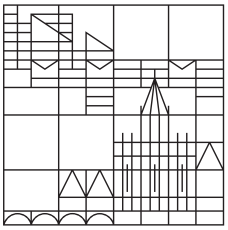
Locality Optimization

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:

Graphs are omnipresent in our world. Biological systems like brains and the spreading of diseases are modeled using graphs. Graph-based representations of social networks among individuals are popular in data science and analysis. Many navigation systems visualize the geographic position, maps, and routes using graphs. One tool to examine a graph's structure is traversal-based algorithms. An example is to find the shortest paths between two places or to compute differential equation-based spreading processes. Databases provide means to store large amounts of data reliable and scalable. The bottleneck to data-intensive information processing in current computing systems is the amount of time spent loading and storing data to secondary storage. Commonly, one of the most crucial aspects of scalability in databases is the careful design of data access. Relational databases have been optimized for decades, while graph databases are relatively new research in this respect. Former systems optimize selecting and joining tabular data, as the queries consist mainly of filtering data from different tables. The most common query types used in graph databases are pattern-based and traversal-based queries. While the optimization of pattern-based queries has been explored to a certain degree, traversal-based ones received only a little attention. The graph's structure governs the sequence in which traversal algorithms inspect records. Minimizing disk accesses is crucial for the performance of traversal-based queries. The principle of leveraging patterns where records accesses are close in space or time is called locality. Rearranging the records is one way to achieve this. When these records are accessed together, they also need to be stored together. We present a survey of state-of-the-art graph record rearrangement strategies, along with the proposition of an improvement to such methods on a particular record structure. Finally, an evaluation of these rearrangement strategies provides insights into their quality. Quantitatively, we measure the amount of block IOs necessary for executing a traversal-based algorithm.

Acknowledgements

I owe an enormous debt to Michael Grossniklaus. As a teacher, his lectures were exceptionally instructive. As a mentor, he always provided me with his guidance and support.

I cannot overexpress my gratitude towards my parents. They raised me to the person that I am. Only their support made it possible to study what is my passion.

Further, I want to thank my siblings Leo Klopfer, Jasmin Wetzel, her husband Marius, and my girlfriend Natascha Reddemann for always being there for me and having an ear open when the times were stormy.

Working with colleagues and spending time with friends augmented my time here in Konstanz. Thanks to Dario Graf, Stephan Perren, Jannik Bamberger, Leo Wörteler, Manuel Hotz and many others.

Finally, I'd like to thank Theodoros Chondrogiannis for the discussions, his clearness, and the ability to keep me focused.

Contents

1. Introduction

Graph-structured data is omnipresent in our world: road networks are naturally modeled using graphs. When thinking about social structures, people can be modeled as nodes and interactions as edges. In dynamical systems like the brain, the system's structure can be modeled as a graph, and the dynamics of the process can be modeled as algorithms on the graph structure like the spreading activation algorithm and by changes to the network itself [4, 24]. The internet and routing protocols used by the hardware nodes rely on graph theory to optimize the flow of information [84]. In recent years, modeling pandemic dynamics using graph-based models, like the susceptible-infected-recovered and the susceptible-infected-susceptible model, has gained a lot of attention [59, 23, 92, 17].

Reliable and maintainable storage of graphs is necessary when they grow larger and complexity. In addition to resolving maintainability issues, databases provide exceptional performance for some operations. Typical queries on graph data are pattern-based queries and traversal-based queries. While pattern-based queries are used to inspect nodes and edges with specific attributes and their neighborhood, traversal-based queries are used for navigation and structural exploration. Given a knowledge graph, if we want to retrieve related concepts to a given one, then a breadth-first traversal can be applied [4]. If one wants to find connections between concepts, shortest pathfinding algorithms provide means to examine connections [72]. Similarly, when planning a route, shortest pathfinding algorithms are employed. Moreover, when searching for some specific kind of place in the surrounding, like when looking for a bar, the next theater, or the next gas station, then another kind of shortest path algorithm is applied, as we will see. [12]

Nevertheless, how does one make sure that these algorithms are fast? The bottleneck for algorithms operating on large-scale input data is primarily the time spent to load the data. Caches are about 50 times faster than DDR4 RAM, which is $1,000$ times faster than a solid-state drive and about $100,000$ times faster than a hard disk drive [91]. In effect, we want to minimize the number of disk IO operations that need to be done when executing a query. This topic has already been tackled in other types of databases, like relational databases. A key element to this is the concept of locality. The reason why caching and buffering works is the so-called locality of reference [96, 54]. Most of the memory accesses target only a fraction of the overall data but with a very high frequency. Here we are going to focus on spatial locality: we want to layout the records such that when an element is accessed, the subsequent access is within the neighborhood of the last one. With X_t the address accessed at time t,

$$\max P(X_{t+1} = A \pm \varepsilon | X_t = A).$$

As disks read and write data based on blocks, packing data such that accesses remain local saves IO operations. More specifically, whenever subsequent access stays within a block, we need one loading operation less.

To perform such reorganization, one can reorganize them statically like in relational databases. These store data in tables, which are filtered and then joined together. The row order is determined by the keys most frequently used for joining tables to optimize the two operations. For graphs, the structure is crucial to the traversal-based queries. Pattern-based queries are explored to a certain degree, and their optimization is similar to relational queries [43]. When dealing with traversal-based queries, what is accessed next depends only on how nodes are connected — the graph structure. Thus we will address the issue by elaborating on static record layout methods based on the graph's structure.

The contributions of this thesis are

- a concise description of the problem.
- measuring the impact of data organization on the IO behavior and thus the performance of traversal-based queries.
- a survey of existing static rearrangement methods.
- the proposition of an extension to the current approaches: reorder the incidence lists after reorganizing the data to reestablish locality and sequential access after rearrangement.
- the implementation of an in-memory graph database, traversal-based queries, and the above improvement.
- an extensive evaluation of the existing methods with and without the proposed extension.

The rest of this thesis is organized as follows. In the second chapter, graphs are defined formally, along with possible representations and the traversal algorithms. The third chapter treats the architecture of graph databases and a widely-used data model for graph databases. After setting the context, the concept locality is defined in the fourth chapter, and an explicit problem definition is given. Recent methods and an extension of those are discussed in the fifth chapter. The sixth chapter presents the experimental evaluation. Finally, in the seventh chapter, the thesis is summarized.

2. Graphs

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

2.1. Definitions

We follow the notation, most authors use in the field [94, 42, 1, 21, 40].

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 the 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 neighborhood 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 neighbors 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 neighbors 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. Two villages 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 neighborhood of this specific village.

A graph is **undirected**, if E is a symmetric relation, $(u, v) \in E \Rightarrow (v, u) \in E$. Otherwise, the graph is called **directed**, the order within the tuple matters, and E is not symmetric. The incoming and outgoing edges are defined by restricting the vertex's position on the edge. 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, a road has a direction in which all cars drive. This behavior 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 is called unweighted. Similarly, labels can be assigned to both nodes and edges. In some cases, these labels may encode the type of entity. Other arbitrary key-value pairs may be assigned to either the nodes or the edges, the so-called properties. An example of 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 connected by the road. To include labels, one could distinguish between highways and minor roads or assign the road's name. The former would model the road type, while the latter would be a (potentially non-unique) identifier.

In case there may exist multiple edges between the same pair of nodes in the same direction, then the graph is called **multigraph**. That is, $E_M = (E, M)$ is a multiset, with $M : E \to \mathbb{N}$. Imagine one tries to model the transportation links between major cities. There are many possible means: highways, railways, flights, and some sea routes. In particular, two cities may be connected by more than one means 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 one 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 set are given later in this chapter. A **cycle** is a trail, where the first visited vertex is also the last visited vertex. If one starts a route from home, goes to work, and returns home after closing time, the 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. Representations

When implementing graphs for computing machinery, there are some possibilities for representing 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 using additional fields or pointers. The definitions of the data types and parts of the complexity analysis are based upon [42, 1, 21, 40, 95]. Besides the ones elaborated on below, there are the compressed sparse column and row (CSC/CSR) representations used for sparse matrices in arithmetics-heavy applications, like in the library Eigen or Matlab [95, 30]. In Figure 1 you can see a visualization of the graph that is used as an example throughout this section. The representations that are described in the main text are list-based. Matrix-based representations can be found in the appendix in Section A.1.1.

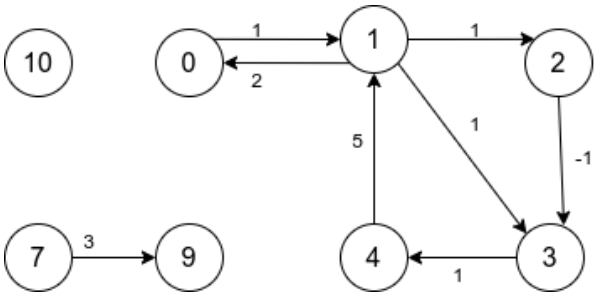


Figure 1. *An example graph used throughout this section.*

2.2.1. Unordered Edge List

The simplest representation uses an unordered list of edges. Each element of the data structure carries the information of exactly one edge. For example, in a directed, weighted graph, the source and target node's indices and the edge weight are one entry. Additionally, an edge list needs to store a list of vertex indices to represent nodes with no edges. Overall this results in $\mathcal{O}|E| + |V|$ 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 inspecting the list of vertices, thus $\mathcal{O}(|V|)$. Vertex insertion's asymptotic runtime is $\mathcal{O}(1)$, assuming the list stores a pointer to its tail. 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. $\mathcal{O}(|E|)$ for edges. 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 neighborhood N_v of a vertex requires a scan of all edges again, 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 Listing 2.

```
0 1 2 3 4 7 9 10

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

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

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

2.2.2. Adjacency List

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

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

The number of nodes can be retrieved in $\mathcal{O}(1)$, as it is 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 iterating 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 looking at $\mathcal{O}(\deg(V))$ elements. For the insertion of an edge, one needs to append one element to the end of the source node's adjacency list, which can be done in $\mathcal{O}(1)$. Removing an edge again requires iterating 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 neighborhood of a vertex is already stored and can be returned in $\mathcal{O}(1)$. In contrast for the incoming neighborhood 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 neighborhood 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 neighborhood queries are in $\mathcal{O}(1)$. An example of this data structure is shown in Listing 3.

2.2.3. Incidence List

This representation is also called incidence table in [42]. 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 duplicate storage of every edge comes with an additional cost of $|E|$ memory but is beneficial when accessing information. Another benefit is that the additional costs can be mitigated by using references.

```
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 4. *An example of the incidence list representation of a graph.*

Most 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 neighborhood. As all edges that are incident to a node are stored, finding all neighbors is an $\mathcal{O}(1)$ operation. Considering the incoming and outgoing neighborhoods, 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 Listing 4.

2.2.4. Summary

While edge lists can represent all graph variations, 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) [71, 29, 70]. When dealing with multi-graphs, the adjacency matrix representation requires additional arrays (one per edge "type") or cannot canonically represent them. The incidence matrix cannot 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 transpose, one gets the sum of the adjacency matrix and the gradient matrix, i.e., the laplacian matrix [15]. Furthermore, it helps in physical flow problems and simulations, e.g., when computing the current and resistances in a graph or simulating micro-circuits [102]. Even though the incidence matrix typically requires less space than the adjacency matrix, both options are rather unfeasible when storing large graphs, and the incidence matrix provides even worse access times than edge lists. The compressed sparse row and compressed sparse column storage formats are very similar to adjacency lists as a side note. 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 the most negligible storage. These formats are used for sparse matrix arithmetics in some of the most popular matrix arithmetics libraries, like [71, 29, 70].

	Edge List	Adjacency Matrix	Incidence Matrix	Adjacency List	Incidence List
Space	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(\|V\| \cdot \|E\|)$	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\| + \|E\|)$
`size` $\|E\|$	$\mathcal{O}(1)$	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(1)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|E\|)$
`find` v	$\mathcal{O}(\|V\|)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`insert` v	$\mathcal{O}(1)$	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(\|V\| \cdot \|E\|)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`remove` v	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(\|V\| \cdot \|E\|)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\deg(V))$
`find` e	$\mathcal{O}(\|E\|)$	$\mathcal{O}(1)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\deg(V))$	$\mathcal{O}(\deg(V))$
`insert` e	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\|V\| \cdot \|E\|)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`remove` e	$\mathcal{O}(\|E\|)$	$\mathcal{O}(1)$	$\mathcal{O}(\|V\| \cdot \|E\|)$	$\mathcal{O}(\deg(V))$	$\mathcal{O}(\deg(V))$
`is_adjacent` u, v	$\mathcal{O}(\|E\|)$	$\mathcal{O}(1)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\deg(V))$	$\mathcal{O}(\deg(V))$
`get` N_v	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|V\|)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(1)$
`get` In_v	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|V\|)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\|E\|)$	$\mathcal{O}(\deg(V))$
`get` 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.*

Finally, the adjacency and incidence lists are similar in many aspects. Both require linear storage space — which is optimal without further compression. Even though not optimal for the operations `get`, `insert` and `remove`, both data structures provide access times that are asymptotically better than linear in most cases. If the edges are distributed uniformly, we have an upper bound for the average degree of $\deg(V) = \frac{2|E|}{|V|} \leq \frac{2|V|^2}{|V|} = 2|V|$. In real networks, the

distribution is often non-uniform but can be modeled using e.g., binomial, Poisson, or power-law type [50]. A power law distribution would mean that few nodes with a high degree and many nodes with a relatively low degree. What is also very appealing is that the adjacency list, and especially the incidence list enables one to return the neighborhood of a vertex in a 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 following subsection.

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

2.3. Algorithms

There are many kinds of graph problems and algorithms tackling them. As we are interested in the access patterns of traversal-based queries, we will focus on traversal and shortest path algorithms. We also look at some partitioning methods, which are frequently used when addressing the issues in the next chapter. Nevertheless, other types of algorithms are of equal importance and have many use cases, like flow problems, finding graph isomorphisms, identifying minimal spanning trees, determining a node's centrality, and others. Comprehensive resources for these problems can be found in [94, 42, 1, 21, 40].

2.3.1. Traversal

Visiting the nodes in a graph is known as graph traversal. The respective algorithm gives the order in which the nodes are visited. The two most crucial graph traversal schemas are the breadth-first search and the depth-first search. Another such schema is the random walk, which is not quite a traversal but instead generates an arbitrary walk. However, it specifies how nodes are visited, but it does not guarantee anything but well-connectedness.

Depth-First Search

Charles-Pierre Trémaux authored one description of the depth-first search (DFS) in 1818. Even though the work was published a lot later, this is the first appearance in modern citation history [69]. He used so-called Trémaux trees to solve any mazes. Each result of a depth-first search is such a Trémaux tree.

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

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

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

These have the property that every two adjacent vertexes are in an ancestor-descendant relationship. Even though Trémaux trees themselves are attractive — 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 one can take a different path, choose that path to chase, and repeat. 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 [62].

Even though DFS is a general traversal schema – go deep first – it can also be used for other purposes like finding shortest paths, connected components, testing planarity, and many more.

