19].

Finally the adjacency and incidence lists are quite similar in many respects: Both require linear storage space — which is optimal without further compression. Even though not optimal for the find, insert and remove operations, both data structures are being better than linear in all respects. Assuming all nodes have the same degree, i.e. all edges are distributed uniformly over the graph, the average degree can be crudely estimated by dividing twice the number of edges by the number of nodes: $\frac{2|E|}{|V|} < |E|$. In real networks, the distribution is mostly non-uniform and exhibits different distributions, e.g. binomial, poisson or power law type [15]. 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 appealing 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 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	$\mathcal{O}(E)$	$\mathcal{O}(V)$	$\mathcal{O}(E)$	$\mathcal{O}(E)^1$	$\mathcal{O}(1)$
In_v	$\mathcal{O}(E)$	$\mathcal{O}(V)$	$\mathcal{O}(E)$	$\mathcal{O}(E)^1$	$\mathcal{O}(\deg(V))$
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.

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

2.2.3 Traversal-based Algorithms

Random Walk

A random walk is a stochastic process, originally defined by Karl Pearson posing the following problem to the readers of the journal *Nature* in 1905 [21]:

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 gathered wide interests and has many connections ranging from financial mathematics [3], over physics and biology (brownian motion[5]) to pure mathematics [27]. Random walks are modelled mathematically using markov chains. For further information on the pure mathematical theory the reader is referred to a comprehensive survey of the topic [17]. Our focus will remain database oriented, that is traversal-based. In [10] 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.

Algorithm 1: 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, \dots, e_{n-1})

begin

```

    visited_edges ← edge_t[n];
    current_node_edges ← expand( $G, v\_id, d$ );
    edge ← NULL;
    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( $G, edge.other\_v\_id, d$ );

```

Pseudo-code describing the algorithm can be found in listing 1. 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 retrieving the neighbourhood of vertex. The average space complexity of a random walk is $\mathcal{O}(\deg(V))$.

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 appearance in modern citation history [18]. 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 point where you can take a different path, chose that path to chase and repeat. 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 [16]. 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.

Algorithm 2: 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  $\leftarrow$  array initialized to  $-1$ ;
    parents  $\leftarrow$  array initialized to  $-1$ ;
    node_stack  $\leftarrow$  create_stack();
    push(node_stack, v_id);
    while node_stack non-empty do
        node_id  $\leftarrow$  pop(node_stack);
        current_node_edges  $\leftarrow$  expand( $G$ , v_id,  $d$ );
        for edge  $\in$  current_node_edges 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;

```

A pseudo-code description of it is given in 2. 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 [13] 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 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 breadth 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 [28]. 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 BFS 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 10.

Like the DFS, the BFS traversal schema can be used to find shortest paths, maximum flows

2.2. GRAPHS

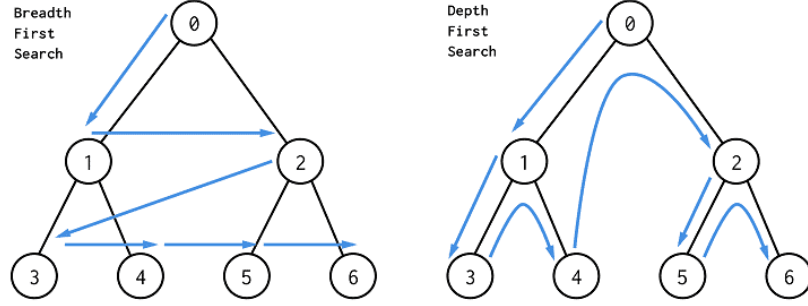


Figure 10 Comparison between a DFS and a BFS traversal.

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 3. We again make the assumption, that we are using an incidence list as data structure and that the expand operator is implemented accordingly.

Algorithm 3: 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_edges 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;

```

Dijkstra

In the original formulation, Edsger Dijkstra formulated the algorithm as a solution to the problem of finding a shortest path between two nodes [7]. Many variants and extensions exist of which we are going to discuss two: A* and ALT [14, 11]. One slight variation makes it possible to find all shortest path from a given starting 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 [6].

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 necessity 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: $T_d, T_m \in \log(|V|)$. Overall the asymptotic runtime complexity using a min-priority queue is $\mathcal{O}(|V| + |E|) \log(|V|)$ [12]. 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 $\mathcal{O}(|E| + |V|^2)$ [12]. Finally more advanced data structures can be used, like a Fibonacci-Heap [6]. These make it possible to update the priority in $\mathcal{O}(1)$ and still find the minimum in $\log(|V|)$. This yields the optimal asymptotical runtime of $\mathcal{O}(|E| + |V| \log(|V|))$.

