FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER’S THESIS

Title:	Query Analysis on a Distributed Graph Database
Student:	Bc. Lucie Svitáková
Supervisor:	Ing. Michal Valenta, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2019/20

Instructions

1. Research current practicies of data storage in distributed graph databases.
2. Choose an existing database system for your implementation.
3. Analyse how to extract necessary information for further query analysis and implement that extraction.
4. Create a suitable method to store the data extracted in step 3.
5. Propose a method for a more efficient redistribution of the data.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 5, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

Query Analysis on a Distributed Graph Database

Bc. et Bc. Lucie Svitáková

Department of Software Engineering
Supervisor: Ing. Michal Valenta, Ph.D.

January 8, 2019

Acknowledgements

I would like to thank my supervisor Michal Valenta for his valuable suggestions and observations, my sister Šárka Svitáková for spelling corrections, my friends Martin Horejš and Tomáš Nováček for their valuable comments and my family for their continual motivation and support.

Declaration

In Prague on January 8, 2019 ....

Citation of this thesis

Svitáková, Lucie. *Query Analysis on a Distributed Graph Database.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Přestože dnes existuje několik produktů grafových databází určených pro distribuovaná prostředí, jsou v nich obvykle data distribuována na jednotlivé fyzické uzly náhodně, bez pozdější revize zatížení a možné reorganizace dat. Tato práce analyzuje současné praktiky ukládání dat v distribuovaných grafových databázích. Podle této analýzy práce navrhuje a implementuje nový modul pro grafový výpočetní framework TinkerPop, který loguje provoz vygenerovaný dotazy uživatelů. Je rovněž naimplementována samostatná aplikace pro ukládání takovýchto logovaných dat do databáze JanusGraph. Tento program rovněž spouští redistribuční algoritmus navrhující efektivnější uložení dat v clusteru. Existující algoritmus od Vaquera a spol., kompatibilní se systémem Pregel, je aplikován s podstatnými rozšířeními. Výsledkem je návrh reorganizace dat s 70–80% zlepšením komunikace mezi uzly clusteru. Takovýto výsledek je porovnatelný s jinou známou metodou Ja-be-Ja, která však vyžaduje výrazně vyšší výpočetní prostředky. Na druhou stranu metoda v této práci zavádí malou disbalanci na uzlech clusteru. Nakonec tato práce uvádí doporučení pro možná budoucí rozšíření a vylepšení.

Klíčová slova grafové databáze, rozdělení grafu, redistribuce, NoSQL, JanusGraph, TinkerPop, Pregel

Abstract

Although products of graph databases intended for distributed environments already exist, the data are usually distributed randomly on particular physical hosts without any subsequent load examination and possible data reorganization. This thesis analyses the current practices of data storage in distributed graph databases. According to this analysis, it designs and implements a new module of the general graph computing framework TinkerPop for logging the traffic generated with user queries. A separate implementation for storage of such logged data in the JanusGraph database is provided. It also executes a redistribution algorithm proposing a more efficient distribution of data. An existing Pregel-compliant algorithm of Vaquero et al. with substantial enhancements is applied. Results with 70–80% improvement of communication among physical hosts of the cluster are obtained, which is comparable to another well-known method Ja-be-Ja with much higher computational demands. On the other hand, the method in this thesis imposes a necessary slight imbalance of the cluster nodes. Finally, this thesis introduces suggestions for future extensions and enhancements.

Keywords graph databases, graph partitioning, redistribution, NoSQL, JanusGraph, TinkerPop, Pregel

Contents

List of Figures

List of Tables

Introduction

In the last decades, modern technologies enabled production of huge amount of data of various types. With these changing demands on the data storage, the database development advanced as well, from punched cards and magnetic tapes to nowadays relational and NoSQL databases. Each database type focuses on a different nature of data. One of such categories are graph-like data containing entities (vertices) which are associated among each other via relationships (edges). In reality, these data can represent, for example, road maps, airline networks, social networks, web graphs or biological networks.

Some of the mentioned areas have experienced a boom in recent years and a subsequent increased interest regarding efficient storage and usage of their data. This is the reason why the graph databases have gained substantial popularity growth recently [1].

The increasing amount of data also places demands on the size of physical storage. The single-server solutions are often no longer sufficient, with the consequence of employing the distributed environments. Several open-source and commercial products offering a distributed graph database already exist. However, this rather new concept usually does not deal with the performance of the final distribution, surely not in the case of open-source solutions. The data are stored on physical nodes randomly by default. The later traffic on the database could, nevertheless, evince some trends for which another distribution of data could be more efficient.

The purpose of this work is to first research current practices of data storage in distributed graph databases. On these grounds, an existing database system is used for further implementation. We analyze how to extract the necessary information for further query analysis and we implement that extraction. Subsequently, we create a suitable method to store these extracted data. Last but not least, we propose a method for a more efficient redistribution of the data and evaluate it.

To achieve this goal, the thesis is structured as follows. The chapter 1 first presents the general theory of graph databases. It is followed by the introduction of the graph partitioning problem and its existing algorithms in chapter 2. The current graph database systems are examined together with their models of data storage in chapter 3. The chapter 4 is devoted to the analysis and design of the data extraction, their storage and redistribution algorithm. The implementation of these concepts is described in chapter 5. The proposed redistribution is evaluated on real datasets and later compared to another algorithm in chapter 6. Finally, the chapter 7 offers several propositions for future work.

Chapter 1

Theory of Graph Databases

The objective of the first chapter is to give a general theoretical overview of graph databases. It begins with a brief historical development of database management systems (DBMS) focusing on relational and NoSQL databases. Then it continues with an introduction to graph databases with emphasis on data models, physical storage possibilities and query languages. Additional technical details about the graph databases are later provided in chapter 3 dedicated to particular products.

1.1 Databases

"A database is a self-describing collection of integrated records" [2]. Records in this definition are any data collected to be stored. The adjective *integrated* means that not only the records themselves but also their relationships are maintained. Last but not least, a collection of such data with relationships is self-describing because it contains a description of its structure.[2]

This definition refers to nowadays databases that emerged with computers. Nevertheless, the history of databases – storing, retrieving, modifying and deleting data – dates back a long time ago. Błażewicz et al. [3] mentions six main phases of database evolution to which we add the last category of *NoSQL databases*[1]:

1. **manual processing**
2. **mechanical and electromechanical equipment** – data stored in binary format on punched cards
3. **magnetic tapes and stored-program computers**
4. **online data processing** – introduced in section 1.1.1.

[1]The referenced source of the categories [3] dates back to 2003 after which the term *NoSQL* has been reintroduced.

5. **relational databases** – introduced in section 1.1.2.
6. **object-oriented databases** – introduced in section 1.1.3.
7. **NoSQL databases** – introduced in section 1.1.4.

1.1.1 Online Data Processing

With the origin of the hierarchical database management system (or IMS – Information Management System), data started to be stored independently on programs that use them. Another type of databases has its origin in this era, 1960's, collectively called CODASYL[2].

This first generation of databases, IMS and CODASYL, is also called *navigational databases*. A navigational database could be imagined as a graph where records have primary keys and can carry a secondary key referencing another record. The data were stored as a linked data list. A programmer was accessing the data via the primary key sequentially or by navigating through the graph [4]. However, even simple queries required complex programs to be written [3].

1.1.2 Relational Databases

Edgar F. Codd released a paper *A Relational Model of Data for Large Shared Data Banks* [5] where he introduced what is referred to as the second generation of DBMSs, relational databases (RDBMS). Although the paper dates back to 1970, even after 49 years the relational databases are still predominant and by far the most popular DBMSs within community [6].

Relational DBMSs are arranged in tables, where rows are representing individual records, and each column determines a type and semantic structure of data. The number of columns in each table for each row is fixed. Each record has a primary key, and relations among records are expressed with foreign keys.

The common practice of a relational DBMS design is following the rules of so-called *normal forms* avoiding data redundancy, actualization anomalies[3] and enabling more effective manipulation with data.

In order to allow to work with the data in relational DBMSs the Structured Query Language (SQL) has been created. It is a language based on relational algebra enabling manipulation and definition of the data or managing access rights and transactions. A couple of other new concepts also emerged with relational DBMSs such as concurrent transaction management, transactional recovery techniques, optimization techniques, and many more.

[2] *CODASYL* is an acronym for Conference on Data System Languages where a number of rules for Database Management Systems (DBMSs) were arranged.

[3] An actualization anomaly is a side effect of a record update resulting in a loss or an inconsistency in data.

The transactions in RDBMSs must fulfill *ACID* properties:

- *Atomicity* – a transaction succeeds as a whole or does not succeed at all
- *Consistency* – a transaction transforms the database from one valid state to another
- *Isolation* – concurrent execution of transactions reaches the same state as sequential launch of these transactions
- *Durability* – changes resulting from successfully committed transactions are persistently stored in the database

1.1.3 Object-Oriented Databases

As an alternative to relational DBMSs, object-oriented databases (OODBMSs) were developed. They brought an option to avoid modeling and data type constraints of relational DBMSs. Using object-oriented programing paradigm, OODBMSs take advantage of inheritance, encapsulation or arbitrary data types. Object-oriented concepts can be also mixed with relational model, resulting in so-called *object-relational DBMS* (ORDBMS). The usage of both OODBMSs and ORDBMSs was, however, partly reduced with object-relation mapping (ORM) offering to programme with objects while taking advantage of relational databases.

1.1.4 NoSQL Databases

NoSQL databases originally denoted databases that do not use SQL. However, nowadays meaning is rather expressed as "not only SQL" databases – that is, DBMSs that can have the support of SQL-like languages. There are several reasons why to use NoSQL databases. Among others, Strauch [7] mentions: avoidance of unneeded complexity, high throughput, horizontal scalability, compromising reliability for better performance, yesterday's vs. today's needs (such as commodity hardware with a high probability of failure, the character of data, etc.).

It's also important to point out the *CAP Theorem* stating that the three concepts that cannot be achieved all together [7]:

- *Consistency* – system is consistent after the execution of an operation.
- *Availability* – system allows to continue an operation even if some hardware/software parts crash or are down.
- *Partition Tolerance* – system can continue an operation even if the network is partitioned into parts that cannot connect to each other.

If the system should be consistent and available, formerly mentioned *ACID* properties are required, whereas if we forfeit consistency, hence favoring availability and partition-tolerance, BASE properties are demanded. NoSQL databases usually adhere to BASE properties which stand for following:

- *Basic Availability* – the system works basically all the time.
- *Soft-State* – the system does not have to be consistent all the time.
- *Eventual consistency* – the system will be in a known-state in some later point.

The NoSQL databases are most often divided into following classes:

- **Key-Value Stores** – data model of key-value stores is an associative array (in programming known as maps or dictionaries). Data are accessed with a key which serves as a primary key. A value associated with the key is a BLOB[4] with any data.

  Examples of key-value stores are Riak, Redis or Oracle Berkeley DB.

- **Document Stores** – document stores resemble key-value stores regarding a key referencing an associated value – in this case a document. However, the document has a defined format which can be further examined.

  Examples of document stores are MongoDB or Apache CouchDB.

- **Wide Column Stores** – also referred to as Column-Oriented Stores or Extensible Record Stores [7]. The data model includes column families, rows and columns. A column family can be imagined as a table of similar rows – each row representing one record but, unlike in relational databases, with a flexible number of columns.

  Examples of wide column stores are Google's Bigtable, Apache Cassandra or Apache HBase.

- **Graph Databases** – are dedicated for data that can be characterized as graphs – records are represented as entities (nodes) and relationships (edges) between them.

  Following sections contain more detail as well as examples of products.

The presented phases of database development do not include an exhaustive list of existing types of databases. Rather, it offers a general overview of data storage possibilities. There haven't been mentioned several other types

[4]BLOB stands for Binary Large Object.

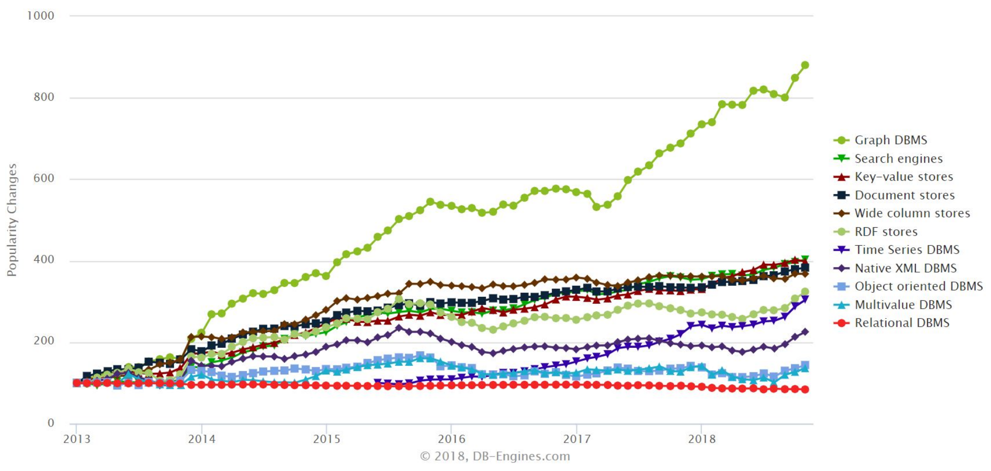


Figure 1.1: DBMS popularity changes since 2013 [1]. Graph DBMSs are represented with the upper green line with circles. Separately from graph DBMSs, RDF stores are depicted with a lighter green line with circles experiencing lower popularity changes than solely the graph DBMSs.

of DBMS such as XML[5] stores, or multi-model databases supporting multiple data models.

1.2 Graph Databases

As already mentioned, graph databases are designed for data forming graphs with nodes (entities) and edges (relationships). There are several types of data with the characteristics of a graph – for example road maps, internet networks or citation dependencies. Graph data have recently experienced a boom with social networks. As a result, over the past five years, graph DBMSs have grown in popularity among other DBMSs the most as shown in Figure 1.1.

1.2.1 Data Model

The data model of graph databases is a graph. Nevertheless, the type of graphs can vary. Therefore, to be more specific, graph databases usually use one of two main graph data models – a property graph and an RDF graph. RDF stores are sometimes categorized as individual databases apart from graph DBMS as is the case of the Figure 1.1; nevertheless, their character is graphical, so it is appropriate to introduce them briefly as well.

[5]XML stands for eXtensible Markup Language defining rules for documents which format is readable for both humans and computers.

- **Property Graph** – is a directed labeled multi-graph. It was developed with the objective of efficient storage and fast querying with fast traversals [9]. It consists of
  - *nodes* – represent individual entities. Each node has a unique identifier.
  - *edges* – represent individual relationships between entities. Because the property graph is a multi-graph, there can be several edges between two nodes. However, an edge always connects exactly two nodes[6] (with the possibility of node equality, that is, an edge creating a loop). Each edge has a unique identifier.
  - *labels* – provide a node tagging – each node can have several labels assigned denoting its classification (such as a person, a teacher, a student, etc.).
  - *types* – provide an edge tagging. However, in contrast to labels, each edge has exactly one type (such as "is managing a", "is the father of", "owns", etc.).
  - *properties* – each node as well as each edge can have a set of properties in the form of key-value pairs (such as "age: 30", "starting-date: 2018-01-01", etc.).
- **RDF Graph** – The acronym *RDF* stands for Resource Description Framework and it is a W3C standard for data exchange on the web [8]. The RDF graph is a graph of triples: *subject – predicate – object* such as "Alice knows Bob" or "Anakin is the father of Luke". In contrast to property graph, elements of an RDF graph (nodes and edges) do not have any internal structure, meaning they do not contain any properties.
  - *subject* – a resource vertex identified with a URI[7] (as the RDF graph is dedicated for data exchange on the web)
  - *predicate* – an edge identified with a URI
  - *object* – a literal value or a vertex identified with a URI

The Figure 1.2 and 1.3 show a representation of the same data example in a property graph and an RDF graph, respectively.

The main differences between property and RDF graphs are [9]:

- RDF graph cannot have multiple identical relationships between two nodes whereas the property graph can.

[6]An edge connecting more than two nodes is not allowed in a multi-graph. A hypergraph, on the other hand, allows such an eventuality.

[7]URI stands for Uniform Resource Identifier.

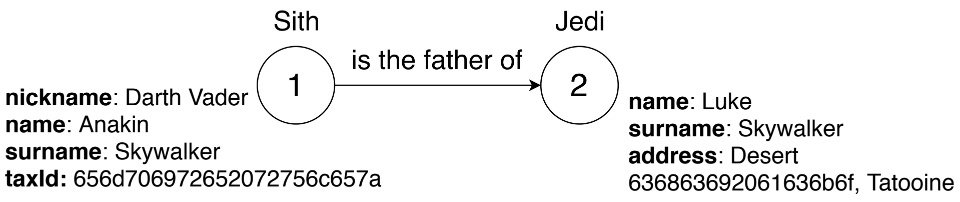


Figure 1.2: Example of a property graph

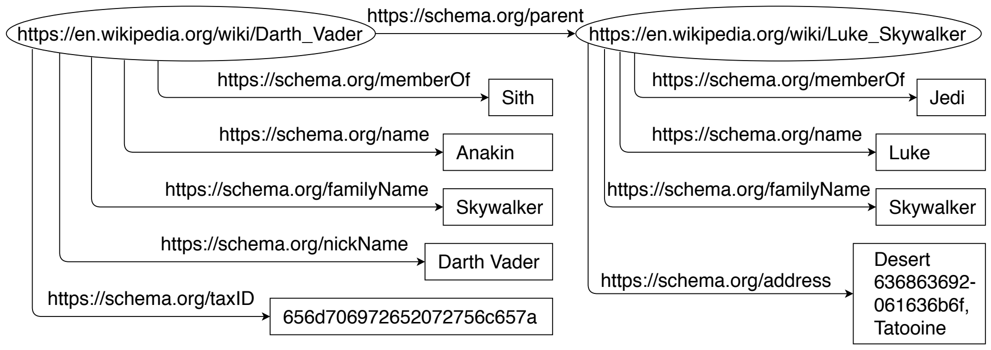

https://en.wikipedia.org/wiki/Darth_Vader
https://schema.org/parent
https://en.wikipedia.org/wiki/Luke_Skywalker
https://schema.org/memberOf
Sith
https://schema.org/name
Anakin
https://schema.org/familyName
Skywalker
https://schema.org/nickName
Darth Vader
https://schema.org/taxID
656d706972652072756c657a
https://schema.org/memberOf
Jedi
https://schema.org/name
Luke
https://schema.org/familyName
Skywalker
https://schema.org/address
Desert 636863692-061636b6f, Tatooine

Figure 1.3: Example of an RDF graph

- RDF graph cannot assign attributes to a relationship without a workaround.

- RDF can assign an element to a subgraph, whereas the property graph cannot.

- Property graph has more efficient storage than the RDF graph as it was designed with this objective.

Overall, apart from the storage of characteristic data with a specific intent (data exchange of triplets on the web), the more utilized graphs are property graphs as they offer more possibilities in total and are more suitable for a general purpose. Graph DBMSs have also been experiencing more attention than RDF stores as shown in Figure 1.1.

1.2.2 Physical Storage

Methods of physical storage differ among particular graph databases. They are usually implemented as an extra layer over another NoSQL database type – a key-value or a document store. The more specific description of such backends is given in chapter 3 introducing individual products and their data storage in detail.

Each graph database can employ a different graph representation for its storage. Given a directed graph[8] $G = (V, E)$, the most common graph representations are (according to Kolář [10], adjusted)

- **adjacency matrix** – matrix $A = [a_{ij}]$ of a size $|V| \times |V|$ where

$$a_{ij} = \text{number of edges between vertices } v_i \text{ and } v_j$$

- **incidence matrix** – matrix $M = [m_{ik}]$ of a size $|V| \times |E|$ where

$$m_{ik} = \begin{cases} 1, & \text{if edge } e_k \text{ starts from vertex } v_i \\ -1, & \text{if edge } e_k \text{ ends in vertex } v_i \\ 0, & \text{otherwise} \end{cases}$$

- **adjacency list** – an array of linked lists, the array has a size $1 \times |V|$. A field $i \in \{0, ..., |V| - 1\}$ of the array represents a vertex v_i and contains a linked list of all its successors.

  The array of linked lists can be extended with another type of information such as an edge identification or an edge/a vertex weight.

1.2.3 Query Language

Unlike relational databases with standardized query language SQL, graph databases do not possess a query language that is generally accepted by all. The reason may be a relatively young concept with much enthusiasm to create an own graph database which, without a language standard, lead to the development of many proprietary languages.

Among these various languages dedicated to graph databases, the following are employed by more than one product.

- **SPARQL** – is a query language for RDF. It offers many opportunities for data manipulation similar to SQL (conjunctions, subqueries, aggregations, and many others) [11].
- **Cypher** – inspired by SQL, Cypher is a declarative language for describing patterns, querying, inserting, modifying and deleting data [12]. It was developed for Neo4j, the most popular, yet single-server graph database. Thanks to its similarity to SQL, Cypher has gained a good reputation and has been released to public.
- **Gremlin** – is a traversal language of the graph computing framework TinkerPop [13]. Gremlin is a functional language and offers both OLTP[9]

[8]Graph definition can be found at the beginning of chapter 2.

[9]OLTP stands for Online Transactional Processing. It is a processing focused on real-time data, continuous load and many inputs.

and OLAP[10] traversals. As the TinkerPop is a widely supported framework by various graph databases, also Gremlin is a widely applied language. More details about both the framework and the language can be found in section 3.2.

[10]OLAP stands for Online Analytical Processing. It is a processing focused on querying, unbalanced load and analytical (aggregated) data.

CHAPTER **2**

Graph Partitioning

The graph partitioning problem has been a well-known and researched task for decades, which gives the advantage of a significant number of already existing methods. These are introduced in following section Algorithms 2.1.

Nevertheless, first, let us define the problem of graph partitioning itself preceded with selected definitions from graph theory necessary for understanding the later introduced concepts.

Graph The graph G is a pair of a set of vertices V and a set of edges E, that is, $G = (V, E)$.

Weight A graph can be extended with weights, assigning each vertex and edge a numerical value called *weight*. A weight function $w(el)$ then returns the weight of an element $el \in \{V \cup E\}$.

Partition A partition of a graph G is then defined according to Bichot and Siarry [14] (modified to fit the notation in this work) as follows:

> Let $G = (V, E)$ be a graph and $P = P^1, ..., P^k$ a set of k subsets of V. The P is said to be a partition of G if:
>
> – no element of P is empty:
> $\forall i \in \{1, ...k\}, P^i \neq \emptyset$;
>
> – the elements of P are pairwise disjoint:
> $\forall (i, j) \in \{1, ..., k\}^2, i \neq j, P^i \cap P^j = \emptyset$;
>
> – the union of all the elements of P is equal to V:
> $\bigcup_{i=1}^{k} P^i = V$

Edge Cut Although a part P^i and a part P^j are disjoint, meaning they do not share any vertices, they can share some edges connecting their vertices. Sum of the weights of these edges connecting vertices of part P^i and part P^j is called an edge cut. This means that an edge cut function $cut(P^i, P^j)$ is defined as

$$cut(P^i, P^j) = \sum_{u \in P^i} \sum_{v \in P^j} \sum_{e \in E^{uv}} w(e),$$

where E^{uv} is the set of edges connecting the vertices u and v. In the case of an unweighted graph, an edge can be assigned a default value of the weight, for example, 1.

Graph Partitioning The problem of graph partitioning is to divide the graph G into k parts $P^1, ..., P^k$ of specific sizes[11] while minimizing the edge cuts between these different parts of the partition,

$$min \left(\frac{1}{2} \sum_{i \in \{1,...,k\}} \sum_{\substack{j \in \{1,...,k\}, \\ i \neq j}} cut\left(P^i, P^j\right) \right),$$

where the division by two is present only to show that each edge cut was counted twice.

This balancing and minimalization task is an NP-complete[12] problem even for the case of $k = 2$ called Minimum Bisection problem [15].

2.1 Algorithms

As already mentioned, a significant number of algorithms for the graph partitioning task already exist. Some of these methods are based on the same principles according to which they can be categorized as f.e. by [16, 17, 18]. Note, that the categories are not always necessarily disjunctive.

This work adopts classification mainly of Buluç et al. [16] nad partly of Fjällström [18]. An extra category of Pregel-compliant algorithms is added as well.

We provide a general overview of graph partitioning algorithms. Various groups of methods are introduced, though not always explained in detail. More relevant algorithms are then discussed more thoroughly.

[11]Often the sizes of partitions are equal, but in the case of different hosts of a graph database cluster, the size corresponds to a host capacity which can differ.

[12] NP-complete problem is a decision problem in computational complexity theory, which belongs to both the NP and the NP-hard classes. The NP (nondeterministic polynomial time) class contains decision problems of positive answer proofs verifiable in polynomial time. The NP-hard class contains problems to which all the problems in NP can be reduced in polynomial time.

2.1.1 Global Algorithms

The most considerable amount of existing algorithms compute their solution while accessing the entire graph. Therefore, it is clear that these methods can work on smaller graphs but they are not intended for a distributed environment. However, they present an essential part of the evolution of graph partitioning algorithms. The global algorithms can either be exact or heuristic as presented below.

2.1.1.1 Exact methods

Exact methods can solve only small problems with long running time and are highly dependent on graph density [19] and bisection width[13]. Majority of the exact algorithms is based on the branch-and-bound method, systematically visiting all potential solutions while cutting out whole subsets of unsuitable solution candidates [21].

2.1.1.2 Heuristic methods

Heuristic methods are providing a “good enough” solution in a reasonable time frame. The problem of graph partitioning is NP-complete as mentioned at the beginning of this chapter 2; therefore, the heuristic methods are the only viable option for any larger problems.

According to Buluç et al. [16], heuristic algorithms can be classified as follows:

- **Spectral Partitioning**

  Algorithms of this group work with *spectrum* of a graph which is a list of eigenvalues of the graph adjacency matrix [17].

  A widely applied concept is an *algebraic connectivity*, also called the *Fiedler vector* after the Czech mathematician Miroslav Fiedler [22]. Algebraic connectivity works with Laplacian Matrix L of a graph $G = (V, E)$, $n = |V|$, which is defined as $L = D - A$ where D is a degree matrix of a graph G, $D = diag(d_1, ..., d_n)$, and A is the adjacency matrix of the graph G [23]. Fiedler vector is then the second smallest eigenvalue of the Laplacian matrix. Partitions are defined according to similarity in values of this vector.

- **Graph Growing**

  Graph Growing is based on easy steps of selecting a random vertex and using the breadth-first search to create a set of surrounding vertices V which, in the case of a bisection, contain half of the vertices or half of the

[13]“The bisection width is the size of the smallest edge cut of a graph which divides it into two equal parts” [20].

total weight of the vertices at the end of the computation. Nevertheless, rather than a standalone method, the graph growing can be used as part of data initialization and its output can serve as an input to f.e. the Kernighan-Lin algorithm [24] described below.

- **Flows**

  Flow algorithms implement the max-flow min-cut theorem, also called the Ford-Fulkerson method, iteratively increasing the value of the flow in a network [25]. However, the method does not guarantee a balanced result at all. For this reason, similarly to Graph Growing, Flows are usually used as subroutines of other algorithms to improve the starting input.

- **Geometric Partitioning**

  If the vertices contain information about its position in space (f.e. GPS coordinates), geometric methods can be performed in order to partition such a graph.

  A simple and often used algorithm is the Recursive Coordinate Bisection comprising four easy steps of determining the longest expansion of domain, sorting the vertices according to coordinates of this domain, partitioning half of the vertices to a subdomain and repeating these steps recursively [27].

2.1.2 Local Improvement Methods

The local improvement methods require an initial partitioning which they take as input and try to reduce its edge cut. Either a method can be selected for initial partitioning or a random initial partitioning can be generated.

- **Node-Swapping Local Search**

  One of the first methods for partitioning a graph-like system was the Kernighan-Lin algorithm, iteratively trying to decrease an edge-cut using a greedy approach until no more improving swaps are available [16]. Local improvement methods are usually variations of this algorithm [18]. One of many improvements of this method is, for example, popular Fiduccia-Mattheyses algorithm, reducing time complexity of Kernighan-Lin's method from $O(|V|^3)$ to $O(|E|)$ [18].

  Although Kernighan-Lin algorithm and its alterations are not considered a suitable method for distributed graphs as they require a cheap random access to all the vertices [28], the TAPER system [29] takes advantage of Greedy Refinement – another modification of Kernighan-Lin – over a derived matrix representing the distributed graph in a simplified form. This promising method works however only on graphs meeting specific requirements.

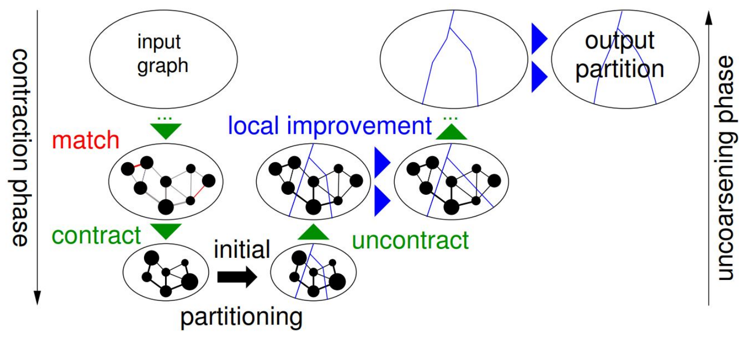


Figure 2.1: The multilevel approach to graph partitioning [16]

- **Tabu Search**

  Similarly to Kernighan-Lin, Tabu Search is a method swapping vertices from a current partition to another. After that, however, the vertex is brought to a tabu list containing vertices forbidden for migration in a certain number of following iterations.

- **Random walk and diffusion**

  Random walk on a graph is a walk[14] in which the next step is selected randomly among the neighbors of a vertex v. Diffusion is a similar process during which an entity is split within a graph, naturally occurring more frequently in dense areas as well as the random walk is staying within dense clusters for a longer time.[16]

2.1.3 Multilevel Partitioning

Multilevel partitioning consists of three main phases [24] also depicted in Figure 2.1:

1. *Coarsening* – The objective of the coarsening phase is to mitigate the partitioning problem to a size small enough to perform otherwise expensive partitioning phase. The nodes are contracted in such a way that the edge cut reflects the cut of the original graph. This same cut value is achieved with

   - merging vertices in such a way that for a coarsed vertex u merging subset of vertices $U \subset V$, its weight is accumulated:

$$w(u) = \sum_{v \in U} w(v)$$

[14]"A *walk* in a graph G is a finite non-null sequence $W = v_0e_1v_1e_2v_2...e_kv_k$, whose terms are alternately vertices and edges, such that, for $1 \leq i \leq k$, the ends of e_i are v_{i-1} and v_i" [30].