A pseudo-code description of it is given in Algorithm 1. This version continues its search at the last found edge instead of the last visited edge. The modification is to enable the usage of implementation for other problems like finding spanning trees, cycles, or paths. The runtime is again dependent on the used data structure, that is used and again the runtime for querying the neighborhood of a node is the varying term. Each node is visited exactly once, and by the handshaking lemma [42] 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

Konrad Zuse gave the first description of breadth-first search (BFS) in modern science in his Ph.D. thesis on the "Plankalkühl". He used it to find connected components [108]. The breadth-first search schema was also used to find shortest path solutions for mazes and wire up placed electrical components on a printed circuit board (PCB). The general traversal scheme in breadth-first search is to explore all next neighbors before continuing with those who are more steps away. Thus it first explores one "level" exhaustively before continuing to the next one. In other words, the only difference between DFS and DFS is the data structure that is used. While DFS uses a stack and thus always inspects the last inserted element, BFS uses a queue. That means it inspects the element that was inserted first. Illustrative comparison of DFS and BFS is shown in Figure 5.

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

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

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

Like the DFS, the BFS traversal schema can find the shortest paths, maximum flows and test if a graph is bipartite with some modifications. 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 Algorithm 2. We again aussume that we are using an incidence list as s data structure and that the expand operator is implemented accordingly.

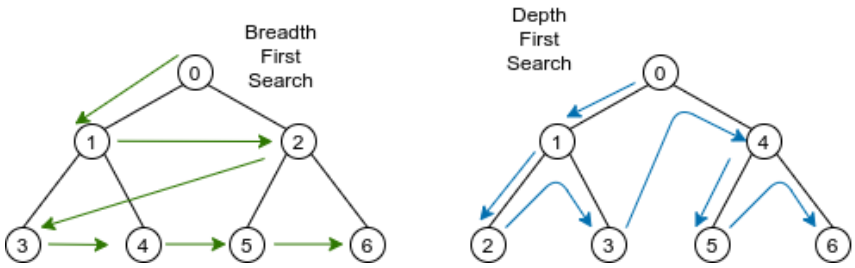


Figure 5. *Comparison between a DFS and a BFS traversal.*

Random Walk

A random walk is a stochastic process, originally defined by Karl Pearson posing the following problem to the readers or the journal nature in 1905 [80]:

> 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 broad interests and has many connections ranging from financial mathematics [8], over physics and biology (Brownian motion[16]) to pure mathematics [103]. Random walks are modeled mathematically using Markov chains. The purely mathematical theory is outlined in a comprehensive survey [68]. Our focus will remain database-oriented and traversal-based. In [34] 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 3: Pseudo-code for a random walk on a graph G.

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

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

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

A random walk's runtime complexity 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 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 vertex neighborhood. The average space complexity of a random walk is $\mathcal{O}(\deg(V))$.

2.3.2. Shortest Path

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

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

2. the A* algorithm finds the shortest path between a single source and a single target node.

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

Dijkstra

In the original formulation, Edsger Dijkstra formulated the algorithm as a solution to the problem of finding the shortest paths between two nodes [27]. Many variants and extensions exist, of which we are going to discuss two — A* and ALT [46, 39]. One slight variation makes it possible to find all the shortest paths from a given source node. The algorithm, however, restricts the graph. Only positive weights are allowed; otherwise, a negative cycle results in an infinite loop [21].

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

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

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

Conceptually Dijkstra's algorithm assigns each node in the graph a distance. The source vertex has the distance 0, while all other vertices have a distance of infinity at the beginning. The current node's path distance is added to the edge weight for all edges and inserted into the priority queue. The algorithm always extracts the vertex with the so-created minimal path until all vertices are visited. A binary heap-based priority queue or a Fibonacci heap is used to define the traversal order to compute the minimum efficiently.

Pseudo-code for the variant that computes all shortest paths can be found in Algorithm 4. Dijkstra's algorithm's runtime complexity 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 maintain those, the runtime complexity is equivalent to what we had with BFS. Using a binary heap-based priority queue yields sub-optimal runtime: $T_d, T_m \in \log(|\mathcal{V}|)$. Overall the asymptotic runtime complexity using a min-priority queue is $\mathcal{O}((|V|+|E|)\log(|V|))$ [40]. 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 $\mathcal{O}(|E| + |V|^2)$ [40]. Finally, more advanced data structures can be used, like a Fibonacci-Heap [21]. These make it possible to update the priority in $\mathcal{O}(1)$ and still find the minimum in $\log(|\mathcal{V}|)$. This yields the optimal asymptotical runtime of $\mathcal{O}(|E| + |V|\log(|V|))$. The worst-case space complexity is again $\mathcal{O}(|V|)$, which is when all nodes are stored in the queue simultaneously. As there is only one shortest path (paths of equal size are discarded) to each node, the queue contains each node only once.

A*

The A* was initially invented in the late '60s to be used for a robot's path planning. It is an extension of Dijkstra's algorithm that does not just use the distance as a metric of priority but adds a heuristic $h : V \to \mathbb{R}$ to the distance.

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

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

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

$v \in V : f(v) = \text{distance}(v) + h(v)$, that has to fulfill certain conditions. With $u, v \in V$ and min distance(u) the minimal distance from the vertex u to the goal vertex

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

The former condition is called consistency, the latter admissibility. As all consistent heuristics are admissible, the first condition is sufficient. An example for graphs using the Euclidean coordinate system is the Euclidean distance [46].

Algorithm 5 shows pseudo-code for the algorithm. The runtime is dependent on the complexity of the heuristics. Overall we have the same worst-case complexity as with Dijkstra's algorithm

for the constant heuristic $\forall v \in V : h(v) = 0$. The best-case of the A* algorithm is when the heuristic is equal to the current vertex distance to the goal vertex. Then exactly min distance nodes are visited, and the algorithm is in $\mathcal{O}$) min distance), which is the global optimum for a single source shortest path problem.

ALT

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

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

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

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

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

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

After that A* is executed as described in Algorithm 7.

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

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

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

Besides the additional space that is used, we also execute Dijkstra's algorithm $|L|$ times and have an asymptotic complexity of $\mathcal{O}(|L| \cdot (|E| + |V| \log |V|))$ using an incidence list to store

the graph and a Fibonacci heap as the data structure for the priority queue. Regarding space we need $\mathcal{O}(|V| \cdot (1 + |L|))$. For small values of $|L|$, we preserve the worst-case complexity as the average-case complexity of ALT. We gain by that that the precomputations take the main runtime penalty while providing a reasonably good heuristic depending on the selection of the landmarks [39]. How to select the landmarks is discussed in [38].

2.3.3. Partitioning

Graph partitioning is the problem of separating the graph into disjoint subsets. The problem is NP-complete [5] and the algorithms below are thus heuristics. They do not provide the globally optimal partition concerning some metrics but approximate an optimal solution as close as possible. Besides the methods presented below, there are a lot of other methods. One method that is of considerable importance is spectral clustering, which we will not elaborate on here in-depth. Briefly, the algorithm computes the Laplacian matrix, extracts k eigenvectors, changes the basis of the matrix according to the k eigenvectors, and clusters the nodes then using a simpler algorithm, like agglomerative clustering [90] or k-means [67] The interested reader is referred to [28, 100, 75]. Apart from methods considering the graph structure as part of the algorithm itself, all other clustering algorithms can be applied if suitable features are extracted based upon the graph structure. As we will see later, one of the related work methods extracts graph features by executing multiple random walks per node and aggregates them into a multi-set to characterize the neighborhood of a node. A comprehensive survey of non-graph-based algorithms can be found in [55, 13, 104, 45]. Other graph feature extraction methods are, for example, described in [74, 47, 48].

Multi-Level Partitioning

The multilevel partitioning algorithm was first described by Henderson and Leland [49]. Its steps are visualized schematically in Figure 6 and given below.

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

A particular implementation of Karypis and Kumar [58] introduces a couple of improvements. A matching scheme for the coarsening stage is called heavy edge matching. It works by iterating over all vertices and matching each one with the vertex attached to the edge carrying the heaviest weight. This routine is applied until all edges are matched. Afterward, all pairs of vertices contribute to a new vertex in a new graph, the edges of each vertex pair are aggregated, and their weight is summed eventually if both nodes had an edge to the same vertex. Moreover, the nodes are assigned a weight that corresponds to the number of vertices matched in one meta-vertex. The partitioning stage experiments with three different methods. The KL-algorithm [60], spectral clustering [28], the graph growing partitioning algorithm — which uses BFS with a limited number of steps from a set of arbitrarily chosen vertices. Finally, it proposes an improved version of the refinement procedure, where only boundary vertices are considered for interchanges between partitions.

The runtime for heavy edge matching is $\mathcal{O}(|E|)$ for each level, and we have at least $\mathcal{O}(\log(\frac{|V|}{|V_c|}))$ steps, where V_c is the set of vertices in the coarsest graph. Assuming $|V_c|$ is a small constant (in the worst-case for refinement $|V_c| = 1$), we get overall $\mathcal{O}(|E| \log(|V|))$ for the refinement procedure. The partitioning step is dependent on the choice of the algorithm. Using the KL k-way algorithm we have $\mathcal{O}(k^2 \cdot |V_c|^2 \cdot \log(|V_c|))$. As $|V_c|$ is small compared to the original graph; this results in a reduced overall runtime complexity. Finally the uncoarsening including projection

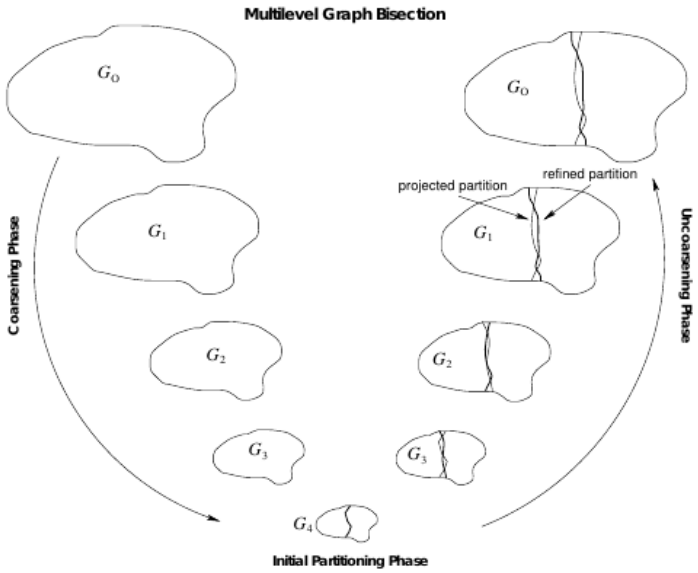


Figure 6. *A visualization of how the multilevel partitioning algorithm works [58].*

and refinement take $\mathcal{O}(\log(|V|) \cdot (|V_i| + \mathcal{O}(k^2 \cdot |V_i|^2 \cdot \log(|V_i|)))$ with $|V_i|$ the number of vertices in the i times coarsened graph. As the projection should already induce a reasonable initial partitioning, the algorithm should converge quickly [49]. The overall runtime depends heavily on $|V_c|$. The refinement phase is the most costly step as it needs to be done for the last level too, resulting in an asymptotic complexity of $\mathcal{O}(\mathcal{O}(k^2 \cdot |V|^2 \cdot \log^2(|V|)))$.

Louvain Method

The Louvain method [14] is an algorithm for community detection. The authors define community detection as a graph partitioning problem, where nodes within a partition shall be densely connected, while nodes belonging to different partitions shall be sparsely connected [14]. Measuring the quality of such partitioning can be done using the modularity of the partition as introduced by Newman and Girvan [36]

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

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

Optimizing the modularity computationally exactly is computationally hard, so as soon as the graph size increases, approximate optimization is required. The Louvain method is such an approximation. Its overall control flow comprises the following steps and is visualized in Figure 7.

1. Initialize the initial partition with each node in an own community
2. $\forall v \in V \forall u \in N_v$: Compute the gain in modularity $\Delta Q(u, v)$ when merging v into the community of u. Place v into the community of the neighboring node u where $\Delta Q(u, v)$ is maximal if the gain is positive.

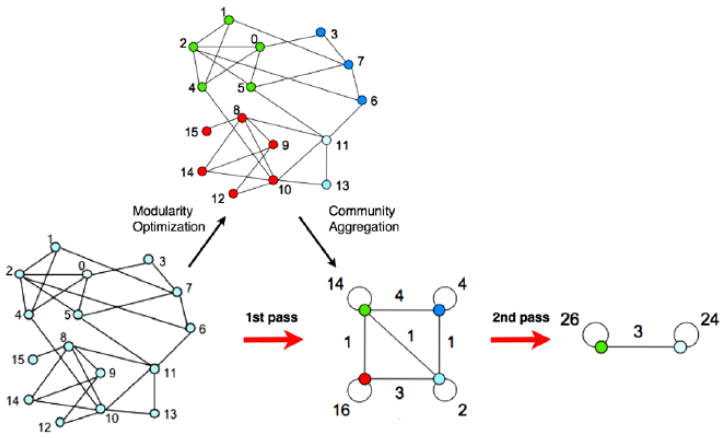


Figure 7. *The steps that the lovain method comprises [14].*

3. Go to step 2 until the improvement is below a threshold.

4. Construct a new graph G_i with the previous graph communities being the vertices of the new one and Aggregate the edge weights. Links between nodes of the same communities are aggregated to self-loops.

5. If the previous graph and the new graph differ, go to step 2 using the G_i as input graph, else terminate.

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

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

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

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

One problem with the Louvain method is its resolution limit. For vast and dense graphs, the total edge weight grows extremely large, and thus the modularity is always tiny. This problem is approached in two different ways. Conde-Céspedes et al. [19] experiment with different other modularity measures that are provably stable the graph size. However, those measures that provide good results require user defined-parameters, which require either domain-specific knowledge of the data or parameter optimization, which implies multiple executions of the algorithm.

3. Graph Databases

First, we introduce a popular data model employed by many current graph databases [37, 7, 3, 83]. Then we look at an implementation of this model in a native graph database called Neo4J is described. The latter also serves as a role model for implementing the in-memory database used in the evaluation part.

3.1. Architecture

Graph databases do not differ in many architectural aspects from relational and other databases. A sketch of the software architecture of database management systems is laid out in Section A.2. The most significant difference is how data is stored on the one hand side and how the queries are evaluated on the other hand side. As mentioned, we focus on the former issue.

Relational databases store data in tables. The links considered in this category of DBMS are primarily used to stitch together the fields of a record stored in different tables into one row again after the split to satisfy a particular normal form. One may also store tables where one table stores nodes and the other table's fields are node IDs to represent relationships.

However, to traverse the graph, one either has to do many rather expensive lookups or store auxiliary structures to speed up the lookup process. In particular, when using B-trees as index structure, each lookup takes $\mathcal{O}(\log(n))$ steps for clustered indices to locate a specific edge. However, as all edges are attached to two nodes, it is only possible to build an unclustered index over edges either grouped by incoming or outgoing neighborhood. The undirected neighborhood permits no such sorting and retrieval as it is unclear to which node an edge is to be sorted. Alternatively, one could store an additional table that holds incidence lists such that the lookup of outgoing or incoming edges is only $\mathcal{O}(\log(n))$ which would speed up breadth-first traversals, but duplicate data. Still, one has to compute joins to continue the traversal in the scope of depth-first searches. Another way to speed things up is to use a hash-based index, but this also has an unavoidable overhead aside from the joins and the question of which hash function and which attributes to use arises.