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 $G = (V, E)$, start vertex id v_id

Output: Path distance distances, predecessor edges parent

begin

distances \leftarrow array initialized to ∞ ;

parents \leftarrow array initialized to -1 ;

path_queue \leftarrow create_min_prio_queue();

enqueue(path_queue, v_id);

while path_queue non-empty **do**

node_id \leftarrow dequeue(path_queue);

current_node_edges \leftarrow expand(G, v_id, d);

for edge \in current_node_edges **do**

if distances[edge.other_v_id] \geq 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;

A*

2.3 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.

2.4. EXAMPLE: NEO4J

- $\lambda : (V \times V) \rightarrow E$ 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.
- $f_P : V \cup E \rightarrow 2^P$ a function that assigns a set of properties to a node or relationship.
- $f_T : E \rightarrow T$ a function that assigns a type to a relationship.
- $f_L : V \rightarrow 2^L$ 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 [2, 23].

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 length 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 11. Neo4j is a graph database employing the

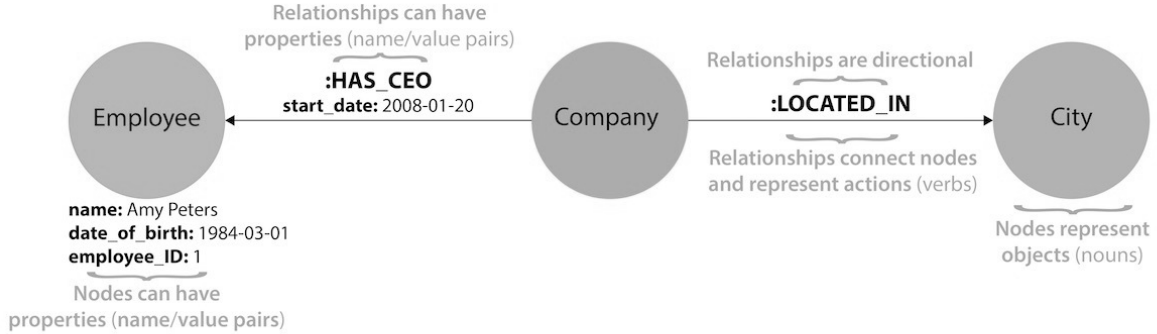


Figure 11 A visualization of the property graph model

property graph model [22], which is used in the evaluation part of this thesis.

2.4 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 12.

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.

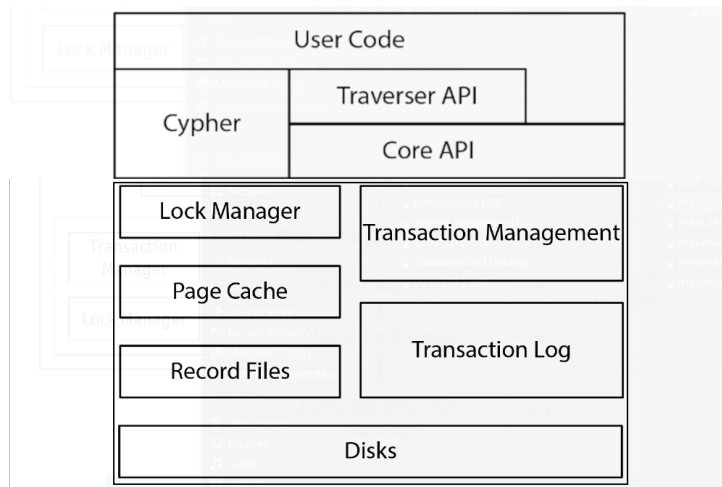


Figure 12 The high level architecture of Neo4J according to Emil Efrim, the co-founder of Neo technologies [22].

