Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU UB WIEN Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

TU WIEN FAKULTÄT FÜR !NFORMATIK Faculty of Informatics

Pygmalion Query: Eine visuelle Abfragesprache für Graphendatenbanken

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Johannes Mauerer BSc
Matrikelnummer 0725917

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Silvia Miksch

Mitwirkung: Paolo Federico MSc

Albert Amor-Amorós MSc

Wien, 3. März 2015

Johannes Mauerer   Silvia Miksch

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Pygmalion Query: A visual query language for graph databases

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Johannes Mauerer BSc
Registration Number 0725917

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Silvia Miksch

Assistance: Paolo Federico MSc

Albert Amor-Amorós MSc

Vienna, 3rd March, 2015

Johannes Mauerer   Silvia Miksch

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Johannes Mauerer BSc
Obere Weissgerberstr. 19/7, 1030 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. März 2015

Johannes Mauerer

Danksagung

Aldous Huxley hat geschrieben, dass nach der Stille die Musik am ehesten das Unbeschreibliche auszudrücken vermag. Ich bin meiner Familie, meinen Freunden, meinen Arbeitskollegen und meinen Betreuern in einer unbeschreiblichen Weise dankbar und ich hoffe, dass euch die Musik so viel Freude bringen wird wie ihr mir bisher gebracht habt.

Acknowledgements

Aldous Huxley wrote that after silence, that which comes nearest to expressing the inexpressible is music. I am grateful in an inexpressible way to my family, my friends, my colleagues and my advisors. I hope that there will be music in your life that delivers you as much happiness as you have brought me until now.

Kurzfassung

Graphendatenbanken, die in den letzten Jahren einen Anstieg an Interesse erfahren haben, erlauben es Netzwerkstrukturen effizienter zu speichern. Aber trotz dem bereits einige Forschung in die Weiterentwicklung dieser Technologie geflossen ist, fehlen für die Abfrage von Teilmengen dieser Daten benutzerfreundliche Oberflächen. Von den zwei Hauptabfragemethoden Graphenmusterfindung und Graphen Traversierung gibt es für Erstere mehr Methoden zur visuellen Unterstützung von Abfragen. Für letztere, trotz ihrer Mächtigkeit, gibt es wenig Fortschritt im Bereich der visuellen Abfragen. Diese Diplomarbeit zielt darauf ab, eine neuartige Benutzerführung für die Erstellung von Graphen Traversierungsabfragen zu konstruieren, genannt Pygmalion Query.

In der Arbeit wurde zuerst eine Literaturrecherche betrieben um bisherig vorhandene Ansätze zu einer solchen visuelle Abfragesprache zu finden. Mithilfe dieser Recherche wurden Lücken in der Forschung aufgedeckt. Nach der Literaturrecherche wurden Bedürfnisse und Anforderungen aus unterschiedlichen Quellen, beispielweise aus Abfragen die Online gestellt wurden oder aus der Dokumentation einer Graphentraversierungssprache, ermittelt. Nach der Selektion einer minimal benötigten Menge an Merkmalen wurde das Design für Pygmalion Query entwickelt. Eine web-basierte Implementierung, erstellt mit Hilfe von frei verfügbaren Frameworks, wurde konstruiert. Nach der Entwicklung von Pygmalion Query wurde eine zweifache Evaluierung durchgeführt. Eine Expertenbewertung diente der Bestätigung des Ansatzes. Mit der Resonanz der Experten wurde eine aktualisierte Version erstellt, der eine kleine Komparative Nutzerstudie zur Testung der Benutzerfreundlichkeit folgte.

Die Resultate aus der Expertenbewertung und der Nutzerstudie deuten einen positiven Benutzerfreundlichkeitseffekt von Pygmalion Query für die Erstellung von Graphentraversierungsabfragen gegenüber den derzeit vorhandenen Methoden an. Die Ergebnisse zeigen dass Teilnehmer der Studie, grösstenteils bestehend aus Novizen auf dem Gebiet, die Ihnen gestellten Aufgaben eher bewältigen können wenn sie die neuartige visuelle Abfragesprache verwenden.

Abstract

Graph databases, which have seen a surge in interest over the last years, allow to more efficiently store network structured data. But while plenty of research has gone into the development of this technology, querying for a subset of the data is lacking user friendly interfaces. Of the two main query methods, graph pattern matching and graph traversal, the first has received more attention and more methods providing visual support in querying are available. The latter, graph traversal, while being very powerful, has seen little advances in visual querying. This thesis aims at providing a novel user interface for graph traversal query formulation - a visual query language for graph databases - entitled Pygmalion Query.

In the thesis, first a literature review is undertaken to discover any previous approaches taken at such a visual query language. With it, gaps in the currently available research are identified. Following the literature review, needs and requirements are identified from different sources, such as queries posted online and documentation for graph traversal query languages. After selection of minimum required features, the design for Pygmalion Query is created. A web-based implementation, built on available frameworks, is implemented. Following the creation of Pygmalion Query, a twofold evaluation is conducted. An expert review serves as the initial confirmation of the approach taken. Using feedback coming from the experts, an updated implementation is created. A small comparative user study is carried to test for usability.

The results of the expert review and user study indicate a positive usability effect of Pygmalion Query in the formulation of graph traversal queries over the currently available solutions. The participants of the study, in greatest part novice users, are more likely to complete the tasks posed to them with the visual query language.

Contents

Kurzfassung xi

Abstract xiii

Contents xv

List of Figures xvi

List of Tables xviii

List of Code Listings xviii

1 Introduction 1
- 1.1 General Introduction . . . 1
- 1.2 Background and Motivation . . . 2
- 1.3 Research Questions . . . 3
- 1.4 Research Methodology . . . 3
- 1.5 Structure of the Thesis . . . 4

2 Concepts, Related and Previous Work 7
- 2.1 Graph Databases and Query Languages . . . 7
- 2.2 Methods and Objects of Visualization . . . 16
- 2.3 Visual (Programming) Languages . . . 19
- 2.4 Visual Query Languages for Graph Databases . . . 24

3 Design 33
- 3.1 Requirements Gathering . . . 33
- 3.2 Requirements Identification . . . 41
- 3.3 Features Design . . . 45

4 Implementation 57
- 4.1 Overview . . . 57
- 4.2 Technical Details . . . 57
- 4.3 Important Features . . . 64

4.4 Summary of the implementation 69

5 Evaluation **71**
5.1 The Evaluation Settings 71
5.2 Expert Review 72
5.3 Implemented Feedback 77
5.4 Comparative User Study 79
5.5 Summary of the evaluation 83

6 Conclusion **85**

A Appendix **87**
A.1 Pygmalion Query File List 87
A.2 User Study Material 88
A.3 Pygmalion Query block naming adaptions 88

Bibliography **95**

Glossary **101**

List of Figures

2.1 Three graph data models: The property graph model (left), hypergraph model (middle) and triples (right). 9
2.2 An overview of necessary transformation to move between different graph data models. 9
2.3 An overview of the evolution of database models, newly created based on [3]. 11
2.4 The Gremlitron, showcasing the different parts of the Apache Tinkerpop stack. 13
2.5 Jacques Bertin table of representation forms, from Semiologie Graphique (published in 1967). 18
2.6 A simple, exemplary Scratch control flow that will move the character 10 steps, then play a sound, then given 1 is smaller than 2 move another 10 steps 10 times with finally being deleted. 20
2.7 An illustration of different visual query language types, based on the definition of Visual Query Languages in [13]. 25

2.8 Graph database optimization via visual querying, simplified from [10] Fig. 2. GUI stands for Graphical User Interface and TGL is the TGL Translator, *"[...] consisting of the query translation component and the result translation component"*. 27

3.1 The chosen requirements from Table 3.1 and their matching to the identified categories. 44
3.2 A process diagram for asynchronous, template driven probing. 46
3.3 The design draft for visualizing the flow category and its attribute vectors. . 48
3.4 Some visualization examples for the potential not-out functional block. 49
3.5 A graphical representation of the functional blocks for query building. 50
3.6 Based on Figure 3.4, utilizing highlights and layover on functional blocks for user guidance. 52
3.7 Screenshot of the text editor "Sublime Text", taken from the official webpage. The birds eye view pane on the right is an example of an overview. 53
3.8 Drafts of the overview view. The left side provides a generic disconnected graph that can be highlighted to show the traversal. On the right aggregations of the main types (Vertices, Edges, Scalar) and a schematic overview of the traversal. 54
3.9 This figure builds on Figure 3.8, (*ii*), a distorted, more detailed view of the current context. 54

4.1 A screenshot depicting the implemented Pygmalion Query with an exemplary query built. 58
4.2 A screenshot depicting the result view of Pygmalion Query. 59
4.3 A visual representation of the schema of the query formulation part of Pygmalion Query. 60
4.4 A visual representation of the schema of the query code generation. 60
4.5 A visual representation of the schema of the query code construction. 60
4.6 A visual representation of the schema of the server communication. 61
4.7 A visual representation of the schema of the results visualization. 61
4.8 A step by step view of the first actions taken to creating a query with Pygmalion Query. 65
4.9 An un-instantiated blocks and an instantiated block. 65
4.10 A comparison of the same block in two different contexts. 66
4.11 The Pygmalion Query overview explained. 66
4.12 The flow from query formulation changes to retrieving context data and visualizing it. 67
4.13 An example of having multiple probes within a query flow. 69

5.1 A short introduction to Pygmalion Query as presented to the experts. 74
5.2 An example of the visualization to display the available percentage of a specific label type. 76
5.3 The revised Pygmalion Query. 79

5.4 The "terminal" which Group 2 of the user study was given to complete the tasks. Queries that resulted in an error or that returned too many results notified the user of this. 80
5.5 The results of the comparative user study, seen in Table 5.2, visualized. . . . 82

A.1 Introduction for the visual query language. 91
A.2 Introduction for the "terminal". 92
A.3 The tasks in the user study. 93
A.4 Additional information gathered within the user study. 94

List of Tables

2.1 Visualization 17
2.2 Mackinlay's guidance on quantitative/ordinal/categorical data representation 19
2.3 Comparison of available query types in popular graph query languages. . . . 30

3.1 List of requirements and their rating in necessity, feasibility and helpfulness. . 43
3.2 Function and data flow design methods. 47
3.3 The not-out functional block. 48

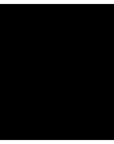
5.1 Overview of the participants of the user study. 81
5.2 Aggregated results of the comparative user study. 82

A.1 File list of Pygmalion Query. 88
A.2 Version 1 (Expert review) and Version 2 (User Study) of the block naming. . 89
A.3 Version 1 (Expert review) and Version 2 (User Study) of the block naming. . 90

List of Code Listings

2.1 Gremlin recursion example 12
3.1 Query example 1 35
3.2 Query example 2 36

3.3 Query example 3 37
3.4 Query example 4 38
3.5 Results querying with templating 55
4.1 Example of Blockly element creation 63
4.2 Example of a query template 68
4.3 Templates for the graph probe 68



CHAPTER 1

Introduction

1.1 General Introduction

More and more data is being collected and stored to serve different purposes. An increasing interest lies on the data structured in networks, coming (for example) from social networks. With this data being available within companies, more people are hired as analysts and try to gather insights out of the collected data. While there has been many advances in the collection and processing of this data, the retrieval in a user friendly manner hasn't progressed so quickly. Often analysts have to rely on available methods such as the structured query language (SQL) to manually write queries to retrieve subsets of data. Especially in tabular stored data, this can be quite straight forward. With data stored as networks, or complex structures, the available retrieval methods involve writing code or relying on a less powerful subset of SQL that has been adapted to fit into the new use case.

As most of the data collection and processing work is done automatically and implemented/maintained by software engineers, the need for a more user friendly way to retrieve subsets of more complex stored data isn't necessarily obvious. But with increasing importance of being able to retrieve data, without having to go back into code, visual query languages pose an interesting field of research.

Picture a very simplified model of the world, only depicting the people that live in it and a connection between two people if they know each other. Furthermore, let's store this model in a database, with people being entities (vertices) and connections being links between entities (edges). If you have this data stored, how do you retrieve a subset giving you all the people that you know? In this thesis, I want to design and develop a user interface that allows anyone to query for graph data.

1.2 Background and Motivation

So why is it necessary and important to be able to have visual support for the formulation of queries for graph databases? Graphs are being used across a wide field and ranges, being used to represent data in:

- transportation and other networks
- geographical information
- semistructured data
- (hyper)document structure
- semantic associations in criminal investigations
- bibliographic citation analysis
- pathways in biological processes
- knowledge representation (e.g. semantic web)
- program analysis
- workflow systems and
- data provenance.

Since over 25 years query languages for graph databases have been started to be investigated. Stemming from interest in hypertext systems, the focus of research in this area now lies within social networks and the semantic web. Many different query languages for graph databases exist, such as G, SPARQL, GraphLog, GRAM, GraphDB, GOOD, G-Log, GUL, UnQL, etc. These languages try to tackle different query functionalities, such as subgraph matching, finding notes connected by paths, comparing and returning paths, aggregation, node creation and approximate matching and ranking. [66] Although graphs are an easy to understand concept, query languages make it hard for a non-expert user to retrieve subsets of an underlying graph. With an increased use of graph databases in different sectors, there is also a greater need to query these databases without having prior knowledge to query languages.

In [10], the authors *"believe that the diagrammatic query and visual result display will ease the task of data management and data analysis."*

1.3 Research Questions

With this thesis, I would like to prove that visual query languages have a place in the world of query languages for graph databases - both for novice as well as for expert users. The main research question for the project is:

Research Question: "How can we visually support the formulation of queries for a graph database?"

To further narrow the scope of the question, the following hypotheses will be addressed in this thesis:

[**H1**]: A novice user can retrieve a subset of data stored as property graphs quicker with a VQL than a text-based query language.

[**H2**]: More experienced users can benefit from VQL by being able to specify more complex queries both faster as well as more consistent.

[**H3**]: A VQL for graph databases can be unlinked from any specific database implementation.

*Additionally to [**H2**] The hypothesis shall show that not only simple queries can be formulated with the VQL.*

To answer the research questions, first a literature research on currently available visual query languages for graph databases and its sub-topics will be conducted. The following two chapters will focus on designing and implementing a new visual query language entitled Pygmalion Query.[1] The following section will outline the research methodology further.

1.4 Research Methodology

To arrive at an answer for the posed research question and hypotheses, the following research methodology is applied:

1. **Literature review and research**
   First a review of available literature is conducted. This includes important concepts that are being referred to in the thesis. With the literature review, the aim is to identify all already available potential answers to the research question in advance.

[1]Pygmalion is a sculptor in Ovid's Metamorphoses who fell in love with a statue he had carved. The link between visual query generation ("building a query") and sculpting lead to the choice of this name.

If a gap is identified, which leads to an unavailable answer, the following methods are used to allow this thesis to find one. The literature review will also serve to identify available visualization methods to be utilized.

2. **Requirements analysis and selection of visualization methods**
   To create a visual query language for graph databases, first the needs of potential users for such an interface need to be gathered. The needs can be gathered by studying the current state of querying for subsets of data stored as graphs. Furthermore, available documentation on query infrastructure allows to identify the intent a query language has. Once the requirements for a visual query language have been defined, design sketches drawn from the previously identified visualization methods are outlined.

3. **Prototype development/implementation**
   The prototype development is aimed at taking the design and providing a means to test if the requirements are met by the visual query language. With most new emerging systems being online, the implementation will be a web based user interface.

4. **Expert review and comparative user study**
   Using the implementation, the evaluation will help to answer the research question. Two different methodologies are applied sequentially: First, an expert review will be used to validate the approach taken at the design/implementation of the visual query language. With learnings from this review, a small comparative user study will be conducted to provide an indication of usability for users.

1.5 Structure of the Thesis

This thesis is structured into five chapters which in many respects correspond to the different steps of the research methodology. The chapters are:

- **Chapter 2 Concepts, Related and Previous Work:** This chapter corresponds to the literature review and research part of the methodology as outlined in section 1.4. Literature research provides the necessary and required insights into the current state of the art as well as introducing utilized concepts across the thesis. In the end of the chapter, a comparison of available visual query languages in the space of graph databases is introduced and the gap which this thesis is trying to fill highlighted.

- **Chapter 3 Design:** In the design chapter of the thesis, two main sections are provided: First, a section on finding required features within the proposed visual query language found through different means. As the all the identified features may or not be ultimately required, a ranking on multiple layers is applied to the requirements. With this ranking, a simple heuristic is chosen to select the features ultimately to be designed. The second part of the chapter will then go into the

design and visualization methods utilized to accomplish this. Specific architectural features are outlined. All features are split across categories which are also defined in this chapter.

- **Chapter 4 Implementation:** In the previous chapters the need for a visual query language for graph traversals was highlighted, requirements identified and finally designed. This chapter will outline the implementation of the visual query language done in the thesis. The main features of the web based approach are described in detail. Frameworks that are utilized in the implementation are also introduced and discussed in level of detail corresponding to their importance in the implementation.
- **Chapter 5 Evaluation:** This chapter again corresponds very closely to the expert review and comparative user study part of the methodology outlined in section 1.4. Using the implementation of the previous chapter, different evaluation methods are used to finally reach an indication of usability. As the work in this thesis is in a relatively understudied field, a larger study will be necessary in the future to statistically significant prove that the approach taken adds value to graph traversal query formulation.
- **Chapter 6 Conclusion:** The conclusion will summarize the findings of the previous chapters. Furthermore, a look is taken on the questions left open in the thesis. Finally, an outlook provides some pointers towards potential future work that might follow this thesis.

CHAPTER 2

Concepts, Related and Previous Work

"Solving a problem simply means representing it so as to make the solution transparent." - **Herbert Simon**

This chapter will establish the current state of (visual) query languages for graph databases and show the structure of the thesis. First a quick introduction into visualization methods is given. This is followed by a look at the current state of the art of visual programming languages. These are overlapping with visual query languages, which will be identified later on. An overview of graph databases, underlying models and querying languages is given in the third section of this chapter. Finally, in the last section, the current state of the art of visual query languages in graph databases is outlined and compared. Gaps in the currently available solutions will serve as the justification and positioning of Pygmalion Query.

2.1 Graph Databases and Query Languages

2.1.1 A brief introduction to graphs

Graph theory was pioneered by Euler[1] in the 18th Century, with different scientific fields (such as Mathematics, Anthropology, Sociology) actively and extensively studying graphs ever since.

"In its simplest form a graph G is a pair (V,E), where V is a finite set of vertices and E is a finite set of edges connecting pairs of vertices." [66] Both vertices and edges can be labeled with attributes. Edges can be directed and undirected. In more complex forms,

[1]Leonhard Euler, 15 April 1707 until 18 September 1783

each vertex of a graph can be its own graph again (e.g. hypernode model). Notation for elements are:

- Vertices: $v_0, v_m \epsilon V$
- Edges: (x_i, a_i, y_j), $x_i, y_j \ \epsilon \ V$

A path ρ between vertices v_0 and v_m in a graph $G = (V, E)$ is a sequence $v_0,\ a_0,\ v_1,\ a_1,\ v_2\ ...\ v_{m-1},\ a_{m-1},\ v_m$, where $m \geq 0$, $v_i \epsilon V(1 \leq i \leq m)$, $a_i \epsilon \Sigma(1 \leq i < m)$, and $(v_i, a_i, v_{i+1}) \epsilon\ E(1 \leq i < m)$.

This simple form of graphs, in which each edge (directed or undirected) has the same meaning, is called single-relational graph. This and other types of graphs will be discussed in subsection 2.1.2.

2.1.2 Data modeling of graphs

In the previous section, the basic model of a graph was outlined. But in most applications, this single-relational graph falls short of being able to depict all the necessary elements. As one of the main factors, this contributes to the choice of the property graph model, which is a multi-relational graph. Furthermore, as [40] point out, a major driver of the property graph model, Blueprints, is increasing in popularity and becoming the de-facto standard: *"[...] most major graph databases propose a Blueprints implementation"*.

The property graph model may be written as $G = (V, E, \lambda, \mu)$. Next to the already familiar set of vertices V and edges E, edges are directed and labeled with the λ function that maps onto a discrete set of categorical values $\lambda : E \to \Sigma$. Properties are a map from elements and keys to values in the function $\mu : (V \cup E) \times R \to S$. [56] Speaking in simple terms, a property graph is a directed (an edge has a tail and a head vertex), attributed (vertices and edges can have an arbitrary number of key/values pairs), multi-relational (edges are types to support multiple types of relationships), binary graph (an edge connects only two vertices). [49]

Another model is the Hypergraph model. The biggest difference to the property graph model lies in the relationships - in the property graph model a 1:1 relationship is available. The Hypergraph extends this to N:N - multiple start and end nodes are allowed. It is noteworthy that the two models are isomorphic, i.e. it is possible to display the information of one in the other.

Triples have their roots in the semantic web movement and are made up of subject-predicate-object data structures. A typical triple would be "Adel makes Noodles" [52]. The most famous triples based data-model is RDF. Tripes can be translated into property graphs and vice-versa [21]. The larger differences between the two are the storage implementation, query languages and performance in different use cases.

Figure 2.1 illustrates an overview over the three discussed graph data models, the property graph model, the hypergraph model and triples.

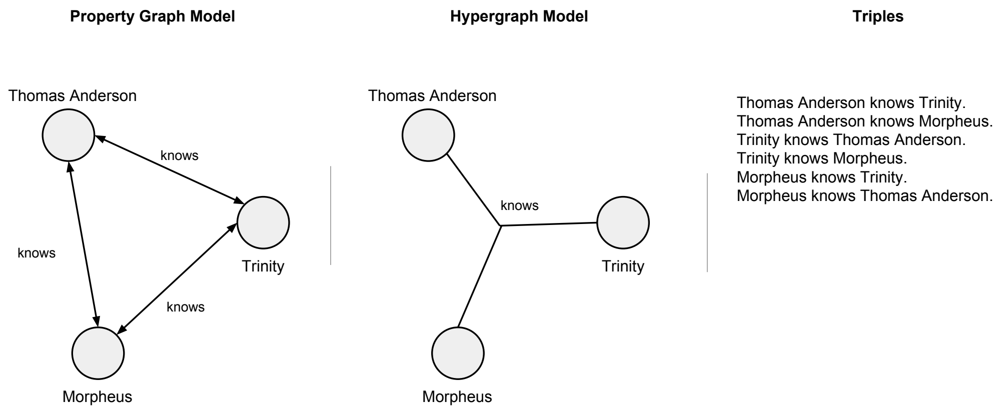


Figure 2.1: Three graph data models: The property graph model (left), hypergraph model (middle) and triples (right).

Furthermore there are more graph types, which are depicted in Figure 2.2 [55]. The figure also displays what changes to each data model are necessary to be made to navigate from one to the other.

