RWTH AACHEN UNIVERSITY
Chair of Information Systems
Prof. Dr. Matthias Jarke

Algorithms for Large Networks in the NoSQL Database ArangoDB

Bachelor Thesis
by Lucas Dohmen
Matr.-Nr. 290333
September 25, 2012

Supervisors: Prof. Dr. Matthias Jarke
Chair of Information Systems
RWTH Aachen University

PD. Dr. Ralf Klamma, AOR
Chair of Information Systems
RWTH Aachen University

Advisors: Dr. Michael Derntl
Chair of Information Systems
RWTH Aachen University

Dr. Frank Celler
triAGENS GmbH

Declaration

I herewith declare with my signature, that I have written this bachelor thesis

"Algorithms for Large Networks in the NoSQL Database ArangoDB"

on my own, that all reference or assistance received during the writing of the thesis is stated completely, and that any citation is referenced to its source truly.

_____ Aachen, September 25, 2012

Lucas Dohmen

Contents

1 Introduction

1.1 Motivation

Database management systems face new challenges in their usage, resulting in the change of existing systems and the development of new solutions. Today they are challenged by the growing datasets of modern applications. The social network Twitter[1] for example had 100 million users in September 2011, with more than half of the users logging in every day[2]. In addition, different use cases benefit from more efficient ways to structure content data like graphs rather than modeling these datasets as relations. Emerging NoSQL Database Management Systems try to solve these problems by storing data in a different way and loosening the ACID principle [HaRe83]. In modern web applications social features are very important: Graphs can be used to model objects and interactions, for example in a social network the vertices could represent the members of the network and the edges could represent their ties of friendship.

It can be unwieldy to represent and query graph structures using a relational database. On the other hand, existing solutions that store graphs natively complicate queries for certain cases that do not depend on the structure of the graph. *ArangoDB* – a new NoSQL Database Management System – offers built-in support for graphs in the database in addition to a document store: The data can be structured as a graph with vertices and edges with flexible number of attributes while content data can be searched with a SQL-like query language. ArangoDB is currently in the beta for the release of version 1.0.

1.2 Thesis Goals

The goals for this thesis are:

1. Implement algorithms to find the shortest path between two vertices, measure the centrality of vertices in the given graph and determine the similarity of two vertices in ArangoDB.
2. Verify the correctness of the implementation by comparison with existing tools.
3. Compare the performance of the implementation and the performance when searching for non-structural data (e.g. content attached to vertices) with existing tools.

[1] `http://twitter.com`
[2] `http://blog.twitter.com/2011/09/one-hundred-million-voices.html`

2 State of the Art

To achieve the thesis goals, we need to know the characteristics of complex networks to compare the systems using realistic data. We will also consider four kinds of information demands for graphs stored in a database and review existing solutions that meet them.

2.1 Graphs and Scale Free Networks

We define a graph $G = (V, E)$ as a set of vertices V and a set of edges E. An edge $e \in E$ connects two vertices $v_1, v_2 \in V$ [BrEr05]. We refer to incoming edges as *inbound* and outgoing edges as *outbound*. The connection can either be *directed* or *undirected* and hence the graph is either called *directed* or *undirected*. Edges may additionally have a *weight* assigned, we then refer to the graph accordingly as either *weighted* or *unweighted*. We refer to non-structural data like the age of a person attached to vertices and edges as *payload*.

For a given vertex v we define:

- The *degree* $deg(v)$ as the number of edges connected to it.
- We define the *out-degree* $deg_o(v)$ as the number of outbound edges, the *in-degree* $deg_i(v)$ as the number of inbound edges.
- The *distance* $d(u, v)$ between two given vertices u and v is defined as the length of the shortest path between them.
- The *eccentricity* $e(v)$ of a vertex v in a graph $G = (V, E)$ is defined as

$$\max_{u \neq v \in V} d(v, u)$$

For a graph $G = (V, E)$ we further define the following properties [GoOe11]:

- We define the *order* $O(G)$ of G as $|V|$
- The *size* $S(G)$ of G is defined as $|E|$.
- We define the *diameter* $D(G)$ of the graph as

$$\max_{v \in V} e(v)$$

- The *radius* $R(G)$ is defined as

$$\min_{v \in V} e(v)$$

In [AlBa02] the authors identify complex networks and describe their properties: The degree distribution of many large networks follows a power-law distribution, meaning the probability that a vertex in the network is connected to k other vertices is $P(k) \sim k^{-\gamma}$. Those networks are referred to as *Scale-Free Networks*: The World Wide Web and different social and biological networks are mentioned as examples for this class of networks.

In 2007 Tim O'Reilly [ORei07] clarified his previously labeled term *Web 2.0* and envisioned the next generation of software. He emphasized the importance of user generated content as it is present in social networking sites. Wilson et al. [WBS*09] for example identified the popular social network Facebook[1] as being scale-free which highlights the growing importance of scale-free networks in today's software.

[1] `http://facebook.com`

2.2 NoSQL Database Management Systems

When Carlo Strozzi introduced his database which did not have an SQL interface, he used the term *NoSQL* in 1998 [Stro98] to describe it. Later in 2009 Johan Oskarsson organized a conference and named it *NOSQL* – coining it as a term for different systems that also did not use SQL as their query language. Today, a number of new Database Management Systems are classified as NoSQL systems that do not save the data as relations. Most NoSQL systems do not provide ACID compliance like traditional Relational Database Management Systems (RDBMSs) do, but instead focus on a higher read/write throughput or more flexible data models. The growing field of NoSQL database was further classified by Catell [Catt11] into the following subcategories:

- A *Key-Value Store* saves values with a defined index for search. One use-case for these databases is to store session data like the content of a shopping cart. Example: Project Voldemort[2].

- *Document Stores* are "schema-less, except for attributes (which are simply a name, and are not prespecified), collections (which are simply a grouping of documents), and the indexes defined on collections (explicitly defined, except with SimpleDB). [...] The document stores generally do not provide explicit locks, and have weaker concurrency and atomicity properties than traditional ACID-compliant databases" [Catt11]. Examples for systems in this category are CouchDB[3] and MongoDB[4].

- *Extensible Record Stores* define groups of attributes in a schema. The attributes of one group on the other hand are added per record. An example is Cassandra[5] which was used by Facebook to build an inbox search feature [Face10].

- *Graph Databases* store their data directly in the form of a graph. An example for this class of NoSQL databases is InfiniteGraph[6].

- *Object-oriented database systems* and *Distributed object-oriented stores* provide their data directly as objects of the programming language while persisting them in the database. The Objectivity/DB[7] is an example for this category.

Two NoSQL databases will be introduced in this thesis: *Neo4j*, which is classified as a graph database and *ArangoDB*, a combination of the categories key-value store, graph database and document store.

[2] `http://project-voldemort.com`
[3] `http://couchdb.apache.org`
[4] `http://mongodb.org`
[5] `http://cassandra.apache.org`
[6] `http://infinitegraph.com`
[7] `http://objectivity.com`

2.2.1 Comparison of Relational and Native Storage of Graphs

Traditional RDBMSs can be used to model graphs, but they typically do not feature graph support as it is present in graph databases.

Vicknair et al. [VMZ*10] compared *Neo4j* with the RDBMS *MySQL* by filling both with 10,000 randomly generated vertices and edges, with the vertices containing a payload consisting of semi-structured data in form of XML or JSON. Then they compared queries, which they classified as structural and data queries:

- One of the structural queries for example searched for vertices with no inbound and no outbound edges.
- The data queries on the other hand searched for vertices with a certain value in their payload.

The authors compare amongst other things the performance of the queries, the ease of programming and the flexibility of the systems with the following results: The structural queries were significantly faster in Neo4j in most cases – in some cases by a factor of 10. When querying for the payload data there was a huge difference between searching for textual and numerical data, because in Neo4j searching for payload is done with Lucene[8]. While this is very efficient for full-text searches, it is not efficient for other types of data, because they are treated as text: The MySQL database was faster by a factor of at least 40 for queries on the databases. For queries on large strings on the other hand, Neo4j outperformed MySQL by an order of magnitude.

The comparison highlights advantages and shortages of a pure graph database like Neo4j: Both databases perform very well in the domain they were created for. In Neo4j for example, it is significantly easier to formulate graph traversals compared to SQL. Vicknair et al. also point out that Neo4j is easily mutable while changing the schema of a MySQL database is much more work.

[8] `http://lucene.apache.org`

2.3 Information Demand in Large Networks

To explore the properties of ArangoDB in conjunction with large networks, we decided to analyze four categories of information demand: The determination of the *shortest path* and *vertex similarity* were chosen as it has multiple use cases when a social network is stored in the database. The computation of *vertex centrality* was implemented to identify important vertices in the graph when analyzing a network. We also want to compare *search for payload* to the performance of ArangoDB as it was explored [VMZ*10] as a weak spot for numerical data in Neo4j.