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

3.2. The Property Graph Model

The property graph model is a widely adopted data model to represent graphs in databases. It can represent the structure of directed or undirected, weighted or unweighted, and typed graphs having additional properties.

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

- V the set of vertices.
- E the set of edges.
- $\lambda : (V \times V) \to E$ a binary relation assigning a pair of nodes to an edge.
- P a set of key-value pairs called properties.
- T a set of strings used as relationship types.

- L a set of strings used as labels.
- $f_P : V \cup E \to 2^P$ a binary relation that assigns a set of properties to a node or relationship.
- $f_T : E \to T$ a binary relation that assigns a type to a relationship.
- $f_L : V \to 2^L$ a binary relation that assigns a node a set of labels.

The property graph model reflects a directed, node-labeled and relationship-typed multigraph G, where each node and relationship can hold a set of properties [6, 86, 87]. In a graph, the edges are usually defined as $E \subseteq (V \times V)$, but in the property graph model, edges have sets of properties and a type, making them records independently. An illustration of this model is shown in Figure 8.

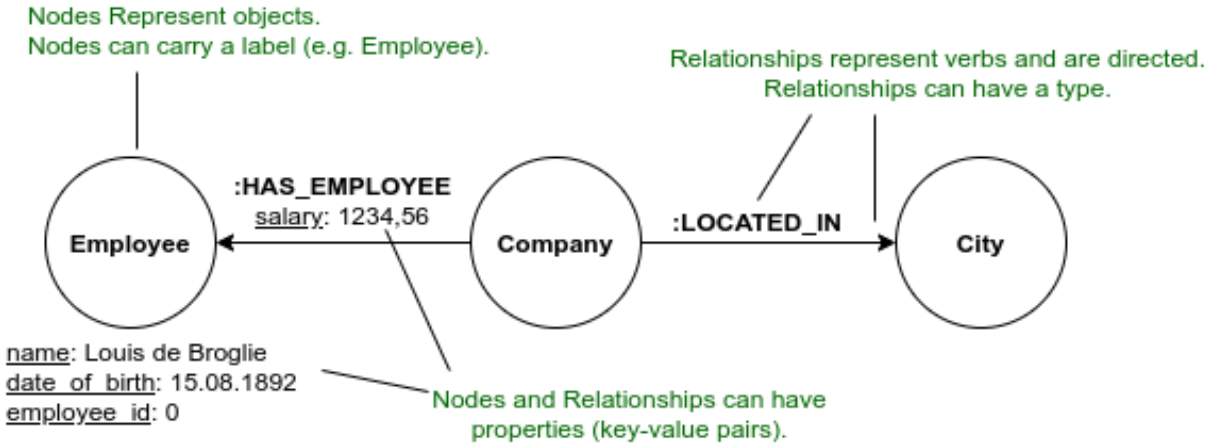


Figure 8. *A schematic visualization of the property graph model.*

Neo4j is a graph database employing the property graph model [85]. The logical operators of this model are described in [51]. The *get_nodes*-operator returns all nodes of the graph. No matter how the nodes are stored, the whole file (portion) needs to be scanned. Furthermore, the *expand*-operator returns the incident edges of a node depending on the direction. Expand only considers a part of all edges, so it does not do full scans but somewhat smaller reads. Finally, the *filter*-operand selects specific nodes or relationships based on properties, labels or, relationship type.

In the next Section 3.3 we are going to discuss how Neo4J implements the property graph model, with our focus on the structure of the graph and the low-level storage scheme.

3.3. Neo4J

Neo4J is a native graph database using the property graph model. The source code of the community edition is available at GitHub [37]. We look at some implementation details of the storage and buffer manager and the record structure. We will not take properties, relationship types, labels, and concepts related to those into account.

3.3.1. High-level Architecture

To get an overview of the architecture, let us consider Figure 9. This description was outlined by the co-founder of Neo4J Emil Effrem, the chief science officer Jim Webber and Ian Robinson, a former engineer at Neo4J, in their book on graph databases [85]. Here we can see that the architectural schema outlined in Section A.2 and especially Section 32 was not entirely applied.

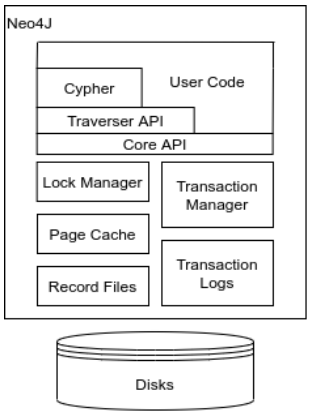


Figure 9. *The high level architecture of Neo4J [85].*

Still, the components are very similar. The ”Page Cache“ is equivalent to the buffer manager, the record files are what is managed by the disk space manager, mechanisms to deal with free slots [77] and (de-)allocations [78] are also part of the software stack, as are the record formats [79] and indexes, corresponding to the access layer. The corresponding components are just put together in a slightly different manner.

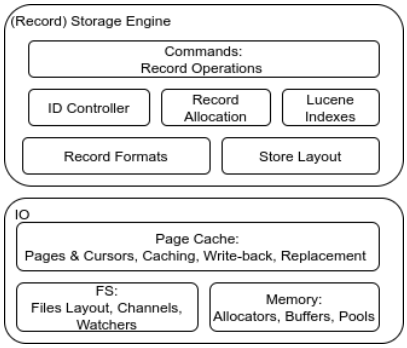


Figure 10. *A visualization of the broad storage and memory organization of Neo4J.*

The detailed composition is shown in Figure 10. The IO package contains the page cache, which is the buffer manager. It also contains facilities to create, grow and shrink files using the `java.nio` library and wrappers around platform-dependent allocation facilities. Thus the (de-)allocation part of the disk space manager resides in the IO package, too. The record storage engine defines the record format and the file layout and means to create and maintain indices. Thus it is similar to the access layer. It also handles the management of free slots — something that the disk space manager usually does. To summarize: the buffer manager and the access layer correspond closely to these two packages, while the disk space manager is distributed mainly over these two packages.

3.3.2. Record and File Structures

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

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

– Properties, labels, relationship types

Each record type is stored in an own file per database in the database management system. An additional system database keeps track of the existence and metadata of the other ones storing user data. This is visualized in Figure 11.

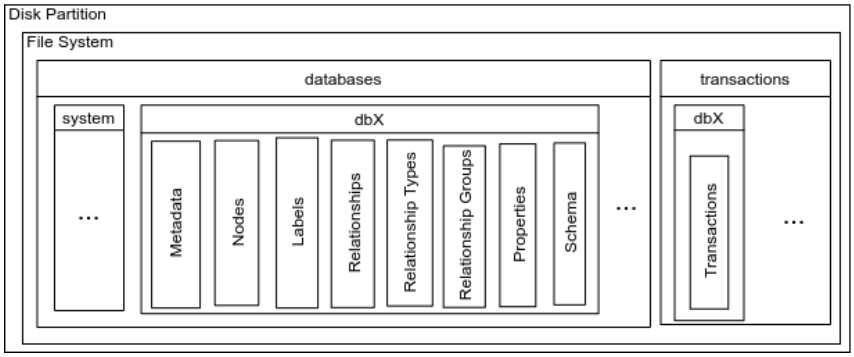


Figure 11. *A visualization of how the files are arranged of Neo4J.*

The records are ordered simply by their insertion order, i.e., the files storing the records are heap files. While variable-length records store strings and arrays, labels, for example, store a pointer to the actual string of the label to be fixed size and thus efficiently retrieved. The same is valid for relationship types and property keys and values that are strings or arrays. The separate storage is done to avoid duplications of strings, e.g., of each label. As mentioned before, we will elaborate on the elements that represent the graph structure for the sake of succinctness. Only one thing is to be mentioned. Properties are stored as linked lists for each node and relationship.

Node Records

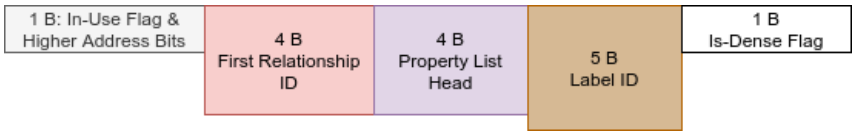


Figure 12. *A visualization of the record structure of a node record [32].*

The record format of nodes consists 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 is outlined below.

1. Byte 1: One bit for the in-use flag. The additional bits are used to compress the node struct by using the other 7 bits to store the most significant bits of the first relationship ID and the first property ID
2. Bytes 2 — 5: The next 4 Bytes represent the head's relationship ID in the linked list of the considered node's relationships.
3. Bytes 6 — 9: Again, 4 bytes encode the head's property ID in the node's linked list of properties.
4. Bytes 10 — 14: This 5-byte section points to the labels of this node.
5. Byte 15: The last byte stores if the node is dense, i.e., has an awful lot of relationships, such that it needs special treatment to remain efficient to traverse over. That is, relationships are grouped by type and direction for this node.

To summarize: the records on disk are stored as in the enumeration above, and as shown in Figure 12. In the database, all IDs get mapped to longs, and their respective space is larger than the space representable by 35 bits — which 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.

Relationship Records

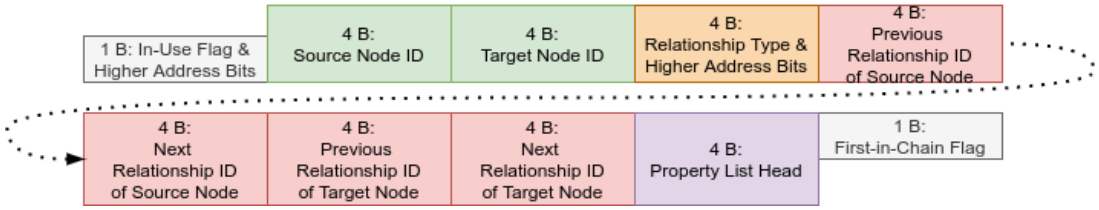


Figure 13. *A visualization of the record structure of a relationship in Neo4J.*

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

1. Byte 1: In-use bit, source node high order bits (3 bits), first property high order bits (4 bits)
2. Bytes 2 — 5: source node ID
3. Bytes 6 — 9: target node ID
4. Bytes 10 — 13: relationship type (16 bit), target node high order bits (3 bits), relationship previous and next ID higher bits for source and target node ($4 \cdot 3 = 12$ bits), one unused bit.
5. Bytes 14 — 17: previous relationship ID the for source node
6. Bytes 18 — 21: next relationship ID for the source node
7. Bytes 22 — 25: previous relationship ID for the target node
8. Bytes 26 — 29: next relationship ID for the target 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 byte's lowest two bits. The other 6 bits are unused.

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

First, as already mentioned, the edges are not physically duplicated but only referenced. The records are fixed-sized, so addressing them is done by multiplying the index by the size of an

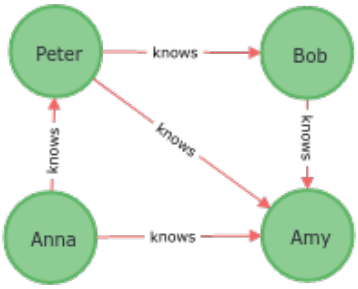


Figure 14. *An example graph.*

entry, meaning one does not need to store primary keys explicitly, and address translation can be done using simple multiplication. Theoretically, one could align the record size to a power of two to turn the multiplication into a bit shift. Next, as doubly linked lists are used, the deletion of an edge is in $\mathcal{O}(1)$ if the ID is known. This variant of deletion assumes that the doubly linked incidence lists are circular, i.e., the head's pervious element is the tail and reciprocally. If this were not the case, the incidence list would need to be traversed to find the last element. Also, the incidence list is stored in the relationships. Thus to traverse from one node's incidence list to another, there is no need to load the node record itself. It suffices to dereference the next element in the incidence list stored by the relationships and store the edge's ID that started the traversal.

To conclude this example, we briefly visualize the just described storage schema. The high-level graph is shown in Figure 14. It contains four nodes and five edges. The underlying instantiated data structures are shown in Figure 15. The light red arcs represent edges; the light green circles represent nodes. The colored boxes on the edges and nodes represent the data structures. The brighter red edges represent the doubly linked incidence lists. Notice that the doubly linked incidence lists' heads and tails are marked by "X" to avoid drawing additional edges. The brighter green arrows represent the source and target nodes, as stored by the edges.

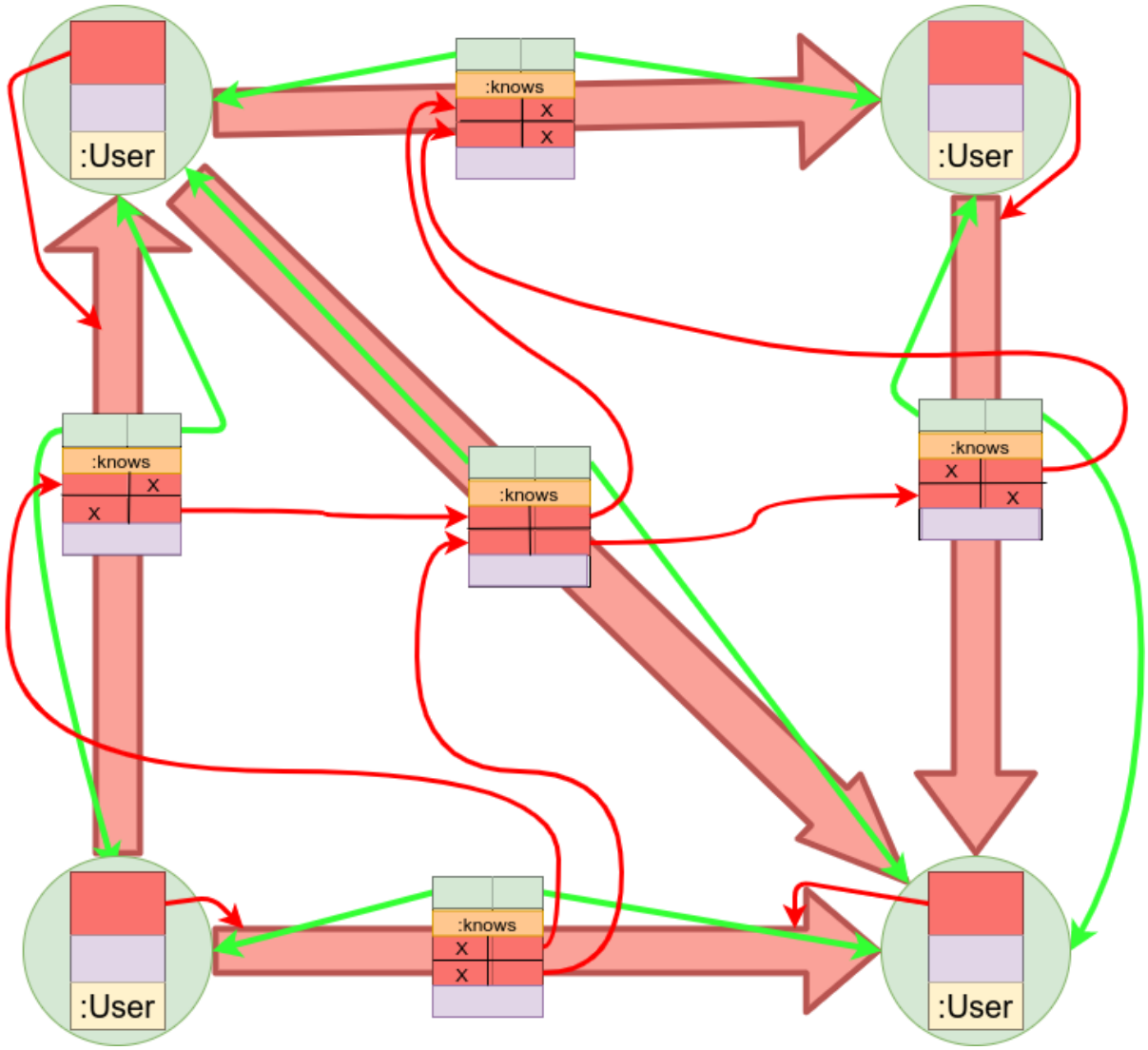


Figure 15. *Visualization of the data structures, as initialized by the example graph shown in Figure 14.*

4. Locality Optimization for traversal-based Queries

4.1. Locality

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

The fundamental principle for the memory hierarchy to work is called *locality of reference* in the literature [54, 96]. This principle expresses that most programs do not access their address space uniformly or randomly but rather tend to access small subsets of all addresses in certain time intervals, depending on the program state. Locality can be approached in two ways [25]:

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

If the same location is accessed multiple times in a short amount of time, the temporal locality is high. Thus temporal locality can be measured using reference frequencies. From a Bayesian point of view, one can say that temporal locality is the probability of an object being re-referenced after the first usage [44].