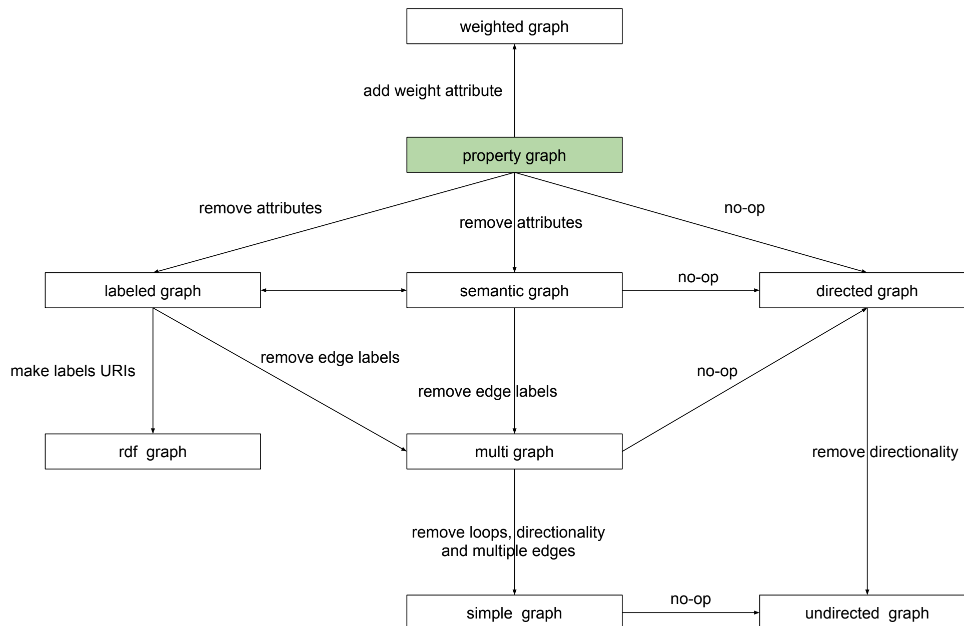


Figure 2.2: An overview of necessary transformation to move between different graph data models.

2.1.3 What are graph databases?

Graph databases have seen a surge in interest of the last years. Different applications, such as Recommender Systems (e.g. Amazons recommendation for items based on search/buying history of others), Social Graphs (e.g. social networks such as Facebook, displaying relevant news from friends of friends) or Bioinformatics (e.g. relating complex webs of information that includes genes, proteins and enzymes) benefit from using graph databases rather than relational database systems (RDBMS). Miller et al. in [47] state that using a graph database over a RDBMS will always depend on the use case, rather than just preference of the new over the old system.

The creation of graph databases was a result of modeling graph-like structures in RDBMS and hitting obstacles that could more easily be overcome by representing the data differently. Güting describes in [27] GraphDB, one of the first graph database implementations. The data model consists of three types, simple classes (e.g. a Book object), link classes (e.g. who wrote a specific book) and path classes (e.g. a highway path). Graph databases fall under the category of NoSQL databases, being one of five sub-categories[33]:

- key-value stores
- wide-column stores
- big table
- document (e.g. MongoDB)
- graph databases.

Essentially, graph databases are databases, in which the underlying data model is of type graph. A database model is (in simple terms) made up of three components:

- a set of data structure types
- a set of operators or inference rules
- and a set of integrity rules.

Speaking specifically about graph database models, the definition is:
"Graph database models can be defined as those in which data structures for the schema and instances are modeled as graphs or generalizations of them, and data manipulation is expressed by graph-oriented operations and type constructors." [3] The authors of [3] also provide a chart showing the evolution of database models, which is depicted in Figure 2.3 and go on to provide a full overview of available graph database models.

In Figure 2.3, the top part ranges back long before 1970. This shows that, while graph databases may not be the first type of database that have been developed, their basis is founded in both graph theory and mathematical logic, dating back before other databases.

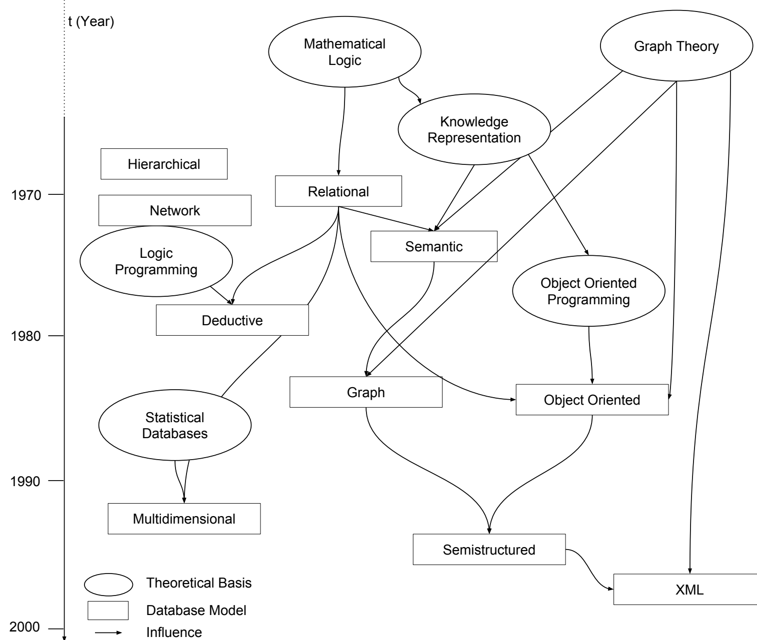


Figure 2.3: An overview of the evolution of database models, newly created based on [3].

2.1.4 Introduction into query languages

An important part of any database is the availability of a query language. *"Query languages are computer languages used to make queries into databases and information systems."* [25] The success of relational databases in the last decades can also be attributed to the success of the SQL for these types of databases [32]. Many different query language implementations for various different types of databases exist. Famous or recent examples include:

- SQL: Structured Query Language, which is used to query relational databases
- XQuery: A query language for XML
- OQL: Object query language
- SPARQL: A query language for RDF graphs
- FQL: Facebook query language which allows to write SQL-like queries [23]

While query languages all aim at the same goal - knowledge discovery in a dataset by retrieving a subset that fits specific properties - each query language is optimized at solving something specific better than others. This may be an optimization for a specific database type, optimizing towards speed, etc. There can thus be no silver bullet for storing/querying data, but rather the choice is linked to the specific use case.

2.1.5 Comparison of graph traversal and graph pattern matching

Within query languages for graph databases, two main concepts are separated: graph pattern matching and graph traversal. This thesis will focus on graph traversal. While being fundamentally different in their concept, both graph traversal as well as graph pattern matching allow for retrieving complex queries in graphs with underlying graph property model - the model utilized here. [53] provides an overview over the differences between the two concepts, from the graph traversal side.

For comparison, two popular implementations are used:

- Graph pattern matching: SPARQL [65]
- Graph traversal: Gremlin [63]

A brief comparison between the two is also provided in [54]. [41] compares SPARQL, Cypher (a Neo4J specific graph query language) and Gremlin as *"dialects of a common graph language"*. A disclaimer to point out when using the above languages as basis for comparison is, that Gremlin can run SPARQL queries.

Because of its procedural nature, graph traversal allows for recursion which SPARQL, in its version 1, doesn't. An easy example to explain this is thinking of a simple social network and querying for all friends and friends of friends of a specific person. In Gremlin, this would be expressed through a statement like the following:

Code Listing 2.1: Gremlin recursion example

```
gremlin> g.v(1).out('friendOf')
       .loop(1){true}{it.out('friendOf').count() == 0}
```

In SPARQL, recursion as above isn't possible. Rather, solutions for example as proposed in [4] have been created. In this extension, a SPARQL function is exposed that takes an arbitrarily complex SPARQL query as a parameter and executes it over a specific endpoint. The implementation is similar to the SPARQL (1.1) Service clause but more powerful.

2.1.6 Existing implementations of query languages for graph databases

With the rise of interest in graph databases, many new query languages have been created often aiming at optimizing queries against large data sets. A small overview of the available query languages is presented here.

Giugno and Shasha [24] present an application-independent graph querying language entitled GraphGrep that utilizes Glide (that combines features from XPath and Smart). With GraphGrep the authors tackle the NP-complete problem of finding a specific subgraph within a database of graphs. He and Singh [30] present a graph query language *"that supports arbitrary attributes on nodes, edges, and graphs."* They also try to tackle the NP-complete problem by the optimization of the search order, usage of neighborhood

sub-graphs and profiles and joint reduction of the search space. The graph algebra used extends the relational algebra. Main purpose of the language is graph pattern matching.

Holzschuher and Peinl [32] compare query languages within a relational database backend versus a graph database backend. The authors use Apache Shindig, the OpenSocial reference implementation with their own Neo4J based backend. They use both available query languages for Neo4J, Gremlin and Cypher and put them in a test against native Java traversal and JPA created queries. The authors find both Gremlin and Cypher outperform against native access, with Gremlin being the best performing engine.

2.1.7 Brief Introduction into Gremlin

Figure 2.4: The Gremlitron, showcasing the different parts of the Apache Tinkerpop stack.

A specific implementation of a graph traversal query language is included in the Apache Tinkerpop framework, previously known as Gremlin. As the Tinkerpop framework underwent changes throughout the writing of this thesis, Apache Tinkerpop is used as the name for the full framework, Gremlin for the graph traversal language and Gremlin Server for the included server.

The Tinkerpop Stack features the following parts (taken from [63]:

- Blueprints (now Gremlin Structure API) - Blueprints is a property graph model interface with provided implementations. Databases that implement the Blueprints interfaces automatically support Blueprints-enabled applications.

- Pipes (GraphTraversal) - Pipes is a dataflow framework that enables the splitting, merging, filtering, and transformation of data from input to output. Computations are evaluated in a memory-efficient, lazy fashion.

- Gremlin - Gremlin is a domain specific language for traversing property graphs. This language has application in the areas of graph query, analysis, and manipulation.

- Frames (Traversal) - Frames exposes the elements of a Blueprints graph as Java objects. Instead of writing software in terms of vertices and edges, with Frames, software is written in terms of domain objects and their relationships to each other.

- Furnace (GraphComputer) - Furnace is a property graph algorithms package. It provides implementations for standard graph analysis algorithms that can be applied to property graphs in a meaningful ways.

- Rexster (GremlinServer) - Rexster is a multi-faceted graph server that exposes any Blueprints graph through several mechanisms with a general focus on REST.

The choice fell on the Apache Tinkerpop Stack because of it's powerful graph traversal language and the increasing popularity of Tinkerpop interfaces in the graph community [62]. With the introduction of the Tinkerpop3 stack, the principal graph model was extended. In the version ≥ 3, vertices now have properties of the type VertexProperty. Compared to the normal key-value properties of a normal property graph (and also edges in Tinkerpop3), these are derived from the same class as vertices and edges itself (Element). This implies that properties can be repeated with the same key (e.g. multiple properties with the key "name") and properties can have properties themselves (e.g. a property with the key "name" can have a property with specific ACLs attached to it).

This section draws heavily from the Gremlin documentation(s), which is (for convenience) linked again below (see footnote for links):

- Graph Traversal JavaDoc[2]

- Tinkerpop 3.0.0.M4 Documentation[3]

- SQL2Gremlin (unofficial)[4]

- Gremlin Docs (unofficial)[5]

Gremlin as a query language provides very complex functionalities. Each of the different features are entitled "Step" as they are all mapped to the traversal of the graph, which happens in these steps. Traversal starts at either a graph, a vertex or an edge. Although many different steps exist, which are all implementations of four general steps:

- map: The most basic step, allowing to map the traverser to some object.

- filter: Filter can be seen in the literal sense, i.e. passing elements through a boolean gate and only allowing them through in case of true.

- sideEffect: Perform an operation on the traverser and then pass it on.

[2]http://www.tinkerpop.com/javadocs/3.0.0.M4/core/com/tinkerpop/gremlin/process/graph/GraphTraversal.html
[3]http://www.tinkerpop.com/docs/3.0.0.M4/
[4]http://sql2gremlin.com/
[5]http://gremlindocs.com/

- flatMap: Similar to map, but in this case the traverser is mapped to an iterator, allowing to go through the elements one by one.

Another general form, which is not part of the main four steps needed to build the other steps is branch, which allows to split the traversal into multiple different streams. The most important concept for a user to grasp is the traversal concept of the query language. As this is the main element of any query, this functionality shall be discussed and designed first. In a very literal sense, the most basic concept of "traversing" can be mapped the flow of liquid in a system of pipes. A graph in this case is any collection of pipes and junctions. Each of the pipes and junctions are labeled and are one-way only (directed graph).

2.1.8 Complex Gremlin Steps

- AddEdge Step: The add edge step allows for an expression to result in the addition of edges. For example an expression that explains co-authorship of between and other authors through existing vertices explaining authors and edges explaining authorship of vertices explaining articles. (The theory behind this is reasoning, making explicit)
- Aggregate Step: Allows for aggregation of the current object of the traversal. For example aggregation of all articles a specific vertex explaining author has created.
- Back Step: Although Graph Traversal move forward, the back step allows to retrieve elements previously seen. A use case for this would be to see all secondary connections of a vertex that lead to reaching a connection further down the line.
- Choose Step: The choose step is similar to the logical if/else statement.
- GroupBy Step: The GroupBy step can easily be compared to the GROUP BY statement in SQL: Take a specific entity type as key and group all entries under these keys. Gremlin allows for 3 lambda functions to be attached to the statement:
  - Key-lambda: Return the keys to the returned results, grouped under the keys.
  - Value-lambda: Retrieve a specific property of the returned results.
  - Reduce-lambda: Return a specific feature (such as count) of the returned list per key.
- Group CountStep: Counting how many times a specific item has been seen during the traversal.
- Inject Step: The inject steps allows to "create" objects in the traversal on the fly. For example, an arbitrary object representing an integer can be added to reference to it in later stage.
- Jump Step: The jump step quite literally describes a jump within the traversal flow based on specific conditions. This allows for while/do and do/while like statements.

- Match Step: In subsection 2.1.5, a comparison between graph pattern matching and graph traversal was done. The match step in Gremlin allows for a declarative way of expressing a pattern matching query.
- Order Step: The order step allows to sort the objects of the traversal stream. It uses a comparator to allow for different comparison functions for the sorting.
- OrderBy Step: To easily compare Elements within the graph by specific properties and sort them, the OrderBy step will allow to do so (in ascending and descending order).
- Path Step: Within the traverser lies a history of all traversal steps done. The path step allows to retrieve these steps - resulting in potentially large lists having to be stored during runtime.
- Select Step: With the select step, either labeled steps within a map or objects out of a map flow can be retrieved.
- Store Step: Store enables lazy aggregation (compared to the aggregate step which is eager computation).
- Subgraph Step: The subgraph step allows to retrieve a set of vertices/edges as defined in the statement. An easy example for this is to retrieve a graph within a larger one which surrounds only one vertex.
- TimeLimit Step: As sometimes not exact results/answers are necessary, but rather relative rankings the timelimit step introduces a way of stopping computation after a specific time interval.
- Tree Step: This step allows for a path resulting from a traversal to be aggregated into a tree.
- Until Step: In relation to the jump step (enabling do-while/while-do loops), the until step introduces simplicity around these concepts by providing a means to specify the breakout condition in the beginning of the statement.

2.2 Methods and Objects of Visualization

2.2.1 A brief introduction to visualizations

Although information visualization has seen a rise in popularity within the last decades, the very essence of it can already be found in famous examples such as the map of the easterly advance by Napoleon's army from the Polish-Russian border towards Moscow and it's subsequent retreat. In the visualization created by Charles Minard in 1869, five variables are represented (man power, geography, direction of offense/retreat, temperature and date). Another famous example is John Snow's map of cholera incidents created in 1854. His visualization, drawing a black bar on the location in a map where another

outbreak was helped identify the spring ultimately responsible for the outbreak. See Table 2.1 for the visualization from Charles Minard (left) and John Snow (right).

Table 2.1: Visualization

The point trying to prove with these early visualization examples is, quoted from [60]: "Visualization is solely a human cognitive activity and has nothing to do with computers.". The author further goes on saying "The principal task of information visualization is to allow information to be derived from data." A task plenty of time older than graph databases, query languages or computers for that matter also carries plenty of research within it. For a visual query language that doesn't focus on the visualization of query results but rather the query forming process, not all learnings can be simply applied but most be combined with modern UXD research. Psomas in [50] sums up the five competencies for UXD as Information Architecture, Interaction Design, Usability Engineering, Visual Design and Prototype Engineering. A field this vast and complex has been researched plenty.

In subsection 2.2.2 elements and visual cues to be utilized in a visual query language will be outlined.

2.2.2 Elements of visualization

When trying to visualize a specific dataset, multiple tasks arise. These range from data selection, to interactive attribute selection to filtering and of course many more. To address all of these, the representation utilized is key. Or, as Herbert Simon put it: "To solve a problem is simply representing it as to make the solution transparent." [58]. Spence [60] lists different ways of representation available, based on Jacques Bertin's table (Figure 2.5 [6]). The table shows different visual representation types (such as points

[6] Taken from http://understandinggraphics.com/visualizations/information-display-tips/

Figure 2.5: Jacques Bertin table of representation forms, from Semiologie Graphique (published in 1967).

or lines) on the horizontal and encoding mechanisms (or retinal variables) on the vertical. The table gives an indication on specific data (specifically qualitative or quantitative) and the suitability of specific encoding for the types.

The process of information visualization usually starts with the ultimate goal, what is the aim of the visualization. In the example of Charles Minard's visualization of Napoleon this could be allowing for easy depiction on why the army failed, even though it was of great large numbers. In John Snow's visualization on the other hand it was to specifically identify a source. Once the aim has been set, the underlying data will have a large impact on the way of representation. Data is usually separated into quantitative, ordinal and categorical data. In graphs, all 3 will be found in the attributes. Mackinlay QUOTE gives guidance on mapping different cues to these types in Table Table 2.2. The elements are ordered as rankings from top to bottom.

The difficulty in creating a visual query language for property graphs is that the underlying data is multivariate, thus posing a substantial challenge.

To understand better the specific use case of a visual query language, the author of [57] writes: *"Every visualization follows the concept of the visualization pipeline [...]"*. The steps outlined are data acquisition, filtering, mapping and rendering. Within these steps, Pygmalion Query can be mapped to step 2, filtering. It is described as an user centered step in which the user selects what subset of data he/she wants to see.

Quantitative	Ordinal	Categorical
Position	Position	Position
Length	Density	Colour hue
Angle	Colour Saturation	Texture
Slope	Colour hue	Connection
Area	Texture	Containment
Volume	Connection	Density
Density	Containment	Colour saturation
Shape	Length	Shape
	Angle	Length
	Slope	Angle
	Area	Slope
	Volume	Area
		Volume

Table 2.2: Mackinlay's guidance on quantitative/ordinal/categorical data representation

2.3 Visual (Programming) Languages

The term "languages" as used in our everyday life differs from the term languages used in information systems. One specific usage of this term is programming languages. Well known programming languages are for example C, Java or PERL. For all of these languages, the typical development style is textual, i.e. text-based. This means that the engineer will use a (simple/advanced) text editor to write code that will then be compiled and executed. But although the most used visual representation of programming languages is textual, visually different examples exist. This section focuses on these "visual languages".

2.3.1 Visual (flow) programming languages - Discourse into Scratch

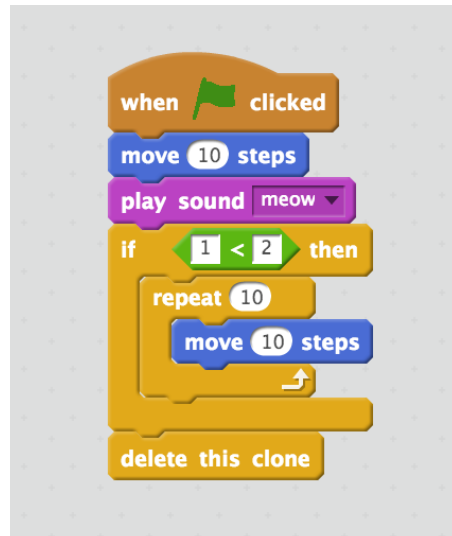


Figure 2.6: A simple, exemplary Scratch control flow that will move the character 10 steps, then play a sound, then given 1 is smaller than 2 move another 10 steps 10 times with finally being deleted.

Scratch is a visual flow programming language built by MIT for the purpose of making it playful to teach programming. Scratch has evolved into a cloud-based version 2, that allows teachers all across the globe to teach children the basic concepts of programming in a visual and easy to understand manner. In the visual language, elements and entities ("sprites") are dragged and dropped onto different "scenes". Most basic operations, such as for-loops, if/else-statements are available as pre-defined elements. Figure 2.6 shows a simple control flow created in scratch [48].

When being created, there is a visual feedback to the user of where elements can be dropped. This is done by providing visual cues in the form of highlights when hovering over/near the correct element(s). Scratch allows for creation of complex programs, such as programs visualizing planetary movements or chemical reactions such as photosynthesis. The underlying code is not visible at any time, so the user is "stuck" with the visual interface.

2.3.2 Types of visual programming languages

A visual programming language is a programming language that allows users to create programs within a visual context, i.e. using visual elements rather than writing code. This is usually done by using different elements and symbols on a canvas that interact with each other to create the control flow.

Typically, VPLs are split into control flow and data flow programming languages. Most of the currently available VPL are blocks-based programming languages which *"[...] are tangible programming languages that rely on the manipulation of artifacts"* [43].

Visual control flow programming languages Purposed for teaching

The most current example which is widely known is the previously introduced Scratch. Scratch is being accompanied by other similar implementations:

- Snap [7]: built on Scratch, Snap extends the former with first class lists, first class procedures, and continuations.
- Catroid [59]
- Alice
- Greenfoot
- Marama [26]

All these languages build on Logo, a block-based sequential programming language. Logo was developed in 1967 and is most well known for red lines being drawn during the program, resulting in computer graphics.

Scratch works with the LEGO concept: Only blocks that are supposed to fit on each other will fit. Blocks have specific inputs that will be shown to the user via a visual feedback when creating the code. This removes syntax errors, as only syntacticly correct blocks and combinations can be created. Put into the words of the creators: *"Scratch scripts are built by snapping together blocks representing statements, expressions, and control structures. The shapes of the blocks suggest how they fit together, and the drag-and-drop system refuses to connect blocks in ways that would be meaningless."* [48]

In Scratch, a concept called "sprites" relates to objects in other programming languages. Each sprite has it's own code - containing the building blocks of Scratch: Command block, function block, trigger block and control structure. With these building blocks, all other elements in Scratch are built. Through design of the block, users can see which parts will connect to each other. Blocks are indivisible - e.g. a control structure cannot be "written wrong" because it's a self contained element. Since inheritance is not available in Scratch, the language is object-based but not object-oriented.

Scratch has multiple visual functionalities that assist the user in creating programs and also motivate through playfulness:

- *Tinkerability*: Users can click and play around with modules as with for example mechanical components.