2.3.1 Shortest Path

A path in a graph can represent a variety of connections: In a social network it could be the connection between two people, on the web it could be the sequence of links between two pages. On the business-networking page Xing[9] for example the user can display alternative connections under the assumption that they would not be direct contacts as shown in Figure 2.1. A feature like that could be implemented using a shortest path algorithm with vertex exclusion.

Figure 2.1: Alternative Routes in Xing

Because ArangoDB can model directed and undirected, weighted and unweighted graphs, the Dijkstra algorithm [Dijk59] was chosen for this thesis as it is suitable for all these situations.

[9] `http://xing.com`

2.3.2 Centrality of Vertices in a Graph

Freeman [Free78] presented three measurements (closeness, degree and betweenness) to determine the centrality of a given vertex. He points out that there is no "ultimate centrality measure", but that they express different kinds of centrality that are ought to be used for different use cases. Hage et al. [HaHa95] added a fourth centrality based upon the eccentricity of vertices.

Figure 2.2 displays a network with multiple candidates for a central vertex: We introduce the four centrality measurements and display the centrality of each vertex according to the measurement by its size to show the difference between them.

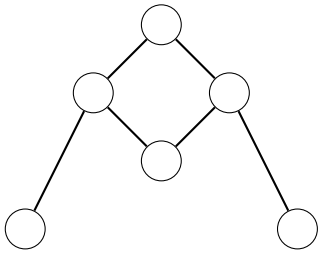


Figure 2.2: Graph with multiple candidates for central vertices

The *degree centrality* as visualized in Figure 2.3 of a vertex v is defined as follows:

$$centrality_d = deg(v)$$

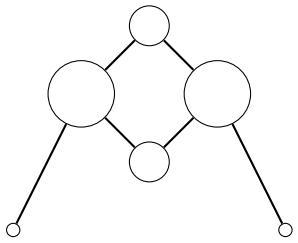


Figure 2.3: Degree Centrality

The *closeness centrality* of a vertex v as visualized in Figure 2.4 is defined as follows:

$$centrality_c = \frac{1}{\sum_{u \in V} d(v,u)}$$

A vertex with a high closeness has a small total distance to all other vertices in the graph.

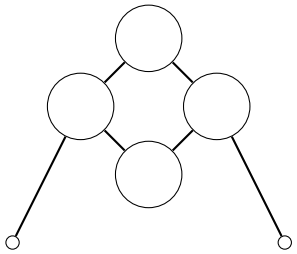


Figure 2.4: Closeness Centrality

The *eccentricity centrality* of a vertex v as visualized in Figure 2.5 is defined as follows:

$$centrality_e = \max_{u \neq v \in V} d(v, u)$$

A vertex with a high eccentricity has a small distance to each of the other vertices.

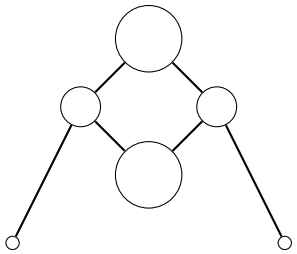


Figure 2.5: Eccentricity Centrality

The *betweenness centrality* of a vertex v is defined as:

$$centrality_b(v) = \sum_{s \neq v \in V} \sum_{t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where $\sigma_{st}(v)$ is the number of shortest paths between s and t that include v and σ_{st} is the number of shortest paths between s and t. This centrality is visualized in Figure 2.6. A vertex with a high betweenness lays on the shortest path for many pairs of vertices in the graph.

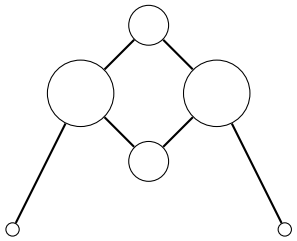


Figure 2.6: Betweenness Centrality

Use case examples for the different centralities are listed in Table 2.1.

Measurement	Use Case	Source
Degree	Potential Communication Activity	[Free78]
Closeness	Independence and Efficiency	[Free78]
Betweenness	Potential to control the communication	[Free78]
Eccentricity	Response time for a site of a facility	[HaHa95]

Table 2.1: Centrality measurements and their usescases

2.3.3 Vertex Similarity

The vertex similarity is a measurement for two given vertices a and b that describes the percentage to which the two are alike: In a social network this can for instance be used to predict if two people will connect in the future (and therefore suggest the connection). This is for example realized by the social network Facebook as described in [BaLe11]. On an online shopping platform like Amazon[10] on the other hand it can be used to recommend similar products to the ones already bought. A feature like that is shown in Figure 2.7.

Figure 2.7: "Customers also bought" on Amazon

As pointed out by Newman [Newm01] two vertices have a high similarity and therefore an increased probability to connect in the future if they have a high number of shared neighbors. We define $\Gamma(x)$ as the cardinality of the set of all neighbors of a vertex x. *Common Neighbors* is the cardinality of the intersection of the vertex neighborhoods defined as follows:

$$CN(x, y) := |\Gamma(x) \cap \Gamma(y)|$$

In a similar way the properties of two vertices can be compared, identified as *Common Properties*. Properties could be the interests of people or the color of two products. We therefore define the properties of a vertex x as $\Psi(x)$. We then define the *Common Properties* of two vertices x and y as follows:

$$CP(x, y) := |\Psi(x) \cap \Psi(y)|$$

[10] `http://amazon.com`

Jaccard's coefficient [DuEv04, p.26] is a variation applicable to both *Common Neighbors* and *Common Properties* that is normalized by the cardinality of the union of the vertex neighborhoods and union of properties accordingly:

$$JN(x,y) := \frac{|\Gamma(x) \cap \Gamma(y)|}{|\Gamma(x) \cup \Gamma(y)|}$$

And respectively:

$$JP(x,y) := \frac{|\Psi(x) \cap \Psi(y)|}{|\Psi(x) \cup \Psi(y)|}$$

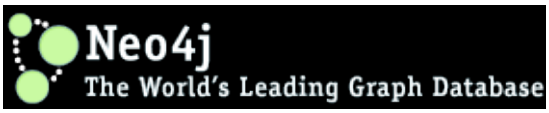
2.3.4 Search for Payload

Not all queries on a large network depend on the structure of the graph. In a social network a user could for instance search for every user that went to his school in a certain year. These queries act on the payload of the vertices meaning data that is not part of the structure, but an attribute of a certain vertex. This includes the search for integers, strings and geo-data. RDBMSs perform very well on these queries opposed to graph databases [VMZ*10].

2.4 Existing Solutions to Analyze Graphs

All algorithms implemented during this thesis are used to analyze data in a graph. We introduce two existing solutions for this problem: The graph database Neo4j and the graphical tool for graph analysis Gephi.

2.4.1 Neo4j



Neo4j[11] is an open-source graph database developed by Neo Technology, Inc.[12]. It is implemented in Java featuring an object-oriented API for the graph, but can also be used in conjunction with other programming languages like Python. Neo4j can be embedded in a Java application or ran standalone as a database server. It features a built-in framework for traversals that offers high expressiveness for corresponding queries.

The Lucene indexing engine is used to search for data independent of the graph structure. This offers high speed for searching textual data, especially in long texts.

We decided to use Neo4j as our reference implementation, because it is available for free so the comparison can be done by anyone without needing to acquire a license. It has the most widespread usage among the free solutions currently available.

Neo4j only has a limited number of algorithms built-in. For example it has no built-in support for determining vertex similarity or centrality. In this thesis we can therefore only compare the calculation of shortest paths and querying of payloads.

2.4.2 Gephi

Gephi[13] is a graphical tool for analyzing graphs written in Java and released as open source. The main focus of the project is to provide "high quality layout algorithms, data filtering, clustering, statistics and annotation" [BHJa09]. Therefore it has a huge set of algorithms built-in. The interface is using workspaces as known from tools like Eclipse to arrange different widgets and store the arrangement. Included among the built-in widgets are a 2D visualization of the graph, a table with all vertices and controls to calculate the above mentioned measurements.

In comparison to a graph database like Neo4j or ArangoDB, it has no functionality like querying from a web application, querying for non-structural data or persisting data apart from simple file storage. It is however possible to import data from and export it for different database systems.

