Actor-based MapReduce-application for calculating Key Performance Indicators

Bachelor thesis

Max Messerich

In fulfillment of the requirements for the degree
Bachelor of Science in Informatics
To be awarded by the
Fontys Hogeschool Techniek en Logistiek

June 12, 2018

Information Page

Fontys Hogeschool Techniek en Logistiek
Postbus 141, 5900 AC Venlo

Bachelor thesis

Name of student:	Max Messerich
Student number:	2672502
Course:	Software Engineering
Period:	February - June 2018
Company name:	traperto GmbH
Address:	Gertrud-Boss-Straße 7
Postcode, City:	47533, Kleve
Country:	Germany
Company coach:	Thorsten Rintelen
Email:	t.rintelen@traperto.com
University coach:	Ferd van Odenhoven
Email:	f.vanodenhoven@fontys.nl
Examinator:	Stefan Sobek
Non-disclosure agreement:	Does not apply.

Statement of Authenticity

I, the undersigned, hereby certify that I have compiled and written the attached document / piece of work and the underlying work without assistance from anyone except the specifically assigned academic supervisors and examiners. This work is solely my own, and I am solely responsible for the content, organization, and making of this document / piece of work.

I hereby acknowledge that I have read the instructions for preparation and submission of documents / pieces of work provided by my course / my academic institution, and I understand that this document / piece of work will not be accepted for evaluation or for the award of academic credits if it is determined that it has not been prepared in compliance with those instructions and this statement of authenticity.

I further certify that I did not commit plagiarism, did neither take over nor paraphrase (digital or printed, translated or original) material (e.g. ideas, data, pieces of text, figures, diagrams, tables, recordings, videos, code, ...) produced by others without correct and complete citation and correct and complete reference of the source(s). I understand that this document / piece of work and the underlying work will not be accepted for evaluation or for the award of academic credits if it is determined that it embodies plagiarism.

Name:	Max Messerich
Student Number:	2672502
Place/Date:	Kleve, June 12, 2018

Signature:

Contents

List of Figures

List of Tables

List of Code Snippets

Acronyms

AWS Amazon Web Services

BFS Breadth-first search

CPU Central Processing Unit

CSV Comma Separated Values

DAG Directed Acyclic Graph

DFS Depth-first search

DSL Domain Specific Language

GUI Graphical User Interface

JSON JavaScript Object Notation

JVM Java Virtual Machine

KPI Key Performance Indicator

SQL Structured Query Language

Definitions

Key Performance Indicator : A KPIs is a number for indicating performance of some aspect of a business.

Amazon Web Services : A cloud services provided by Amazon.

Function as a Service A cloud service technology for executing programs without a server.

Abstract

This report describes a generalized *MapReduce*-application for calculating KPIs based on graph representations of business structures. The graphs used by this application are created by a secondary application which extracts all relevant data for the calculating KPIs from multiple undefined data sources. For this purpose, the graph creation application only requires meta information defining the structure of the business in question. By making use of the Actor model, the generalized *MapReduce*-application is capable of parallelizing a large portion of the required steps in calculation KPIs, which results in a great performance increase compared to sequential processing of the data.

Chapter 1

Introduction

This chapter briefly introduces the company traperto and the software project which will be executed in conjunction with this thesis.

1.1 Company description

traperto is a software company based in Kleve which develops web applications for large companies like Vodafone, unitymedia, OBI and Deichmann. The companies main product is called trapero campus which customers use to manage and track the training of their employees. The application allows companies to manage online courses, seminars and other forms of training. Employees can register their attendance through the software and track their progress trough a dashboard. In addition to the Learning Management Software, traperto also offers *KPI Cockpit* which provides customers Key Performance Indicators (KPIs) for all levels of their organizational hierarchy. This tool gives customers an overview over the performance of their business and allows them to detect problematic developments in their organization. In total, nine people work at traperto of which six are software developers. The CEO of the company and the head of marketing and sales are responsible for consulting customers and project management.

1.2 What is a Key Performance Indicator

A KPI is a number which indicates the performance of some business aspect. An example for a KPI is *Number of units sold*, which describes the how often a products were sold. By comparing this KPI for different products, a business can make informed decisions about is product portfolio.

1.3 Problem Statement

The project is about improving the back-end of the KPI Cockpit which faces two issues traperto wants to solve: Firstly, large parts of the software need to be re-implemented in order to be usable for a new customer, because the organizational structures, the provided data and the KPIs to be calculated are always unique. Secondly, the application performs poorly because large quantities of data need to be handled and KPIs need to be calculated for all nodes within complex organizational structures.

Chapter 2

Background

This chapter explains the background of the project. First, a problem definition is given which points out the high level requirements of the KPI Cockpit and the issues of the current implementations of the software. Due to the abstract nature of the application, the domain model is defined in mathematical terms. In order to make the structures to be represented more comprehensible, an example of a business structure to be modeled is given. Additionally, this chapter also describes how the structure of the domain model can be used to reach the desired performance improvements of the KPI Cockpit.

2.1 Problem description

The KPI tool is an application which aims to provide users easy access to information about how their business is performing with data they provide on at least a daily basis.

2.1.1 KPI for all levels of a business

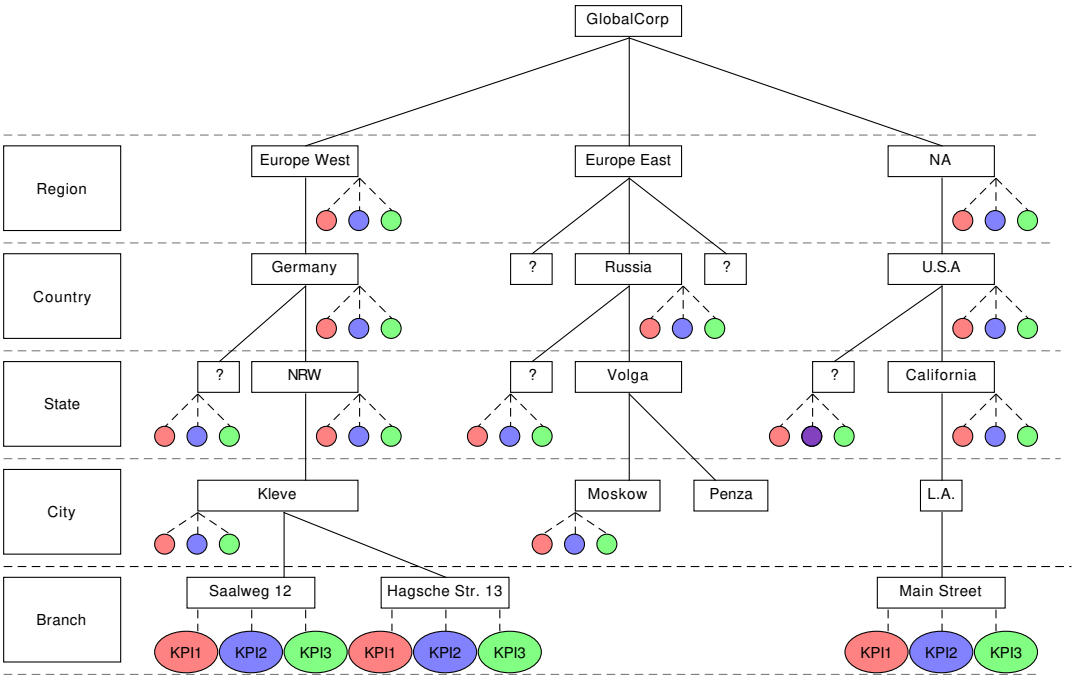


Figure 2.1: Key Performance Indicators (KPIs) for all levels of the organizational structure

The main feature of the KPI-Cockpit is to provide quick access to highly generalized but also very specific business KPIs. Figure 2.1 shows an example of a customer who uses the application to show three KPIs (indicated by red, blue and green). KPIs of a certain node depend on the KPIs of its children: Kleve's score for KPI1 depends on the Branch nodes Saalweg 12 and Hagsche Str.13. NRW's KPIs depend on its City nodes, Europe West depends on its Country nodes and global KPIs depend on all Region nodes of the organizational structure.

2.1.2 Support for any business structure

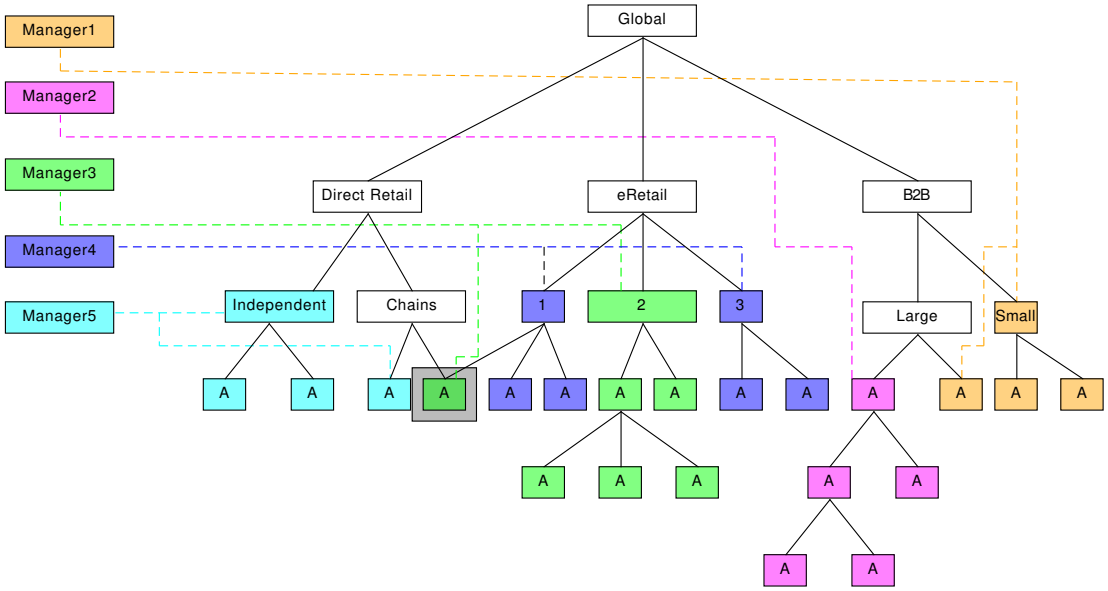


Figure 2.2: Complex organizational structure to be represented.

Currently, traperto needs to re-develop large portion of the software in order to provide it to a new customer. In order to reduce the amount of effort required for adapting the KPI Cockpit for a new customer, the application must be capable of representing any type of organizational structure. These structures usually consist of multiple interconnected business dimensions like a hierarchy of managers, regions of the world the business is active in and sales channels. Figure 2.2 show an example of such an organizational structure which consist of a sales channels and a managers responsible for nodes within the sales channels. The KPI Cockpit needs to calculated KPIs for each vertex in this graph. In most cases, the represented structure form trees. However, there a relationships within a business in which an element has two ore more parents parents. An example of such a relationship can be seen in Figure 2.2: The vertex marked with the gray outline has two parents, because the company represented by this vertex is a chain active in direct retail and also has a web-shop. Another example of such a relationship might be a worker who is supervised by two or more managers. Relationships like this are unusual but need to be supported in order to assure the flexibility of the application. For this reason, the business layers to be modeled are graphs and not trees. A more detailed example of a business structure modeled in the application can be found in section 2.2. Chapter 3 defines the requirements of the data layer

2.1.3 Performance issues

traperto's customers are already using older version of the KPI Cockpit which suffer from performance issues. Loading a page of the web front-end of the application can take up to 20 seconds. In order to improve the usability of the KPI Cockpit, loading times must be reduced.

2.1.4 Pre-Calculated Key Performance Indicators (KPIs)

In order to prevent the performance issues from affecting the user experience, traperto has chosen to pre-calculate all KPIs visible in some versions of the application. While this reduces loading times to a minimum, it also limits the flexibility of the KPI Cockpit: KPIs can not be calculated for a set of entities defined by a user defined filter. This limits the quality of the information provided trough the application, because users can not specify to for which parts of the business they want to view KPIs(e.g Calculating KPIs for all shops which employ less than four workers). Due to performance issues, the current versions of the KPI Cockpit can not offer this functionality. If the new system does not face the same issues and achieves acceptable loading times for calculation KPIs on demand, filtering elements of the business for specific KPIs becomes an option.

2.1.5 Simple maintenance and local use for development

The application needs to be easy to set up and maintain, because of the limited time available for infrastructure maintenance: The employee responsible for this task is also the lead back-end developer and also involved in project management. In order

2.1.6 Structure of the provided data

id	KPI	Static data*	Date n-1	Date n
0	A	100	4	5
0	B	50	1	3
0	C	20	2	6
1	A	60	6	9
1	B	20	6	9
1	C	30	6	9
2	A	60	2	7
2	B	40	4	3
2	C	20	1	2

Table 2.1: Example of a business report

At the core of the data model lie the business reports which are provided on a daily basis and contain the values which will be used to calculate KPIs. Table 2.1 gives an example of a business report containing data required by the KPI-cockpit.

- **id** The identification of the business entity the values are for.
- **KPI** The identification of the KPI to be calculated.
- **Static data** Static data providing additional information for KPIs. (e.g expected sales for calculating the KPI *Number of Units solds*).
- **Dates** Values required for the calculation of the KPI.

2.2 Representing the structure of a business

This section describes an example of a business structure to be modeled in the KPI Cockpit. Due to the abstract nature of the KPI Cockpit, this model is not the domain model of the application but rather an example of a business structure which needs to be represented. Additionally, the calculation of KPIs is explained within the discussed model. The actual domain model is a multilayer directed acyclic graph which defines the properties of the model in a mathematical way. This graph and its properties are described in chapter 3.

2.2.1 Modeling a chain for KPI calculation

Ths subsection describes how a chain, which consist of multiple branch stores, is modeled in the KPI Cockpit and how KPIs are calculated for it.

Representing the chain

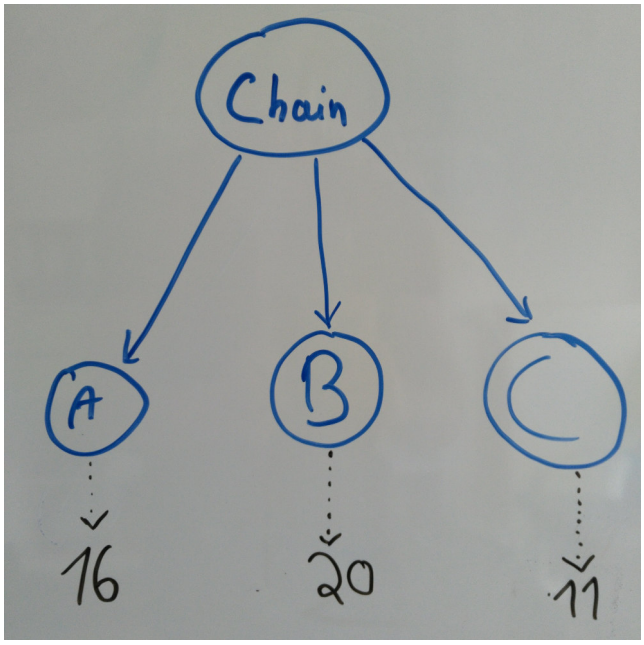


Figure 2.3: Calculating KPIs from children.

The diagram Figure 2.3 shows how a chain part of the sales channel hierarchy of a corporation to be represented in the KPI Cockpit. The chain consists of multiple branch stores which are represented by the vertices at the bottom of Figure 2.3. The arrows pointing from the chain vertex to the branch vertices represent the relationship between the chain and its branch stores: *Stores A, B, C are part of the chain.* The *part of* relationship between any two vertices is the essential property of the data model.

Calculating KPIs for the chain

The relationships between entities defines how their KPIs are calculated. In order to calculate the KPI *Number of units sold* for the branch, the application needs to know the values for that KPI for the branch stores $\{A, B, C\}$ which are 16, 20 and 11. In this example, these values of the chain are provided in the business report provided to the KPI Cockpit. Once these values are available, X can be calculated for the chain which is the sum of X of $\{A, B, C\}$ which is 47. In general, the KPI X of any vertex within the data layer can only be calculated if X is known for all its child vertices. In other words, the *part of* relationship describing the relationship from a parent vertex to a child vertex is inversed for the flow of information.

2.2.2 Modeling the entire sales channel hierarchy for KPI calculation

Obviously, large corporation have larger and more complex sales channels which need to be modeled in the KPI Cockpit. However, if the data model is structured in such a way that for each element the elements which it is *part of* can be deduced, the data model works the same as the example of the single chain. The properties of the required structure are defined in section 3.2.

Modeling the sales channel

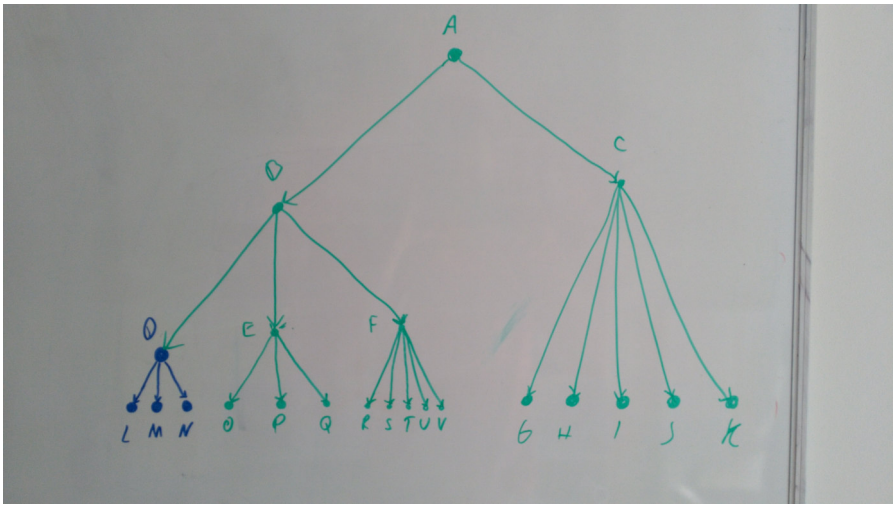


Figure 2.4: Calculating KPIs from children.

Figure 2.4 visualizes the representation of the sales channel hierarchy of a business structure. This hierarchy splits up the sales channel in directed retail (left halve of the diagram) and e-retail trough third party web-shops (right halve of the diagram). The direct retail channel consists of three chains, of which one is the chain modeled in the previous subsubsection (blue sub-graph). e-retail consists of five web-shops which sell products of the business. The source vertex on top represents the entire sales channel.

Calculating KPIs for the sales channel

Since the sales channel follows the logic of the chain (see 2.2.1), KPIs can be calculated in the same way for each vertex in the sales channel: In order to calculate the KPI *Number of units sold* for the root vertex(A), the KPI needs to known for the vertex representing the direct retail channel(B) and the e-retail vertex(C). In order to calculate the KPI for direct retail, the KPI of the chains need to be known (D,E,F). In order to calculate the KPI of the e-retail vertex, the KPI of the web-shops (G,H,I,J,K) needs to be know and so on. When the values for all children and children's children are available, the KPI can be calculated for the entire sales channel. As in the chain example, the values for the lowest vertices within Figure 2.2.1 are provided in the business reports.

2.2.3 Adding managers to the data layer

Since trapertos' customers are large corporations, a single layer representing a sales channel will not be sufficient for modeling an entire business structure. For example, the company might be interested in KPIs for individual manager responsible for certain parts of the sales channel. This subsection describes how the the layer of managers can be connected to the sales channel layer. With the connection to the sales channels, KPIs can be calculated for managers. In most cases, the graph modeling the customers business structure will consist of multiple layers in order to allow users to show how different aspect of their business affect their performance.