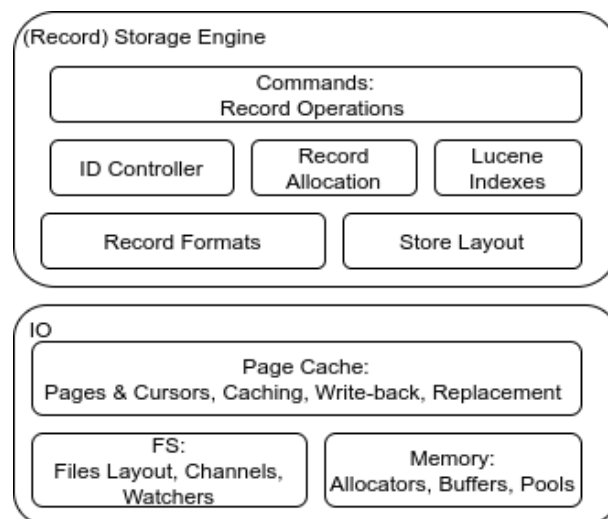


Figure 13 A visualization of the broad the storage and memory organization of Neo4J.

IO & File Layout

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 13. Most importantly the files ending with `.db` contain all fixed size records representing the structure of the graph and small amounts of data.

Caching & Indexes

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.

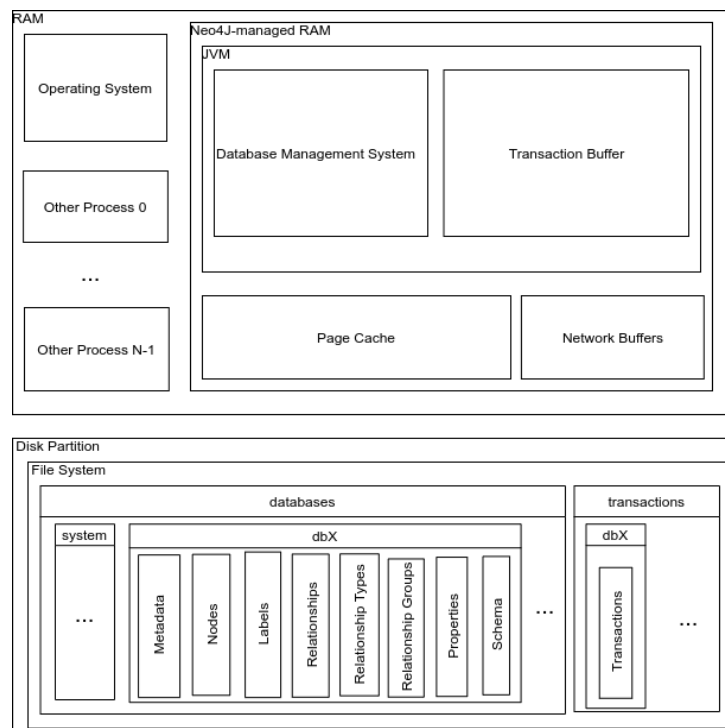


Figure 14 *A sketch of how Neo4J occupies memory.*

The overall memory and storage state of a Neo4J instance and its environment may thus be visualized like this figure 14.

Record Structures

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 $15 \text{ Bytes} \cdot 100 = 1500$ 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 awful lot of relationships and is treated a bit differently. That is a relationships are stored by type and direction for this node into groups.

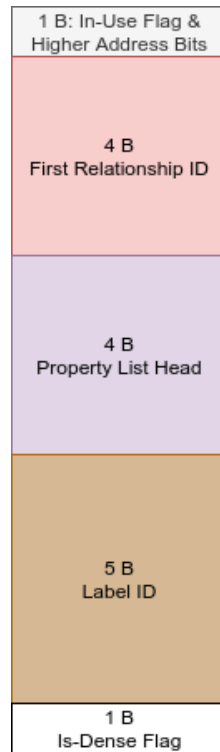


Figure 15 *A visualization of the record structure of a node.*

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.

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.

2.4. EXAMPLE: NEO4J

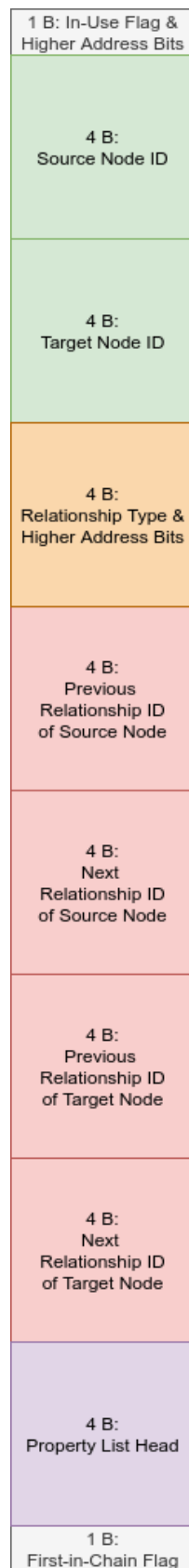


Figure 16 *A visualization of the record structure of a relationship in Neo4J.*

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.