[11] `http://neo4j.org`
[12] `http://neotechnology.com`
[13] `http://gephi.org`

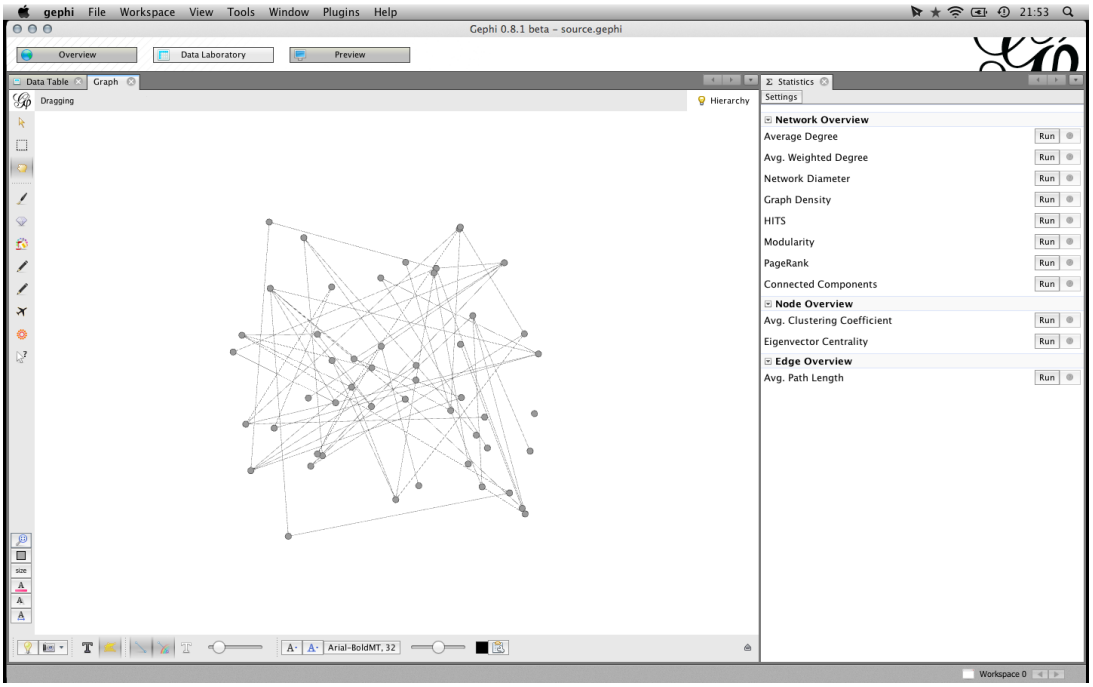


Figure 2.8: Gephi

Gephi is a solid, well documented graph analysis tool that is available for free: It was chosen over a commercial tool, so the comparison can be redone by anyone without the need to acquire a license. Gephi is a good choice for a reference implementation amongst the free solutions, because they released information about the algorithms they used in their Wiki as a reference.

2.5 ArangoDB

ArangoDB is an open-source NoSQL database system developed by triAGENS GmbH[14] that combines a key-value store, a graph database and a document store. The schema is determined automatically and is not provided by the user while allowing the user to model the data as either documents, key-value pairs or graphs. ArangoDB is multi-threaded and memory-based: Only the raw data is frequently synchronized with the file system while supporting data is only stored in memory. Due to Multiversion Concurrency Control (MVCC) documents are not deleted – instead, a new version of the document is stored which allows parallel read and write actions. ArangoDB features the following index types:

Hash Indices are useful to search for equality.

Skip Lists can be used when searching for ranges.

Full-Text Indices will provide fast search in long strings (in development).

Geo-Indices allow searching for documents near a certain location or within a certain area.

There are multiple ways to access the data in ArangoDB:

1. The *REST* [Fiel00] interface provides simple measures to create, access and manipulate the data using the document-identifier or query-by-example.

2. The *Arango Query Language* (AQL) is inspired by SQL, but adds features due to the dynamic nature of the document store reflected in differences compared to SQL: Documents can be constructed by the select statement and nested documents can be queried. The statements are transmitted via HTTP from the application to ArangoDB and the format of the response is JSON.

3. Furthermore *JavaScript* is embedded in ArangoDB using the V8 engine. The documents are accessible as JavaScript objects and the user can define JavaScript functions that can then be executed via the REST interface. The JavaScript API is currently the only way to access all graph information. It will therefore be used to formulate our queries in the form of JavaScript functions.

[14] `http://triagens.de`

2.5.1 Documents and Collections

In ArangoDB the schema-less equivalent to a table is the collection. To demonstrate the API, Listing 2.1 shows an example in which a collection named `theses` is created. Then a document is added to the collection.

```
1 theses = db.theses;
2 //=> [ArangoCollection 2113959, "theses" (status new born)]
3
4 theses.save({ author : "Lucas Dohmen" });
5 //=> { "_id" : "2113959/3228071", "_rev" : 3228071 }
```

Listing 2.1: Creating a Document in ArangoDB

In Listing 2.2 the search for a document is demonstrated.

```
1 query = theses.byExample({ author : "Lucas Dohmen" });
2 my_thesis = query.next();
3 //=> { "_id" : "2113959/3228071", "_rev" : 3228071, "author" : "Lucas
    Dohmen" }
```

Listing 2.2: Searching for a Document in ArangoDB

As mentioned before, ArangoDB is only appending data. Therefore updating a document means replacing it, which is demonstrated in Listing 2.3.

```
1 theses.replace(my_thesis, {
2   author : my_thesis.author,
3   name : "Algorithms for Large Networks in the NoSQL Database ArangoDB"
4 });
5 //=> { "_id" : "2113959/3228071", "_rev" : 4211111, "_oldRev" : 3228071
    }
```

Listing 2.3: Updating a Document in ArangoDB

2.5.2 Graphs

The JavaScript functionality that is not part of the language core is organized in modules in ArangoDB – the module-inclusion is handled as it is proposed by CommonJS[15]. Therefore in order to use one of the provided modules within ArangoDB, the module must be included via `require` and then bound to a variable.

One of the existing modules is called `graph` and it provides three prototypes as shown in Figure 2.9:

- The `Graph` prototype handles adding, removing and getting `Vertices` and `Edges`.
- A `Vertex` provides methods to modify its properties and getting connected edges.
- The `Edge` prototype also provides properties and an additional label and is created by providing the two `Vertices` it should connect.

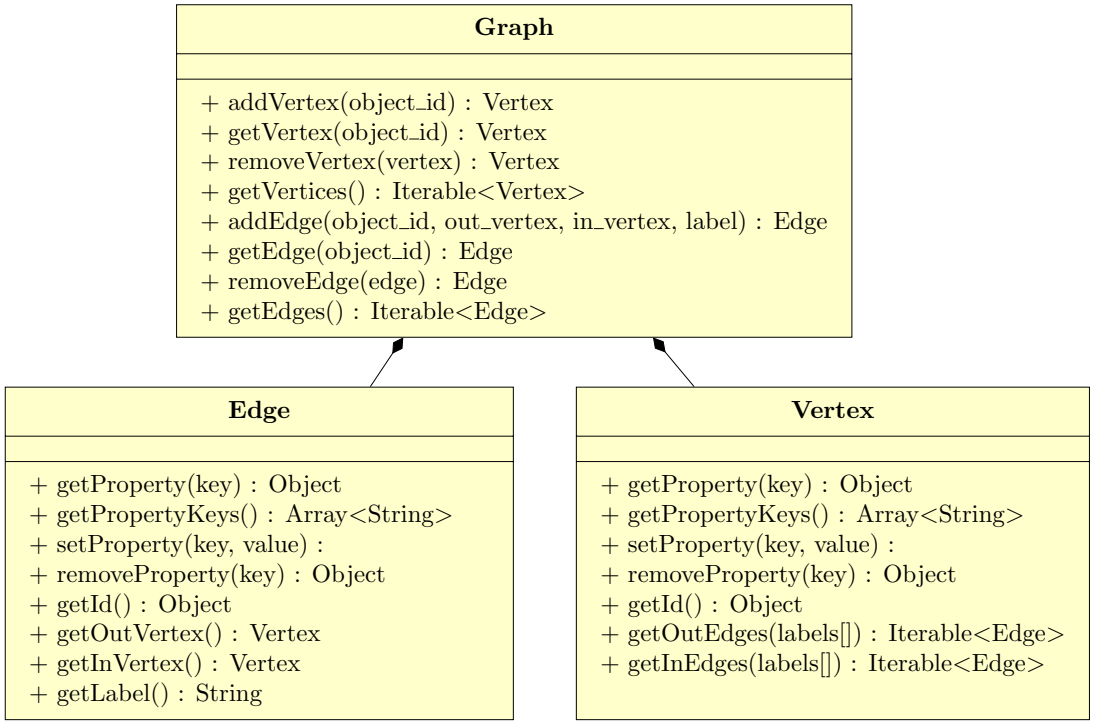


Figure 2.9: UML class diagram for existing functionality of the graph module

[15] `http://commonjs.org`

A `Graph` object is created by providing information about the collections where the vertices and edges should be stored as shown in Listing 2.4.

```
1 // Requiring the graph module:
2 var Graph = require("graph").Graph;
3
4 // Creating a graph using the collections "vertices" and "edges" for
      storage:
5 g = new Graph("my_graph", "vertices", "edges");
```