Modeling multiple business layers

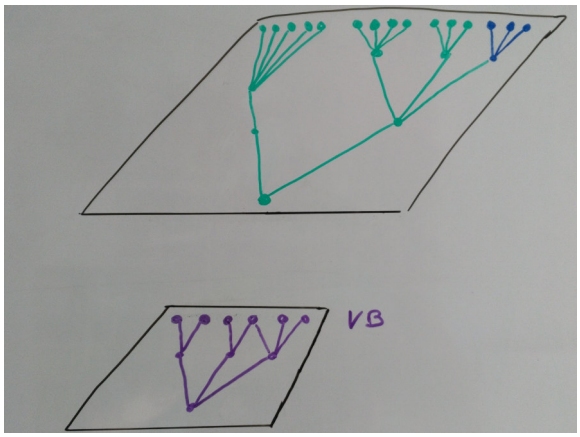


Figure 2.5: Multilayer graph representing a domain model with two layers.

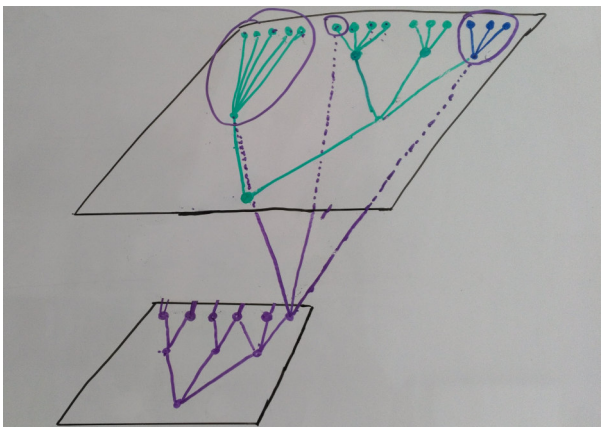


Figure 2.6: Multilayer graph representing a domain model with interconnected layers.

Figure 2.5 shows two hierarchies displayed in two layers or dimensions. Two layer on top represents the sales channels discussed in the previous subsubsection, the layer below represents a hierarchy of managers. The hierarchy of manager shows that the structure to be represented are not trees but graphs in which any vertex may have more than one parent. In this case, a vertex connected to two parents means that the represented manager has two supervisors. Figure 2.6 shows the same two business dimensions which have been connected by an interlayer graph connection vertices of the manager dimension to the vertices of the sales channel (The exact properties of the interlayer connections are defined in section 3.4.2). In the context of this domain model, a connection between a manager vertex and a vertex of the sales channel means: The manager is responsible for the sub-graph of the sales hierarchy below the vertex he/she connected to. For example, the manger connected to the chain (blue in layer on top) is responsible for the chain and its branch stores.

2.2.4 Calculating KPIs through multiple business layers

By creating connections between a the managers and the sales channel hierarchy, paths towards vertices connected to data from a business report are established. Therefor, KPIs can be calculated for managers. In order to calculate KPI X for the manager vertex connected to the sales channel dimension, the values of the connected sales channel vertices need to be known. For example, the manager in question is responsible for the previously discussed chain (blue tree in Figure 2.6). X needs to be know for this vertex in the sales channel (and the two other subgraphs of the sales channel the manager is connected to) before X can be calculated for the manager.

2.3 Solving the performance issues

The current implementations of the KPI Cockpit suffer from performance issues, because no all processes executed on the data layer are run sequentially. The reason for this is use of trapertos' default technologies *PHP* and *MySQL* in the application, which have poor support for concurrency. This section describes some core ideas of the application which will address this problem.

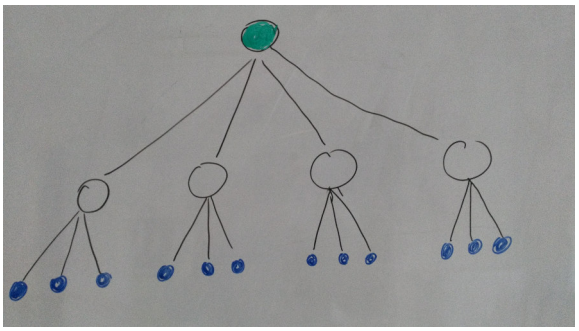


Figure 2.7: Diagram showing a data structure to be processed.

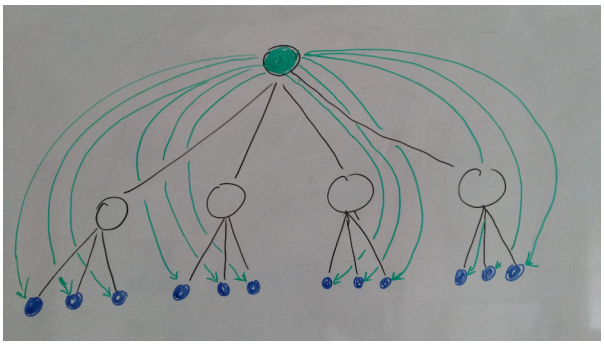


Figure 2.8: Diagram showing the relational structures of the current version of the application.

2.3.1 Parallelizing calculations

Figure 2.7 shows a typical structure for which the application needs to calculate KPIs. In this example, the application needs to calculate a KPI for the vertex on top. The current version reduces the complexity of the queries to find all values required for this calculation by linking the vertex on top to all vertexes of interest for the calculation (see Figure 2.8). This reduces the amount processing required, because the data which needs to be processed does not need to be found. However, this also removes the ability to calculate more specific KPIs, because the data used for calculating is static. Additionally, the process of extracting the data is still a sequential process, because the the database does not parallelize the queries involved in executing the calculation. Since multiple thousands sets of data are required for the calculation, this process takes an unacceptable amount of time: In one version of the KPI Cockpit, users needs to wait upwards of 20 seconds until the calculation is finished, in an other version traperto pre-calculates all values and trades in the flexibility of the application. In order to improve the performance of the data collection process, the business structures will be created as graphs which can be traversed with a DFS-algorithm, which can be parallelized at each recursion.

Chapter 3

Mathematical definition of the data layer

Since the intention of the KPI Cockpit is to work for any customer, it can not be designed around a static domain model which represents the exact properties of a specific business structure. Instead, this section defines the properties of a graph which aims to provided a generalized data structure in which can represent any business structure.

3.1 Representing the relationship between two elements of a business structure

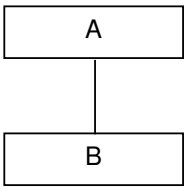


Figure 3.1: A undirected graph

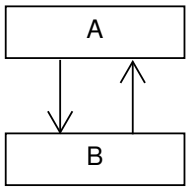


Figure 3.2: A bidirectional graph

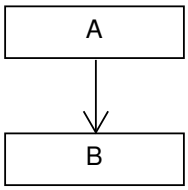


Figure 3.3: A directional graph

In order to capture the properties of the relationship between any two elements of a business structure, the graph representing their relationship needs to be directed: The graph

$$G = (V, E), V = \{A, B\}, E = \Big\{\{A, B\}\Big\}$$

can be described as a undirected graph (Figure 3.1) or as bidirectional graph (Figure 3.2), because E is a set of unordered pairs.

$$\{u, v\} \in E \implies \{v, u\} \in E$$

since $\{u, v\} = \{v, u\} \mid (u, v) \in V$. Therefor the direction of the relationship between A and B in the graphs Figure 3.1 can only be deduced from the context of the two vertecies (For example, if it is known that A is a supervisor and B is a supervised). Since the KPI-Cockpit needs to be as abstract as possible in order to assure its capability to represent any business structure, making the data-layer context specific is not an option. If the graph $G' = (V, E')$ is directed, meaning that E' is a set of *ordered* pairs

$$E' = V \bigtimes V, (\alpha, \beta) \notin E' \mid \alpha = \beta$$

, the direction of the relation can be deduced from the ordering of vertecies within an edge pair: The edge $E_1 = (A, B)$ defines that the relationship points from A to B, the edge $E_2 = (B, A)$ defines that the relationship points from B to A.

3.2 Representing a layer of the business structure

The direction of an edge in the graph G modeling the relationship from vertex A to vertex B indicates that B is part of A. In order to calculate KPIs within complex structures which consist of more than two vertecies, the KPI Cockpit needs to be able to deduce the *part of* relationship for all vertecies of a directed graph $G_A = (V_A, E_A)$. A is a layer of the business structure (e.g sales-channels or managers). V_A is a set of vertecies representing the entities within layer A and $E_A = \{V_A \bigtimes V_A \mid (\alpha, \beta) \notin E_A \, \alpha = \beta\}$ describe the edges between the vertices V_A. Using topological sorting, the *part of* relationship can be deduced for all vertecies within G_A. The algorithm produces a linear ordering of vertecies of G_A where a vertex u appears before a vertex v if $(u, v) \in E_A$. Since topological sorting is only possible in directed graphs containing no cycles, the graph modeling a layer of a business structure must not contain any cycles:

$$\exists P_{uv} = u_1, u_2, \cdots, v \implies \nexists P_{vu} = v_1, v_2, \cdots, u$$

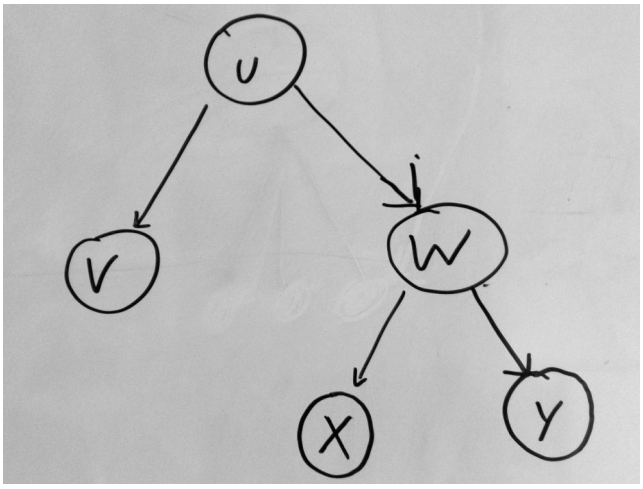


Figure 3.4: Calculating KPIs from children.

Topological sorting of the directed graph G displayed in Figure 3.4 produces a the order U, V, W, X, Y. This is the order in which KPIs-calculation must be executed: Since vertecies v and w are part of vertex u, the KPIs of v and w must be known before the KPIs of u can be calculated. The KPIs for v are available from data provided in a business report, w is calculated from x and y (The algorithm can be found in A.2).

3.3 Connecting data to the business

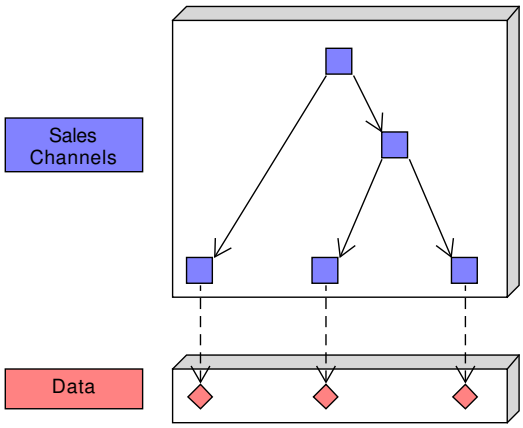


Figure 3.5: Hierarchical layer connected to data

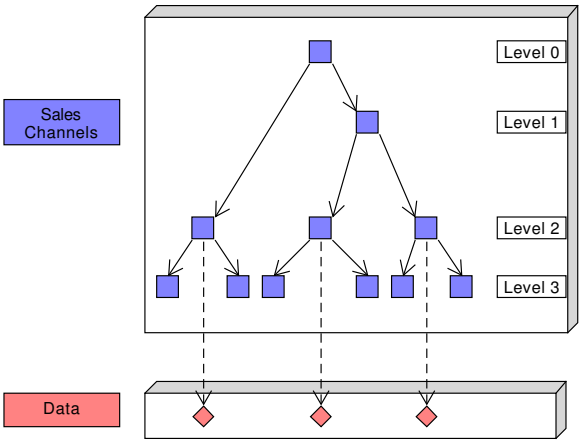


Figure 3.6: Wrong association between a organizational hierarchy and data

Connecting the data from the business reports to a hierarchy enables the KPI Cockpit the calculation of KPIs for all vertecies within the organizational layer the data is connected to. However, the data must not be connected a vertex v of layer L where the outdegree $d_L^+ > 0$: In Figure 3.6, data is provided for vertecies of the organizational layer on level 2 which have children ($d_L^+ > 0$). There is no information which would allow the application to deduce KPIs from these vertices to the vertecies of level 3.

3.4 Representing multiple business layers

In many cases a single hierarchy is not sufficient in order to represent an entire organizational structure. For example a large corporation will have more complexity than a singular hierarchy of sales channels split up into retailers and web-shops. The business structures of the companies currently using the KPI Cockpit consist of roughly ten distinct layers (e.g. areas, manager, business segments etc). Since the companies want information about the performance of their business from as many view points as possible, all layers of the business need to be incorporated into the application. This subsection describes describes a graph which consists of multiple and interconnected single layer graphs as described in section 3.2. This section describes the properties of the layers and the connections between them.

3.4.1 Properties of the multilayer graph

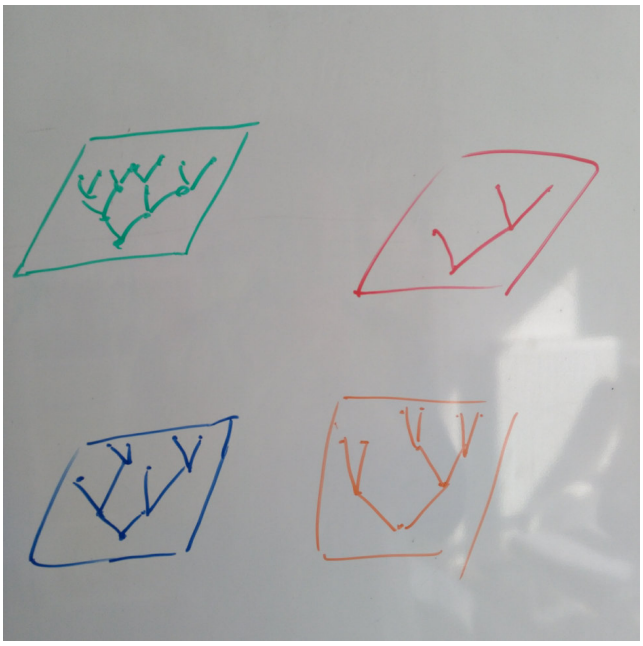


Figure 3.7: Multiple business layers

Let $G = (V_G, E_G, L)$ be a multilayered graph where L a set of business layers to be modeled in the KPI Cockpit (De Domenico et al. (2013)). V is the set of all vertecies in G which contains all entities of the business which are relevant to the KPI Cockpit. $V_G \subset V \bigtimes L$ describes the same set of vertecies in which each vertex is associated with the layer it is part of. Figure 3.7 shows the different layers and their intralayer graphs. Since each layer $l \in L$ models a separate layer of the business, a vertex can only exists on one layer within the Graph:

$$\{(u, L_\alpha) \in V_G\} \implies \{(u, L_\beta)V_G\} \mid u \in V, \{L_\alpha, L_\beta\} \in L$$

Because of the unique association of vertecies to layers, a graph $G_D = (V_D, E_D), V_D = (v, d) \in V_G, D \in L$ with the properties in described in subsubsection 3.2 can be established for all layers.

3.4.2 Properties of the interlayer graph

In order to enable the KPI Cockpit to calculate KPIs for all layers of the business, the graphs modeling the disjoint layers of the business structure need to be interconnected. Interconnecting the layers creates paths from vertecies which are not part of layers adjacent to data to vertecies in layers which are adjacent to data. If there exists a path to any given vertex of the multilayer network to data from business reports, KPIs can be calculated for that vertex. This subsection defines that there can not be loops between any two layers and that there may not be transitive relationships within a graph connecting two layers. Finally, the graph containing all inter-layer graphs is presented as a simple directed graph in which layers and their inter-layer graphs are represented by vertecies and edges.

Acyclic interlayer connections

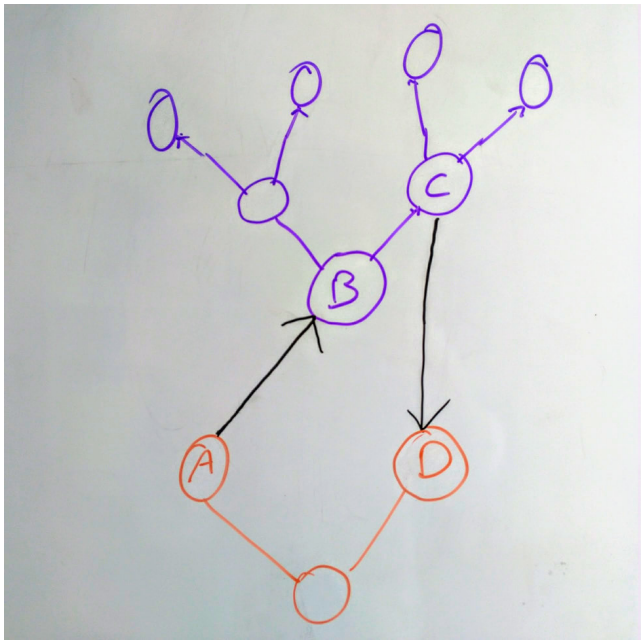


Figure 3.8: Loop in interlayer connections

The interlayer graph $G_I(A, B) = (V_{A,B}, E_{AB})$ connects the layers $A, B \in L$ of the business structure where

$$V_A = \Big\{(u, A) \in V_G \mid u \in V, A \in L\Big\}$$

$$V_B = \Big\{(v, B) \in V_G \mid v \in V, B \in L\Big\}$$

$$V_{A,B} \subset \Big\{V_A \cup V_B\Big\}$$

$$E_{AB} \subset \Big\{V_A \bigtimes V_B, ((u, A), (v, B)) \in E_{AB} | (u, A) \in V_A, (v, B) \in V_B\Big\}$$

The graph between any two layers A and B allows each vertex $v \in V_A$ to be connected with any vertex $u \in V_B$. If A is connected to B via an interlayer graph $G_I A, B$, there can not be a interlayer graph $G_I B, A$ connecting layer B to layer A. This rule prevents loops between A and B, which is important because the edge between any nodes n and m within the graph G implies that m is part of n . An edge from m to n would make this relation ambiguous.

Non-transitive relations in inter-layer graphs

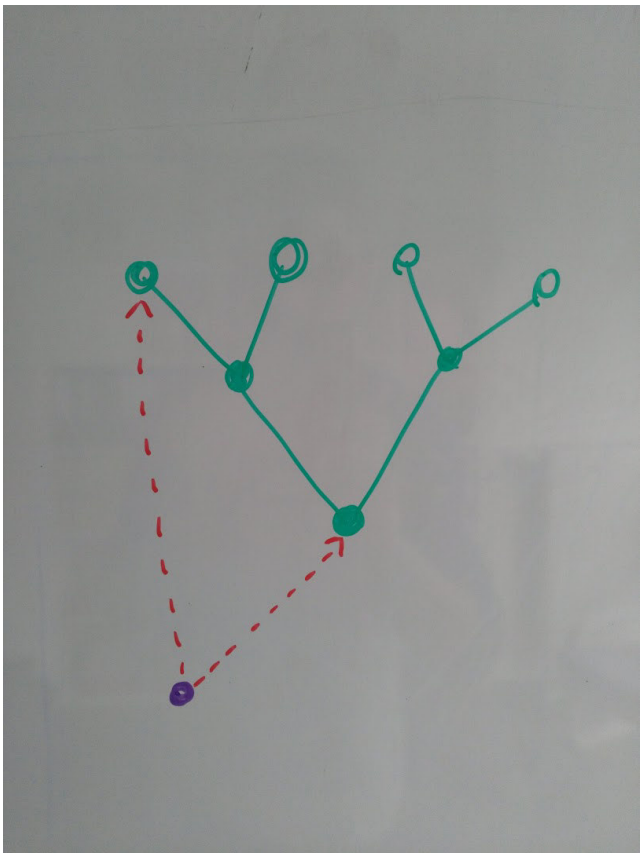


Figure 3.9: Transitive relationships in interlayer connections

$$G'_{D_B} = (V'_B, E'_B) \mid V'_B \subset V_B, E'_B \subset E_B$$

$$\exists((u,a),(v,b)) \in E_{AB} \implies \nexists((u,a),(w,b)) \in E_{AB} \mid \{(v,b),(w,b)\} \in V'_B, (u,a) in V_A$$

In order to prevent redundant connections between layers, a vertex $(w,b) \in V_B$ of layer B must not be connected to a vertex $(u,a) \in V_A$, of layer A if and edge connects (u,a) to another vertex $(v,b) \in V_B$ which creates a subgraph (G'_{D_B} of the layer B's graph G_{D_B} which both (v,b) and (w,b) are part of. Figure 3.9 visualizes the redundancy of the edge $((u,a),(w,b))$: The edge $((u,a),(v,b))$ creates a subgraph which includes (w,b).