[7] http://snap.berkeley.edu/

- *Example by design*: Instead of simply showing blocks, each of them comes with example parameters to illustrate functionality.
- *Block autonomy*: Each block can be tested on its own. When clicked, the block shows the output of its run in a talk bubble.
- *Execution visibility*: When the code is running, visual feedback is provided to the user on which blocks are currently utilized. Users can also move in a step-by-step flow, similar to breaking points.
- *No error messages*: Most syntax errors are removed through the above stated LEGO concept. While errors can of course still occur, the user can play around instead of having to debug as parts of the program will even run with errors.
- *Showing hidden items*: Single variables (and lists), which are hidden at runtime in programming languages are shown through monitors on the stage.
- *Limit choice*: Available amount of Blocks in Scratch is always strived to be minimized so to not confuse users with too much choice. The authors of [48] write: "every command consumes screen space in the command palettes, so there is a higher "cost" to increasing the command set". This is achieved by a) keeping available amount of Blocks low and b) aggregate blocks into one, e.g. the scientific math function block with a drop down of available functions.

Block based visual programming languages
Blockly is a general purpose visual programming language that is extendible[8]: "Blockly is a web-based, graphical programming editor. Users can drag blocks together to build an application." [22] The main paradigm of Blockly is, as the name already gives away, building programs and control flows by combining different blocks. Scratch, introduced in the previous section, also belongs to the family of block based visual programming languages. Other examples are:

- Scratch
- Blockly
- StarLogo TNG: In StarLogo TNG, users can create 3D games and simulations - agents are controlled by the resulting block control structures.
- App Inventor: The software allows users to Android applications.
- Stencyl: Similar to App Inventor, Stencyl allows to create mobile iOS (and Flash) applications.
- TaleBlazer: A platform for creating multiplayer location based games for iOS and Android [46].

[8]https://code.google.com/p/blockly/

Topic based visual programming languages
OpenMusic is a visual programming languages that allows to visually create music. The building elements are boxes and connects, similar to Scratch but with the added part of connections (whereas Scratch realizes this via different block input/outputs) [8].

2.3.3 Visual data flow programming languages

Schaefer names LabVIEW (Laboratory Virtual Instrumentation Engineering Workbench), OpenDX and MeVisLab as specialized VDFPLs. [61]

The definition of a VDFPL is: "A visual data flow programming language (VDFL) contains visual, multi-dimensional objects for conveying semantics. Operation of the visual object is functional and the execution of objects is based on the data flow execution paradigm." [45] The author further goes on to list nine characteristics/function rules derived from the definition:

1. The operation of a node is functional.
2. A node is executed as soon as its inputs are populated by new data.
3. Data flows via data arcs as a stream of discrete data tokens.
4. A node can have zero or more input data ports. Respectively, a node can have zero or more output data ports.
5. If the input ports do not exist, the node is executed once as soon as the program executes.
6. An executed node produces new values for all its output data ports.
7. In order to be executed again, a node must receive new input values for each of its input data ports.
8. Each data port is attached to a single data arc.
9. Data arcs cannot fuse together, but a data arc can be branched into multiple data arcs containing a copy of the original data token(s).

2.3.4 Query language as programming language

While a query language lacks the necessary control structures (such as for-loops), different expanded query languages such as TSQL exist that introduce these necessary structures. A traversal of a graph can be seen as a single control flow by itself, with the application of filters representing if-statements and multiple calls of the same traversal element representing for-loops. Gremlin, with the concept of it's pipes (in Tinkerpop2) can thus easily be mapped to such a structure. This mapping helps in the creation of a visual query language for graph databases, as many principles defined in a visual programming language can also apply to such a VQL.

2.4 Visual Query Languages for Graph Databases

2.4.1 Overview

Many different approaches at creating VQLs for different applications have been suggested. A very recent approach at creating a visual query language has been done by Choi and Wong in Jan 2014: "VXQ: A visual query language for XML data". The authors propose a VQL for XML, based on the assumption that the complexity of XQuery for querying XML data, the W3C proposed standard query language, is leading to shortcoming. [17] visKWQL takes the keyword based query language (KWQL) and adds a visual layer to support users in the query formulation. The introduced visual layer enables the user to make queries more advanced by simple drag and drop actions, combinations of elements and color coding. [29] In "from a procedural to a visual query language for OLAP", authors Cabibbo and Torlone describe a multi-dimensional data model for OLAP and give both an algebraic approach as well as graphical approach. The authors go on to show that both querying approaches have the same expressive power. [11] In all of the above research, the approach of VQLs have been taken for simplicity and usability. Jiang, Mandel and Nandi have created a VQL for multitouch devices, thus coming from a different motivation. The language is transforming SQL queries into multitouch gestures, allowing for simple queries, aggregations, joins and more. [36]

2.4.2 Visual query language

Visual query languages fall into a category of visual programming languages - within it they fall under the subcategory of managing data in databases. The definition of visual query language is: "Visual Query Languages (VQLs) are languages for querying databases that use a visual representation to depict the domain of interest and express related requests. VQLs provide a language to express the queries in a visual format, and they are oriented towards a wide spectrum of users, especially novices who have limited computer expertise and generally ignore the inner structure of the accessed database." In the article of the definition, VQLs are separated into categories by their visual representation:

- Form-based: Especially suited to relational databases as the form can represent the underlying tables directly.
- Diagrammatic: Instead of representing the underlying data structure, diagrams serve to visualize concepts such as creation of a bridge between disconnected elements.
- Iconic: With iconic representations, both the objects and operations of the database are depicted.
- Hybrid: Utilizing some or all of the above.

Figure 2.7 illustrates the four different VQL types. The output of a visual query language doesn't necessarily fall under the VQL itself and has thus been disregarded in

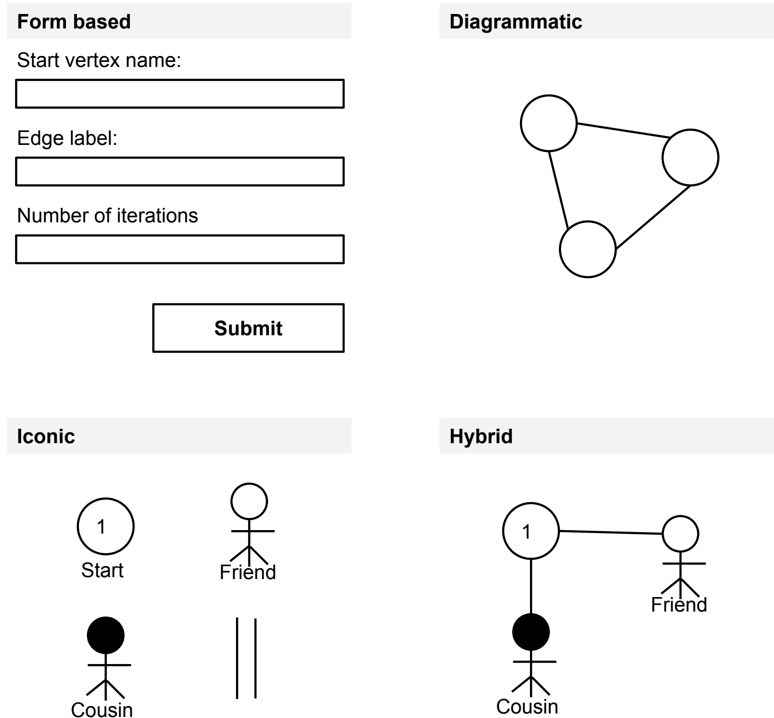


Figure 2.7: An illustration of different visual query language types, based on the definition of Visual Query Languages in [13].

most previous works done in this field.
The fundamental difference between a VQL and a textual query language is thus the query formulation. Often users are provided a way to browse through the data before going into formulation. Formulation is split into three categories:

- "By schema navigation": Moving from concepts of interest to other concepts of interests by specifying new conditions each time.
- Subqueries: Specifying the query via composition of partial results.
- Matching: Providing the structure of a possible answer. (e.g. Graph pattern matching)
- Range selection: Providing users with filters on multiple dimensions of the underlying data and allowing for specifying ranges within those filters.

User studies comparing traditional, textual query languages against VQLs are usually in favor of the VQLs for ease of use, but show downsides in power of expression and specific use cases [13].

VQLs have been created for many different purposes. The authors of [44] develop a visual language for modeling and executing traceability queries. They go beyond the layer of query formulation and abstract the underlying data from the query engine and model. This will be similarly achieved by using Apache Tinkerpop. Jin and Szekely [39] create a comic-like visual query language for temporal patterns. By mapping comic strip features such as characters, multiple panels, time features, etc. to the formulation of a query, pattern occurrences can be more easily retrieved.

A visual context query language is outlined in [67] but doesn't aim at visual query formulation but rather at using visuals as the query itself (images).

2.4.3 Existing VQLs for graph databases

Different research towards a visual query language for graph databases has already been conducted and several results have shown good performance and expressability. Popular early approaches are GraphLog [20], Query By Diagram* (QDB*) [2] and Paste-3 [42]. The prototype of GraphLog as in [20] doesn't support aggregation queries and is built on Smalltalk-80. Angelaccio, Catarci and Santucci [2] very early arrive at the conclusion that textual query languages are not a good fit for the non expert users. In their paper they introduce QDB*, a visual query language for recursive queries aimed at easy of use and compare their solution to GraphLog. The authors themselves state *"QDB* is mainly a navigational language on E-R diagrams."* Because of the recursion focus it may still be considered as a graph query language.

"Konduit VQB: a Visual Query Builder for SPARQL on the Social Semantic Desktop" introduces a VQL for RDF data, based on SPARQL queries. The authors introduce an approach similar to filters in Microsoft Excel. [28] Utilizing the same technologies, Hogenboom et al. present RDF-GL, a graphical query language (GQL) for RDF. They introduce some previous efforts, some of them being aimed at XML with others targeted at knowledge representations languages such as RDL/OWL. Graph databases like Neo4J allow for mapping from RDF to itself, this allows for this specific approach to be applicable here. The final result allows for specific SPARQL SELECT queries. The authors point out that *"not every aspect of a textual query language can be covered by symbols of a graphical query language"* [35] and [31].

Butler, Wang et al. [10] & [9] create a full graph database system (with MySQL as underlying database implementation) that aims at optimizing queries via visual queries. The user is presented with a GUI that is fully abstracted from the actual query logic (see Figure 2.8).

The authors use diagrammatic representation in query building by utilizing GraphLog. Users are presented with the structure of the underlying data, which is imported by the query interface. Nodes can represent either an abstract entity (less specific) or an attribute's atomic value (more specific). Edges can be defined via the already available relations or by specifying new relations. After the query has been specified, it is first translated into an XML representation and then further into a CORAL (see [51]) query program. Rather than evaluating the visual query formulation, the authors focus on the query optimization path through CORAL.

Figure 2.8: Graph database optimization via visual querying, simplified from [10] Fig. 2. GUI stands for Graphical User Interface and TGL is the TGL Translator, *"[...] consisting of the query translation component and the result translation component"*.

Clark [19] presents a visual query builder for Drupal[9]. With RDF modules in Drupal it's easy for publishers to expose the content as RDF, but still requires knowledge in SPARQL to consume it. One of the main challenges identified is the initiation of the query (e.g. writing a statement in SPARQL) which is solved by exposing a single point of interaction with drag-and-drop query functionality. Clark cites [14] stating that the visual query formulation enables inexperienced users to overcome the mental computation necessary in SPARQL. In [18], a demonstration shows how a visual module is used to build queries that are immediately translated into SPARQL queries. As soon as this is done, another Drupal module is used to specify the query output. The focus on the research lies within this second part.

Blau et al. [6] create QGraph, a visual language for both querying and updating graphs. *"A key feature of QGRAPH is that the user can draw a query consisting of vertices and edges with specified relations between their attributes."* The query language is pattern matching based. Bhowmick, Choi and Zhou [5] have developed VOGUE, a visual graph query processing framework that provides the user feedback on query creation time interactively. With this novel approach, query formulation and processing become dependent of each other. Queries are evaluated at building time, against a pre-computed action aware frequent index (A^2F) and action aware index (A^2I). With the implementation of the query engine the authors show that overall query time is reduced and even with the additional processing the response time grows gracefully with increasing database size. While the approach in [5] is aimed towards (large collection of) small and medium sized networks, in [34] the same approach is applied to large networks with the query engine entitled QUBLE.

2.4.4 Comparison of available VQLs and identification of needs

As the literature review shows, many different previous approaches for VQL for graph databases exist, which will now be compared. Before comparison, different attributes of VQLs are established.

Abstraction from graph database implementation
A concept of which its importance is very much linked to the use case is the abstraction of graph database implementation. This specifies how easy it is to replace the original

[9]see https://drupal.org/ for more information on Drupal.

database choice with another one (e.g. replacing Neo4J with GraphDB). Values of this feature are boolean, thus is an abstraction from the graph database implementation available or not.

Graph Model
Different graph models, as outlined in subsection 2.1.2 exist. This importance of this feature for a VQL again depends highly on the specific use case. While some utilizations might easily work in a basic graph model, other uses might rely heavily on the additional features a property graph model provides.

Bi-directionality
Most VQLs rely on a previously established query language and translate visual queries into this language or enable running queries from the visual context somehow in the underlying language. If a VQL is defined as bi-directional, it means that it's possible to translate a query both from visual to textual as well as vice-versa (i.e. from textual to visual).

Graph pattern matching
This attribute specifies if the VQL can match sub-graphs based on previously provided patterns, the more common approach of query languages for graph databases.

Graph traversal
In contrast to above, this attribute specifies the availability of graph traversals from the VQL.

Availability of graph measures
Can the VQL be utilized to calculate measures such as centrality measures?

Creation time feedback
Does the user receive any feedback on potential outcomes during the creation of the query? This feature is important on different levels, as outlined also in [5]:

- Idle time of the query processor is minimized (which normally is at least the time it takes for query formulation).
- The SRT can be improved significantly, which is the time from the user pressing the run icon to the time when the user gets the query results.
- Finally, the overall user experience is enhanced as relevant guidance and feedback during query formulation can be provided.

Utilized query language
If available - what known query language does the VQL utilize?

As established different research has been conducted towards creation of a visual query language for graph databases. Table 2.3 provides an overview of the different functionalities and goals of these query languages.

VQL	Abstracted from GDB impl.	Graph Model	Bi-directional visual queries	Graph pattern matching	Graph traversal	Availability of Graph Measures	Creation time feedback	Underlying query language
[20] GraphLog		Simple directed graph		×				G+
[2] QDB*		RDBMS			×			CLOSURE-OF
[5] Vogue	×	Simple directed graph		×			×	Vogue
[34] Quble	×	Simple directed graph		×			×	SPARQL
[19] SPARQL Views		Triples	×	×				SPARQL
[28] Konduit VQB		Triples		×			(auto-complete)	SPARQL
[31] RDF-GL	×	Triples		×				SPARQL
[6] QGraph	×	Property Graph		×				QGraph
[29] visKWQL		Triples	×		×	×		KWQL
[12] Cigales	×	Simple directed graph		×		×		Cigales
[16] Graphite	×	Simple directed graph		×				G-Ray
[38] GBLENDER	×	Simple directed graph		×			×	GBLENDER
Pygmalion Query	×	Property Graph	×		×		×	Gremlin

Table 2.3: Comparison of available query types in popular graph query languages.

In table Table 2.3, the eight different attributes identified are compared for the previously reviewed VQLs for graph databases. The table shows that most VQLs are not capable of graph traversal and focus on graph pattern matching. The underlying data model is usually RDF/Triples. This is already one of the major factor separating Pygmalion Query from other available solutions. Creation time feedback is seen in no other graph traversal VQL. Availability of graph measures, something often found in other visual tools such as Gephi[10] or NetworkX[11] is an interesting part of any tool/software dealing with network analysis, but was identified as not necessary for an initial visual graph traversal query formulation approach.

This table serves as the main identification for a solution as the proposed Pygmalion Query. Two of the primary features identified here are the combination of property graph model databases and traversal of a graph.

[10]http://gephi.github.io/
[11]https://networkx.github.io/

CHAPTER 3

Design

3.1 Requirements Gathering

To capture a design targeted at both novice and expert users, first requirements need to be gathered. This can be done in multiple ways, including looking at graph query examples or studying the query language and identifying the necessary features. Both of these approaches are viable and shall be both explored within this thesis. Graph query examples can be (by no means an exhaustive list) found by searching through specific scholar articles focusing on graphs, looking through online forums to find problems, inspecting the Gremlin documentation and simple trial and error with a given database. For each identified example query, specific features and attributes need to be collected to in the end come up with a list of needed features for Pygmalion Query. Following this approach, in this section the requirements for the VQL are first gathered from query examples and then by studying the query language itself. After this, the requirements are mapped to necessary features and attributes in the query language. The following sections in this chapter are then focused around a design to display these. The requirements in the following sections will be enumerated with $R_{100}, R_{101}, ..., R_{1NN}$ for the requirements from example queries and $R_{200}, R_{201}, ..., R_{2NN}$ for requirements from the documentation for easier referral. $R_{300}, R_{301}, ..., R_{3NN}$ are the requirements coming from studying other (visual) query languages and coming from the state of the art research.

3.1.1 R_{1NN} Requirements by example: Gathering graph database queries

The main sources to gather examples of graph database queries were scholarly articles[1], the Gremlin users google group[2] and other online resources such as Stackoverflow[3]. Each

[1] e.g. via https://scholar.google.com/
[2] https://groups.google.com/forum/#!forum/gremlin-users
[3] http://stackoverflow.com/

of the found example query is attributed with a numerical identification, some meta data such as a name, the source and a description of the expected result and query. Furthermore, the valid gremlin query is provided. Finally, the requirements for the query and potential features and attributes are gathered.
Before getting into specific examples, there are a few requirements that can be identified without the need for actual queries. When a user has a potential result in mind, the graph schema isn't necessarily visible to him/her. In this case, before even starting to build a query, the user needs to know the available properties of vertices and edges (= labels). So a first requirement identified is the need for knowing the properties and labels of vertices and edges. Going even further up the query building, the user might not even know the available graphs that he or she can query. This provides the starting point for list of requirements:

- R_{101}: Allow for selection of available graphs.
- R_{102}: Show available vertex properties and labels.
- R_{103}: Show available edge properties and labels.

The following list of example queries are purposely displayed on a single page each.

1: Facebook graph search example 1

Taken from http://wrightimc.com/2013/08/12/the-giant-list-of-facebook-graph-search-queries/

The above specified link specifies numerous queries that are possible within the (relatively) new Facebook graph search feature - a natural language query engine within Facebook. One of the interesting queries outlined in the linked list to queries is:

Korean restaurants in x-city

which translates to the following Gremlin query:

Code Listing 3.1: Query example 1

```
g.V()
 .has('type', 'restaurant')
 .as('a')
 .out('food_type')
 .has('name', 'Korean')
 .back('a')
 .out('located_in')
 .has('name', 'x-city')
 .back('a');
```

This relatively simple query already shows a lot of features Gremlin has to offer. The query starts by selecting all available vertices - a identified requirement is thus to give the user a starting point from where to start traversing. Following this, a context is saved into a variable, allowing to later retrieve the starting point saved again. The next steps allow the user to filter until finally a result is received. A requirement can thus be to display the result. Furthermore, users need to define the specific filter values, which can change with each step (e.g. each time back() is called the available vertices might have changed and thus the amount of available values). This is a more complex requirement - adapt available context based on place in the traversal. Listing the identified requirements:

- R_{104}: Provide a starting point in the query.
- R_{105}: Allow to save context in the query.
- R_{106}: Display the results of a query.
- R_{107}: Adapt available properties and labels (similarly to R_{102} and R_{103}) based on the context.
- R_{108}: Allow filtering of available data.

2: Facebook graph search example 2

Taken from http://wrightimc.com/2013/08/12/the-giant-list-of-facebook-graph-search-queries/

As example 1, this query is based on Facebook graph search:

people who are not my friends that work at x and like x

which translates to the following Gremlin query:

Code Listing 3.2: Query example 2

```
g.V()
 .as('a')
 .out('knows')
 .filter{ !it.get().value('name') == 'johannes' }
 .back('a')
 .out('works_at')
 .has('name', x)
 .back('a')
 .out('likes')
 .has('name', x)
 .back('a');
```

Very similar to example 1, this query is just a series of filter steps starting from all possible nodes. Additionally to the steps in example 1, the user is selecting the inverse of some filter result. This additional step is relatively complex and requires the user to jump out of the general flow of data into the scope of the filter. A requirement is thus to allows inverse filtering and providing sub-context. With more steps being made available the need for an arbitrary amount of steps also emerges.

- R_{109}: Allow inverse filtering.
- R_{110}: Provide sub-context for more advanced Gremlin steps.
- R_{111}: Provide arbitrary amount of simple/complex Gremlin steps.

3: Querying recursive friendships

Based on http://stackoverflow.com/questions/9486201/graph-traversal-how-do-i-query-for-friends-and-friends-of-friends-using-greml

---

In a question asked at Stackoverflow, the user wants to query for friends of friends (of friends ...). This query can be expanded into a more interesting query:

Calculate degree of separation for all my friends of friends recursively

which translates to the following Gremlin query (assuming vertex 1 is me):

Code Listing 3.3: Query example 3

```
g.V(1)
 .repeat(__.out('friends_with'))
 .until{ !it.get().out('friends_with').hasNext() }
 .path()
```

This example, as short it might seem, is a powerful expression. From a starting vertex, all friends_with paths are traversed recursively until no further vertices are found. After this, the path how each of the vertices have been found is given. Each return array size is the degree of separation. Different than in the relatively easy out() or has() statements, the settings of the repeat and until step can be arbitrarily complex. The output is different then just vertices or edges and thus also needs guidance for the user.

- R_{112}: Make complex settings available to the user.
- R_{113}: Handle different results.

4: Calculating a weighted group count

Based on https://groups.google.com/forum/#!searchin/gremlin-users/groupCount/gremlin-users/H_JQ___XyvY0/iq6pxEsCQ_kJ

Coming from another online forum, the official gremlin-users group: Gremlin allows to group elements together and count their occurrence via groupCount(). Furthermore it's possible to weigh each of these counts by some other attribute. In the question, the goal was to sum up the edge weights and then multiply these by the group count. For easier understanding the query has been adapted a bit:

Calculate the group count weighted by the sum of edge weights.

which translates to the following Gremlin query (assuming vertex 1 is me):

Code Listing 3.4: Query example 4

```
g.V(1)
 .inE()
 .as('e')
 .inV()
 .group('a')
 .by{ it.value('name') }
 .by{ it.count() * it.e.values('weight').sum }
```

More Gremlin traversal steps are introduced within this example. Further, the context of the traversed elements change - starting with V() the user has vertices as data flow. With inE() this changes to be edges to be changed again with inV() to vertices. Finally the data flow element is changed again into grouped values, a hash map with names as keys and the vertices with same name as array of values.

- R_{114}: Show user the current data flow elements.
- R_{115}: Adapt available steps based on data flow elements.

3.1.2 R_{2NN} Requirements by documentation: Inspecting the Gremlin query language

Looking at examples gives a great start finding requirements, but through the limited quantity and thus scope one might not be able to gather all needed features and attributes. As Gremlin allows for specification of arbitrary steps, not all possible queries might be fully covered within the visual query language but rather made available through a complex query step. R_{201}: Allow any Gremlin query to be created by allowing arbitrary steps. Identifying requirements by documentation[4] focuses on finding further features and attributes that might not be covered from the examples. Furthermore, potential non-requirements are identified as well - which either don't fall into the scope of the thesis or may not be relevant in the visual query context. For better structure, this section will loosely follow the structure as provided in the documentation.