- merging any parallel edges into a single edge with a summed weight of the parallel edges as well.

So it is still possible to provide balanced partitioning.

2. *Partitioning* – When the coarsening part derives a sufficiently small graph, a partitioning method is performed. Any algorithm of the earlier mentioned groups can be used if it can incorporate the notion of weights.

3. *Uncoarsening* – Uncoarsening phase includes adopting solution of the previous step and later enhancement of that solution on the fine level. This phase can consist of more iterations of uncoarsening, always working with finer level until the original graph is reached.

There are several approaches to how to perform each phase. A well-known software package METIS [31] provides several implementations of the multilevel partitioning. For each phase, a number of various algorithms can be chosen.

2.1.4 Metaheuristics

The popularity of metaheuristics has been projected into graph partitioning as well. According to Sorensen and Glover [32]:

> A metaheuristic is a high-level problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms. The term is also used to refer to a problem-specific implementation of a heuristic optimization algorithm according to the guidelines expressed in a metaheuristic framework.

Some of the examples of metaheuristics are ant colonies, evolutionary algorithms, already mentioned tabu search and many more [32]. The methods are often metaphors of natural or man-made principles. Regarding graph partitioning, these algorithms can be used independently or as part of other methods such as multilevel algorithms [16].

There has not been yet introduced a metaheuristic solving partitioning of a distributed graph to our knowledge. DMACA – a Distributed Multilevel Ant-Colony Algorithm – could sound like a promising solution. However, the distributed part of its name is devoted to slave calculation on different machines while each slave still needs to have an image of the whole graph [33].

As metaheuristic methods are based on various principles, the introduction of each of these algorithms is already beyond the scope of this work. However, the method Ja-be-Ja introduced in 2.2.2 incorporates simulated annealing in its steps; therefore, a brief description of this method is provided.

Simulated Annealing is a method based on a technique from metallurgy involving heating and gradual cooling of a material. It uses a parameter of temperature T initially set to T_{init}. When searching a state space, a new state is accepted if

- its better than a current state
- or if for a random number $x \in [0,1]$ the equation $x < e^{\frac{-\delta}{T}}$ holds true, where $\delta = newState.cost() - currentState.cost()$ [34].

For each next iteration of a state selection, the temperature T is gradually cooled with a cooling function until it reaches zero. Such a selection means possible acceptance of worsening solutions at the beginning and lowering the probability of such a degradation over time. In other words, it tries to avoid getting trapped in a local optimum.

2.2 Pregel-Compliant Algorithms

This section first introduces the Pregel platform based on its original paper [35] and then states two selected algorithms compliant with this system.

Pregel is a computational model for processing large-scale graphs. It was presented by researches from Google to meet their computational needs of enormous datasets distributed over a large number of clusters.

The main idea of the platform is a sequence of so-called *supersteps*. During each superstep S

- each vertex can read messages sent to it in preceding superstep $S - 1$
- the same user-defined function is called on each vertex
- each vertex can send a message to another vertex (a neighboring vertex or any other vertex but with a specified ID) that will be received in the following superstep $S + 1$
- each vertex and its out-going edges can change its value
- each vertex can decide about mutating the topology of the graph

This “think like a vertex” approach enables running each vertex task of a superstep in parallel. Within a superstep no synchronizing communication is needed. The last stated option of a possible mutation execution is rigorously processed with well-defined rules at the end of a superstep.

The algorithm terminates when each vertex votes to halt, and no further messages should be transmitted, or a user-defined maximum of iterations is reached. At the beginning of the algorithm, each vertex has active status. When the vertex has no further job to process, it deactivates itself to an

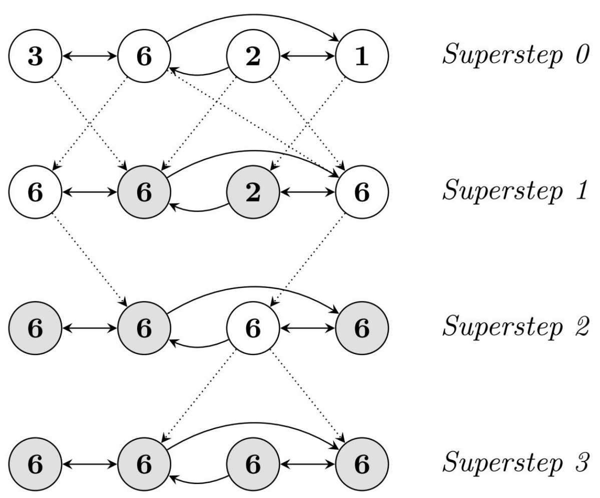


Figure 2.2: Maximum Value Example in Pregel Model. Dotted lines are messages. Shaded vertices have voted to halt.[35]

inactive state by voting to halt. If a message for an inactive vertex is sent later during computation, the vertex is set back to an active state and receives the message. The vertex can later set itself inactive again if it already has no work to do and no messages to process.

An example of the Pregel algorithm is shown in Figure 2.2. The processed task is to propagate the largest value of a vertex within a graph. This means that each vertex first sends its value to its neighbors and if a vertex receives a value larger than its current value it adopts it and sends a message to its neighbors containing this new value.

Pregel system further provides two additional mechanisms:

- *Combiners* – as sending messages to vertices creates some communication overhead, a combiner can be defined summoning all the messages from a partition for a given vertex and combining them into one message. Such a combination can be useful when for example only a sum of the values sent in the messages is relevant.

- *Aggregators* – for a shared piece of information within vertices, aggregators have been introduced. An aggregator receives values sent by vertices in a superstep S, reduces this information to a single value (such as a minimum or a sum) and transmits this value to all vertices, that have this value available in the next superstep $S + 1$.

Thanks to elaborated checkpointing the Pregel framework is fault tolerant. Moreover, the nature of the algorithm indicates that the system is well scalable which is supported by experiments revealing linear scalability.

Class of algorithms naturally suitable for Pregel could be the diffusion system described in 2.1.2. Perhaps the most profound method based on diffusion systems and dedicated to distributed graphs is DiDiC – Distributed Diffusive Clustering [36]. Originally developed for clustering a P2P supercomputer, DiDiC requires only communication between neighbors and an arbitrary initial partitioning. However, it can sustain only an upper bound on the number of clusters which means it does not guarantee a specific amount of partitions and the result is also not necessarily balanced [36, 28].

Two promising algorithms, Vaquero et al. [37] and Ja-be-Ja [28], compliant with Pregel framework are further introduced in more detail.

2.2.1 Vaquero et al.

Paper of Vaquero et al. [37] introduces a method for graph partitioning, promising an adaptive solution on dynamic large-scale graphs. That means, even when the graph is large and changing in time, the method suggests an effective repartitioning and endeavors to maintain its quality as the graph develops.

The proposed method uses the label propagation algorithm that is going to be introduced first. After that, the actual algorithm is going to be described, followed by a definition of its constraints.

2.2.1.1 Label Propagation

The label propagation algorithm serves for identifying communities within a graph. In general, as explained by Raghavan et al. [38], each vertex has first assigned a unique label. Then, the labels propagate through the network as such, that a vertex v determines its label in an iteration t according to labels of its neighbors. A vertex chooses the prevailing label of its neighbors, adopting a label uniformly randomly in the case of a tie. The propagation stops when there are no more label adoptions possible. At the end of this process, vertices with the same label form a partition.

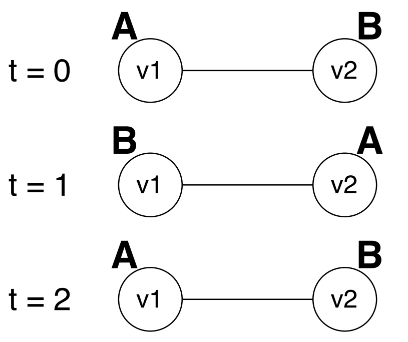
To introduce the label propagation more formally, let $L_1(t), ...L_p(t)$ be the labels active in the network at a moment t. The steps of the propagation then are:

i Initialize the labels of all the vertices. For a vertex v, $L_v(0) = v$.

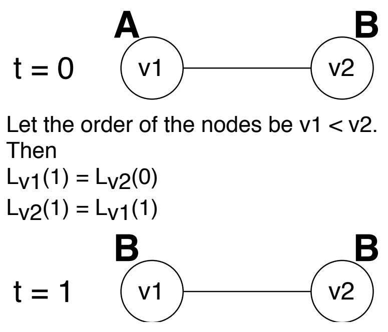
ii Arrange the vertices in a random order.

iii For each $v \in V$ chosen in that specific order such that $v_1, ..., v_m < v < v_{m+1}, ...v_k$, let

$$L_v(t) = f(L_{v_1}(t), ..., L_{v_m}(t), L_{v_{(m+1)}}(t-1), ..., L_{v_k}(t-1)), \qquad (2.1)$$



(a) Label propagation for all the nodes at the same iteration.



(b) Label propagation with prevented oscillation.

Figure 2.3: Label propagation without and with oscillation prevented.

where $k \in \{1, ..., |D_v|\}$ with D_v being a set of neighbors of the vertex v and f returns the most frequent label among the neighbors where ties are broken uniformly randomly. However, if the tie is between the current partition and another one, the current partition is preferred to avoid an extra migration overhead.

iv If all the vertices already have the same label as is the most frequent label among its neighbors (meaning no more label adoptions are possible), stop the algorithm. Else, set $t = t + 1$ and go to ii.

Step iii is present to cope with a problem of oscillation. If the oscillation was not prevented and all the vertices were choosing its label at iteration t according to its neighboring labels from the same iteration t, two vertices could try to endlessly swap its labels as shown in Figure 2.3a with labels A and B. Therefore, the above function from step iii is applied, averting the oscillation as depicted in Figure 2.3b.

2.2.1.2 Steps of Vaquero et al. algorithm

1. The graph is partitioned with any initial partitioning procedure.

2. Vertices perform label propagation as explained above with two major adaptations.

   - In the first step i, the vertices are not assigned a unique label each. Instead, their initial partition is adopted as a label at $t = 0$. This initial labeling ensures the same number of different labels in the first step as the number of partitions.
   - Instead of steps ii and iii which require demanding ordering of all the nodes, it confronts the oscillation problem with a random factor

to migration decisions. Each vertex adopts a label with a probability s, $0 < s < 1$.

This approach introduces an issue with convergence time. With $s = 0$ no swaps occur and when $s = 1$ the swapping is not prevented. At the same time, the lower the s, the lower the number of overall migrations and so the longer the time to converge. Moreover, the higher the s, the higher the chasing effect and so is the number of wasted migrations, prolonging the convergence time again. The experience from the original paper [37] says that the best setting of s is $s = 0.5$.

In order to simplify, the constraining method maintaining balanced partitions will be explained under these basic steps of the algorithm.

3. Migration of vertices which were set to change its partition.
4. Any new incoming vertices are assigned a partition according to the same strategy – prevailing partition among their neighbors.

2.2.1.3 Constraints

In order to create a balanced partitioning, Ugander and Backstrom [39] – on who Vaquero et al. [37] bases – constraint the partition size with both lower and upper bound equal to $\left\lfloor (1-f)\,\frac{|V|}{n} \right\rfloor$ and $\left\lceil (1+f)\,\frac{|V|}{n} \right\rceil$, where f is some fraction $f \in [0;1]$ and n is the number of partitions.

However, Vaquero et al. approach partition balancing with their own constraint. They define the maximum number of vertices allowed to change a partition P^i to a partition P^j, $i \neq j$ in an iteration t as

$$Q^{i,j}(t) = \frac{C^j(t)}{|P(t)| - 1}, \tag{2.2}$$

where $C^j(t)$ is a capacity constraint on partition P^j such that $|P^j(t)| \leq C^j$; $P(t) = \{P^1(t), ..., P^n(t)\}$, so $|P(t)| = n$.

In other words, the remaining capacity of a partition P^j is equally divided among other partitions as available space for its vertices to migrate to the partition P^j. This is an upper bound constraint. The authors do not explicitly talk about any lower bound constraint.

2.2.2 Ja-be-Ja

Ja-be-Ja[15] promises a distributed, local algorithm for balanced partitioning [28]. It is a graph repartitioning method resembling the previously described algorithm of Vaquero et al., yet with significant differences. Ja-be-Ja uses a friendly terminology of colors and energies which helps a quicker

[15]"Ja-be-Ja" means "swap" in Persian [28].

comprehension. It also offers solution both for edge- and vertex-cuts, and it discusses a case of weighted graphs.

Different colors of vertices represent different partitions as well as different labels do in label propagation.

The energy of a graph, that is, the number of edges between vertices with different colors (meaning the edge-cut), is defined as

$$E(G, P) = \frac{1}{2} \sum_{v \in V} \left(d_v - d_v \left(\pi_v \right) \right), \tag{2.3}$$

where $\pi : V \to \{1, ..., k\}$ is a function assigning a color to a vertex. So π_v is the color of vertex v. d_v is the number of neighbors of the vertex v, and $d_v(\pi_v)$ is the number of neighbors of the vertex v with the same color as the vertex v has. Hence, the expression $\left(d_v - d_v \left(\pi_v \right) \right)$ refers to number of neighbors of vertex v with different color. Each edge in the energy function is counted twice which is the reason for division by two.

In other words, the problem of balanced partitioning is in Ja-be-Ja expressed as a minimization of the energy function E.

2.2.2.1 Steps of Ja-be-Ja

1. Colors are assigned to vertices uniformly at random. The number of different colors equals the number of different partitions.

2. Each vertex samples its neighbors (or random vertices – as explained in part 2.2.2.2) and computes the utility (change in energy) of its color exchange. The two colors are exchanged if the swap results in energy decrease. If more neighbors are identified as swapping candidates, the one with the highest energy decrease is chosen for the color exchange. See part 2.2.2.3 for more detail. However, if the winning candidate is located on a different physical node then the current vertex and some other potential candidates are located on the same physical node as the vertex, a candidate from the same physical node is chosen.

Note, that in Vaquero et al. the labels are adopted, so the number of a particular label occurrences changes with the necessity to introduce constraints on label selection. In comparison with that, the Ja-be-Ja color swapping keeps the number of each color occurrences unchanged.

2.2.2.2 Sampling

There are three ways of how to choose a set of potential vertices for swapping.

- Local – only the neighbors of a given vertex are chosen for the sample set.

- Random – the sample set if chosen uniformly randomly from the entire graph. Each host maintains a sample of random vertices from which its local vertices pick their random sample.
- Hybrid – it combines both the local and random approach in a specific order. If the local method does not bring any vertices that could improve the current state, a random sample is selected and investigated for exchange.

Experiments in the original paper [28] show that the hybrid approach performs the best in a majority of the cases, so it is also used by the authors in all subsequent evaluations.

2.2.2.3 Swapping

The decision to swap a color between two vertices has two parts

1. a utility function of a color swap
2. policy to avoid local optima

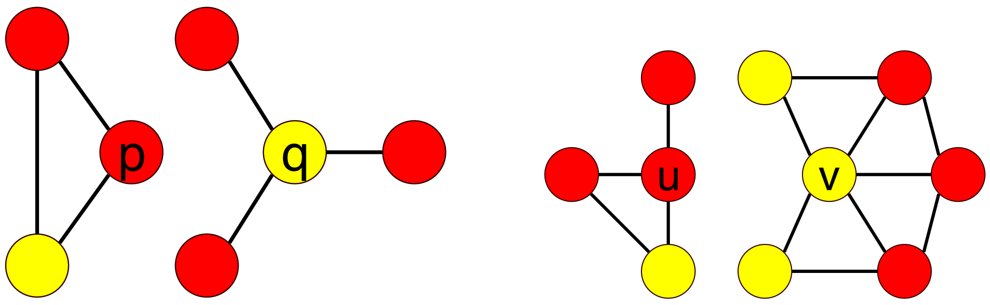
The objective of the utility function is to reduce the energy of an edge-cut. In order to do so, the number of neighbors with the same color $d_v(\pi_v)$ should be maximized for every vertex v. Therefore, a vertex v and a vertex u swap their color only if it reduces their energy, which means if:

$$d_v(\pi_u)^\alpha + d_u(\pi_v)^\alpha > d_v(\pi_v)^\alpha + d_u(\pi_u)^\alpha, \tag{2.4}$$

where α is a parameter which can influence a distribution preference. To give a better insight of the α parameter, let us consider the example adopted from [28] in Figure 2.4. If $\alpha = 1$, the vertices p and q in Figure 2.4a will exchange their colors, because $1 + 3 > 1 + 0$, in other words, the total number of neighbors with the same color increases. On the other hand, in the case of Figure 2.4b the colors of vertices u and v do not change because $1 + 3 \not> 2 + 2$. However, if $\alpha > 1$, these vertices will swap the colors. This means that the alpha can relax the condition of colors exchange which can help the following exchange of the remaining two yellow and two red vertices in 2.4b.

Furthermore, overcoming the problem of getting trapped in a local optimum, Ja-be-Ja algorithm incorporates the method of simulated annealing. That means a variable $T \in [1; T_0]$ is introduced, representing a temperature starting at T_0 and decreasing over time. The impact of the temperature is the acceptance of degrading solutions at the beginning of the algorithm (that is, accepting exchanges that increase the energy) and restricting these worsening adoptions over time. The utility function U is then defined as

$$U = \left[d_v\left(\pi_u\right)^\alpha + d_u\left(\pi_v\right)^\alpha\right] \times T_r - \left[d_v\left(\pi_v\right)^\alpha + d_u\left(\pi_u\right)^\alpha\right], \tag{2.5}$$



(a) Color exchange between p and q is accepted if $\alpha \geq 1$.

(b) Color exchange between u and v is accepted only if $\alpha > 1$.

Figure 2.4: Examples of two potential color exchanges [28].

where $T_r = max\{1, T_{r-1} - \delta\}$. Parameter δ influences the speed of the cooling process. Over time the temperature sets to 1 after which the utility function comes back to the logic of the equation 2.4.

2.2.2.4 Weighted graphs

In the case of weighted graphs, the Ja-be-Ja algorithm offers a solution consisting of an adaptation of the number of neighbors d_v. The number of neighbors with a color c is defined as

$$d_v(c) = \sum_{u \in D_v(c)} w(u, v), \tag{2.6}$$

where $w(v, u)$ represents the weight of an edge (or edges) between the vertices u and v.

2.2.3 Summary

In conclusion, the Vaquero et al. algorithm seems less computationally strenuous as it always stays in local scope whereas Ja-be-Ja needs to communicate after each iteration within the whole graph to swap corresponding vertices and to gather a set of random vertices for each partition in the next round. The swapping process of Ja-be-Ja can also be expected to be much slower than the label adoption of Vaquero et al.

Ja-be-Ja algorithm always results in a well balanced solution (if the initial partitioning is balanced), because the proportion of particular colors does not change. On the other hand, it cannot perform a bit elastic behavior regarding balancing (it cannot be set that the partition proportions can, for example, differ up to five percent). Nevertheless, the Vaquero et al. method proposes only an upper bound on each partition, which can create balanced solutions when the majority of partitions space is taken by data. However, if the amount

	Vaquero et al.		Ja-be-Ja	
computationally demanding	less	+	more	-
partition balancing	worse	-	always	+
elasticity of balancing	better	+	never	-
manages local optima	no	-	yes	+
manages weighted graphs	no	-	yes	+

Table 2.1: Comparison of Pregel-compliant algorithms, Vaquero et al. and Ja-be-Ja

of data on each partition is, for instance, less then half of its available space, it could bring an unbalanced solution, even entirely shifting data out of some partitions.

Last but not least, Ja-be-Ja algorithm confronts the problem of local minima by applying simulated annealing. Vaquero et al. do not even mention this issue. Furthermore, Ja-be-Ja offers a solution for weighted graphs whereas Vaquero et al. do not.

Table 2.1 summarizes these conclusions dividing them into five categories. The resulting values are labeled with plus or minus signs to denote the performance of the given algorithm in the corresponding category.

Chapter 3

Technology

Although there exist several single-server solutions of graph databases such as the very popular Neo4j or AllegroGraph and Amazon Neptune, with graphs of a size of huge social networks or stock markets, it is inevitable to work with data distributed over several nodes. The market already offers several products of distributed graph databases. These particular software solutions are introduced in the main part of this chapter. The second part acquaints with the graph computing framework Apache TinkerPop, its structure, and language.

3.1 Distributed Graph Databases

As it is common for a software community, the solutions of distributed graph databases are both open-source and commercial. We are going to mention representatives of both groups, focusing in more detail on open-source implementations.

3.1.1 JanusGraph

JanusGraph is a distributed transactional[16] graph database. It is a fork of the open-source project Titan which further development was stopped in 2015 when its mother company Aurelius was acquired by DataStax [40]. Since its origin, JanusGraph has been growing in its popularity significantly [41].

Like its predecessor, JanusGraph can store hundreds of billions of vertices over several independent machines. For retrieval and modification of data, it uses the Gremlin query language which is a component of the Apache TinkerPop framework (described in 3.2).

[16]As section 1.1 introduced transactions, a transactional database means that an operation or a set of operations is encased in a transaction which can run in parallel with other transactions always resulting (with BASE concept eventually) in the same state.

3.1.1.1 Data Storage

JanusGraph does not provide its own storage backend. Instead, it allows the user to choose external solutions in accordance with the user's needs. If users wanted to change their storage backend, the only adjustment they need to do is to change the configuration.[42]

Supported Storage Backends

- **Apache Cassandra** – Apache Cassandra is an open-source wide column store, NoSQL DBMS that is distributed and fault-tolerant. It is claimed that Cassandra's replication is best-in-class [43] and it is linearly scalable without any bottlenecks [44], successfully running with 75, 000 physical nodes storing over 10 PB of data.
- **Apache HBase** – Apache HBase is an open-source database created according to Google's Bigtable, that is distributed and scalable. It is a Hadoop project running on the Hadoop Distributed File System (HDFS), and so it has been tailored to store large tables of billions of rows and millions of columns [45].
- **Google Cloud Bigtable** – Google Cloud Bigtable is Google's database powering Google Search, Analytics, Maps or Gmail [46]. Bigtable is thus designed for massive loads, low consistent latency and high throughput. However, Cloud Bigtable is a commercial product with a set list of pricing.
- **Oracle Berkeley DB** – Berkeley DB is an open-source storage (with a commercial option) running in the same Java Virtual Machine (JVM) as JanusGraph. The advantage is very high performance [47]. On the other hand, all the data have to fit a local disk (being only a single-server usage), limiting a graph to 10 – 100s million vertices [42].

Connection Modes

The storage backends can be connected to JanusGraph in different modes [48]. These are:

- **Local Server Mode** – Storage backend is run on the same local host as JanusGraph and a related user application. The storage and JanusGraph communicate via localhost socket. The mode is depicted in Figure 3.1.
- **Local Container Mode** – If a storage backend is not supported on a demanded operating system (as Cassandra does not natively run on Windows and OSX, for example), a Docker container can be used to overcome this obstruction.

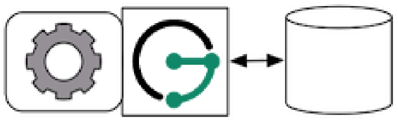


Figure 3.1: Local Server Mode of JanusGraph with storage backend [48]

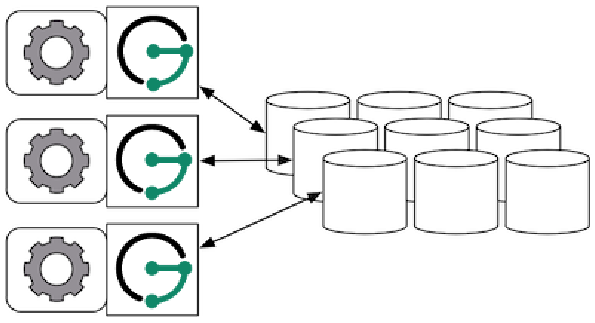


Figure 3.2: Remote Server Mode of JanusGraph with storage backend [48]

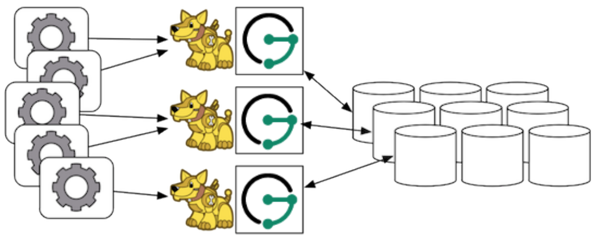


Figure 3.3: Remote Server Mode with Gremlin Server of JanusGraph with storage backend [48]

- **Remote Server Mode** – In a distributed environment, the storage backend runs on several physical machines, whereas a JanusGraph instance runs on one or few. The storage cluster stores the whole graph, and JanusGraph instances can connect to the cluster via a socket. As shown in Figure 3.2, a user application can be run in the same JVM as a Janusgraph instance.
- **Remote Server Mode with Gremlin Server** – If a JanusGraph instance is wrapped with Gremlin Server, a user application can communicate with Gremlin Server remotely as a client. This mode also removes the requirement of a user application to be written in Java as later explained in 3.2. It is depicted in Figure 3.3.
- **JanusGraph Embedded Mode** – A storage backend and JanusGraph can also run in the same JVM, removing the overhead of communicating over the network. In an embedded mode, JanusGraph runs a storage daemon, thus it connects to its cluster. The figure 3.4 shows a situation

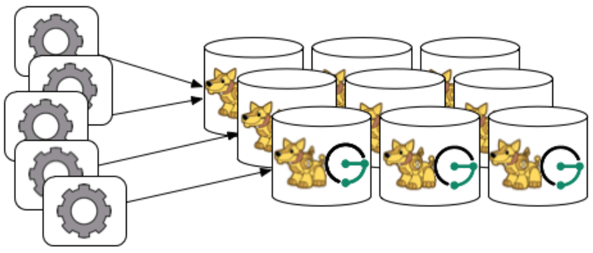


Figure 3.4: Embedded Mode of JanusGraph with storage backend [48]

	Local Server	Local Container	Remote Server	Remote Server with Gremlin Server	Embedded
Apache Cassandra	✓	✓	✓	✓	✓
Apache HBase	✓		✓	✓	
Google Cloud Bigtable	✓		✓	✓	
Oracle Berkeley DB					✓

Table 3.1: Possible combination of storage backends and connection modes

when Gremlin Server wraps a JanusGraph instance in order to enable remote communication with an end-user application. An embedded mode requires some garbage collector tuning though [48].

Nevertheless, not all of the supported storage backends support all of the mentioned modes. A table 3.1 contains these possible combinations of data storages and connection modes.

Data Model

JanusGraph stores its graph in an adjacency list which was introduced in section 1.2.2. However, JanusGraph does not store adjacency vertices in the list, but incident edges to keep the references unambiguous (as the represented graph is a multi-graph). Such adjacency lists are stored as key-value pairs, where the key is a vertex ID and the value is represented with all incident edges and also properties of the corresponding vertex. The layout is shown in Figure 3.5.

3.1.2 OrientDB

OrientDB is an open-source NoSQL multi-model database. Its objective is to remove the need of looking for specific dedicated solutions when possessing

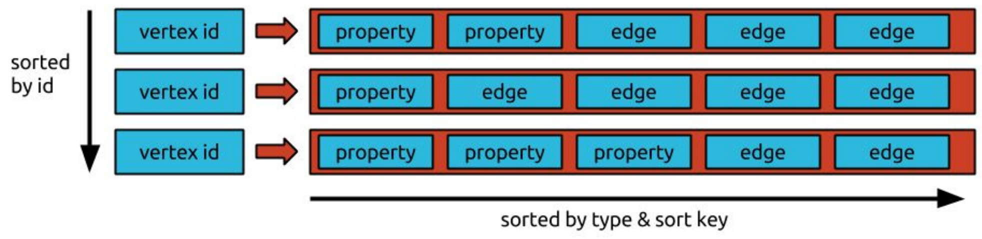


Figure 3.5: JanusGraph data layout [49]

data of various types. The solution of OrientDB is to store all the different data under one hood.

In order to make an easy adjustment for developers accustomed to relational databases, OrientDB provides an SQL interface which can be used for any operation with the database. Since version 3.0, OrientDB also implements TinkerPop 3 interfaces.

3.1.2.1 Data Storage

In comparison with JanusGraph, OrientDB does not support external storage backends. It provides its own implementation of data storage instead. Nevertheless, the store still has three different modes of possible usage [50].

Connection modes

- **plocal** – "paginated local storage" is a connection mode which uses a disk cache and a page model. The data are accessed from the same JVM in which OrientDB is running. So the communication overhead over the network is not present.
- **remote** – data are stored in a remote storage which is accessed via network.
- **memory** – no persistent storage is used, all the data stay in the memory. Consequently, as soon as JVM is shut down, all the data are lost. Therefore, this mode is suitable for testing purposes and cases in which a graph does not have to persist.

Data Model

In introducing parts of the OrientDB tutorial, it is proclaimed that unlike some other multi-model DBMSs, OrientDB does not provide only an interface for models translating it into another one. Instead, its engine has been built to natively support graph, document, key-value and object models at once [50].

Nevertheless, OrientDB does translate Graph API into Document API and further to a key-value or object-oriented storage as shown in Figure 3.6. The

```
        THE USER

 ||                 ||
_||_                ||
\  /                ||
 \/                _||_
+-------------+    \  /
|  Graph API  |     \/
+-------------+-----------------+
|         Document API          |
+-------------------------------+
| Key-Value and Object-Oriented |
+-------------------------------+
```

Figure 3.6: OrientDB APIs [51]

translation does not add any extra complexity, meaning it does not cause an increase of read and write operations, which means the native support of the models.

Both vertices and edges are stored as instances of the classes `Vertex` and `Edge`, respectively. Links between the vertices and edges are not pointers but direct links, which is one of the basic concepts of OrientDB [52].