3.4.3 Simple directed graph representing layers and their connections

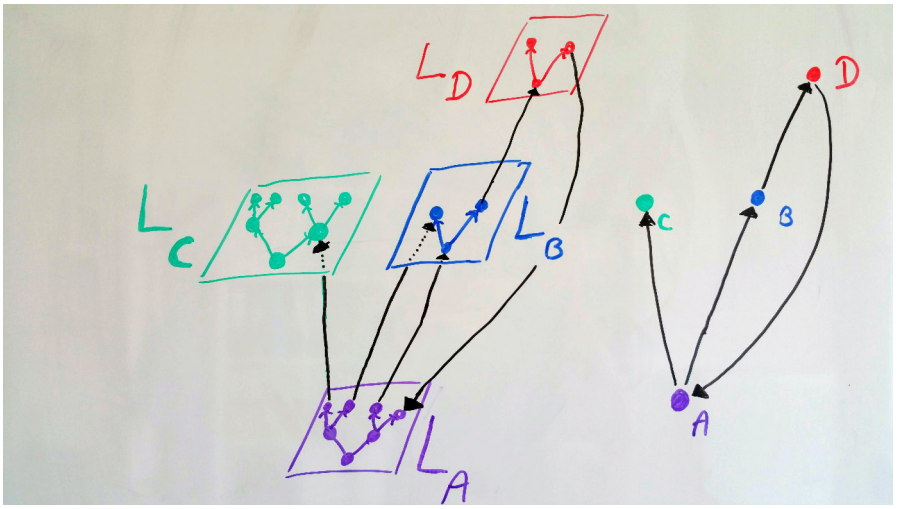


Figure 3.10: Layers represented as vertecies

Since vertecies can only exist within one layer (see subsection 3.4.1) and acyclic nature of the inter-layer graphs (see subsection 3.4.2), the layers and the edges between them can be interpreted as a single layer directed graph in which each layer is represent by a single vertex and the set of edges between any two layers by a single edge (see Figure 3.10). This reduces the complexity of the defining graphs according to the properties described in this chapter, because the upholdance of the definition can be validated based on the simplified interpretation of the graph.

Chapter 4

Creating graph representations of business structures

The data processing application needs to model the business structures of customers in order to calculate KPIs for all entities within their organizations. Since these companies are often large cooperation consisting of tens of thousands of highly connected entities, manually modeling these structures is not an option. Therefor the structure of organizations needs to be derived from internal customer data used for business administration. This chapter describes a program which can creates a graph representation (see chapter 3) from any file structure which presents data in a logical fashion. For this, the program only requires input contextualizing the data available in the input files. The mathematics involved in this program are closely related to the set and relational-theory of the *Relational Model*(Codd (1970)). In order to ease the reader into the subject, the first section provides a simple example of a business structure and input data is provided. This section also visualizes the graph created by the application. The following sections describe how set and relational mathematics of the *Relational Model* are used to create graphs from any logically arranged data structure. Finally, the design and implementation of this program as a Java application is discussed.

4.1 Exemplary business structure, input files and application output

This section briefly describes a simple business structure and three Comma Separated Values (CSV)-files providing data about entities within the modeled organization. This example will be used to trough out this chapter to explain the program responsible for creating multi-layer directed acyclic graphs (Chapter 3) for arbitrary business structures. However, the reader needs to keep in mind that the application is intended to work with any file and data structure from which specific values can be extracted. The amount of data to be processed of the example is negligible, while the actual files to be imported contain information about thousands of interconnected business entities. Next, three CSV-files are given which form the input data of the application which produces the graph discussed in the third subsection of this section.

4.1.1 Exemplary business structure

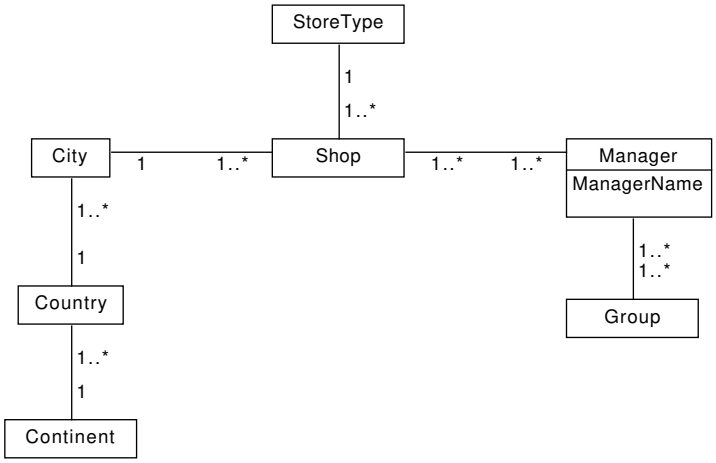


Figure 4.1: Simple domain model of a business structure

Identifying fields omitted.

Figure 4.1 displays the structure of an imaginary company for which a graph representation needs to be created. The core of the business are shops, which are managed by one or more managers. Managers are responsible for one ore more shop and part of different groups of managers. The shops are separated into different types according to their size. Since the company is active on the global market, each shop is associated with a location consisting of a city, country and a continent. The domain model (Figure 4.1) also visualizes the meta information which needs to be provided to the application: The next subsection gives some exemplary input files for the application, which can only be interpreted if the relationship between different sets of data is provided. For example, the application needs to know that the column *ManagerName* and *ManagerID* are part of the domain type *Manager*.

4.1.2 Exemplary input files

ShopID	City	Country		Continent	ShopType
$P_{F_1 1}$	$P_{F_1 2}$	$P_{F_1 3}$	$P_{F_1 4}$	$P_{F_1 5}$	$P_{F_1 6}$
1	London	England	A-2F	Europe	Large
2	New York	USA	B-2A	America	Large
3	Boston	USA	B-2A	America	Medium
4	Tokio	Japan	B-2A	Asia	Medium
5	Berlin	Germany	C-6F	Europe	Medium
6	Berlin	Germany	A-5F	Europe	Large

Table 4.1: CSV file F_1 displayed as a table.

ManagerID	Group	ManagerName
$P_{F_2 1}$	$P_{F_2 2}$	$P_{F_2 3}$
63	Group2	Smith
122	Group3	Johnson
22	Group1	Meier
22	Group2	Meier
22	Group3	Meier
64	Group2	Jansen
88	Group1	Peters

Table 4.2: CSV file F_2 displayed as a table.

ShopID	ManagerID
$P_{F_3 1}$	$P_{F_3 2}$
3	64
4	22
4	64
1	122
5	88
2	22
2	64
1	63
6	88

Table 4.3: CSV file F_3 displayed as a table.

Blue rows show meta information provided from an external source. Gray rows indicate set identity. Neither are part of the file.

The files F_1, F_2 and F_3 provide structural information about the business. Each of these files is well structured and separate data entries can easily be extracted: Each row in the files provides data about one business entity, F_1 contains data about shops. More specifically, the city, country and continent in which it is located are given and the type (size) of each shop is provided. F_2 provides data about managers by associating each manager identification number with a name and one ore more groups (The poor normalization of this table is an accurate representation of the actual input files). F_3 associates shops to their managers.

4.1.3 Resulting graph

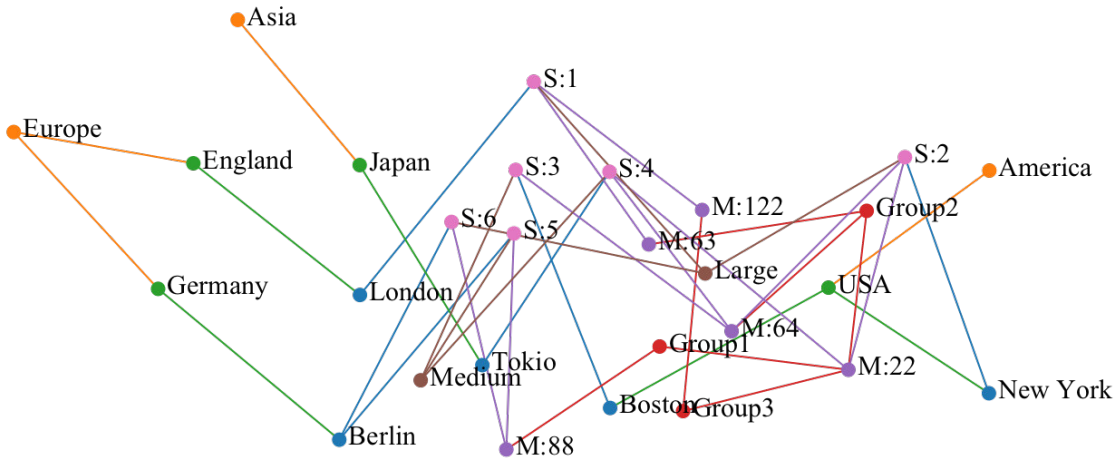


Figure 4.2: Graph extracted from the exemplary input files F_1, F_2 and F_3

Generated with D3js based on the output of the Graph-Extractor. The graph is presented as undirected graph to reduce visual clutter.

Figure 4.2 visualizes the output of the application in form of an actual graph. The application has removed all duplicate information from the input data and replaced it with unique edges. For example, *Europe* is continent of three shops (1, 5 and 6). This entries are replaced by a single vertex "Europe" by the application and correctly connected to the vertices "England" and "Germany".

4.2 Mathematical definition of the program

This section describes the processes involved in converting multiple input files to a graph representation of a business according to the graph definition described in chapter 3.

4.2.1 Accessing values from data sources

In order to increase or allow the usability of the application implementing this program, the data set contained in the input file are addressed with unique string identifiers in each file. Associating these string identifiers with the appropriate data sets is the first step in making a file readable for the application (See fist code snipped, part **1** in Snippet 5). The following sections use the set identifier (e.g $P_{F_1 1}$) where necessary and the string identifier (e.g *City*) where possible.

4.2.2 Creating a set of property sets from all files

The application needs to import each file from the set of F by producing a set of domain properties $S_F = \{S_x\}$ where each set contains elements of one attribute type (e.g, the name of a shop, the age of manager, a unique identifier of a manager). Since the data from all files needs to be considered, the attributes from all files need to be defined as a single set. There is a property P which defines some aspect of the business structure and is shared by all business entities of the same type[1]. The business entities have instances (values) of these properties which can be extracted from the input files in F. Since there are multiple properties in a file, there is a set of property accessible in each file $I \in F$:

$$D_I = \{P_{I_1}, \ldots, P_{I_n} | I \in F\}$$

[1]The term *Property* is equal to the concept of *instance variables* in object oriented programming: A class defines the the type of an instance variable and each instance of that class may or may not have an value for this attribute.

This set represents raw data which can be extracted from each file. The definitions of D_{F_1},D_{F_3},D_{F_3} are provided in their respective tables (see Table 4.1,Table 4.2,Table 4.3, gray rows).

Figure 4.3: Visualization of D_I

Color depends on the source file. Labels contain the value of the first column.

Figure 4.3 visualizes all sets in D_I. At this point, no meta information is available to the application and the contents of the input data can not be interpreted. Therefor each line from the input files (Table 4.1,Table 4.2,Table 4.3) is seen as seen as an unique entity.

In order to more information from the input file, projections of the sets in D_I need to be defined:

$$E_A \subset R_A \mid R_A = \prod_{S \in D_A} S, A \in F$$

E_A could be interpreted as the normalization set of $D_A \mid A \in F$, since it essentially defines multiple tables derived from one table. The accuracy in representation and the complexity of the resulting graph depends on the degree of normalization or amount of sub-tables created from D_I. The set of all normalization sets will be addressed as E ($E_B \in E \mid B \in F$).

Defining the normalization set E for the provided example

Since a graph upholding the properties defined in chapter 3 can be defined and verified based on *layers* and the connections between them (see subsection 3.4.3), this program requires meta information which defines the layers present in a business structure. Since each vertex is part of a layer, it can be and connected to other vertices based on the connections of the its to other layers. In order to create an accurate representation of the domain model, the seven entity types or *layers* depicted in Figure 4.1 need to be extracted from the provided input files. The entity types *Shop*, *City*, *Country*, *Continent* and *StoreType* are defined in F_1 (Table 4.1, *Manager* and *Group* are contained in F_2 (Table 4.2. Therefor the normalization set for these two files are defined as:

$$E_{F_1} = \{(\text{ShopID}), (\text{City}), (\text{County}), (\text{Continent}), (\text{ShopType})\}$$

$$E_{F_2} = \{(\text{ManagerID}, \text{ManagerName}), (\text{Group})\}$$

F_3 does not contain an additional type of the domain level. However, on a mathematical or database level it contains two entities: The entities defined by the set $P_{F_3 1}$ (= *The ManagerIDs contained in* F_3) and the type defined by the set $P_{F_3 2}$ (= *The ManagerIDs contained in* F_3), Therefor the normalization set for F_3 is defined as:

$$E_{F_3} = \{(\text{ManagerID}), (\text{ShopID})\}$$

4.2.3 Deriving multiple domain types from D_I

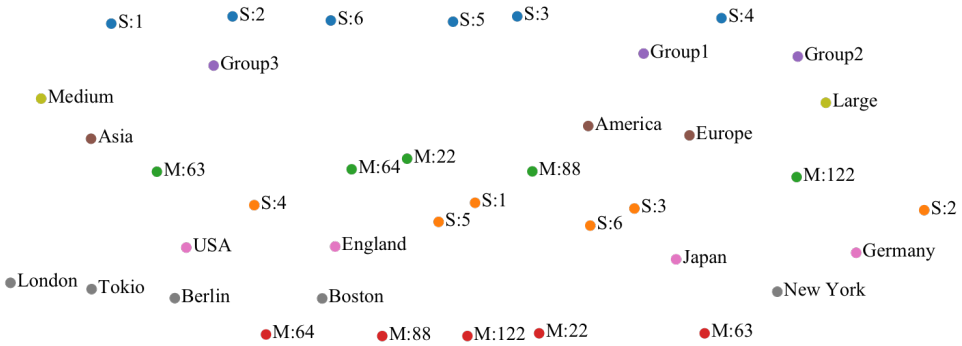


Figure 4.4: Visualization of entities extracted based on the normalization sets.
Each entity type resulting from a normalization set has a unique color.

Based on the normalization sets, the entities displayed in Figure 4.4 can be extracted. Since entity instances are defined by relations on D_I producing sets, all redundant information is removed from the data. Because of this, only one vertex is created for per entity. Because of this the one vertex is created for the continent "Europe" (see Table 4.1 and one vertex is created for the Manager with *ManagerID* 22 from F_2 (see Table 4.2). At this point, the application creates two vertices for each manager and shop, because there are two separate sets defining instances for both types. This issue will be resolved next.

4.2.4 Joining normalization sets

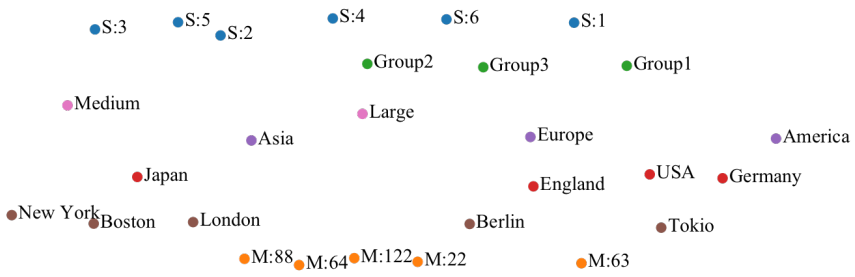


Figure 4.5: Visualization of entities extracted based on joined normalization sets

.

Prefix S for *Shops*, Prefix M for *Managers*

Depending on the file structures of the $A \in F$ there may be multiple normalization sets in E describing the same entity. In the given example, *Shop* and *Manager* are both defined by two sets of properties(*Shop* in F_1 and F_3, *Manager* in F_2 and F_3). In order to to combine the information of these property sets, they need to *naturally* joined. Formally, a natural join is defined by:

$$R * S = \{(a, b, c) : R(a, b) \wedge S(b, c)\}$$

In the given example the set identities are provided trough meta information. Since the sets $P_{F_2 1}$ and $P_{F_3 2}$ have the same identity, all normalization sets which contain these two set can be joined.

$$X = (\text{ManagerID}, \text{ManagerName}) \mid X \in E_{F_2}$$
$$Y = (\text{ManagerID}) \mid Y \in E_{F_3}$$
$$X * Y = \{(ManagerID, MangagerName) : X \wedge Y\}$$

Since the same logic applies to the *Shop* entity and the relations (ShopID) $\in P_{F_1}$ and (ShopID) $\in P_{F_3}$, there is only one set of *Manager* and *Shop* displayed in Figure 4.5. At this point, the join operation does not provide any additional information about these two entity types, because the two sets contain the same data and are simply merged. The next subsection describes how join operation are used in order to connect entities.

4.2.5 Connecting entity types

At this point, the program has extracted all unique entities from the three input files. However, the resulting graph does not represent the business defined in Figure 4.1, because the connections of the structure contained in the input files are not part of the extracted data. The normalization sets in E defined in 4.2.2 do not contain information which would allow the program to extract connections between entities. Therefor, the normalization sets will be adapted to contain unique identifiers of the domain types they are connected to. Since the principal is the same for all connections, this subsection will only focus on the relationship locations and shops. Therefor, the normalization set for the data extraction is defined as:

$$C = \{\{(\text{ShopID, City}), (\text{City, Country}), (\text{County, Continent}), (\text{Continent})\}\}$$

ShopID	City
$P_{F_1 1}$	$P_{F_1 2}$
2	New York
3	Boston
5	Berlin
6	Berlin
1	London
4	Tokio

City	Country
$P_{F_1 2}$	$P_{F_1 3}$
New York	USA
Boston	USA
England	England
Berlin	Germany
Japan	Asia

Country	Continent
$P_{F_1 3}$	$P_{F_1 5}$
USA	America
Japan	Asia
England	Europe
Germany	Europe

Continent
$P_{F_1 5}$
America
Asia
Europe

Table 4.4: The relations created from the normalization set C

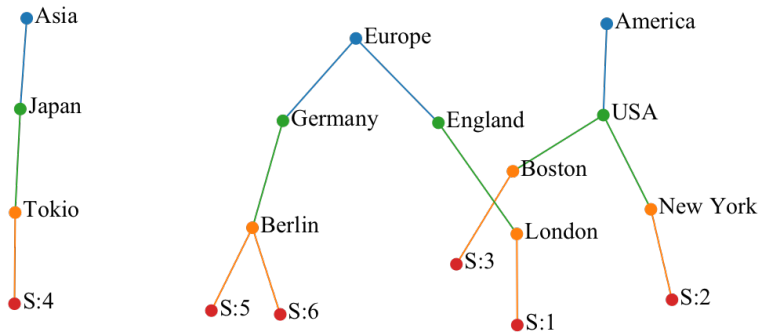


Figure 4.6: Visualization of *Continent*, *Country*, *City* and *Shops*

.

Colors indicate domain type. Edges are colored according their source vertex.