The Graph
Within Gremlin, 3 main elements are handled: The graph, vertices and edges. Except for the identifier of the graph and some basic meta information such as available amount of vertices and edges, no further features of the graph shall be provided within Pygmalion Query. R_{202}: Only basic graph level features are provided. Vertices provide one of the main data flow elements. Vertices have properties which keys need to be available to the user (as per previous requirements). Within Tinkerpop3, properties may also have properties - which shall not be part of this thesis. R_{203}: Meta-properties are not a requirement. Property keys are string values, but values can be different formats. When selecting or filtering on a property, the user should thus be aware of which format is to be chosen. R_{204}: Provide the user with allowed choices when selecting/filtering property values. Parallel to vertices, the same requirements apply to the edges of the graph.
Gremlin is not only a query language for retrieval of data, but can also be utilized to modify the underlying graph. As Pygmalion Query is designed foremost for the retrieval of data, graph transformations are not a requirement. R_{205}: Graph transformations are not a requirement. Also related to this, Gremlin allows for import of multiple different graph formats (e.g. GraphSON, a JSON based format). While this is not necessary, the export of results in this or another format can be helpful to anyone trying to retrieve data. R_{206}: Allow to export results in a standardized file format.

The Traversal
In the beginning of this chapter, a short introduction into the available Gremlin steps was given. Gremlin provides the user with the most needed functions, but also allows to specify lambda functions at any given time in the query, which can extend the provided functionality. R_{207}: Allow the user to see what functionalities are not available in Pygmalion Query but are potentially possible. The traversal itself is the most important concept within the query language and has multiple features/attributes to it. It can be split into the elements "flowing" in the traversal, the data, and the different steps

[4]Referring to the documentation found at http://www.tinkerpop.com/docs/3.0.0.M4/

transforming the data, the functions. Within the data, there is the type, the quantity and the available attributes depending on the type. Within the functions, there is the input and output data type, the quantity transformation, the input/output function type. The following list of requirements captures these elements:

- R_{208}: Show the type of the data at different query points.
- R_{209}: Show the quantity of the data at different query points.
- R_{210}: Show the available attributes of the data at different query points.
- R_{211}: Show the input and output data type of a function.
- R_{212}: Indicate the quantity transformation of the function.
- R_{213}: Guide on the possible input and output functions.
- R_{214}: Provide further context/helpful text on each function.
- R_{215}: Help the user in configuring the function.

The steps Gremlin provides are descriptive already, but for the visual query language a different metaphor might be necessary: R_{216}: Provide a metaphor on top of Gremlin steps that helps the user to utilize them. As these steps are each powerful in its own way, it is necessary to help the user choosing steps - this can be done in multiple ways (e.g. categorization of steps, suggestion of steps). R_{217}: Guide the user with the selection of steps.

Advanced Gremlin Functionality
The Gremlin documentation shows the high complexity of the whole system - from distributed server architecture, OLTP and OLAP to driver implementations in different languages. The whole language is highly linked to each of the different parts that were more separated in Tinkerpop2. For Pygmalion Query, most of these things lie out of scope, as the user when querying for data doesn't mind the underlying architecture. The main requirement for this is thus - R_{218}: Allow the user to create queries without having to dive into the architecture.

3.1.3 R_{3NN} Requirements from the state of the art research

Many requirements have been identified both through the documentation as well as the studying of query examples. These were mostly focused around the concrete querying part done through Gremlin. Pygmalion Query is foremost a user interface, which has more implications. These implications for example can be "convenience" requirements that help the query building, but are not immediately linked to Gremlin. Among these are:

- R_{301}: Allow any Gremlin user to easily use Pygmalion Query.

- R_{302}: Allow the user to store and load built queries.
- R_{303}: Show the created query so that users can also execute the raw query when necessary.
- R_{304}: Allow to run a query once completed.
- R_{305}: Allow collaboration on query creation.
- R_{306}: Allow creation of queries from different devices.
- R_{307}: Give the user warnings.

3.2 Requirements Identification

Having identified requirements from example queries, the official Gremlin documentation and the state of the art research, in this section these requirements are summarized and ranked. The potential feasibility is identified, and a decision towards implementation is made. In the following section of this chapter, these requirements are then conceptually outlined from necessary architecture to visual design.
To judge the necessity and feasibility, each requirement is rated on a scale of 1-5 for necessity (1 unnecessary) and 1-5 for feasibility (1 unfeasible). Furthermore a third score from 1-5 is added to assign a value to potential user helpfulness (1 not adding helpfulness). Minimum thresholds for each requirement are discussed after the rating, and a discussion on the final identified requirements and their categorization. A sum of the scores is not provided due to its doubtful meaningfulness (e.g. a low necessity score shouldn't be hidden by a high feasibility score).

Necessity	Feasibility	Helpfulness	Requirement
5	5	5	R_{101}: Allow for selection of available graphs.
5	5	5	R_{102}: Show available vertex properties and labels.
5	5	5	R_{103}: Show available edge properties and labels.
5	5	4	R_{104}: Provide a starting point in the query.
3	2	4	R_{105}: Allow to save context in the query.
3	4	5	R_{106}: Display the results of a query.
3	2	5	R_{107}: Adapt available properties and labels.
5	3	5	R_{108}: Allow filtering of available data.
3	4	4	R_{109}: Allow inverse filtering.
2	1	3	R_{110}: Provide sub-context for more advanced Gremlin steps.
3	3	3	R_{111}: Provide arbitrary amount of simple/complex Gremlin steps.
3	2	4	R_{112}: Make complex settings available to the user.
4	3	5	R_{113}: Handle different results.
4	3	5	R_{114}: Show user the current data flow elements.
5	3	4	R_{115}: Adapt available steps based on data flow elements.
2	2	3	R_{201}: Allow any Gremlin query to be created by allowing arbitrary steps.
4	5	4	R_{202}: Only basic graph level features are provided.
5	5	3	R_{203}: Meta-properties are not a requirement.
4	3	5	R_{204}: Provide the user with allowed choices when selecting/filtering property values.
5	5	2	R_{205}: Graph transformations are not a requirement.
3	3	5	R_{206}: Allow to export results in a standardized file format.
3	1	3	R_{207}: Allow the user to see what functionalities are not available in Pygmalion Query but are potentially possible.
4	3	5	R_{208}: Show the type of the data at different query points.
3	2	5	R_{209}: Show the quantity of the data at different query points.
4	3	5	R_{210}: Show the available attributes of the data at different query points.

Necessity	Feasibility	Helpfulness	Requirement
4	5	4	R_{211}: Show the input and output data type of a function.
4	4	5	R_{212}: Indicate the quantity transformation of the function.
4	4	5	R_{213}: Guide on the possible input and output functions.
4	4	4	R_{214}: Provide further context/helpful text on each function.
4	3	5	R_{215}: Help the user in configuring the function.
3	3	4	R_{216}: Provide a metaphor on top of Gremlin steps that helps the user to utilize them.
3	2	5	R_{217}: Guide the user with the selection of steps.
5	4	5	R_{218}: Allow the user to create queries without having to dive into the architecture.
5	4	5	R_{301}: Allow any Gremlin user to easily use Pygmalion Query.
3	2	4	R_{302}: Allow the user to store and load built queries.
4	4	4	R_{303}: Show the created query so that users can also execute the raw query when necessary.
5	3	5	R_{304}: Allow to run a query once completed.
2	1	3	R_{305}: Allow collaboration on query creation.
2	2	3	R_{306}: Allow creation of queries from different devices.
5	4	4	R_{307}: Give the user warnings.

Table 3.1: List of requirements and their rating in necessity, feasibility and helpfulness.

Table 3.1 shows the scoring on the 3 discussed metrics. The table also highlights the selected requirements based on a simple threshold:

$$(Necessity \geq 4 \; and \; Feasibility \geq 3) \\ or \\ (Necessity \geq 3 \; and \; Feasibility \geq 3 \; and \; Helpfulness \geq 4)$$

This easy formula can be translated into: If something is necessary and feasible, include it. If it's somewhat necessary, feasible and at least somewhat helpful, also include it. All green requirements compute to true in the above formula and are colored green. These requirements are selected and will be categorized for the design of the query language in the following section.

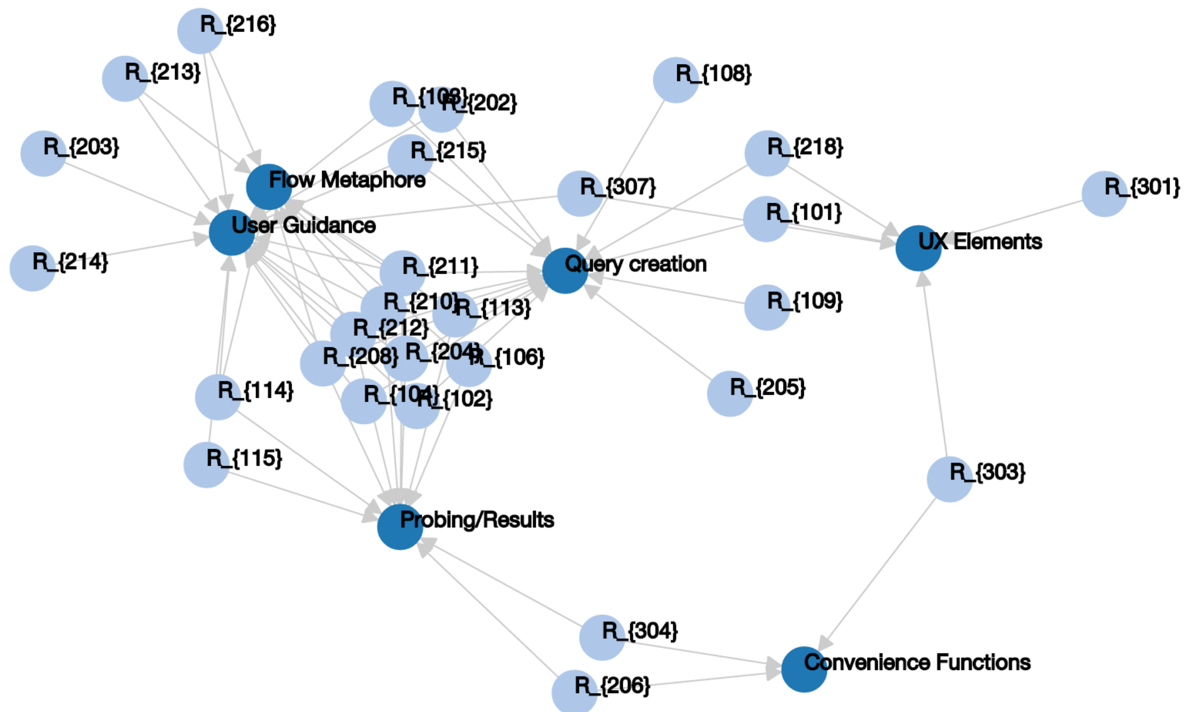


Figure 3.1: The chosen requirements from Table 3.1 and their matching to the identified categories.

Following the identification, scoring and selection of requirements summarized in Table 3.1, features are now surfaced out of requirements. Features are first defined at a higher level, the category. Each category will then be then discussed with the identification of the lower level features. Each category has requirements in table Table 3.1 related to it.

- UX Elements
- Flow metaphor
- Query creation
- User Guidance
- Probing/Results
- Convenience functions

The graph in Figure 3.1 shows the matching of the different requirements to a category. The main categories are the Flow Metaphor, User Guidance, Query Creation

and Probing/Results, whereas UX Elements and Convenience functions are only related to little amount of requirements.

As these requirements and categories are highly interconnected, the design for the features within category is also dependent on the design of the others. Based on this, the following sections are structured in a manner that accounts for this: First the probing/results will be discussed, as it is more of an architectural design that allows other categories to draw from it. After this, the Flow Metaphor will be explained and designed. After this, the Query creation will be the main chapter for the Blockly feature design, followed by additional features that build on this for the User Guidance. Finally, the UX Elements and Convenience Functions are designed.

3.3 Features Design

3.3.1 Probing/Results

Probing/Results related requirements

R_{115}: Adapt available steps based on data flow elements.
R_{206}: Allow to export results in a standardized file format.
R_{106}: Display the results of a query.
R_{113}: Handle different results.
R_{114}: Show user the current data flow elements.
R_{304}: Allow to run a query once completed.
R_{204}: Provide the user with allowed choices when selecting/filtering property values.
R_{208}: Show the type of the data at different query points.
R_{210}: Show the available attributes of the data at different query points.
R_{211}: Show the input and output data type of a function.
R_{212}: Indicate the quantity transformation of the function.

The simplest form of probing and results within Gremlin can be achieved by simply building a query in the Gremlin console and running it each step. As the addition of any traversal step results in another valid query, this is a correct approach. With Pygmalion Query, the shortcomings of this usage scenario are tackled. While running the query within the console and waiting for the output is in the form of query - result, the design for the VQL of this thesis introduces probing. Probing allows to wrap the query in a template that returns a specific result type, e.g. the quantity of the current traversal. Additionally, with a separation of the user interface with the query backend, the user doesn't have to wait for any results but can continue with the query building with any available result being pushed to him or her asynchronously. The main factors in this category are:

- Probing: wrapping queries in templates.
- Results: Handling results of different steps and displaying them.

- Asynchronicity: Sending and receiving queries without interrupting the workflow.
- Processing: Using the available results to guide users and display warnings.
- Context: Use any available results as context for further query creation steps.

Figure 3.2 shows the timeline for asynchronous probing process. The user is creating query at time 1 (Q_1). Before the query is being sent to the server, it is wrapped in a probing template. As the user continues to create the query, the results reach the interface once done computing at time 2 (with the query at state Q_2).

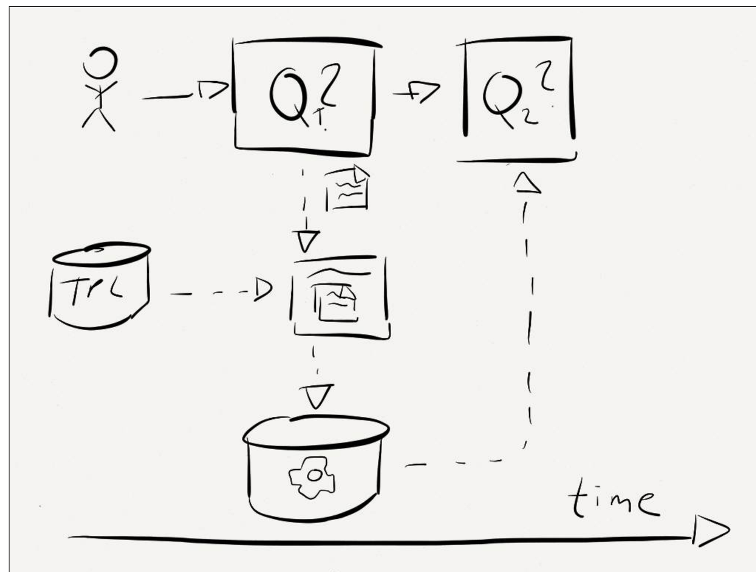


Figure 3.2: A process diagram for asynchronous, template driven probing.

An important part of this whole process is the intent: Either the user send the query for processing actively, or an implicit probing is happening automatically. Pygmalion Query utilizes both of these functionalities. Each new part of the query requires information to help the user in being guided - and thus needs more context to be drawn from the database. Each query step will thus be assigned it's own probe(s) to retrieve the necessary context.

3.3.2 Flow Metaphor

Flow Metaphor related requirements

R_{106}: Display the results of a query.
R_{113}: Handle different results.
R_{114}: Show user the current data flow elements.
R_{213}: Guide on the possible input and output functions.
R_{216}: Provide a metaphor on top of Gremlin steps that helps the user to utilize them.

R_{208}: Show the type of the data at different query points.
R_{210}: Show the available attributes of the data at different query points.
R_{211}: Show the input and output data type of a function.
R_{212}: Indicate the quantity transformation of the function.

An important part of the query building for graph traversals is the awareness of the flowing data that passes through different functions and thus changes in type and quantity. There are thus two main factors to the flow:

- Function
- Data

Both of these will change with different steps. The layers of change for function are: availability of function, input/output type and available settings. For data, the type and the quantity will change with the steps. In addition to that, functions also have more context such as the name, a more detailed explanation and potentially the low level Gremlin steps that one function aggregates. With Figure 2.5 and Figure 2.7 in mind, Table 3.2 outlines the necessary flow parts to be visualized as well as the design method. The third column (Availability) gives an indication if the information necessary to display the part is available before runtime or needs implicit probing.

Flow part	**Design method**	**Avail.**
Function identifier	Iconic representation: Symbol for ease of understanding, name and description in additional text.	
Function input/output type	As the identifier, the input/output type is best displayed with iconic representation.	
Function settings	The function settings can range from simple selection of an attribute to complex statements. A form based approach is chosen to account for the range of complexity. Furthermore, depending on the available information, more graphical representations of the available attributes might be chosen.	
Data type	Similar to the function input/output type, the data type should visualize the incoming/outcoming data and what transformation is happening. Different elements within the data are to be colored differently. The shape of the available data objects points to the type.	
Data quantity	The quantity of the data is visualized with a sanky-diagram approach, thus the size of the line (as also seen in the visualization by Charles Minard in Table 2.1.	

Table 3.2: Function and data flow design methods.

Utilizing the probing as outlined in the previous section, the necessary information not available before runtime can be queried at query creation time. Figure 3.3 shows a design draft to visualize the elements as outline in Table 3.2.

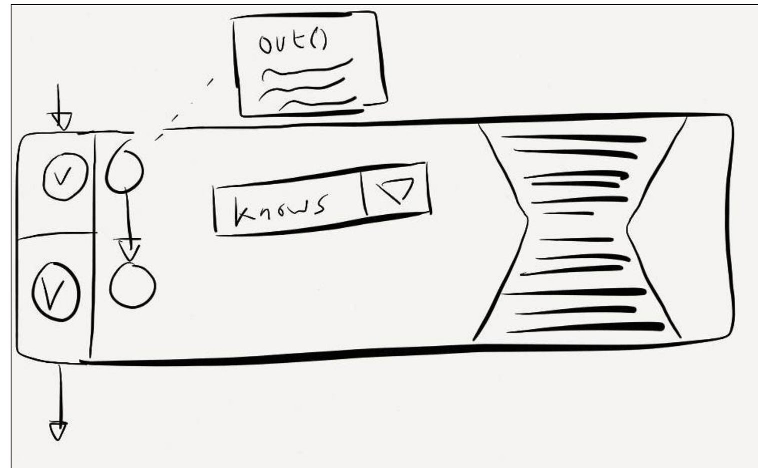


Figure 3.3: The design draft for visualizing the flow category and its attribute vectors.

The design draft shown in Figure 3.3 is independent of the actual function of such a functional block. An exact Gremlin step, such as *out*() can be mapped to this design as well as multiple steps combined as one, for example a functional not-out block: $as('a').out(X).filter\ !it.get().value(Y) == Z\ .back('a')$ with 3 to be specified attributes (the edge label, the key and value of the non-vertex). As the design allows for both the simple Gremlin to functional block mapping as well as the more complex approach, Pygmalion Query is not limited to either, but rather allows for both.

As in the draft above, with different blocks there may be different information and context to convey to the user once the non-static information has been received. A concrete example, continuing with the more complex functional block above:

Flow part	**Values**
Function identifier	Not-out, crossed edge as icon
Function input/output type	Any function outputting/receiving vertices
Function settings	3 attributes: edge label, vertex key/attribute
Data type	Vertex
Data quantity	Expect to either decrease or stay the same

Table 3.3: The not-out functional block.

Table 3.3 specifies the available information. In Figure 3.4, three potential visualizations are shown. The block width stays the same for all functional blocks, as only this way it is ensured that continuous visualizations (such as the sanky diagram in (*i*) and (*iii*)) are possible. While in (*i*), the sanky diagram shows the cardinality as a whole, in (*iii*) each of the potential keys and influences on limiting on them is shown. In (*ii*), the data visualization is linked to the quantity of the available edges, a simple histogram giving the user more context on what each selection can lead to.

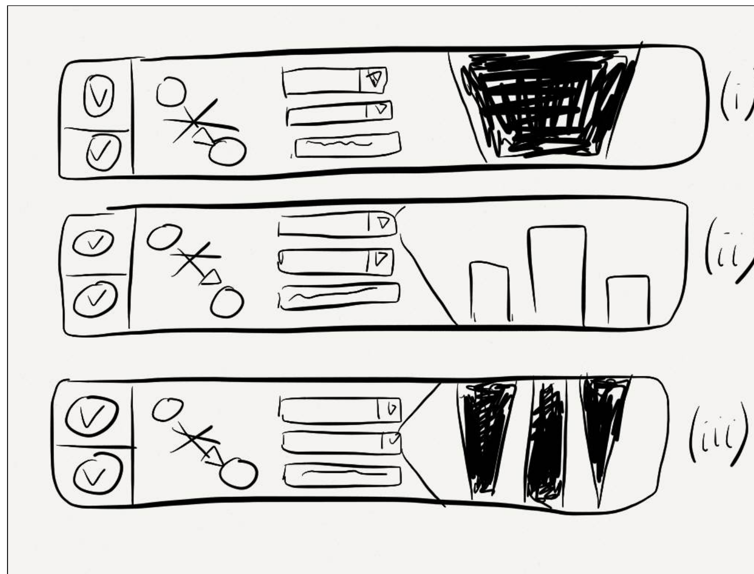


Figure 3.4: Some visualization examples for the potential not-out functional block.

As shown in these more concrete examples, the flow metaphor is covered extensively through these blocks: both functional as well data flows are integrated. Dynamic data visualizations allow to give more context, while static functional elements help the user to keep the overview.

3.3.3 Query Creation

Query creation related requirements

R_{101}: Allow for selection of available graphs.
R_{102}: Show available vertex properties and labels.
R_{103}: Show available edge properties and labels.
R_{104}: Providing starting point in query.
R_{108}: Allow filtering of available data.
R_{109}: Allow inverse filtering.
R_{202}: Only basic graph level features are provided.
R_{204}: Provide the user with allowed choices when selecting/filtering property values.
R_{205}: Graph transformations are not a requirement.

R_{208}: Show the type of the data at different query points.
R_{210}: Show the available attributes of the data at different query points.
R_{211}: Show the input and output data type of a function.
R_{212}: Indicate the quantity transformation of the function.
R_{215}: Help the user in configuring the function.
R_{218}: Allow the user to create queries without having to dive into the architecture.

In it's simplest form, the query creation is just a combination of the previously outlined functional blocks. This is the very similar to the Gremlin console approach, in which one simply attaches (by writing) new functional blocks to the query. While in the Gremlin console the selection for different elements is essentially the keyboard and its characters, Pygmalion Query provides the user with a set of available elements that can be dragged and dropped.