Listing 2.4: Creating a graph

All vertices and edges are persisted in the database. Both can store arbitrary properties due to the schema-less nature of ArangoDB via the `getProperty` and `setProperty` methods. In Listing 2.5 we create two vertices and an edge between them – in addition we access a property of one vertex:

```
1  // Create the Vertices:
2  person_1 = g.addVertex();
3  person_2 = g.addVertex();
4
5  // Create an edge between the vertices with the label "knows":
6  e = g.addEdge(person_1, person_2, "knows");
7
8  // Accessing attributes of vertices:
9  person_1.getProperty("age"); // => undefined
10 person_1.setProperty("age", 23);
11 person_1.getProperty("age"); // => 23
```

Listing 2.5: Vertices and Edges

3 Concept and Implementation

The entire functionality was implemented using JavaScript to run inside of ArangoDB. Programming in JavaScript has certain known pitfalls. Douglas Crockford therefore defined a strict subset of JavaScript [Croc08] that he regards as safe. He then published a tool called JSLint[1] to check if source code follows his guidelines. The tool was used throughout the implementation in order to avoid those pitfalls and to obey to the standard code style guide he proposed.

The basic graph functionality was available in ArangoDB before this thesis as described in Section 2.5.2. In order to provide a solid foundation for the graph algorithms the present unit tests were enhanced and afterwards the existing functionality was refactored.

We marked the added functionality as bold in the UML diagram (Figure 3.1) for the three prototypes (as JavaScript is not a class-based, but a prototype-based programming language) after the implementation described in this chapter. JavaScript does not support private methods. Therefore we prefixed them with an underscore as it is common practice in the community. There is also no `void` as a return type for functions as it is possible in Java, instead `undefined` was used.

For the test suite we used the tool jsUnity[2], because it does not depend on the browser or Node.js[3] as other testing frameworks do. The code was written following Test-Driven-Development [FrPr09], for each of the algorithms:

- A realistic graph was generated as described in Section 3.1. Reference data was generated with the corresponding reference implementation on the generated graph.
- An integration test was written that compared the reference data to our own implementation.
- For every step along the way to make the integration test pass:
  - An unit test was written to check the specific functionality.
  - Just enough code to make the test pass was written.
- We then refactored the entire functionality.

[1] `http://jslint.com`
[2] `http://jsunity.com`
[3] `http://nodejs.org`

The documentation was written using Doxygen[4] which is used throughout the code of ArangoDB to generate a unified documentation as it is independent from the language the code is written in. The generated documentation can be found on the homepage of the project[5].

ArangoDB is entirely released as an open source project under the Apache License and is available on Github[6]. The implementation of the algorithms described in this chapter – unless otherwise noted – can be found there. Most of the implementation is in the graph module[7] while the tests are in the JavaScript unit test folder[8]. Certain parts of the code that are not a part of the functionality but are used for comparison or as utilities were written in Ruby and Java. These parts are not in the repository mentioned above, but are also open source and available online. The link is provided alongside the description in each case.

[4] `http://stack.nl/~dimitri/doxygen`
[5] `http://arangodb.org/manuals`
[6] `https://github.com/triAGENS/ArangoDB`
[7] js/common/modules/graph.js
[8] js/common/tests

Figure 3.1: UML class diagram for final graph module

3.1 Generating Graphs and Reference Data

In order to provide reference data for the integration and performance tests, realistic networks in different sizes had to be generated.

The Barabási-Albert model [AlBa02] is an algorithm to generate networks. It starts with a small random network and then simulates the natural growth of the network by adding a new vertex u in every step with edges to every existing vertex v at the probability of its preferential attachment which is defined as follows:

$$\Pi(v) = \frac{deg(v)}{\sum_{u \in V} deg(u)}$$

We implemented a command line application named graphshaper[9] to provide this functionality. It is written in Ruby and released as open source. The utility allows to generate graphs of arbitrary size. In addition, the utility also generates tuples of vertices for which the shortest path should be calculated if required for the test. The generated graphs can then be used to compare the results of two different systems: The network is generated as CSV data and can then be imported manually or automatically into the desired system.

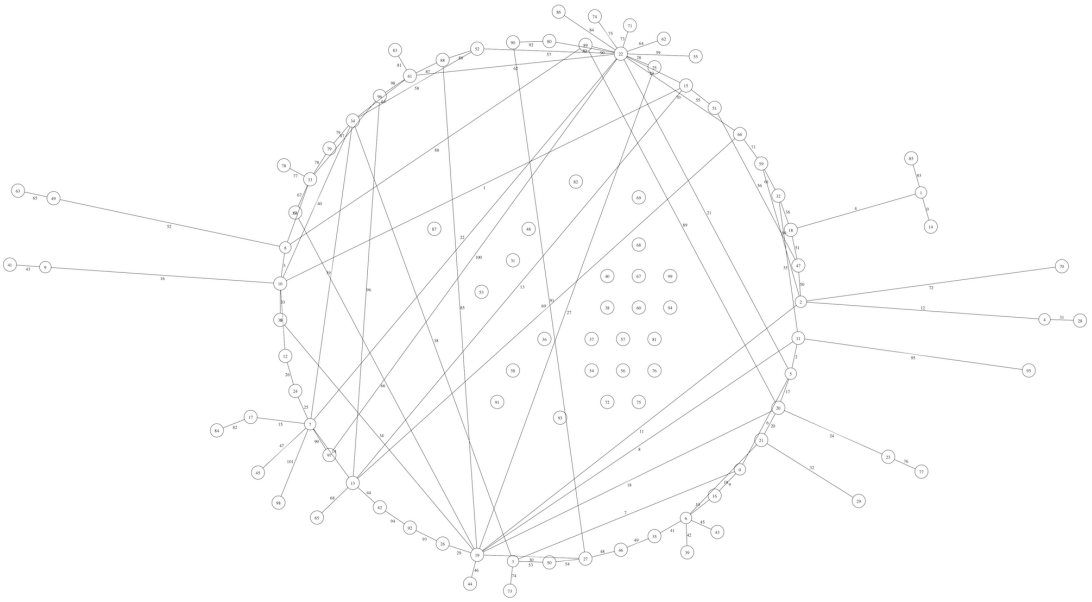


Figure 3.2: Example for a graph generated with graphshaper

[9] `https://github.com/moonglum/graphshaper`

In order to create reference data, the generated graph was then loaded into ArangoDB, Neo4j and Gephi:

- Graphshaper can directly save the graph into ArangoDB via its REST interface.
- The code used to load the data into Neo4j and then calculate the results of the algorithm was written in Java and has been released as open source.[10]
- The data for Gephi was imported, calculated and exported via the GUI.

In addition to comparing the results, we did performance tests which are described in Chapter 4. For those tests we needed to generate payload data. This was implemented as a small Ruby script[11] that generates CSV data containing a name, an age between 0 and 99 and a biography containing a "Lorem Ipsum" placeholder text. An example for data generated by the script can be found in Table 3.1.

ID	Name	Age	Bio
0	"Carlie Ullrich II"	36	"Eum a similique eius non facere. Sint..."
1	"Kristofer Schmidt II"	96	"Et laborum cupiditate quibusdam ea..."
2	"Jeremie Kilback"	93	"Rerum dolorem quae amet iusto ratione..."
3	"Dr. Nasir Kunde"	54	"Ducimus et rerum nisi aut. Magnam..."
4	"Ian Murazik"	96	"Quos pariatur sit veritatis provident ut..."
5	"Nathen Cassin II"	21	"Qui porro adipisci velit praesentium..."

Table 3.1: Example Payload Data (Bio appears shortened)

[10] https://github.com/moonglum/neo4j-graph-algorithms
[11] https://gist.github.com/3479d0fecf19929a0644

3.2 Querying Different Kinds of Graphs

As described before, the graphs stored in the database are of different kinds. The user can choose to treat it as a weighted or unweighted, directed or undirected graph and ignore certain types of edges. This decision is made for every query separately. To allow this versatility while keeping the API consistent without duplicating code, the determination of neighbors used throughout the implementation was separated into a method on the vertex prototype called `getNeighbors`. To determine how the graph should be viewed, the method takes an `Object` as its argument, which is the best practice in JavaScript to allow optional parameters. The method takes the following options:

direction Should only inbound, outbound or all vertices be used? Defaults to all.

weight Should the graph be treated as weighted? If so, the name of the attribute containing the weight has to be given. Defaults to unweighted.

default_weight If a vertex does not have the attribute given to *weight*, this weight is used. Defaults to `Infinity`.

weight_function Like the option *weight*, but takes a function instead of an attribute name. The function takes an edge as its argument and returns a number for the weight. Defaults to unweighted.

labels A vertex is only used, if its label is in the array of strings set here. Defaults to all labels.

only A function that takes an edge as its argument and returns, whether or not the edge should be used. Defaults to all edges.