In other words, a graph containing a vertex v_1, a vertex v_2 and an undirected edge e_{12} connecting vertices v_1 and v_2, would be stored as follows

- v_1 as an instance of the class Vertex (or its subclass) with a reference to e_{12}
- v_2 as an instance of the class Vertex (or its subclass) with a reference to e_{12}
- e_{12} as an instance of the class Edge (or its subclass) with references to v_1 and v_2

The distributed architecture of OrientDB uses the Hazelcast project distributing data among the nodes of a cluster. The distributed configuration has some limitations such as no auto-sharding, some absent map/reduce functions or some issues with constraints on edges.[53]

3.1.3 ArangoDB

ArangoDB is another multi-model database supporting graph, document, and key-value storage. Therefore, ArangoDB is a similar product to OrientDB, both experiencing equal attention by users [41]. Nonetheless, ArangoDB is written in C++ while OrientDB is a Java application. Both approaches come with its pros and cons.

Alongside with a community edition which is available under the Apache 2.0 license, there is an enterprise edition with additional features for companies.

ArangoDB provides its own query language AQL (ArangoDB Query Language) which resembles SQL. Although there has been a discussion within the ArangoDB Community about supporting the TinkerPop 3 framework [54], there has not been such a release yet.

3.1.3.1 Data Storage

ArangoDB used to provide only its custom solution of a storage engine based on memory-mapped files (mmfiles) [55]. Beginning with version 3.2 released in July, 2017 [56] another option of pluggable backend storage is provided as well. A user can select the RocksDB, an embeddable persistent key-value store developed by Facebook [57].

ArangoDB offers a distributed organization of data over nodes of a cluster as well. The distributed mode depends on the model used, key-value store being the best and graph data performing the worst regarding the scalability.

Connection Modes

- **MMFiles** – Memory Mapped Files is a native default storage engine of ArangoDB that needs to fit the whole dataset into memory together with indexes. Writing operations block reading operations on a collection level.

- **RocksDB** – Size of data is limited by disk space, not memory. Indexes are also stored on a disk which results in much faster restart due to no need of index rebuilding. RocksDB writes and reads concurrently [55].

Data Model

Each vertex and edge is stored as a document in ArangoDB. Documents in ArangoDB form collections which are uniquely identified with an identifier and a unique name. A collection also needs to have an assigned type. There are two types of collections so far in ArangoDB – the Document and the Edge.

Each type of an edge (for example "knows" or "owns") is an edge collection. An edge collection has special attributes *_from* and *_to* declaring an inbound and an outbound vertex. Each type of vertex (for example "a person", "a company",...) is a document collection (also called vertex collection).

3.1.4 Other Open-source Distributed Databases

There are more solutions to distributed open-source databases in the market. However, they are rather of a smaller size, often lacking detailed information about data storage or experiencing no further development.

- **Dgraph** – a distributed graph database using a GraphQL language for its traversal.

  Dgraph does not support external storages; it uses its own embeddable key-value database BadgerDB written in pure Go [58]. Unfortunately, the documentation is not as rich as in other analyzed solutions. It also lacks detailed information about a graph representation in storage.

  Nonetheless, Dgraph was established in May, 2016 [59] which makes it a very young product. Version 1.0.0. was released just at the turn of the years 2017 and 2018 and the project is currently under active development. Therefore, improvements in many ways can be yet expected.

- **Virtuoso** – on the contrary, the Virtuoso Universal Server is a very old product regarding software age. Created in 1998 but with the last release dated back to May 2018 [60], Virtuoso is a time-tested solution of a multi-model database.

  Nevertheless, Virtuoso is a multi-model relational database. Therefore, although it supports RDF graphs, these are stored as data in a relational database with the need to use joins when querying a graph. For this reason, it is not a graph database by definition.

- **InfoGrid, HyperGraphDB, FlockDB** – the development of these open-source graph databases has already been abandoned.

3.1.5 Other Commercial Distributed Databases

The market offers several commercial solutions for distributed graph databases as well. As the products try to protect their know-how to keep their competitive power, they usually do not provide detailed information about their internal implementation. This is the reason why it is almost impossible to find specifics about their data storage and modeling. Nevertheless, a short summary of these products is provided for a better general view of the already introduced distributed graph database domain.

- **DataStax Enterprise Graph** – a distributed database built on Apache Cassandra. Such an arrangement resembles JanusGraph – formerly Titan – architecture, as the DataStax bought the whole team of Titan developers [40] as already mentioned.

- **Microsoft Azure Cosmos DB** – a multi-model database offered by Microsoft on its cloud platform Azure. Its data can be accessed with SQL, CQL (Cassandra Query Language), MongoDB protocol, Gremlin language or a proprietary API [61].
- **Stardog** – a distributed graph database supporting both property graph and RDF graph data model with SPARQL query language and Gremlin traversal language [62].

There are of course several graph databases whose development has been ended, such as **BlazeGraph**, **Sparksee** or **GraphBase**. On the other hand, there also exist very promising brand new commercial graph databases that provide handling of a huge amount of real-time data. A Croatian product **MemGraph** has a strong competitor in **TigerGraph** offering a native parallelism handling 256,000 transactions per second [63].

3.2 Apache TinkerPop

Apache TinkerPop is an open-source graph computing framework for both Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) [64]. The TinkerPop project originated in 2009 and it is still actively developed with the latest subversion 3.3.4 of Tinkerpop 3 released in October, 2018 [65].

The framework is so-called vendor-agnostic which means that it does not expect a product of any specific manufacturer. It is currently utilized by all the major graph systems such as Neo4j, JanusGraph, OrientDB, DataStax Enterprise Graph, and many others [66].

Apache TinkerPop is distributed under the Apache 2.0 license. Consequently, it can be modified and redistributed when preserving the copyright and attaching a specific disclaimer [67].

3.2.1 Structure

Serving as an abstraction layer over graph databases, Apache TinkerPop is a collection of technologies incorporating its own and third-party libraries and systems [68]. The TinkerPop is written in Java language as a Maven[17] project. It is composed of several parts depicted in Figure 3.7 and introduced in this section.

Note, that the current TinkerPop 3 is discussed as version 3 has existed for three years so far. For this reason, the obsolete TinkerPop 2 will not be examined here. However, for users familiar with TinkerPop 2 it is important to mention that the two versions vary significantly. The formerly individual TinkerPop 2 projects have been merged into nowadays Gremlin[18] [69].

[17]Maven is a building tool for software project management and a build automatization.

[18]Specifically, TinkerPop 2 *Blueprints* are now Gremlin Structure API, *Pipes* are Graph-

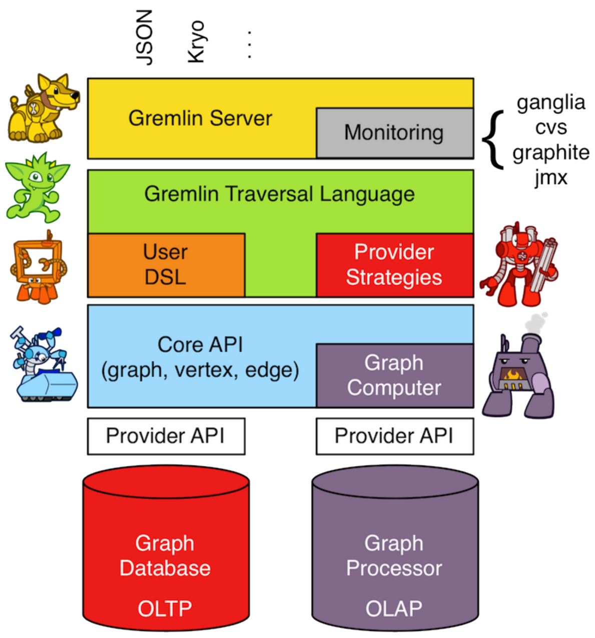


Figure 3.7: Apache TinkerPop framework [68]

3.2.1.1 Gremlin Server

Although TinkerPop provides the Gremlin Console for direct communication with an underlying database, there also exist the Gremlin Server enabling communication with remote databases. A request with a script written in the Gremlin language (described below) is sent, and its result is received in a response. The connection can be created via HTTP or WebSocket.[68]

The communication via WebSocket can be written either in Java, as the whole TinkerPop project is developed in Java, or in any different language as long as a compliant driver exists. Available drivers already include several languages such as Python, Go, .NET, PHP, Typescript or Scala [64].

The Server also provides monitoring capabilities and supports Ganglia/Graphite/JMX and other metrics [66].

3.2.1.2 Gremlin Traversal Language

Gremlin Traversal Language (shortly Gremlin) serves for retrieval and modification of data in any graph database, provided that the database supports the TinkerPop.

Belonging to the functional group of languages, Gremlin can create a query as a sequence of operations chained with dots and evaluated from left to right.

Traversal, *Frames* transformed into Traversal, *Furnace* was divided into GraphComputer and VertexProgram and *Rexster* is now known as Gremlin Server [69].

The example “Find Anakin, traverse to his son and then to his sister and retrieve her name” would in Gremlin look like

```
g.V().has('name', 'Anakin').out('son').out('sister').values('name')
```

This query means in detail [70]:

1. g – current graph traversal that could have been created as

   ```
   graph = GraphFactory.open('conf/star-wars-db.properties')
   g = graph.traversal()
   ```

   g is so called `GraphTraversalSource` and the Gremlin traversals are spawned from it.

2. V() – all the vertices of the graph. Another method that a `Graph-TraversalSource` offers is $E()$ – all the edges of the graph. Return type of both $V()$ and $E()$ is a `GraphTraversal` containing many methods returning `GraphTraversal` as well (so that the steps could be chained).

3. has('name', 'Anakin') – filtering the vertices to get only those with property 'name' equal to 'Anakin'.

4. out('son') – traversing outgoing edges with type 'son' from vertices of Anakins (actually one vertex of Anakin Skywalker).

5. out('sister') – traversing outgoing edges with type 'sister' from vertices of Anakin's sons (Luke Skywalker).

6. values('name') – retrieve value of the property 'name' of all the sisters of Anakin's sons ('Leia').

Each operation separated with a dot is called a *step*. Gremlin language provides a significant number of steps. For example `map`, `filter`, `sideEffect`, `toList`, `addE`, `aggregate`, `limit` and many others, all thoroughly explained in its Documentation [69].

Path An important step for this thesis is the `path()`-step. Added to the sequence of steps, the path operation returns the whole path a traversal walked when creating the query left of the path entry.

Extending the previous example of Anakin's children with `path` brings the following result:

```
g.V().has('name','Anakin')
     .out('son').out('sister').values('name').path()
==> [v[1],v[2],v[3],Leia]
```

The path-step is enabled thanks to the `Traverser` in which all the objects propagating through traversal are wrapped [69]. In other words, a traverser keeps all the metadata about the traversal and these metadata can be accessed.

Note that the usage of the word *path* does not strictly correspond to rigorous graph theory. A path in graph theory is defined as follows (adjusted according to Bondy and Murty [30])

Definition. *A path is a finite non-null sequence $P = v_0 e_{01} v_1 e_{12} v_2 ... e_{(k-1)k} v_k$, where terms e are distinct edges and terms v are distinct vertices, such that, for $1 \leq i, j \leq k$; the ends of e_{ij} are v_i and v_j.*

The important part of the definition is the distinct edges and distinct vertices. If this condition is loosened, and the sequence can contain more occurrences of the same edge or vertex, we call such a sequence a *walk* [30]. The TinkerPop function `path` allows multiple incidences of a given element; therefore, the function actually represents a *walk*, not a *path*.

In this text we are going to use the term *path* in conformity with the TinkerPop framework to avoid any misunderstandings because in the majority of cases we use the term referring to logging traversal paths with the function `path`[19].

3.2.1.3 Graph Computer

The `GraphComputer` interface enables OLAP processing during which the whole graph can be accessed several times. As such, the formation of results can take minutes or hours. Accordingly, the `GraphComputer` was designed to run in parallel and operate on a distributed multi-cluster environment. Two main elements of the Graph Computer logic are the `VertexProgram` and the `MapReduce` interfaces.

- **Vertex Program**

  Section 2.2 introduced Pregel computational model developed to process large-scale graphs. It is based on "vertex-centric" thinking when during each iteration called the *superstep* each vertex executes a user-defined function and sends messages to other vertices.

  The `VertexProgram` is an implementation of such a "think like a vertex" approach. Its method `execute` represents the function performed

[19]TinkerPop framework also contains a `simplePath` function for filtering results containing only simple paths, that is, paths with distinct elements – so actually *paths* of the rigorous graph theory.

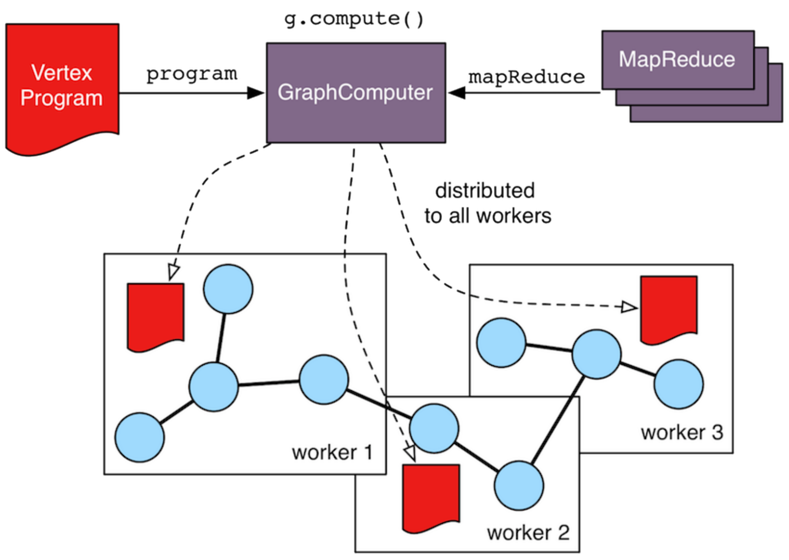


Figure 3.8: OLAP processing in Apache TinkerPop [69]

every iteration on each vertex. Furthermore, it also contains methods such as `setup` called at the beginning of computation, `terminate` called at the end of each iteration or `getMemoryComputeKeys` to acquire properties common to all vertices over the whole computation. All the `VertexProgram` methods can be found in TinkerPop Javadocs [71].

The vertices are in practice divided into subgroups managed by individual workers. The worker executes the user-defined function on all its vertices. That behavior is shown in Figure 3.8.

- **MapReduce**

  The `MapReduce` is an implementation of the standard MapReduce model for big data processing. It comprises two main functions

  – `map` – converting an input of a key-value pair into an output of a key2-value2 pair.

    For example, taking the first sentence of this description and completing a task of counting the number of words of given length, the mapping function would return *3: The, 9: MapReduce, 2: is, 2: an, 14: implementation, 2: of, 3: the, 8: standard, 9: MapReduce, 5: model, 3: for, 3: big, 4: data, 10:processing.*

  – `reduce` – converting an input of a key2-list(value2) pair to an output of a key3-value3.

    The input from the previous example would look like *3: [The, the, for, big], 9: [MapReduce, MapReduce], 2: [is, an, of], 14: [implementation], 8: [standard], 5: [model], 4: [data], 10: [processing]* and the reduce function would create an output of *3: 4, 9: 2, 2: 3, 14: 1, 8: 1, 5: 1, 4: 1, 10: 1*.

In TinkerPop, the `MapReduce` is also applied for all the aggregating requests of the `VertexProgram`; for this reason, it is a part of the Figure 3.8.

3.2.1.4 Other Components

Other parts of the TinkerPop framework from Figure 3.7 are briefly described here. In the explanations, the term "providers" constitutes those providers of graph databases who want their systems to support the TinkerPop framework.

- **Core API** – contains all the definitions of a `Graph`, `Vertex`, `Edge`, etc. which must be implemented by the providers.
- **Provider Strategies** – providers can define specific compiler optimizations to leverage unique features of their systems [66].
- **User DSL** – the abbreviation *DSL* stands for Domain-Specific Language. The Gremlin language uses rather low-level knowledge of a graph; therefore, the queries may sometimes seem to be rather complicated. To eliminate the unnecessary interference from a user, a DSL can be specified to facilitate the query definition. For example, instead of `g.V().has('jedi','name', 'Luke')` the DSL could enable a simple query `sw.jedi('Luke')`.

Chapter **4**

Analysis and Design

The objective of the implementation part of this work is to propose a different distribution of a graph stored in a distributed graph database in order to make user queries more efficient. For this reason, we need to identify one or more subprocesses that slow down the computation of a query result.

Performance of a user query is reflected by the traversal it triggers. The query is composed of subsequential steps which serve for navigating the traversal throughout the graph. In fact, the global analysis of the graph is not required, the traversal uses only its local subsets of data. This behavior does not bring any severe performance caveats for single-server graph databases.

Nevertheless, in the case of a distributed graph, the traversal can involve elements stored on different physical hosts, which degrades the locality conduct. The network communication slows down the data retrieval substantially[20]. Therefore, in order to make the traversal more effective, we want to minimize the number of physical hosts involved in a query.

To complete this objective, we need to know the usual traffic on the graph. This task could be logically divided into two main parts:

- extraction of necessary information from passed queries for further analysis

- analysis of the extracted data resulting in a proposal of a more efficient distribution.

Note, that we do not consider usage of replicas in the graph database. Similarly to other related papers, we always work with the notion of each instance stored exactly once. The introduction of replicas is proposed as one of several possible extensions and enhancements in chapter 7.

[20] When considering the data already loaded in memory on both hosts, then fetching data locally from the main memory takes about 100 nanoseconds [72] whereas reading 32 bytes on 10 Gbps Ethernet takes approximately 40,000 nanoseconds [73].

This chapter first defines the functional and non-functional requirements of the implementation. Subsequently, the model for extracted data storage is designed. Later, the overall architecture design is introduced, followed by details of its individual parts, including the selection of repartitioning algorithm.

4.1 Requirements Analysis

Software system requirements are usually categorized into functional and non-functional groups. Functional requirements describe the real functionalities of the system – that is, *what* the system should do. On the other hand, the purpose of non-functional requirements is to express *how* the system should do it, i.e., complying with what constraints.

Both groups of requirements are here further divided into two categories following the logical separation at the beginning of this chapter – one category setting requirements for the query data extraction and the other category managing further analysis and redistribution, abbreviated as the "Graph Redistribution Manager".

4.1.1 Query Data Extraction

The functional and non-functional requirements for the Query Data Extraction are presented in Table 4.1.

The requirement of Traversal Monitoring determines that only the OLTP traversals are monitored. The OLAP queries usually trigger traversals of the whole graph. That is visible, for example, from the TinkerPop OLAP functionality of the Pregel-like Vertex Program operating on each vertex. Logging all the elements of the graph does not create any added value, therefore, the OLAP queries are not supported.

4.1.2 Graph Redistribution Manager

While the Query Data Extraction has a clearly stated objective, the Graph Redistribution Manager (GRM) could be designed in several ways including various functionalities and extensions as it represents a standalone application. To abide by the scope of this work, the GRM is approached as a Proof of Concept (PoC) – fully functional piece of code to demonstrate the feasibility of proposed methods – but without any extra functionalities such as a user interface.

Nevertheless, we set some requirements even for the PoC, although the demands on the system are lower. The reason for incorporating one of the non-functional requirements – interoperability – should be further described here. The idea is to allow the GRM system to work over either a production database or a simplified representation of the production graph.

- *Production database* – The GRM works directly with the production database. Therefore, the algorithm runs also over the production database.
- *Production graph image* – The GRM works over a graph database containing an image of the production database. The image does not have to incorporate all the data of the production database, but:
  - its graph must have the same topology
  - each element has to contain the original ID
  - each element has to contain information about the partition (the physical node) on which it is allocated.

  The computational burden of repartitioning can be shifted in this manner from production to another environment.

The architecture in 4.3 presents both these options.

All the functional and non-functional requirements for the Graph Redistribution Manager are defined in Table 4.2.

4.2 Extracted Data Storage

In order to create an efficient redistribution method, it is necessary to perform an effective manipulation of the extracted meta-information. To achieve that, the data must be properly stored. The purpose of this section is to find a suitable representation for the extracted data.

4.2.1 Storage

The extracted data produced by the Query Data Extraction part defined in 4.1 represent paths of graph traversals. Consequently, the nature of the data is also a graph, with elements of the original graph. Moreover, it is also necessary to know the whole topology of the original graph as the paths do not have to include all the graph elements[22].

For this reason, the extracted data should be stored in the whole graph as well-identified extra information of the original elements. As already mentioned in the previous section, two options of a connected graph database are possible. Neither of these options prevents the storage of the extra data unless they place substantial physical demands.

The specific product of a graph database will be chosen in the chapter Implementation 5.

[21]Note, that we use the term *path* in compliance with the TinkerPop framework as explained in section 3.2.1.2.

[22]The fact that some elements were not a part of the logs does not mean that these elements should be redistributed at random. They can be utilized in future queries, therefore, for elements not logged yet, the graph structure should be taken into account.

Requirement	Description
Functional Requirements	
Traversal Monitoring	It must be possible to monitor the traversed elements (meaning a path) of an OLTP graph traversal triggered by a user query.
Path Logging	The monitored paths[21] of individual traversals are logged into a separate file.
Format Setting	Supported format of logging can be set in a configuration file.
Anonymization	To enable logging from production databases, the code should be prepared for an option of logging an anonymized output.
Non-functional Requirements	
TinkerPop Compliance	As the number of TinkerPop-compliant graph DBMSs is significant, the solution should be a part of the TinkerPop framework in order to allow the data from any database supporting the TinkerPop to be logged.
Extensibility	It must be easy to add new features and customize the implementation. For example, it should be possible to easily implement and set another type of format than provided.
Documentation	The code of implementation should be properly documented including a README file.
Licensing	The additional implemented features must comply with the source licensing (as the TinkerPop Compliance is already stated, this means the Apache License 2.0).

Table 4.1: Functional and non-functional requirements for the Query Data Extraction

Requirement	Description
Functional Requirements	
Redistribution	Based on extracted data of past graph traversals and on provided distributed graph database, the program can offer a more efficient distribution of data.
Non-functional Requirements	
General Framework Preference	The implementation should be aimed at a general framework rather than a specific product in order to allow a wider range of databases to be employed (in graph databases the general framework is the TinkerPop). However, a particular system will be chosen; therefore, vendor-specific adjustments can be applied if necessary.
Interoperability	The database connected to the GRM system can either be a production database or another database containing a graph of the same topology with the same element IDs and partition information. The graph database with such a graph must always be provided.
Documentation	The code of implementation should be properly documented including a README file.

Table 4.2: Functional and non-functional requirements for the Graph Redistribution Manager

4.2.2 Model

Section 3.2.1.2 presents the definition of a *path*-step[23]. A path contains both vertices and edges. As discussed at the beginning of this chapter, the factor which we want to optimize to achieve better performance is the communication between partitions. This means that the elements we are interested in are edges. Note, that it is also possible to incorporate the information about vertices. This is discussed in chapter 7 of possible enhancements, yet not included in our implementation.

4.2.2.1 Edge Traffic Representation

The number of times a traversal walked an edge means the number of "jumps" (of network communication) from one partition to another when the edge

[23]As explained in 3.2.1.2 we use the term *path* which allows multiple occurrences of an element.

connects two vertices on different partitions. The more frequent traffic on the edge, the more likely we want the edge vertices (and the edge) to be allocated on the same partition. Accordingly, what we want to store is the information of traffic on each edge, which means the sum of times each traversal walked over the particular edge. Regarding the log file containing paths of traversals, it is the number of occurrences of the edge in that log file. Let us denote this number as a *weight* of an edge.

One way how to store the value of the weight is to create a property *weight* for each edge with a default value, for instance, equal to 1. However, this approach presents a serious limitation. A graph database can contain a schema which defines not only possible types of an edge but also a set of its permitted properties. This means that in order to add a new property of weight for redistribution purposes, the schema of all the edges would have to be altered (and for each new edge type it would have to be ensured that the weight property is added to its schema as well). If we wanted to remove the information for the redistribution, all the original edges would have to be modified again. For this reason, it is not the desired solution in the case of a configured production database.

Another possibility is to add a new edge of type T between the vertices incident with an edge that has been queried. The new edge would represent the original edge, and it would contain the property of weight (with default value equal to 1) and possibly a timestamp[24]. For each traversal on the original edge, the weight would be incremented on the new edge which represents it.

The redistribution algorithm would run

- on edges of type T
- on an original edge e if there is not an edge of type T representing the edge e (this would happen when the edge e was not included in the log files).

Nonetheless, duplicating all traversed edges could seem overly demanding regarding physical space. This model, however, can be reduced when all the edges between two vertices v and u are represented with only one new edge of type T. The explanation of why this approach works is split by possible types of a graph representation:

- *Edge data stored as a part of vertex data* – when an edge is stored as part of vertex data such as in JanusGraph 3.1.1, it does not matter if each edge is duplicated and its weight incremented or if there is one substitute

[24]The chapter 7 contains a discussion of incorporating fading into the redistribution algorithm. For such an enhancement, a property containing a timestamp would be necessary. This work, however, does not include fading.

for all of such edges. In both cases, all the edges are stored together, so when a graph is redistributed, and vertices of the substituting edge are allocated on different partitions, the separation concerns all of the edges alike.

- *Edge data stored separately* – when an edge is stored separately from vertex data such as in OrientDB 3.1.2, each edge between two vertices v and u can be theoretically stored on any partition. Using only one substituting edge of type T could thus seem insufficient. However, if we used only one edge e_t representing all these edges and the algorithm decided that its incident vertices should be allocated on separate partitions P^i and P^j, it is desirable that the edges between them are stored either on P^i or P^j as well, otherwise there would have to be performed more network communication.

  Therefore, in this case of storage, when "cutting" the substituting edge, the redistribution algorithm could redistribute its original edges either evenly between P^i or P^j or using any other approach. In either way, the usage of a substituting edge is possible.

- *Vertex data stored as a part of edge data* – in the case of vertex information kept within edge data (which is not implemented in any of the analyzed databases), the redistribution algorithm would be more suitable if it was creating a vertex cut. With the application of vertex cuts, the proposed model would be suitable with explanation analogous to the first case of edge data stored as a part of vertex data.

  With edge cuts, the model would still fit with the same explanation as in the previous case of edge data stored separately from vertex data. Nonetheless, again, there are no known databases storing the graph in this representation, so this option is unlikely to occur.

It has been shown that the proposed model is correct in various graph representations. Furthermore, with one edge containing minimum of data and representing all the edges between two vertices, the extra physical space necessary is not substantial. For these reasons, we choose this model to store the logged data of traversal paths.

4.3 Architecture

The purpose of this section is to provide a high-level architecture perspective. Individual details of some parts will be then presented in the following sections. As it emerges from the analysis above, the whole system contains two separate segments – the Query Data Extraction (also Logging in short) and the Graph Redistribution Manager – which need to interact. This interaction is shown in Figure 4.1. The parts that have to be implemented are highlighted with blue color.

Following the non-functional requirements of the Query Data Extraction, specifically the General Framework Preference, it is convenient to design this part as a new module of the TinkerPop framework. This approach brings the following advantages

- Vendor-agnostic implementation as a result of the GDBMS-independence property of TinkerPop itself. No matter which graph database is used, as long as it supports the TinkerPop, the logging extension can be applied.
- The code sustainability is simple as the module can be held in a separate fork of the original TinkerPop project. Most likely, the fork would have to contain few adjustments of the original files which could be a subject of a merging issue when accepting new changes from the original project. Nevertheless, predominantly, the main logic of the module exists independently in the fork without any influence of modifications in the original project.

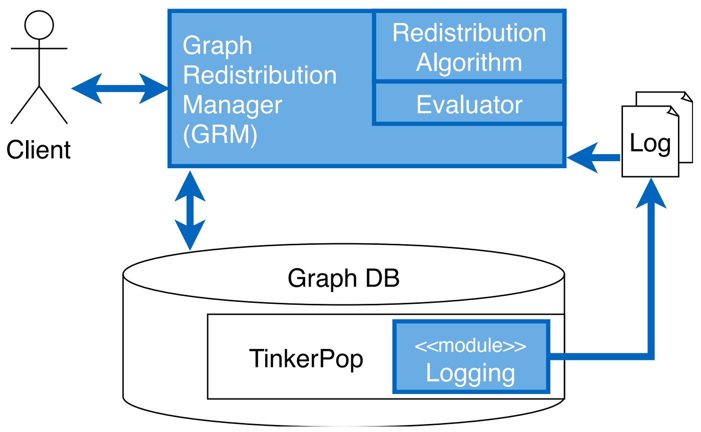
According to non-functional requirements of the Graph Redistribution Manager, it is required that the GRM can connect either to a production database or to another graph database containing a graph of the same topology (and other characteristics). The version with a directly connected production database is shown in the Figure 4.1a while the option with an image database can be seen in 4.1b.

From the figures, it is visible that thanks to the separate design of query logging, the difference between the two versions is not relevant. In both cases, the logging module must be a part of the production database. The Graph Redistribution Manager only reads the logs, which means that the loading part is independent of the database used entirely.

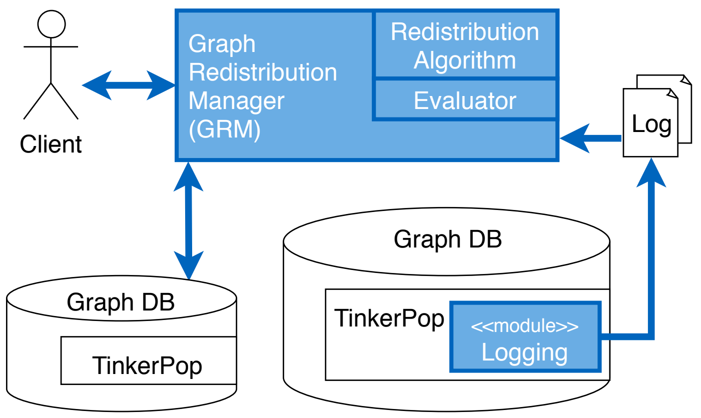
On the other hand, the GRM stores its necessary information in the provided database which can be either production or another one but has to support the TinkerPop framework. It is also likely that a specific type of a graph database will be necessary for the configuration because some specific adjustments will be inevitable as stated in the non-functional requirements of the GRM 4.2. Nonetheless, this vendor-specific restriction concerns only the part of the architectural design represented with the arrow directly from GRM to a graph database.

4.4 Graph Redistribution Manager

The purpose of the Graph Redistribution Manager is to produce a proposal for more effective distribution of data in a graph database based on logs with extracted data from past queries. The Figure 4.1 already shows that the GRM



(a) Version with a shared database.



(b) Version with a separate production database.