The tables in the tables in Table 4.4 show the data extracted when the normalization set C is used. Each row of in the tables represent one entity of the business structure for which one vertex is created (See Figure 4.6). With the information in these tables, the connections between the entities defined in C can be recreated by resolving the foreign keys in each relation: The foreignkey "Europe" is part of the tuples (*England*, *Europe*) and (*Germany*, *Europe*). These two foreign keys are represented by the two edges {*Europe*, *Germany*} and {*Europe*, *England*} (see Figure 4.6).

Connecting entity types through join operation

ManagerID	Group	MangerName	ShopID
$\{P_{F_21}, P_{F_32}\}$	P_{F_22}	P_{F_22}	P_{F_22}
22	Group1	Meier	4
22	Group3	Meier	4
22	Group1	Meier	2
22	Group3	Meier	2
...	...	...	...

Table 4.5: Excerpt from the joined relation describing managers.

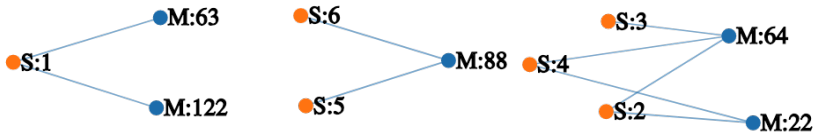


Figure 4.7: Visualization of *Mangers* and the *Shops* they are responsible for
.

Colors indicate domain type. Edges are colored according their source vertex.

The previous subsection described the process of joining normalization sets based on their primary identifiers. In case of the *Manager* and *Shop*, the relationship between the types is defined in a separate data set. Therefor the application needs to take two relations into account in order to fully capture the details of the *manager* type: The relation $(ManagerID, ManagerName) \in R_{F_2}$ and the relation $(ManagerID, Group, ShopID) \in R_{F_3}$. By joining the two relations on the *ManagerID*, a relation describing the managers and their shops is created (See Table 4.5) By merging the tuples of this relation on the primary identifier *ManagerID*, one set of data is produced which contains all relevant information of each manager:

$$ManagerID = \{22\}, \text{Group} = \{\text{Group1}, \text{Group2}, \text{Group3}\}, ManagerName = \{\text{Meier}\}, \text{ShopID} = \{4,2\}$$

This set represent the final structure of the data extracted from the input files. The extracted properties are sets of strings, because the graph extraction application collects each unique entry found for each property for each entity and ignores the multiplicity and typing of properties. In order to keep the structure of properties consistent across all data sets, the application creates singleton sets for properties with single values (e.g having single strings and sets of strings when processing *ShopIDs* of the *Manager* unnecessarily increase the complexity of interacting with the extracted data).

4.3 Providing meta information

This section describes a DSL used for providing the necessary meta information to the application. First, the configuration of input files is discussed which allows the application to interpret the data in the source files. Based on this configuration, `DomainTypes` and `DomainDimensions` are established which define the final *normalization sets* described in subsection 4.2.5. In order to make the configuration process as fluent as possible, the DSL is build into Java and uses fluent interfaces for creating the required information.

4.3.1 Fluent interface for defining data sets

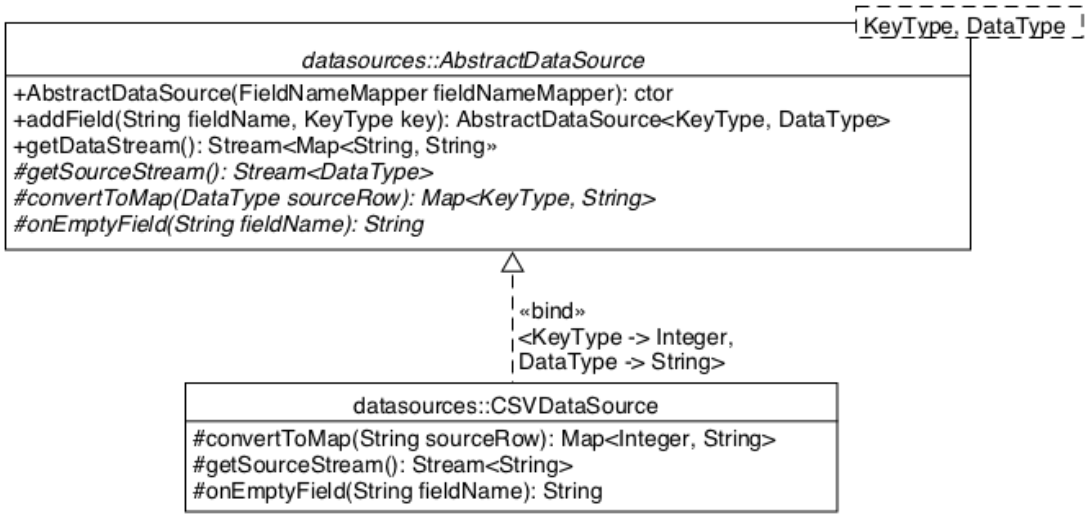


Figure 4.8: Class diagram of `AbstractDataSource` and its implementation `CSVDataSource`

The class diagram in Figure 4.8 shows the part of the application responsible for generalizing access to source files by creating associative arrays which map field names to values. The field names for the example domain model are defined in the tables 4.1,4.2 and 4.3. The `AbstractDataSources` provides a fluent interface for chained linking of set-identifiers to data trough the `addField`-method (see Figure 4.8):

```
AbstractDataSource file1 = new CSVDataSource(filePath)
      .addField("ShopID", 0)
      .addField("City",1)
      .addField("Country", 2)
      .addField("Continent", 4)
      .addField("Type",5);
```

Code Snippet 1: Defining the field names for F_1

4.3.2 Defining multiple data sources from one input file

ShopID	City	Region	Country		Continent	ShopType
$P_{F_{11}}$	$P_{F_{12}}$	$P_{F_{13}}$	$P_{F_{14}}$	$P_{F_{15}}$	$P_{F_{16}}$	$P_{F_{17}}$
5	Berlin	East	Germany	C-6F	Europe	Medium
3	Boston	Massachusetts	USA	B-2A	America	Medium

Table 4.6: Example of additional data sets contained within F_1.

In some cases an input file can contain multiple types of data sets which can not be imported with same logic. The application needs to import multiple data sets from a file is required, because changes to the business structure are provided by the customer in form of a updated version of the import files. Manually adapting the file structure to have a consistent logic is not an option, because the same this process will have to be repeated for every change in the business structure. An example of an additional data set is displayed in Table 4.6, which may be part of file F_1 (see Table 4.1). This data set can not be imported with the configuration listed in Figure **??**, because the additional *Region* column changes the indexes of the all columns to its right. By

defining an additional `CSVDataSource` for rows containing the additional column, import errors can be avoided which relatively low effort.

4.3.3 Defining the structure of data sources with generics

The generics `DataType` and `KeyType` define the structure of the data input (See Figure 4.8). They move the responsibility of defining the concrete nature of the input files to the implementation of the `AbstractDataSource`. It is not relevant for the abstract class how input files are structured, because it just assumes that there is a way to produce a stream of raw data (`getSourceStream`) which can be transformed to a map linking set identifiers to appropriate data(`convertToMap`). This is made possible by the `DataType`-generic, which defines the type of data in the stream of raw data (`getSourceStream`) and the parameter-type of the method responsible for converting its elements (see `convertToMap`). In case of the `CSVDataSource`, this generic is a `String` because CSV files are read line by line. The key type of the map produced by the `convertToMap`-method is defined by the `KeyType`-generic, which is determined by the method of accessing values within the unspecified data structure `DataType`. In case of the `CSVDataSource`, this generic is an `Integer`, because each line from the input file is split on commas, producing an integer-index array. With `DataType` and `KeyType` defined, the `CSVDataSource` can transform the input files F_1 (Table 4.1), F_2(Table 4.2) and F_3(Table 4.3): By passing the first line 1,London,England,A-2F,Europe,Large of F_1 to the `convertToMap`-method, which accepts a string due to the `DataType`-generic, the following map is produced:

$$M_1 = \Big\{(0 \mapsto 1), (1 \mapsto \text{London}), (2 \mapsto \text{England}), (3 \mapsto \text{A-2F}), (4 \mapsto \text{Europe}), (5 \mapsto \text{Large})\Big\}$$

With the meta information provided by the user (see Figure **??**), the `getDataStream`-method can transform the integer keys of this map to the appropriate set identifiers:

$$M_S = \Big\{(\text{ShopID} \mapsto 1), (\text{City} \mapsto \text{London}), (\text{Country} \mapsto \text{England}), (\text{Continent} \mapsto \text{Europe}), (\text{ShopType} \mapsto \text{Large})\Big\}$$

Finally, the keys of this map can be translated according to the input provided through the DSL (see Figure **??**:

$$M_S = \Big\{(\text{ShopID} \mapsto 1), (\text{City} \mapsto \text{London}), (\text{County} \mapsto \text{England}), (\text{Continent} \mapsto \text{Europe}), (\text{ShopType} \mapsto \text{Large})\Big\}$$

The implementation of the `CSVDataSource` can be found in Snippet 6.

4.3.4 Fluent interface for defining normalization sets

This subsection describes a fluent interface for defining *DomainTypes* which implement the normalization set defined in subsection 4.2.5 with additional typing of entity properties.

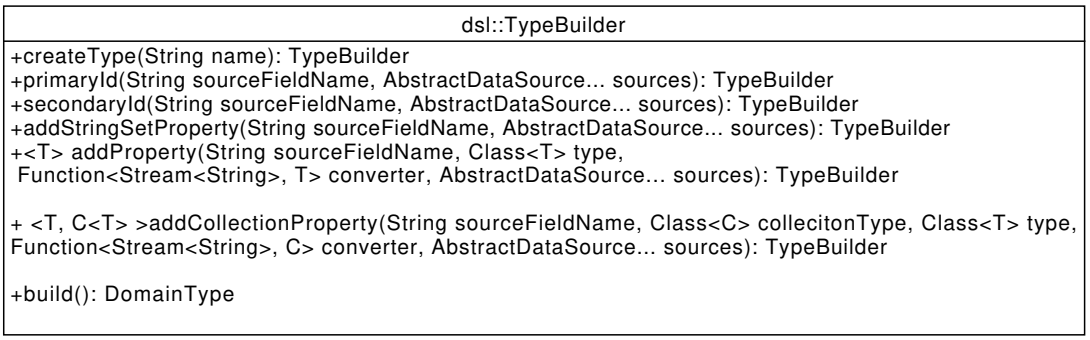


Figure 4.9: Class diagram of the TypeBuilder

With the field names for the files defined, the normalization sets desctibed in subsection 4.2.5 can be created. Since the normalization sets for the input data depends on the business structure (see Figure 4.1), these sets are

```
DomainType manager = TypeBuilder.createType("Manager")
.primaryId("ManagerID",file2,  file3)
.addProperty("ManagerName", String.class,
stringStream->stringStream.findFirst().get(),file2)
.build();
```

Code Snippet 2: Defining the manager with the DSL

defined by recreating the domain model of the business with another fluent interface: The `TypeBuilder`. This builder creates *DomainTypes* consisting of identifier- and property fields, which represent the *normalization sets*. This interface also makes it possible to define the type of properties and their multiplicity: Since the extraction application treats all entities as sets of strings, the user needs to specify how the extracted values for the property can be transformed to the desired type. This is implemented with the *Visitor pattern* which uses Java's **Function** as visitor. The `TypeBuilder` links each property to a Function which consumes a `Stream` of `Strings` and converts each entry to objects of the desired type (see `addProperty` and `addCollectionProperty` in Figure 4.9). The use of the visitor pattern makes it possible to define transformation operations on input data on the DSL level of the graph extraction application. In case of the *Manager*, the property *ManagerName* is the typed as **String** (*String.class*, second parameter of the `addProperty`-method in in line 3 of Snippet 2). The value of *ManagerName* is defined by the lambda implementation of the `converter`-Function, which returns the first element from the stream of values extracted for the property(Lines 3-4 in Snippet 2).

Method generics for type enforcement The `TypeBuilder` guides the user in defining correct data transformations for properties by the means of method generics: If the user sets the `type`-parameter to *Double.class*, the `converter`-parameter will only accept a `Function<Stream<String>, Double` as input (See method generic *T* in Figure 4.9). The same technique is also used for creating collections for properties which can contain multiple values: The `addCollectionProperty` will only accept a conversion Function which produces a `Collection` of type `C` containing the generic type `T`.

4.3.5 Fluent interface for combining entity types

At this point, the `DomainTypes` defined in the previous subsection are not connected. The DSL offers yet another fluent interface for connecting the `DomainTypes` defined in the previous subsection: The `link-type` method of the `DomainDefinition`, which connects the provided `DomainTypes`.

```
DomainDefinition definition =
new DomainDefinition().
linkTypes(type,shop).
linkTypes(group,manager).
linkTypes(manager,shop).
linkTypes(continent,country,
          city,shop);
```

Code Snippet 3: Defining connections between `DomainTypes` trough the DSL

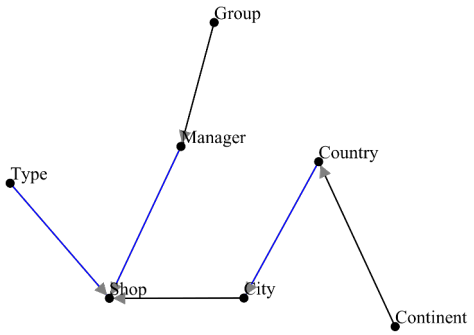


Figure 4.10: Visualization of the connections between `DomainTypes`.

The code listing Snippet 3 shows the DSL-code for recreating the connections of the domain model displayed in Figure 4.1. The `DomainTypes` provided to the `linkTypes`-method are connected by registering the *B* as child of *A* and *A* as parent if the are provided ordered alphabetically. The created parent-child relationships between the `DomainTypes` of the example business structure can be seen in Figure 4.10, where the source of each edge is

the parent type and the sink the child type. The `linkTypes`-method connects two `DomainTypes` by adding the relation containing both primary identifiers of the two types to set of relations to be extracted for each type.

4.3.6 Enforcing the properties of the graph

The graph created by the graph extraction application can be verified based on the meta data provided through the DSL, since the *DomainTypes* and their connections are equal to the *layers* described in section 3.4: If the graph of *DomainTypes* and the edges between them do not contain back edges, the graph of business entities created based on the *DomainTypes* can not contain any back edges. In order to test for back-edges, the *Depth-first-search*-algorithm needs to be executed on the graph. The algorithm can classify the edges of the graph into the categories *tree edge*, *cross edge*, *forward edge* and *back edge* (Cormen et al. (2009)).

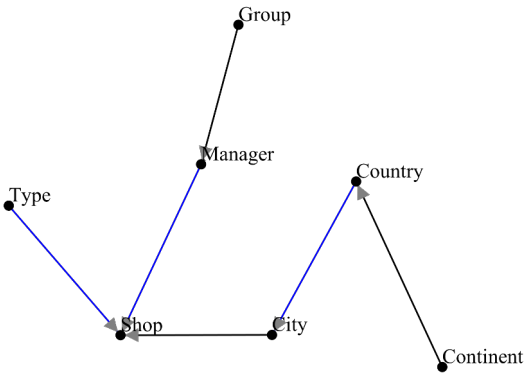


Figure 4.11: Visualization of the example domain model graph.

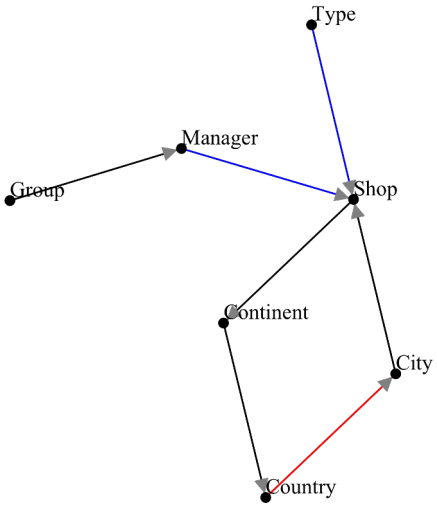


Figure 4.12: Visualization of a domain model graph containing a loop.

Black: Tree edge, Blue: Cross edge, Red: Back edge, Arrows indicate edge direction

Figure 4.11 and Figure 4.12 show the edge classifications produced by the *Depth-first-search* for two graphs consisting of `DomainTypes` defined with the DSL. The first graph mirrors the domain model depicted in Figure 4.1 and defines the structure of the graph extracted for the exemplary business structure discussed in this chapter (see Figure 4.2). In the second graph, an edge pointing from the shop to the continent has been added. This creates a cycle between the vertices *City*, *Shop*, *Continent* and *Country*. The edge between *Country* amd *City* is identified as back edge, because the vertices of the graph are processed in lexicographical order: The algorithm starts at the *City*-vertex from which it visits the *Shop*-vertex. Next, the *Continent* and the *Country* are visited. Since all child vertices of *City* have been visited by the algorithm when the edge between *Country* and *City* is discovered, it is identified as a back edge. If a back edge is detected in the graph of `DomainTypes`, the application throws an exception before the actual data processing begins. The implementation of this algorithm can be found in Snippet 7.

4.4 Creating interconnected vertices

This section describes how the graph extraction application creates vertices from data extracted from the source files and how the resulting graph can be used.

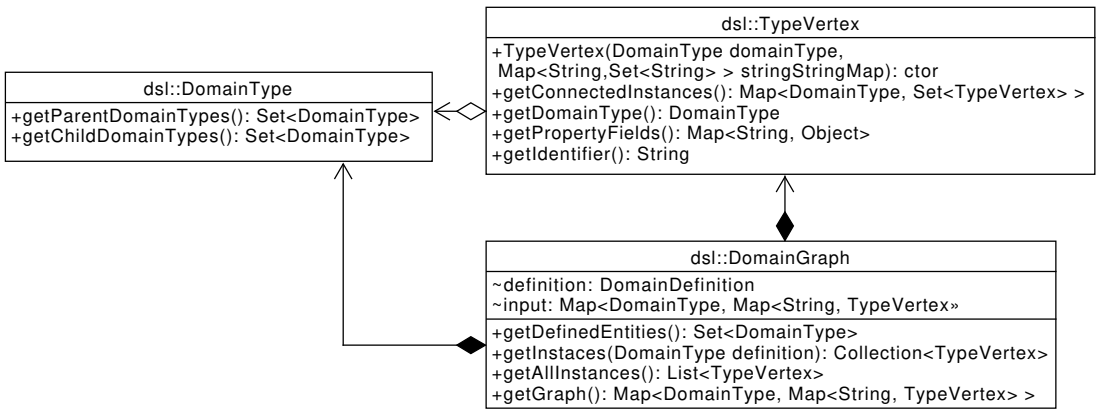


Figure 4.13: Class diagram of the `DomainGraph`.