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

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

If a small range of addresses is accessed very often, then spatial locality is high. If the range is limited to one address, then spatial locality is equivalent to temporal locality. Thus temporal locality is a particular case of temporal locality [44]. With ε a radius we can characterize spatial locality by:

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

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

Several components profile the memory usage to leverage these concepts. In the memory hierarchy, all on and off-chip caches (i.e., SRAM) are handled by hardware [54].

At the main memory level (DRAM), the operating system manages what is fetched, buffered, and evicted from disk to main memory. The optimal buffering strategy is to load what is needed before its usage and evict the objects whose usage is furthest in the future [96].

The best approximation to the optimal strategy is the least recently used algorithm when it comes to eviction. It aims to keep things in memory that has the highest chance to exhibit temporal locality. The things in memory that have not been referenced for the longest time

have a lower chance to be temporally local in the future [88]. Put differently *caches and buffers exploit temporal locality.*

As this information is not available in general, objects are loaded when referenced, often with additional addresses that are hoped to be needed, too — this is called prefetch or predictive fetching [93, 54]. Prefetch tries to exploit spatial locality. Several components try to exploit this:

- Compiler-generated prefetches: The compiler knows what addresses the program accesses in which sequence and tries to minimize the time spent waiting for IO. We call this relocation instruction scheduling [2]. Other compiler-generated heuristics are applied, e.g., in domain-specific compilers, like in the TVM compiler for neural networks [18].
- The operating system may use specialized data structures and algorithms to estimate if prefetching should be done, based on the previous accesses. An example is the "spatial look-ahead" algorithm by Baier and Sager [54, 10], but there exist many more e.g. [56, 41, 63, 20]. Most of these methods can find correlations between addresses and their neighborhood, file accesses, and pointed-to objects.
- A unique role in the context of prefetches and spatial locality take databases. These can predict content-based correlations, e.g., by knowing what table is queried in the case of relational databases, but can also augment data by using auxiliary data structures like indices. The most remarkable capability in this context is to reorganize data based upon how it is queried. Relational databases store data in tables and often sort these tables based upon either a specific field (like the primary key) or a set of fields. In combination with being able to analyze the query before executing, this sorting allows reordering the memory accesses, such that as many accesses as possible are sequential [82, 89].

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

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

4.2. Problem Definition

The records of the graph need to be grouped and ordered to optimize spatial locality for traversal-based queries. Ultimately disk storage and IO is block-based, and disk access is page-based. That is, the vertices and edges must be grouped into blocks. This aggregation can happen statically or dynamically. We focus here on the static reordering. That is, the underlying data is stored only on request.

Assumptions: In the remainder of this thesis, we are assuming that the graph is represented in the property graph model Section 3.2 and uses incidence lists (see Section 2.2.3) as the storage schema. We are not taking properties, labels, and relationship types into account. We are focusing on spatial locality here; that is, the page replacement algorithm is fixed but arbitrary.

Finally, in the remainder of the thesis, when talking about traversal-based queries, we mean all queries described in Section 2.3, but the random walk.

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

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

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

As partitioning a graph optimally [5], as well as finding an optimal linear arrangement [35] are both NP-complete problems [66], we use the formulation "as high as possible" instead of optimal or maximal.

To measure the spatial locality, we introduce two measures that are used in the evaluation chapter:

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

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

As we change the unit of access from addresses of records to blocks on a disk, the definition of locality needs to be adapted. Temporal locality is now based on blocks instead of addresses.

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

Here we have a block B instead of an address A. If blocks are well-formed, the number of blocks that are accessed should decrease overall. This decrease occurs because these multiple elements that are accessed in a short period are stored in the same block instead of several.
Spatial locality can be reformulated in the same sense:

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

With a well-suited ordering of the blocks, the probability that neighboring blocks are accessed should be increased. Finally, when the incidence list is ordered, the access to blocks should be in a monotone increasing block address sequence. Thus, the access should be sequential, and thus the probability of accessing adjacent blocks increases as opposed to unordered access that jumps back and forth.

4.3. Locality in Vertex, Edge and Incidence List Order

Why are these three criteria necessary? Why are there only two measurements for three criteria? Those questions will be explained in an example. Something that is of importance for the traversal — but not as straightforward to see as anode and edge grouping and order — is the pointers' order in the incidence list, as we are going to see.

The graph used in the below example looks as shown in Figure 16. We use a storage schema that is motivated by the one of Neo4J. Nodes and relationships are stored in separated files,

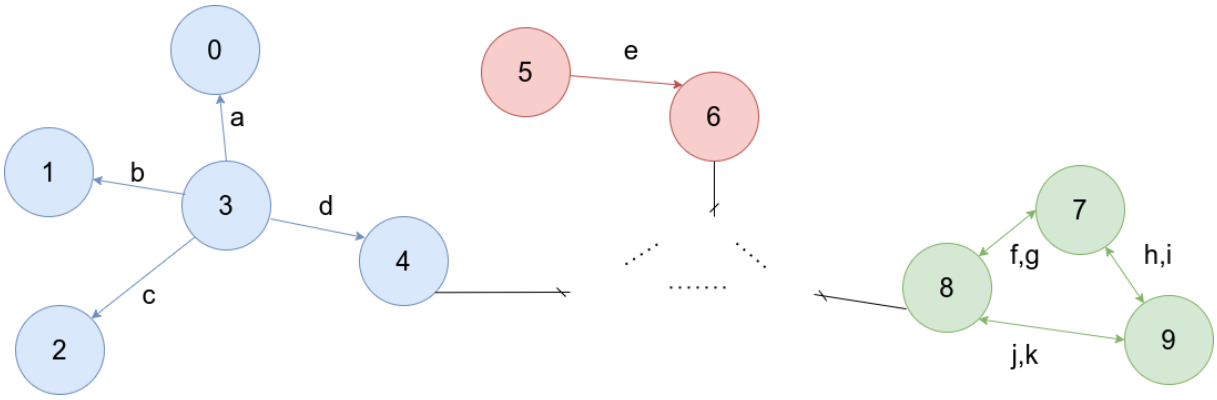


Figure 16. *Parts of a graph that is used in subsequent examples. Cut through edges mean edges to any non-visualized component of the graph. The dotted lines indicate, that other nodes and edges are between the three shown components.*

the incidence list is stored in the edges' records, and the nodes contain a pointer to the head relationship of their incidence list each. Moreover, we assume that we can only read sequentially if the blocks are adjacent. As this is a relatively short example for the sake of succinctness, things are just shown on a conceptual level. We assume that three nodes or two relationships fit onto a disk block. Realistically 8 to 16 nodes and 5 to 8 relationships fit on a 512-byte disk block. An average graph in the Stanford network analysis platform graph dataset collection has thousands of nodes and edges. Taking the californian road network graph as an example, the whole graph would take

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

to store all nodes and

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

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

First, consider the split of vertices and edges into blocks in the upper half of Table 2. None of the vertices in the blocks are neighboring each other. Thus when traversing the graph, each step requires to load a block. The same is true for the edges: None of the edges in the same block are connected to the same vertex. Each edge causes a page fault and a load of another block(s). These additional block IOs may happen in current state-of-the-art graph databases like Neo4J. The placement into blocks is currently by insertion order, thus depending on the input dataset's ordering. In the lower half of Table 2, vertices are grouped into blocks based on their neighborhood, and edges are grouped by the vertices to which they are connected.

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

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

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

Next, Table 3 shows two different orderings of the blocks, this time with a focus on the edges only. An edge is stored in the neighborhood of its source node in the upper table but far apart

from its target node. Thus two single reads are required to go from one vertex over an edge to another vertex and retrieve its incident edges. In the lower table, the blocks are adjacent, and one sequential read is enough to go from source to target and fetch the target's incidence list.

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

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

Table 3. *Suboptimal and improved block order.*

Finally, consider the visualization of the incidence list of node 3, given the page placement in the upper part of Table 2 and Table 4. Theoretically, the only difference is the order in which the list points to the relationships. We need to do four single reads when using the order given in the upper table in terms of traversed blocks. The list may be loaded sequentially with one read operation instead of four single reads if we rearrange the pointers according to the lower table. This phenomenon may also appear when the blocks are formed and ordered in an improved way, but only when scaling up to a larger neighborhood graph. An edge can be placed either near the other edges of the source or the target node. When jumping back and forth, the impact will grow with the gaps between blocks in which those edges are stored. Finally, if the blocks are cached, this induces a higher load on the buffer manager, as more pages need to reside in memory simultaneously and as they are frequently re-referenced instead of being read once and evicted then. The higher the degree — i.e., the length of the incidence list — of the node, the more severe the effect.

incidence list of node 3	c	a	d	b

incidence list for node 3	a	b	c	d

Table 4. *Suboptimal and improved incidence list order.*

5. Locality Optimization of Record Layout

5.1. G-Store: Multilevel Partitioning

G-Store is a disk-based storage manager for graph data implemented and published by Steinhaus et al. [95]. To the best of the author's knowledge, this is the first structured approach to improve locality in graph databases by altering records' placement into blocks. More specifically, to maximize performance, they try to place adjacent nodes close to each other to be read sequentially. The rearrangement of the records is done when importing a new data set and is static after insertion. G-Store uses an adjacency list as the data structure and does not store these in an own file but directly next to the vertex in the same file. The broad schema of the placement method developed by Steinhaus et al. [95] is derived from multilevel partitioning methods described in Section 2.3.3. Briefly, the multilevel partitioning algorithm works in three steps: coarsening, turn-around, and uncoarsening. The coarsening phase tries to reduce the original graph to a smaller one that broadly preserves the underlying graph's structure. A more expensive partitioning algorithm can then be applied to this smaller graph to solve the actual problem approximately and fast. During the uncoarsening, the approximate solution is refined, and the coarser graph is projected back until the original graph is mapped and restored. Finally, in the last step, the partitions are mapped to actual blocks. Note how similar the procedure is to the actual multilevel partitioning algorithm. Thus we are more interested here in the differences from the reference algorithm. A broad overview of the method is shown in Figure 17.

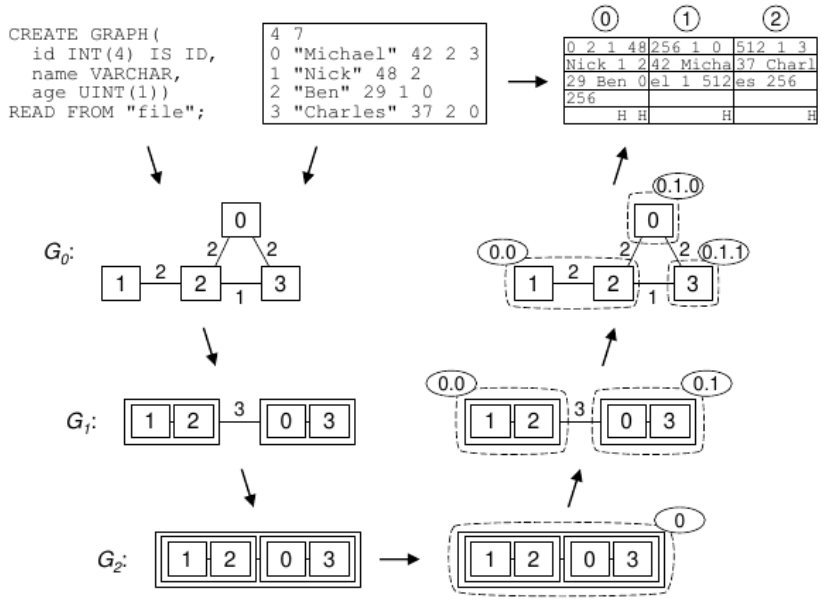


Figure 17. *A broad overview of the multilevel partitioning method applied by G-Store [95].*

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

Coarsening

The coarsening phase of G-Store's partitioning algorithm is called heavy edge matching (HEM) in Karypis formulation of the algorithm. Thus the algorithm takes G_i as an argument and returns G_{i+1} along with a projection Z_{i+1} specifying which finer nodes map to which coarser node. Coarsening proceeds in the original algorithm until a specific lower limit of vertices is reached. In the METIS implementation this is hard-coded to $\max(20k, 40\log_2 k)$ where k is a user defined parameter specifying the number of partitions. The algorithm used in G-Store keeps coarsening until there are no edges, i.e., only one vertex. Thus, it can be seen as a form of hierarchical agglomerative clustering [90] with the inverse edge weight acting as a distance function.

Another modification is that in contrast to METIS [58], not only two edges are matched at a time but possibly many, and there is an upper limit to the vertex weights. This depends on the coarsening factor $c(i) = \frac{|V_i|-|V_{i+1}|}{|V_i|-|\{v \in V | N_v = \emptyset\}|}$, with i the level of coarsening, where $i = 0$ it the original graph. The counter is the number of reduced vertices, and the denominator is the number of nodes in the larger graph minus the irreducible nodes. Initially, the allowed vertex weight θ is the size of a block. If this factor $c(i) < 0.3$ then the node weight θ is doubled as long as $\theta \leq 32$. Otherwise, the number of nodes that are allowed to be matched is incremented.