Figure 4.1: Architecture design. Blue elements denote parts that have to be implemented.

accepts log files, and communicates with a client and a graph database. It also presents two crucial parts of the GRM, a redistribution algorithm and an evaluator.

The evaluator is a piece of code that can assess a specific distribution based on provided log files. Therefore, it is utilized for comparison of the final proposed distribution and the original partitioning. It could also be used when monitoring the progress of the redistribution algorithm.

The design of the primary process of the GRM – data loading, running an algorithm and handling a result – is expounded in the activity diagram 4.2. The necessary conditions for a successful result (such as “log data with an expected format”) are omitted.

The action *Extract necessary data* is examined in detail in the previous

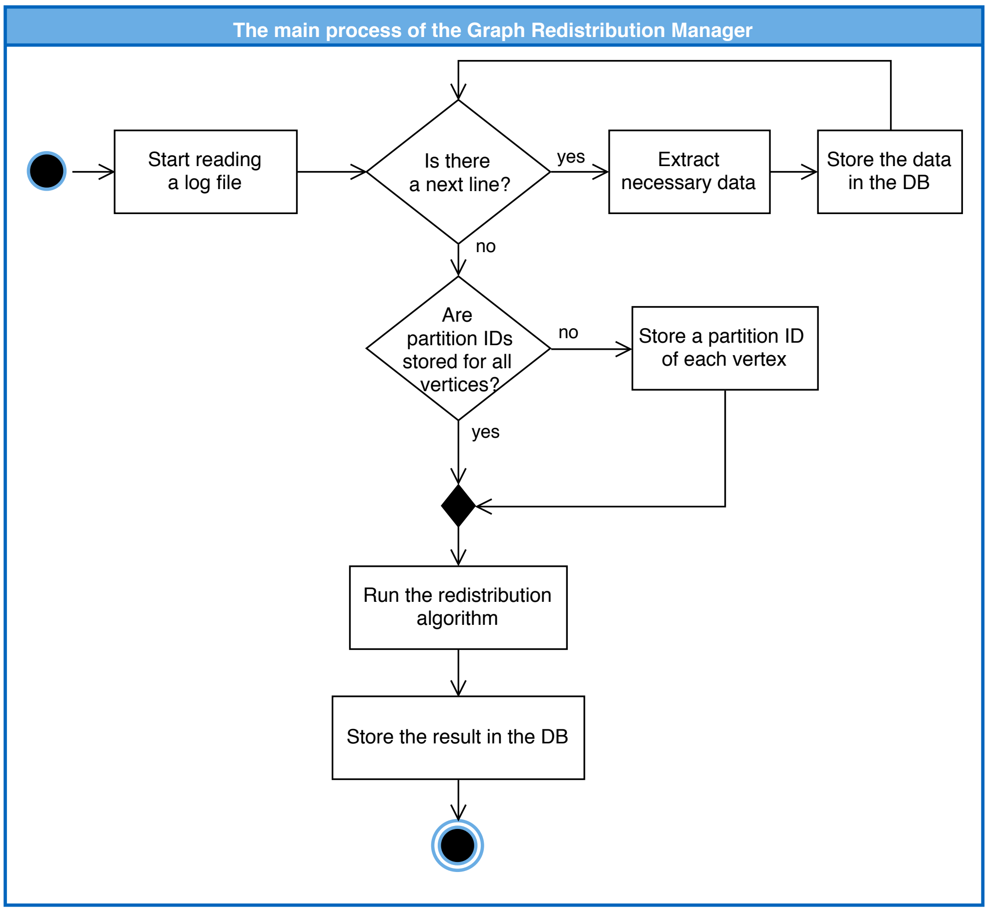

The main process of the Graph Redistribution Manager
Start reading a log file
Is there a next line?
yes
Extract necessary data
Store the data in the DB
no
Are partition IDs stored for all vertices?
no
Store a partition ID of each vertex
yes
Run the redistribution algorithm
Store the result in the DB

Figure 4.2: Activity diagram of the main process in the Graph Redistribution Manager

section 4.2. The diamond of "Are partition IDs stored for all vertices?" and its subsequent action "Store a partition ID of each vertex" is suitable for both architecture designs of Figure 4.1 – a production database will store information about the partition ID of each vertex while the image database contains it as already defined. In the case of a production database this action could seem unnecessary. Nonetheless, the explicit presence of partition ID as a property of a vertex allows better performance of the ensuing algorithm – if the redistribution algorithm had to always ask about a partition ID of each vertex, it could slow down the algorithm as its retrieval can involve computational steps, and this information is necessary very frequently. At the same time, the demands for physical space of the extra property are negligible.

4.5 Algorithm Design

Section 2.2 introduced the Pregel platform convenient for computational tasks on distributed graphs. Two of compliant algorithms were later presented, Vaquero et al. 2.2.1 and Ja-be-Ja 2.2.2. Their comparison is summarized in Table 2.1.

The conclusion of these methods could imply that the Ja-be-Ja algorithm offers more advantages than the Vaquero et al. algorithm. However, the computational demands are higher in the case of Ja-be-Ja. This section examines the possible enhancements of the Vaquero et al. method in order to be able to perform higher quality, yet to maintain the advantage of lower computational demands.

4.5.1 Balancing

Vaquero et al. constraints the label propagation with upper bound defined in Equation 2.2. This equation ensures that the number of vertices assigned to a partition never exceeds its capacity. Thus this constraint should be maintained.

Nonetheless, no lower bound is determined by the authors of Vaquero et al. As mentioned, the original paper [39] on which Vaquero et al. bases, applies both lower and upper bound equal to $\left\lfloor(1-f)\,\frac{|V|}{n}\right\rfloor$ and $\left\lceil(1+f)\,\frac{|V|}{n}\right\rceil$, with some fraction $f \in [0;1]$ and number of partitions n. With the usage of $\frac{|V|}{n}$, this approach expects all the partitions to have an equal size. This does not have to be the reality of course.

Therefore, we propose the lower bound for a partition P^i as

$$\left\lfloor(1-f)\cdot|V|\cdot\frac{C_i}{\sum\limits_{j\in J} C_j}\right\rfloor, \tag{4.1}$$

where f is some fraction $f \in [0;1]$, C_i is capacity of the partition P^i and $J = \{1, ..., n\}$, n equals the number of partitions.

The lower bound ensures that a partition will not be vacant. In other words, it contributes to a better balancing of the final result. On the other hand, if the fraction f was set too high, it could restrict the algorithm prematurely from adopting necessary labels, finishing with a worse outcome. For this reason, the effect of f will have to be thoroughly examined in the evaluation part of this work presented in chapter 6.

Incorporating both lower and upper bound should produce balanced results which will not always perform a perfectly accurate balancing (because of the fraction f) but will to some extent also allow more natural grouping of the elements. In other words, the adjusted version should evince both satisfactory balancing and elasticity.

4.5.2 Local Optima

Vaquero et al. do not confront the problem of the local minimum at all, although it involves their method as well. During the computational process, a vertex always accepts the most frequent label among its neighbors. To avoid the discussed oscillation a probability s is then applied to determine whether the label is actually adopted. The experimentally best value for this probability is equal to 0.5 as mentioned in 2.2.1.

It would be worth experimenting with the possibility of accepting a label different from the most frequent one, that is, a possibly worsening solution. For this purpose, we propose the incorporation of simulated annealing (also used in Ja-be-Ja) which was explained in section 2.1.4. With simulated annealing, it is possible to begin the algorithm with some probability of accepting a degrading solution and lowering this probability as the method progresses.

4.5.2.1 Label Selection

For the described purpose of label selection of a vertex v we propose the following equation

$$\left\lfloor r \cdot \frac{T}{T_{init}} \cdot \left| l\left(D_v\right) - 1 \right| \right\rfloor, \tag{4.2}$$

where r is a random number $r \in [0;1]$, T is a current temperature, T_{init} is an initial temperature, D_v is a set of neighbors of the vertex v and l is a function returning a set of labels of vertices in the provided set. In our case we lower the temperature T down to 0 (instead of 1 as in the method definition in 2.1.4).

If the number of different labels among the neighbors of a vertex was, for instance, 4, this equation returns one value from $\lfloor [0;1] \cdot [0;1] \cdot (4-1) \rfloor = \{0,1,2,3\}$. This value represents an index to the sequence of sorted labels with index 0 as the most frequent label and index 3 as the least common label among the neighbors. At the beginning of the simulated annealing, the fraction $\frac{T}{T_{init}}$ equals one, allowing to choose randomly any of the four values. However, as the temperature decreases, the fraction decreases as well, with, for example, $T = \frac{1}{2} \cdot T_{init}$ receiving the set of possible labels equal to $\left\lfloor [0;1] \cdot \frac{1}{2} \cdot (4-1) \right\rfloor = \{0,1\}$, and down to $T = 0$, that is $\lfloor [0;1] \cdot 0 \cdot (4-1) \rfloor = \{0\}$ – representing the most frequent label and original approach of Vaquero et al.

4.5.2.2 Probability of Adoption

The probability s of adopting a selected label to avoid oscillation is originally always equal to 0.5. However, with the approach of simulated annealing, the probability of oscillation is already reduced with the random selection of neighboring labels. Therefore, it is reasonable to alter the probability of oscillation s as well.

The lower the temperature, the lower the probability of choosing a different label. Accordingly, the proposed probability s of label adoption is

$$s = S + \frac{T}{T_{init}} \cdot (1 - S), \tag{4.3}$$

where S is a constant corresponding to the resulting probability when the temperature reaches 0. As already mentioned, S is equal to 0.5.

With this approach, at the beginning of the algorithm with $T = T_{init}$, the label is always accepted as $s = 0.5 + 1 \cdot 0.5 = 1$. As the computation progresses, the lowering temperature also lowers the probability of adoption, resulting in $T = 0$, that is, the original $s = 0.5 + 0 \cdot 0.5 = 0.5$.

4.5.3 Weighted Graphs

To adjust the Vaquero et al. algorithm for a weighted graph, the approach of Ja-be-Ja is adopted. In section 2.2.2.4 the equation 2.6 was introduced, adjusting the number of neighbors of a given color to the sum of the weights on the edges connecting these neighbors with the original vertex.

This means that, even if there were, for example, two neighbors with an identical label and the sum of the weights on the edges connecting them with the original vertex equaled 4, one neighbor with another label but the sum of the weights equal to 6 would be ordered as more frequent – because based on the traffic represented with weights, it is indeed busier.

We started with the comparison table 2.1 containing two advantages and three disadvantages of the Vaquero et al. algorithm and three advantages and two disadvantages of Ja-be-Ja method. Adjusting Vaquero et al. algorithm with the proposed enhancements, we try to mitigate all the limitations from the table – improving partition balancing, confronting local optima and incorporating weighted graphs. Chapter 6 will examine the effects of the proposed parameters as well as an overall success rate of this modified algorithm.

Chapter 5

Implementation

This chapter presents the implementation part of this thesis that was designed in previous chapter 4. Based on the technology described in chapter 3 it first selects a specific graph database for the storage of extracted data. Subsequently, it introduces the implementation of the query data logging which is accomplished as a module of the TinkerPop framework. Last but not least, the inner structure of Graph Redistribution Manager is explained.

5.1 Graph Database Selection

Given the discussion in 4.2.1, a distributed graph database must be selected as a storage for the extracted data (either if it is directly a production database or an image of such database). The chosen solution should be an open-source project in order to be extended if necessary. Following the non-functional requirements of the Graph Redistribution Manager set in Table 4.2, the selected product should support the TinkerPop framework in order to increase the potential reusability of the implementation.

Based on the preceding analysis of 3.1 and just mentioned requirements, two databases, JanusGraph and OrientDB, can be taken into consideration as they are both a TinkerPop-compliant open-source product.

5.1.1 TinkerPop Support

Regarding the TinkerPop framework, JanusGraph implements the TinkerPop support from its origin as its predecessor Titan did, because it utilizes the Gremlin Traversal Language as the only query language for its manipulation. OrientDB, on the contrary, supports the TinkerPop framework since version 3.0.0 released in April 2018 [50] as it is instead focused on its SQL interface.

5.1.2 Other Characteristics

Whereas JanusGraph is a graph database, the OrientDB is a multi-model database that can store more types of data than graphs. The main difference between JanusGraph and OrientDB graph functionality is a different model for its data storage. JanusGraph applies adjacency lists, whereas OrientDB keeps its graph elements separately in individual classes `Vertex` and `Edge`. Nevertheless, both approaches do not prevent the proposed model of extracted data to be employed as examined in 4.2.2. For this reason, the difference in the model should not play a decisive role in the GDBMS decision.

Regarding the distribution, JanusGraph uses external storage backends which take care of the distributed environment. OrientDB, on the other hand, provides its specific database solution. Accordingly, it also manages the distributed arrangement. However, its distributed organization still suffers from several problems as seen in open issues on the GitHub page of the OrientDB [74].

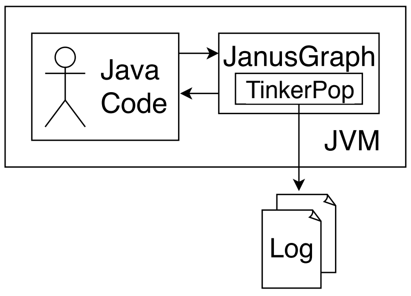
Furthermore, not only has OrientDB some troubles with distribution, but its relatively new TinkerPop implementation is also still imperfect, besides others suffering from issues with concurrent Gremlin calls or not yet supporting all the TinkerPop features [74].

To sum it up, if we need an open-source distributed graph database which supports the TinkerPop framework, we can conclude that in order to avoid unnecessary problems and potentially a dead end, the currently more suitable solution is to select the JanusGraph database.

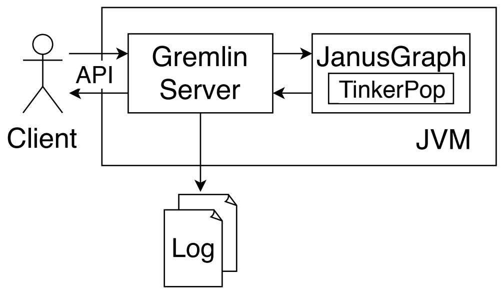
5.2 Query Data Extraction

In the description of JanusGraph 3.1.1 five different connection modes were presented. The logging can be applied to all of them. However, to explain the differences, the five modes can be generalized into two main arrangements:

- **local mode** – when the client operates in the same Java Virtual Machine (JVM) as the JanusGraph, the logging is performed as part of the TinkerPop implementation of JanusGraph. This is shown in Figure 5.1a. In order to enable the logging functionality, the client code must explicitly call the logging strategy.
- **server mode** – when the Gremlin server is executed to enable instant access to the graph database, the logging can be carried out
  - by TinkerPop implementation of JanusGraph as already shown in Figure 5.1a. A client can explicitly create a command to use the query data logging feature. This command is sent to the Gremlin server which provides it to JanusGraph.



(a) Local mode.



(b) Server mode.

Figure 5.1: Query Data Extraction in different connection modes.

- by Gremlin server as shown in Figure 5.1b. The Gremlin server can also be configured to log the processed query data. Such server setting allows logging all the necessary information without a client notice. Therefore, this approach is convenient when logging the data from the production database, for example.

As designed in Architecture section 4.3, the Query Data Extraction is implemented as a separate TinkerPop module. Some minor adjustments in the original files had to be performed, hence these are introduced first. Subsequently, the logging module itself is described. The implementation is publicly available on GitHub web page [75]. The cited project is a fork of the original TinkerPop project, version 3.3.3, with our alterations. The version 3.3.3 has been chosen because it is the officially compatible version of the utilized JanusGraph version 0.3.0 [76] (although version 3.3.4 of Apache TinkerPop is already available). This is also the reason why all the changes are part of the `processed-result-log` branch.

5.2.1 TinkerPop Modifications

In sum, four original files had to be adjusted to enable the query data logging. Two of these files are just POM [25] files adding the new logging module into the TinkerPop project. The other two files are modified to enable the logging from Gremlin server. The server logging can be set in the server configuration file `gremlin-server.yaml`.

- **pom.xml** – the only adjusted line adds the logging module as part of the whole TinkerPop project

```
<module>proceesedResultLogging-gremlin</module>
```

[25]POM stands for Project Object Model. *pom.xml* files contain configuration and other information about a Maven project.

- **gremlin-server/pom.xml** – in order to enable "silent" server logging as in Figure 5.1b, the Gremlin server must depend on the logging module to be able to call its functionality.

```
<dependency>
   <groupId>org.apache.tinkerpop</groupId>
   <artifactId>processedResultLogging-gremlin</artifactId>
   <version>${project.version}</version>
   <optional>true</optional>
</dependency>
```

- **gremlin-server/(...)/Settings.java** – the Settings file manages the server settings configured by the yaml file. Accordingly, the settings for logging have to be added and registered to allow its configuration there.

```
public ProcessedResultManager.Settings processedResultLog =
    new ProcessedResultManager.Settings();
(...)
final TypeDescription PRLSettings = new
    TypeDescription(ProcessedResultManager.Settings.class);
constructor.addTypeDescription(PRLSettings);
```

- **gremlin-server/(...)/AbstractEvalOpProcessor.java** – processing operations dealing with script evaluation functions, the `AbstractEval-OpProcessor` contains decisions whether to trigger additional operations based on settings from the configuration file. Therefore, the following condition has been added

```
if(settings.processedResultLog.enabled){
   ProcessedResultManager.Instance(settings.processedResultLog)
   .log(script, itty);
}
```

5.2.2 Logging Module

The TinkerPop project contains several basic Gremlin modules with *gremlin* prefix (such as the `gremlin-server` or the `gremlin-core`), and then modules with specific modifications to enable the usage of third-party products (such as the `hadoop-gremlin` or the `neo4j-gremlin`). Accordingly, the whole module representing the additional intrinsic behavior of TinkerPop is called `gremlin-processedResultLogging`. The name intentionally does not contain the word *query*, because the module was written in a general way to be able to log any additional operation performed on a resulting traversal. One

```
tinkerpop ........................................ tinkerpop project v3.3.3
└─ gremlin-processedResultLogging ........................... the module
   ├─ src/main/java ............................ implementation sources
   │  └─ org.apache.tinkerpop.processedResultLogging
   │     ├─ ProcessedResultManager ................ the managing class
   │     ├─ processor ............................... processor package
   │     │  ├─ Result Processor ....... interface for a result processor
   │     │  └─ specific processors
   │     ├─ result ...................................... result package
   │     │  ├─ ProcessedResult ..... abstract class for processed result
   │     │  └─ specific result types
   │     ├─ formatter ............................... formatter package
   │     │  ├─ ProcessedResultFormatter .... interface for a formatter
   │     │  └─ specific formatters
   │     ├─ util .......................................... util package
   │     │  ├─ ObjectAnonymizer ... edge/vertex output anonymization
   │     │  └─ SimpleLogger ................ logger for the local mode
   │     └─ LogPathStrategy .................... the logging strategy
   ├─ pom.xml ............................. Project Object Model file
   └─ README ............................................ README file
```

Figure 5.2: Structure of the Processed Result Logging module

of that possibilities and already an implemented extension is logging of the traversed paths (that is, providing the data about a query processing).

The description for users and developers of how to use the logging module (for example how to use different options of settings or how to implement a new extension) is provided in the README file available either on the mentioned GitHub page [75] or as a part of the Appendix A.1, thus narrowing the explanation in this section solely to code description.

The TinkerPop project incorporates the Apache Rat Plugin [77] (the Release Audit Tool) which prohibits a successful compilation unless the whole code is provided with the required copyright policy. For this reason, each new file of the logging module begins with a disclaimer of licensing the file under the Apache License, Version 2.0.

A simplified structure of the module is depicted in Figure 5.2. Apart from the mentioned *pom* and *README* file, the module contains the source code divided into six groups which description follows.

5.2.2.1 Processed Result Manager

`ProcessedResultManager` is a singleton[26] class managing the whole process from acquiring settings, creating a processed result to logging it.

It contains an inner class `Settings` declaring six keys that can determine the logging behavior. Their specification and default values can be found in the README file A.1.

The main function of the `ProcessedResultManager` is logging, which first checks the correctness of the settings, throwing appropriate exceptions when necessary.

If the settings do not cause any exceptions, a processor is called to process the result which is later logged with a logger. The logger is an implementation of the *SLF4J*[27] Logger.

In the case of server logging, the processed result logging is part of all other standard Gremlin server logs (using Log4J) to follow the current logging logic of the server. Thanks to a specific logger name, the processed result logging can be still set into a separate file. In the case of local mode, a `SimpleLogger` 5.2.2.6 is utilized.

5.2.2.2 Result Processor

The interface `ResultProcessor` serves for processing an original result into a `ProcessedResult` 5.2.2.3. It declares a method `process` and possible exceptions for cases when a traversal does not provide the metadata about a path (the query was, for example, insertion of data) or when a query does not use a traversal at all (for example when creating a traversal on a graph). These exceptions are, however, sometimes expected to occur as not all of the queries necessarily use traversals or produce metadata. Therefore, these exceptions are logged as warnings only.

The `process` method of the specific `PathProcessor` – the only processor implemented – contains the following steps.

1. It takes an input – a traversal of a result.
2. It creates a copy of the traversal in order not to influence the original traversal which is further used by the TinkerPop (written out to a console).
3. It adds an additional operation `path()` to its end unless it already is a traversal called with the `path` operation. The extra `path` function

[26]Singleton is a design pattern of a class containing only one instance in the whole program that can be globally accessed.

[27]*SLF4J* stands for Simple Logging Facade for Java enabling determining the logging backend at runtime. The TinkerPop uses as the specific logging backend the *log4j* utility.

retrieves metadata containing the whole path a traversal walked as explained more in detail in 3.2.1.2.

4. It iterates over the altered traversal, that is, over the elements of the walked paths, and creates the final result out of them.

5. It returns the `ProcessedResult` containing list of elements of the path.

Another interface, `AnonymizedResultProcessor`, extending the `ResultProcessor` with a method `processAnonymously` serves for explicit division of processors creating anonymized and only original processed results. The `PathProcessor` is an implementation of the `AnonymizedResultProcessor`, which in its `processAnonymously` method anonymizes all elements becoming a part of the final result (removing any textual context).

5.2.2.3 Processed Result

The `ProcessedResult` is an abstract class representing a result of a `ResultProcessor` containing an output of the additional operation run on an original result. The class contains, for example, a private Object `result` and an abstract method `toString`.

A specific processed result can be so far a list of lists of objects, `List<List<Object>>`. The particular extension of the `ProcessedResult` must also provide implementation of the `JsonSerializer` and its method `serialize` which converts the result into a JSON[28] object, later stored into a log file. The resulting format is shown in the following section 5.2.2.4.

5.2.2.4 Formatter

The `ProcessedResultFormatter` serves for formatting a `ProcessedResult` into the final log string. Therefore, the interface `ProcessedResultFormatter` declares a method `format`, with parameters of a `ProcessedResult` and a `String` representing the original query of the result. The concrete implementation of the `ProcessedResultFormatter` can but does not have to use the query in its output.

Presently, three formats are available

- **Basic Formatter** – the basic formatter logs only the result in its original form (using the method `toString` of the `ProcessedResult`).

- **Query Formatter** – logs both the query and the result, unchanged, as

```
#QUERY:
  g.V(1).outE('knows').inV().values('name')
#PR:
```

[28]JSON stands for JavaScript Object Notation.

```
v[1],e[7][1-knows->2],v[2],vadas
v[1],e[8][1-knows->4],v[4],josh
```

- **JSON Formatter** – logs the data in JSON format. Its output can look like

  ```
  {
     Q:"g.V(1).outE('knows').inV().values('name')",
     R:[
        [{"v":1},{"e":7},{"v":2},{"string":"vadas"}],
        [{"v":1},{"e":8},{"v":4},{"string":"josh"}]
     ]
  }
  ```

  For that purpose the `JsonFormatter` uses the open-source GSON library by Google [78]. This is also the reason why the `ProcessedResult` must implement the function `serialize` of `JsonSerializer`.

5.2.2.5 Helpers

Two helper classes have been created in order to make the code well arranged and their functionalities reusable.

- `ObjectAnonymizer` – creates an anonymized version of an object. So far it can anonymize a graph element – an edge or a vertex. It removes all the information except an element ID (which means that if an ID was explicitly assigned and contains any information, it can still be exposed – this warning is part of the code).

- `SimpleLogger` – is an implementation of the SLF4J interface `Logger`. It serves for logging in the local mode when the only records logged are processed results (in comparison with Gremlin server logging which is a part of all other log records of the server to follow the conformity of the server behavior.). It is a simple logging into a file without any extra overhead, but as it implements the SLF4j interface, it can be easily exchanged to a different implementation as well.

5.2.2.6 Logging Strategy

In comparison with the Gremlin server, in which a result is explicitly processed in a corresponding part of the source code, the local processing does not contain any suitable single point of interference. Therefore, in order to enable logging in the local mode, a Traversal Strategy has to be written to call the logging on each result.

"A `TraversalStrategy` analyzes a `Traversal` and, if the traversal meets its criteria, can mutate it accordingly" [69]. A particular strategy as such can provide extra decoration, optimization, provider optimization, finalization or verification functionality. A strategy we want to create can be categorized into *finalization* strategies as it performs final adjustments of a traversal [69].

When creating a traversal source `g` as `g = graph.traversal()` which is further used for vast majority of queries, a method `withStrategies` can be called, registering strategies later used on each traversal.

Our `ProcessedResultLoggingStrategy` primarily calls the logging of the `ProcessedResultManager` on each traversal. Last but not least, it also prevents the recursive calling of the strategy on cloned traversals with adding a marker to the original traversal.

5.3 Graph Redistribution Manager

The Graph Redistribution Manager performs the main steps presented in its design in diagram 4.2 – it reads a log file, stores partition IDs to vertices if not yet present, runs the redistribution algorithm which it implements and stores its result (proposal of a more efficient distribution) in the database.

Apart from that, it can also load a dataset and store it in the configured database. Then it can create queries according to access patterns described in the following chapter of Evaluation 6. It can run these queries, which generates the necessary log files used as the input to the primary part of the GRM.

The GRM is written in Java language as a Maven project in conformity with the TinkerPop project so that the potential developers – most likely users of the TinkerPop framework – would know the language and the building tool well. The GRM project can be publicly found on the GitHub web page [79].

The project structure is depicted in Figure 5.3. Individual classes and packages are further described in separate sections.

5.3.1 GRM.java

For evaluation purposes of chapter 6, two datasets are applied. This usage of two datasets, both of which need several specific adjustments, lead to an explicit division of the main class `GRM` into two classes `GRMT` and `GRMP` for Twitter and Pennsylvania road map dataset, respectively. Consequently, the `GRM` is an abstract class containing shared functionalities for the two subclasses such as connecting to or injecting data into the graph.

This structure is very convenient for the current extent of the program with the necessity of a significant number of executions of a given dataset but if more datasets were about to be utilized, the `GRM` would be a subject of refactoring.

```
src/main ........................................ application directory
├── java ...................................... implementation sources
│   ├── GRM.java ............... abstract class with shared functionalities
│   ├── GRMT.java ................... executable class for Twitter dataset
│   ├── GRMP.java ............. executable class for Pennsylvania dataset
│   ├── logHandling ........................... log loading and storing
│   ├── dataset ....................... dataset loading, query creation
│   ├── cluster ................................. partition ID retrieval
│   ├── partitioningAlgorithms .............. redistribution algorithms
│   └── helpers ......................................... helper classes
└── resources ............................ implementation resources
    ├── config.properties ........... configuration of a graph db choice
    ├── graph-berkeley-embedded.properties . janusgraph config for the
    │   Berkeley DB
    ├── graph-cassandra-embedded.properties janusgraph config for the
    │   Cassandra
    ├── cassandra.yaml ....................... Cassandra configuration
    └── datasets ........................................ datasets used
├── pom.xml ................................. Project Object Model file
└── README ............................................... README file
```

Figure 5.3: Structure of the Graph Redistribution Manager project

5.3.2 LogHandling

The package `logHandling` contains tools for manipulation with the log that was generated by the Logging module. The classes could be categorized according to two objectives

- *loaders* – serve for loading the data from a log file and for subsequential storage of these data in the graph database.

  The `LogFileLoader` loads a log file with an iterator that iterates over individual records, each containing paths produced by one traversal. Another loader, `PRLogToGraphLoader` is then implemented to store these results in the graph database. The `PRLogToGraphLoader` is a general interface that is implemented in `DefaultPRLogToGraphLoader` realizing the model of extracted data storage designed in 4.2.2.

- *representing objects* – the log contains information that can be subdivided into objects of more levels of graininess. Each log record represents all the paths that were generated by one traversal. This information is represented with the `PRLogRecord` class. Individual paths of the traversal are encapsulated into the `PRPath` object containing elements of the finest granularity, the PRElements – standing for particular edges and vertices of the graph.

5.3.3 Dataset

The dataset package includes the functionality for loading the datasets, creating their query and running them in order to generate the log files. The interfaces DatasetLoader and DatasetQueryRunner are introduced with specific implementations for the particular datasets.

5.3.4 Cluster

Each vertex must possess the information about its original partition ID. This explicit property (as discussed in 4.4) can either be already provided in the database or has to be yet added. The interface PartitionMapper serves for this purpose, offering a mapping function for partition ID retrieval.

In the evaluation part we mimic the real distributed environment as the only information we need from it is the partition ID of each vertex. As the datasets already contain consecutive IDs of its elements, we take these IDs and return modulo *number-of-nodes-in-cluster* as their partition ID. This operation is provided in DefaultPartitionMapper implementing the PartitionMapper interface. The discussion of how to obtain the real values of partition IDs in a genuine distributed graph database is provided at the end of this chapter in section 5.3.8.

5.3.5 Partitioning Algorithms

The package partitioningAlgorithms currently contains one class VaqueroVertexProgram. It implements the VertexProgram of the TinkerPop framework introduced in section 3.2. It provides the proposed Vaquero et al. algorithm presented in 4.5.

After initial setup, the execute method is called on each vertex of the graph. It contains the main decision of what label to adopt abiding by the constraints of upper bound, lower bound and the probability of adoption. At the end it sends information about its label value to all of its neighbors using the local MessageScope[29].

The execute method runs on each vertex in parallel. After an iteration, a synchronization barrier represented with terminate method is applied to decide whether the algorithm should continue or halt. It stops either if all the vertices voted to halt or a predefined maximum number of iterations was reached, set to 200.

[29]MessageScope is a class containing identification of message receiver vertices in the VertexProgram.