Most requirements are covered within the flow metaphor and the functional blocks. The query as a whole consists of the combination of different functional blocks, as shown in Figure 3.5, leads to the full picture. Functional blocks can also be available combined as an entity, allowing to template specific parts. This leads to having simple functional blocks, complex functional blocks and templates which can consist of $2, ..., N$ of the two.

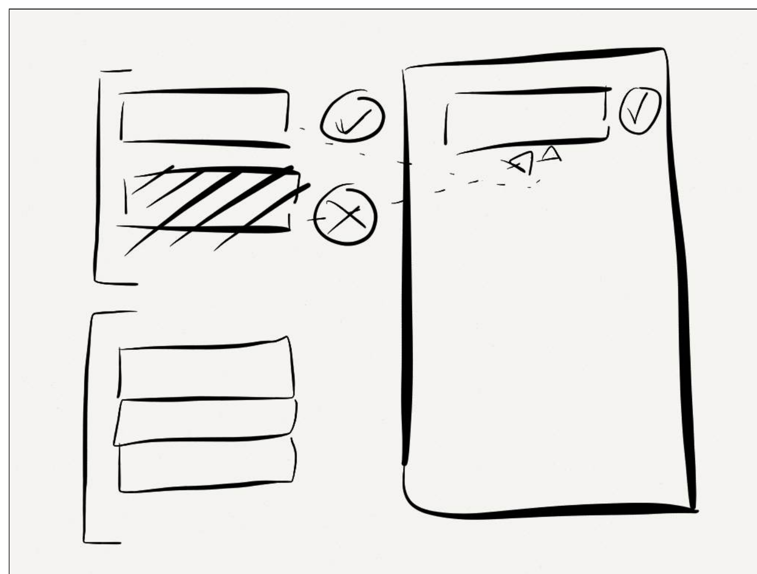


Figure 3.5: A graphical representation of the functional blocks for query building.

The syntactical correctness of the query is provided through the functional blocks. Semantical correctness provides a more complex task, but is being achieved to some extend by implicit query creation time probing, data visualization and user guiding/limiting when selecting/connecting new blocks (as shown in Figure 3.5 by not allowing a specific block to connect to another).

3.3.4 User Guidance

User Guidance related requirements

R_{203}: Meta-properties are not a requirement.
R_{213}: Guide on the possible input and output functions.
R_{214}: Provide further context/helpful text on each function.
R_{216}: Provide a metaphor on top of Gremlin steps that helps the user to utilize them.
R_{102}: Show available vertex properties and labels.
R_{103}: Show available edge properties and labels.
R_{104}: Providing starting point in query.
R_{115}: Adapt available steps based on data flow elements.
R_{202}: Only basic graph level features are provided.
R_{204}: Provide the user with allowed choices when selecting/filtering property values.
R_{208}: Show the type of the data at different query points.
R_{210}: Show the available attributes of the data at different query points.
R_{211}: Show the input and output data type of a function.
R_{212}: Indicate the quantity transformation of the function.
R_{215}: Help the user in configuring the function.
R_{307}: Give the user warnings.
R_{114}: Show user the current data flow elements.

User Guidance heavily draws from the previously outlined visualizations. This category aims at providing the user with steps, processes, signals and warnings during the query creation. Most of the information necessary to do so is only available at creation time and thus dynamic rather than static. The two main parts of this category are:

- Guiding on static information
- Guiding on dynamic information

Before going further into each of these parts, the available methods to guide the user are highlighted. So far it was identified that the query is created of functional blocks that provide context on their function as well as data. Probing serves as context pipeline, drawing more information from the query server during the creation process. The "playing field" of user guiding is thus focused around the canvas on which functional blocks are provided and dropped. The shape of these blocks is taken up by the context each block provides and fixed to allow for continuos visualizations. Potential methods can be identified as:

1. Highlighting of elements: changing color, changing line width, giving notes on elements.
2. Activating/Deactivating elements: Limiting the provided user options.
3. Attaching of elements: Using functional block typing to disallow/allow connections.

4. Giving context through layover fields: With the possible space on the functional blocks, layovers provide space for further information.

5. Providing overview: Helping the user to position himself/herself within the traversal.

Each of the above provided methods can be used within different parts of the query building. Element 1, highlighting of elements is used with the results of the asynchronous result processing, to bring a users attention back to a specific functional block or section. A simple traffic light highlighting of the context retrieval for each block is utilized. Also related to probing/results, providing information on the context retrieval can be shown with a layover field.

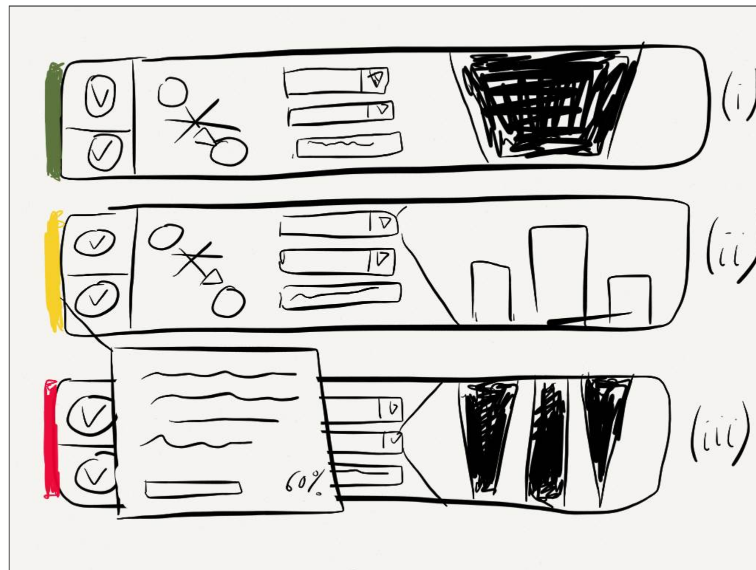


Figure 3.6: Based on Figure 3.4, utilizing highlights and layover on functional blocks for user guidance.

Activating/Deactivating and attaching of elements was shown in Figure 3.5 already. Another part of this is the providing of selection fields rather than free text fields within functional blocks.

Element 5, providing overview, allows the user to visualize the traversal in a similar manner as (*iii*) in Figure 3.4 by highlighting the chosen path. As the size of the graph should not define the suitability of the visual query language, the space limitations need to be taken into account. [60] names scrolling, overview+detail, distortion, suppression and zoom and pan (e.g. Google maps on a phone) as possible methods to overcome space limitations. Together with the space limitations are also potential limitations in keeping the whole graph in memory while displaying - only overcome by lowering the amount of available details. A recent example of overview+detail can be seen in the text editor

Sublime Text[5], which provides an overview over the current text file in birds eye view on the right side, together with highlighting the current position. Figure 3.7 shows a screenshot from the official website.

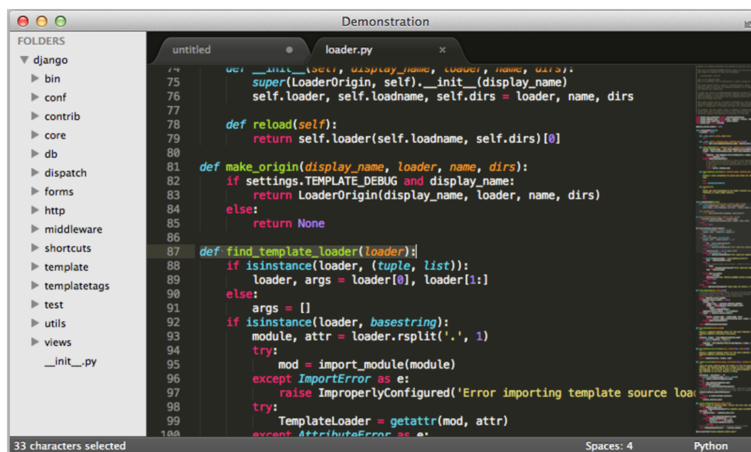


Figure 3.7: Screenshot of the text editor "Sublime Text", taken from the official webpage. The birds eye view pane on the right is an example of an overview.

To lower the amount of detail shown in an overview for space considerations, a more abstract view on the graph must be provided - Figure 3.8 provides two drafts on how to achieve something like this.

Both drafts shown in Figure 3.8 show only the main elements of a graph/results of a graph query, i.e. vertices, edges and scalars. By aggregating on vertex and edge type level, an important detail can be shown that will be still scalable in many graphs. By using distortion around the current position, more details such as cardinalities can be displayed through line width. Figure 3.9 shows such an implementation.

As this overview builds on *(iii)* in Figure 3.4, it can be included in the dynamic right side of a functional block. It is to be decided on implementation time if this approach is suited or the overview might be taken out of the functional blocks in its own "overview pane".

3.3.5 Convenience functions

Convenience functions related requirements

R_{304}: Allow to run a query once completed.
R_{206}: Allow to export results in a standardized file format.
R_{303}: Show the created query so that users can also execute the raw query when necessary.

Little requirements relate to convenience functions. These are especially linked to the execution of the query and the export of results. With the previously established

[5]See http://www.sublimetext.com/

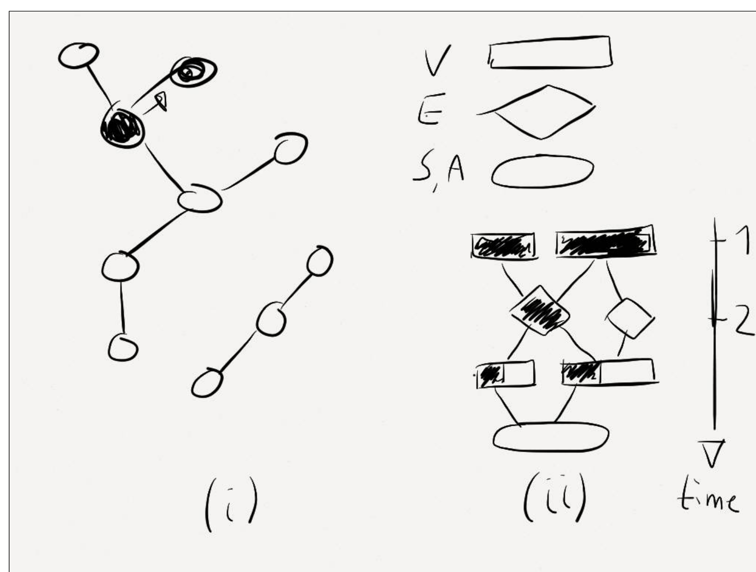


Figure 3.8: Drafts of the overview view. The left side provides a generic disconnected graph that can be highlighted to show the traversal. On the right aggregations of the main types (Vertices, Edges, Scalar) and a schematic overview of the traversal.

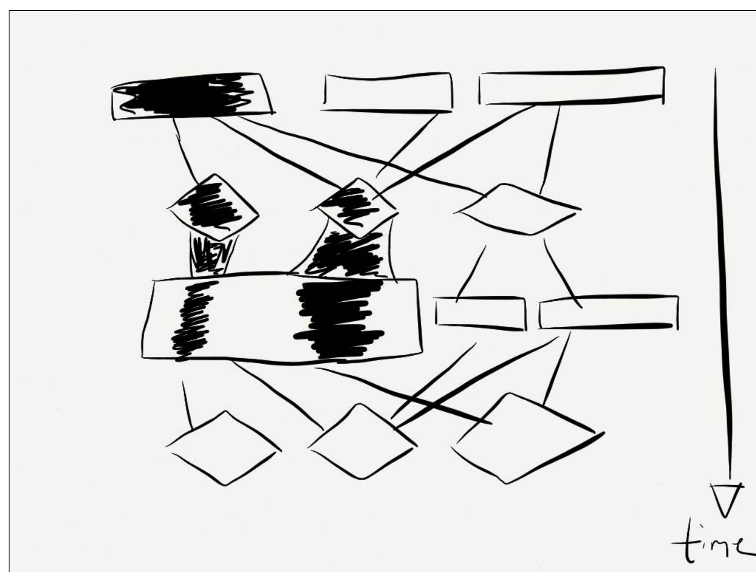


Figure 3.9: This figure builds on Figure 3.8, (*ii*), a distorted, more detailed view of the current context.

probing/results, queries are already run against the query server. To run a query once completed is thus only a matter of providing the user with a means of sending it. A potential hurdle in this is the size of the result, which needs to be limited in size. With

the usage of probing templates, this is achieved. The display of results coming from these runs are discussed in the next category, UX Elements.

An important part of running the query within the templating functionality is to allow the user to see what limitations on a query result might be imposed. At the same time, with the availability of this functionality users can also make use of this to query for specific types of results. Code listing 3.3.5 shows the query to run, a count template and the final query sent to the server.

Code Listing 3.5: Results querying with templating

```
Query:
g.V(1)
 .out('knows')

Result template:
{{query}}.count()

Result query:
g.V(1)
 .out('knows')
 .count()
```

To capture the different types of results retrieved, result/probe functional blocks are introduced. With these, the same user guidance as with normal blocks can be shown (i.e. connecting only to specific outputs, highlighting of result state, etc.). Also, results can be shown in an overlay to account for the space limitations. Potential probing blocks within Pygmalion Query are:

- Count probe: $query.count()$
- Unique count probe: $query.dedup().count()$
- Group count probe: $query.groupCount()$
- Table probe: $query$
- Graph probe: $query.path()$

3.3.6 UX Elements

UX Elements related requirements

R_{218}: Allow the user to create queries without having to dive into the architecture.
R_{301}: Allow any Gremlin user to easily use Pygmalion Query.
R_{303}: Show the created query so that users can also execute the raw query when necessary.
R_{307}: Give the user warnings.
R_{101}: Allow for selection of available graphs.

The main part of Pygmalion Query is the canvas on which the user can create the query through functional blocks. Apart from this canvas, more elements can be provided.

- Resulting code: Showing the resulting Gremlin code to easily copy/paste.
- Visual query code: While the saving and loading of visual queries is not within the scope of the thesis, the visual query code can still be displayed to be copied and pasted.
- Overview pane: As discussed in the User Guidance category, a potential overview pane might be provided.
- Results view: Additionally to the overlay provided on blocks, a results view can help visualize larger results, e.g. in a table.
- Settings and controls: Making Pygmalion Query adaptable without code (e.g. setting the path to the server) and providing additional functionalities such as clearing the query.

Being able to immediately access the code that is being created by the visual query language allows to seamlessly switch between code and user interface. The results view is necessary to account for larger results, e.g. a table of all found vertices and their properties.

CHAPTER 4

Implementation

"The computing scientist's main challenge is not to get confused by the complexities of his own making." - **E. W. Dijkstra**

Chapter 3 outlined the design of Pygmalion Query. Needs were identified and mapped to available visualization methods. This chapter will give an account of the implementation of Pygmalion Query, along with the justifications on the framework decisions.

4.1 Overview

The screenshot in Figure 4.1 shows an overview of the implementation of Pygmalion Query. In it, the main query creation view is shown in which a query on a patents database is created. Pygmalion Query is a web based visual query language which acts as a client to connect (via websocket) to a Gremlin Server instance. The major part of the user interface is covered by the query interface, with some elements such as websocket configuration, generated query code and results outside of it. While this screenshot shows the surrounding browser elements, further figures will not do so.

To display the results of a query, users can switch between query interface and a results view. Figure 4.2 shows an example of such a result view, visualizing a node-link chart for a specific vertex, including it's immediate neighbors and the edges leading to them.

4.2 Technical Details

In this section, I will outline the architecture and technical details of Pygmalion Query. First, a high level architecture overview will be given, showing all elements in the implementation. After this, an overview of the available frameworks is provided with a

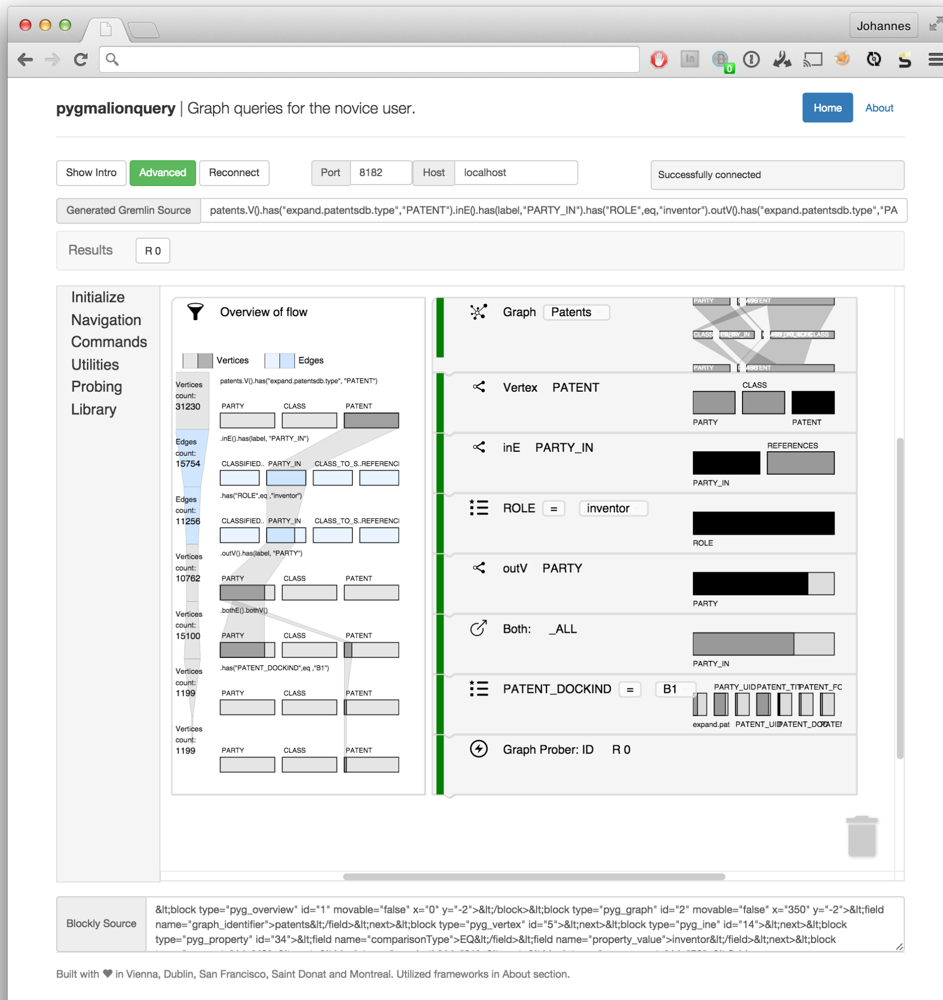


Figure 4.1: A screenshot depicting the implemented Pygmalion Query with an exemplary query built.

justification for the choice. Following this section, a closer look at the structure of the implemented VQL is provided.

4.2.1 Architecture elements of Pygmalion Query

As previously pointed out, the main part of pygmalion is the query formulation (see Figure 4.1). The design in chapter 3 also requires some further parts to make Pygmalion Query work. The five main elements are:

Figure 4.2: A screenshot depicting the result view of Pygmalion Query.

- Query formulation
- Code generation
- Query construction
- Server communication
- Result processing

Query formulation allows the user to drag and drop previously outlined blocks onto a canvas and gives the user the possibility to adapt specific settings for the block. Also an overview of the current state of the data flow is provided. Figure 4.3 provides a schematic view of the parts included in this architectural element.

The query formulation is the only visible part to the user of the Pygmalion Query architecture. Each block generates a specific part of the query (code) to be sent to the server. This means an interpreter needs to run in the background that takes the query canvas and its blocks and transforms it into a machine readable query. The user may or may not look at the generated code. Figure 4.4 gives a schematic overview of the code generation.

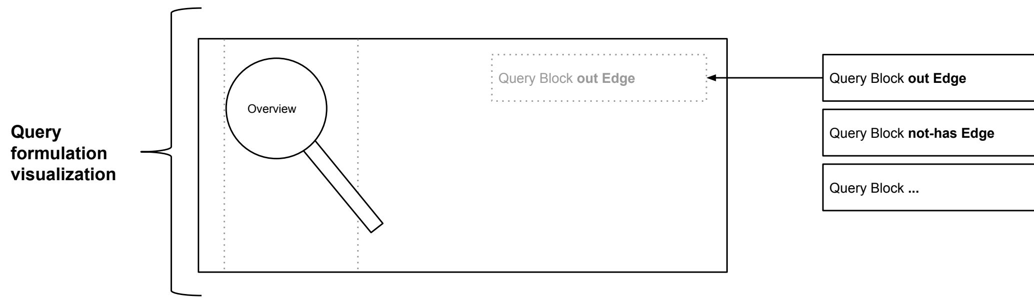


Figure 4.3: A visual representation of the schema of the query formulation part of Pygmalion Query.

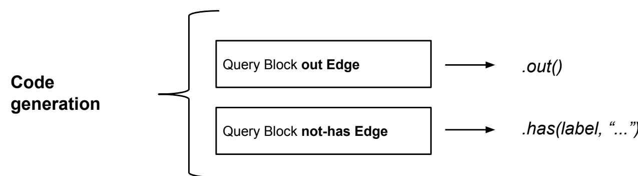


Figure 4.4: A visual representation of the schema of the query code generation.

With the concept of probing and different result blocks (see Figure 3.2), the query often isn't sent "as-is" but rather wrapped in additional query code. This code generation allows for templating for more functionality. Figure 4.5 visualizes this concept.

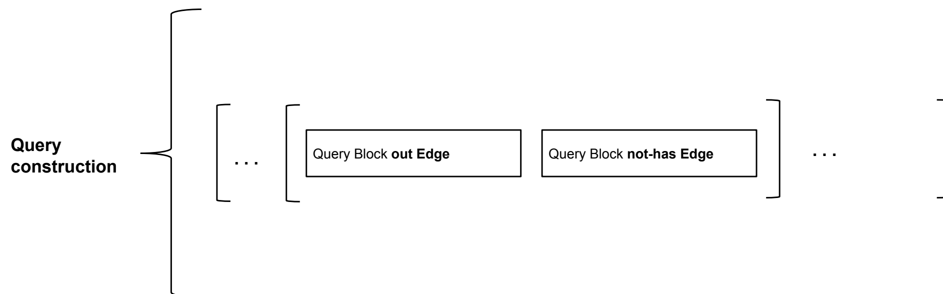


Figure 4.5: A visual representation of the schema of the query code construction.

With queries constructed, the server communication will handle sending queries via websocket connection to be executed. With the availability of a websocket interface directly from Tinkerpop, this doesn't require any further middleware. Figure 4.6 shows the schema for this element.