These options can be combined. This gives the user fine-grained control over the selection of neighbors for a given vertex. For example, `getNeighbors` could be called in the following way:

```
1 vertex.getNeighbors({
2   weight: true,
3   only: function (edge) {
4     return (edge.getProperty("rating") > 3);
5   }
6 });
```

Listing 3.1: Getting Neighbors

The method would then return an Array of neighbors containing only those that have a value of greater than three stored in the property `rating`. In addition, for every neighbor the function would return the weight of the edge leading to it.

3.3 Shortest Path

The shortest path algorithm was implemented as a method on the `Vertex` prototype. It takes two arguments:

target A `Vertex`.

options An `Object`. It takes the options described in Section 3.2 to describe how the graph should be handled and an additional option `cached` to control the caching behavior.

The algorithm searches for all shortest paths leading to the target. Its implementation is based upon the Dijkstra algorithm [Dijk59]:

1. Create an `Object` containing the distance values for all vertices. The start vertex gets a zero assigned. For all other vertices the distance is not set and therefore `undefined` (treated as infinity).
2. Create two lists:
   - The first contains all vertices that have been determined and starts empty.
   - The second one contains all vertices that have to be visited and starts containing only the start vertex.
3. In every step: Take the vertex with the smallest distance from the list of vertices that have to be determined and add it to the determined list. For this vertex:
   a) Iterate over all neighbors provided by the above described method that have not yet been determined and add them to the todo list. If the currently saved distance is higher than the distance of the current vertex plus the distance to it, the distance is updated. Also the predecessors for this vertex is set to the current one. If the distance is equal, the current vertex is added to the predecessors.
   b) If one of the neighbors was the target, the search was successful. If the list of vertices yet to be determined is empty, there is no path between the two vertices. In both cases the algorithm now terminates.

The result of the algorithm can be interpreted as a tree, which can now be traversed to get all shortest paths.

In case the `cached` option was set to true, the results of the algorithm described above are saved in the `Graph` object for the two vertices. Every function that could possibly change the results of the algorithm automatically empties the entire cache.

To demonstrate the usage of the method, Listing 3.2 shows a part of the unit tests for the shortest path functionality. As described, the return value of the function is an Array that contains the shortest paths. Each of the paths is an Array of the IDs for the vertices on the shortest path.

```
1  testGetALongerDistinctPath : function () {
2    var v1, v2, v3, e1, e2, path;
3
4    v1 = graph.addVertex(1);
5    v2 = graph.addVertex(2);
6    v3 = graph.addVertex(3);
7
8    e1 = graph.addEdge(v1, v2);
9    e2 = graph.addEdge(v2, v3);
10
11   path = v1.pathTo(v3)[0];
12
13   assertEqual(path.length, 3);
14   assertEqual(path[0].toString(), v1.getId());
15   assertEqual(path[1].toString(), v2.getId());
16   assertEqual(path[2].toString(), v3.getId());
17 }
```

Listing 3.2: Getting a Distinct Shortest Path

3.4 Vertex Similarity

Four different kinds of vertex similarities were implemented for ArangoDB as introduced in Section 2.3.3. As we are dealing with schema-less data, we have to refine our definition of CP: A property is common between two vertices if and only if both have a property with this key and the value for the property is equal.

The vertex similarity is implemented with two distinct methods `commonNeighborsWith` and `commonPropertiesWith`, both implemented on the `Vertex` prototype. Both methods take optional arguments via the options `Object`: Both provide an optional parameter `normalized` to normalize the result with the union of all neighbors for JN or respectively all properties of both vertices for JP. They also both provide `listed` as an argument to list all shared properties respectively neighbors instead of just the number. This can not be combined with the `normalized` option. `commonNeighborsWith` additionally provides all options presented in Section 3.2 for neighbor selection.

The method `commonNeighborsWith(vertex_id, options)` determines CN or JN for the current vertex v_1 and the given vertex v_2:

- The sets of neighbors V_1 and V_2 for the two vertices are determined by the `getNeighbors()` method with the provided parameters.
- We calculate $V_\cap = V_1 \cap V_2$
- If the option is...

  listed Return $V_\cap$

  normalized Return $\frac{V_\cap}{V_1 \cup V_2}$

  else Return $|V_\cap|$

The determination of CP and JP is done by the method `commonPropertiesWith(vertex_id, options)` for the current vertex v_1 and the given vertex v_2:

- We determine $V_\cup$ as the set of all properties that either v_1 or v_2 has.
- We iterate over the property names in $V_\cup$:
  - If both vertices have the property, and the properties have the same value, add the property name to $V_\cap$
- If the option is...

  listed Return $V_\cap$

  normalized Return $\frac{V_\cap}{V_\cup}$

  else Return $|V_\cap|$

The usage of the method is demonstrated in Listing 3.3 again taken from a corresponding unit test. It shows the results with the `listed` option described above.

```
1  testListedCommonNeighborsWith: function () {
2    var v1 = graph.addVertex(1),
3      v2 = graph.addVertex(2),
4      v3 = graph.addVertex(3),
5      v4 = graph.addVertex(4),
6      v5 = graph.addVertex(5),
7      e1 = graph.addEdge(v1, v3),
8      e2 = graph.addEdge(v1, v4),
9      e3 = graph.addEdge(v2, v4),
10     e4 = graph.addEdge(v2, v5),
11     commonNeighbors;
12
13   commonNeighbors = v1.commonNeighborsWith(v2, {
14     listed: true
15   });
16
17   assertEqual(commonNeighbors.length, 1);
18   assertEqual(commonNeighbors[0], v4.getId());
19 }
```

Listing 3.3: Getting a list of common neighbors

3.5 Vertex Centrality

As described in Section 2.3.2, we implemented four centrality measurements. The degree was simply implemented via the methods `degree`, `inDegree` and `outDegree` on the `Vertex` prototype.

The other three centralities were implemented via the method `measurement` on the `Vertex` prototype. Both take the name of the measurement as the only argument. To implement this method, we were required to determine the geodesics of the graph and iterate over them. We therefore created a helper method that determined the geodesics of the graph named `geodesics` on the `Graph` prototype. As mentioned in the description of the betweenness centrality, it is essential to be able to group the geodesics between two vertices to weigh them differently. Therefore the method `geodesics` takes an optional parameter `grouped` to provide this information.

For the *betweenness centrality*, we iterate over the grouped geodesics G of the given graph calculating $centrality_b$:

- Iterate over the geodesics g in G: If the geodesic contains the vertex, we add them to the set G_i. Then we add the following value to $centrality_b$:

$$\frac{|G_i|}{|G|}$$

For the *eccentricity centrality* for a given node v, we iterate over all nodes u of the graph calculating $centrality_e$ initially set to zero:

- If $d(v,u) > centrality_e$, set the new $centrality_e$ to $d(v,u)$

For the *closeness centrality* for a given node v, we iterate over all nodes u of the graph calculating $centrality_f$ initially set to zero:

- Add $d(v,u)$ to $centrality_f$

The closeness centrality is now $centrality_c = \frac{1}{centrality_f}$.

In Listing 3.4 the usage of the betweenness centrality is demonstrated with the help of the corresponding unit test.

```
testBetweenness: function () {
  var v1 = graph.addVertex(1),
    v2 = graph.addVertex(2),
    v3 = graph.addVertex(3),
    v4 = graph.addVertex(4),
    v5 = graph.addVertex(5);

  graph.addEdge(v1, v2);
  graph.addEdge(v2, v3);
  graph.addEdge(v2, v4);
  graph.addEdge(v3, v4);
  graph.addEdge(v3, v5);
  graph.addEdge(v4, v5);

  assertEqual(v1.measurement('betweenness'), 0);
  assertEqual(v2.measurement('betweenness'), 3);
  assertEqual(v3.measurement('betweenness'), 1);
  assertEqual(v4.measurement('betweenness'), 1);
  assertEqual(v5.measurement('betweenness'), 0);
}
```

Listing 3.4: Getting Betweenness Centrality

4 Evaluation

In addition to comparing the results to Neo4j and Gephi, we also compared the performance of both tools to our implementation. It was tested on a machine running OpenSUSE with five 2.8 Ghz CPUs each with no processes but the essential ones running in the background. In addition, the CPU mode was set to `performance` to let the CPUs run at full-speed at all time. Neo4j was run embedded in a Java application to be comparable with embedded JavaScript as present in ArangoDB.