5.3.6 Helpers

The `helpers` package contains two helper classes. The `HelperOperators` class implements the `BinaryOperator` interface, defining binary operators for reduction of the data in the `VaqueroVertexProgram`.

The second class `ShuffleComparator` implements the `Comparator` interface. It is utilized only in the Road network dataset. Specifically, it is used for traversal shuffling when implementing a random walk on the graph.

5.3.7 Resources

The file `config.properties` serves for setting the *graph.propFile* property. It should contain the name of a file with the specific backend properties. Presently, two options are available: `graph-cassandra-embedded.properties` and `graph-berkeley-embedded.properties`. These specific property files contain standard properties of the corresponding backend storages for the JanusGraph. In the case of Cassandra backend usage, the JanusGraph further supplies the Cassandra with its configuration file `cassandra.yaml`.

The directory `datasets` contains the original data [80, 81]. In both cases, the data are provided as a set of pairs, each pair representing an edge with an outgoing and an incoming vertex, respectively.

5.3.8 Real Distributed Environment

The purpose of this last short section is to discuss the retrieval of physical node IDs in a real distributed environment of the JanusGraph with Cassandra backend and the current obstruction of its physical redistribution.

5.3.8.1 Identification of a Physical Node of a Vertex

In order to manage its attached backend storage, JanusGraph utilizes the `StoreManager`, an interface which can be accessed via

```
StandardJanusGraph.getBackend().getStoreManager().
```

Some implementations of the `StoreManager`, such as `CQLStoreManager`, keep an instance of the class `Cluster` from the Cassandra driver package. This class enables the access to the `Metadata` class which can retrieve all replicas of a given partition key[30] and keyspace.

However, the `Cluster` instance has a private access from the `CQLStoreManager` class. Therefore, in order to operate with the `Cluster` instance, the current implementation of the `CQLStoreManager` would have to be extended with the method `getCluster`.

[30] The partition key is the information according to which an element is assigned to a physical node. In Cassandra, it is the first column of the primary key [82].

Call of the partition ID retrieval could then look in a following manner:

```
((CQLStoreManager)graph.getBackend().getStoreManager())
   .getCluster().getMetadata().getReplicas(keyspace,partitionKey)
```

In general, implementing the physical node information retrieval depends on the concrete implementations of the JanusGraph backends. Their connection modes also influence such retrieval because different connection modes of the same backend can have more than one `StoreManager` implementations in the JanusGraph. Moreover, the JanusGraph architecture is not initially designed to support this feature since it encapsulates the `StoreManager` and hides these lower level features.

5.3.8.2 Physical Partitioning of Data

Originally, JanusGraph supported explicit partitioning on a physical level by using lexicographic ordering of IDs. This ordering had to be supported on the backend level (Cassandra configured with the `ByteOrderPartitioner`). It means that JanusGraph was able to directly assign the partitions using the layout of the JanusGraph record ID. This layout had a 64-bit long representation having the following structure

```
[ 0 | PARTITION | COUNT | PADDING ].
```

Note, that the PARTITION part of the ID is at the beginning of the ID, which means it has the most significant vote in the lexicographic ordering. As a result, it affects the final partition of the record most substantially.

The layout above has been, however, changed, now having this form

```
[ 0 | COUNT | PARTITION | PADDING ].
```

With this change, the ability to manipulate the ordering was lost because the PARTITION part of the ID has now a lesser vote in the ordering and the COUNT part cannot be modified.

The current JanusGraph documentation presents a whole chapter on how to implement the explicit partitioning [83]. However, after the layout change, the content of this chapter is no longer valid.

Chapter 6

Evaluation

This chapter presents the evaluation of the proposed enhancements of the Vaquero et al. algorithm introduced in 4.5. In order to do so, the implementation of 5 has been carried out, available on the GitHub web page [75, 79].

First, the applied datasets are introduced together with user queries that have been generated for them. Subsequently, the evaluation methods are presented followed with settings description and finally the results themselves. At the end of the chapter a summary comparing our proposed method with another described Pregel-like algorithm, Ja-be-Ja, is provided.

6.1 Datasets

In order to examine the implemented program, two different datasets were utilized. They vary significantly in their characteristics, and both represent real-world problems. First, data of Twitter [80], a social network for posting and reading messages were examined. For the second case, the road network of Pennsylvania [81] was selected.

For each dataset, its number of nodes, edges, average clustering coefficient, and diameter is provided. A clustering coefficient of a vertex expresses connectivity of its neighbors (equal to 1 when forming a clique[31] with its neighbors, equal to 0 if none of its neighbors are connected). A diameter of a graph is the maximum eccentricity[32] of its vertices.

It is not uncommon for graph repartitioning problems to emulate smaller environments for evaluations. For example, authors of the *Ja-be-Ja* algorithm [28] described in section 2.2.2 used datasets of sizes up to 63,731 ver-

[31]Clique is a subgraph in which each vertex is connected to all other vertices of that subgraph.

[32]An eccentricity of a vertex is its maximum distance to any other vertex in the graph while the distance between two vertices is the minimum length of paths connecting them. This time we refer to the term *path* from rigorous graph theory.

tices and 817,090 edges. Their Twitter dataset contains only 2,731 entities and 164,629 relationships.

Our implementation enables partition allocation mimicking (as described in 5.3.4), so it also allows datasets of nearly any size to meet capabilities of our physical sources without any bearing on the quality of results. We decided for datasets containing 76,268 vertices with 1,768,149 edges (Twitter dataset) and 1,088,092 vertices with 1,541,898 edges (Pennsylvania road map dataset).

For both of the datasets (as well as for the majority of generally available datasets), there do not exist any attached queries. Authors of papers usually create artificial logs, use only structure of the graph (either without weights or with artificially generated weights) or do not state in any way how they obtain the initial state on which they base their computations. We define the behavior of a typical user for each dataset according to which we generate corresponding queries. These emulated queries are then executed in the JanusGraph with the new TinkerPop module which triggers the logging extension producing query logs.

For the social network dataset we create $|V|/8 \approx 9,500$ queries, and for the road map dataset, we generate a comparable number of $|V|/100 \approx 10,000$ queries.

Both datasets are described in more detail below together with their access patterns applied for the generated queries.

6.1.1 Twitter

The Twitter dataset is obtained from [80] gathered by crawling the public sources. Only files with the graph structure were utilized, ignoring the data with actual user data (that is, properties of nodes).

The dataset represents data of the Twitter social network where users “tweet” – post short messages called “tweets” – and read these posts from other users. A user can “follow” other users so that their messages would appear to the user when signed in the application. The relationship is not reciprocal, meaning a user does not have to follow the user who follows him.

In other words, a node represents a user, an out-going edge is a relation of “follows” and an in-coming edge means an “is being followed” relationship. The dataset statistics are stated in Table 6.1.

Graph 6.1 contains the information about distribution of vertex degrees – overall degrees, in-degrees and out-degrees. Note, that the y-axis has a log scale. The majority of vertices has a lower degree, but there also exist individual nodes with a very high degree. The number of vertices grows exponentially with a lower degree. This power-law behavior[33] of degree distribution is called

[33]Power law is a relationship in which one variable varies as a power of another variable.

Property	Value
Nodes	76,268
Edges	1,768,149
Average clustering coefficient	0.5653
Diameter (longest shortest path)	7

Table 6.1: Twitter dataset statistics [80]

the *scale-free network* and is common for social networks, World Wide Web network or airline systems [84].

6.1.1.1 Queries

For the first dataset of the Twitter social network, two access patterns have been devised.

- A typical Twitter user signs in the application and reads posts of people he or she follows. This basic and probably the most frequent behavior could be expressed as selection of a vertex and traversing all of its out-going neighbors. In other words, the breadth-first search[34] uncovering the first layer of nodes on out-going edges.
- This basic behavior is extended with another type of user action – searching for new people to follow. To improve the probability of finding relevant users, the search is defined as looking for people who are followed by people being followed by the user. For instance, if Luke Skywalker followed Obi-Wan Kenobi who followed user Yoda it would be likely that Luke also wants to follow Yoda.

As already mentioned at the beginning of this chapter, number of queries equals to $\frac{|V|}{8}$ which brings about 9,500 queries. Starting vertices of the queries are chosen at random (using class `Random` of the Java library *java.util*). Every time a vertex is selected at random, it must fulfill the condition 6.1

$$\frac{log(max(\{2; out\_degree(v_i)\}))}{log(max\_out\_degree)} > r, \tag{6.1}$$

where r is a random number from $[0; 1]$, max is a function returning the maximum value from the set, out_degree is a function returning an out-degree of the provided vertex and max_out_degree is a maximum out-degree present in the graph. The idea is that users who do not follow larger number of people,

[34] The breadth-first search is a graph traversal method during which a starting vertex is selected, and then the graph is explored “layerwise” – uncovering all the neighbor vertices in layer 1, after that uncovering all neighbors of that neighbors in layer 2, et cetera.

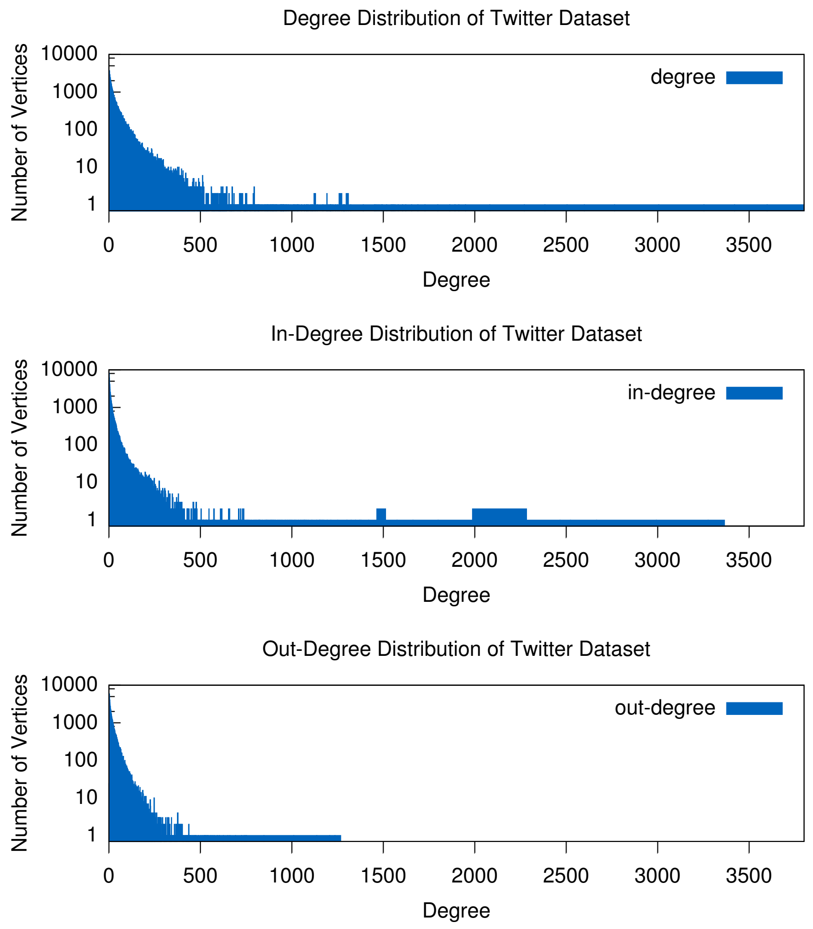


Figure 6.1: Degree distribution of the Twitter dataset

do not use the Twitter application as frequently as users following a significant number of people.

Simultaneously, if the condition was simply based on $\frac{out\_degree(v_i)}{max\_out\_degree}$ fraction, the probability of accepting a vertex with a low out-degree would be extremely small, for example, $\frac{1}{1251} \approx 0.08\%$. For this reason, a logarithm function is used to slightly reduce this discriminating effect, with $\frac{log(max\{2;1\}}{log(1251)} \approx 9.7\%$.

This procedure of starting vertex selection is repeated until the required number of starting nodes (that is, the number of queries) is reached.

The probability P_{ik} of unfolding out-going neighbors of a *k-th* neighbor of vertex v_i is defined as

$$P_{ik} = \frac{average\_degree}{average\_degree * out\_degree(v_i)} \tag{6.2}$$

with the underlying idea that users who do not follow larger number of people have higher probability that they want to find more users they could follow.

Property	Value
Nodes	1,088,092
Edges	1,541,898
Average clustering coefficient	0.0465
Diameter (longest shortest path)	786

Table 6.2: Road network dataset statistics [81]

To restrict this searching behavior a maximum number of unfolded neighbors is set, equal to $\frac{|V|}{8}$ vertices.

6.1.2 Road Network

The Road network dataset is taken from [81]. It represents the road map of the state Pennsylvania, USA. Endpoints and intersections are represented as vertices, while the roads are interpreted as undirected edges (which are provided as two directional edges of opposite direction).

The statistics about this dataset are stated in Table 6.2. In comparison with the Twitter dataset, the Road network contains approximately 13–times as many nodes, but a comparable number of edges. This means that the average vertex degree of the datasets differs significantly.

The Graph 6.2 provides more detailed information about overall degree of vertices in the graph. As the edges are unidirectional, the out- and in-degrees are not provided. It is also the reason why the degrees are only even numbers.

The graph differs from the previous Twitter dataset immensely. There are only nine different degrees in this graph, while in the Twitter graph there are 3,769 different degrees. In the previous dataset, the degrees evince a specific distribution, whereas in the Road network data, we can rather discuss the particular degrees than try to deduce a distribution with so few x-axis values.

Note, that the y-axis has a log scale. This means that the road map contains few big cities (or intersections), but the vast majority of these crossroads is of a smaller size, with a lower amount of roads connecting them with other roads.

6.1.2.1 Queries

One of the first ideas of a typical user behavior on a road map could be finding the shortest path between two points. The general task of the shortest path, however, is an OLAP query for which the logging module is not intended as explained in section 4.1. Therefore, a modified behavior has been created. A user searches for the shortest path between two points of maximal distance equal to 10.

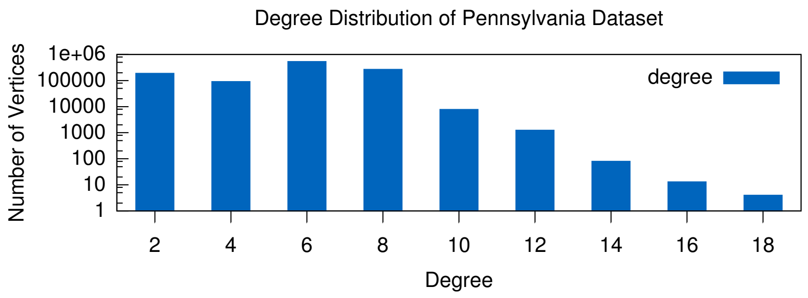


Figure 6.2: Degree distribution of the Road network dataset

First, a vertex is selected randomly. Similarly to the Twitter dataset, it must fulfill the condition 6.1 to be included in the set of starting vertices.

Subsequently, an ending vertex is found with the application of a random walk from the starting vertex. The length of the random walk is chosen randomly from the range $[5; 10]$ and it is not allowed to follow a vertex that it has already traversed (in fact, following the rigorous graph theory, we create a random path). With such a setting, it can happen, that a random walk of a given length cannot be found because of a dead end. Such an eventuality is omitted, therefore, the actual number of queries is lowered to about $8,000$.

The maximum length of such random walk is set to 10 – taking into consideration measurements discussed in Google Groups of Gremlin users [85] showing the computational demands of such a search, and number of the following tasks of finding the shortest path.

With the knowledge of the existing distance between the two nodes, the search of the shortest path can be restricted to the length of the corresponding random walk, which means that this task is no longer an OLAP query[35].

To sum it up, the user query is a search of the shortest path of a maximal given length between two vertices. Such a task, however, produces a substantial amount of traversed paths, which means a large number of log records. With the maximal depth of 10 and vertices of degree equal to 3 the breadth-first search can produce $2^{10} = 1024$ different paths. With $8,000$ queries, this would create a number of records difficult to process with our resources. Therefore, the number of actually logged paths of one traversal is restricted to 15, always including the shortest path.

[35]Note, that it is also possible to create an OLTP type of query that finds the shortest path within the whole graph. However, the task is so demanding that the execution crashes on lack of resources.

6.2 Evaluation Methods

In section 4.5 we proposed several adjustments of the Vaquero et al. algorithm. The effects of the new factors are independently measured. First, the list of these factors is introduced. Subsequently, the particular measured effects are described.

6.2.1 Factors

- **Cooling Factor** – a temperature of the simulated annealing is used as presented in equations 4.2 and 4.3. It means that the temperature is always applied in a fraction, therefore, its initial value does not matter. The cooling factor, however, used for cooling the temperature $T_{new} = T_{init} \times coolingFactor^{iteration}$, influences the speed of the cooling process which has a crucial effect on getting trapped in a local optimum.

- **Imbalance Factor** – the equation 2.2 of a lower constraint introduces an imbalance factor f. It determines how much percent of the proportionally assigned number of vertices a node can be undersized. This also means that, for example, with imbalance factor of 10% and three identical physical nodes, it is possible, that two nodes are undersized to 10%, thus the third node being oversized to 20%. The capacity of the nodes is always set in the way that the upper bound of the capacity of a node does not restrict the migration – in the previous example, it is always allowed to oversize the node to 20%.

- **Adoption Factor** – the equation 4.3 proposed a new probability of adoption incorporating temperature into its computation. The equation can be written as
$$s = S + a \cdot \frac{T}{T_{init}} \cdot (1 - S),$$
where the a denotes an adoption factor. When set to 0, the equation equals back the original $s = 0.5$. When equal to 1, the proposed equation holds. A middle value of $a = \frac{1}{2}$ is examined as well.

6.2.2 Effects

- **Improvement** – as discussed in 4.2, we measure the performance according to network communication, that is, the number of crosses between physical nodes a traversal has to make in order to process a query. For this reason, we count the improvement as a percentage value of
$$1 - \frac{\textit{number of crossings after the redistribution}}{\textit{number of crossings before the redistribution}}.$$
The measurement is done on queries that have been used as an input log for the redistribution.

- **Number of Iterations** – how many iterations of the Vaquero et al. algorithm it takes to create a result. The ending condition of the method is a situation when no more vertices change their label, or the maximum number of iterations set to 200 is reached.
- **Capacity Usage** – nodes in a cluster are initially evenly distributed (see 6.3 for details), whereas the redistribution algorithm proposes relocation of that data. It is tracked how the capacity usage of individual nodes within the cluster changes, with the percentage imbalance of a node N_i measured as
$$\frac{N_i - \frac{|V|}{|N|}}{\frac{|V|}{|N|}},$$
where N denotes the set of nodes of a cluster, N_i representing the current number of vertices on the node i, and V is the set of vertices of a graph[36]. Note, that the average $\frac{|V|}{|N|}$ should be a proportional number of vertices assigned to a given physical node based on its capacity. As our setting contains nodes of equal sizes, the number can be simplified as an arithmetic average.

Additionally, the redistribution was run on a different number of nodes in a cluster to imagine better what values to expect with horizontal scaling.

6.3 Settings

Partition IDs For the measurement purposes our cluster contains three nodes of equal sizes by default. Emulating our environment, we do not store the data on three physical nodes in reality, but we assign the partition IDs artificially. We take advantage of the provided data with already assigned consecutive vertex IDs and use a modulo function to distribute them initially.

Capacity As already mentioned in 6.2.2, the capacity of each node in the cluster is set high enough not to restrict the label adoptions with the upper constraint – the effects of the lower constraint as such can be tracked independently on other effects. The capacity of the nodes always equals.

Environment The measurements were performed on a machine with AMD Ryzen 5 1600 Six-Core Processor, 3.8 GHz, 16 GB RAM and 64-bit Windows 10. However, all the computations are measured with the number of inner steps, which means that the machine computational power does not influence our results. It only restricts the capabilities of our dataset sizes.

[36]Not incorporating edges because all the elements in JanusGraph are stored within vertices – edge data are part of the vertex data as described in section 3.1.1

We used the JanusGraph version 0.3.0 with TinkerPop version 3.3.3. For the backend storage, we operated both with Cassandra, version 2.1.20 and Oracle Berkeley DB, version 7.4.5 (the backend storage does not influence our results). All these product versions are provided natively with JanusGraph distribution.

6.4 Results

For each of the examined values, we performed ten measurements in the case of the Twitter dataset and four measurements in the case of the Road network dataset. There are two reasons for this difference – first, the Road map contains about fourteen times more nodes than the Twitter dataset, therefore, the computation is much slower. Second, thanks to the nature of the Road network data (discussed below) the standard deviation between its results is much lower, meaning that more repetitions do not bring much added value.

Complete results with all the graphs and tables are available in the Appendix B. More detailed data are also provided on the attached medium of this work. In this section, we mainly discuss individual effects and depict graphs containing meaningful behavior. The text is structured according to the particular factors introduced in 6.2.1.

6.4.1 Cooling Factor

Effects of the cooling factor have been examined with values from the set {*0.95; 0.96; 0.97; 0.98; 0.99; 0.995; 0.999*}.

First, the development of cross-node communication improvement with various values of the factor is shown in Figure 6.3. The lower the value of the factor, the faster the cooling process of $T_{new} = T_{init} \times coolingFactor^{iteration}$ (because $coolingFactor \in [0; 1]$), and also the higher the probability of getting trapped in a local optimum. On the other hand, when the value of the factor is too high, the cooling process is too slow, resulting in a too frequent acceptance of degrading solutions with low power to produce a satisfying solution.

From the graph 6.3 it seems that the values 0.98 and 0.99 can generate the highest improvements. However, the graph 6.3 shows only one run of each factor value. The cumulative information about the best, the worst and the average result of different factor setting is depicted in Figure 6.4.

The effect of the cooling factor value on the improvement differs with the dataset used. The Twitter dataset is a scale-free graph (as mentioned in 6.1.1) which is more difficult to partition. The higher the value of the factor, the better the solution, but, an overly high value already degrades the final result as described earlier.

The road map, on the contrary, is a graph of a simpler topology, which can be partitioned in significantly more suitable ways. Thanks to that characteristic, even a very high cooling factor can find one of many convenient

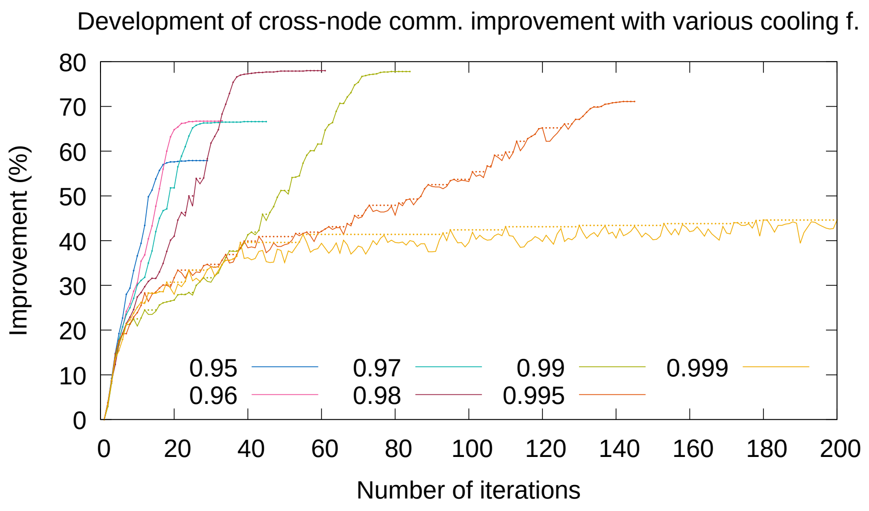


Figure 6.3: Development of cross-node communication improvement with various cooling factors. The dots represent the maximal value of the process so far. Measured on the Twitter dataset.

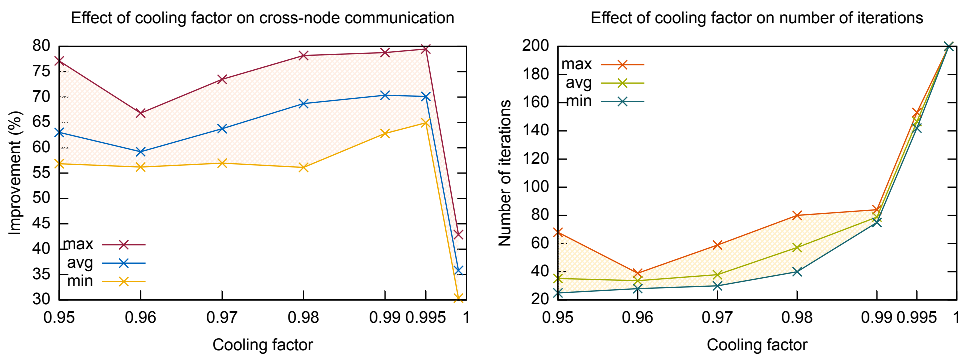
distributions, resulting in the improvement evincing a linear dependency on the examined cooling factors. At the same time, the improvement always lies within 71–77% and the results of one factor value has an average standard deviation of only 0.25%. The average standard deviation for the Twitter dataset is equal to 5.51% due to its more difficult structure and interconnectivity which does not allow so many relevant solutions, meaning it is more difficult to find some.

On the other hand, from the Figure 6.4 it seems that the dependency of number of iterations on the cooling factor is independent on the graph structure. The number of iterations always grows with the value of the cooling factor exponentially.

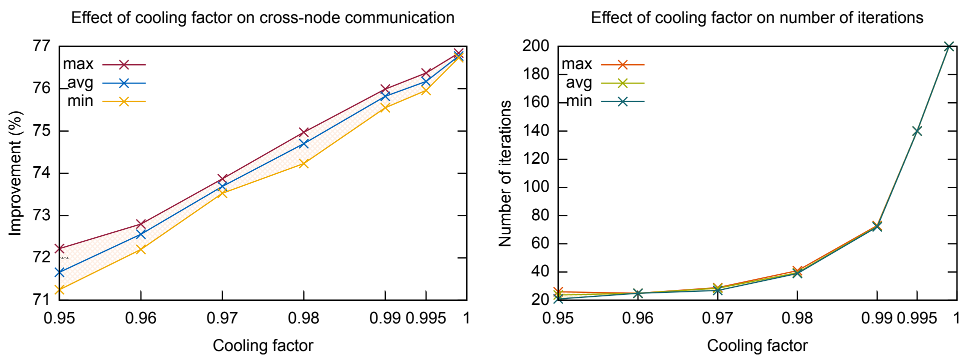
Taking into consideration the dependency on communication improvement and the number of iterations, in both cases the cooling factor of 0.99% produces very good results regarding cross-node communication, and its number of iterations is still of a reasonable value. That is why this value has been chosen for further experiments.

6.4.2 Imbalance Factor

The Figure 6.5 presents effects of the imbalance factor. The imbalance factor was examined with values from the set {*0; 0.01; 0.02; 0.03; 0.04; 0.05; 0.1; 0.15; 0.2; 0.3; 0.4; 0.5*}. As expected, the value of 0 results in 1 or 2 iterations



(a) Twitter dataset



(b) Road map dataset

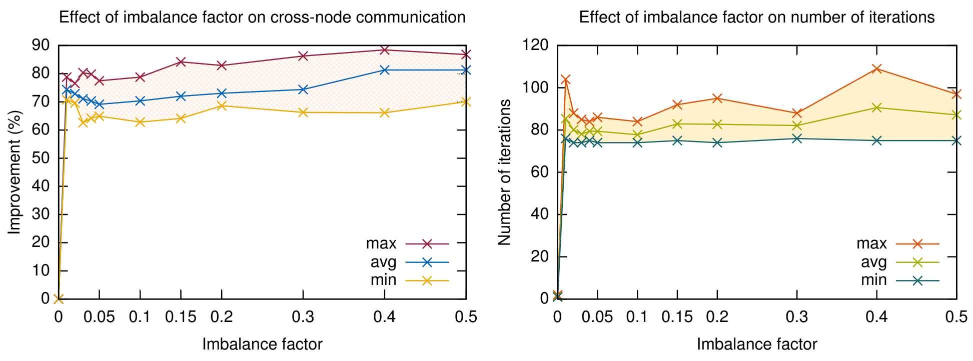
Figure 6.4: Effect of cooling factor on cross-node communication and number of iterations.

during which the very slight imbalance of the initial distribution is evened up and no further migrations are allowed.

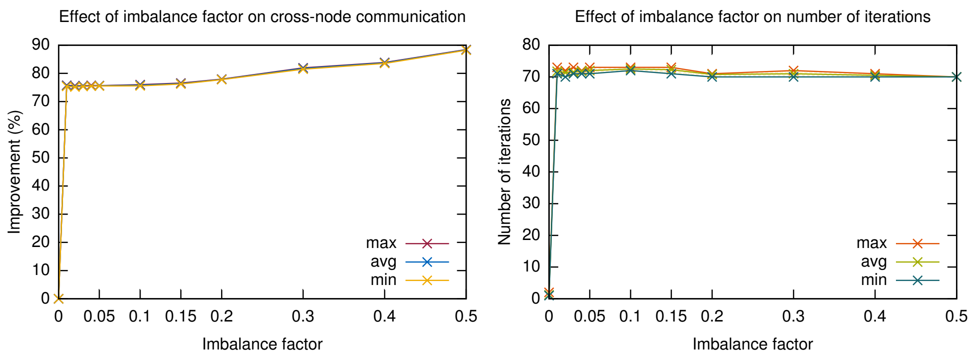
In the Road network dataset, the improvement grows with higher imbalance allowed. This behavior can be expected, because the higher the imbalance permitted, the more relevant data together on one cluster, which means the lower the necessary communication.

The high improvement even with very small imbalance factor of 1% could be rather unanticipated. With this value, the improvement of cross-node communication is still tightly around 75.5%. The reason is most probably the fact that with 1,088,092 vertices, the 1% of allowed underloading on a physical node is equal to 3, 627 vertices. That is still quite a generous space, evidently large enough for the necessary migration.