Once the provided meta information is validated, the application executes the data transformation described in section 4.2. When this process is complete, the application has a map which connects each `DomainType` with data sets which each contain all information of one entity of that type. From each of these entity data sets a `TypeVertex` is instantiated (See constructor of `TypeVertex`) and made accessible trough the `DomainGraph`. The

4.4.1 Converting extracted values to typed properties

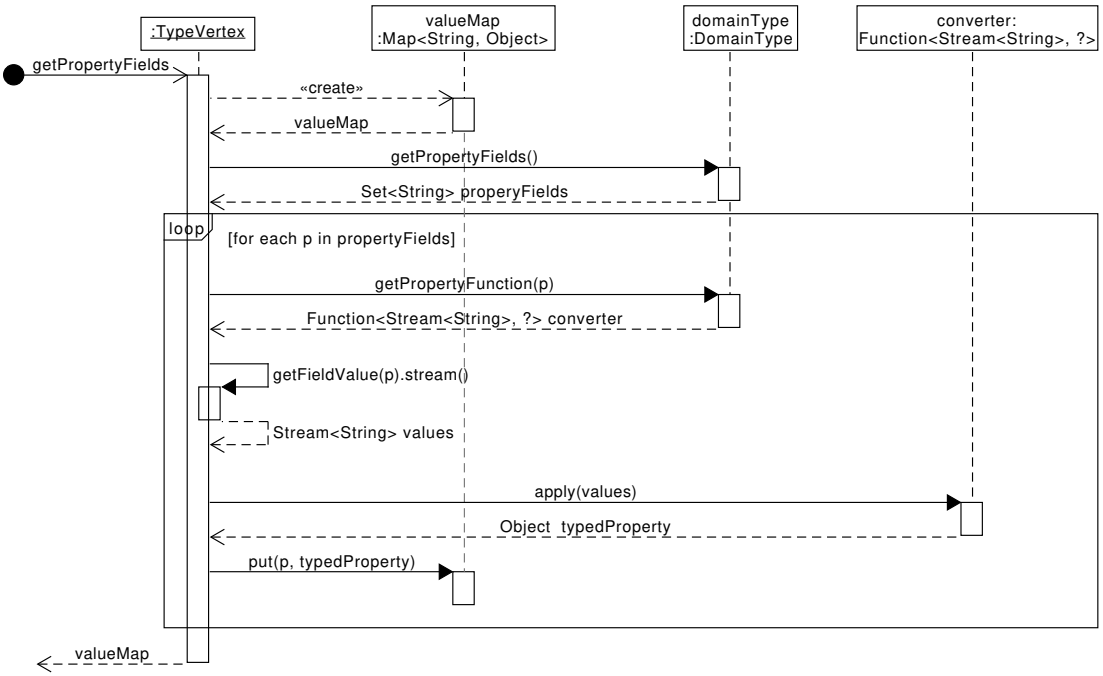


Figure 4.14: Sequence diagram of converting property values to typed objects.

The constructor of `TypeVertex` also requires a `DomainType`, because the properties of the the vertex are extracted as sets of strings without information about their type or multiplicity. In order to convert these sets to the appropriate properties, the `TypeVertex` requires the meta information provided by the user: For each property of the `TypeVertex`, the `DomainType` has a Function which transforms the set of strings to a collection of objects or a single objects. The process of retrieving a map containing all properties is depicted in

the sequence diagram Figure 4.14: Since the property value sets may contain, identifiers of the `TypeVertex` or foreign keys foreign keys to other `DomainTypes`, the `TypeVertex` requires information about which of the extracted data sets contain properties (`getPropertyFields()` in Figure 4.14). For each property identifier, the `TypeVertex` retrieves the user defined conversion `Function` (`getPropertyFunction(p)` and applies it to a `Stream` of the strings sets containing the raw values of the property(`getFieldValue(p).stream()` and `apply(values)` in Figure 4.14).

4.4.2 Replacing foreign identifiers with references

A subset of the foreign identifiers obtained during the extraction process may not be resolvable, because the entities they identify were not extracted from the input files. This problem may occur due to inconsistencies in the input files or because entities were ignored during the input process(For example, closed shops are ignored because they are not of relevance for KPI calculation, but the associations between managers and shops still reference the closed shops). For this reason, the foreign identifiers are not a reliable way of inferring connections between entities. Therefor, the foreign identifiers contained in the `TypeVertecies` are replaced with object references to the `TypeVertices` they identify (see code snipped Snippet **??**, lines 274-301). During this process all foreign identifiers which can not be resolved are ignored and the final data structure only contains valid connections between entities. This also reduces the amount of operations required for processing the graph structure, because invalid links are never processed.

4.4.3 Undirected representation of edges

The connections between vertices are represented as undirected edges, meaning that if vertex A is the parent of vertex B, both vertexes will have a reference to each other. This leads to the situation discussed in section 3.1, in which the direction of an edge can not be inferred between the two vertices. However, the edge direction between vertecies is still known, because it is part of the meta information available in the `DomainTypes`, which differentiates between *parent* and *child* types. This decreases the complexity of the edge representation in the data structure, because edges to `TypeVertecies` of *child* and *parent* `DomainTypes` can be stored without considering the edge direction.

Chapter 5

MapReduce graph processing-application

In order to calculate KPIs for all vertices of the business structure, the graphs created with the application described in chapter 4 need to processed by a graph processing application. This chapter describes and application, following principals of *MapReduce* (Dean & Ghemawat (2004)), which navigates the graph though a user defined path while extracting relevant data from the graph. When the traversal is finished, the collected values are *reduced* by a user defined operation in order to calculate KPIs. In order to execute the mapping process as fast as possible, the graph is traversed with a BFS-algorithm executed in parallel.

5.1 Meta information in model classes

Since the graph extracted by the graph extraction application is a mathematical object, it needs to be contextualized in order to be processed. Therefor, the meta information provided by the user in form of the `DomainTypes` (see subsection 4.3.4), which describe the entities of the business structure, is persisted in form of JSON schemas. Theses schemas are used to generate model classes for each graph, which the graph extraction application uses remove the high level of abstraction of the data structure and offers a domain specific programming interface in form of the `QueryContainer`. This makes it possible to use the same graph processing application for any business structure while providing an easy to understand way of interacting with the graph.

5.2 Programming the map-reduce process

This section describes the interaction with the graph processing application based on an example query for the exemplary business structure defined in chapter 4. The following sections will use this `QueryContainer` to explain the graph processing application. Additionally, the use of the command pattern enabling the programming interface is described.

5.2.1 Exemplary query

```
QueryContainer<Group, Shop, String, List<String>> shopsWithOneManager =
new QueryContainer<Group, Shop, String, List<String>>(path(Group.class)
.add(Manager.class)
.add(Shop.class))
.addFilter(Shop.class, shop -> shop.getManager().size()==1)
.defineEntityFunction(Shop.class, shop -> shop.getId())
.defineResult(stringStream -> stringStream.collect(Collectors.toList()));
```

Code Snippet 4: A query collecting the identifiers of shops with one manager by traversing the graph from *Group* to *Manager* to *Shop*

Snippet 4 shows the required input for defining a *map-reduce* operation to be executed by the graph processing application. The user needs to define the path of model classes to be traversed trough the graph with a fluent

interface (`path` in line 1). In this case, the application will navigate from a *Group* to *Mangers* to *Shops*. Line 3 specifies a filter for the *Shop*: Only shops which are managed by a single manager must be considered during the traversal. The next line defines what the extraction process for the *Shop*: Since the user is interested in identifiers of shops which have one manager, the application needs to extract identifier of shops. Finally, the creating of the expected result is defined in line 5, which creates a list of all *Shop*-identifiers which where extracted during the traversal.

5.2.2 Command pattern for context specific operations

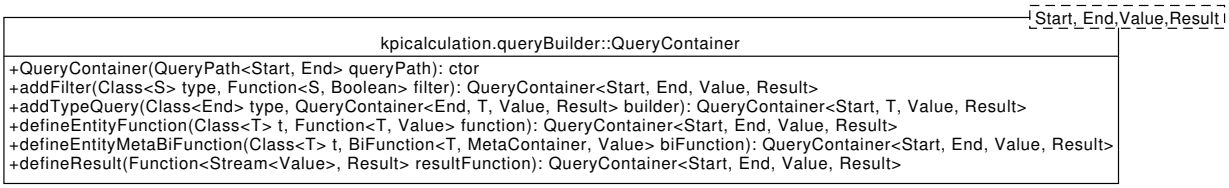


Figure 5.1: Class diagram of the `QueryContainer`

This subsection describes how the application makes use of model classes to provide a user friendly programming interface.

Instantiating model classes with vertex data

Since the graph is traversed based on a path defined by model classes, the class representing the data of any visited vertex is is know. The `DatabaseActor` provides the `WorkerActor` with JavaScript Object Notation (JSON) representation of the vertices, which makes it easy to instantiate the model classes with the data contained in the vertices (see the `instantiate`-method of the `WorkerActor` in Line 200 of Snippet 9). The `QueryContainer` links types of the traversed path with `Functions` for filtering model instances and extracting values from them. Therefor, the data of each vertex can be applied to those `Functions` (See line 239 and line 249 of the `WorkerActor` in Snippet 9).

Extracting typed values from model class instances

Since the application needs to combine multiple extracted values from vertices of the graph, the values to be processed need to be of the same type (It would not make sense to accept strings, lists or json files as parameters for a summation of sales data). The generic *Value* of the `QueryContainer` is used to enforce the type of data extracted from the vertices (see `defineEntityFunction` in Figure 5.1). It also enforce the type of data consumed by the function responsible for creating the `Result` of the query: If the user specifies that a `String` needs to be extracted from traversed entities, only a function which produce `Strings` can be supplied to the `QueryContainer` (See line 3 of Snippet 4). Since the `Result`-generic is specified as `List` of `Strings` (see line 2 of Snippet 4), the `defineResult`-method will only accept a `Function` which consumes a `Stream` of `Strings` and turns it into a list (See line 4 of Snippet 4).

5.3 Using actors to execute `QueryContainers`

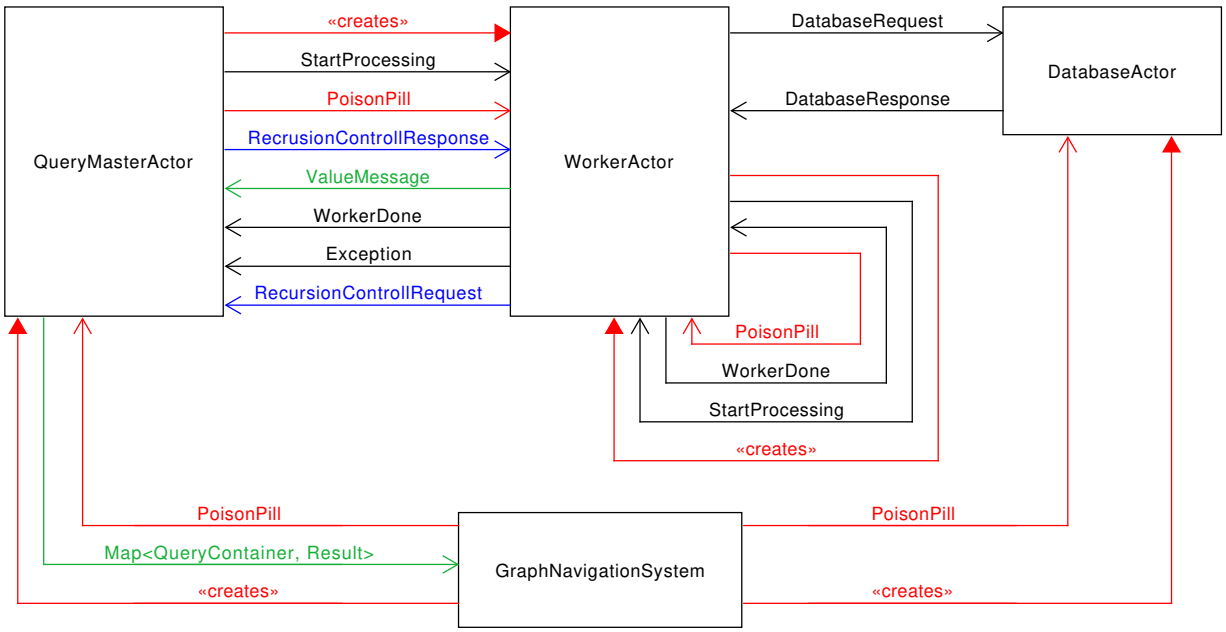


Figure 5.2: Communication diagram of the graph processing application

The graph processing application is designed around **AKKA**, an implementation of the *Actor Model* (Hewitt (2010)) for the Java Virtual Machine (JVM). Since the actor model is a computational model which solves concurrency issues by communication, the interaction between the components is visualized in the communication diagram displayed in Figure 5.2. This diagram will be used to explain the process of executing a `QueryContainer`. The use of the *Actor system* is justified in Appendix C.

5.3.1 GraphNavigationSystem

In order to execute a `QueryContainer`, it needs to be provided with the identifier of the initial vertex to the `GraphNavigationSystem`, which is the central component of the the processing application. On application startup, this component initializes the actor system for executing queries and creates a single `DatabaseActor` for database operations. The main responsibility of this component is starting the user defined queries and returning the result of the query to the user when it has been processed. Therefor, the `GraphNaviagation` creates a new `QueryMasterActor` for each `QueryContainer` and supplies it with a reference to the `DatabaseActor`. The result is returned to the outside-world as a `CompletionStage`.

5.3.2 QueryMasterActor

In order to start the execution of `QueryContainer`, the `QueryMasterActor` creates a `WorkerActor`, providing it with a reference to the `QueryContainer`, a reference to the `Database-Actor` and the identifier of the initial vertex to be processed . Once this initial child actor is created, the `QueryMasterActor` starts the execution of *Map* process by sending a `StartProcessing`-message to the `WorkerActor`. In the progress of the this process the `QueryMasterActor` will receive `ValueMessages` from `WorkerActors` (See Figure 5.2), which contain the user defined values extracted from specific vertices of graph (see Figure 5.2.2). When a `WorkerDone` message is received from the child actor, the *Map*-process is complete and all collected values are applied to the `resultFunction` in order to produce the result expected by the user. When the result is calculated, it is send to the `GraphNavigationSystem` in form of map and the child actor is killed with a `PoisonPill`.

5.3.3 WorkerActor

During the *mapping*-process of executing a query, the application builds a network of `WorkerActors`, which mirrors the structure of tree to be traversed. Due to the complexity of this process, an additional diagram inspired by the sequence diagram shows the interaction between the `WorkerActor` and other components is available in the appendix (Figure C.4).

Requesting data and waiting for processing approval Upon receiving a `StartProcessing` message, the `WorkerActor` sends a `DatabaseRequest`-message to the `DatabaseActor` for extracting the data of its vertex from the database. When it receives the requested data in form of a `DatabaseResponse`, it checks whether the vertex has more than one parent of the previous type in the path of the `QueryBuilder`. If this is the case, a `RecursionControlRequest` is send to the `QueryMasterActor`, which can deny or approve the request (see section 5.4 for more details). Otherwise the actor approves the request itself and start processing the data of its vertex.

Filtering the vertex Upon receiving an approved `RecursionControlResponse`, the `WorkerActor` starts processing the data of its vertex. If the `QueryContainer` contains a query of for the type of the vertex, it instantiates the model class with the data of the vertex and applies to the filter Function. When executing the `QueryContainer` defined in Snippet 4, each `WorkerActor` processing a *Shop* will create a shop instance. If the shop has more than one manager identifier, it passes the filter. If it does not, the processing of the vertex is finished.

Extracting values from the vertex If the filter is passed and the `QueryContainer` contains a mapping function for the type of the vertex, the data of the vertex is used to instantiate the model class and applied to it and the created `Value` is send to the `QueryMasterActor`.

Creating child actors for child vertices In order to traverse the graph among the user provided path, the foreign identifiers pointing towards the next type in the path need to be processed as well. Therefor, the actor extracts the identifiers of the next type in the path from the data of its vertex. If no identifiers are found, the processing is finished. Otherwise, the `WorkerActor` creates a new actor for each of the child vertices and parallely starts their execution with `StartProcessing` messages (See line 321 of Snippet 9). The benefits the parallel execution is clearly visible in performance test discussed in Appendix D, which compares the execution time of sequentially executing the traversal versus doing it in parallel.

Waiting for all children to finish The `WorkerActor` keeps track of the state of its children by keeping references two those actors which have not sent a `WorkerFinished`-message. Upon receiving such a message, the `WorkerActor` kills it child with a `PoisonPill` and removes the reference to that actor. If no actor references remain, the `WorkerActor` knows that all children of its vertex have been processed and sends a `WorkerFinished`-message to its parent, which may be another `WorkerActor` or the `QueryMasterActor`.

5.3.4 DatabaseActor

The `DatabaseActor` is responsible for interacting with the database. In order to assure the integrity of the database connection, only one `DatabaseActor` can exist in the application: If the `WorkerActors` were to be responsible for executing queries them selfs, an indeterminable amount of threads would simultaneously try to use the database connection. By delegating the responsibility to singleton `DatabaseActor`, the database connection is always used once at any given point int time. Since the queries are executed asynchronically, the process of handling a `DatabaseRequest` never blocks and is executed almost instantaneously. Since the there is only once actor using the database connection, it is easy to react to database exception and reschedule failed queries without involving the `WorkerActor`, who remains unaware of the details of the database interaction. For example, it is likely that a database can not keep up with tree traversal process and suffer a queue overflow. When the callback of a query returns such an exception, the query can simply be executed again until it is completed. During this time, the `WorkerActor` sleeps and does not use any threads. The results of successful queries are send to the `WorkerActors` in form of `DatabaseResponses`, which encapsulate the query result as JSON string.

5.4 Recursion control

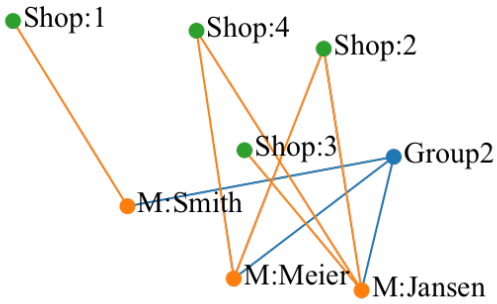


Figure 5.3: Graph structure relevant for *Group2*

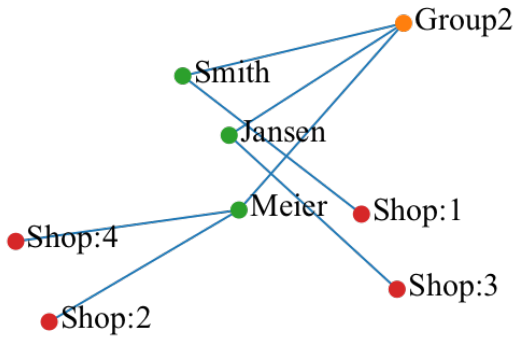


Figure 5.4: Tree structure processed by the graph processing application.

In order to calculate KPIs for any given entity of the business, all data relevant for the calculation needs to be discovered at most once (see Figure 2.2.2) during the calculation process. This issue is visualized in Figure 5.3 and Figure 5.4: Assuming that the data relevant for the calculation of KPIs is contained in the shop vertices, the vertices *Shop:1*,*Shop:2*,*Shop:3* and *Shop:4* need to be considered. Because, *Shop:2* and *Shop:4* are managed by the managers *Meier* and *Jansen* , they will be discovered twice. This invalidates the calculation, because the the values associated with these two shops will also be considered twice. The simplest way of assuring correct calculations is removing duplicates when calculating the result. By using this method, the algorithm can run completely parallelized. However, this also means that many of the extracted values are discarded, because the entire sub-tree below a vertex with more than one parent is processed for each parent. Depending on the size of the sub-tree at such a vertex, this can greatly increase the running time of the data collection process. Another way of assuring that each vertex is only considered once, is to assure that the extraction process of each vertex is only started once. However, this would prevent the graph processing from being executed in parallel, because the information about which vertices have been visited needs to be stored in a centralized and synchronized way to prevent race conditions. In order to minimize the processing time and assure the tree-structure of the processed sub-graph, the application combines the two approaches: Vertices which have more than one parent vertex of the previous type in the query path use recursion control. All other vertices are processed without recursion control, because the number of invocation of the extraction process for any vertex is equal to the number of its parent vertices. This enables the application to process the sub-graph below the vertex *Group2* displayed Figure 5.3 as a the rooted sub-tree displayed in Figure 5.4: Only the edges from the manager *Meier* to the shops *Shop:2* and *Shop4* were processed.

Chapter 6

Quality Control

This chapter briefly describes how the quality of the application developed during this project is assured.

6.1 Unit tests

The graph creation application has been fully unit tested by creating the graph of the example business discussed in chapter 4 and validating the connections created between the vertex types. In order to prove its functionality, multiple graphs have been produced with the application (some are visualized in (see Figure B.1 and Figure B.2) and have been used to correctly calculate KPIs for the business modeled by the graphs. The implementation of the DFS-algorithm (see Snippet 7) has also been fully unit tested. The graph processing application has been tested and is proven to always produce the same output for the same `QueryContainer` and same starting vertex. Since the unit test uses the query described in section 5.4 as test case, in which recursion control is required, the recursion control also proven to work. Additionally, the application has been manually tested by confirming the results of an accumulation of sales based on the input files of the graph.