4.1 Centrality

The performance of ArangoDB and Gephi is currently not comparable when calculating centrality measurements: While Gephi takes seconds to calculate them for a small graph of 500 nodes, ArangoDB takes between 20 and 30 minutes. Therefore the results of Gephi were only used for checking the correctness of the implementation.

In our integration test, we compared the results of our implementation to those calculated by Gephi and we achieve the same results accurate to five decimal places.

It is worth pointing out though that Gephi takes a lot of RAM for the task. When generating the reference data on Mac OS X, a graph with 4000 vertices and random generated edges exceeded 500MB of RAM without any calculations or additional payload data.

This results in an error message prompting the user to increase the memory limit for the application occurring when starting with an empty project and generating 4000 random vertices.

When a network with 10,000 vertices is stored in ArangoDB, it only takes about 300 MB of RAM. So even though ArangoDB is currently not comparable from a performance standpoint at the current point of time, this highlights an opportunity for future development.

4.2 Shortest Path

4.2.1 Performance

To test the performance of the shortest path implementation we compared it to Neo4j:

- Only the time for calculating the paths for the generated test cases (consisting of a start and end vertex) was tracked, not the time needed for importing the data.
- Each test was run 10 times – for each run first a graph was generated and then imported in both databases. For each test we determined the average of the 10 test runs.
- We made four tests: For 500, 1000, 1500 and 2000 nodes.

The results are in Figure 4.1.

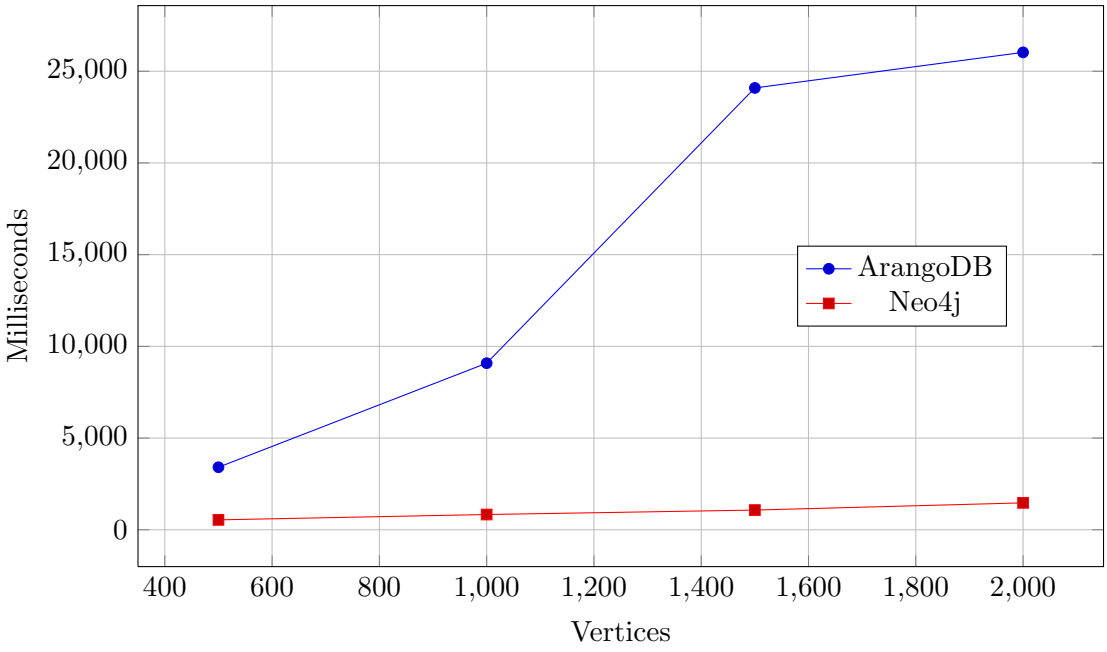


Figure 4.1: Comparison without caching

The results clearly point out that our implementation is much slower than the implementation in Neo4j: While the time consumption for the queries stays almost constant in Neo4j, ArangoDB needs much more time to answer the query. This difference increases even more when the graph has more than 1500 vertices.

We then added caching and ran the test 5 times for each of the graphs and ran it on 10 different graphs with 500 vertices each using the average. The results of the comparison are in Figure 4.2: *ArangoDB1* does not use caching, *ArangoDB2* does.

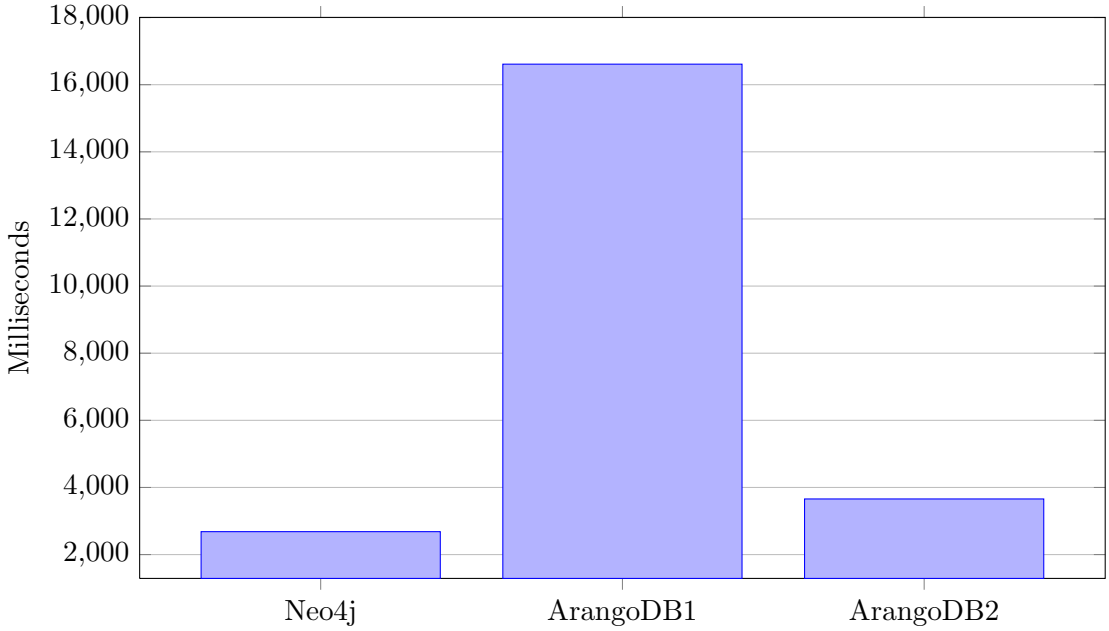


Figure 4.2: Comparison with and without caching

This optimization only helps in cases where the same paths are asked for multiple times: The current implementation of caching is only saving start and end vertices for every query, no subpaths. But the situation occurs for example when querying for centrality measurements.

Our performance test shows that our implementation should be improved further in the future to keep up with the time that Neo4j takes for the same task.

4.2.2 Resource Consumption

To test the resource consumption we ran the path algorithms on a graph with 5,000 vertices in an infinite loop.

We measured the CPU usage in percent (where 100% means that one CPU is working to full capacity) while querying for shortest paths on the graph. From our results shown in Figure 4.3 we can see that Neo4j is able to work multithreaded and therefore uses multiple cores. Even though ArangoDB is also able to do that, the JavaScript execution is always run in a single thread.

Both databases are memory based which resulted in no disk usage during our tests as no data got changed and therefore no synchronization was needed.

A fair comparison of the RAM usage is currently not possible. As our implementation runs in the virtual machine of the V8 and Neo4j runs in the JVM, the RAM usage on the system has no significance. Even though there are tools available for inspecting the internal memory usage of the JVM, there is currently no support for these statistics in ArangoDB. Therefore this comparison is not possible at this point of time.