Finally, the server returns results of different kinds (e.g. a set of vertices, a set of edges, etc.) and needs to be handled. Different visualization methods can be utilized and

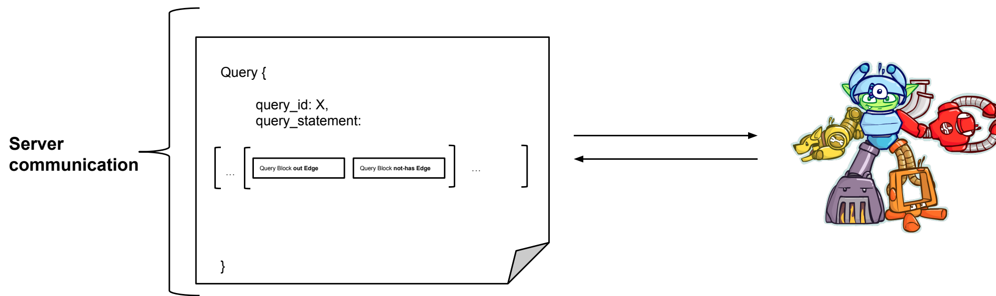


Figure 4.6: A visual representation of the schema of the server communication.

might need further querying to ensure the user sees the correct result. Figure 4.7 shows the concept: The server returns a set of vertices as result. As the user is expecting a full node/link chart, further queries are sent to the server to retrieve the necessary edges for this.

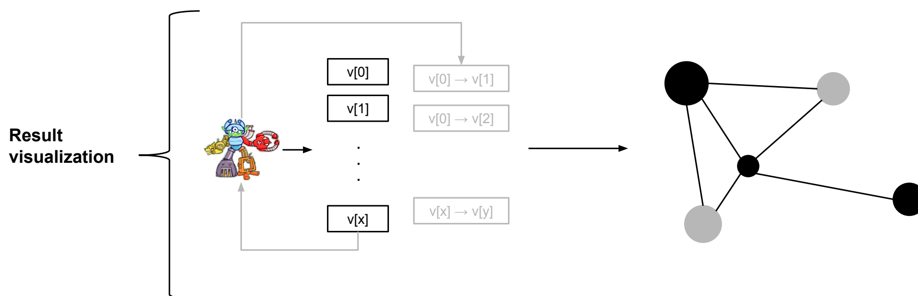


Figure 4.7: A visual representation of the schema of the results visualization.

4.2.2 Utilized frameworks within Pygmalion Query

Each of the elements of the previous section might require different frameworks/libraries to be put into implementation. The following extended list of architecture elements gives an overview of the minimum required elements.

- Query formulation: Data flow programming language and data visualization
- Code generation: Communication to query formulation and templating
- Query construction: Templating
- Server communication: Websocket connection
- Result processing: Data Visualization and query processing

Visualization frameworks

The query formulation and result processing elements have data visualization needs, while the formulation itself also needs a way to capture the data flow programming principle with the designed blocks. A few possible web visualization libraries exist. These can be split into two groups: general visualization libraries and chart purposed graph libraries. For the creation of graph traversal queries within the Gremlin pipeline method, any library will be possible. To display temporary/final results, a library that supports displaying of property graphs is necessary. The following list gives a few option for both groups:

- General visualization libraries
  - Raphaël JS (http://raphaeljs.com/)
- Chart purposed visualization libraries
  - D3.js (http://d3js.org/)
  - Sigma.js (http://sigmajs.org/)
  - VivaGraph (https://github.com/anvaka/VivaGraphJS)
  - Springy.js (http://getspringy.com/)
  - JavaScript InfoVis Toolkit (http://philogb.github.io/jit/)
  - Vega - building on D3 (http://trifacta.github.io/vega/)
  - Vis.js (http://visjs.org/)
  - Arbor.js (http://arborjs.org/)
- Visual programming languages
  - Blockly
  - Scratch

While each of the above mentioned libraries has its strengths, the nature of Pygmalion Query has requirements not matched by all:

- User Interaction (i.e. Drag and drop, hover behavior, etc.)
- Snapping of objects
- Graph drawing algorithms
- Displaying of multiple attributes for vertices and edges

The use case for query building by the user is very different from the displaying of temporary/final graph results. For freedom in building an interface, Raphaël serves the best purpose, not being limited to drawing data charts. Furthermore, the chart libraries serve a good purpose when trying to display graphs - which goes beyond the scope of

this thesis and has been researched extensively. The class of available general purpose visual programming languages are the best viable option for Pygmalion Query as they are maintained for different projects. Blockly, as introduced in subsection 2.3.2, allows to create new elements in a simple manner as outlined in Code Listing 4.1.

Code Listing 4.1: Example of Blockly element creation

```
Blockly.Blocks['text_length'] = {
  init: function() {
   this.setHelpUrl('http://some/url');
   this.setColour(160);
   this.appendValueInput('VALUE')
      .setCheck('String')
      .appendField('length');
   this.setOutput(true, 'Number');
   this.setTooltip('Returns number of letters ...');
  }
};
```

Blockly also provides a block factory with user interface, which allows to build code as in Code Listing 4.1 by using Blockly itself[1]. This already shows how adaptable Blockly is. Furthermore, Blockly comes with more features helpful to Pygmalion Query:

- Realtime collaboration: Blockly comes pre-built with the Google Drive realtime collaboration framework and makes it possible (when implemented in such a way) to allow multiple users to work on the same design in realtime.

- Generators: Blockly has built in Generators for Javascript, Python and Dart programming languages. The Generator interface structures the design-to-code translation and makes it easy to add new interpreters.

- Storage: Blockly supports storing the current blocks either via Google cloud storage or in a local browser session out of the box[2].

As Blockly only provides a way to formulate a query but has no data visualization capabilities, more libraries are necessary to realize the query formulation process. As for example seen in Figure 3.3, the query formulation requires a high inter-operability between the query formulation and the data visualization library. In [7], the authors describe D3.js as *"an embedded domain-specific language for transforming the document object model based on data"*. The focus of the library lies on "Compatability, Debugging, Performance". [37] also praises D3.js as *"easy to learn and is deeply tied to standard web technologies of HTML, CSS and Javascript"*. Finally, D3.js provides a wide range of features, such as the directed force graph layout[3].

[1]see https://blockly-demo.appspot.com/static/demos/blockfactory/index.html

[2]See https://developers.google.com/blockly/installation/cloud-storage.

[3]See this example http://bl.ocks.org/mbostock/4062045

Templating and server communication frameworks

With Blockly and D3.js, the query formulation, code generation and most of the result visualization is possible. To wrap these elements in a web-based user interface, AngularJS is used as the Model-View-Controller framework. AngularJS provides functions to communicate with the middle layer, to save user interactions, to display different pages etc.. To connect to the Gremlin Server, a websocket connection needs to be established. AngularJS doesn't support this in its standard version. Members of the Tinkerpop community have already created a Javascript client to interact with the Gremlin Server, which is utilized in the implementation by building a wrapper as an AngularJS service[4].

4.3 Important Features

In this section, the implementation of Pygmalion Query will be outlined in detail. The following structure will be followed: First, the query creation with Blockly is explained. Necessary adaptions to Blockly provide a transition to data visualization, going into the utilized D3.js functionalities. After this, a short explanation of the templating, querying and server communication follows. The result construction and visualization will serve as the the part chapter of this section.

4.3.1 Query formulation with Blockly

As outlined in the design of the VQL, the query formulation is done by using a set of blocks that connect together to create a continuous data flow/graph traversal. New blocks in Blockly were created (see Code Listing 4.1 for an example) that either map to specific Gremlin steps or aggregate multiple steps into one. The user is provided with different choices depending on the current state of the query. Figure 4.8 shows the creation of a query from the beginning. First, the user is presented with a pre-loaded block allowing to select the available graphs. After the graph has been selected, an overview of the graph schema is shown and more options appear in the menu to the left. The user can now drag and drop the next block to connect with the first one, where Blockly will give a visual and auditive feedback of a successful connection. Further categories and block options appear after that.

Blocks have two states, comparable to the class/object model in object oriented programming. In the beginning, they are not instantiated and can thus only provide a very basic level of information. Figure 4.9 shows an example of this concept: On the left side, an uninstantiated block is shown that only shows some basic name, symbol and tooltip explanation. On the right side, the same block is instantiated and populated with data that has been retrieved from the server, based on the current context.

The user is guided in this process - if the block doesn't attach to the previous block, it will not be instantiated and thus not change anything about the current context.

[4]See https://github.com/jbmusso/gremlin-client

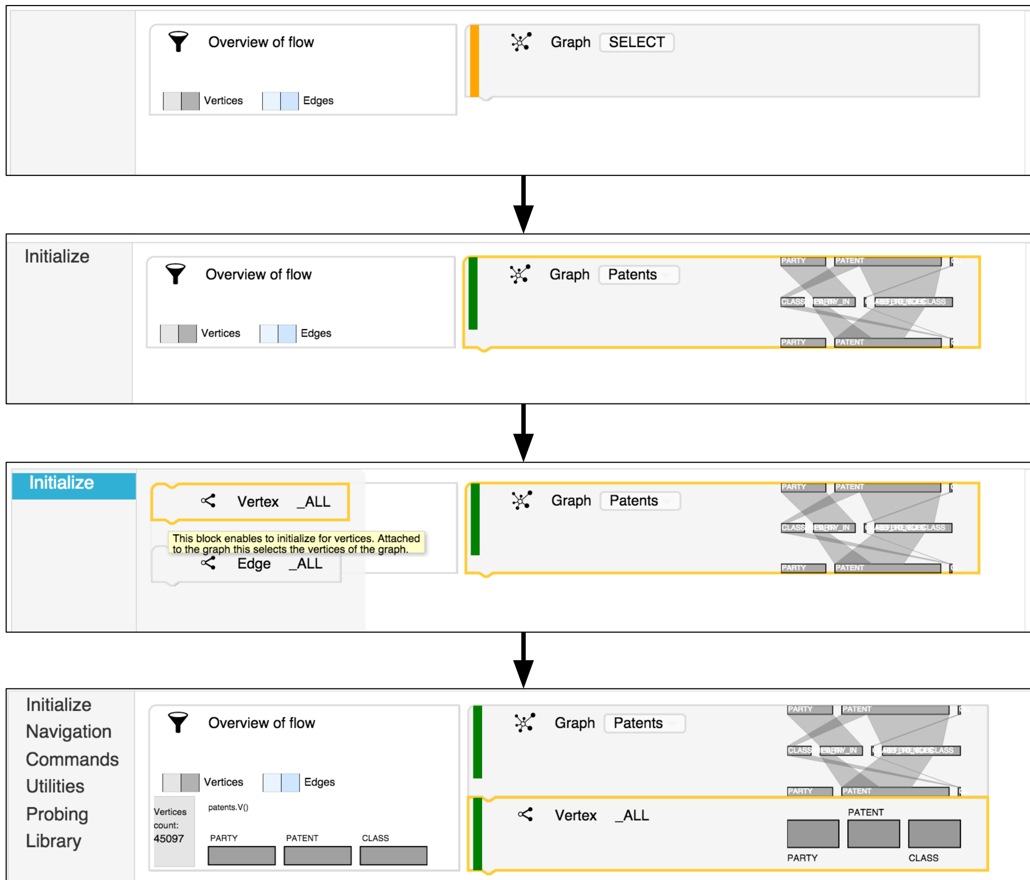


Figure 4.8: A step by step view of the first actions taken to creating a query with Pygmalion Query.

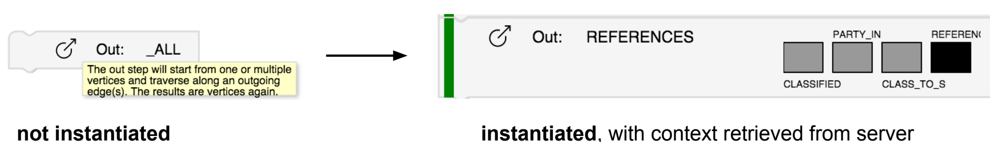


Figure 4.9: An un-instantiated blocks and an instantiated block.

An instantiated block is enriched with D3 visualization that helps to better convey the possible settings. The available configurations of a block depend on the context. Figure 4.10 shows the same instantiated block twice, but with different context. While the first block shows four possible options to select, which are all filled completely, the second block (with context B) gives less options and the options are not fully filled. In

this case, context A is before some further filtering in which all vertices are available. Context B has already some vertices filtered and thus has less options and not the whole spectrum of each available option.

Figure 4.10: A comparison of the same block in two different contexts.

A detailed account on how to retrieve the available options is provided in the section on templating, querying and server communication. Generally, each added block that has also been initiated will be passed on to the Pygmalion Query internal BlockHandler (see section A.1). The BlockHandler will differentiate the different block types and retrieve the necessary information from the server by using templated queries.

4.3.2 Visualizing the overview

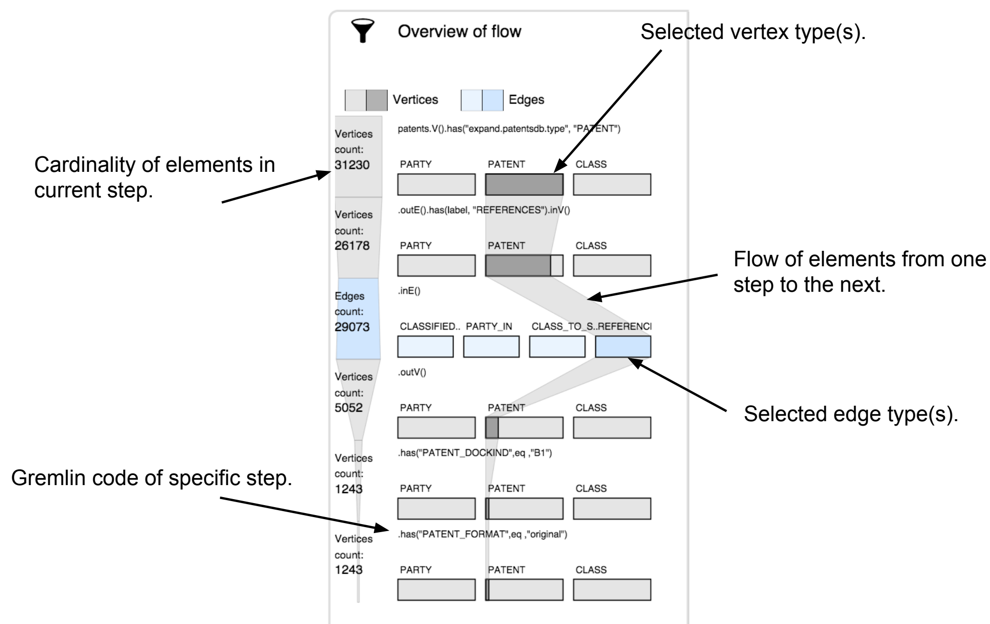


Figure 4.11: The Pygmalion Query overview explained.

In Figure 3.8, a design for providing an overview of the current query flow was identified. This "birds-eye" view on the traversal process still needs to be able to be linked to the different steps which is achieved by aligning the overview part to its respective block. Technically, the overview is a fixed block within Blockly that is being updated

with the query flow. Figure 4.11 gives an explanation of the different elements within the overview.

The overview is split into two parts: the left part only shows the cardinality of elements flowing, with the color choice giving an indication of the type of elements. The cardinality is both visualized as diagram with the width relating to the percentage of the maximum of elements overall in the query and also as text indicating the absolute values. The right side of the overview gives a more fine grained visualization. The types of elements are still separated via the color. Additionally, the split by label is provided for each step. The cardinality for each label is displayed by filling the rectangle relating to a specific label. The flow from one step to the next is visualized in a similar manner to the cardinality on the left. Data flowing from one label to another will have the width transform as available. Finally, each step is also given the underlying code.

4.3.3 Templating, querying and server communication

While Gremlin itself works as a query creation and then submission, Pygmalion Query is sending a lot of more queries implicitly to the server that the user isn't aware of or has to handle. In the previously explained visualization steps, a lot of data was shown (e.g. the cardinality of the available elements within each step) that needs to be retrieved. Furthermore, results need to be retrieved and displayed if requested by the user.

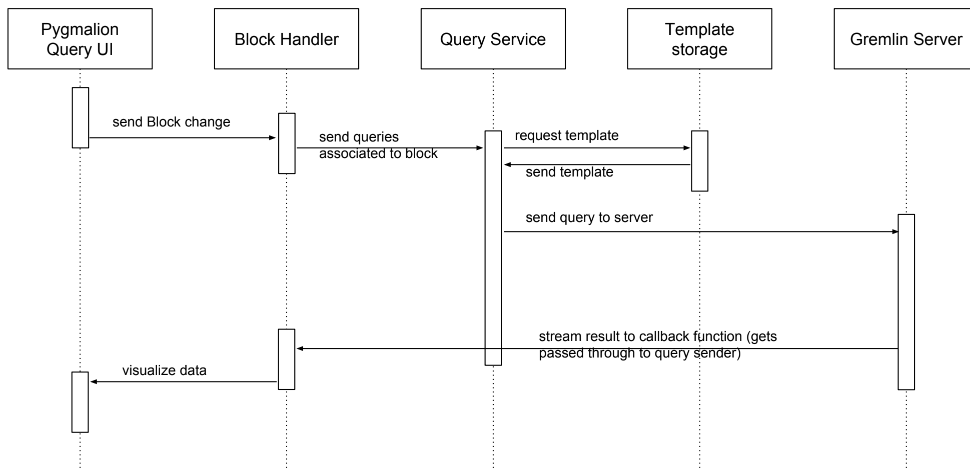


Figure 4.12: The flow from query formulation changes to retrieving context data and visualizing it.

To handle the load of extra queries, Pygmalion Query is communicating via an AngularJS service that wraps the available Gremlin Javascript client. Each query consists of an ID, a query object, a template name and a callback function that retrieves the resulting data. Figure 4.12 shows how the call flow is between the different elements for retrieval of necessary data.

The templating part in this flow is handled via the $interpolate function within AngularJS [5].

Code Listing 4.2: Example of a query template

```
ELEMENT_CONNECTION: '{{query}}
               .dedup().as("a"){{part2}}
               .dedup().as("b")
               .select("a", "b").by(label).by(label)
               .groupCount()',
```

The code snippet in Code Listing 4.2 illustrates the query template principle. In this example, two parts are filled in, query and part2. This query serves as the query to find the connections between two different steps as illustrated in the overview. With this templating, users don't rely on the underlying database to support the *label* property as defined in Tinkerpop graphs, but can specify any property as the label.

For convenience, the connection host and port to the server can be set on the user interface.

4.3.4 Result construction and visualization

While Pygmalion Query is mainly focused around the query building, which is sending implicit queries constantly to support the user with options and overview, there are explicit blocks that the user can use to request results. These probes, as introduced in subsection 3.3.4, allow the user to retrieve (partial) results at any given moment during the query formulation process. Figure 4.13 shows an example of a query that has multiple probes during different stages. Each probe can produce multiple results (when something about the query is changed above the probe) which are all stored and selectable in the menu above the query canvas.

Probes are essentially the same as any other block, but the produced Gremlin code is empty for the final query. The underlying data the block handler requests from the server is a template (or multiple templates). In the case of the graph probe (as shown), the full node/link diagram is retrieved. This entails getting the available edges/vertices and then querying for the missing elements. Two exemplary templates are shown in Code Listing 4.3. The BOTH_VERTICES templates will retrieve the vertices on either side of an edge, with a limit on 30 vertices. The VERTEX_ARRAY template will retrieve all vertices which IDs are in a specified array.

Code Listing 4.3: Templates for the graph probe

```
BOTH_VERTICES: '{{query}}.bothV().dedup().limit(30)'
VERTEX_ARRAY: '{{graph}}.V({{vertex_array}}).dedup()'
```

[5] Something that ECMAScript 6 now provides out of the box http://tc39wiki.calculist.org/es6/template-strings/

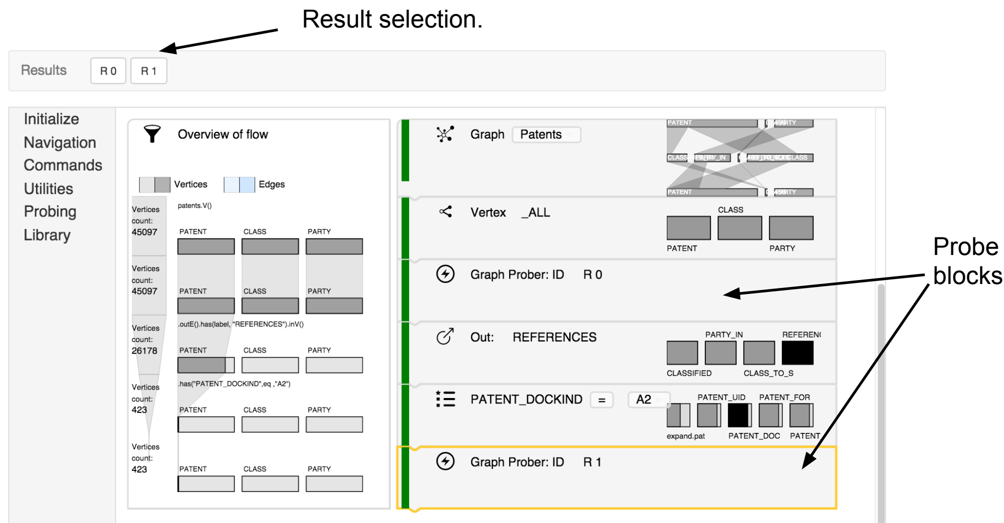


Figure 4.13: An example of having multiple probes within a query flow.

The user can seamlessly switch between query formulation and results view, allowing to probe for current results and adapt/continue with the query based on the knowledge gained.

4.4 Summary of the implementation

In this chapter, the implemented Pgymalion Query was introduced. Pygmalion Query is a web-based visual query language, that utilizes D3.js for visualizations and Blockly to create the block-by-block flow with its identified closeness to visual data flow programming languages. The user interface and the controlling of the different views is being done by AngularJS, a MVW (Model-View-Whatever) framework (with the Whatever part left open to be decided by each implementation, such as Controller, Presenter, etc.).

The whole implementation is done in Javascript, running completely on the client side. This is enabled through the available websocket connection in the standard Apache Tinkerpop Gremlin Server. The workflow for the user of the final visual query language is as simple as dragging and dropping blocks onto a canvas for them to connect. Blocks are then populated with data from the server, giving the user aforementioned semantic context to continue with the query formulation.

For the most part, any utilized framework was left untouched. To realize some of the more intrusive features required for the visualization part within Blockly, some default settings had to be changed and some functions adapted.

CHAPTER 5

Evaluation

5.1 The Evaluation Settings

An important part of providing evidence to prove or disprove the hypotheses of this thesis is the evaluation. Different approaches exist to do so, which have been utilized in previous developed VQLs.

- **User study:** In [39], the authors evaluated the comic-strip query language Query-Marvel against another, form-based query language PLForm. A total of 26 people participated by going through an online training on the task and then completing those tasks with both query languages in randomly selected order.
- **Case study/User Study:** The authors of [44] evaluated their study by first evaluating the VQL against a set case study and then later having 18 participants complete the query process. In addition to the simple user study in [39], the participants also rated their own IT related knowledge and answered questions on the study. This approach was needed due to the more domain specific knowledge required.
- **Expert Review:** As noted in [64], expert reviews may have an edge over user experiments when it comes to high-level cognitive tasks. The authors write "A few usability experts can find a large percentage of a system's usability problems.". [1] further state that "[...] experts can comment on usability issues while users can point out small problems related to tasks".