6.2 Code quality

In order to assure the readability of the code, all methods within the graph creation application and the graph processing application are documented with `JavaDoc` and the more complex operations within the application have additional code comments in order to explain the processes. Especially the `DomainDefinition` has been extensively documented, since numerous complex data structures are processed with interconnected stream operations in order to extract graph structures from input files.

Chapter 7

Conclusion

At the beginning of my thesis I was presented with a simple problem: My employer needed an application for quickly calculating KPIs which was easy to adapt for new customers, because the current version of the software required large amounts of efforts to be adapted for a new customer and took a long time to calculate KPIs. The issues of the software began at modeling the business structures of the often large corporations, which could only be deduced from more or less logical input files which often include small error which make them a nightmare to process. Since there is no way to prevent additional development effort for these issues, I decided to build an abstract application which offers a programming interface for quickly defining multiple logical sets from input files and import them: The `AbstractDataSource` (see subsection 4.3.1). This programming interface powers the graph extraction application, which can generate a complex (and beautiful) graph representation of business structures. With this application, a lot of development effort can be avoided for future version of the KPI Cockpit, because the logic of the application works for all input files. The next major issue to be solved were the performance problems of the KPI Cockpit, which is implemented in PHP using a Structured Query Language (SQL) database and is based on code developed during the past 10 years. Because of the complexity of the code base, I decided against analyzing the current version of the software in any depth in order to focus on building an improved version. The resulting application can process 10000 vertices in 500 milliseconds and can be sped up even further by increasing the throughput of the database. Due to the complexity of the two application, the deployment of the application was not tackled. However, the graph processing application is designed to be deployed as a distributed application. traperto is pleased with the speed of the application is especially interested in the graph representation of their customers.

Bibliography

Codd, E. F. (1970), 'A relational model of data for large shared data banks', *Communications of the acm* .

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009), *Introduction to Algorithms, Third Edition*, 3rd edn, The MIT Press.

De Domenico, M., Solé-Ribalta, A., Cozzo, E., Kivelä, M., Moreno, Y., Porter, M. A., Gómez, S. & Arenas, A. (2013), 'Mathematical formulation of multilayer networks', *Physical Review X* **3**(4).

Dean, J. & Ghemawat, S. (2004), Mapreduce: Simplified data processing on large clusters, *in* 'Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6', OSDI'04, USENIX Association, Berkeley, CA, USA, pp. 10–10.
URL: *http://dl.acm.org/citation.cfm?id=1251254.1251264*

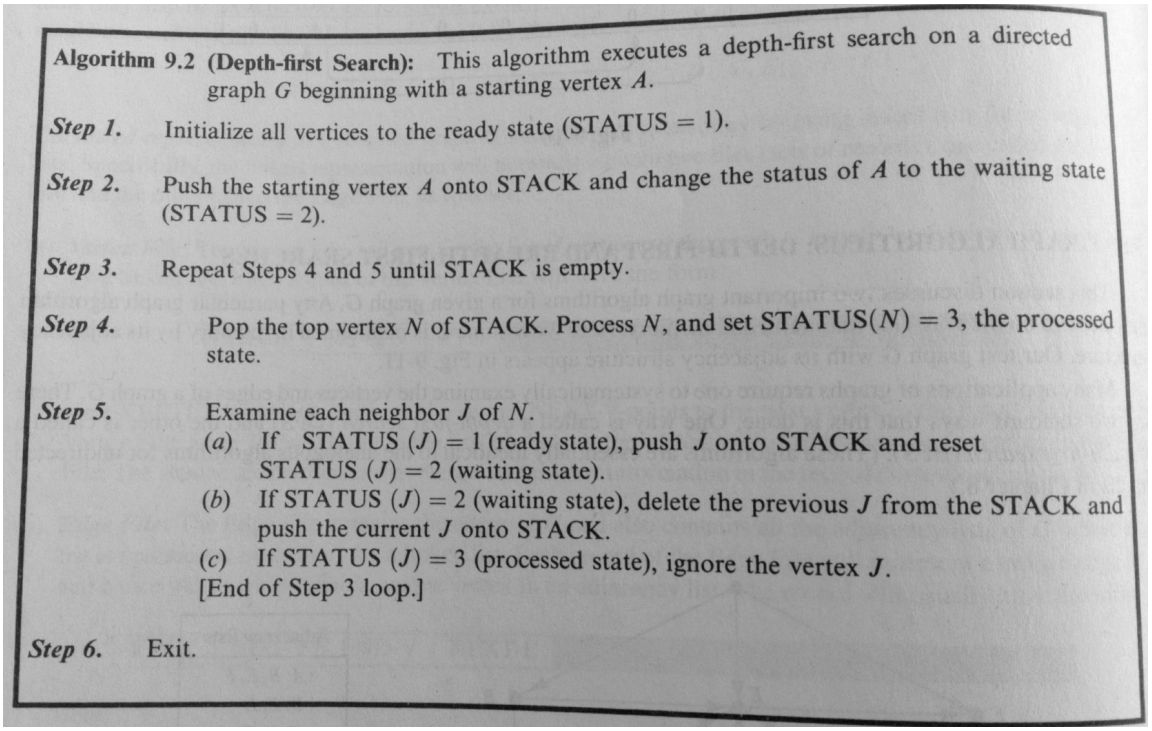
Hewitt, C. (2010), 'Actor model for discretionary, adaptive concurrency', *CoRR* **abs/1008.1459**.
URL: *http://arxiv.org/abs/1008.1459*

Lipschutz, S. & Lipson, M. (2010), *Schaum's Outlines of Theory and Problems of Discrete Mathematics Third Edition*, McGraw-Hill.

Appendix A

Graph algorithms

A.1 Depth-first search



Algorithm 9.2 (Depth-first Search): This algorithm executes a depth-first search on a directed graph G beginning with a starting vertex A.

Step 1. Initialize all vertices to the ready state (STATUS = 1).

Step 2. Push the starting vertex A onto STACK and change the status of A to the waiting state (STATUS = 2).

Step 3. Repeat Steps 4 and 5 until STACK is empty.

Step 4. Pop the top vertex N of STACK. Process N, and set STATUS(N) = 3, the processed state.

Step 5. Examine each neighbor J of N.
(a) If STATUS (J) = 1 (ready state), push J onto STACK and reset STATUS (J) = 2 (waiting state).
(b) If STATUS (J) = 2 (waiting state), delete the previous J from the STACK and push the current J onto STACK.
(c) If STATUS (J) = 3 (processed state), ignore the vertex J.
[End of Step 3 loop.]

Step 6. Exit.

Figure A.1: The depth first algorithm. Taken from Lipschutz & Lipson (2010)

A.2 Topological sorting

Algorithm 9.4: The algorithm finds a topological sort T of a directed cycle-free graph S.

Step 1. Find the indegree INDEG(N) of each vertex N of S.

Step 2. Insert in QUEUE all vertices with zero degree.

Step 3. Repeat Steps 4 and 5 until QUEUE is empty.

Step 4. Remove and process the front vertex N of QUEUE.

Step 5. Repeat for each neighbor M of the vertex N.
(a) Set INDEG(M) : = INDEG(M) – 1.
[This deletes the edge from N to M.]
(b) If INDEG(M) = 0, add M to QUEUE.
[End of loop.]
[End of Step 3 loop.]

Step 6. Exit.

Figure A.2: Algorithm for finding a topological sort of a graph. Taken from Lipschutz & Lipson (2010)

Appendix B

Graphs produced with the graph extractor

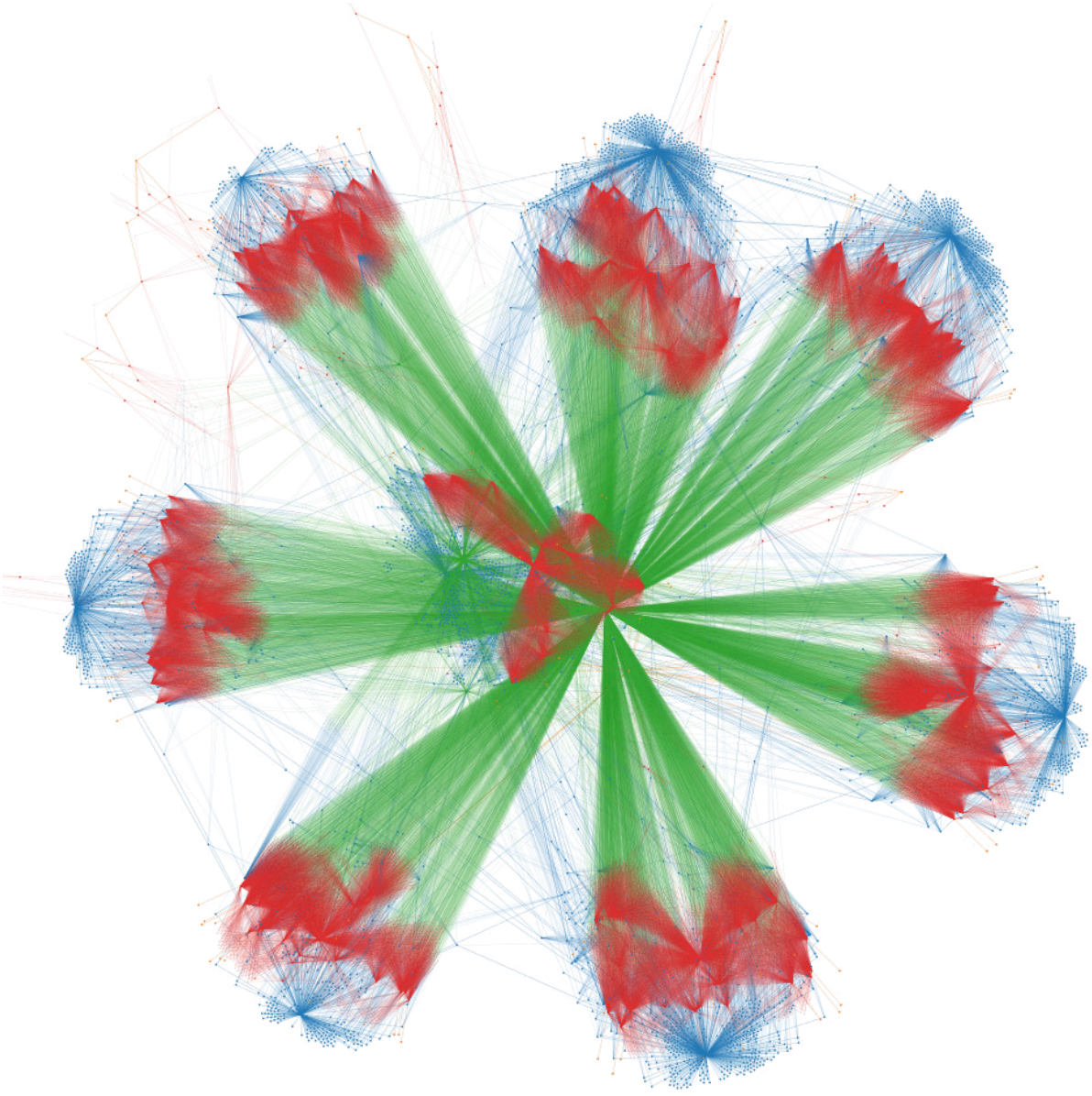


Figure B.1: A graph consisting of 21729 vertecies and 106299 edges

.

In Figure B.1, green edges show associations separate the shops into types, red elements represent managers and the shops they are responsible for, blue elements separate the business into regions and cities and associate them with shops, orange elements represent group of managers. Generated with D3js.

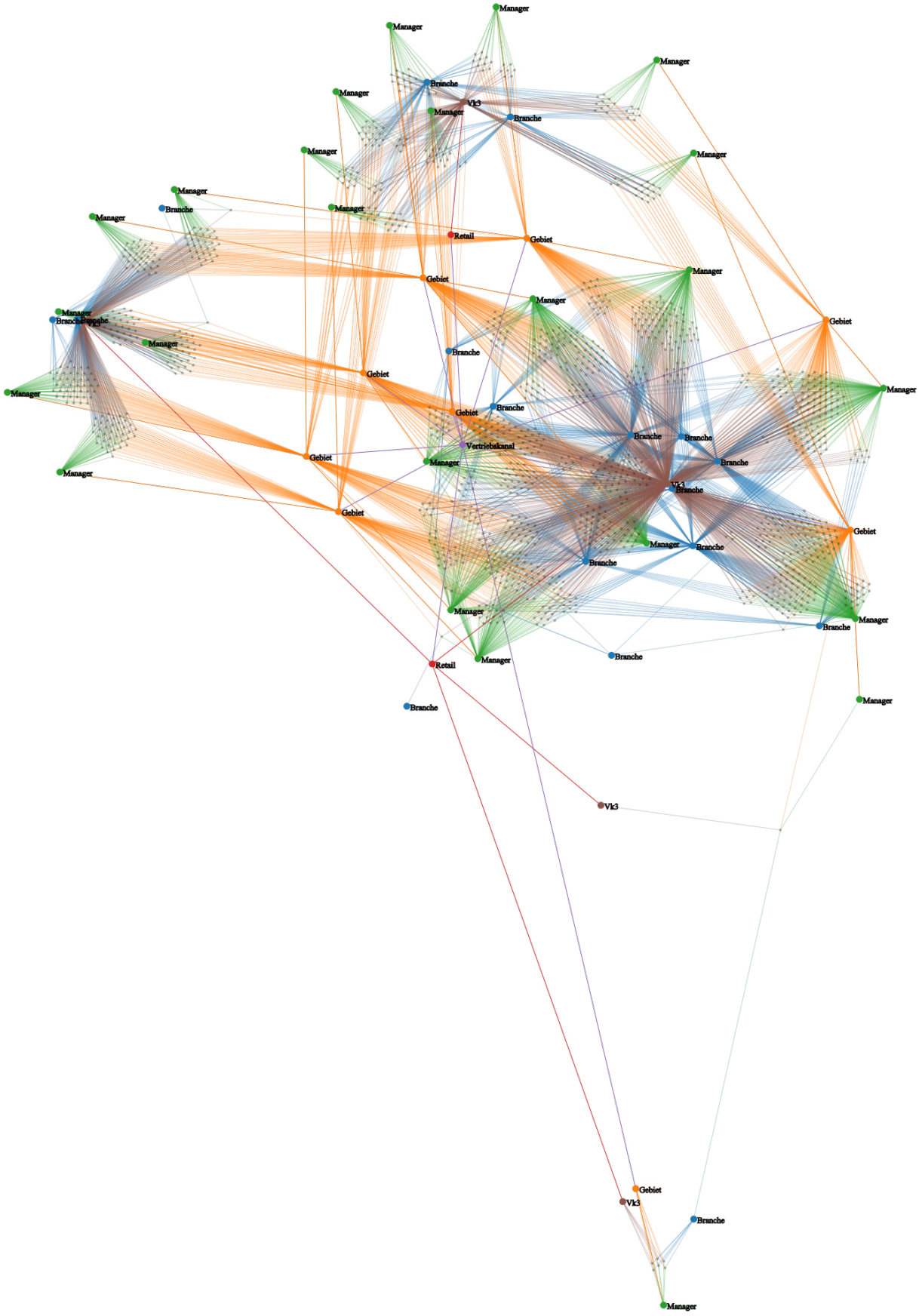


Figure B.2: A graph consisting of 2737 vertecies and 10713 edges

.

In Figure B.2 the small black dots represent shops which are managed by mangers (green elements). Both shops and managers are associated to areas (orange). Blue elements associate into retailers or shop types. Purple and red elements separate the business into retail types (e.g e-commerce, normal retail, chains). Generated with D3js.

Appendix C

Using the actor model

This chapter justifies the use of the actor system for the graph processing application in order to parallelize BFS-algorithm for the required tree traversal. First, the traversal is described when using a recursive and sequentially executed BFS

C.1 Sequential traversal

The key issue of parallelizing the tree traversal within the graph structures is maintaining the state of each invocation of the traversal. When executing the traversal sequentially, one can rely on the call stack to handle the correct ordering of recursive executions of the tree traversal.

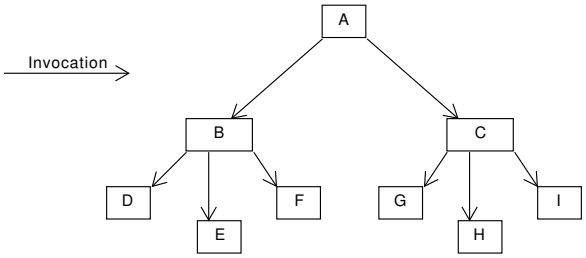


Figure C.1: An example tree to be traversed with the BFS-algorithm.

Vertices are visited in alphabetical order

For example, in the tree displayed in Figure C.1, the BFS starts at vertex A and recursively invokes the traversal for B and C. Next, D, E and F are visited from B. After the children of B are visited, the sub-tree below C is discovered. Next text traversal is started for the vertices D, E and F. Since they have no child vertices, the traversal stops at these vertices. Therefor, the traversal of the the sub-tree below B is completed. The same happens below the vertex C: The traversal is started recursively for the vertices G, H and I. Since none of them have child vertices for which another recursion is invoked, all vertices of the sub-tree below C have been visited. Now all recursive invocation which could have been started by the children of A are completed and the traversal of the tree starting a this vertex is finished.

C.2 Parallel traversal

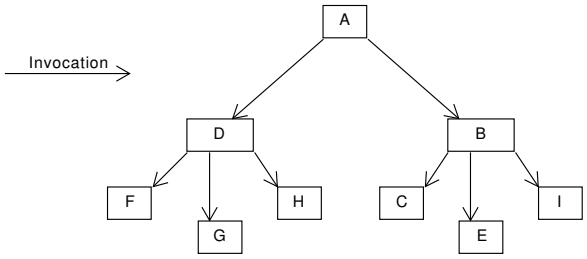


Figure C.2: Calculating KPIs from children.

Vertices are visited in alphabetical order

The BFS-algorithm can easily be parallelized by starting each recursive invocation of the traversal process in parallel. However, this means that the call stack can no longer be used to maintain the state of the traversal, because each parallel invocation of the traversal has its own call stack. This means that the traversals finish in a indeterminable order: In the example given in img:bfs2, the vertex C has been processed before vertex D, besides the fact that D is the child of the initial vertex. Since, traversals finish in random order and the application needs to update the state of the parent traversal when a child traversal has completed. The easiest way of maintaining the state of the traversal invocations is to recreate the structure of the tree to traversed with the states of the invocation and their dependency to states of other invocations.

C.3 Using the actor model to keep track of a decentralized state

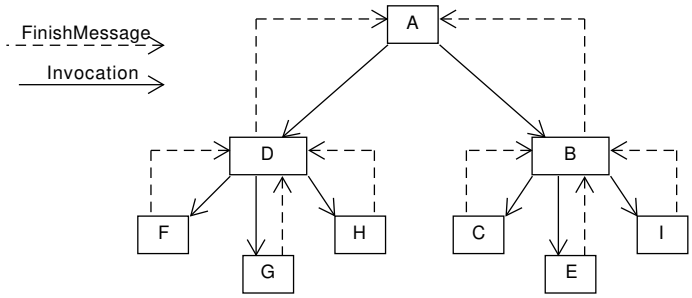


Figure C.3: Calculating KPIs from children.

Vertices are visited in alphabetical order

The structure of maintaining the decentralized state of multiple processes can be model with the `Actor model` (Hewitt (2010)), which solves concurrency issues with *stateful*-actors which run in parallel and communicate using messages. By creating a new actor for each invocation of the BFS traversal, the state of each traversal can be tracked independently and communicated to the traversal it was started by. This is visualized in Figure C.2: Each vertex is aware of the traversals it started and only changes its state to finished when all of its child processes have finished. This means that traversal can be started in random order while being able to assure that each vertex is processed correctly.