Turn-Around

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

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

Uncoarsening

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

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

Further, it defines three objective functions:
The first function is closesly related to the minimal linear arrangement problem [66] and expresses that nodes that share an edge shall be minimally far appart from each other.

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

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

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

The last objective function penalizes the number of blocks that are linked. That is to reduce the number of overall linked blocks according to [95].

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

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

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

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

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

Projection

This part of the algorithm constructs a first version of the finer-grained partition numbers ϕ_i from the coarser ones ϕ_{i+1}. The enumeration follows a Dewey numbering scheme [26]. That is, per level, one place in the partition label is added. If the vertex $\phi_{i+1}(Z(v)) = 2$ and vertex $\phi_i(v) = 1$, then the overall partition label of the vertex is 2.1.

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

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

Reordering

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

Refinement

Finally, the refinement step tries to optimize a weighted sum of all the objective functions by moving vertices to other partitions. In the reference algorithm, this step uses the gain as defined in Section A.1.2. Three used-defined parameters α, β, γ control the weighting. Another one specifies the number of iterations that shall be executed r. In each iteration, the algorithm steps over the partitions $\phi_{i,j}$ and creates a two dimensional array A of dimension $|\phi_{i,j}| \times |P_{i,j}|$ with $P_{i,j} = \{\phi_i(u) | v \in \phi_{i,j}, u \in N_n\} \setminus \phi_{i,j}$. Each entry is defined with v the vertex that is to be moved to partition k:

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

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

5.2. ICBL: Diffusion Set-based Clustering

Yaçar and Gedik propose another method to form and order blocks [106, 107].

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

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

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

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

As conductance takes edges between blocks into account and cohesiveness measures the edges within a block, they are complementary [106]. Thus the locality of a block us defined as the geometric mean of the measures above, where the conductance is subtracted from one:

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

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

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

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

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

ICBL is an acronym for the single steps performed by this algorithm. It is designed to be implemented and executed using the map-reduce model. First, ICBL extracts so-called diffusion sets as features. Then it clusters the vertices based upon these features to split the graph into subgraphs. After that, it hierarchically clusters the subgraphs obtained in the previous step to form blocks. Finally, the just-formed blocks are ranked and laid out on disk. A visualization of the method is shown in Figure 18.

Identify Diffusion Sets

The diffusion set $\mathcal{D}_v$ of a vertex $v \in V$ is a characterization of the neighborhood of a vertex. To identify the diffusion set, ICBL carries out t random walks (Section 2.3.1, Algorithm 3) of length l. The authors characterize a random walk using the vertices, so from our definition, we need to extract the multiset of vertices from the walk.

For choosing the parameters t and l Yaçar and Gedik propose heuristics: For t, construct the cumulative degree distribution $f(d) \mapsto P(x \leq d)$ and chose the minimal degree value such that the derivative of the cumulative degree distribution $f'(d) = 1$. The value of the derivative was derived empirically. Regarding l, the authors assume that the network exhibits the small world phenomenon [61]. In a network that has that property, the probability is high that there exists a path between any two nodes $u, v \in V$ of length $\ln |V|$. The random walk's length should not be too long as all nodes might be visitable. Thus the heuristic for choosing l is $1 + \lceil \frac{\ln |V|}{k} \rceil$ where k is the number of clusters in the next step.

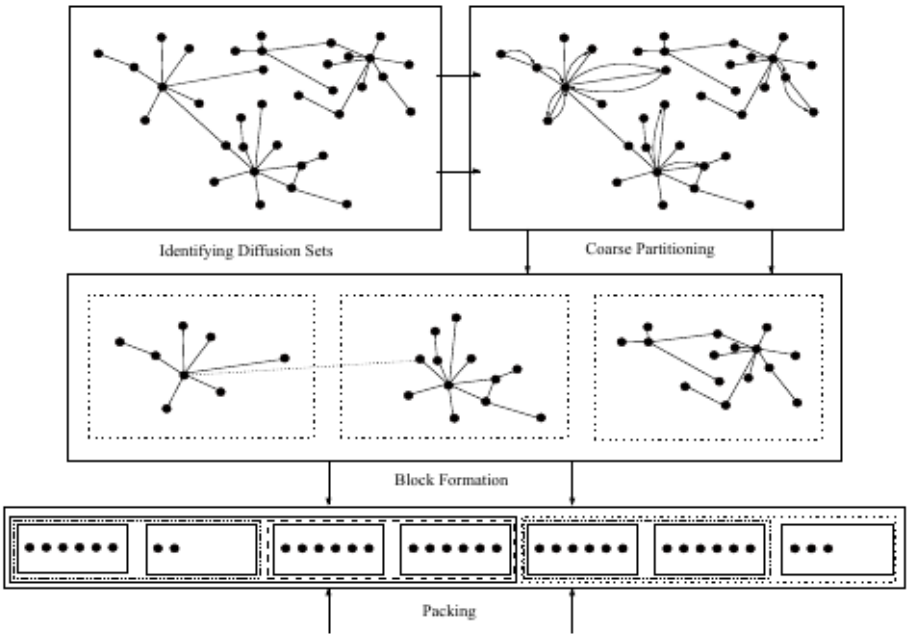


Figure 18. *A broad overview of the ICBL method by Yaçar and Gedik [106].*

Coarse Clustering

After generating the diffusion sets, the graph is clustered using a variation of the k-Means algorithm [67]. It is used to partition the graph into k smaller subgraphs, such that the computationally more expensive agglomerative hierarchical clustering [90] algorithm used in block formation can be executed in parallel. Instead of using a geometric distance function (like the Manhatten or the Euclidean distance between points on a plane), an alternated version of the Jaccard distance function [53] is used. The Jaccard function $J(u,v) = 1 - \frac{\mathcal{D}_u \cap \mathcal{D}_v}{\mathcal{D}_u \cup \mathcal{D}_v}$, i.e. the distance is the number of common elements divided by the set of all elements in both sets. The altered distance is adjusted for multisets:

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

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

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

Block Formation

For each subgraph, agglomerative hierarchical clustering [90] is used to form blocks and label them for the ranking process. Each vertex starts in an own partition. In every step the two closest partitions are merged. The distance function here is the minimum of the previously defined weighted Jaccard distance of all nodes in the partition:

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

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

Layout

Finally, the tree is traversed to extract the formed blocks. These blocks are then sorted according to the label. Next, each subgraph is treated as a vertex. The distance between the two subgraphs is the inverse of the number of edges between them. Those with the lowest distance get merged, and the whole graph is laid out to disk.

Summary

In relational databases, the records are sorted to achieve locality [82, 89]. Block formation is less of an issue there, as the sorting order of the records yields both the formation and the blocks' order. In contrast, graphs need to be partitioned into blocks, and if this is done, the sorting order is far from trivial. Both partitioning and linear arrangement are NP-complete problems [66].

To summarize, previous methods first partitioned the graph by using an adapted version multilevel partitioning algorithm, combining feature extraction with traditional clustering algorithms [55], the Louvain method [14] or the METIS implementation [58] of the multilevel partitioning algorithm [49]. Then based on the partitioning, the blocks were formed and ordered. In G-Store [95] this is done in the uncoarsening phase. In ICBL [107, 106], hierarchical agglomerative clustering combined with a labeling scheme is used. Bondhu [52] uses a scheme where the vertex with the highest partition is placed in the middle, and then iteratively, the neighbors with the highest edge weight are then placed next to it, and the two nodes are merged in the graph.

G-Store uses adjacency lists as a data structure. Thus the edges are placed directly next to the vertices in the very same file. For ICBL, the very same is true. They represent the graph using an adjacency list. In the evaluation part of Yaçar's and Gedik's work, the authors apply their order to Neo4J's incidence list structure [87, 85]. As already discussed in Section 3.3.2, the incidence list is implemented using an edge list with the incidence list included in the edge's record structure. To adapt ICBL's adjacency list to Neo4J, they insert the relations in the order of the adjacency list and store the nodes to their appearance in the edge list. Regarding Bondhu[52], the relationship arrangement is not mentioned in the paper.

5.3. Incidence List Rearrangement

As the just inserted edges contain the incidence lists of the vertices, the order of these changed. If the edges are stored by insertion order, the incidence lists are sorted in a sense: Edges that were inserted earlier also appear earlier in the incidence list and the other way around. Thus, reordering the incidence lists such that the first element in the list is also the first appearing relationship of that list in the underlying file will result in fewer jumps as shown in Table 4.

Consider the case in which we would reorganize the edges' position in the file but not reorder the incidence lists. It is somewhat likely that relationships in the same block and the same incidence list are not accessed sequentially. Instead, many accesses to other elements are made

in between that are potentially widely spread throughout the file. Depending on the node's degree (i.e., the length of the incidence list), the size of the graph, the buffer's capacity, and the number of queries that are concurrently executed, this might cause a lot of additional IOs.

6. Experimental Evaluation

6.1. Setup

Implementation

The implementation is written in C and currently has approximately 15000 lines of code. At the most basic level, it comprises data structures like hash tables, lists, queues, and Fibonacci heaps, but also record structures. The latter are similar to the ones used in Neo4J. The difference to the structures of Neo4J is that properties, labels, and relationship types are currently not supported. Besides that, the data structures are described as in Section 3.3. The database only operates in-memory, implementing an access layer using the `expand` and `get_nodes` operations. Instead of using two files, two hash tables are used to store the records — one for nodes and one for edges.
The access layer atop, all traversal, shortest path algorithms, and the Louvain method is implemented. These algorithms are augmented to log the IDs of accessed nodes and relationships. Further, the static locality optimizing data layout algorithms of G-Store and ICBL are contained, along with the Louvain method and the incidence list's ordering. Additionally, routines for counting the number of accessed blocks based on the logged sequence of nodes and relationships and an importer for datasets from the SNAP dataset collection are implemented.

Cost Model

The cost model used to quantify the locality improvement is based on the sequence of nodes and relationships that are accessed. We assume that the cache or buffer has the size of exactly one block, such that only the accessed block is to be considered "in memory". First, the block number of the accessed record is calculated by

$$\frac{\text{record ID} \cdot \text{record size}}{\text{block size}}$$

As long as consecutive accesses to records produce the same block number, the access is counted as one block IO. Otherwise, — i.e., if the block number changes — it is counted as additional block IO. If the block number change differs only by one block, it is counted as a sequential block IO. When considering the definition of block-based spatial locality

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

we set ε to 8. This would correspond to a system with a block size of 512 Bytes and a page size of 4096. Further, when also using prefetching, even larger values for ε can occur.
The above-described model should correspond closely to how the disk is stored. Besides an offset of one (or multiple) blocks for the maintenance of free blocks and eventually free slots, the block alignment is the same: All IDs in this schema are stored implicitly by the nodes' position in the file. That is — as described in Section 3.3— with a node size of 64 bytes when the node record starts at 192 bytes + header offset, then it has the ID 3.
Finally, we are only going to measure the number of block IOs and the number of non-sequential IOs as mentioned above and in Section 4.2.

Name	Type	Nodes	Edges	Description
C.elegans	Directed	131	764	Frontal part of the neural network of a C. elegans [57]
Email	Directed	1,005	25,571	E-Mail traffic between european research institutions [64]
DBLP	Undirected	317,080	1,049,866	Computer science citation network [105].
Amazon	Undirected	334,863	925,872	Co-purchased items network crawled from Amazon [105].
YouTube	Undirected	1,134,890	2,987,624	YouTube social network [73].

Table 5. *Details on the datasets that are used during evaluation.*

Data Sets and Queries

We use datasets from the Stanford Network Analysis Project [65]. More specifically we use datasets starting from 131 nodes and 764 edges, ranging to 1,134,890 nodes and 2,987,624 edges. All datasets that are used so far are unweighted, i.e. all edges have a weight of 1. A summary of the considered networks is shown in Table 5. The queries executed to gather the access sequences are breadth first searchm depth first search, Dijkstra's algorithm, the A* algorithm and the ALT algorithm. The A* algorithm is supplied a perfect heuristic, that is, the Dijkstra algorithm's output starting from the target node. ALT is executed using five landmarks. As described above, the accessed nodes and relationship IDs are logged by the implementation of these algorithms.

Environment

The evaluation is executed on a 2015 MacBook Pro with an Intel Core i5-4278U processor running at 2.6 GHz, with a boost of 3.1GHz. The bus width is 64 bits. It was manufactured using a 22nm process, has two cores with two threads each, has 128 KiB L1 with 64 KiB for instructions and data each, 512KiB L2 and 3MiB L3 Cache. The main memory consists of $2 \cdot 4$GiB SODIMM DDR3 RAM clocked at 1600 MHz by Micron Technology. We are not using the disk directly, but it may be indirectly used by the operating system when paging or swapping is exchanging pages, or process images. The disk is a 256GB SSD produced by Samsung and uses the SATA protocol over the PCIe 3.0 x4 interface. A combination of 8 GiB RAM and a 10 GiB swap file was insufficient to support larger datasets for G-Store's and the ICBL record layout algorithms.

6.2. Results

For each of the datasets, we applied five different layouts. First, the layout was left as is, governed mainly by the input dataset's order. Second, we randomized the order. Third, the Louvain method is applied to partition the graph. The vertex records are then sorted based upon the partition stably. This means that if two records have the same partition number, the previous order is preserved. Fourth, the algorithm developed for G-Store is applied. Finally, ICBL orders the records.

The IDs of the vertex records were reassigned correspondingly and mapped to blocks using the

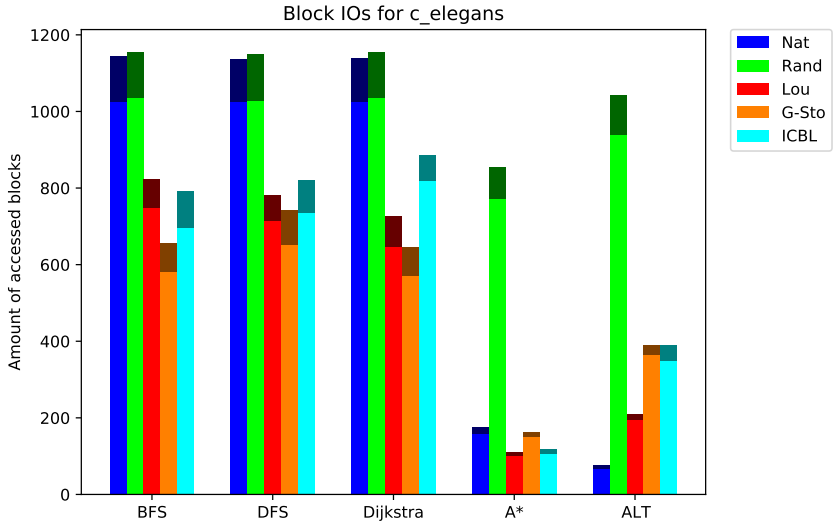


Figure 19. *Comparison of the number of block accesses using the layouts produced by mentioned methods when executing the algorithms.*

above orders. Relationships were then remapped, matching the vertex order based on outgoing connections.
Consider the results shown in Figure 19. In this figure, we used the C. elegans dataset. We can see that the natural and the random layout imply additional block IOs by a factor of approximately 1.5 for traversal-based algorithms that visit all nodes. For the algorithms that traverse only a fraction of the nodes, the results are not so precise. For A*, the partition-based layouts are all slightly better, while for ALT, they are worse up to a factor of approximately 6.