The research question of this thesis split into diverse hypotheses. That's why a two-fold approach was chosen: To evaluate Pygmalion Query, first an expert review was conducted. As Pygmalion Query tries to tackle a complex task, specifying graph traversal queries on potentially unknown graphs, two different experts reviewed both the design as well as the implementation. After the expert review, changes to Pygmalion Query

were made to account for the feedback given. With this changed implementation, a small comparative user study was conducted.

This chapter is structured into three sections. In the first section, the review of the design of Pygmalion Query by the User Interface experts is outlined. Links to the implementation will be drawn by the author. In the second section, the review of the implementation is discussed. The experts actively used Pygmalion Query and spoke out loud, leading to insights on the general usability as well as understanding of the advantages/disadvantages of specific features in Pygmalion Query. In the final section, the key take-aways from the expert review are summarized and potential actions are proposed to make Pygmalion Query more usable.

5.2 Expert Review

The two reviewers are both visualization experts, with different focus:

- **Expert 1**
  Main fields of expertise: Usability and Human-Computer Interaction. Furthermore a background in psychology with a focus in cognitive science. This expert is close to research in visualization and has some programming experience.
  Querying/Graph knowledge: Has some querying experience both in relational and graph databases - but only declarative languages such as SQL or Cypher.

- **Expert 2**
  Main fields of expertise: Information Visualization. Background in humanities (philosophy, sociology, psychology). This expert is close to research in social network analysis.
  Querying/Graph knowledge: No querying or programming experience.

5.2.1 Expert review of design mockups

In this part of the review, the experts were shown the mocks of Pygmalion Query as shown in chapter 3. While these mocks provide a good overview of the main features of Pygmalion Query, the design was adapted during the implementation phase. This allows to draw conclusions already to see, if any features that are not outlined in the design might have made it into the implementation due to the authors own usability needs discovered.

Visual encodings
The main visual encodings are discussed to be (stacked) bar charts. While this might be okay as first glance, absolute numbers are necessary to asses the size of the query results. The only coloring in the mockups is the traffic light coloring, which has a clear meaning for technicians but might not have it for other users.

Overview
Expert 2 states that he would always like to start with a representation of the schema. Also, he states that it's very important to use different shapes to differentiate between vertices and edges. The final Gremlin code is visible to the user, Expert 2 would also like to see the first dozens of results for an overview. Expert 1 further states that it might be too much information for an untrained user. The wording of vertex/edge might be too technical - Expert 1 suggests using node/link.

Query formulation and block settings
On the general concept of the blocks, Expert 2 asks if it is also possible to start with a simple lookup/search. Expert 1 states that the structure of the query is clear with the block design. The underlying results (from implicit probing) might not be visible enough through the blocks/overview and requires to look at the results. Expert 1 isn't sure how he can deal with errors in previous blocks, i.e. if it's possible to make changes to blocks somewhere on the query. Furthermore, Expert 1 misses an indication if boolean AND/OR is implemented in the filters. Not only seeing the final query but also each Gremlin part for each block can help the user to have a better understanding of the query. Also, a feature to transform Gremlin code into a Pygmalion Query can help learn both Gremlin and Pygmalion Query.

Probing
Little comments were given on the probing concept. Expert 2 states though that the idea is fascinating to allow the user to switch from query to the result and continue with the query once figured out how the current state looks like for refinement.

5.2.2 Comments on the design review

The experts gave comments on some of the features provided in the design of Pygmalion Query. Some of the additional features highlighted have been included in the implementation already, being:

- Showing absolute values for result cardinality per step.
- Showing the per block created Gremlin code.
- Making unavailable choices in filters oder other blocks invisible.

Overall, the feedback from the experts on the design of the visual query language gives an indication that the basic features are in place, and questions/wishes arise more towards specific points of which some are already addressed.

5.2.3 Expert review of the implementation

After the design review, the experts spent some time to use the implemented Pygmalion Query. They were advised to speak out loud and otherwise use the VQL on their own

without much advise. The experts spent this time together, both giving comments of things that came up to them.

For the expert review, a very short introduction to pygmalion was presented to them (see Figure 5.1). The introduction very briefly summarized the justification for Pygmalion Query and the idea behind graph traversals was presented.

Quick Start | Introduction while having coffee.

Pygmalion query helps creating graph traversal queries without writing code. Based on the Blockly language, creating queries is as easy as dragging and dropping elements on the screen.

But first things first! Pygmalion query is based on the Tinkerpop stack - a graph database independent graph framework with a powerful graph traversal language included. Graph traversal queries allow to dig into any graph by "walking" along the vertices and edges. The following image (taken from here) shows this concept:

g.v(1).outE('friend').inV.outE('friend').inV

This image translate to the following code: *g.V(1).outE('friend').inV.outE('friend').inV*

With Pygmalion query, the above code can be expressed visually by dragging and dropping elements from the toolbar on the left onto the canvas on the right. If the part fits - it will connect. Each block has extra visualizations to help understand what can be done. Additionally, an overview of the graph traversal is provided next to the query, the shows how different objects are being traversed and how the cardinality is changing.

Got it!

Figure 5.1: A short introduction to Pygmalion Query as presented to the experts.

Expert 1 started thinking out loud with the introduction. As he has knowledge in Cypher he was able to draw experience from there - but concluded that the introduction provided him with little help as the terminology (Gremlin, Blockly) wasn't properly explained. The concept of traversal he found useful as explained, also the concept of step by step actions. On the example shown in Figure 5.1, Expert 2 suggested to align the Gremlin code with the steps in the image.

Over the course of the review, the experts received some hints and insights if they got stuck during phases of the query building. Before they started building queries, the graph was selected for them and a short verbal description was provided. The summary of their findings in this section are aggregated on higher level categories.

Starting the query

The experts struggled with the concept of Blockly in the beginning. As the block library

only shows one label once a graph has been selected (see Figure 4.8), there was no attention on it. With a hint they managed to draw the first block onto the canvas, but weren't aware of the necessary connection to the previous block.

A consensus among the experts in this state was that the category/categories in the library should align with the current y-value of the latest block. This points towards a lack of information on Blockly and the block concept as blocks are also allowed to be dropped in between existing ones. The experts found the general user interface too small but were able to solve the problem by using the browser feature to zoom in.

Visualizations throughout Pygmalion Query

The experts agree that the provided schemata of the graph shown in the graph block is too small and confusing in the beginning - while they later agree that it's helpful. Furthermore, there is a mismatch in the visual representation of the schemata of the graph versus the graph example in the introduction which leads to confusion. Both coloring and shapes should correspond.

Query flow overview

The overview first provided a confusion to the expert as there wasn't enough context provided. The mix of interactivity in both the query blocks as well as the overview (tooltip on mouse over) confused the experts, with Expert 2 realizing the different mouse pointer on elements that allow for selection. Expert 2 mentions that he needs to "re-calculate" for each block what vertex/edge types are at the current state. This shows a lack of expressiveness in the overview visualization.

The block library

The experts focused some amount of time on the block library (the toolbox in Blockly language). The naming for the categories wasn't providing them with any information and some of the block naming/symbolism confused them. Expert 1 mentions that most of the used blocks are within the navigation category, some of which maybe should be available within one click rather than multiple. Additionally, unavailable blocks (such as the initial vertex/edge type blocks) could be removed from the available categories in a similar fashion as the start is. Both experts agree that the Blockly approach might not be suitable in the VQL as the blocks are always added to the end of the current query and thus involve a lot of dragging and dropping. This points again to the missing information of being able to drag/drop blocks in between available ones. A suggested approach is to add a block directly from the parent by showing a + sign.

The query formulation

As pointed out in the review parts on starting the query, the experts struggled with parts of the query formulation. As the tooltip with further information on the block was disregarded, a situation arose in which a repeat block (a more complex structure allowing to execute blocks multiple times) was utilized without the knowledge of being able to add more blocks in the repeat part. The first time the experts used the filter block, only

two options for filtering were available - one being the label, the second the label specific ID of the vertex. As this provides too many options, confusion arose with how to use the input field.

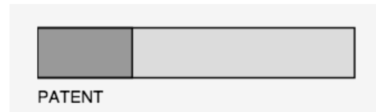


Figure 5.2: An example of the visualization to display the available percentage of a specific label type.

The available percentage of label types within a specific step, as visualized shown in Figure 5.2 was unclear to the experts.

The experts shared the problem of not knowing how to proceed to a specific vertex label to another (i.e. traversing along an edge). As the schemata was too small it only helped after some hints. Something pointed out by the experts in subsection 5.2.1 as feature request, to not show the unavailable options, was pointed out to be confusing in the implementation. This might be attributed to lack of information on the "how" of the filter block.

Implementation bugs

The experts encountered an implementation bug, mainly due to refresh errors (e.g. selecting a property without the flow overview refreshing). This was the only bug encountered, which was resolved by re-attaching the block to its parent (which the experts then routinely did). Something that was seen as bug are the missing representations as shown in Figure 5.2 when the amount of elements are too small, so that the dark grey part vanishes.

Results and probing

The experts agreed on the helpfulness of the probe blocks and the ease of switching between query and results. As the available graph probe limited the visualized vertices/edges on a hard absolute limit (30 vertices, 100 edges), the experts were confused around the lack of edges. This again points towards the need for better information conveying, as this knowledge was available in the tooltip for the block. Also, more explanation would have been needed to highlight the coloring (dark blue/light blue) of the vertices (actual available elements/elements additionally gathered to show a network). Expert 1 pointed out that a potential addition to the probe block functionality would be to show/hide additionally received elements that are not within the immediate result.

5.2.4 Discussion of the Expert Review

The expert review provided interesting insights into the usability of Pygmalion Query. Some of the key potential usability improvements are summarized below:

1. Provide a better introduction: Less terminology, more coherent link to actual implementation.

2. Guide the user when starting the query off: Giving hints at the start on how to continue and what the block concept is about.
3. No problem with web-based approach: Using browser features to zoom in as well as recognized buttons/fields help the user.
4. Provide better information on blocks: The available tooltip information was disregarded.
5. Rethink the block library categories: At the moment they are confusing. Additionally, the block naming/symbolism requires improvement to be easier to understand and distinguish.
6. The query overview needs further explanations: The necessary information is visualized but not understandable without introduction.
7. Provide more guidance on filtering/properties: More indication on the available options is necessary, providing hints when there are too many available is required.
8. Add information to the block setting tooltips: The mouse over tooltip for block settings (see Figure 4.10 on the right side) allows for expansion and more details.
9. Provide more insight into the results view: While being very helpful some context and introduction is needed.
10. Make it easier to add blocks: Dragging and dropping is too tiresome in some cases and confusing in the beginning.

The main problem identified was the lack of context on how to use the block interface and what the different parts of Pygmalion Query really show. While some of the above things can be addressed with more context and information, some require refactoring. Blockly provides many features that weren't used in the implementation, which would lead to a conclusion of not having had to use this framework. More advanced features, such as the repeat block, providing a repository of ready made queries, adding blocks in between others or storing the current state/loading it weren't utilized by the experts (of which all but the last are available at the current state of Pygmalion Query).

5.3 Implemented Feedback

As outlined in subsection 5.2.4, problems with the implementation were found that prevented the experts from fully diving in Pygmalion Query. In this section, this feedback is transformed into implementation changes to be tested in a comparative user study. The changes are split into two categories (which make up the subsections in this section):

- The query canvas
- User Guidance

The query canvas tackles any feedback relating to the structure of Pygmalion Query, while user guidance concerns the procedural issues highlighted.

5.3.1 Changes to the query canvas

Using the items in subsection 5.2.4, changes in this section tackle:

- Provide a better introduction.
- Provide better information on blocks.
- The query overview needs further explanations.
- Add information to the block setting tooltips.
- Provide more insight into the results view.

The introduction was considered unhelpful by the experts. Too many terms were utilized that lacked introduction. For the user experiment, the introduction was split into: Graphs, Graph traversal, the graph of the study and a brief how-to of the tool. See section A.2 for screenshots of the revised introduction. To remove the need for block explanations in tooltips, blocks were renamed and resized to account for a textual description of the block, rather than an identifier. See section A.3 for a list of the changes. Finally, a simple change to the results view was made. The information on the restriction of a result (i.e. limited vertices/edges retrieved) was copied from the tooltip of the probe block into the results view.

5.3.2 Changes to user guidance

Using the items in subsection 5.2.4, changes in this section tackle:

- Guide the user when starting the query off.
- Provide more guidance on filtering/properties.
- Rethink the block library categories.
- Make it easier to add blocks.

One of the most important points coming up in the expert review tackled the user guidance. While Pygmalion Query provides features such as checks for syntactical correctness through only connecting allowed blocks together, starting off and using Blockly to build queries turned out to be a difficult thing. To tackle this, the user interface was changed. Figure 5.3 shows the new user interface (compare with Figure 4.1).

To account for the unhelpful categories, they were removed completely from the screen. The query and query flow overview were switched and with the removed categories, more space was freed up for the query blocks. Each block now has another functional section - commands. This new section only features a button to introduce new blocks, which

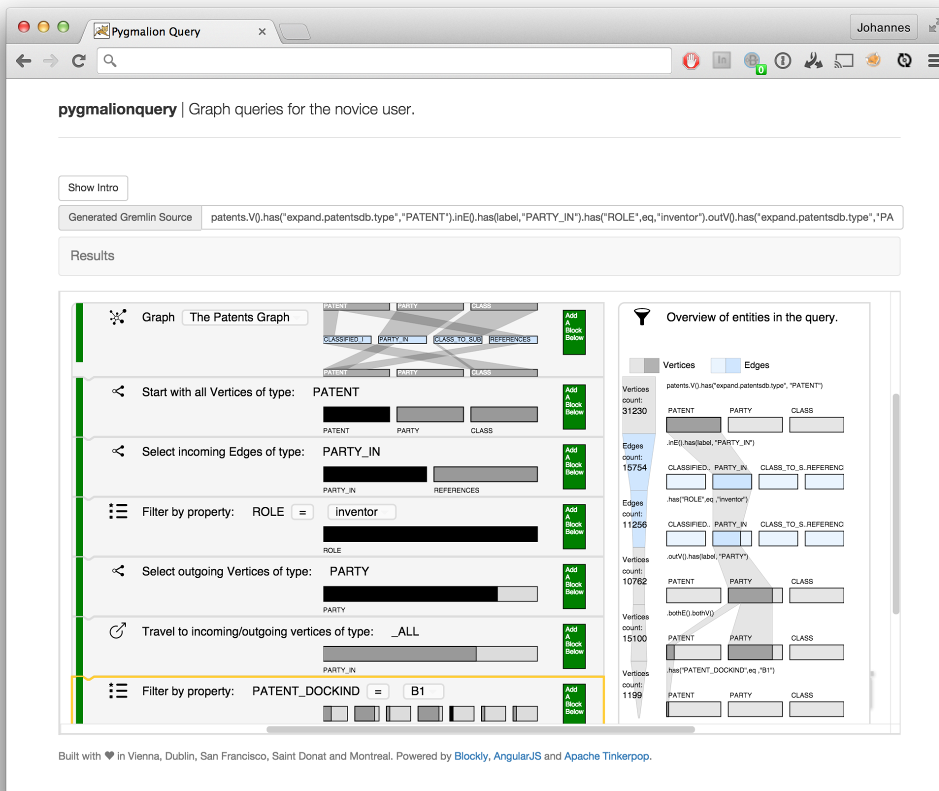


Figure 5.3: The revised Pygmalion Query.

pops out the blocks available to connect. This guides the user as blocks that cannot be connected will not be visible.

5.4 Comparative User Study

5.4.1 The setup of the user study

To be able to better judge the usability in a real context, a small comparative user study was set up. Due to resource limitations, the study wasn't performed in a scale that can yield any statistically significant results, but rather give an indication for comparing the current system (console/terminal code input, see Figure 5.4) vs. Pygmalion Query. The study heavily draws from [15], in which the authors compare a visual query language (QDB*) to SQL. In the paper, three factors were measured:

1. Effectiveness: Can the user arrive at the correct result?

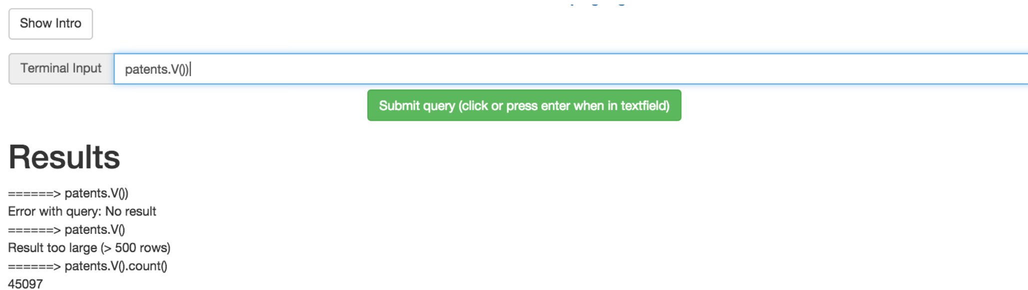


Figure 5.4: The "terminal" which Group 2 of the user study was given to complete the tasks. Queries that resulted in an error or that returned too many results notified the user of this.

2. Efficiency: How quick does the user arrive at the correct result?

3. Satisfaction: How comfortable is the user while using the tool. This factor might be influenced also by surrounding elements such as the physical and organizational environment.

In the paper, effectiveness was measured via a boolean completed/not completed, efficiency via the time to completion and satisfaction wasn't measured.

The comparative user study of Pygmalion Query adopted this approach to the following setup: 14 users were split into two groups each. Both of the groups received the same 3 tasks to complete, and were asked to submit their result + the time it took them. All of this wasn't done supervised, but rather online. Each of the groups received the same short introduction to graphs, graph traversal and the graph used in the study.

- Group 1 additionally received a short introduction to Pygmalion Query and was asked to complete the tasks in the VQL.

- Group 2 on the other hand received a short introduction to Gremlin and was asked to complete the task in a simple Terminal/Console type of online interface with text input/results.

The user study was conducted using a patents data graph with around 45 000 vertices and 54 000 edges that are stored as Neo4J database. To run the study, Pygmalion Query along with an instance of Tinkerpop Gremlin Server was deployed onto Google Compute Engine. A single instance (n1-standard-1) with 1 vCPU and 3.8 GB memory[1] running backports Debian 7 Wheezy v20150127 was utilized. Pygmalion server ran on an apache tomcat 7 server on this instance, with the user interface directly connecting to the Gremlin Server via a WebSocket connection. A maximum of 6% of the available CPU was utilized during the study.

[1] See the official documentation on machine types at https://cloud.google.com/compute/docs/machine-types.

Before the user study was conducted, a supervised trial run was performed to test the study setup itself. Only minor adoptions were made to the description and task outlining after that. The group consisted of people in the age of 25-35, with variable knowledge in graph theory and graph traversals but no previous experience with Gremlin. While coming from a different background each of the participants works in an analytical field and has used SQL before. The choice on this factor ensures that the participants will understand the nature of the tasks. Table 5.1 provides an overview over the participants of the study. Participants 1-7 used the VQL while participants 8-14 used the Terminal. Users answered questions on their expertise with graphs, graph traversal and Gremlin with:

1. Not at all familiar
2. Not too familiar
3. Somewhat familiar
4. Very familiar

ID	Field of Expertise	Familiarity with graphs	Familiarity with graph traversals	Familiarity with Gremlin
1	UX research	1	1	1
2	Computer Science	4	2	1
3	Business Analyst	2	2	1
4	CS/Business	4	1	1
5	Computer Science	4	4	1
6	Business	1	1	1
7	Information Systems	2	2	1
8	Telecommunications Engineer	4	1	1
9	Physics	4	4	1
10	Computer Science	2	4	1
11	Statistics	1	1	1
12	N/A	1	2	1
13	Computer Science	1	4	4
14	Business Informatics	1	4	4

Table 5.1: Overview of the participants of the user study.

The three tasks were in increasing difficulty and had an upper time limit. The time limits were set that someone with experience would easily have time to spare.

5.4.2 Results and discussion of the comparative user study

The following table shows the completion rate (effectiveness) and average time it took to complete (efficiency) aggregated per type (VQL or Terminal). A full overview of the results from the study per participant can be found in

Task	VQL		Terminal	
	Avg. Time (min.)	Completions	Avg. Time (min.)	Completions
Task 1	1.3	6	n/a	0
Task 2	5	7	5	5
Task 3	10	3	9	1

Table 5.2: Aggregated results of the comparative user study.

As previously pointed out, due to the small size of the user study, a statistical analysis may not yield statistically significant results. Additionally, some caveats apply that are pointed out below. Table 5.2 does suggest an indication towards an easier entry for the formulation of graph traversal queries with Pygmalion Query for novice users. Of the overall 15 possible completions possible, 80% were accomplished by Group 1 and 20% by Group 2. Although the tasks did increase in complexity, for an experienced user the time differences would be negligible. Group 1 showed an increase in average time taken for completion, suggesting that a trial and error approach together with the visualizations and user guidance for immediate feedback helped.

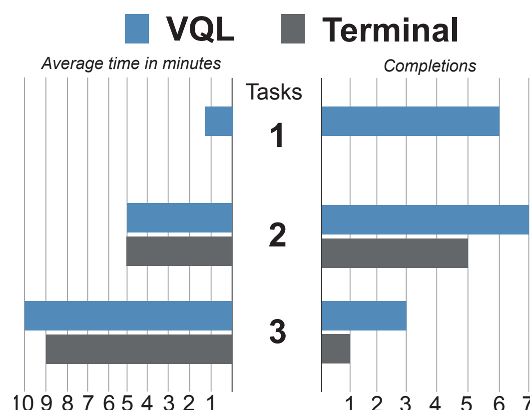


Figure 5.5: The results of the comparative user study, seen in Table 5.2, visualized.

Participants in group 1 completed on average 2.28 tasks with a standard deviation of

0.76, while participants in group 2 completed an average of 0.86 tasks with a standard deviation of 0.69. Due to the small study size and caveats pointed out below, no statistical tests on the differences are computed.

Caveats
The user study suffered of some caveats introduced with the format. Participants in Group 2 received a web based simplified terminal. This was supposed to simulate the Gremlin console, but failed to completely do so. The Gremlin console will also show more details on potential erroneous queries and thus help the user with query formulation. For a complete insight into the usability, the group size would have to be larger and additionally also have a comparison between a very simple, non-textual version (i.e. without any of the visualizations).

5.5 Summary of the evaluation

Two different evaluation methods were utilized to test the hypotheses. An expert review was conducted first, which highlighted some usability issues. These issues were accounted for in an updated implementation, which in turn was tested with a comparative user study. In this user study, 14 participants were evenly split randomly into two groups. Both groups received the same instructions, with only one small paragraph explaining the tool (either a simulated terminal or Pygmalion Query). Participants of the VQL group completed on average 2.28 tasks, while participants of the simulated terminal group completed an average of 0.86 tasks. Next to the small sample size, some other caveats were identified. Both the expert review as well as the user study indicate a usability gain when using the novel visual query language over currently available approach.