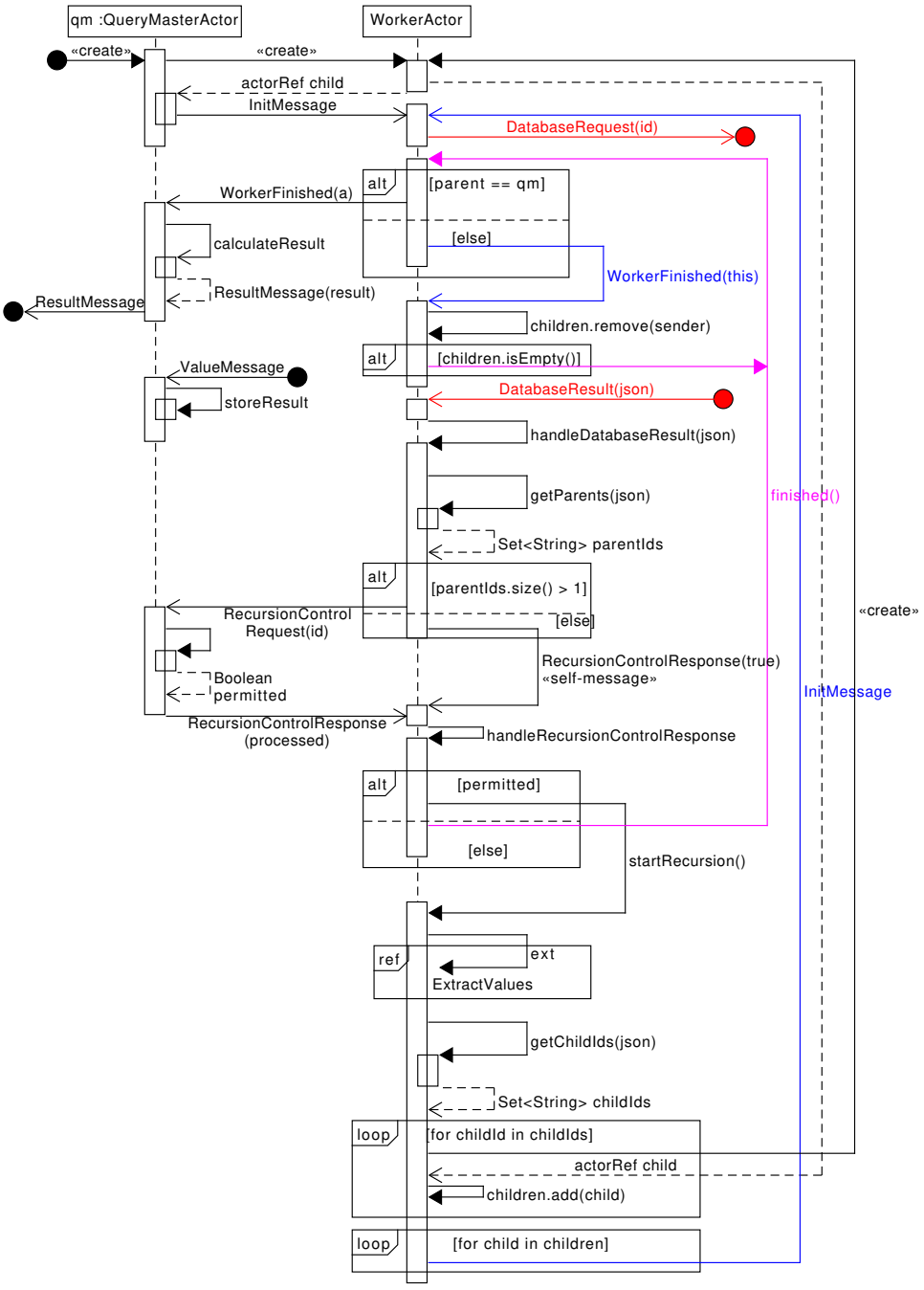


Figure C.4: Diagram visualizing the processes in the `WorkerActor`

.

Appendix D

Performance comparison between sequential DFS and parallel BFS

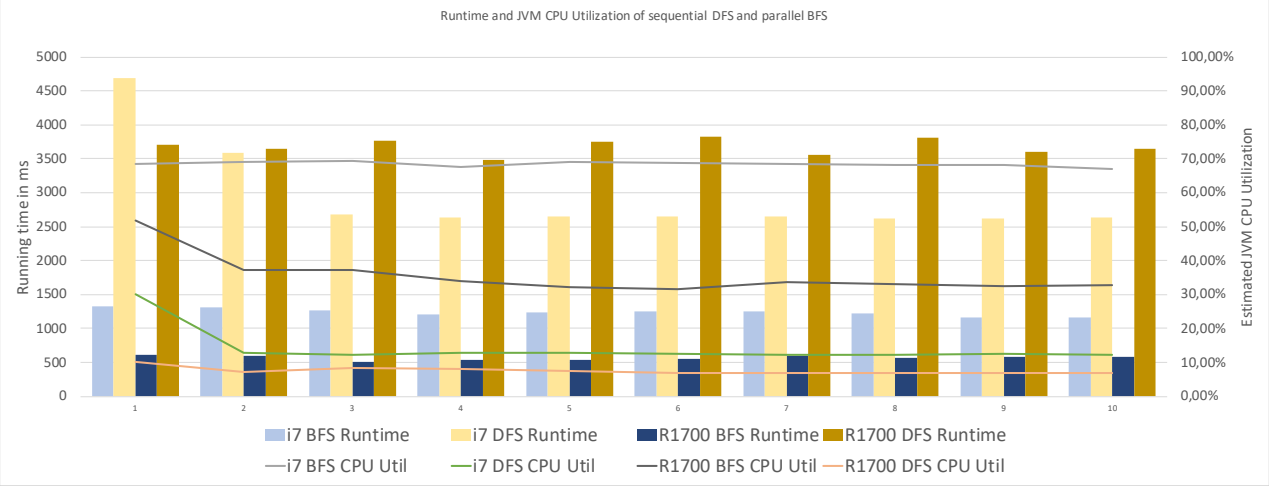


Figure D.1: Performance measurements of DFS and parallel BFS

Figure D.1 displays the performance of the sequential DFS-algorithm and a parallelized version of the BFS-algorithm for processing 10798 vertices for calculation 14 KPIs for the highest point of the business structure graph displayed in Figure B.2. In both cases, the graph is stored in a *MongoDB*-database on the Solid State Drives of the system, which needs to be considered when interpreting the results of the performance test. The used *Intel Core i7* has 4 cores running at 2.5 gigahertz (which can be boosted to 4 gigahertz) which can run 8 threads in parallel. The *AMD Ryzen R7 1700* has 8 cores running at 3 gigahertz (which can be boosted to 3.7 gigahertz) which can run 16 threads in parallel. Both system have 16 gigabytes of memory. The graph shows the first 10 executions of the same query after the initialization of the graph processing application is completed. The reason for the higher Central Processing Unit (CPU) usage at beginning of test is the initialization of a pool 100 connections to the database. Because of this, the measurements of the first execution are ignored. The estimated JVM CPU utilization shows that the DFS-algorithm is executed sequentially and uses a single thread to process the graph (Roughly 13% CPU utilization of the i7 and about 7% utilization of the r1700). The BFS-algorithm, which can traverses the graph in parallel, clearly out-performs the DFS-algorithm: With th i7, which has better single-core performance, the average performance increase factor is 2.6 when using BFS. On the r1700 this factor is 6.9. The CPU utilization of the JVM of 61% is the result of the database running on the same system, since the estimated CPU utilization of the system is 99% during the test. It is likely that the application would see a performance increase on the i7 system, if the database was located on a different machine. On the r7 system however, the estimated CPU utilization of the system remains at about 50%. The most likely reason for this is the database performance, which is limited by a connection pool of

a 100 connections. Therefor it is likely that the performance of the BFS-algorithm could be increased even further on r1700 system by increasing the throughput of the underlying database system.

Appendix E

Code snippets

```
        /*
        ------1.-------
        Define data to be imported from all files.
         */
        //Note that index 3 of file1 is ignored.
        Path filePath = Paths.get("testInput1.txt");
        AbstractDataSource file1 = new CSVDataSource(filePath)
                .addField("ShopID", 0)
                .addField("City",1)
                .addField("Country", 2)
                .addField("Continent", 4)
                .addField("Type",5);

        Path filePath2 = Paths.get("testInput2.txt");
        AbstractDataSource file2 = new CSVDataSource(filePath2)
                .addField("ManagerID", 0)
                .addField("Group",1)
                .addField("ManagerName", 2);

        Path filePath3 = Paths.get("testInput3.txt");
        AbstractDataSource file3 = new CSVDataSource(filePath3)
                .addField("ShopID", 0)
                .addField("ManagerID",1);
        /*
        ------2.-------
        Define domain types based on input data
         */
        DomainType shop = TypeBuilder.createType("Shop").primaryId("ShopID",file1,
        ↪  file3).build();
        DomainType type = TypeBuilder.createType("Type").primaryId("Type", file1).build();
        //Multiple dataSources for ManagerID since the manager is defined in file2 and file3.
        DomainType manager = TypeBuilder.createType("Manager")
                .primaryId("ManagerID",file2,  file3)
                .addProperty("ManagerName", String.class,
                stringStream -> stringStream.findFirst().get(), file2).build();
        DomainType city = TypeBuilder.createType( "City")
                .primaryId("City", file1).build();
        DomainType group = TypeBuilder.createType("Group")
                .primaryId("Group", file2).build();
        DomainType country  = TypeBuilder.createType("Country")
                .primaryId("Country",file1).build();
```

```
        DomainType continent =  TypeBuilder.createType("Continent")
                .primaryId("Continent",file1).build();
        ------3.-------
        Define graph structure.
         */
        DomainDefinition definition = new DomainDefinition().
                linkTypes(type,shop).
                linkTypes(group,person).
                linkTypes(person,shop).
                linkTypes(continent,country,city,shop);
        /*
        ------4.-------
        Extract graph based on DomainDefinition and all associated DataSources.
         */
        DomainGraph graph = definition.createGraph();
        /*
        Use data. In this case: Create a json containing all entities and
        edges for a three dimensional graph representation.
         */
        AbstractVertexD3Creator jsonCreator  = new D33DJsonCreator(graph);
        jsonCreator.createJson("data.json");
```

Code Snippet 5: Code snippet for creating a graph representation for the case discussed in chapter 4

```
public class CSVDataSource extends AbstractDataSource<Integer,String>{

    private final Path filePath;

    public CSVDataSource(FieldNameMapper fieldNameMapper, Path filePath) {
        super(fieldNameMapper);
        this.filePath = filePath;
    }

    @Override
    protected Map<Integer, String> convertToMap(String sourceRow) {
        String[] split = sourceRow.split(",");
        Map<Integer, String> rowMap = new HashMap<>();
        for (int i = 0; i < split.length; i++) {
            String s = split[i];
            rowMap.put(i , s);
        }
        return rowMap;

    }

    @Override
    protected Stream<String> getSourceStream() {
        try {
            return Files.lines(filePath, Charset.forName("UTF-8"));
        } catch (IOException e) {
            e.printStackTrace();
        }
        throw new IllegalArgumentException("Could not open the file "+filePath.toString());
    }

    @Override
```

```
33      protected String onEmptyField(String fieldName) { return "-";}
34  }
```

Code Snippet 6: Code snippet of the `CSVDataSource`. Some methods omitted.

E.1 Depth-first-search algorithm

```
1   public class DepthFirstSearch<VertexType> {
2       /**
3        * Container containing the graph.
4        */
5       private final GraphContainer<VertexType> container;
6       /**
7        * Stores which vertecies have been visited by DFS.
8        */
9       private Set<VertexType> visited = new HashSet<>();
10      /**
11       * Stores the predecessor for each vertex from which parent vertex it was visited.
12       * (Some vetertecies may have multiple parents. DPS processes each vertex once.
13       This stores from which of the parents the vertex was visited).
14       */
15      private Map<VertexType, VertexType> predecessorMap = new HashMap<>();
16      /**
17       * Stores the timestamp on which each vertex was visited.
18       */
19      private Map<VertexType, Integer> startTimes = new HashMap<>();
20      /**
21       * Stores the timestamp on which the DPS-tree was finished for each vertex.
22       */
23      private Map<VertexType, Integer> finishTimes = new HashMap<>();
24      /**
25       * Stores the current timestamp.
26       */
27      private int time = 0;
28      public DepthFirstSearch(GraphContainer<VertexType> container) {
29          this.container = container;
30      }
31      public Map<GraphEdge<VertexType>, EdgeType> classifyEdges(){
32          /*
33          Starts the execution for each vertex which has not been visited before.
34          The check is required, because a vertex may have been visited by a previous invocation
35          of the recursion method.
36           */
37          container.getVertexMap().values().stream().forEach(vertexTypeVertexAdapter -> {
38              if(!visited.contains(vertexTypeVertexAdapter)){
39                  recursion(vertexTypeVertexAdapter, null);
40              }
41          });
42          /*
43          Classify all edges of in the graph by comparing the start and finish times
44          of all parent vertices to their children.
45           */
46          return container.getVertexMap().values().stream()
47          .map(vertex -> container.getChildren(vertex).stream()
48          .map(child -> {
```

```
            int parentStart = startTimes.get(vertex);
            int parentFinish = finishTimes.get(vertex);
            int childStart = startTimes.get(child);
            int childFinish = finishTimes.get(child);

            EdgeType edgeType = EdgeType.CROSS_EDGE;
            if (parentStart < childStart && parentFinish > childFinish) {

                if (predecessorMap.get(child).equals(vertex)) {
                    edgeType = EdgeType.TREE_EDGE;
                } else {
                    edgeType = EdgeType.FORWARD_EDGE;
                }
            }
            if (parentStart > childStart && parentFinish < childFinish) {
                edgeType = EdgeType.BACK_EDGE;
            }

            return Map.entry(new GraphEdge<>(vertex, child), edgeType);
        })).flatMap(classifiedEdgeStream -> classifiedEdgeStream)
        .collect(Collectors.toMap(o -> o.getKey(), o -> o.getValue()));
    }

    public void recursion(VertexType vertex, VertexType parent) {
        time++;
        startTimes.put(vertex,time);
        visited.add(vertex);
        Set<VertexType> children = container.getChildren(vertex);
        for(VertexType childVertex : children){

            if(!visited.contains(childVertex)){
                predecessorMap.put(childVertex, vertex);
                recursion(childVertex, vertex);
            }
        }
        finishTimes.put(vertex, time);
        time++;
    }
}
```

Code Snippet 7: Implementation of the *Depth-First-Search*-algorithm for edge classification