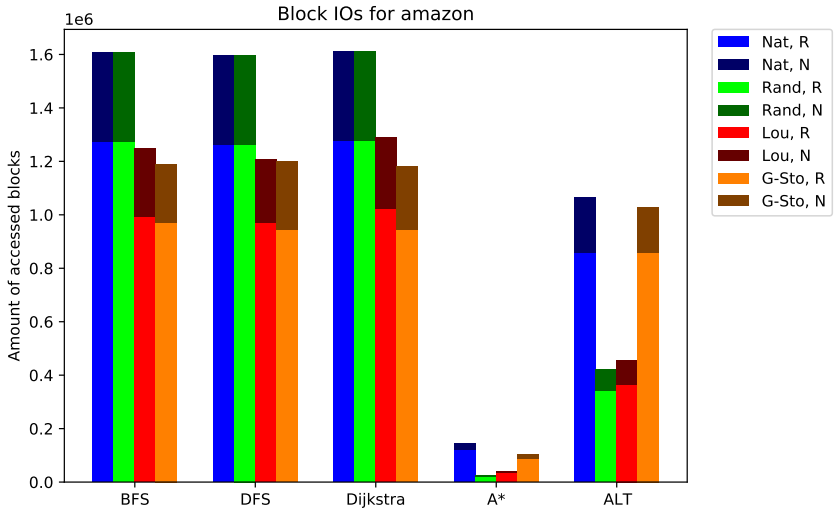


Figure 20. *Comparison of the number of block accesses using the layouts produced by mentioned methods when executing the algorithms on the Amazon dataset.*

We have seen how the results look on the smallest dataset. Now let us consider the largest dataset that was used. In Figure 22, ICBL is not shown. ICBL relies on an adapted version of K-Means clustering, where the centers are chosen by the maximum of the aggregated diffusions set of the cluster. This leads to the behavior that hub vertices are the ones that are chosen most often. In this situation, the following assignment of the vertices assigns most of the nodes to one

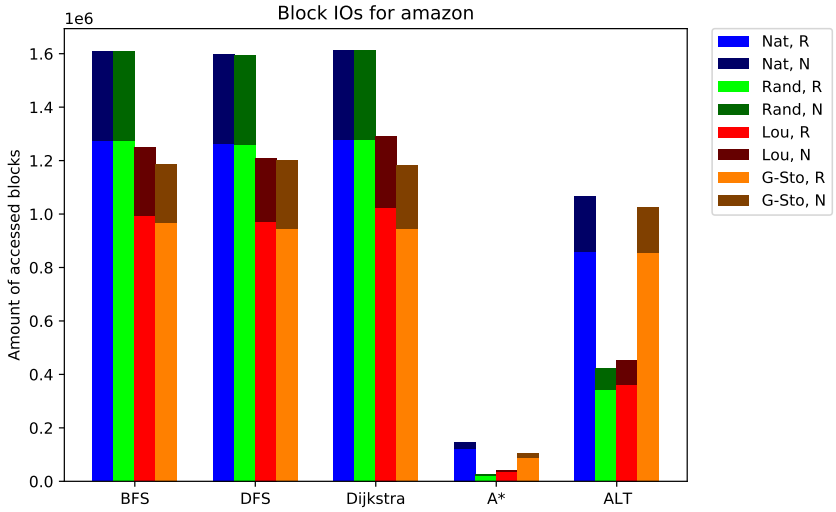


Figure 21. *Comparison of the number of block accesses using the layouts produced by mentioned methods when executing the algorithms on the Amazon dataset.*

cluster. When the initial centers are peripheral, a few clusters get vast, and many are relatively small. In the worst-case, most of the nodes end up in one cluster. The worst-case happened quite often in praxis. The next step is hierarchical clustering, with the distance function being the minimal distance between nodes in the respective subcluster. In other words, one either recomputes the distance for each merge for all vertices in both clusters pairwise, or one stores the pairwise distance matrix. This matrix has the size $|V|^2 \cdot 4$. For 300000, this would result in an allocation of 335 GiB, which is unfeasible on most machines. The authors of this method did not describe how to deal with this scenario or make sure the clusters stay balanced in size during coarse clustering. It is thus excluded from datasets that would result in allocation errors.

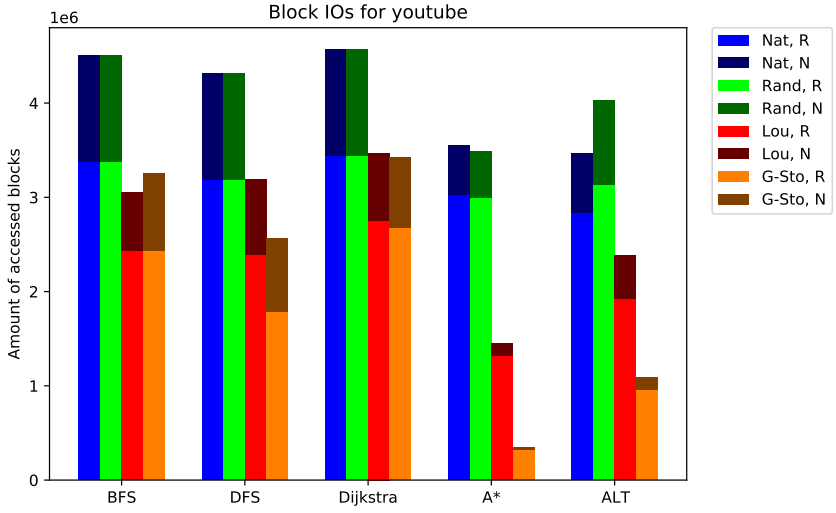


Figure 22. *Comparison of the number of block accesses using the layouts produced by mentioned methods when executing the algorithms on the YouTube dataset.*

As the datasets grow in size, the partition-based layouts tend to outperform the random and natural layout on all queries, as shown in Figure 22, which visualized the amount of block IOs on the YouTube dataset.

Regarding spatial locality, Figure 20 shows the block-based IOs for the amazon dataset. Fig-

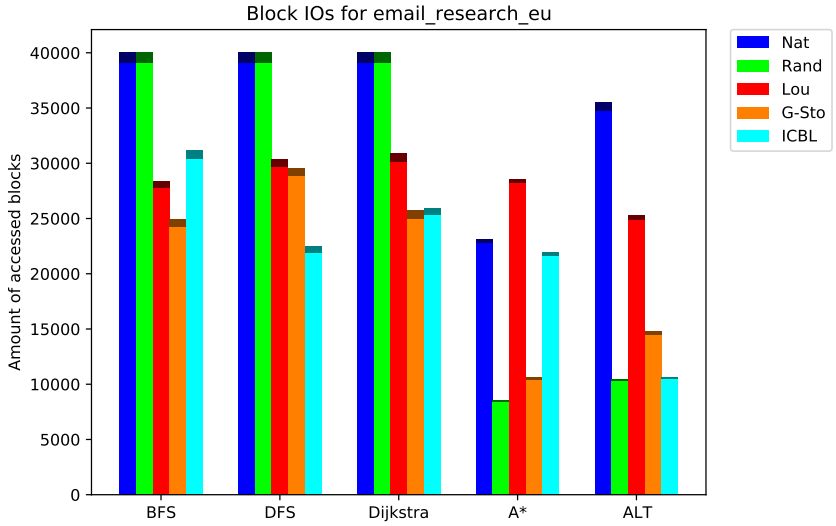


Figure 23. *Comparison of the number of block accesses using the layouts produced by mentioned methods when executing the algorithms on the Email dataset.*

ure 21 shows the page-based IOs for the same dataset. When access is block-based, G-Store performs approximately equivalent to the natural layout, while on page-based access, it uses only half of the accesses. During no experiment, the total theoretical gain of 8 was achieved between block- and page-oriented IO.

The Louvain method performs overall also better than the natural and random layouts. However, it performs most times worse than G-Store's layout method, as it has no means to order partitions. Also, it is in contrast to the other methods per se non-hierarchical concerning specific block size and its output. In Figure 23 we can see that it performs by far worst in comparison with the other partition-based methods. When applying the A* algorithm is even worse than the natural layout.

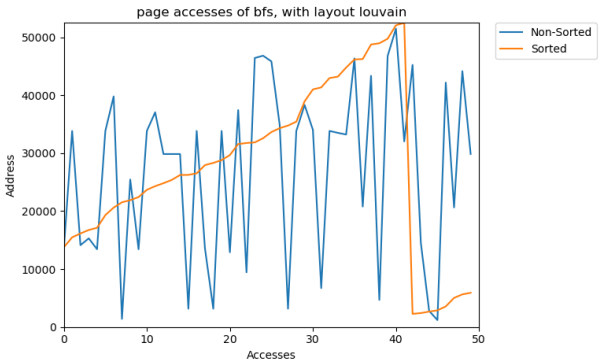


Figure 24. *A sequence of 50 page access with sorted incidence list when traversing the YouTube social network graph laid out using the Louvain method breadth first*

Finally, sorting the incidence lists has an impact on the sequence in which blocks are accessed. In the following figures, 50 consecutive accesses are visualized. These are taken randomly from all accesses (visualizing more or even all accesses leads to a fully colored square). However, the accesses are not synchronized in the sense that they are answering the same request. They only show access x to access $x + 50$. Using sorted incidence lists leads to fewer accesses (see, for example, Table 15 and Table 14). Figure 24 shows how many of the 50 pages across a range of

approximately 35000 are accessed in a perfect monotone increasing sub-sequence. This access sequence is generated using the YouTube dataset, the BFS traversal algorithm, and the Louvain method-based record layout. Accesses without a sorted incidence list are shown in blue.

Similar sequentially accesses are shown in Figure 25 for the DBLP dataset laid out using G-Store's method and the ALT algorithm as query. This time it spans approximately 130000 blocks and needs 3 monotone increasing subsequences.

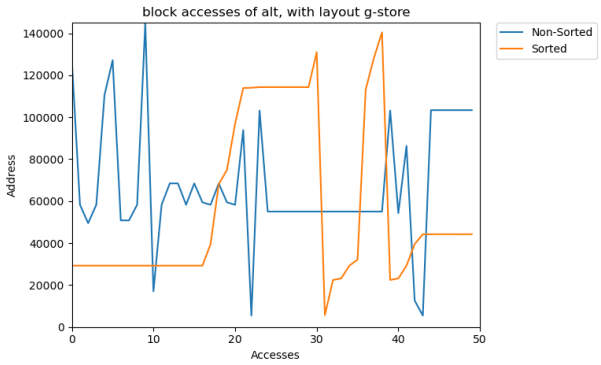


Figure 25. *A sequence of 50 page access with sorted incidence list when executing the ALT algorithm on the DBLP citation network graph laid out using G-Store's method.*

However this is not always the case. In Figure 26 we can see that with both sorted and unsorted incidence lists, the accesses jump quite a lot when the natural layout is preserved.

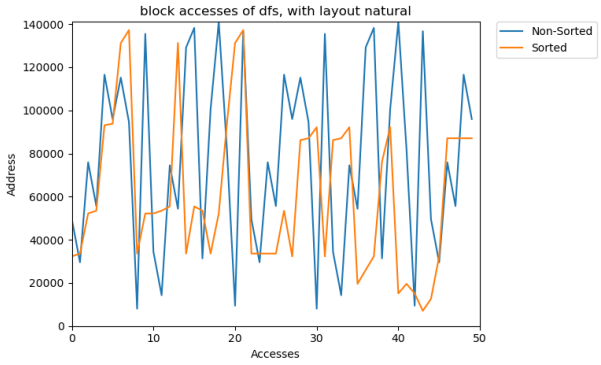


Figure 26. *A sequence of 50 page access with sorted incidence list when executing a depth-first traversal on the DBLP citation network graph with preserved natural layout.*

The key points that the results above show can be summarized as follows.

- Static rearrangement methods indeed decrease the overall number of block accesses and thus increase the temporal locality of all blocks and spatial for pages.
- Sorting the incidence lists leads to often sequential access sequences.
- ICBL's coarse clustering is crucial for larger datasets but is probably not described fully in the papers. If it fails to balance the size of the cluster, systems will run out of memory.
- The Louvain method sometimes performs well, sometimes not. This qualitative degradation is due to not ordering the blocks within a partition and the non-hierarchical approach.

7. Conclusion

7.1. Summary

In this thesis, we examined different methods to reorganize the records of a graph database. This was done to improve the referenced addresses' spatial locality and the temporal and spatial locality of the disk blocks. Then the principle of locality and the problem were defined. After that, we surveyed existing approaches to the problem and derived an improvement over the state-of-the-art methods for static graph data rearrangement to minimize block accesses by maximizing locality. An essential aspect of this was to reorder the incidence list when reordering data to avoid random access patterns that could easily be avoided. Using the implementation of an in-memory database, where the record structures are motivated by Neo4J, we evaluated the methods on a broad range of different datasets with different sizes. The total number of block accesses and non-sequential accesses were used as a metric that measures closely how much disk accesses are needed and if these accesses can be mitigated using prefetching techniques. We saw that the records' static partition-based reordering leads to fewer disk accesses in the evaluation section. The sorting of the incidence list leads to fewer disk accesses when looking at the relationship records.

7.2. Future Work

In this thesis, we restricted ourselves to static rearrangements. However, as the queries change, the access patterns do too. From that, the need to rearrange the data dynamically emerges. Additionally, the results have shown that a static layout can be beneficial for one query while performing poorly for another query. Static record layout methods as proposed in Chapter 5 are restricted to a specific kind of data, while dynamic reordering may be applied independently of the data model, purely driven by the queries and their access patterns. As the results show, a layout that works well for one query may not work for another.
Also, it is to be examined how far the Leiden algorithm produces superior results in contrast with the Louvain method. A runtime comparison between all described methods may not only shed light on the difference between these two algorithms but about the scalability and feasibility of all methods in general.
Furthermore, it would be interesting to determine the locality's degradation when removing or adding nodes and edges for all algorithms as it was done by Hoque and Gupta [52].
Finally, after implementing a disk-based storage mechanism and buffering, an exciting fact to determine would be the total runtime of the traversal-based queries and the impact of the reorganization on the number and rate of page faults.

Bibliography

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

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

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

[4] John R Anderson. "A spreading activation theory of memory". In: *Journal of verbal learning and verbal behavior* 22.3 (1983), pp. 261–295.

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

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

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

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

[9] Seung-Hee Bae et al. "Scalable and Efficient Flow-Based Community Detection for Large-Scale Graph Analysis". In: *ACM Trans. Knowl. Discov. Data* 11.3 (Mar. 2017). ISSN: 1556-4681. DOI: `10.1145/2992785`. URL: `https://doi.org/10.1145/2992785`.

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

[11] Reet Barik et al. "Vertex Reordering for Real-World Graphs and Applications: An Empirical Evaluation". In: *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2020, pp. 240–251.