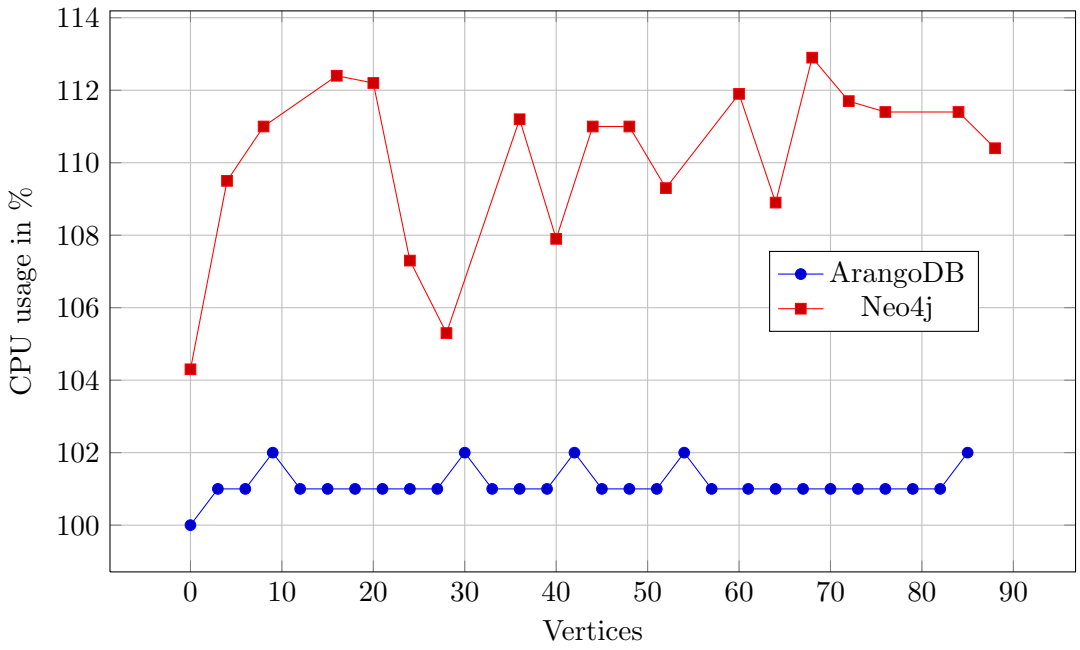


Figure 4.3: CPU usage over time while searching for shortest path

4.3 Payload

4.3.1 Performance

We defined three queries to compare the performance of Neo4j and ArangoDB when searching for payload. They were run on vertex sets of different size.

Our first query asked for an integer value that is greater than 20 and lesser than 30. As the results in Figure 4.4 show, ArangoDB is between 12 and 29 times faster as Neo4j at this task.

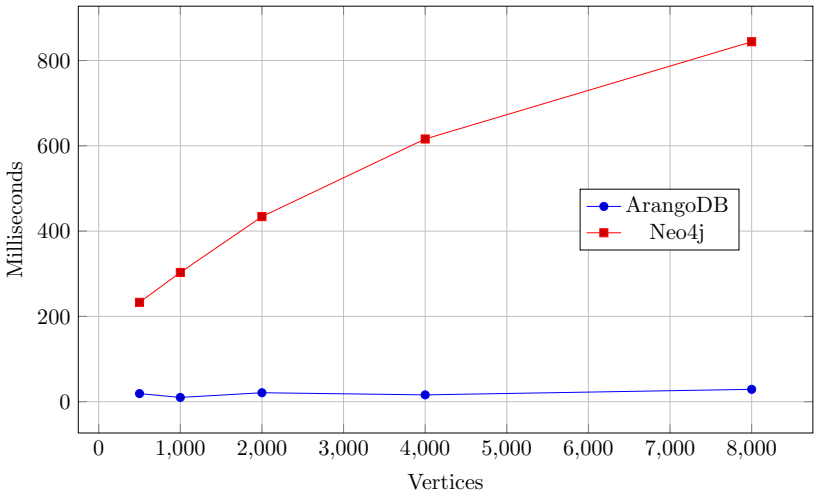


Figure 4.4: Comparison of the performance when searching for integers

Our second query asked for an exact string match. As the result in Figure 4.5 show, the task is executed in under five seconds in both Neo4j and ArangoDB.

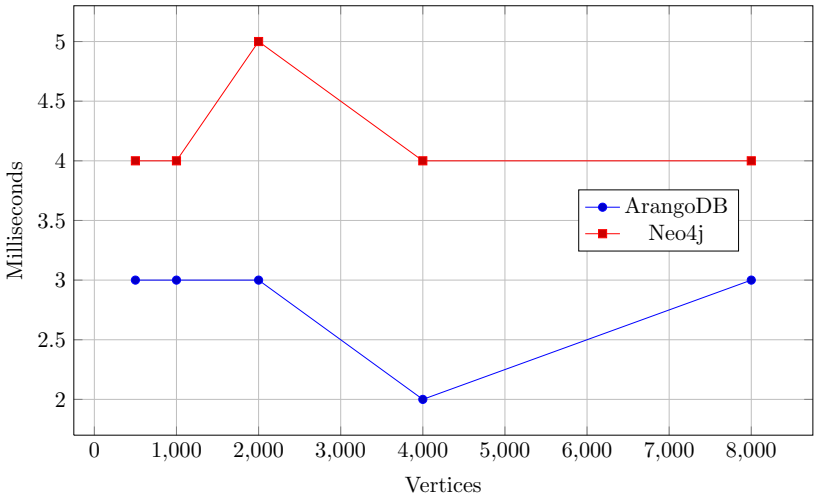


Figure 4.5: Comparison of the performance when searching for strings

Finally, we queried for strings beginning with a certain word. The stored data consisted of two paragraphs of text. The full-text index is not ready yet in ArangoDB. Therefore we used the skip list index for this task. Even though it can be used in this case, it is currently not possible to query for a substring that is not at the beginning of the string in ArangoDB. The results of this comparison are shown in Figure 4.6. Neo4j is faster at this task for graphs of a size of up to 1,000 vertices. For graphs with more vertices however, ArangoDB is faster.

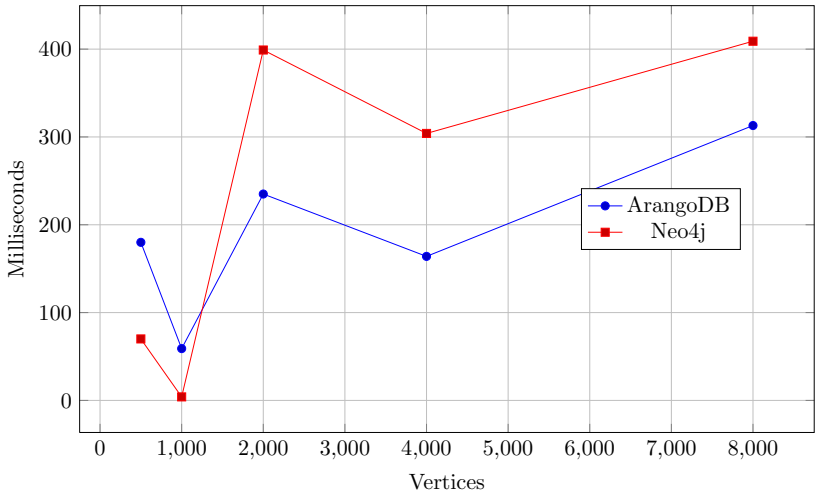


Figure 4.6: Comparison of the performance when searching in texts

4.3.2 Resource Consumption

To compare the resource consumption while querying for payload data, we searched for an exact string match in both databases filled with 8,000 vertices in an infinite loop. Both databases did not read from or write to the disk during runtime as no data has been changed. As mentioned in Section 4.2.2, the memory consumption could not be compared.

We measured the CPU usage in percent where 100% means that one CPU is working to full capacity. As mentioned before, the JavaScript execution is run on a single thread in ArangoDB. Neo4j however is run multithreaded. As the results in Figure 4.7 show, Neo4j almost saturates two CPUs while taking longer to execute this task.

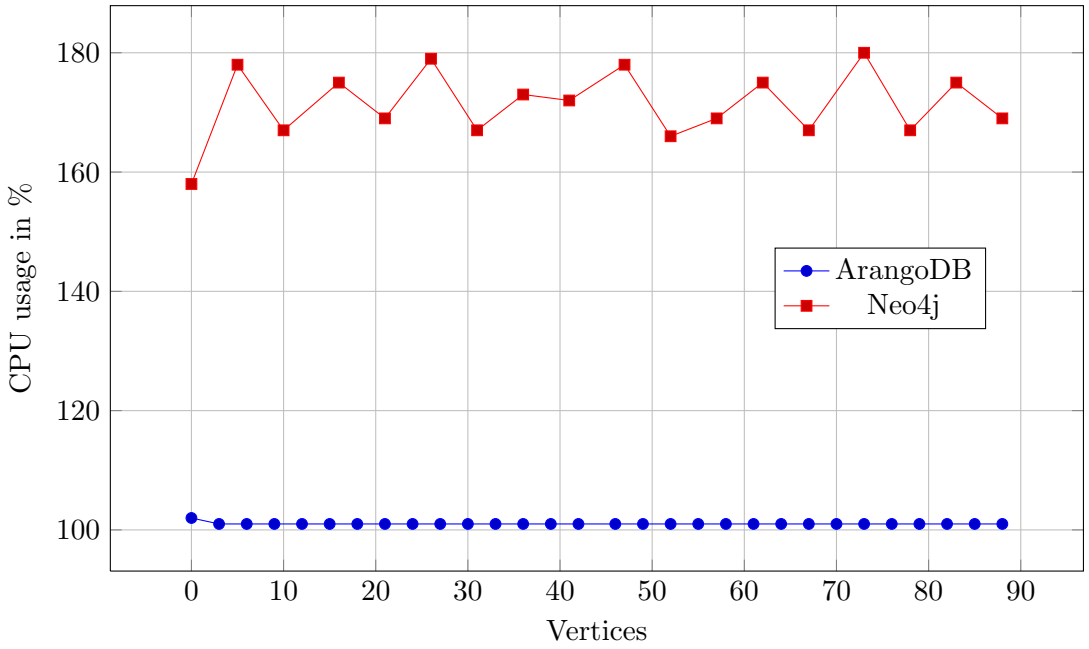


Figure 4.7: CPU usage over time while searching for payload

We then compared the storage needed on disk for saving the graphs. Figure 4.8 also shows the size of the generated CSV file containing the same data. In comparison with Neo4j, ArangoDB needs more disk space. There are two main reasons for that:

1. ArangoDB uses journaling to keep track of changes in the database which is part of the database size.
2. ArangoDB reserves more space than needed for the data and increases the size of the database in a predetermined step size (which is reached at 8000 vertices in our example).

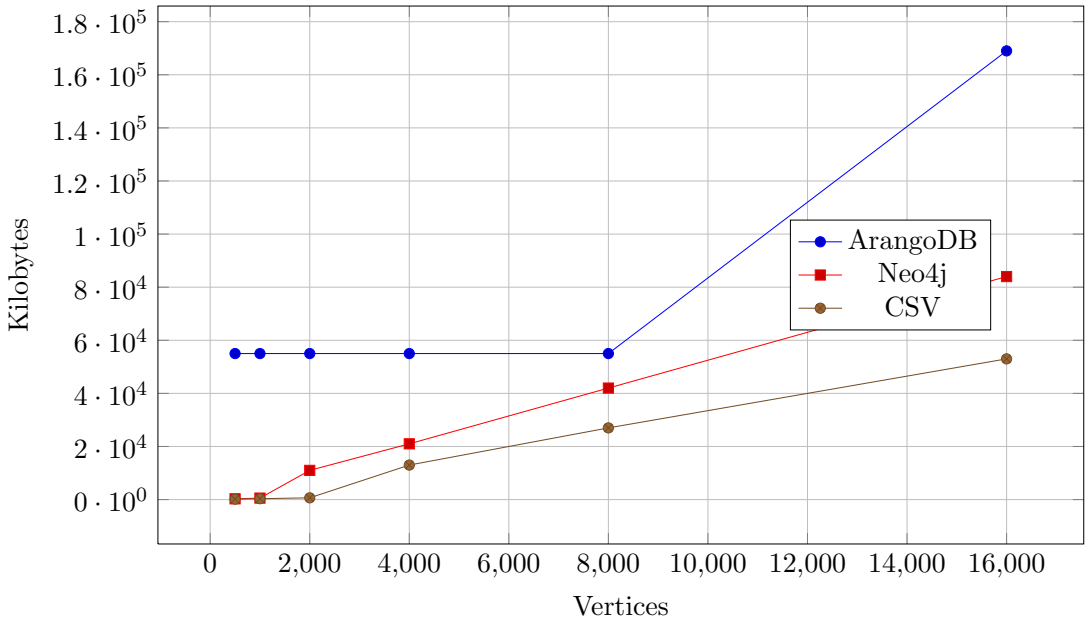


Figure 4.8: Diskspace usage

5 Conclusion and Outlook

During this thesis we implemented one shortest path algorithm, four similarity and four centrality measurements in ArangoDB:

- We implemented the *Dijkstra algorithm* to find the shortest path in weighted and unweighted, directed and undirected graphs as a method on the `Vertex` prototype.
- The search for *common neighbors* and *common attributes* of vertices was implemented as a method on the `Vertex` prototype with the option to normalize the results.
- We implemented *degree centrality*, *betweenness centrality*, *closeness centrality* and *eccentricity centrality* as methods on the `Vertex` prototype

Our implementations return the same results as their counterparts in the reference implementations Neo4j and Gephi. This is a proof of the functionality and correctness of the implementation of the desired functionality. We also compared the performance of our implementation and the search for payload with the reference implementations.

Our evaluation has shown that we should address the performance issues: Even though the resource consumption is comparable to Neo4j, the speed of execution leaves room for improvement. Our extensive test suite allow us to optimize this area in the future without risking incorrect results. The performance is therefore our foremost goal for future improvements to the graph functionality.

Our first steps to improve the performance decreased the runtime for the shortest path algorithm in certain cases. One possible improvement would be to extend the caching functionality to also support partial caching: Currently the cache is used if and only if the exact same path was asked for before. This could be improved to also save all shortest paths that are calculated along the way. Furthermore when determining a shortest path the calculation could be accelerated by also using the cache to terminate the algorithm early if a path to a vertex is found for which the path to the destination vertex is already known.

Currently the cache is emptied very pessimistically by deleting it entirely when something in the graph has changed. This behavior could be adjusted to only empty those parts of the cache that are affected.

We will also investigate the impact of updating an adjacency matrix while adding, changing and removing vertices. The resulting data could be used to accelerate the calculation further.

Besides we will investigate the integration of the shortest path algorithm in the execution of AQL queries. Actually it is possible to execute the JavaScript functionality implemented in this thesis via the REST interface by writing the functionality in JavaScript and executing it via the built-in application server. But the integration in AQL will simplify the usage from an external application server.

Other information about the stored graphs like the centrality of all vertices or the adjacency matrix should be available via the REST interface to allow external tools to leverage the graph capabilities of the database. This could be used by web applications for monitoring important users in a community or by visualization tools to display the stored graph.

ArangoDB just reached Version 1: Additional features that are not directly linked to the graph functionality like the previously mentioned geo- or fulltext indices and other functionality like replication are in development. The introduction of MRuby as a second embedded language for ArangoDB will introduce the challenge of library code like the graph functionality implemented in two different languages.

The efficient storage of structural and payload data allows the database to store large networks on small disk space. Our tests on querying payload also have shown that non-structural data can be accessed even on big datasets. Our algorithms on the other hand will be improved in the future to answer queries on those networks in a moderate time.

Even though the performance should be improved in the future, ArangoDB is now usable as a basic graph database as our results show. This complements the document store and key-value store capabilities of the database. A social networking application for example could benefit from this combination. Our implementation and the comparison with existing solutions have shown the potential of ArangoDB.

Bibliography

[AlBa02] R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74:47–97, 2002.

[BrEr05] U. Brandes and T. Erlebach. *Network analysis: methodological foundations*. Springer Berlin Heidelberg, 2005.

[BHJa09] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks, 2009.

[BaLe11] Lars Backstrom and Jure Leskovec. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the fourth ACM international conference on Web search and data mining*, WSDM '11, pages 635–644, New York, NY, USA, 2011. ACM.

[Catt11] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4), 2011.

[Croc08] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, 2008.

[DuEv04] G. Dunn and B. Everitt. *An Introduction to Mathematical Taxonomy*. Dover Books on Mathematics. Dover Publications, 2004.

[Dijk59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 10.1007/BF01386390.

[Fiel00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[FrPr09] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 2009.

[Free78] Linton C. Freeman. Centrality in social networks - conceptual clarifications. *Social Networks*, 1978/79.

[GoOe11] Wayne Goddard and Ortrud R. Oellermann. Distance in graphs. In Matthias Dehmer, editor, *Structural Analysis of Complex Networks*, pages 49–72. Birkhäuser Boston, 2011.

[HaHa95] Per Hage and Frank Harary. Eccentricity and centrality in networks. *Social Networks*, 17(1):57 – 63, 1995.

[HaRe83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[Face10] Kannan Muthukkaruppan. The underlying technology of messages.

[Newm01] M. E. J. Newman. Clustering and preferential attachment in growing networks. *Phys. Rev. E*, 64:025102, Jul 2001.

[ORei07] Tim O'Reilly. What is web 2.0: Design patterns and business models for the next generation of software. *Communications & Strategies*, 65, 2007.

[Stro98] Carlo Strozzi. Nosql - a relational database management system.

[VMZ*10] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, pages 42:1–42:6, New York, NY, USA, 2010. ACM.

[WBS*09] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P.N. Puttaswamy, and Ben Y. Zhao. User interactions in social networks and their implications. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 205–218, New York, NY, USA, 2009. ACM.

List of Figures

List of Tables