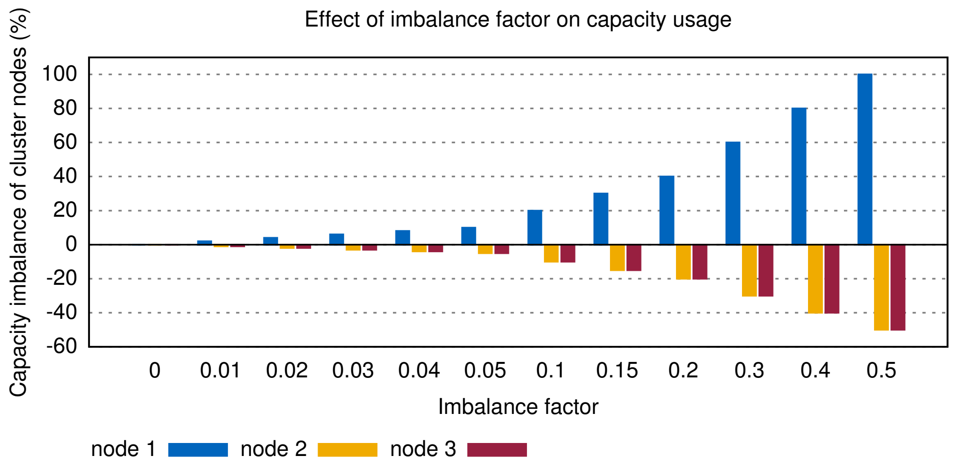
In the Twitter dataset, the behavior in the interval of [0.05;0.5] has the same characteristic as in the Road map graph. However, the small values of the imbalance factor perform unanticipated conduct, receiving slightly better results with even lower values. We explain this behavior as a potential



(a) Twitter dataset



(b) Road network dataset



(c) Capacity usage in the Road network dataset

Figure 6.5: Effect of imbalance factor on cross-node communication, number of iterations and capacity usage.

combination of three possible factors.

- The higher the imbalance allowed, the higher the migration of vertices to one physical node. Accordingly, with a lower factor, the vertices tend to move more within the nodes, back and forth, creating better local solutions per node. That can be convenient for the given queries that are defined as very local with a maximum diameter of the length 3.
- From the figure 6.5a we can see that the lower the factor in the given interval [0.01;0.05] the roughly higher the number of iterations. Commonly, with a higher number of iterations, the results evince better improvements because more potential redistributions have been performed.
- The number of executed runs can be insufficient, the lower values of the factor had several better performing runs in a row above their actual average.

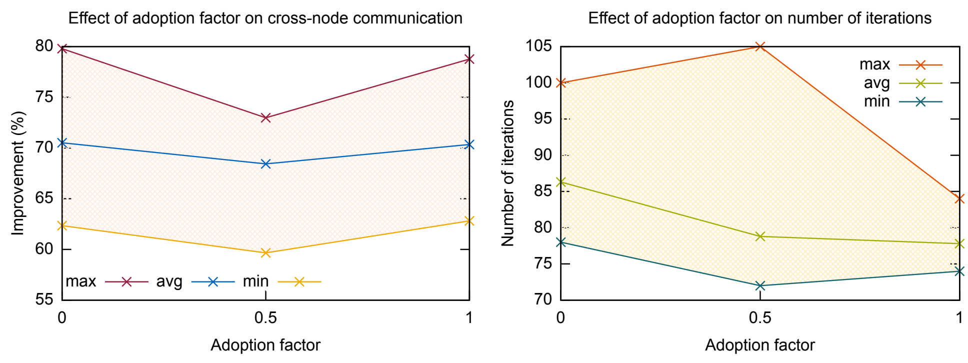
Nevertheless, for both the datasets, it is important to mention, that even with a low allowed imbalance of 1–10% we gain improvement of approximately 70%, with the maximum around 78%. That is a very appealing result. It means that even with low imbalance, we can receive results of comparable quality to Ja-be-Ja algorithm which claims to reduce the edge-cut by 70–80%. Both results are discussed more in detail in summary of this chapter 6.5.

The effect of the imbalance factor on the number of iterations does not seem to be significant. In the Twitter dataset, the number of iterations slightly grow with higher imbalance factor as it takes a longer time to migrate more vertices to one node. In the Road network dataset, all the iterations are within the range $70 - 73$ for all the examined values of imbalance factor.

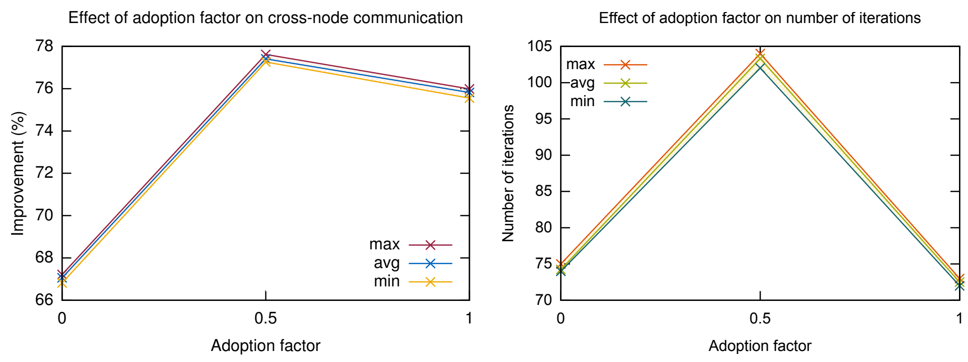
While the other analyzed factors do not tend to influence the capacity usage of the nodes in the cluster, the imbalance factor has, of course, an effect on it. The data always tend to maximally use the allowed space and reorganize themselves mainly on one cluster, underloading the other nodes as much as permitted. It seems reasonable to take advantage of the capacity available. On the other hand, the final result can be in total imbalanced – even more with the higher the imbalanced factor or, the higher the number of nodes in a cluster.

6.4.3 Adoption Factor

The adoption factor seems to have effects sensitive to a dataset as visible in Figure 6.6. In the case of the Twitter dataset, the cross-node communication improvement is comparable in both the cases of adoption factor equal to 0 and 1. The value 1, however, performs better in the number of required iterations



(a) Twitter dataset



(b) Road network dataset

Figure 6.6: Effect of adoption factor on cross-node communication and number of iterations.

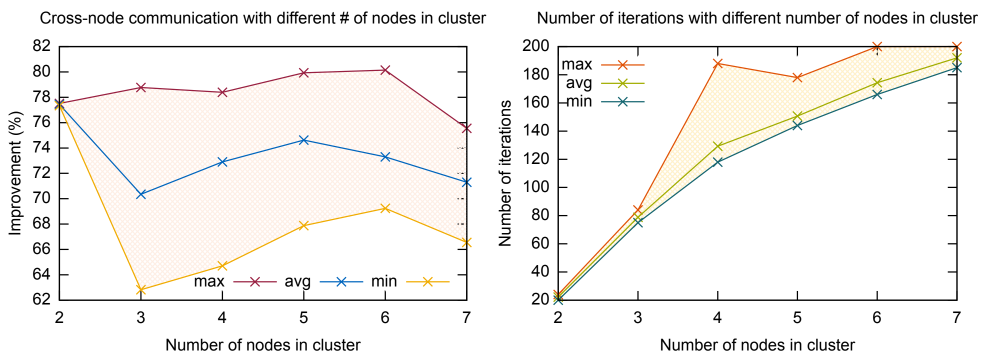
because it did not reject so many labels from the beginning of the process as the value 0. On the contrary, in the Road network dataset, the number of iterations for these values equals, while the improvement is better in the case of adoption factor set to 1.

The middle value of 0.5 then also behaves differently. While in the Twitter dataset it has the worse improvement with the middle average value of the number of iterations, in the Road network dataset it has the best improvement but with the cost of the much higher number of iterations.

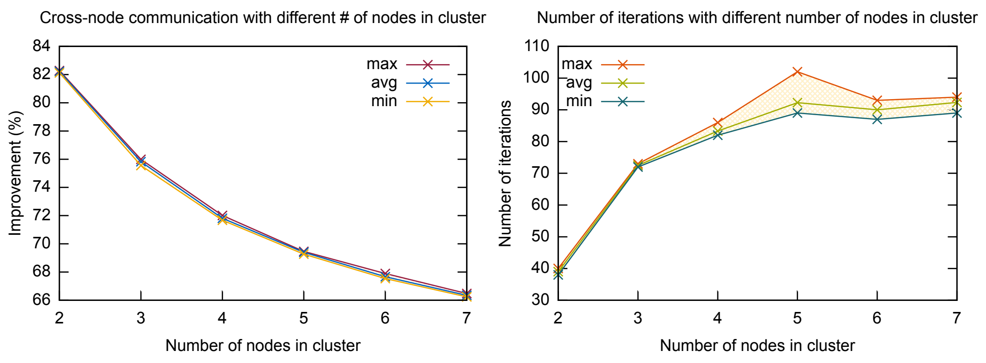
Nevertheless, in both the datasets, it seems that the best value to apply is 1 (regarding the improvement and the number of iterations, combined), but the behavior should always be examined for each dataset separately.

6.4.4 Number of Nodes in Cluster

The effects of the number of nodes in the cluster are shown in Figure 6.7. In both datasets, we can see the growing tendency of the required number of



(a) Twitter dataset



(b) Road network dataset

Figure 6.7: Effect of adoption factor on cross-node communication and number of iterations.

iterations with the higher number of nodes in the cluster as it takes a longer time to migrate the vertices to suitable nodes.

The achieved improvement with a different number of nodes is dependent on the topology of the dataset. The Road network is a simpler graph with more balanced degrees than the Twitter dataset. It has more possible solutions of partitioning with a satisfactory improvement. It can be generally expected that the higher the number of nodes, the higher the cross-node communication, so a possible improvement is decreased.

The average improvement in the Twitter dataset stays within the range of [70; 77] with all the examined number of nodes. The topology of small amount of vertices with very high degree and substantial number of vertices with very low degree allow such reorganization within the nodes of cluster.

6.5 Summary

With the enhanced Vaquero et al. algorithm proposed in 4.5 it is possible to create results of high communication improvement and of a reasonable balance in very satisfactory time. The table 6.3 sums up the most important results of the evaluation.

Even with small imbalance values of 1–10% the cross-node communication can be improved up to 70–80%. The necessary number of iterations for the algorithm in such case moves around 80–85 or 72 depending on the dataset.

The Ja-be-Ja algorithm claims to produce the same improvement of 70–80% with four hosts in the cluster. We did our measurements mainly on three hosts, but in our Twitter dataset, the four hosts caused even better results than the three hosts – the improvement increased by 2.5% (with the cost of 50 extra iterations). In the Road network dataset the results decreased by 4% (still $> 70\%$) in improvement and increased by 11 iterations on average.

The Ja-be-Ja result is always balanced but the necessary number of iterations for the Twitter set containing only 2,731 vertices, and 164,629 edges is 350. Moreover, one iteration in Ja-be-Ja is more computationally demanding and with much higher communication overhead. Furthermore, the variance in the improvement of proposed results with the same setting can be significant; therefore, in order to obtain the best solution possible, more executions of the algorithm should be performed. It is highly probable that this is the case of Ja-be-Ja algorithm as well.

We can conclude that we have an algorithm creating comparable improvements of cross-node communication to another well-known method yet with extensively better computational and communication demands. On the other hand, it is necessary to allow an imbalance of nodes in the cluster, preventing the redistribution from providing a hundred percent balanced solution. However, even small and acceptable imbalances produce great commensurate results.

	Twitter dataset			Road network dataset		
factor value	average improvement	maximum improvement	average number of iterations	average improvement	maximum improvement	average number of iterations
cooling factor						
0.98	68.72%	78.18%	57.2	74.7%	74.97%	39.5
0.99	70.35%	78.77%	78.8	75.82%	75.99%	72.5
imbalance factor						
0.01	74.36%	78.77%	85.4	75.59%	75.77%	71.75
0.05	69.13%	77.49%	79.4	75.59%	75.66%	72
0.1	70.35%	78.77%	77.8	75.82%	75.99%	72.5
adoption factor						
0	70.51%	79.79%	86.3	67.06%	67.22%	74.25
1	70.35%	78.77%	77.8	75.82%	75.99%	72.5
number of nodes in cluster						
3	70.35%	78.77%	78.83	75.82%	75.99%	72.5
4	72.9%	78.39%	129.3	71.81%	72.01%	83.33

Table 6.3: The most significant results of the evaluation. Unless stated differently, the factors were set as: $coolingFactor = 0.99$, $imbalanceFactor = 0.1$, $adoptionFactor = 1$, $numberOfNodes = 3$.

Chapter 7

Enhancements

The implementation output of this work contains several parts which could be further enhanced or experimented with. In this short chapter, we propose some of these improvements, but the real possibilities are certainly far more abundant.

7.1 Query Data Extraction

The TinkerPop module is a separate piece of code written with an intent to be easily extensible. Consequently, other post-processing of a result than path logging could be implemented and new formats can be intuitively supplied.

Nevertheless, the module could also be further enhanced. One of the ideas is to implement the logging functionality differently, creating a *log* step that would be added in the code only where necessary. That could be very convenient for cases when users execute actions further processed with another layer containing the actual queries.

The current implementation does not sanitize the situation when the information logged to a file exceeds the maximum size allowed. A corresponding implementation resolving this issue should be surely provided.

7.2 Graph Redistribution Manager

The Graph Redistribution Manager is a proof of concept as discussed in 4, which means that a larger number of improvements could be performed. We propose some examples:

- *data preprocessing* – the GRM reads log files containing individual paths of traversals. The resulting weight on a newly added edge between two vertices then represents the number of occurrences of the real edges between these vertices in the log file (more details can be found in 4.2.2). Therefore, rather then always incrementing the weight by one while reading the log file, the MapReduce implementation could be applied to preprocess the file and consecutively to save that cumulated information in the database at once.

- *physical redistribution* – implement the proposed redistribution of data on physical nodes when that is possible (the current obstruction is discussed in 5.3.8).

- *best result selection* – the GRM could be extended with the possibility of running several executions while storing each result separately. Later, the best solution could be selected and applied.

7.3 Algorithm

The adjusted Vaquero et al. algorithm could be further modified in several areas

- *upper bound* – the current upper bound is based on the capacity of physical nodes. Together with the lower bound of a controlled possible underloading, this can create an imbalance on one node containing the majority of data. Using the entire capacity of one node seems reasonable but another upper bound could be experimented with (perhaps combining the node capacity and maximum allowed oversizing).

- *physical size of a vertex* – the current implementation balances the number of vertices, assuming each vertex has an equal size. The actual physical demands of each vertex could be introduced.

- *replicas* – the notion of replicas has been omitted in this work. Their introduction would bring new possibilities to the repartitioning task, which would have to be incorporated in the algorithm as well.

- *fading* – a trend of the database usage can change over time. It means that during the computation of a redistribution algorithm, the most recently utilized connections should be preferred over the older ones. The information about the last update on the edges could be used for such algorithm modification.

- *vertex weight* – until now only edges contain the property of weight because the task was defined as a minimalization of edge-cuts, that is, the optimization of cross-node communication. The problem could also be extended with a simultaneous balancing of physical node access. That would require the introduction of vertex weight as a user can access the node asking only about one vertex as well.

7.4 Experiments

Apart from the possible inner modifications of current implementation, more experiments should be performed. It would be beneficial to run both the adjusted Vaquero et al. algorithm and Ja-be-Ja method on identical data and with the same environment setting. Comparison of the number of iterations, time requirements on an iteration and other characteristics would provide more detailed information about the real benefits and limitations of one method over the other.

Further analysis and implementation of more redistributing algorithms could be encountered, followed by subsequent comparison with the other already implemented solutions.

Last but not least, more datasets of various sizes and topologies should be used to examine and compare the methods.

Conclusion

The main goal of this master thesis was an analysis of queries executed in distributed graph databases. First, the current practices of data storage in distributed graph databases were researched with focus on open-source products. Afterwards, we analyzed how to extract the necessary information for further query analysis and implemented this solution as a new module of the general graph computing framework TinkerPop [64]. As a target graph database for storage of that extracted data, the JanusGraph [86] was chosen. A new implementation providing the storage of the data in the database was performed. It also offers a redistribution algorithm producing a more efficient solution for data distribution in the database.

The implementation part of this work has two separate outputs. The new TinkerPop module for logging purposes has been designed in the way that it could be applied both as a silent server logger set in server configuration or as an explicitly called logger during query creation. The module is written with the usage of several interfaces, which makes it easily extensible either with new formats of the logged data or with new functions applied on the result of a query. The currently supported operation run on the query result is a *path*-step returning all the paths that have been traversed in order to resolve a query. The anonymization of the data is available as well.

The other implementation part is a proof of concept, provisionally called the Graph Redistribution Manager. It can load and store datasets for evaluation purposes together with the generation and execution of their defined queries. With the application of the new module, the queries produce log files that serve as an input for the primary functionality of the Graph Redistribution Manager. First, the log files are loaded in the database in a simplified form of newly added weighted edges. Second, a partition ID is stored for each vertex simulating the physical node of a cluster on which the vertex is located. Finally, a redistribution algorithm is executed, resulting in a proposal of more efficient data distribution regarding the communication between physical hosts.

According to the analysis of partitioning algorithms, the algorithm of Vaquero et al. [37] has been selected, although it experiences several drawbacks. For this reason, significant adjustments were proposed and applied. We incorporated the method of simulated annealing to avoid local optima more successfully. Furthermore, we introduced a lower bound of a maximum allowed imbalance based on host capacity and the number of vertices. We also adjusted the calculations for weighted graphs.

The performance improvements of our proposed redistribution surpassed our expectations. We gained 70–80% lower communication demands among physical nodes of a cluster. These results are comparable to another well-known repartitioning algorithm Ja-be-Ja which, however, requires a higher number of iterations with substantial communication overhead. On the other hand, it can provide a perfectly balanced solution while our method requests the introduction of an imbalance. Nevertheless, even small imbalances of 1–10% can produce the stated high-quality results.

The accomplished implementation provides all the mentioned functionalities. Nonetheless, it can be a subject of several other enhancements and extensions. Our proposals for such future work were introduced in the last chapter of this work. The main advantage of the current solution is the general applicability of the separate logging TinkerPop module, a good basis for further experiments provided by the Graph Redistribution Manager and our original enhancements of the Vaquero et al. algorithm that bring eminent results.

Bibliography

[1] DB-Engines.com. DBMS popularity broken down by database model, Popularity changes per category, September 2018 [online]. September 2018, [Accessed 18 September 2018]. Available from: `https://db-engines.com/en/ranking_categories`

[2] Berg, K. L.; Seymour, T.; et al. History of Databases. *International Journal of Management & Information Systems*, volume 17, no. 1, 2013: pp. 29–35, doi:10.19030/ijmis.v17i1.7587.

[3] Błażewicz, J.; Kubiak, W.; et al. *Handbook on Data Management in Information Systems*. International Handbooks on Information Systems, Berlin: Springer, 2003, ISBN 9783540438939, [Accessed 23 November 2018]. Available from: `https://books.google.cz/books?id=AvLziHKyuLcC`

[4] Bachman, C. W. The programmer as navigator. *Communications of the ACM*, volume 16, no. 11, November 1973: pp. 653–658, doi:10.1145/355611.362534.

[5] Codd, E. F. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, volume 13, no. 6, June 1970: pp. 377–387, doi:10.1145/362384.362685.

[6] DB-Engines.com. DB-Engines Ranking [online]. November 2018, [Accessed 26 November 2018]. Available from: `https://db-engines.com/en/ranking`

[7] Strauch, C. *NoSQL Databases*. [Lecture Selected Topics on Software–Technology Ultra–Large Scale Sites, Manuscript], Stuttgart Media University, 2011, [Accessed 26 November 2018]. Available from: `http://www.christof-strauch.de/nosqldbs.pdf`

[8] World Wide Web Consortium. Resource Description Framework (RDF) [online]. [Accessed 07 October 2018]. Available from: `https://www.w3.org/RDF/`

[9] Barrasa, J. RDF Triple Stores vs. Labeled Property Graphs: What's the Difference? [online]. August 2017, [Accessed 27 November 2018]. Available from: `https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/`

[10] Kolář, J. *Orientované grafy, reprezentace grafů. Lekce 3.* [University Lecture], Faculty of Information Technology, Czech Technical University in Prague, Prague, Summer Semester 2016. Available from: `https://edux.fit.cvut.cz/courses/BI-GRA/_media/lectures/gra-predn-03cb.pdf`

[11] World Wide Web Consortium. SPARQL 1.1 Query Language [online]. March 2013, [Accessed 28 November 2018]. Available from: `https://www.w3.org/TR/sparql11-query/`

[12] Neo4j, Inc. Intro to Cypher [online]. [Accessed 28 November 2018]. Available from: `https://neo4j.com/developer/cypher-query-language/`

[13] The Apache Software Foundation. The Gremlin Graph Traversal Machine and Language [online]. [Accessed 28 November 2018]. Available from: `https://tinkerpop.apache.org/gremlin.html`

[14] Bichot, C.-E.; Siarry, P. *Graph Partitioning.* Hoboken: John Wiley & Sons, Incorporated, first edition, 2013, ISBN 9781118601259, [Accessed 21 November 2018]. Available from: `https://ebookcentral.proquest.com`

[15] Andreev, K.; Räcke, H. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, Barcelona, Spain, 2004, pp. 120–124, doi:10.1145/1007912.1007931.

[16] Buluç, A.; Meyerhenke, H.; et al. Recent Advances in Graph Partitioning. In *Algorithm Engineering. Lecture Notes in Computer Science*, volume 9220, Cham: Springer International Publishing, 2016, pp. 117–158, doi:10.1007/978-3-319-49487-6_4.

[17] Schaeffer, S. E. Graph Clustering. *Computer Science Review*, volume 1, no. 1, August 2007: pp. 27–64, doi:10.1016/j.cosrev.2007.05.001.

[18] Fjällström, P.-O. Algorithms for Graph Partitioning: A Survey. *Linköping Electronic Artlicles in Computer and Information Science*, volume 3, no. 10, 1998: pp. 1–34, ISSN 1401-9841, [Accessed 18 November 2018]. Available from: `http://www.ep.liu.se/ea/cis/1998/010/cis98010.pdf`

[19] Ferreira, C. E.; Martin, A.; et al. The node capacitated graph partitioning problem: A computational study. *Mathematical Programming*, volume 81, no. 2, January 1998: pp. 229–256, doi:10.1007/BF01581107.

[20] Stacho, L.; Vrt'o, I. Bisection Width of Transposition Graphs. *Discrete Applied Mathematics*, volume 84, no. 1–3, May 1998: pp. 221–235, doi:10.1016/S0166-218X(98)00009-2.

[21] Land, A. H.; Doig, A. G. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, volume 28, no. 3, July 1960: pp. 497–520, doi:10.2307/1910129.

[22] Fiedler, M. Algebraic Connectivity of Graphs. *Czechoslovak Mathematical Journal*, volume 23, no. 2, January 1973: pp. 298–305, [Accessed 18 November 2018]. Available from: `https://dml.cz/bitstream/handle/10338.dmlcz/101168/CzechMathJ_23-1973-2_11.pdf?sequence=2`

[23] Weisstein, E. W. Laplacian Matrix. From MathWorld – A Wolfram Web Resource [online]. [Accessed 18 November 2018]. Available from: `http://mathworld.wolfram.com/LaplacianMatrix.html`

[24] Karypis, G.; Kumar, V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, volume 20, no. 1, 1998: pp. 359–392, doi:10.1.1.39.3415.

[25] Cormen, T. H.; Leiserson, C. E.; et al. *Introduction to Algorithms*. Cambridge, Massachusetts: MIT Press and McGraw-Hill, third edition, 2009, ISBN 978-0-262-53305-8, [Accessed 24 September 2018]. Available from: `https://labs.xjtudlc.com/labs/wldmt/reading%20list/books/Algorithms%20and%20optimization/Introduction%20to%20Algorithms.pdf`

[26] Sanders, P.; Schulz, C. Engineering Multilevel Graph Partitioning Algorithms. In *19th European Symposium on Algorithms*, Berlin: Springer, September 2011, pp. 469–480, doi:10.1007/978-3-642-23719-5_40.

[27] Simon, H. D. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, volume 2, no. 2–3, 1991: pp. 135–148, doi:10.1016/0956-0521(91)90014-V.

[28] Rahimian, F.; Payberah, A. H.; et al. A Distributed Algorithm for Large-Scale Graph Partitioning. *ACM Transactions on Autonomous and Adaptive Systems*, volume 10, no. 2, June 2015: pp. 1–24, doi:10.1145/2714568.

[29] Firth, H.; Missier, P. TAPER: query-aware, partition-enhancement for large, heterogenous, graphs. *Distributed and Parallel Databases*, volume 35, no. 12, June 2017: pp. 85–115, doi:10.1007/s10619-017-7196-y.

[30] Bondy, J. A.; Murty, U. *Graph Theory With Applications*. North-Holland, fifth edition, 1982, ISBN 0-444-19451-7, 12 pp., [Accessed 16 September 2018]. Available from: `https://web.archive.org/web/20081004204910/http://www.ecp6.jussieu.fr/pageperso/bondy/books/gtwa/pdf/GTWA.pdf`

[31] Karypis, G. Family of Graph and Hypergraph Partitioning Software [online]. [Accessed 18 November 2018]. Available from: `http://glaros.dtc.umn.edu/gkhome/views/metis`

[32] Sorensen, K.; Glover, F. Metaheuristics. In *Encyclopedia of Operations Research and Management Science*, edited by S. I. Gass; M. C. Fu, New York: Springer, third edition, 2013, pp. 960–970, [Accessed 20 November 2018]. Available from: `https://www.researchgate.net/publication/237009172_Metaheuristics`

[33] Tashkova, K.; Korošec, P.; et al. A distributed multilevel ant–colony algorithm for the multi–way graph partitioning. *International Journal of Bio–Inspired Computation*, volume 3, no. 5, January 2011: pp. 286–296, doi:10.1504/IJBIC.2011.042257.

[34] Schmidt, J. *8. Simulované ochlazování; Simulated Annealing, SA*. [University Lecture], Faculty of Information Technology, Czech Technical University in Prague, Prague, Winter Semester 2017. Available from: `https://edux.fit.cvut.cz/courses/MI-PAA/_media/lectures/08/paang8ochl.pdf`

[35] Malewicz, G.; Austern, M. H.; et al. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, Indianapolis, Indiana, USA, 2010, pp. 135–146, doi:10.1145/1807167.1807184.

[36] Gehweiler, J.; Meyerhenke, H. A distributed diffusive heuristic for clustering a virtual P2P supercomputer. In *2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW)*, Atlanta, Georgia, USA, 2010, pp. 1–8, doi:10.1109/IPDPSW.2010.5470922.

[37] Vaquero, L. M.; Cuadrado, F.; et al. Adaptive Partitioning for Large-Scale Dynamic Graphs. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, Madrid, Spain, 2014, pp. 144–153, doi:10.1109/ICDCS.2014.23.

[38] Raghavan, U. N.; Albert, R.; et al. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, volume 76, no. 3, October 2007, doi:10.1103/PhysRevE.76.036106.

[39] Ugander, J.; Backstrom, L. Balanced label propagation for partitioning massive graphs. In *Proceedings of the sixth ACM international conference on Web search and data mining*, Rome, Italy, 2013, pp. 507–516, doi: 10.1145/2433396.2433461.