[12] Hannah Bast et al. "Route planning in transportation networks". In: *Algorithm engineering*. Springer, 2016, pp. 19–80.

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

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

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

[16] 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.

[17] Sheryl L Chang et al. "Modelling transmission and control of the COVID-19 pandemic in Australia". In: *Nature communications* 11.1 (2020), pp. 1–13.

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

[19] 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.

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

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

[22] E. Cuthill and J. McKee. “Reducing the bandwidth of sparse symmetric matrices”. In: *ACM '69*. 1969.

[23] Fatimah S Dawood et al. “Estimated global mortality associated with the first 12 months of 2009 pandemic influenza A H1N1 virus circulation: a modelling study”. In: *The Lancet infectious diseases* 12.9 (2012), pp. 687–695.

[24] Peter Samuel Dayan. “Reinforcing connectionism: Learning the statistical way”. PhD thesis. University of Edinburgh Edinburgh, UK, 1991.

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

[26] Melvil Dewey. *Decimal Classification and Relativ Index for Libraries: Clippings, Notes, Etc*. Library bureau, 1894.

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

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

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

[30] 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.

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

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

[33] Santo Fortunato and Marc Barthelemy. “Resolution limit in community detection”. In: *Proceedings of the national academy of sciences* 104.1 (2007), pp. 36–41.

[34] 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.

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

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

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

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

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

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

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

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

[43] Andrey Gubichev. “Query Processing and Optimization in Graph Databases”. In: 2015.

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

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

[46] 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.

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

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

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

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

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

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

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

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

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

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

[57] Marcus Kaiser and Claus C Hilgetag. “Nonoptimal component placement, but short processing paths, due to long-distance projections in neural systems”. In: *PLoS Comput Biol* 2.7 (2006), e95.

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

[59] William Ogilvy Kermack and Anderson G McKendrick. “A contribution to the mathematical theory of epidemics”. In: *Proceedings of the royal society of london. Series A, Containing papers of a mathematical and physical character* 115.772 (1927), pp. 700–721.

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

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

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

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

[64] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. “Graph evolution: Densification and shrinking diameters”. In: *ACM transactions on Knowledge Discovery from Data (TKDD)* 1.1 (2007), 2–es.

[65] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

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

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

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

[69] 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.

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

[71] *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).

[72] Marvin Minsky. “Semantic information processing”. In: (1982).

[73] Alan Mislove et al. “Measurement and Analysis of Online Social Networks”. In: *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC’07)*. San Diego, CA, Oct. 2007.

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

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

[76] Naoto Ozaki, Hiroshi Tezuka, and Mary Inaba. “A simple acceleration method for the Louvain algorithm”. In: *International Journal of Computer and Electrical Engineering* 8.3 (2016), p. 207.

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

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

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

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

[81] Renfrey Burnard Potts. “Some generalized order-disorder transformations”. In: *Mathematical proceedings of the cambridge philosophical society*. Vol. 48. 1. Cambridge University Press. 1952, pp. 106–109.

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

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

[84] Yakov Rekhter, Tony Li, Susan Hares, et al. *A border gateway protocol 4 (BGP-4)*. 1994.

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

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

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

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

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

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

[91] Gianpaolo Spadini, Ilya Karpov, and David Kencke. “Future high density memories for computing applications: Device behavior and modeling challenges”. In: Oct. 2010, pp. 223–226. DOI: 10.1109/SISPAD.2010.5604521.

[92] Devi Sridhar and Maimuna S Majumder. *Modelling the pandemic*. 2020.

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

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

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

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

[97] V. A. Traag. "Faster unfolding of communities: Speeding up the Louvain algorithm". In: *Phys. Rev. E* 92 (3 Sept. 2015), p. 032801. DOI: 10.1103/PhysRevE.92.032801. URL: https://link.aps.org/doi/10.1103/PhysRevE.92.032801.

[98] Vincent A Traag, Paul Van Dooren, and Yurii Nesterov. "Narrow scope for resolution-limit-free community detection". In: *Physical Review E* 84.1 (2011), p. 016114.

[99] 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.

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

[101] Ludo Waltman and Nees Jan Van Eck. "A smart local moving algorithm for large-scale modularity-based community detection". In: *The European physical journal B* 86.11 (2013), pp. 1–14.

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

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

[104] Rui Xu and Donald C Wunsch. "Survey of clustering algorithms". In: (2005).

[105] Jaewon Yang and Jure Leskovec. "Defining and evaluating network communities based on ground-truth". In: *Knowledge and Information Systems* 42.1 (2015), pp. 181–213.

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

[107] 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.

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

A. Appendix

A.1. Graphs

A.1.1. Matrix-based Graph Representations

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 $A \in |V| \times |V|,\ u, v \in \{0, \ldots, |V| - 1\}$ and $w_{u,v}$ the weight of the edge $e = (u, v) \in E$ then

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

Additionally, 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 cannot represent multi-graphs without further modification. The space complexity of an adjacency matrix is thus $\mathcal{O}(|V|^2 + |V|)$. The

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

number of nodes can be retrieved in $\mathcal{O}(1)$, as it is simply the size of the stored mapping. For the number of edges, one needs to iterate over all matrix elements 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 adding one row and one column to the matrix and one entry to the mapping. This includes reallocating the matrix, which is non-deterministic and independent of the matrix size. However, 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 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 and 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 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,

$\mathcal{O}(1)$ runtime. Finally, finding the neighborhood N_v of a vertex requires a scan of a row and a column again, i.e., an asymptotic runtime of $\mathcal{O}(2|V|)$. For the incoming and outgoing sets of a vertex, one needs to access only a row or a column resulting in $\mathcal{O}(|V|)$ steps per operation. An example of this data structure is shown in Listing 27.

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

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

As with adjacency matrices, 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 $\mathcal{O}(|V| \cdot |E| + |V|) = \mathcal{O}(|V| \cdot |E|)$.

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

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

Figure 28. *An example of the incidence matrix representation of a graph.*

The number of nodes can be retrieved in $\mathcal{O}(1)$, as it is simply the size of the stored mapping. The number of edges can also be retrieved in $\mathcal{O}(1)$ as it is the second dimension of the matrix. Finding a vertex is just an array lookup, thus $\mathcal{O}(1)$. Insertion requires adding one row and one column to the matrix and 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. To find an edge, one needs to scan one row of either the source or the edge's target node, 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 neighborhood N_v of a vertex requires a scan of a row again and checking all non-zero entry columns for the neighbor, 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 neighbors shall be returned – have to be checked. An example of this data structure is shown in Listing 28.

A.1.2. Additional Graph Partitioning Algorithms

Kernighan-Lin Algorithm

The Kernighan-Lin (KL) algorithm [60] was invented by Brain Kernighan — one of the creators of Unix and co-author of the de facto standard book "The C Programming Language" and Shen Lin. It was developed and is used for laying out digital circuits on a chip. It is also used by many other more complex graph partitioning algorithms as the multilevel partitioning algorithm implementation of Karypis and Kumar [58] described in the next part.

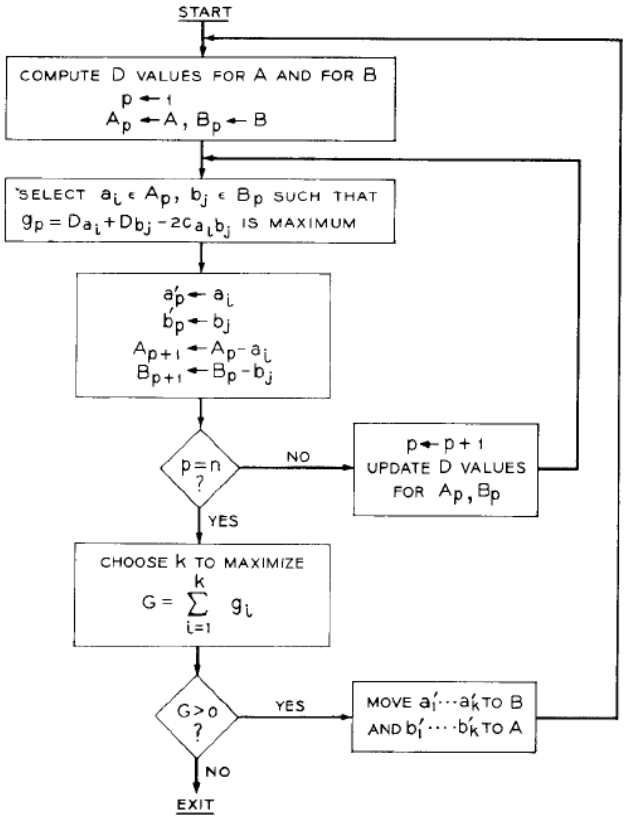


Figure 29. *A flow diagram of the KL algorithm [60].*

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

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

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

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

The internal cost is thus the sum of all edge weights within a partition incident to a specific vertex, while the external cost is the sum of all edge weights to vertices in the other partition for

a specific vertex. The overall cost function is just the difference between external and internal cost. The gain of exchanging two vertices between partitions is proven [60] to be — with $v \in V_i, s \in V_j, i \neq j$

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

The most basic form works as follows.

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

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

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

5. Go to 3 until the partitions are empty.
6. choose k to maximize

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

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

The steps are summarized in the flow diagram in Figure 29. Regarding the extensions, an improvement schema is provided to avoid local optima. Apply the algorithm to the so-created partitions and union the partitions alternatingly (i.e., $A_1 \cup B_2, A_2 \cup B_1$) and use these partitions as initialization of the algorithm. To create partitions of unequal size with n_1 being the minimal desired partition size and n_2 the maximal desired partition size, add dummy vertices, such that we have $2n_2$ vertices in total and restrict the number of changes that are allowed to n_1. For partitioning the graph into k-partitions, one needs to split the initial set of vertices into k partitions. Then apply the 2-way partitioning algorithm pairwise to all partitions until convergence.

One pass of the basic form of the algorithm is in $\mathcal{O}(|V|^2 \cdot \log(|V|))$. The k-way partitioning algorithm requires per iteration $\binom{k}{2} = \frac{k(k-1)}{2}$ executions of the algorithm, thus $\mathcal{O}(k^2 \cdot |V|^2 \cdot \log(|V|)$ in total, assuming that the number of iterations is small as shown empirically by the authors [60]. Further improvements were developed in the years after publication, for example, a linear-time implementation applicable to hypergraphs by Fiduccia and Mattheyses [31].

Leiden Method

The Leiden method [99] is an extension of the louvain method. It changes and extends the louvain method in three aspects.

1. It changes the quality function introducing an additional resolution parameter.
2. It incorporates improvements by other authors on how nodes are moved between partitions [101, 76, 9, 97].
3. It adds a refinement stage after the contraction of the graph has converged.

The quality function that is used by the Leiden method overcomes the problem of the resolution limit [98, 33]. It is called the Constant Potts Model [81, 98] and defined by

$$CPM = \sum_{u,v \in V} (w_{(u,v)} + \gamma)\delta(u,v)$$

where G is a graph, P is a partitioning of G, $w_{(u,v)}$ is the edge weight between u and v, and γ is the resolution parameter. Intuitively it controls the coarseness of the partition and replaces the term $\frac{w_u w_v}{2m}$. A partition is formed, if its density is at least γ and the density between partitions should be less than γ [99].

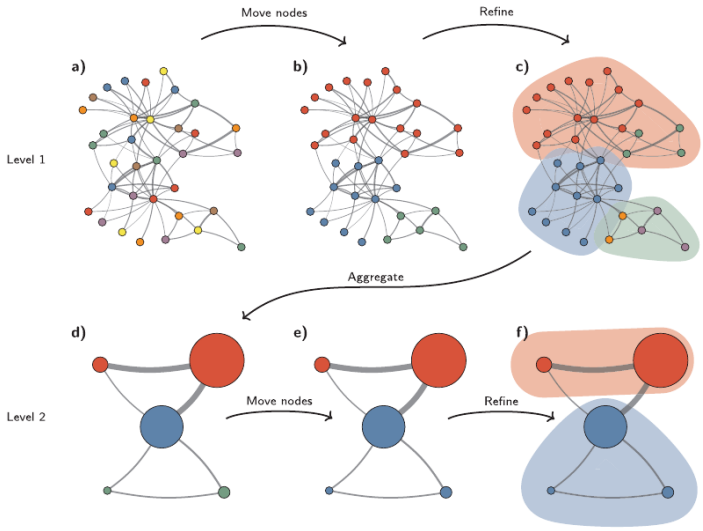


Figure 30. *The steps that the leiden algorithm executes [99].*

The node movement is changed in a couple of ways. The Leiden algorithm keeps track of which neighborhoods changed and only visits these nodes again, in contrast to Louvain, which keeps iterating all nodes[9, 76]. Initially, it adds all nodes to a queue, removes the first one, and adds the node's neighborhood to the queue only if it was moved.

The refinement stage is added after moving nodes between communities and before aggregating the communities into a new graph. Starting with the partitioning P, the algorithm derives a refined partitioning P_{ref}. First, all nodes are again initialized as an own singleton community. Now nodes which are in their own community C^i_{ref} can again be merged with into another community C^j_{ref}, but this time only with those from the same community in the previous partitioning $C \in P$. In essence, it forms from a given partitioning a finer-grained partitioning within the communities. Additionally, not the merge with the highest increase in the quality function is chosen, but the choice is made randomly with a higher probability for better quality values to avoid getting stuck in local optima [97].

Empirically, the Leiden algorithm converges faster with a higher modularity score, with appropriate values for γ. A good value to start is $\gamma = \frac{1}{2}$, which results in an approximation of the Louvain algorithm's quality function — the modularity of Newman and Girvan [36].

A.2. Database Architecture

The reason why we use databases is twofold: First, every computer is equipped with different kinds of memory, which differ in size, capacity, speed, and price per byte. This induces the so-called memory hierarchy, the principle that few fast, expensive, low capacity memory is used close to the central processing unit, which gets layerwise augmented with increasingly slower, less expensive, higher capacity memory. The last layer, which has the highest capacity, defines the overall capacity, while the smallest one is crucial for performance. Thus what is shown as secondary storage in Figure 31 is orders of magnitude slower in both latency and throughput. Nevertheless, it is also able to store orders of magnitude more data. To mitigate

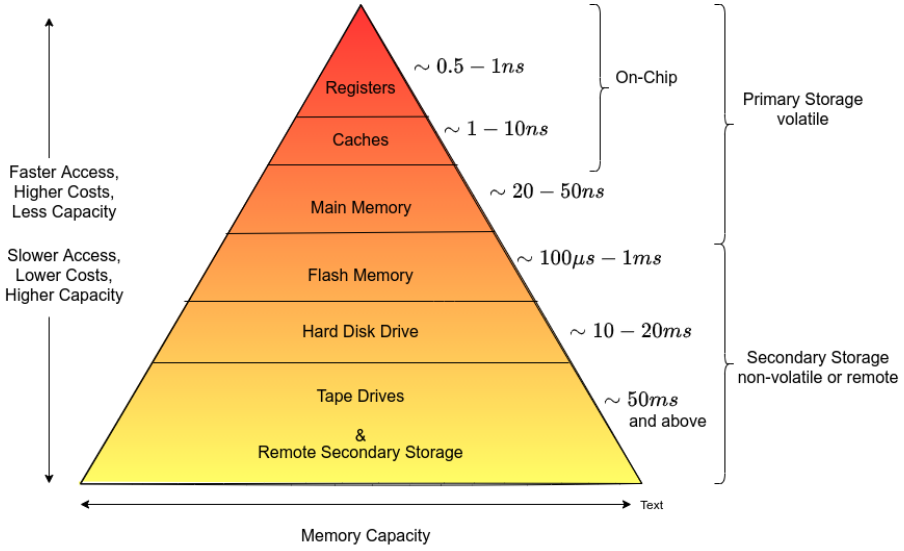


Figure 31. *The memory hierarchy used in today's computing systems.*

the effects of this, the accesses between primary and secondary memory need to be handled very carefully for data-intensive — also called IO bound — applications. Second, the operating system (OS) handles the first reason. However, application-specific payloads enable further optimizations when it comes to how data is stored and accessed. Put differently, the operating system cannot infer certain information, as it does not constrain how data is stored, and as it does not profile in what patterns data is referenced or queried. Databases take care of these issues by different mechanisms, which will be lined out from a high-level perspective to understand how a database works on its architectural lower levels. Put differently, we are not going to discuss query processing, transactions, concurrency-related components, and recovery facilities. Most of the information below is outlined comprehensively in [82, 89]

Let us consider the high-level architecture of a general database management system as shown in Figure 32 — with a focus on the storage and access elements.