E.2 Graph Extraction application

```
public class DomainDefinition {
    private Set<DomainType> domainTypes = new HashSet<>();
    /**
     * @param types
     * @return
     */

    public DomainDefinition linkTypesRequiredConncetion(DomainType... types){
        return innerLinkTypes(true,types);

    }
```

```
12
13      public DomainDefinition linkTypes(DomainType... types) {
14          return innerLinkTypes(false,types);
15      }
16
17
18      private DomainDefinition innerLinkTypes( boolean required,DomainType... types){
19                  /*
20             The target DomainType needs to contain the identifying field(s) of the source DomainType.
21              This makes it possible to establish which instances of the target-DomainType are
    ↪   connected to
22              the source-DomainType:
23              source.UniqueName (id) = ABC,
24              targetA.UniqueName (attribute) = ABC,
25              targetB.UniqueName (attribute) = ABC,
26              targetC.UniqueName (attribute) = XYZ,
27              ...
28               */
29              Arrays.stream(types).forEach(type -> domainTypes.add(type));
30
31              for (int i = 1; i < types.length; i++) {
32                  types[i - 1].addChildType(types[i], required);
33              }
34              return this;
35      }
36
37
38      /*
39      Checks whether the graph structure is valid.
40       */
41      protected void validateGraph() {
42          GraphContainer<DomainType> graphContainer = new DomainTypeVertexAdapterFactory().
43                  createGraphAlgorithmContainer(getAllEntityDefinitions());
44          AbstractVertexD3Creator error = new DPSD3Creator(graphContainer);
45          error.createJson("test.json");
46
47          DepthFirstSearch<DomainType> domainTypeDepthFirstSearch = new
            ↪  DepthFirstSearch<>(graphContainer);
48
49          Map<GraphEdge<DomainType>, EdgeType> graphEdgeEdgeTypeMap =
            ↪  domainTypeDepthFirstSearch.classifyEdges();
50
51
52          if (graphEdgeEdgeTypeMap.values().contains(EdgeType.BACK_EDGE)) {
53              throw new IllegalArgumentException("The graph contains back edges: " +
                ↪  graphEdgeEdgeTypeMap);
54          }
55
56      }
57
58      /**
59       * Creates a graph based on the DomainTypes.
60       * @return
61       */
62      public DomainGraph createGraph() {
63
```

```
64          validateGraph();
65
66          prepareDataSource();
67
68          /*
69          Extract entities from the data sources.
70           */
71          Map<DomainType, Map<String, TypeVertex>> entityDefinitionListMap = GetEntities();
72
73
74          return new DomainGraph(this, entityDefinitionListMap);
75      }
76
77
78      /**
79       * Prepares the data sources for extraction by setting the fields from the
80       * sources which need to be extracted.
81       */
82      protected void prepareDataSource() {
83          Map<AbstractDataSource, Set<DomainType>> dataSourceToDomainTypesMap =
            ↪  getDataSourceToDomainTypesMap();
84          dataSourceToDomainTypesMap.keySet().parallelStream().forEach(dataSource -> {
85              Set<String> values = dataSourceToDomainTypesMap.get(dataSource).stream()
86                      .map(entityDefinition -> entityDefinition.getSourceFieldNames())
87                      .flatMap(strings -> strings.stream()).collect(Collectors.toSet());
88
89              dataSource.setRelevantFieldNames(values);
90
91          });
92      }
93
94      public Set<DomainType> getAllEntityDefinitions() {
95          return domainTypes;
96      }
97
98      /**
99       * Returns a map mapping each relevant data source to all domain types extracted from it.
100      *
101      * @return
102      */
103     protected Map<AbstractDataSource, Set<DomainType>> getDataSourceToDomainTypesMap() {
104         return getAllEntityDefinitions().stream()
105                 .map(type -> type.getDataSources().stream()
106                         .map(source -> new AbstractMap.SimpleEntry<>(source, type)))
107                 .flatMap(entryStream -> entryStream)
108                 .collect(Collectors.groupingBy(o -> o.getKey(),
109                         Collectors.mapping(o -> o.getValue(), Collectors.toSet())));
110     }
111
112     /**
113      * Extracts all tuples of the normalization relations for each DomainType from all
        ↪  datasources.
114      * The maps represent tuples of each relation, the set contains all relations defining the
        ↪  DomainType.
115      *
116      * @param dataSources
```

```
117         * @return
118         */
119        protected Map<DomainType, Set<Map<String, String>>>
           ↪  extractDomainTypeData(Map<AbstractDataSource,
120                Set<DomainType>> dataSources) {
121            /**
122             * Extract unique data sets of each DomainType from all datasources.
123             */
124            Stream<Stream<Stream<AbstractMap.SimpleEntry<DomainType, Map<String, String>>>>> data =
125                    dataSources.entrySet().parallelStream().map(sourceToTypes -> {
126                Stream<Map<String, String>> dataStream = sourceToTypes.getKey().getDataStream();
127                return dataStream.map(dataRow -> {
128                            /*
129                            Per line for each DomainType to be extracted:
130                             */
131                    return sourceToTypes.getValue().parallelStream().map(entityDefinition -> {
132                        /*
133                         * Read the fields of to the domaintype to be read from the datasource from
    ↪  the current data row.
134                         * Return <EntiyDefinition, Map<String,String> mapping the definition to
135                         * a map of entity data.
136                         */
137                        Map<String, String> entityData =
                           ↪  entityDefinition.fieldToReadFrom(sourceToTypes.getKey())
138                        .stream().map(entityFieldName -> dataRow.entrySet().stream()
139                        .filter(dataFieldName ->dataFieldName.getKey().equals(entityFieldName))
140                                        .findFirst())
141                                .filter(stringStringEntry -> stringStringEntry.isPresent())
142                                .map(dataFieldNameOptional -> dataFieldNameOptional.get())
143                                .collect(Collectors.toMap(o -> o.getKey(), o -> o.getValue()));
144                        if (entityData.containsKey(entityDefinition.getIdField())) {
145                            return new AbstractMap.SimpleEntry<>(entityDefinition, entityData);
146                        }
147                        return null;
148                    }).filter(domainTypeMapEntry -> domainTypeMapEntry != null);
149                });
150            });
151            /*
152            Flatten the stream into on stream map entries
153            */
154            return data.flatMap(keyValuePairStream -> keyValuePairStream.flatMap(simpleEntryStream
               ↪  -> simpleEntryStream))
155                    .filter(entityDefinitionMapKeyValuePair ->
156                            entityDefinitionMapKeyValuePair != null &&
157                                    /*
158                                    If there is only one entry in the data set, it can only be an
    ↪  identifier
159                                    which is worthless on its own.
160                                     */
161                                    entityDefinitionMapKeyValuePair.getValue().size() > 1)
162                    /*
163                    Make entries unique -> key to succsess!
164                     */
165                    .distinct()
166                    /*
```

```
167                 Group the entries based on their DomainType and put the relations of the type
    ↪  into a set.
168                  */
169                 .collect(Collectors.groupingByConcurrent(o -> o.getKey(),
170                         Collectors.mapping(o -> o.getValue(), Collectors.toSet())));
171
172     }
173     /**
174      * Joins the extracted dataset on the primary id of their type.
175      * T
176      *
177      * @param rawTypeData
178      * @return
179      */
180     protected Map<DomainType, Set<Map<String, Set<String>>>> joinDataSetsOnID(Map<DomainType,
181             Set<Map<String, String>>> rawTypeData) {
182         /*
183         Combines raw dataset into List based on each id of each domainType.
184
185         e.g typeA = {
186             {tid: 1,val1 : A, val2:B, val3:C},
187                 {tid: 2,val1 : W, val2:X, val3:Y},
188                 {tid: 1,val4: Z}, {tid: 1,val4: D}
189                 }
190         ---> typeA : { [1] : {{tid: 1,val1 : A, val2:B, val3:C},
191                     {tid: 1,val4: D}, {tid: 1,val4: Z},
192                        [2] : {{tid: 2,val1 : W, val2:X, val3:Y}}
193          */
194         Map<DomainType, Map<Set<String>, List<Map<String, String>>>> entityData =
    ↪  rawTypeData.entrySet().parallelStream()
195                 .map(entityDefinitionSetEntry -> {
196
197
198                     Map<Set<String>, List<Map<String, String>>> collect1 =
    ↪  entityDefinitionSetEntry.getValue()
199                             .parallelStream().collect(Collectors.groupingByConcurrent(o ->
200                                     entityDefinitionSetEntry.getKey().getIdFields().stream()
201                                             .map(idField -> o.get(idField))
202                                             .collect(Collectors.toSet())));
203
204                     return new AbstractMap.SimpleEntry<>(entityDefinitionSetEntry.getKey(),
    ↪  collect1);
205
206
207                 }).distinct().collect(Collectors.toMap(o -> o.getKey(), o -> o.getValue()));
208
209         /*
210         Merge the previously combined set containing the same ids
211         { [1] : {{tid: 1,val1 : A, val2:B, val3:C}, {tid: 1,val4: D}, ,
212           [2] : {{tid: 2,val1 : ,W val2:X, val3:Y}, tid: 1,val4: Z}}
213         -->   {tid: [1],val1 : [A], val2:[B], val3:[C],val4: [C,D]},
214               {tid: [2],val1 : [W], val2:[X], val3:[Y], val4: []}
215
216          */
217         return entityData.entrySet().stream().map(domainTypeMapEntry ->
    ↪  domainTypeMapEntry.getValue().entrySet()
```

```
218                 .parallelStream().map(entityDescriptor -> {
219
220                     Map<String, Set<String>> collect =
                        ↪  entityDescriptor.getValue().parallelStream()
221                             .flatMap(entityProperties -> entityProperties.entrySet().stream())
222                             .collect(Collectors.groupingBy(o -> o.getKey(),
223                                     Collectors.mapping(o -> o.getValue(), Collectors.toSet())));
224
225             /*
226             Make sure that all properties are contained!
227             Otherwise ignore the data set.
228              */
229                     boolean propertyMissing = domainTypeMapEntry.getKey().getPropertyFields()
230                             .stream().anyMatch(property -> !collect.containsKey(property));
231
232                     if (!propertyMissing) {
233                         return new AbstractMap.SimpleEntry<>(domainTypeMapEntry.getKey(),
                            ↪  collect);
234                     }
235                     System.out.println("Ignoring: " + domainTypeMapEntry.getKey()+" - "
236                             +domainTypeMapEntry.getKey().getPropertyFields() +" ---" + "-" +
                             ↪  collect);
237                     return null;
238
239                 })).flatMap(entryStream -> entryStream).filter(domainTypeMapEntry ->
                    ↪  domainTypeMapEntry != null)
240                 .collect(Collectors.groupingBy(o -> o.getKey(),
241                         Collectors.mapping(o -> o.getValue(), Collectors.toSet())));
242
243
244     }
245     /**
246      * Creates TypeVerticies from the each Map contained contained within the set of each
    ↪  DomainType.
247      * @param instanceData
248      * @return
249      */
250     private Map<DomainType, Map<String, TypeVertex>> createTypeVertecies(Map<DomainType,
251             Set<Map<String, Set<String>>>> instanceData) {
252         return instanceData.entrySet().parallelStream().map(domainTypeSetEntry ->
253                 domainTypeSetEntry.getValue().stream().map(stringSetMap ->
254                 {
255                     try {
256                         return new TypeVertex(domainTypeSetEntry.getKey(), stringSetMap);
257                     } catch (TypeVertex.InvalidDataSetException e) {
258                         System.err.println("Could not create entity for data set :"+e);
259                     }
260                     return null;
261                 }))
262                 .flatMap(typeVertexStream -> typeVertexStream)
263                 .filter(typeVertex -> typeVertex != null)
264                 .collect(Collectors.groupingBy(o -> o.getDomainType(), Collectors.toMap(o ->
                    ↪  o.getIdentifier(), o -> o)));
265     }
266
267
```

```
268     /**
269      * Extracts unique entries from the data source based on the identifier of
270      * each entityDefinition.
271      *
272      * @return
273      */
274     private Map<DomainType, Map<String, TypeVertex>> GetEntities() {
275         Map<AbstractDataSource, Set<DomainType>> dataSources = getDataSourceToDomainTypesMap();
276         long t = System.currentTimeMillis();
277         Map<DomainType, Set<Map<String, String>>> domainTypeData =
            ↪  extractDomainTypeData(dataSources);
278         Map<DomainType, Set<Map<String, Set<String>>>> joinedDomainTypeDataSets =
            ↪  joinDataSetsOnID(domainTypeData);
279         Map<DomainType, Map<String, TypeVertex>> entities =
            ↪  createTypeVertecies(joinedDomainTypeDataSets);
280
281         /**
282          * Connect instances.
283          * Note: It may occure that specific entities have not been created because their data
    ↪  was ignored at input level.
284          * E.g if rows defining shops are ignored if the shops are "closed", foreign keys
    ↪  referencing this shop can not be
285          * resolved.
286          */
287         entities.entrySet().stream().forEach(entityDefinitionMapEntry -> {
288
289         entityDefinitionMapEntry.getValue().entrySet()
290         .parallelStream().forEach(instanceEntry -> {
291
292         instanceEntry.getValue().getForeignIds().entrySet()
293           .stream().forEach(foreignKeyEntry -> {
294           foreignKeyEntry.getValue().stream().forEach(foreignKey -> {
295              TypeVertex connected = entities.get(foreignKeyEntry.getKey()).get(foreignKey);
296              if (connected != null) {
297                 connected.addConnectedInstance(instanceEntry.getValue());
298                 instanceEntry.getValue().addConnectedInstance(connected);
299               } else {
300                 System.err.println(" Failed to find " + foreignKeyEntry.getKey() + "--->" +
                    ↪  foreignKey);
301               }
302                });
303                });
304             });
305         });
306         long t1 = System.currentTimeMillis();
307         System.out.println("Graph creation finished:  " + (t1 - t));
308         return entities;
309     }
310
311
312 }
```

Code Snippet 8: Source-code of the DomainDefinition class.

Snippet **??** shows the source code of the `DomainDefinition` which is responsible for extracting graphs from source files based on the meta information provided by the users. Since this is the core part of the graph extraction application, it has been extensively tested (95% test coverage). Since it makes use of Java's parallel

streams for transforming the input files to vertexes representing business entities, the application scales well with additional CPU-cores (Extracting 18772 vertecies and 342578 edges between them takes about 8 seconds on an Intel Core i7 with 2.5ghz and 4 cores).

E.3 `WorkerActor` implementation

```
1  public class WorkerActor<Value> extends AbstractActor {
2      /**
3       * The Query to be executed. Contains all required meta information.
4       */
5      private final QueryContainer<?, ?, Value, ?> builder;
6      /**
7       * The current index in the query path.
8       */
9      private final Integer currentIndex;
10     /**
11     The id of the vertex to be processed.
12     Parent id for giving information about the traveresed sub-tre.
13      */
14     private final String id, parentId;
15     /**
16     Reference to the parent actor.
17      */
18     private final ActorRef parent;
19     /**
20      * Reference to the QueryMasterActor.
21      */
22     private final ActorRef master;
23     /**
24      * Reference to the DatabaseActor
25      */
26     private final ActorRef databaseActor;
27     /*
28     Set of all references to child WorkerActors.
29      */
30     private Set<ActorRef> children;
31     /**
32      * The current type of the queryPath
33      */
34     private final Class currentType;
35     /*
36     The JSON containing all data of the current vertex.
37      */
38     private JsonNode document;
39
40     public static <Value, Result> Props props(ActorRef master, ActorRef parent, ActorRef
       ↪  databaseActor, QueryContainer<?, ?, Value, Result> builder, Integer currentIndex,
       ↪  String id, String parentId) {
41         return Props.create(WorkerActor.class, databaseActor, builder, currentIndex, id,
           ↪  parentId, parent, master);
42     }
43
44     public WorkerActor(ActorRef databaseActor, QueryContainer<?, ?, Value, ?> builder, Integer
       ↪  currentIndex, String id, String parentId, ActorRef parent, ActorRef master) {
45         this.databaseActor = databaseActor;
```

```
46          this.builder = builder;
47          this.currentIndex = currentIndex;
48          this.id = id;
49          this.parentId = parentId;
50          this.parent = parent;
51          this.master = master;
52          this.currentType = builder.getPath().get(currentIndex);
53      }
54
55      /**
56       * Defines the behaviour of the actor for received Messages
57       * @return
58       */
59      @Override
60      public Receive createReceive() {
61          return receiveBuilder()
62                  .match(StartProcessing.class, this::handleStartProcessing)
63                  .match(DatabaseResult.class, this::handleDatabaseResult)
64                  .match(WorkerFinished.class, this::handleWorkerFinished)
65                  .match(RecursionControlResponse.class, this::handleRecursionControlResponse)
66                  .build();
67      }
68
69      /**
70       * Handles StartProcessing message.
71       * Tells the DatabaseActor to extract the entity associated with the id
72       * provided in the constructor.
73       *
74       * @param message
75       */
76
77      private void handleStartProcessing(StartProcessing message) {
78          String typeName = currentType.getSimpleName();
79          String nextIdField = null;
80          if (currentIndex + 1 < builder.getPath().size()) {
81              nextIdField = builder.getPath().get(currentIndex + 1).getSimpleName();
82          }
83
84          databaseActor.tell(DatabaseRequest.create(typeName, id, nextIdField, getSelf()),
            ↪  getSelf());
85      }
86
87      /**
88       * Handles DatabaseResults.
89       * If the result is empty, the actor is finished.
90       * If the entity wrapped in the retrieved json files has multiple parents of
91       * the previous type in the query path, the actor send a RecursionControlRequest
92       * to the QueryMaster who decides whether the entity must be processed.
93       * Otherwise, this actor send a self message to start processing.
94       *
95       * @param
96       */
97      private void handleDatabaseResult(DatabaseResult message) {
98          String json = message.getDocument();
99          final JsonNode jsonDoc;
100         if (json == null) {
```

```
101             finished();
102             return;
103         }
104         try {
105             jsonDoc = Util.MAPPER.readTree(json);
106         } catch (IOException e) {
107             e.printStackTrace();
108             master.tell(e, getSelf());
109             return;
110         }
111         if (jsonDoc == null) {
112             finished();
113             return;
114         }
115         this.document = jsonDoc;
116         List<String> parentIds = getParentIds(document);
117 
118         /**
119          * Ask the recursion controller for permission to start recursion if the
120          * vertex has multiple parent vertecies of the previous type in the query path.
121          * The RecursionController will deny the processing of this vertex if it has been
122          * processed before.
123          */
124         if (parentIds.size() > 1) {
125             master.tell(new RecursionControlRequest(builder.getPath().get(currentIndex), id,
                ↪  self()), getSelf());
126         } else {
127             self().tell(new RecursionControlResponse(true), getSelf());
128         }
129 
130     }
131 
132     /**
133      * Handles WorkerFinished messages.
134      * If all children are finished, the actor send a WorkerFinished message to its parent.
135      *
136      * @param message
137      */
138     private void handleWorkerFinished(WorkerFinished message) {
139         getContext().stop(getSender());
140         if (children.isEmpty()) {
141             finished();
142             return;
143         }
144         children.remove(message.getRef());
145         Optional<ActorRef> first = children.stream().findFirst();
146 
147         if (!first.isPresent()) {
148             finished();
149             return;
150         }
151         //ENABLE FOR SEQUENTIAL DFS !!!!
152 //        children.stream().findFirst().get().tell(new StartProcessing(), getSelf());
153     }
154 
155     /**
```

```
156         * Handles RecursionResponses.
157         * If the request is denied, the recursion must not start. Since there is nothing
158         * to do, the actor is finished.
159         * @param message
160         */
161        private void handleRecursionControlResponse(RecursionControlResponse message) {
162            if (message.permitted()) {
163                startRecursion();
164            } else {
165                finished();
166            }
167
168        }
169
170        /**
171         * Tells the parent actor that the work of this actor is complete.
172         */
173        private void finished() {
174            parent.tell(new WorkerFinished(getSelf(), builder, id), getSelf());
175        }
176
177        /**
178         * Returns a list of string foreign ids contained in the idField of the provided document.
179         *
180         * @param document
181         * @param
182         * @return
183         */
184        private List<String> getChildIds(JsonNode document, String idField) {
185            if (document.has(idField)) {
186                List<String> ids = new ArrayList<>();
187                ArrayNode array = (ArrayNode) document.get(idField);
188                array.forEach(jsonNode -> ids.add(jsonNode.asText()));
189                return ids;
190            }
191            return Collections.emptyList();
192        }
193
194        /**
195         * Instantiate an object of the provided model class from the JsonNode.
196         *
197         * @param document
198         * @param type
199         * @return
200         * @throws IOException
201         */
202        static Object instanciate(JsonNode document, Class type) throws IOException {
203            return Util.MAPPER.readValue(document.toString(), type);
204        }
205
206        /**
207         * Returns the foreign ids of the parent vertecies from the previous
208         * index of the path.
209         *
210         * @param document
211         * @return
```

```
212         */
213         private List<String> getParentIds(JsonNode document) {
214             if (currentIndex == 0) {
215                 return Collections.emptyList();
216             }
217             List<String> ids = new ArrayList<>();
218             ArrayNode array = (ArrayNode) document.get(builder.getPath().get(currentIndex -
                ↪  1).getSimpleName());
219             array.forEach(jsonNode -> ids.add(jsonNode.asText()));
220             return ids;
221         }
222 
223         /**
224          * Executes user provided filters on vertices of a the currentType (if available)
225          * @param document
226          * @param currentType
227          * @return True if no filter is available or the current vertex passes the filter test.
228          */
229         private boolean executeFilter(JsonNode document, Class currentType) {
230             Object model = null;
231             Function<Object, Boolean> documentBooleanFunction = builder.getFilter(currentType);
232             if (documentBooleanFunction == null) {
233                 return true;
234             }
235             try {
236                 model = instanciate(document, currentType);
237             } catch (IOException e) {
238                 master.tell(e, getSelf());
239                 return false;
240             }
241             return documentBooleanFunction.apply(model);
242         }
243 
244         /**
245          * Applies the Function or BiFunction for the current type on the extracted entity.
246          */
247         private Value executeValueFunctions() throws IOException {
248             Function<Object, Value> documentValueFunction =
                ↪  builder.getDocumentValueFunction(currentType);
249             if (documentValueFunction != null) {
250                 Object model = instanciate(document, currentType);
251                 return documentValueFunction.apply(model);
252             }
253             BiFunction<Object, MetaContainer, Value> biFunction =
                ↪  builder.getEntityBiFunction(currentType);
254             if (biFunction != null) {
255                 Object model = instanciate(document, currentType);
256                 MetaContainer metaContainer = null;
257                 if(currentIndex > 0){
258                     Class parentClass = builder.getPath().get(currentIndex-1);
259                     metaContainer = new MetaContainer(parentClass,parentId);
260                 }
261                 return biFunction.apply(model, metaContainer);
262             }
263 
264             return null;
```

```
265     }
266
267     private void startRecursion() {
268         /*
269         Filter
270          */
271         if (!executeFilter(document, currentType)) {
272             finished();
273             return;
274         }
275         /*
276         Extract value
277          */
278         try {
279             Value value = null;
280             value = executeValueFunctions();
281             if (value != null) {
282                 master.tell(new ValueMessage<>(builder, id, value), getSelf());
283             }
284         } catch (IOException e) {
285             e.printStackTrace();
286         }
287
288         /*
289         Check for sub-query
290          */
291         if (currentIndex == builder.getPath().size() - 1) {
292             Set<? extends QueryContainer<?, ?, Value, ?>> childQueries =
                ↪  builder.getSubQueries(currentType);
293
294             if (childQueries != null) {
295                 /*
296                 Get the next type to be extracted from the the paths of each sub-query.
297                 Create new actors with the sub-query as argument for each child id.
298                  */
299                 children = childQueries.stream().map(subQuery -> {
300                     Class nextType = subQuery.getPath().get(1);
301                     List<String> childIds = getChildIds(document, nextType.getSimpleName());
302                     return childIds.stream().map(childId ->
303                             getContext().actorOf(WorkerActor.props(master, self(),
304                                     databaseActor, subQuery,
305                                     1, childId, id)));
306                 }).flatMap(actorRefStream -> actorRefStream).collect(Collectors.toSet());
307             }
308         } else {
309             /*
310             Create actors for a children of the next type in the query path.
311              */
312             List<String> childIds = getChildIds(document, builder.getPath().get(currentIndex +
                ↪  1).getSimpleName());
313             this.children = childIds.parallelStream().map(nextId -> {
314                 ActorRef recursion = getContext().actorOf(WorkerActor.props(master, self(),
                    ↪  databaseActor,
315                         builder, currentIndex + 1, nextId, id));
316                 return recursion;
317             }).collect(Collectors.toSet());
```

```
318         }
319         if (children == null) {
320             finished();
321             return;
322         }
323         children.parallelStream().forEach(actorRef -> actorRef.tell(new StartProcessing(),
            ↪ getSelf()));
324         //switch to use sequential dfs:
325 //            children.stream().findFirst().get().tell(new StartProcessing(), getSelf());
326     }
327 }
```

Code Snippet 9: Code snipped of the `WorkerActor` which is responsible for processing one vertex of the graph.