[40] DataStax, Inc. DataStax Acquires Aurelius, The Experts Behind TitanDB [online]. [Accessed 18 September 2018]. Available from: `https://www.datastax.com/2015/02/datastax-acquires-aurelius-the-experts-behind-titandb`

[41] DB-Engines.com. DB-Engines Ranking of Graph DBMS [online]. September 2018, [Accessed 18 September 2018]. Available from: `https://db-engines.com/en/ranking/graph+dbms`

[42] JanusGraph.org. Part III. Storage Backends [online]. [Accessed 22 September 2018]. Available from: `https://docs.janusgraph.org/latest/storage-backends.html`

[43] The Apache Software Foundation. Apache Cassandra [online]. [Accessed 22 September 2018]. Available from: `http://cassandra.apache.org/`

[44] Cockcroft, A.; Sheahan, D. Benchmarking Cassandra Scalability on AWS - Over a million writes per second [online]. November 2011, [Accessed 22 September 2018]. Available from: `https://medium.com/netflix-techblog/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e`

[45] The Apache Software Foundation. Apache HBase [online]. [Accessed 22 September 2018]. Available from: `http://hbase.apache.org/`

[46] Google, Inc. Cloud Bigtable [online]. [Accessed 23 September 2018]. Available from: `https://cloud.google.com/bigtable/`

[47] Oracle Corporation. Oracle Berkeley DB Java Edition [online]. [Accessed 23 September 2018]. Available from: `https://www.oracle.com/technetwork/database/berkeleydb/overview/index-093405.html`

[48] JanusGraph.org. Chapter 18. Apache Cassandra [online]. [Accessed 23 September 2018]. Available from: `https://docs.janusgraph.org/latest/cassandra.html`

[49] JanusGraph.org. Chapter 40. JanusGraph Data Model [online]. [Accessed 26 September 2018]. Available from: `https://docs.janusgraph.org/latest/data-model.html`

[50] OrientDB Ltd. OrientDB Manual [online]. [Accessed 28 September 2018]. Available from: `https://orientdb.com/docs/last/`

[51] OrientDB Ltd. Graph or Document API? [online]. [Accessed 05 October 2018]. Available from: https://orientdb.com/docs/2.2.x/Choosing-between-Graph-or-Document-API.html

[52] OrientDB Ltd. Basic Concepts [online]. [Accessed 05 October 2018]. Available from: http://orientdb.com/docs/2.0/orientdb.wiki/Concepts.html

[53] OrientDB Ltd. Distributed Architecture [online]. [Accessed 05 October 2018]. Available from: https://orientdb.com/docs/3.0.x/distributed/Distributed-Architecture.html

[54] ArangoDB.com. TinkerPop 3 [online]. *GitHub repository*, [Accessed 06 October 2018]. Available from: https://github.com/ArangoDB-Community/arangodb-tinkerpop-provider/issues/15

[55] ArangoDB.com. ArangoDB v3.3.16 Documentation [online]. [Accessed 06 October 2018]. Available from: https://docs.arangodb.com/3.3/Manual/

[56] ArangoDB.com. ArangoDB [online]. *GitHub repository*, [Accessed 06 October 2018]. Available from: https://github.com/arangodb/arangodb

[57] Facebook, Inc. RocksDB [online]. [Accessed 06 October 2018]. Available from: https://rocksdb.org/

[58] Dgraph Labs, Inc. BadgerDB [online]. *GitHub repository*, [Accessed 07 October 2018]. Available from: https://github.com/dgraph-io/badger

[59] Dgraph Labs, Inc. Dgraph [online]. *GitHub repository*, [Accessed 07 October 2018]. Available from: https://github.com/dgraph-io/dgraph

[60] OpenLink Software. OpenLink Virtuoso Universal Server Documentation [online]. [Accessed 07 October 2018]. Available from: http://docs.openlinksw.com/virtuoso/

[61] Microsoft Corporation. Databáze Azure Cosmos [online]. [Accessed 19 October 2018]. Available from: https://azure.microsoft.com/cs-cz/services/cosmos-db/

[62] Stardog Union. Stardog Release Notes [online]. [Accessed 20 October 2018]. Available from: https://www.stardog.com/docs/release-notes/

[63] TigerGraph.com. August 23 Meetup in London [online]. [Accessed 20 October 2018]. Available from: https://www.tigergraph.com/2018/08/14/august-23-meetup-in-london/

[64] The Apache Software Foundation. Apache TinkerPop [online]. May 2018, [Accessed 25 August 2018]. Available from: `http://tinkerpop.apache.org/`

[65] The Apache Software Foundation. Apache TinkerPop [online]. *GitHub repository*, [Accessed 26 August 2018]. Available from: `https://github.com/apache/tinkerpop`

[66] The Apache Software Foundation. Apache License [online]. [Accessed 25 August 2018]. Available from: `http://tinkerpop.apache.org/providers.html`

[67] The Apache Software Foundation. Apache License [online]. January 2014, [Accessed 25 August 2018]. Available from: `http://www.apache.org/licenses/LICENSE-2.0`

[68] The Apache Software Foundation. Getting Started [online]. [Accessed 28 November 2018]. Available from: `http://tinkerpop.apache.org/docs/3.3.4/tutorials/getting-started/`

[69] The Apache Software Foundation. Apache TinkerPop [online]. [Accessed 26 August 2018]. Available from: `http://tinkerpop.apache.org/docs/3.3.3/reference/`

[70] JanusGraph.org. Chapter 6. Gremlin Query Language [online]. [Accessed 28 November 2018]. Available from: `https://docs.janusgraph.org/latest/gremlin.html`

[71] The Apache Software Foundation. Apache TinkerPop 3.3.3 API [online]. [Accessed 29 November 2018]. Available from: `http://tinkerpop.apache.org/javadocs/3.3.3/full/`

[72] Norvig, P. Teach Yourself Programming in Ten Years. Answers [online]. [Accessed 03 December 2018]. Available from: `http://norvig.com/21-days.html#answers`

[73] Majkowski, M. How to achieve low latency with 10Gbps Ethernet [online]. June 2015, [Accessed 03 December 2018]. Available from: `https://blog.cloudflare.com/how-to-achieve-low-latency/?fbclid=IwAR0jI6fqarXwpw0e0DYx0G65eEoVQCQXsNhnw_nzzjVjeV1AfEJEuEWSGa4`

[74] OrientDB Ltd. Issues – orientechnologies/orientdb [online]. *GitHub repository*, [Accessed 07 December 2018]. Available from: `https://github.com/orientechnologies/orientdb/issues/`

[75] svitaluc. svitaluc/tinkerpop: Mirror of Apache TinkerPop [online]. *GitHub repository*, [Accessed 08 December 2018]. Available from: `https://github.com/svitaluc/tinkerpop`

[76] JanusGraph. Releases – JanusGraph/jansugraph [online]. *GitHub repository*, October 2018, [Accessed 11 December 2018]. Available from: `https://github.com/JanusGraph/janusgraph/releases`

[77] The Apache Software Foundation. The Rat Maven Plugin [online]. [Accessed 24 December 2018]. Available from: `http://creadur.apache.org/rat/apache-rat-plugin/`

[78] Google, Inc. Google/GSON: A Java serialization/deserialization library to convert Java Objects into JSON and back [online]. *GitHub repository*, [Accessed 12 December 2018]. Available from: `https://github.com/google/gson`

[79] svitaluc. svitaluc/grm: Graph Redistribution Manager [online]. *GitHub repository*, [Accessed 20 December 2018]. Available from: `https://github.com/svitaluc/grm`

[80] McAuley, J.; Leskovec, J. Learning to discover social circles in ego networks. In *Advances in Neural Information Processing Systems*, 2012, [Accessed 1 October 2018]. Available from: `https://snap.stanford.edu/data/ego-Twitter.html`

[81] Leskovec, J.; Lang, K.; et al. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, volume 6, no. 1, 2009: pp. 29–123, [Accessed 1 October 2018]. Available from: `https://snap.stanford.edu/data/roadNet-PA.html`

[82] DataStax, Inc. Partition key [online]. [Accessed 21 December 2018]. Available from: `https://docs.datastax.com/en/glossary/doc/glossary/gloss_partition_key.html?fbclid=IwAR2rmU7QPwX3F2NzA6hqQqulSwLeUErd2fTrfYpRUfCSshx4Hpc4XvEw2RU`

[83] JanusGraph.org. Chapter 35. Graph Partitioning [online]. [Accessed 28 November 2018]. Available from: `https://docs.janusgraph.org/latest/graph-partitioning.html`

[84] Barabasi, A.-L.; Bonabeau, E. Scale–Free Networks. *Scientific American*, volume 288, no. 5, May 2003: pp. 50–59, doi:10.1038/scientificamerican0503-60.

[85] Patrikalakis, A. Finding ways to make a shortest path query run faster [online]. April 2016, [Accessed 21 December 2018]. Available

from: https://groups.google.com/forum/?fbclid=IwAR15q5tw9y_BvsSZigLueT5KHY5Y07aiTmUTnX-ons0hsXzkvoa7b7ZyBIs#!topic/gremlin-users/rhIhEY9R4E0

[86] JanusGraph.org. JanusGraph [online]. [Accessed 24 December 2018]. Available from: http://janusgraph.org/

Appendix **A**

README

A.1 Processed Result Logging

The following text of README file can be found on the GitHub web page [75], in branch `processed-result-log`, package `gremlin-processedResultLogg-ing`.

Processed Result Logging

Processed Result Logging enables to silently perform an additional operation on a result of a query and to log the outcome of this operation in a predefined format.

In order to successfully log an outcome of an additional operation, the operation must be supported for processed result logging. A List of supported processors is part of this readme, but it is not difficult to extend the specific interface with another processor (for further information see section For Developers).

It is possible to use a provided formatter or to create a custom implementation of a formatter in order to log various types of information. A List of provided formatters is part of this readme as well.

For business purposes, a processor can provide an anonymized log of the processed result to prevent sensitive data from being exposed. The List of supported processors contains the information if the processor supports anonymous logging or not. Some formats do not have to be yet compatible with anonymized logging.

List of supported processors

Currently supported result processors, their full class names and whether they support anonymized logging:

- PathProcessor
  - org.apache.tinkerpop.processedResultLogging.processor .PathProcessor
  - supports anonymized logging

For detailed information see section Processors.

List of provided formatters

Currently supported formatters for processed result logging, their full names and whether they are compatible with anonymized logging:

- basic
  - org.apache.tinkerpop.processedResultLogging.formatter .BasicProcessedResultFormatter
  - is compatible with anonymized logging
- query
  - org.apache.tinkerpop.processedResultLogging.formatter .QueryProcessedResultFormatter
  - is not yet compatible with anonymized logging

For description of the formatters see section Formatters.

Settings

In order to set a processed result logging, corresponding options must be set in a configuration.

When used in Gremlin Server environment, the configuration is loaded from the file `conf/gremlin-server.yaml`. The key to a set of this module properties is `processedResultLog`.

Subkey	Description	Default
enabled	Enables Processed Result Logging in Gremlin Server environment. In order to start Processed Result Logging, set this property to 'true'.	false
asyncMode	Determines whether the logging should be done in a separate thread or in the current thread.	false
processor	An optional property determining a processor to be run on a result. Argument of the `processor` property is the fully qualified classname of a class providing the processed result logging. It is a class implementing the ResultProcessor interface.	org.apache .tinkerpop .processed-ResultLogging .processor .PathProcessor
formatter	An optional property setting a log format. Argument of the `formatter` property is the fully qualified classname of a class providing the formatter. It is a class implementing the ProcessedResultFormatter interface. See the [List of provided formats] and [Formats] section for further information.	org.apache .tinkerpop .processed-ResultLogging .formatter.Basic-ProcessedResultFormatter
anonymized	An optional property setting an anonymized log. An anonymized log should not contain any sensitive information about data stored in a graph.	false
localMode	If set to false, the environment is assumed to be Gremlin Server and the logging will be handled through its `sl4j` logging backend. If true, the logging is done through `SimpleLogger` class implemented in this package and it is necessary to use `ProcessedResultLoggingStrategy` with the `Traversal` from which the results are supposed to be logged.	true

The configuration is stored in the class `ProcessedResultManager.Settings` which can be injected to the `ProcessedResultManager` singleton instance.

Example

Settings for Processed Result Logging with the PathProcessor and the QueryFormatter can, for example, look as follows

```
processedResultLog: {
   enabled: true
   processor: org.apache.tinkerpop.processedResultLogging.processor.
      PathProcessor
   formatter: org.apache.tinkerpop.processedResultLogging.formatter.
      QueryProcessedResultFormatter
}
```

Processor property does not have to be mentioned in this case as the PathProcessor is a default value. For an anonymized property we used false value (by default).

Processors

PathProcessor

If a query result is of a GraphTraversal type, a path method is silently performed on the result and its outcome is logged. For description of the path-step see Documentation.

Example of a basic path output:

```
v[1], e[7][1-knows->2], v[2], vadas
```

anonymized

An anonymized version of the PathProcessor produces output without the result and edge label.

Example of a basic path output in the anonymized mode:

```
v:1,e:7,v:2
```

Formatters

BasicProcessedResultFormatter

Basic formatter is logging only a processed result without any other piece of information.

Example of basic format output:

```
v[1], e[7][1-knows->2], v[2], vadas
v[1], e[8][1-knows->4], v[4], josh
```

QueryProcessedResultFormatter

Query formatter first logs the query and then its processed results. In order to determine the two types of information, query is preceded with `#QUERY:` line and the set of processed results is preceded with `#PR:` line. This format is not yet compatible with anonymized logging.

Example of basic format output:

```
#QUERY:
g.V(1).outE('knows').inV().values('name')
#PR:
v[1], e[7][1-knows->2], v[2], vadas
v[1], e[8][1-knows->4], v[4], josh
```

LLOJsonFormatter

LLO JSON formatter logs the query and its processed results in a JSON format. This formatter is implemented to work with `PathProcessor` which returns `LLOProcessedResult`. The serialization to JSON is implemented in `LLOProcessedResult.Serializer` and will work with Gson library. This format is not compatible with anonymized logging.

Example of LLO JSON format output:

```
{
   Q:"g.V(1).outE('knows').inV().values('name')",
   R:[
      ["v":1,"e":7,"v":2,"person":"vadas"],
      ["v":1,"e":8,"v":4,"person":"josh"]
   ]
}
```

For Developers

New extensions of Processed Result Logging are welcomed, but please follow the rules below.

processor

If you want to support a new method, create a class called `<method-name> Processor` which implements the interface `ResultProcessor` and locate it within `processor` package.

If you want to create an anonymized version of an output as well, your class should implement the interface `AnonymizedResultProcessor` which already extends the `ResultProcessor` interface. Please make sure that your anonymized version of an output truly cannot contain any sensitive information.

formatter

If you want to create a new formatter, create a class called `<formatter-name> Formatter` which implements the interface `ProcessedResultFormatter` and locate it within `formatter` package.

Formatter has two parameters - a `String` and `ProcessedResult`. The `String` represents the query in its original form. If you want to implement an anonymized log, make sure that your result does not contain the query or that any sensitive information has been removed from it.

A.2 Graph Redistribution Manager

The following text of README file can be found on the GitHub web page [79].

Graph Redistribution Manager

Graph Redistribution Manager (GRM) is a proof of concept solution of data redistribution in distributed graph database systems. The process of redistribution starts with loading a dataset in JanusGraph database. Then, a set of generated queries is ran against the database, generating a processed result log via the custom Processed Result Logging module of the TinkerPop project. The paths from the log are added to the database. Last but not least, a partitioning algorithm is executed and a new partition for each vertex is proposed.

VaqueroVertexProgram

`VaqueroVertexProgram` is the Vaquero et al. [37] algorithm implemented in the TinkerPop `VertexProgram` framework which is bases on Pregel computation system. Therefore, the VaqueroVertexProgram works in iterations. After initial setup, the `execute` method is ran for each vertex of the graph. These iterations are ran in parallel manner, which also means that after each iteration, there is a synchronization barrier and the `terminate` method is called. The `terminate` method determines whether the algorithm should continue into the next iteration or whether the program should halt. After the end of the execution, each vertex is assigned a new partition ID which can be accessed via the `vertex.value(PARTITION)` method. The `PARTITION` is a constant with the key of the property of the new partition ID.

Iteration process

In each iteration, vertices communicate with each other via messages. Every vertex sends its partition ID (PID) to all of its neighbors. A new partition ID of the vertex is determined by the frequency of PIDs of the vertex neighbors. The more frequently the PID is represented among the neighbors, the higher the chance that it will be acquired. However, there are more conditions which determine whether the new PID will be acquired. One of them is the current balance of the partitions. The `imbalanceFactor` sets the lower bound of how much the partitions can be imbalanced regarding a proportional vertex distribution. The upper bound of each partition is determined by its maximum capacity. Another condition solves an oscillation problem of two or more vertices changing their PID back and forth.

Datasets

The current implementation supports two datasets for testing the Vaquero et al. algorithm.

- Twitter dataset - [80]
- Pennsylvania road network dataset - [81]

Loading and query testing is implemented via two classes: `PennsylvaniaDatasetLoaderQueryRunner` and `TwitterDatasetLoaderQueryRunner`.

Configuration and usage

The `VaqueroVertexProgram` can be built with the `VaqueroVertexProgram.Builder` which enables configuration of the algorithm.

```
vertexProgram = VaqueroVertexProgram.build()
                  .clusterMapper(cm)
                  .acquirePartitionProbability(0.5)
                  .imbalanceFactor(0.80)
                  .coolingFactor(0.99)
                  .adoptionFactor(1)
                  .evaluatingMap(runner.evaluatingMap())
                  .evaluateCrossCommunication(true)
                  .scopeIncidentTraversal(__.outE())
                  .evaluatingStatsOriginal(runner.evaluatingStats())
                  .maxPartitionChangeRatio(1)
                  .maxIterations(200)
                  .create(graph);
algorithmResult = graph.compute().program(vertexProgram).workers(12)
                     .submit().get();
```

In the code snippet above, an instance of the `VaqueroVertexProgram` is created with the `VaqueroVertexProgram.Builder`, which is then submitted to the `GraphComputer`.

Configuration

- `clusterMapper` - sets the provided `ClusterMapper` to determine the original partition ID from a vertex ID.
- `acquirePartitionProbability` - sets the probability for vertices to acquire a new partition.
- `imbalanceFactor` - sets the imbalance factor which determines how much the partitions can be imbalanced regarding the lower bound.
- `coolingFactor` - simulated annealing is used when executing the program. The cooling factor determines how quickly the temperature is going to drop, affecting the probability of vertices, which partition to acquire.
- `adoptionFactor` - determines how to calculate the probability of acquiring a new partition.
- `evaluatingMap` - provides a map of vertices to number of their appearance in the query set. It is used for evaluating the improvement of the partitioning between iterations.
- `evaluateCrossCommunication` - sets whether or not to enable cross-node communication evaluation between iterations.
- `scopeIncidentTraversal` - sets the incident vertices to which to send a message (`__.outE()` means that the messages will be sent to incident vertices using only out-going edges).
- `evaluatingStatsOriginal` - sets the initial statistics of cross-node communication.
- `maxPartitionChangeRatio` - sets the limit of how many vertices can change partition during one iteration (input is a fraction of the vertex count).
- `maxIterations` - sets the maximum number of iterations before the program halts.

Another important configuration is selecting the proper graph configuration file. In `config.properties` you can currently state either `graph-berkeley-embeded.properties` or `graph-cassandra-embedded.properties` for the `graph.propFile` property. This provides the Berkeley DB or the embedded Apache Cassandra storage backend, respectively.

Project structure

logHandling

- `PRLogFileLoader` - loads a log file in the memory or provides an `Iterator<PRLogRecord>` of the log file.
- `PRLog` - represents a log file when loaded in the memory.
- `PRLogRecord` - represents all paths generated by one `Traverser`, that is, by one query.
- `PRPath` - list of `PRElement` s representing one path of the `Traverser`.
- `PRElement` - represents an edge or a vertex of the graph.
- `PRLogToGraphLoader` - this interface describes operations necessary for loading a log in the graph, which are `addSchema`, `removeSchema` and `loadLogToGraph`.
- `DefaultPRLogToGraphLoader` - the default implementation of the `PRLogToGraphLoader` loading the `PRLog` to a given graph.

dataset

- `DatasetQueryRunner` - defines an interface to run a set of queries against the database.
- `DatasetLoader` - defines an interface to load a dataset in the graph database.
- `PennsylvaniaDatasetLoaderQueryRunner` - implements both the interfaces above. Handles the Pennsylvania road network dataset.
- `TwitterDatasetLoaderQueryRunner` - implements both the interfaces above. Handles the Twitter network dataset.

cluster

- `PartitionMapper` - this interface defines the method `map(vertexID)` to get the partition ID from `vertexID`.
- `DefaultPartitionMapper` - a default implementation of the `PartitionMapper`, which uses modulo division of the hash of `vertexID` (that was before converted to `originalID` via `IDManager.fromVertexId`) to get the partition ID.

helpers

- `HelperOperator` - a set of binary operators to use when reducing data in the `VaqueroVertexProgram`.
- `ShuffleComparator` - a random comparator which is used for traversal ordering when implementing a random walk on the graph.

GRMP

Executable class that runs the complete ”benchmark” of the Pennsylvania road network dataset.

GRMT

Executable class that runs the complete ”benchmark” of the Twitter network dataset.

GRM

The base class of the `GRMP` and the `GRMT`, containing shared components and resources.

Appendix B

Evaluation Results

B.1 Twitter Dataset

B.1.1 Cooling Process

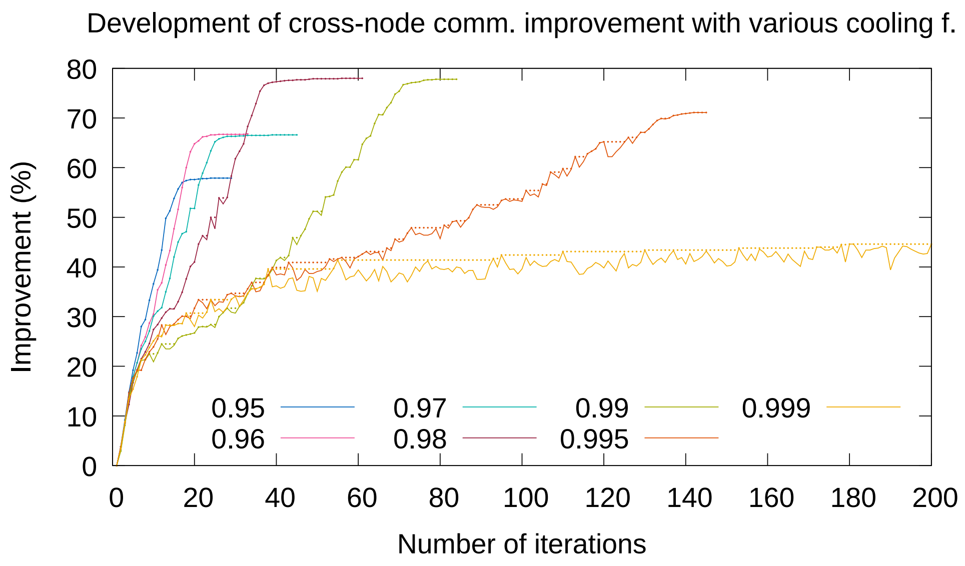


Figure B.1: Development of cross-node communication improvement with various cooling factors. The dots represent the maximal value of the process so far. Measured on the Twitter dataset.

B.1.2 Effects of Cooling Factor

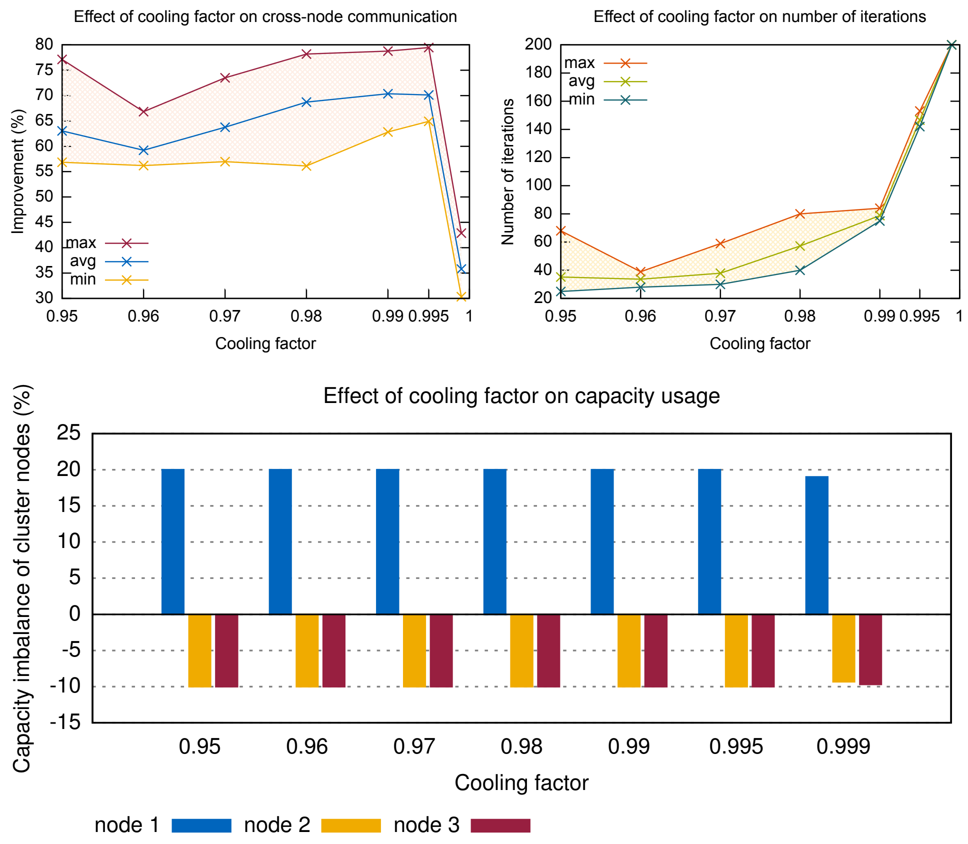


Figure B.2: Effects of cooling factor on improvement, number of iterations and capacity usage, respectively. Measured on the Twitter dataset.

cooling factor	minimal improvement	maximal improvement	average	standard deviation
0.95	56.84%	77.13%	63.03%	5.94%
0.96	56.19%	66.83%	59.22%	3.01%
0.97	56.97%	73.50%	63.76%	5.51%
0.98	56.12%	78.18%	68.72%	7.53%
0.99	62.82%	78.77%	70.35%	5.17%
0.995	64.93%	79.47%	70.12%	3.81%
0.999	30.34%	42.88%	35.80%	6.33%

Table B.1: Effect of cooling factor on cross-node communication. The improvement is computed as $1 - \frac{\text{\# of after-crossings}}{\text{\# of before-crossings}}$. Measured with *imbalance factor* = 0.1, *adoption factor* = 1, *number of clusters* = 3. The results are based on ten individual measurements for each factor value, on the Twitter dataset.

cooling factor	minimal # of iterations	maximal # of iterations	average	standard deviation
0.95	25	68	35.2	12.6
0.96	28	39	33.7	3.5
0.97	30	59	37.9	8.6
0.98	40	80	57.2	14.2
0.99	75	84	78.8	3.2
0.995	142	153	146.6	3.5
0.999	200	200	200.0	0.0

Table B.2: Effect of cooling factor on number of iterations. Measured with *imbalance factor* = 0.1, *adoption factor* = 1, *number of clusters* = 3. The results are based on ten individual measurements for each factor value, on the Twitter dataset.

cooling factor	N1	N2	N3	$\frac{N1-avg}{avg}$	$\frac{N2-avg}{avg}$	$\frac{N3-avg}{avg}$
0.95	30,506	22,881	22,881	20.00%	-10.00%	-10.00%
0.96	30,506	22,881	22,881	20.00%	-10.00%	-10.00%
0.97	30,506	22,881	22,881	20.00%	-10.00%	-10.00%
0.98	30,506	22,881	22,881	20.00%	-10.00%	-10.00%
0.99	30,506	22,881	22,881	20.00%	-10.00%	-10.00%
0.995	30,506	22,881	22,881	20.00%	-10.00%	-10.00%
0.999	30,253	23,054	22,961	19.00%	-9.68%	-9.32%

Table B.3: Effect of cooling factor on capacity usage. The average number of vertices per node, $\frac{|V|}{|N|}$, is equal to approximately $25,423$. The results are based on ten individual measurements for each factor value, on the Twitter dataset.

B.1.3 Imbalance Factor

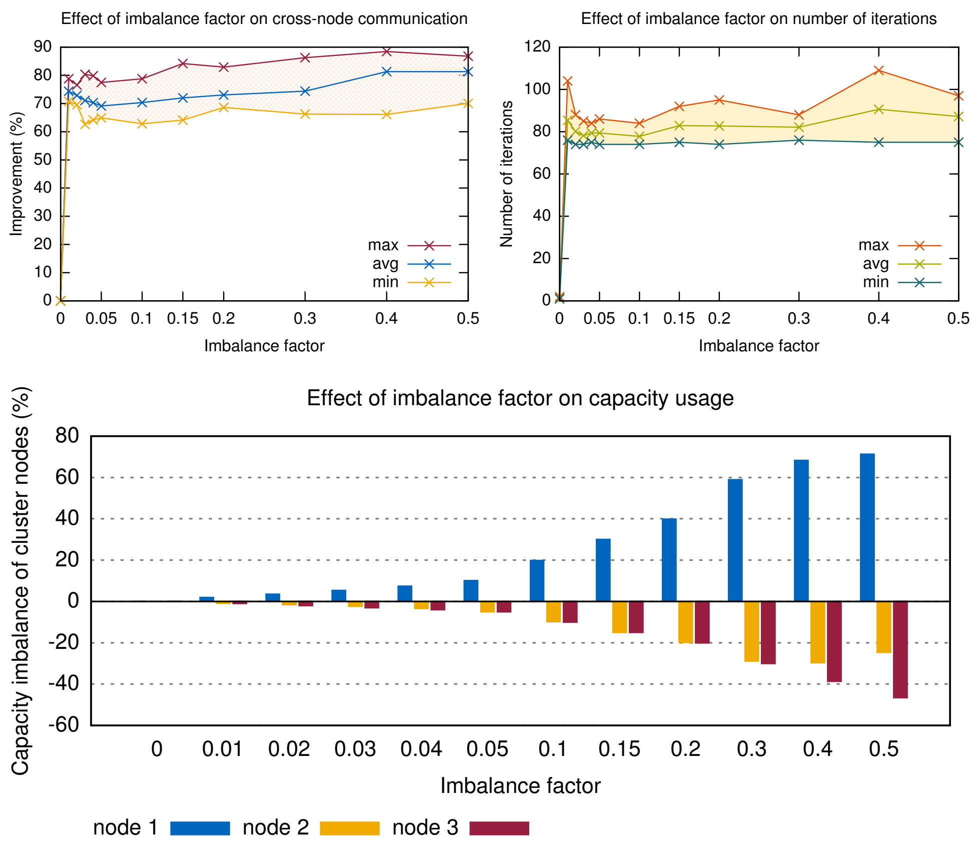


Figure B.3: Effects of imbalance factor on improvement, number of iterations and capacity usage, respectively. Measured on the Twitter dataset.

imbalance factor	minimal improvement	maximal improvement	average	standard deviation
0	0.01%	0.01%	0.01%	0.00%
0.01	70.73%	78.77%	74.36%	2.50%
0.02	69.58%	76.58%	72.88%	2.69%
0.03	62.59%	80.34%	71.09%	5.62%
0.04	64.18%	79.80%	70.29%	4.81%
0.05	64.89%	77.49%	69.13%	4.02%
0.1	62.82%	78.77%	70.35%	5.17%
0.15	64.10%	84.16%	72.01%	6.24%
0.2	68.61%	82.91%	73.05%	4.53%
0.3	66.24%	86.24%	74.39%	6.03%
0.4	66.14%	88.42%	81.32%	7.04%
0.5	70.03%	86.77%	81.29%	4.54%

Table B.4: Effect of imbalance factor on cross-node communication. The improvement is computed as $1 - \frac{\text{\# of after-crossings}}{\text{\# of before-crossings}}$. Measured with *cooling factor* = 0.99, *adoption factor* = 1, *number of clusters* = 3. The results are based on ten individual measurements for each factor value, on the Twitter dataset.

imbalance factor	minimal # of iterations	maximal # of iterations	average	standard deviation
0	1	2	1.2	0.42
0.01	76	104	85.4	9.63
0.02	74	88	80.1	4.68
0.03	74	85	78.1	3.87
0.04	75	84	79.4	3.17
0.05	74	86	79.4	4.33
0.1	74	84	77.8	2.90
0.15	75	92	82.9	5.80
0.2	74	95	82.7	7.82
0.3	76	88	82.1	4.98
0.4	75	109	90.6	11.08
0.5	75	97	87.2	7.63

Table B.5: Effect of imbalance factor on number of iterations. Measured with *cooling factor* = 0.99, *adoption factor* = 1, *number of clusters* = 3. The results are based on ten individual measurements for each factor value, on the Twitter dataset.

imbalance factor	N1	N2	N3	$\frac{N1-avg}{avg}$	$\frac{N2-avg}{avg}$	$\frac{N3-avg}{avg}$
0	25,423	25,423	25,422	0.00%	-0.00%	-0.00%
0.01	25,896	25,202	25,170	1.86%	-0.87%	-0.99%
0.02	26,312	25,041	24,915	3.50%	-1.50%	-2.00%
0.03	26,764	24,844	24,660	5.27%	-2.27%	-3.00%
0.04	27,291	24,571	24,406	7.35%	-3.35%	-4.00%
0.05	27,964	24,152	24,152	10.00%	-5.00%	-5.00%
0.1	30,443	22,944	22,881	19.75%	-9.75%	-10.00%
0.15	33,048	21,610	21,610	29.99%	-15.00%	-15.00%
0.2	35,548	20,381	20,339	39.83%	-19.83%	-20.00%
0.3	40,385	18,087	17,796	58.85%	-28.85%	-30.00%
0.4	42,763	17,902	15,603	68.21%	-29.58%	-38.63%
0.5	43,526	19,162	13,580	71.21%	-24.63%	-46.58%

Table B.6: Effect of imbalance factor on capacity usage. The average number of vertices per node, $\frac{|V|}{|N|}$, is equal to approximately 25,423. The results are based on ten individual measurements for each factor value, on the Twitter dataset.