All further record structures like relationship types, properties, arrays and strings are outlined in the appendix for the interested reader and omitted here for the sake of succinctness.

3 Problem Definition

3.1 Locality

Metrics

3.2 Problem Definition

3.3 Example: Vertex, Edge and Incidence List Order

4 Related Work

4.1 G-Store: Multilevel Partitioning

4.1.1 Coarsening

4.1.2 Turnaround

4.1.3 Uncoarsening

Projection

Reordering

Refinement

4.2 ICBL: Diffusion Set-based Clustering

4.2.1 Identify Diffusion Sets

4.2.2 Coarse Clustering

4.2.3 Block Formation

4.2.4 Layout

5 Methods

5.1 Louvain method

5.2 Partition order

5.3 Incidence List Rearrangement

6 Experimental Evaluation

6.1 Experimental Setup

6.1.1 Implementation

6.1.2 Environment

6.1.3 Data Sets and Queries

6.2 Results

7 Conclusion

7.1 Discussion

Remention contributions

7.2 Future Work

Dynamic rearrangement (Partition order)

7.3 Summary

Bibliography

- [1] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.
- [2] Renzo Angles. “The Property Graph Database Model.” In: *AMW*. 2018.
- [3] Louis Bachelier. “Théorie de la spéculation”. In: *Annales scientifiques de l’École normale supérieure*. Vol. 17. 1900, pp. 21–86.
- [4] Andries E Brouwer and Willem H Haemers. *Spectra of graphs*. Springer Science & Business Media, 2011.
- [5] 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.
- [6] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.
- [7] Edsger W Dijkstra et al. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), pp. 269–271.
- [8] *Eigen: The Matrix class*. Dec. 5, 2020. URL: https://eigen.tuxfamily.org/dox/group_TutorialMatrixClass.html (visited on 03/11/2021).
- [9] 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.
- [10] 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.
- [11] Andrew V Goldberg and Chris Harrelson. “Computing the shortest path: A search meets graph theory.” In: *SODA*. Vol. 5. Citeseer. 2005, pp. 156–165.
- [12] M. Goodrich and R. Tamassia. “Algorithm Design and Applications”. In: 2014.
- [13] J. Gross and Jay Yellen. “Graph Theory and Its Applications”. In: 1998.
- [14] 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.
- [15] Petter Holme. “Rare and everywhere: Perspectives on scale-free networks”. In: *Nature communications* 10.1 (2019), pp. 1–3.
- [16] D. Knuth. “Dancing links”. In: 2000.
- [17] László Lovász et al. “Random walks on graphs: A survey”. In: *Combinatorics, Paul erdos is eighty* 2.1 (1993), pp. 1–46.
- [18] 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.
- [19] *Matrix Storage Schemes*. Oct. 1, 1999. URL: <http://www.netlib.org/lapack/lug/node121.html> (visited on 03/11/2021).

- [20] *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).
- [21] Karl Pearson. “The problem of the random walk”. In: *Nature* 72.1867 (1905), pp. 342–342.
- [22] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases: new opportunities for connected data*. "O'Reilly Media, Inc.", 2015.
- [23] Marko A Rodriguez and Peter Neubauer. “The graph traversal pattern”. In: *Graph Data Management: Techniques and Applications*. IGI Global, 2012, pp. 29–46.
- [24] Angelika Steger. *Diskrete Strukturen: Band 1: Kombinatorik, Graphentheorie, Algebra*. Springer-Verlag, 2007.
- [25] Robin Steinhaus, Dan Olteanu, and Tim Furche. “G-Store: a storage manager for graph data”. PhD thesis. University of Oxford, 2010.
- [26] Louis Weinberg. “Kirchhoff’s’ third and fourth laws’”. In: *IRE Transactions on Circuit Theory* 5.1 (1958), pp. 8–30.
- [27] Norbert Wiener. *Collected Works, Vol. 1*. 1976.
- [28] Konrad Zuse. “Über den allgemeinen Plankalkül als Mittel zur Formulierung schematisch-kombinativer Aufgaben”. In: *Archiv der Mathematik* 1.6 (1948), pp. 441–449.