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

1. take care of (de-)allocations of disk space,

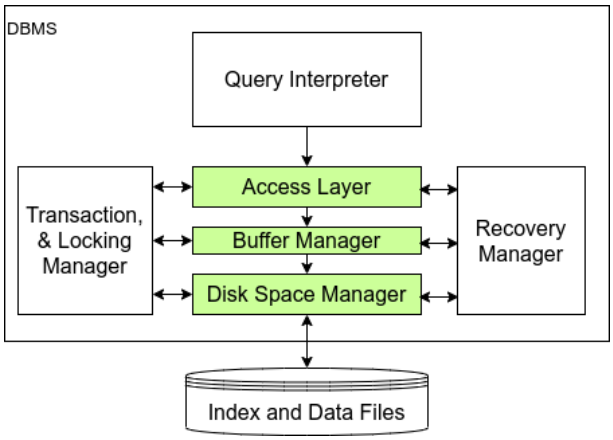


Figure 32. *The typical structure of a relational database management system [82].*

2. abstract storing data on a physical device using the operating system: Files, split into logical disk blocks, accessed using OS facilities, and
3. provide data structures in order to maintain records within a file, blocks.

A buffer manager is used to mediate between external storage and main memory. It provides the concept of a page and maintains a designated pre-allocated area of main memory — called the buffer pool — to load, cache, and evict pages into or from main memory [82]. A conceptual illustration of this is shown in 33. Its 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.

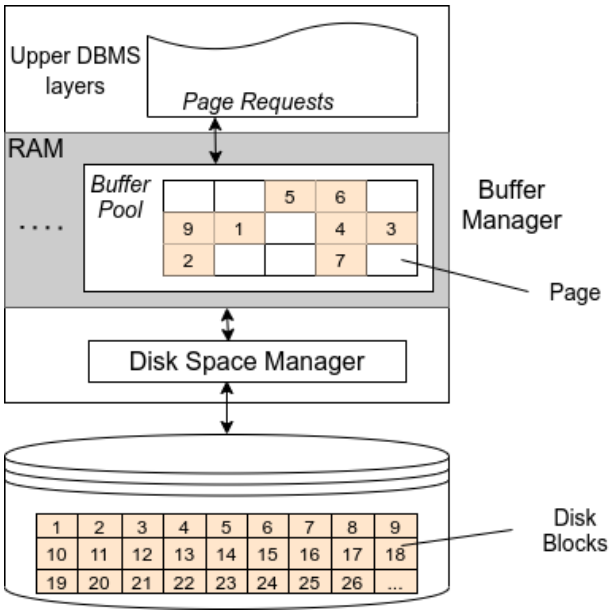


Figure 33. *A visualization of the interaction of a database with memory [82].*

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

A file consists of a set of blocks split into slots. A slot stores one record, with each record containing a set of fields. Records can be layout in row- or column-major order. That is, one can

store sequences of tuples or sequences of fields. The former is beneficial if many updates, insert, or delete operations are committed to the database, while the latter optimizes the performance when scans and aggregations are the most typical queries to the system. Records may be of fixed or of variable size, depending on the types of their fields. Another option is to store the records' structure along with pointers to the values of their fields in one file 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. These approaches have upsides and downsides when it comes to scans, searches, insertions, deletions, and updates.

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

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

A.3. Locality Optimization of Record Layout

Bondhu: An alternating Ordering Scheme

Bondhu [52] is a data layout technique for online social network data. The authors define the cost of a placement similar to the ranking locality above, with r defined as before:

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

Without citing G-Store, Hoque and Gupta propose to use the multilevel partitioning algorithm implemented in METIS [58] to partition the graph. Additionally the authors propose to use the Louvain method [14] to find communities, but they don't provide results on this method. The louvain method is described in more detail in the next section.

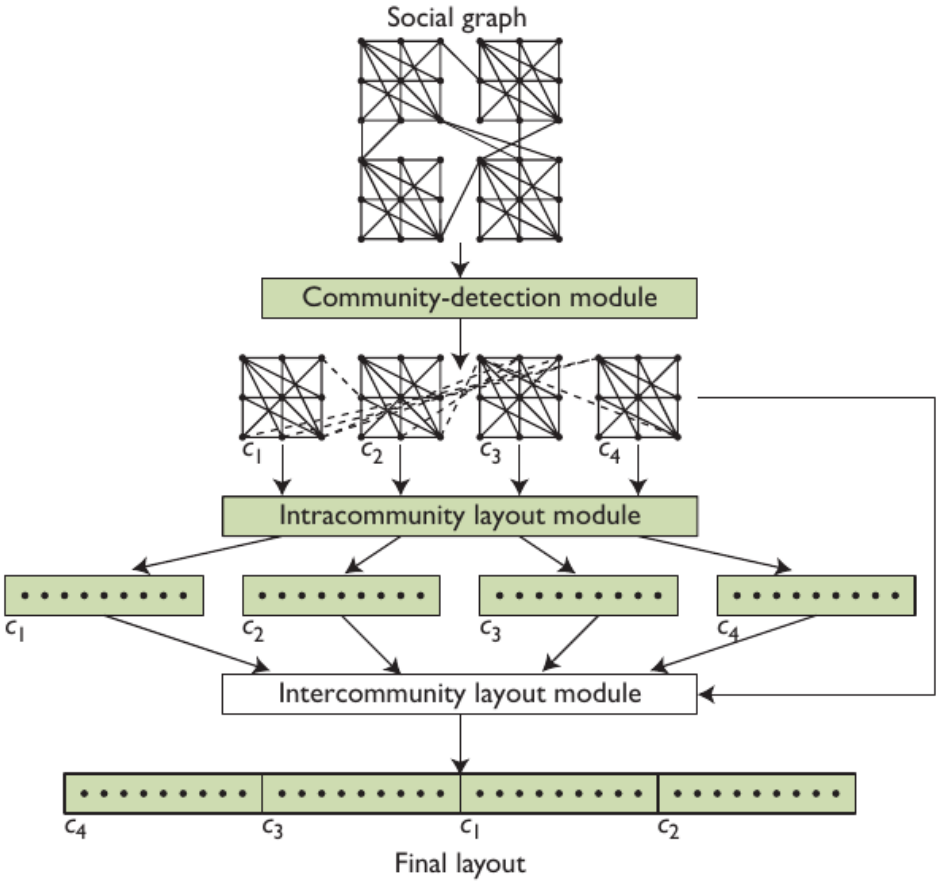


Figure 34. *Broad steps performed by the bondhu algorithm [52].*

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

Louvain-like Formation and RCM-based Ordering

As a first step, a contraction algorithm that optimizes the modularity or the CPM [98, 81] function is executed. During the contraction step of the Louvain or the Leiden algorithm [99], it is easy to stop merging when communities reach a certain number of nodes. That is, as soon as a community has $\frac{\text{Block Size}}{\text{node size}}$ nodes, then do not consider merging nodes into it anymore. This results in block-sized partitions. The aggregation step of these algorithms can then be performed as is. Afterward, executing the algorithm in its standard form yields a hierarchy of graphs, similar to the method employed in G-Store.

This can then be used to achieve an ordering of the blocks: Uncoarsen the graph by applying the Dewey numbering scheme [26], employed by ICBL and G-Store. Additionally, a heuristic for improving the vertex ordering can be applied layerwise. A recent comprehensive survey of minimum linear arrangement approximation algorithms can be found in [11]. An appealing option would be to apply the reverse Cuthill-McKee (RCM) algorithm [22], that approximates the solution in $\mathcal{O}(|V| \cdot \deg V) = \mathcal{O}(|E|)$. It first finds a peripheral vertex. It starts a modified version of the breadth-first search that sorts the nodes by traversal order and degree. The quality of the algorithm is highly dependent on the input graph. As the input graph is the already approximately ordered graph from the last level, input ordering effects should be mitigated. The algorithm compared to other state-of-the-art methods is described by Barik et al. [11].

After the algorithms above are finished, the vertices are laid out in this order to file. The relationships are written to file in the same order by grouping and storing the vertex's outgoing edges and to follow the vertex order. The outgoing edges of the vertex stored at the first slot in the vertex file are stored beginning at the first slot of the edge file. The outgoing edges of the second node start right after the edge group of the first node. This scheme is motivated by the assumption that traversals in directed graphs follow the edge direction.

B. Evaluation Data

C. Elegans Frontal Neural Network Dataset

	BFS	DFS	Dijkstra	A^*	ALT
Natural	118	112	113	16	8
	1025	1026	1026	158	67
Random	120	121	118	81	102
	1036	1028	1035	772	940
Louvain	74	66	79	10	15
	749	714	647	101	196
G-Store	75	89	75	13	24
	581	652	570	149	366
ICBL	96	86	66	12	40
	695	734	819	107	349

Table 6. *Comparison of IOs per method, query and record type on the C. elegans dataset with unsorted incidence lists.*

	BFS	DFS	Dijkstra	A^*	ALT
Natural	114	114	111	11	35
	952	944	952	110	366
Random	122	114	121	71	97
	961	960	957	665	785
Louvain	83	82	69	14	17
	222	389	145	60	99
G-Store	73	86	75	16	30
	178	280	228	77	168
ICBL	84	81	67	12	46
	146	167	355	106	197

Table 7. *Comparison of IOs per method, query and record type on the C. elegans dataset with sorted incidence lists.*

European Research Email Network Dataset

	BFS	DFS	Dijkstra	A^*	ALT
Natural	970 39123	972 39114	972 39123	326 22834	774 34759
Random	973 39105	966 39097	971 39106	99 8389	132 10319
Louvain	605 27778	638 29728	773 30127	321 28255	400 24875
G-Store	685 24234	647 28923	751 25016	206 10397	331 14441
ICBL	771 30453	633 21861	585 25379	347 21617	162 10448

Table 8. *Comparison of IOs per method, query and record type on the email dataset with unsorted incidence lists.*

	BFS	DFS	Dijkstra	A^*	ALT
Natural	952 29223	947 29221	966 29225	316 16340	762 25755
Random	908 29236	949 29231	932 29237	43 4127	72 5432
Louvain	629 5169	592 8945	539 8247	407 8042	399 4978
G-Store	748 8024	650 6454	549 6490	204 4488	111 1010
ICBL	558 4504	733 9007	664 4849	164 6316	268 5776

Table 9. *Comparison of block IOs per method, query and record type on the email dataset with sorted incidence lists.*

DBLP Citation Network Dataset

	BFS	DFS	Dijkstra	A^*	ALT
Natural	317072 1338193	317071 1314632	317071 1344250	88454 598168	239225 1135678
Random	317063 1338218	317064 1314495	317070 1344206	2338 20784	91026 506800
Louvain	209269 1003659	231464 906895	190242 846871	27762 249979	219532 723742
G-Store	250482 842998	187065 972704	215604 752748	16340 156619	97658 409640

Table 10. *Comparison of IOs per method, query and record type on the DBLP dataset with unsorted incidence lists.*

	BFS	DFS	Dijkstra	A^*	ALT
Natural	317072 1335990	317064 1322882	317076 1341846	15934 145842	242959 1141861
Random	317058 1335872	317065 1322426	317062 1341851	2725 23601	88404 487566
Louvain	190245 495939	218779 300461	206096 179801	32042 65190	174006 426641
G-Store	187067 486164	221947 285141	231457 287917	19841 87398	127806 168100

Table 11. *Comparison of IOs per method, query and record type on the DBLP dataset with sorted incidence lists.*

Amazon Frequently bought together Dataset

	BFS	DFS	Dijkstra	A^*	ALT
Natural	334854 1275141	334846 1261562	334854 1278263	23591 121074	206718 859489
Random	334844 1274993	334839 1261169	334845 1278126	4630 21735	81595 340428
Louvain	254490 994592	237746 971234	267880 1022546	7217 33642	90032 363483
G-Store	217652 969107	254487 946004	237742 945848	17030 88412	169893 856992

Table 12. *Comparison of IOs per method, query and record type on the Amazon dataset with unsorted incidence lists.*

	BFS	DFS	Dijkstra	A^*	ALT
Natural	334831 1271978	334831 1253135	334838 1275516	29305 148144	213728 881883
Random	334826 1271472	334833 1253034	334829 1275273	5267 24768	105294 446134
Louvain	210935 531443	194200 184171	237722 327139	7544 16324	105341 97106
G-Store	224334 349005	254463 395952	210939 192827	20196 47784	186835 418902

Table 13. *Comparison of IOs per method, query and record type on the amazon dataset with sorted incidence lists.*

YouTube Social Network Dataset

	BFS	DFS	Dijkstra	A^*	ALT
Natural	1134885 3376451	1134874 3180208	1134887 3434910	520744 3029489	626139 2837347
Random	1134878 3378223	1134879 3180625	1134885 3436253	494930 2995538	900538 3134437
Louvain	624186 2432604	805764 2385390	714975 2748602	130763 1315246	464393 1918687
G-Store	817115 2431757	783068 1780594	749022 2679796	24225 326708	128005 957401

Table 14. *Comparison of IOs per method, query and record type on the C. elegans dataset with unsorted incidence lists.*

	BFS	DFS	Dijkstra	A^*	ALT
Natural	1134879 3369934	1134869 3274601	1134870 3427327	466807 2760899	673661 2877905
Random	1134841 3372967	1134858 3274050	1134858 3428718	545467 3193624	743401 2932793
Louvain	885148 1238579	760368 1003650	828413 809341	129288 308880	439661 733981
G-Store	771704 550333	907899 1055840	646850 958365	21961 147532	112100 326985

Table 15. *Comparison of IOs per method, query and record type on the YouTube dataset with sorted incidence lists.*