B.1.4 Adoption Factor

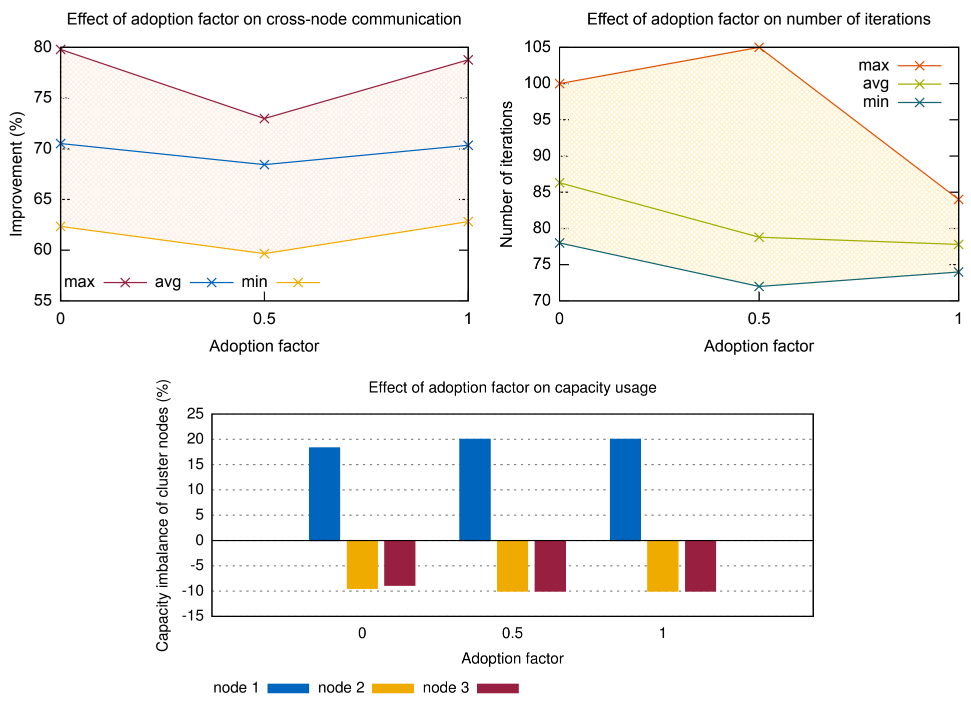


Figure B.4: Effects of adoption factor on improvement, number of iterations and capacity usage, respectively. Measured on the Twitter dataset.

adoption factor	minimal improvement	maximal improvement	average	standard deviation
0	62.35%	79.79%	70.51%	6.25%
0.5	59.66%	72.98%	68.44%	4.13%
1	62.82%	78.77%	70.35%	5.17%

Table B.7: Effect of adoption factor on cross-node communication. The improvement is computed as $1 - \frac{\text{\# of after-crossings}}{\text{\# of before-crossings}}$. Measured with *cooling factor* = 0.99, *imbalance factor* = 0.1, *number of clusters* = 3. The results are based on ten individual measurements for each factor value, on the Twitter dataset.

adoption factor	minimal # of iterations	maximal # of iterations	average	standard deviation
0	78	100	86.3	8.14
0.5	72	105	78.8	9.76
1	74	84	77.8	2.90

Table B.8: Effect of adoption factor on number of iterations. Measured with *cooling factor* = 0.99, *imbalance factor* = 0.1, *number of clusters* = 3. The results are based on ten individual measurements for each factor value, on the Twitter dataset.

adoption factor	N1	N2	N3	$\frac{N1-avg}{avg}$	$\frac{N2-avg}{avg}$	$\frac{N3-avg}{avg}$
0	30,220	22,973	23,075	18.98%	-9.67%	-9.31%
0.5	30,506	22,881	22,881	20.00%	-10.00%	-10.00%
1	30,506	22,881	22,881	20.00%	-10.00%	-10.00%

Table B.9: Effect of adoption factor on capacity usage. The average number of vertices per node, $\frac{|V|}{|N|}$, is equal to approximately $25,423$. The results are based on ten individual measurements for each factor value, on the Twitter dataset.

B.1.5 Number of Nodes in a Cluster

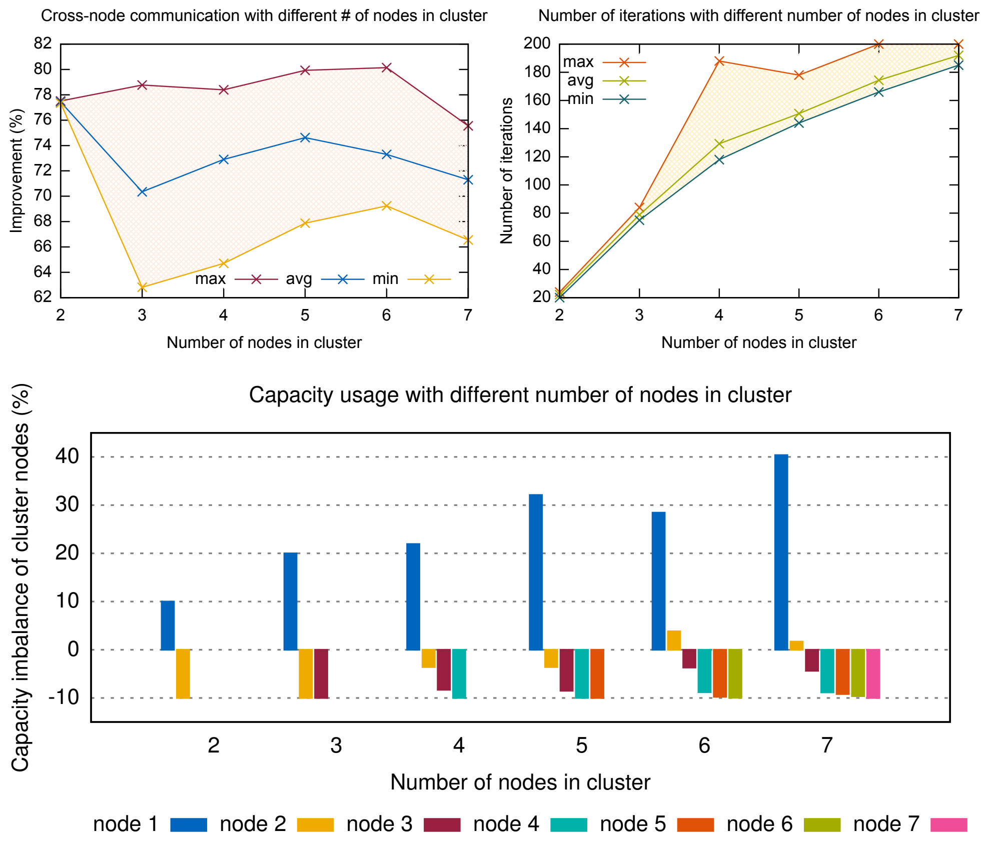


Figure B.5: Effects of number of nodes in a cluster on improvement, number of iterations and capacity usage, respectively. Measured on the Twitter dataset.

number of nodes	minimal improvement	maximal improvement	average	standard deviation
2	77.34%	77.51%	77.44%	0.06%
3	62.82%	78.77%	70.35%	5.17%
4	64.71%	78.39%	72.90%	4.94%
5	67.88%	79.93%	74.62%	3.21%
6	69.24%	80.14%	73.30%	3.02%
7	66.55%	75.55%	71.30%	2.91%

Table B.10: Cross-node communication with different number of nodes in cluster. The improvement is computed as $1 - \frac{\#\text{ of after-crossings}}{\#\text{ of before-crossings}}$. Measured with *cooling factor* = 0.99, *imbalance factor* = 0.1, *adoption factor* = 1. The results are based on ten individual measurements for each number of nodes, on the Twitter dataset.

number of nodes	minimal # of iterations	maximal # of iterations	average	standard deviation
2	20	24	22.20	1.14
3	75	84	78.83	3.19
4	118	188	129.30	21.11
5	144	178	150.70	10.15
6	166	200	174.30	12.04
7	185	200	192.00	4.83

Table B.11: Number of iterations with different number of nodes in cluster. Measured with *cooling factor* = 0.99, *imbalance factor* = 0.1, *adoption factor* = 1. The results are based on ten individual measurements for each number of nodes, on the Twitter dataset.

number of nodes	$\frac{N1-avg}{avg}$	$\frac{N2-avg}{avg}$	$\frac{N3-avg}{avg}$	$\frac{N4-avg}{avg}$	$\frac{N5-avg}{avg}$	$\frac{N6-avg}{avg}$	$\frac{N7-avg}{avg}$
2	10.00%	-10.00%					
3	20.00%	-10.00%	-10.00%				
4	21.94%	-3.61%	-8.33%	-10.00%			
5	32.10%	-3.59%	-8.53%	-10.00%	-10.00%		
6	28.46%	3.81%	-3.69%	-8.82%	-9.77%	-9.99%	
7	40.40%	1.70%	-4.39%	-8.87%	-9.20%	-9.64%	-10.00%

Table B.12: Capacity usage with different number of nodes in cluster. The results are based on ten individual measurements for each number of nodes, on the Twitter dataset.

B.2 Road Network Dataset

B.2.1 Effects of Cooling Factor

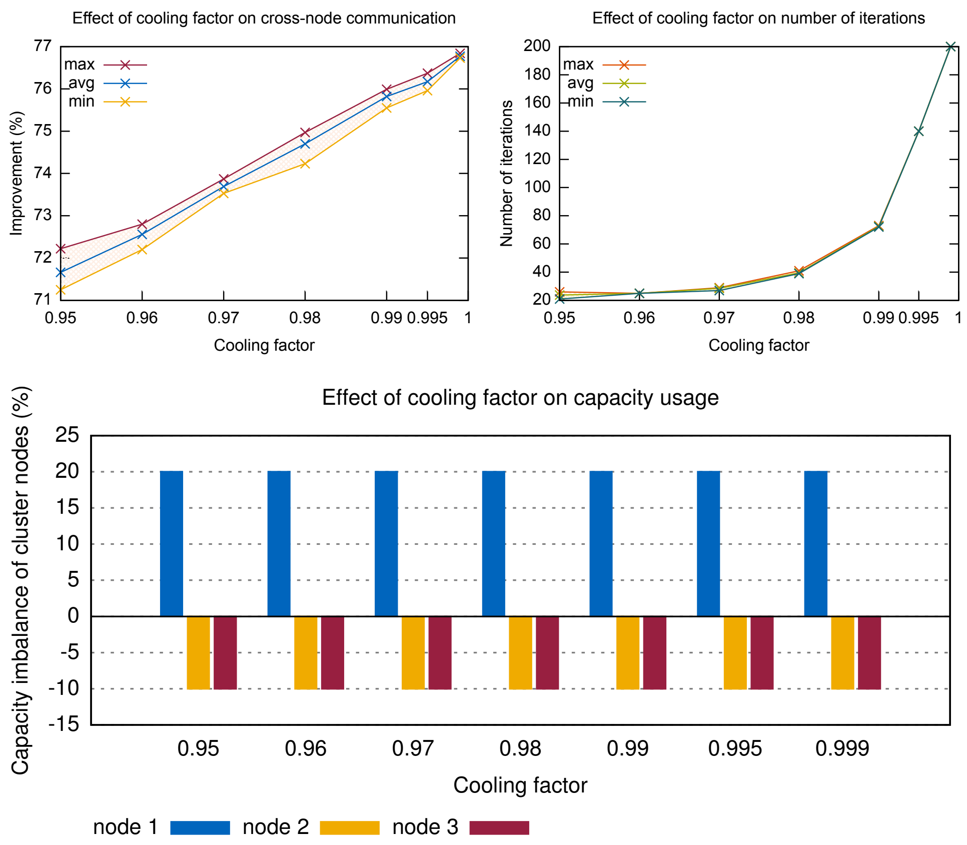


Figure B.6: Effects of cooling factor on improvement, number of iterations and capacity usage, respectively. Measured on the Road network dataset.

cooling factor	minimal improvement	maximal improvement	average	standard deviation
0.95	71.25%	72.22%	71.66%	0.41%
0.96	72.20%	72.80%	72.56%	0.26%
0.97	73.53%	73.87%	73.69%	0.14%
0.98	74.23%	74.97%	74.70%	0.33%
0.99	75.55%	75.99%	75.82%	0.21%
0.995	75.96%	76.37%	76.17%	0.21%
0.999	76.73%	76.84%	76.77%	0.05%

Table B.13: Effect of cooling factor on cross-node communication. The improvement is computed as $1 - \frac{\text{\# of after-crossings}}{\text{\# of before-crossings}}$. Measured with *imbalance factor* $= 0.1$, *adoption factor* $= 1$, *number of clusters* $= 3$. The results are based on four individual measurements for each factor value, on the Road network dataset.

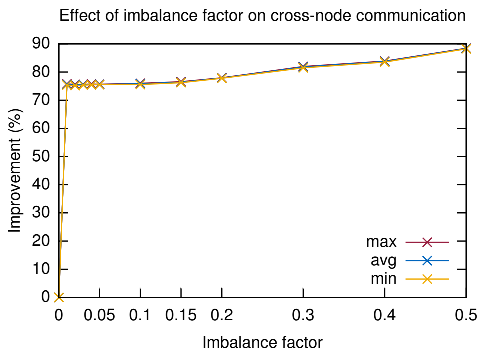
cooling factor	minimal # of iterations	maximal # of iterations	average	standard deviation
0.95	21	26	23.75	2.06
0.96	25	25	25.00	0.00
0.97	27	29	28.50	1.00
0.98	39	41	39.50	1.00
0.99	72	73	72.50	0.58
0.995	140	140	140.00	0.00
0.999	200	200	200.00	0.00

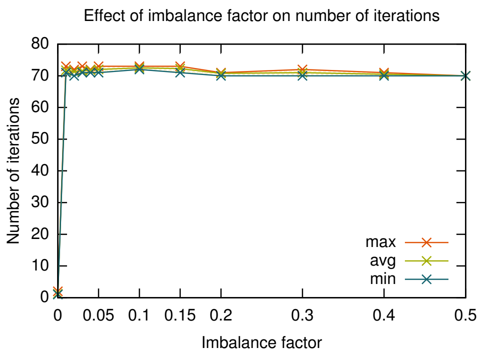
Table B.14: Effect of cooling factor on number of iterations. Measured with *imbalance factor* $= 0.1$, *adoption factor* $= 1$, *number of clusters* $= 3$. The results are based on four individual measurements for each factor value, on the Road network dataset.

cooling factor	N1	N2	N3	$\frac{N1-avg}{avg}$	$\frac{N2-avg}{avg}$	$\frac{N3-avg}{avg}$
0.95	435,236	326,428	326,428	20.00%	-10.00%	-10.00%
0.96	435,236	326,428	326,428	20.00%	-10.00%	-10.00%
0.97	435,236	326,428	326,428	20.00%	-10.00%	-10.00%
0.98	435,236	326,428	326,428	20.00%	-10.00%	-10.00%
0.99	435,236	326,428	326,428	20.00%	-10.00%	-10.00%
0.995	435,236	326,428	326,428	20.00%	-10.00%	-10.00%
0.999	435,236	326,428	326,428	20.00%	-10.00%	-10.00%

Table B.15: Effect of cooling factor on capacity usage. The average number of vertices per node, $\frac{|V|}{|N|}$, is equal to approximately $362,697$. The results are based on four individual measurements for each factor value, on the Road network dataset.

B.2.2 Imbalance Factor





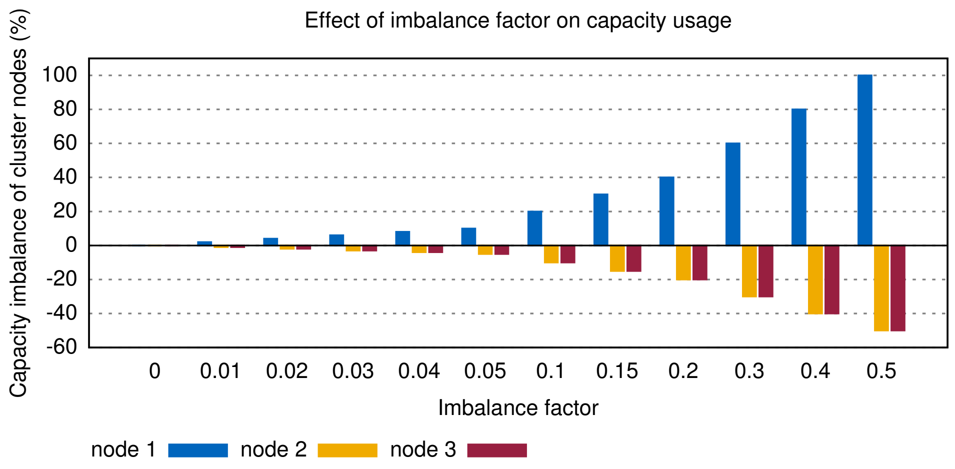


Figure B.7: Effects of imbalance factor on improvement, number of iterations and capacity usage, respectively. Measured on the Road network dataset.

imbalance factor	minimal improvement	maximal improvement	average	standard deviation
0	0.01%	0.01%	0.01%	0.00%
0.01	75.39%	75.77%	75.59%	0.16%
0.02	75.18%	75.58%	75.38%	0.20%
0.03	75.35%	75.62%	75.53%	0.13%
0.04	75.45%	75.72%	75.52%	0.13%
0.05	75.51%	75.66%	75.59%	0.07%
0.1	75.55%	75.99%	75.82%	0.21%
0.15	76.21%	76.54%	76.39%	0.16%
0.2	77.77%	77.93%	77.87%	0.07%
0.3	81.46%	81.96%	81.79%	0.23%
0.4	83.54%	83.88%	83.73%	0.14%
0.5	88.22%	88.45%	88.34%	0.11%

Table B.16: Effect of imbalance factor on cross-node communication. The improvement is computed as $1 - \frac{\text{\# of after-crossings}}{\text{\# of before-crossings}}$. Measured with *cooling factor* = 0.99, *adoption factor* = 1, *number of clusters* = 3. The results are based on four individual measurements for each factor value, on the Road network dataset.

imbalance factor	minimal # of iterations	maximal # of iterations	average	standard deviation
0	1	2	1.25	0.50
0.01	71	73	71.75	0.96
0.02	70	72	71.33	1.15
0.03	71	73	71.50	1.00
0.04	71	72	71.75	0.50
0.05	71	73	72.00	1.15
0.1	72	73	72.50	0.58
0.15	71	73	72.25	0.96
0.2	70	71	70.75	0.50
0.3	70	72	71.00	0.82
0.4	70	71	70.50	0.58
0.5	70	70	70.00	0.00

Table B.17: Effect of imbalance factor on number of iterations. Measured with *cooling factor* = 0.99, *adoption factor* = 1, *number of clusters* = 3. The results are based on four individual measurements for each factor value, on the Road network dataset.

imbalance factor	N1	N2	N3	$\frac{N1-avg}{avg}$	$\frac{N2-avg}{avg}$	$\frac{N3-avg}{avg}$
0	362,698	362,697	362,697	0.00%	0.00%	0.00%
0.01	369,950	359,071	359,071	2.00%	-1.00%	-1.00%
0.02	377,204	355,444	355,444	4.00%	-2.00%	-2.00%
0.03	384,458	351,817	351,817	6.00%	-3.00%	-3.00%
0.04	391,712	348,190	348,190	8.00%	-4.00%	-4.00%
0.05	398,966	344,563	344,563	10.00%	-5.00%	-5.00%
0.1	435,236	326,428	326,428	20.00%	-10.00%	-10.00%
0.15	471,506	308,293	308,293	30.00%	-15.00%	-15.00%
0.2	507,776	290,158	290,158	40.00%	-20.00%	-20.00%
0.3	580,314	253,889	253,889	60.00%	-30.00%	-30.00%
0.4	652,854	217,619	217,619	80.00%	-40.00%	-40.00%
0.5	725,394	181,349	181,349	100.00%	-50.00%	-50.00%

Table B.18: Effect of imbalance factor on capacity usage. The average number of vertices per node, $\frac{|V|}{|N|}$, is equal to approximately $362,697$. The results are based on four individual measurements for each factor value, on the Road network dataset.

B.2.3 Adoption Factor

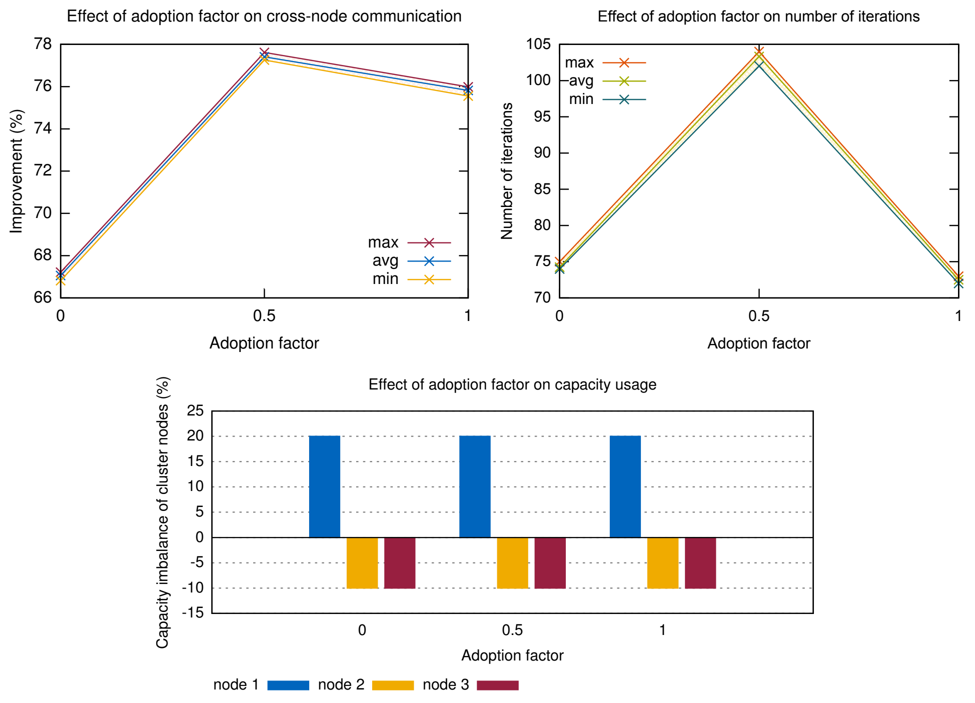


Figure B.8: Effects of adoption factor on improvement, number of iterations and capacity usage, respectively. Measured on the Road network dataset.

adoption factor	minimal improvement	maximal improvement	average	standard deviation
0	66.82%	67.22%	67.06%	0.17%
0.5	77.26%	77.62%	77.41%	0.19%
1	75.55%	75.99%	75.82%	0.21%

Table B.19: Effect of adoption factor on cross-node communication. The improvement is computed as $1 - \frac{\text{\# of after-crossings}}{\text{\# of before-crossings}}$. Measured with *cooling factor* = 0.99, *imbalance factor* = 0.1, *number of clusters* = 3. The results are based on four individual measurements for each factor value, on the Road network dataset.

adoption factor	minimal # of iterations	maximal # of iterations	average	standard deviation
0	74	75	74.25	0.50
0.5	102	104	103.33	1.15
1	72	73	72.50	0.58

Table B.20: Effect of adoption factor on number of iterations. Measured with *cooling factor* = 0.99, *imbalance factor* = 0.1, *number of clusters* = 3. The results are based on four individual measurements for each factor value, on the Road network dataset.

adoption factor	N1	N2	N3	$\frac{N1-avg}{avg}$	$\frac{N2-avg}{avg}$	$\frac{N3-avg}{avg}$
0	435,236	326,428	326,428	20.00%	-10.00%	-10.00%
0.5	435,236	326,428	326,428	20.00%	-10.00%	-10.00%
1	435,236	326,428	326,428	20.00%	-10.00%	-10.00%

Table B.21: Effect of adoption factor on capacity usage. The average number of vertices per node, $\frac{|V|}{|N|}$, is equal to approximately $362,697$. The results are based on four individual measurements for each factor value, on the Road network dataset.

B.2.4 Number of Nodes in a Cluster

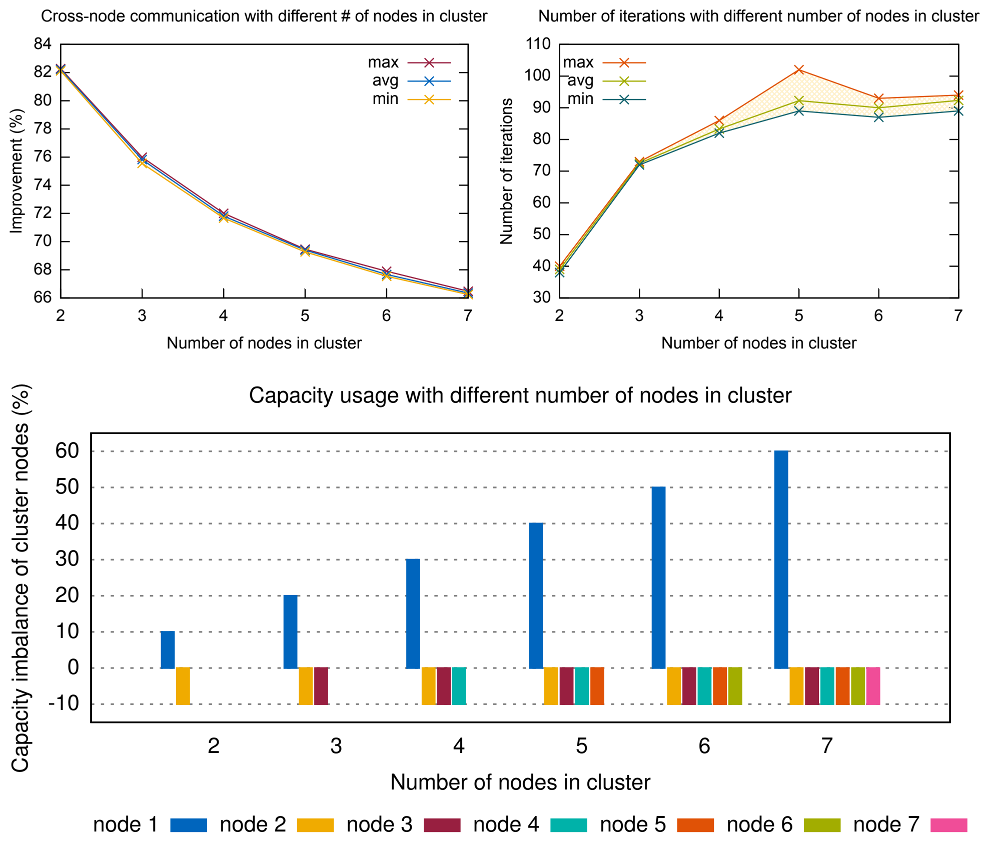


Figure B.9: Effects of number of nodes in a cluster on improvement, number of iterations and capacity usage, respectively. Measured on the Road network dataset.

number of nodes	minimal improvement	maximal improvement	average	standard deviation
2	82.12%	82.28%	82.22%	0.09%
3	75.55%	75.99%	75.82%	0.21%
4	71.67%	72.01%	71.81%	0.18%
5	69.27%	69.46%	69.40%	0.11%
6	67.54%	67.90%	67.67%	0.20%
7	66.24%	66.48%	66.35%	0.12%

Table B.22: Cross-node communication with different number of nodes in cluster. The improvement is computed as $1 - \frac{\text{\# of after-crossings}}{\text{\# of before-crossings}}$. Measured with *cooling factor* = 0.99, *imbalance factor* = 0.1, *adoption factor* = 1. The results are based on four individual measurements for each number of nodes, on the Road network dataset.

number of nodes	minimal # of iterations	maximal # of iterations	average	standard deviation
2	38	40	39.00	1.00
3	72	73	72.50	0.58
4	82	86	83.33	2.31
5	89	102	92.25	6.50
6	87	93	90.00	3.00
7	89	94	92.33	2.89

Table B.23: Number of iterations with different number of nodes in cluster. Measured with *cooling factor* = 0.99, *imbalance factor* = 0.1, *adoption factor* = 1. The results are based on four individual measurements for each number of nodes, on the Road network dataset.

number of nodes	$\frac{N1-avg}{avg}$	$\frac{N2-avg}{avg}$	$\frac{N3-avg}{avg}$	$\frac{N4-avg}{avg}$	$\frac{N5-avg}{avg}$	$\frac{N6-avg}{avg}$	$\frac{N7-avg}{avg}$
2	10.0%	-10.0%					
3	20.0%	-10.0%	-10.0%				
4	30.0%	-10.0%	-10.0%	-10.0%			
5	40.0%	-10.0%	-10.0%	-10.0%	-10.0%		
6	50.0%	-10.0%	-10.0%	-10.0%	-10.0%	-10.0%	
7	60.0%	-10.0%	-10.0%	-10.0%	-10.0%	-10.0%	-10.0%

Table B.24: Capacity usage with different number of nodes in cluster. Measured with *cooling factor* = 0.99, *imbalance factor* = 0.1, *adoption factor* = 1. The results are based on four individual measurements for each number of nodes, on the Road network dataset.

Appendix **C**

Contents of CD

```
│ readme.txt ........................ the file with CD contents description
├── src .......................................the directory of source codes
│   ├── grm .............................. the directory of the GRM project
│   ├── tinkerpop ............the directory of the TinkerPop logging module
│   └── thesis ..............the directory of LaTeX source codes of the thesis
│       ├── figures ............................ the thesis figures directory
│       └── thesis.tex...............the LaTeX source code files of the thesis
├── data.xlsx...........................generated data of the evaluation
└── text .......................................... the thesis text directory
    ├── thesis.pdf ..................... the Diploma thesis in PDF format
    └── thesis-task.pdf ..................... the Thesis task in PDF format
```