CHAPTER 6

Conclusion

In this thesis, a visual query language for the formulation of graph traversal queries on graph databases was developed and evaluated. The need for such a new tool was identified through literature research. It was shown that graph pattern matching and graph traversal are the two main methods utilized to retrieve a subset of data from a graph. Graph pattern matching has seen more advances in non-textual, visually supported query formulation tools while graph traversal hasn't seen such developments and lacks a graph database implementation unspecific visual query language. Through the identification of the gaps this thesis tries to fill, several features such as creation time feedback, bi-directional visual queries and abstraction from graph database implementation have been pointed out as currently not available in any unified solution. In the thesis, the design for Pygmalion Query was outlined, a visual query language for graph traversals that fills these gaps. Requirements were gathered from multiple sources, resulting in selection of necessary features. These were first theoretically designed and afterwards implemented. The web-based implementation was then evaluated in an expert review, providing insights into potential usability defects. A comparative user study followed, with an adapted implementation taking into account the feedback from the experts.

Following the main research question, *"How can we visually support the formulation of queries for a graph database?"*, it was hypothesized that novice (H1) as well as expert users (H2) will benefit from the visual query language over the currently available approaches. Furthermore, hypothesis three (H3) stated that the solution can be unlinked from any specific graph database implementation. Pygmalion Query serves as the answer to the research question. The evaluation indicates that the implemented visual query language provides an easier entry point for graph traversal query formulation for novice users without having to be dependent on a specific graph database implementation. H1 and H3 are thus indicatively (given the caveats pointed out in the previous chapter) confirmed. H2, accounting for expert users, is left to further investigation.

With the thesis, a first step in the direction of providing a tool for formulating any kind of graph traversal queries visually was made. While doing so, more complex queries weren't tackled yet. The user study was executed in a small scale with some caveats - the biggest being the the simplifying of the state as-is for graph traversal query formulation and no available comparison to a simplified visual query language without any further visualization. The developed design leaves room for expanding Pygmalion Query into implementing all available steps of the underlying graph traversal language. The retrieval of data during query formulation, entitled as probing concept, has been introduced without tackling potential performance issues resulting.

Future work on Pygmalion Query is suggested, as the evaluation in this thesis indicates usability gains through the visual query language. There are multiple different areas of development that have been left open. The current scope of the visual query language doesn't tackle all available queries, which will have to be introduced to argue for an equal leveled co-existence along the different code approaches that exist for graph traversals. The probing concept has left room open for performance and efficiency gains, which need to be tackled before allowing to query on larger, potentially distributed graphs. A larger comparative user study needs to be conducted, both tackling the caveats pointed out as well as testing in a more controlled environment. Specific elements within Pygmalion Query, such as the query flow overview can see different implementations that need to be evaluated against each other. Pygmalion Query is built on Blockly, due to the outlined closeness to visual data flow programming languages. But in its current state, many of the features available through Blockly, such as variables or realtime collaboration aren't utilized. Finally, following the trend of many web based applications, further development in the direction of mobile usability of Pygmalion Query is an attractive area of future research.

APPENDIX A

Appendix

A.1 Pygmalion Query File List

The following table gives an overview over the most important files in Pygmalion Query. Some files, such as the static HTML and CSS files as well as any third party libraries are not included.

File	Description
|-	
app.js	The main AngularJS initialization file. Creates the angular app and provides the routing.
config.js	Configurations for Pygmalion Query. This is the only file that needs to be changed by Pygmalion Query users.
pygmalion-blocks.js	The Pygmalion Query Blockly blocks.
pygmalion-d3-field.js	A new field type for Blockly, allowing to inject a D3 svg into a block.
pygmalion-generator-adaption.js	Overwriting the global Blockly generator. More return fields are needed in Pygmalion Query than provided in the original code generator.
pygmalion-gremlin-generator.js	The code generation from block to code for Gremlin.
pygmalion-blocklysvg.js	Overwriting global variables for Blockly to allow interaction with D3 fields and radically change the Blockly design.
|-controllers	
about.js	The about controller simply shows the about page with some details on Pygmalion Query.
main.js	The main controller, glueing the different services together. Anything displayed will be populated within this controller.

File	Description
|-directives	
d3-directive.js	The D3 directive allows to create the same charts and graphics as in the D3 field by using the angular way of creating new DOM elements.
|-model	
query-model.js	The query model defines the structure of a query within Pygmalion Query and will fill templated fields via the angular $interpolate function.
|-services	
angular-blockly.js	This is the adoption of Blockly for AngularJS, to inject Blockly into the page. Also, change events from Blockly are evaluated here and then passed on to the BlockHandlerService (block-handler-service.js).
block-handler-service.js	This service handles instantiation of blocks and will send queries for execution to populate the necessary information.
gremlin-service.js	This service is an angular wrapper for the Gremlin JS client to ensure that queries can only be sent once the connection is established.
|-visualization	
graph.js	The main visualization code. Things such as the force graph, bar charts and overview are defined in D3 javascript here.

Table A.1: File list of Pygmalion Query.

A.2 User Study Material

This appendix features screenshots of the introductions and form presented to the participants of the user study.

A.3 Pygmalion Query block naming adaptions

The table below shows naming changes in probes after the expert review.

Version 1	Version 2
Vertex	Start with all Vertices of type:
Edge	Start with all Edges of type:
Property	Filter by property:
out	Travel to outgoing vertices via edge type:
both	Travel to incoming/outgoing vertices of type:

Version 1	**Version 2**
in	Travel to incoming Vertices of type:
inE	Select incoming Edges of type:
outE	Select outgoing edges of type:
inV	Select incoming Vertices of type:
outV	Select outgoing Vertices of type:
Overview	Overview of entities in the query.

Table A.2: Version 1 (Expert review) and Version 2 (User Study) of the block naming.

Participant ID	**Type**	**Task 1 correct**	**Task 1 time**	**Task 2 correct**	**Task 2 time**	**Task 3 correct**	**Task 3 time**
1	VQL	1	1.5	1	6	1	5
2	VQL	1	0.5	1	1.5	1	15
3	VQL	1	2	1	8	0	3
4	VQL	1	1	1	5	1	10
5	VQL	0	0.5	1	9	0	2
6	VQL	1	1	1	5	0	10
7	VQL	1	2	1	0.5	0	10
8	Terminal	0	4	1	1	0	8
9	Terminal	0	2	1	2	0	8
10	Terminal	0	5	1	2	0	5
11	Terminal	0	15	0	15	0	15
12	Terminal	0	2	0	7	0	10
13	Terminal	0	2	1	5	0	5
14	Terminal	0	2	1	15	1	9

Table A.3: Version 1 (Expert review) and Version 2 (User Study) of the block naming.

Quick Start | Introduction while having coffee.

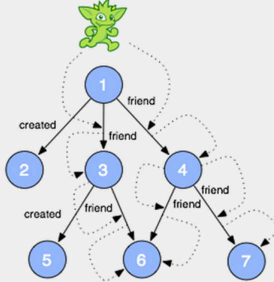
Welcome and **thank you** for participating in this study!

This study focuses around a user interface comparison, specifically around making it easier to write queries for graphs. In this introduction a very brief overview on graphs are given first and afterwards some information on the tool.

Graphs.

A graph (as shown above) consists of so called "vertices" and "edges". In the graph above, the vertices are the circles and the edges are the connections between those vertices. Both vertices and edges can have properties. **A simple example:** Think of the vertices to be people, and the connections telling the relation between them. So in the small graph above, there is one person that has some form of relation to 6 other people. A person (vertices) can have properties such as name, age, height. The connections (edges) can have properties such as type of relation (friend, family), duration of relation (in years) etc. Both vertices and edges have one specific property, called the "label". This indicates the type of the element (e.g. Person, Animal for vertices or knows, family for edges).

Getting a subset: **Graphs Traversal.**



Graphs can get very large (millions of vertices and edges), so analysts typically need to only get a small set of these available entities. This study is focused around the retrieval of this data by graph traversal. The easiest way to think of graph traversal is to think of a liquid running in the graph. It either starts at multiple people or connections and flows to other people or connections. The liquid can be stopped based on filter values (such as only reach people age > 15). The result of such a traversal can be anything from a count of vertices, all the names of selected vertices etc.. The image on the left shows a little "Gremlin" starting at a vertex and running along all the connections that are of type friend. The output of this query can be represented as "friends of friends of vertex 1".

The graph in **this study**

The graph for this study is a network of patents and parties involved. Both vertices and edges have properties, as outlined above.

Using the system.

Pygmalion query helps creating graph traversal queries without writing code. To create queries, blocks are dragged and dropped onto a canvas and connected to each other. Each block has some interactive elements (such as select fields or bar charts) that can be clicked to limit a query.
To get started, a graph needs to be selected in the first block. A schema of the graph will appear in the block that tells how the different Vertices and edges are connected. Vertices are grey and edges are blue. After selection, pygmalion query will retrieve provide a button to show available blocks. To connect to the graph block.
These can be dragged and dropped to connect with the block. On the right side of the blocks, there is an overview that shows what type of elements are currently selected (for example vertices) and how the flow of data is from one to the next step.
To delete blocks either select it and press delete or drag and drop it into the trash bin on the bottom right! To completely reset the query please just close the browser tab and follow the link again!

Figure A.1: Introduction for the visual query language.

Quick Start | Introduction while having coffee.

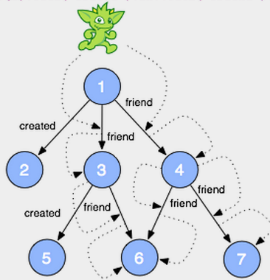
Welcome and **thank you** for participating in this study!

This study focuses around a user interface comparison, specifically around making it easier to write queries for graphs. In this introduction a very brief overview on graphs are given first and afterwards some information on the tool.

Graphs.

A graph (as shown above) consists of so called "vertices" and "edges". In the graph above, the vertices are the circles and the edges are the connections between those vertices. Both vertices and edges can have properties. **A simple example:** Think of the vertices to be people, and the connections telling the relation between them. So in the small graph above, there is one person that has some form of relation to 6 other people. A person (vertices) can have properties such as name, age, height. The connections (edges) can have properties such as type of relation (friend, family), duration of relation (in years) etc. Both vertices and edges have one specific property, called the "label". This indicates the type of the element (e.g. Person, Animal for vertices or knows, family for edges).

Getting a subset: **Graphs Traversal.**



Graphs can get very large (millions of vertices and edges), so analysts typically need to only get a small set of these available entities. This study is focused around the retrieval of this data by graph traversal. The easiest way to think of graph traversal is to think of a liquid running in the graph. It either starts at multiple people or connections and flows to other people or connections. The liquid can be stopped based on filter values (such as only reach people age > 15). The result of such a traversal can be anything from a count of vertices, all the names of selected vertices etc.. The image on the left shows a little "Gremlin" starting at a vertex and running along all the connections that are of type friend. The output of this query can be represented as "friends of friends of vertex 1".

The graph in **this study**

The graph for this study is a network of patents and parties involved. Both vertices and edges have properties, as outlined above.

Using the system.

Below you'll find a a "terminal input" to write graph traversal code. This code looks like the in purple in the image above, where you specify different "steps" as code. You start with a specific graph (in the example above "g", in this study "patents"). From this, you select either multiple vertices or edges with the "V()" or "E()" command. There are many commands that are available like these.
A possible query would be: patents.V().count()
This query would return the amount of vertices in the graph. Within this study, the only necessary commands are listed below:

- patents: The identifier for the graph. Needs to come first.
- V(): Select all vertices in the graph.
- E(): Select all edges in the graph.
- inV(): Select all incoming vertices from an edge.
- outV(): Select all outoing vertices from an edge.
- has(key, value): Filter vertices/edges that have key = value.
- out([optional label]): Move from vertices to vertices via outgoing edges - optional edge label can be specified.
- in([optional label]): Move from vertices to vertices via incoming edges - optional edge label can be specified.
- count(): Count the amount of elements at the stage where this step is added.
- key(): Return all the currently available property keys.
- label(): Show the currently available labels. *Hint: vertices in this dataset don't have labels, the labels to use are the property with key 'expand.patentsdb.type'*
- dedup(): Deduplicate the current output.

More information can be found on Gremlin Docs.

Figure A.2: Introduction for the "terminal".

The tasks

Each of the tasks has a time window (in minutes).

Please proceed on to the next task no mattter how far you've gotten. Please do input your current state before moving on.

Task 1 (2 Minutes Maximum): Please write below what type of vertices and edges exist and how you were able to see/query it. *

Task 1: How long in minutes did it take you? *

Task 2 (10 Minutes Maximum): How many patents with the dock kind "A7" exist in the graph? No matter if you can answer this question or not, please paste your final query below. (The Gremlin Code) *

Task 2: How long in minutes did it take you? *

Task 3 (10 Minutes Maximum): In this graph, patents are linked to patents. There are different links between these patents, and these links have properties. How many vertices do patents link to, with the link type reference that has the property application reference? No matter if you can answer this question or not, please paste your final query below. (The Gremlin Code) *

Task 3: How long in minutes did it take you? *

Figure A.3: The tasks in the user study.

The basics

Please answer the questions below. Your name is only necessary in case there are some follow up questions and will not be used in the result. Thank you!

What's your name? *

What is your field of expertise (e.g. Computer Science, Business, etc.) *

How familiar are you with graphs/graph theory? *

- Not at all familar
- Not too familar
- Somewhat familiar
- Very familiar

How familiar are you with graph traversals? *

- Not at all familar
- Not too familar
- Somewhat familiar
- Very familiar

How familar are you with Tinkerpop Gremlin? *

- Not at all familar
- Not too familar
- Somewhat familiar
- Very familiar

Figure A.4: Additional information gathered within the user study.

Bibliography

[1] C. Abras, D. Maloney-Krichmar, and J. Prrece. User-centered design. In *Encyclopedia of Human-Computer Interaction*, pages 763–768. Berkshire Encyclopedia of Human-Computer Interaction, 2 edition, 2004.

[2] M. Angelaccio, T. Catarci, G. Santucci, R. Ii, T. Vergata, and O. R. Roma. QBD*: a Graphical Query Language with Recursion. *IEEE Transactions on Software Engineering*, 16(10):1150–1163, 1990.

[3] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39, Feb. 2008.

[4] M. Atzori. Computing Recursive SPARQL Queries. In *2014 IEEE International Conference on Semantic Computing*, pages 258–259. IEEE, June 2014.

[5] S. Bhowmick, B. Choi, and S. Zhou. VOGUE: Towards A Visual Interaction-aware Graph Query Processing Framework. *CIDR*, 2013.

[6] H. Blau, D. Jensen, and N. Immermann. A Visual Language for Querying and Updating Graphs. 2002.

[7] M. Bostock, V. Ogievetsky, and J. Heer. DÂş: Data-Driven Documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–9, Dec. 2011.

[8] J. Bresson, C. Agon, and G. Assayag. OpenMusic. In *Proceedings of the 19th ACM international conference on Multimedia - MM '11*, page 743, New York, New York, USA, 2011. ACM Press.

[9] G. Butler, G. Wang, Y. Wang, and L. Zou. A graph database with visual queries for genomics. In *Procs. of the 3rd Asia-Pacific Bioinformatics Conf.*, pages 31–40, 2005.

[10] G. Butler, G. Wang, Y. Wang, and L. Zou. Query optimization for a graph database with visual queries. In M. Li Lee, K.-L. Tan, and V. Wuwongse, editors, *DASFAA'06 Proceedings of the 11th international conference on Database Systems for Advanced Applications*, volume 3882 of *Lecture Notes in Computer Science*, pages 602–616, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[11] L. Cabibbo and R. Torlone. From a procedural to a visual query language for OLAP. In *Proceedings. Tenth International Conference on Scientific and Statistical Database Management (Cat. No.98TB100243)*, pages 74–83. IEEE Comput. Soc, 1998.

[12] D. Calcinelli and M. Mainguenaud. Cigales*: A Visual Query Language for Geographical Information System: The User Interface. *Journal of Visual Languages & Computing*, 5:113–132, 1994.

[13] T. Catarci. Visual Query Languages. In L. LIU and M. T. ÖZSU, editors, *Encyclopedia of Database Systems*, pages 3399–3405. Springer US, Boston, MA, 2009.

[14] T. Catarci, M. Costabile, S. Levialdi, and C. Batini. Visual Query Systems for Databases: A Survey. *Journal of Visual Languages and Computing*, 8:215–260, 1997.

[15] T. Catarci and G. Santucci. Diagrammatic vs. Textual Query Languages: A Comparative Experiment. In *Proc. of the 3rd IFIP 2.6 Working Conference on Visual Database Systems*, pages 69–83, 1997.

[16] D. H. Chau, C. Faloutsos, H. Tong, J. I. Hong, B. Gallagher, and T. Eliassi-Rad. GRAPHITE: A Visual Query System for Large Graphs. In *2008 IEEE International Conference on Data Mining Workshops*, number 4, pages 963–966. IEEE, Dec. 2008.

[17] R. H. Choi and R. K. Wong. VXQ: A visual query language for XML data. *Information Systems Frontiers*, Jan. 2014.

[18] L. Clark. ISWC Demo - SPARQL Views. https://www.youtube.com/watch?v=5FFQnBA5B6k, 2010.

[19] L. Clark. SPARQL Views : A Visual SPARQL Query Builder for Drupal. In A. Polleres and H. Chen, editors, *Poster and Demo Proceedings of the 9th International Semantic Web Conference (ISWC)*. CEUR-WS.org, 2010.

[20] M. P. Consens and A. O. Mendelzon. GraphLog. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - PODS '90*, pages 404–416, New York, New York, USA, 1990. ACM Press.

[21] S. Das, J. Srinivasan, M. Perry, E. I. Chong, and J. Banerjee. A Tale of Two Graphs: Property Graphs as RDF in Oracle. 82:762–773, 2014.

[22] E. Deitrick, J. Sanford, and R. B. Shapiro. BlockyTalky: A Low-Cost, Extensible , Open Source, Programmable , Networked Toolkit for Tangible Creation. 2014.

[23] Facebook Inc. FQL. https://developers.facebook.com/docs/reference/fql/, 2014.

[24] R. Giugno and D. Shasha. GraphGrep: A fast and universal method for querying graphs. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 2, pages 112–115. IEEE Comput. Soc, 2002.

[25] M. A. Gomarasca. Elements of Informatics. In *Basics of Geomatics*, pages 185–230. Springer Netherlands, Dordrecht, 2009.

[26] C. Groenouwe and J.-j. Meyer. Instant playful access to serious programming for non-programmers with a visual functional programming language, 2013.

[27] R. H. Güting. GraphDB : Modeling and Querying Graphs in Databases. In *VLDB '94 Proceedings of the 20th International Conference on Very Large Data Bases*, pages 297–308. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1994.

[28] S. Handschuh, K. Möller, and O. Ambrus. Konduit VQB: a Visual Query Builder for SPARQL on the Social Semantic Desktop. In *Proceedings of the Workshop on Visual Interfaces to the Social and Semantic Web (VISSW 2010)*, 2010.

[29] A. Hartl, K. Weiand, and F. Bry. visKQWL, a visual renderer for a semantic web query language. In *Proceedings of the 19th international conference on World wide web - WWW '10*, page 1253, New York, New York, USA, 2010. ACM Press.

[30] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, page 405, New York, New York, USA, 2008. ACM Press.

[31] F. Hogenboom, V. Milea, F. Frasincar, and U. Kaymak. RDF-GL: A SPARQL-Based Graphical Query Language for RDF. In R. Chbeir, Y. Badr, A. Abraham, and A.-E. Hassanien, editors, *Emergent Web Intelligence: Advanced Information Retrieval*, Advanced Information and Knowledge Processing, pages 87–116. Springer London, London, 2010.

[32] F. Holzschuher and R. Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j. *EDBT '13 Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204, 2013.

[33] H. Huang and Z. Dong. Research on architecture and query performance based on distributed graph database Neo4j. In *2013 3rd International Conference on Consumer Electronics, Communications and Networks*, pages 533–536. IEEE, Nov. 2013.

[34] H. H. Hung, S. S. Bhowmick, B. Q. Truong, B. Choi, and S. Zhou. QUBLE: towards blending interactive visual subgraph search queries on large networks. *The VLDB Journal*, 23(3):401–426, Aug. 2013.

[35] N. T. Inc. Neo4J. http://www.neo4j.org/.

[36] L. Jiang, M. Mandel, and A. Nandi. GestureQuery: A Multitouch Database Query Interface. *Proceedings of the VLDB Endowment*, 6(12):1342–1345, 2013.

[37] L. Jiang, M. Mandel, and A. Nandi. GestureQuery: a multitouch database query interface. *Proceedings of the VLDB Endowment*, 6(12):1342–1345, Aug. 2013.

[38] C. Jin, S. S. Bhowmick, X. Xiao, B. Choi, and S. Zhou. GBLENDER: visual subgraph query formulation meets query processing. In *Proceedings of the 2011 international conference on Management of data - SIGMOD '11*, pages 1327–1330, New York, New York, USA, 2011. ACM Press.

[39] J. Jin and P. Szekely. QueryMarvel: A visual query language for temporal patterns using comic strips. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 207–214. IEEE Computer Society Washington, DC, USA, Sept. 2009.

[40] S. Jouili and V. Vansteenberghe. An Empirical Comparison of Graph Databases. In *2013 International Conference on Social Computing*, pages 708–715. IEEE, Sept. 2013.

[41] B. P. Kinoshita. Cypher, Gremlin and SPARQL: Graph dialects. http://kinoshita.eti.br/2014/09/09/cypher-gremlin-and-sparql-graph-dialects.html, 2014.

[42] M. Kuntz and R. Melchert. Pasta-3's graphical query language: direct manipulation cooperative queries, full expressive power. In *VLDB '89 Proceedings of the 15th international conference on Very large data bases*, pages 97–105. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1989.

[43] D. J. Loveless, B. Griffith, M. E. Bérci, E. Ortlieb, and P. M. Sullivan. *Academic Knowledge Construction and Multimodal Curriculum Development*. IGI Global, 2014.

[44] P. Mäder and J. Cleland-Huang. A visual language for modeling and executing traceability queries. *Software & Systems Modeling*, 12(3):537–553, Apr. 2012.

[45] M. Marttila-kontio. *Visual data flow programming languages: challenges and opptortunities*. PhD thesis, University of Eastern Finland, 2011.

[46] M. P. Medlock-Walton. *TaleBlazer: A Platform for Creating Multiplayer Location Based Games*. Master's thesis, Massachusetts Institute of Technology, 2012.

[47] J. Miller. Graph Database Applications and Concepts with Neo4j. *Proceedings of the Southern Association for Information Systems Conference*, pages 141–147, 2013.

[48] MIT. Scratch. http://scratch.mit.edu/, 2014.

[49] J. Powell, H. Shankar, M. Rodriguez, and H. V. de Sompel. EgoSystem: Where are our Alumni? *code{4}lib Journal*, 24, 2014.

[50] S. Psomas. The Five Competencies of User Experience Design. http://www.uxmatters.com/mt/archives/2007/11/the-five-competencies-of-user-experience-design.php, 2007.

[51] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The CORAL deductive system. *The VLDB Journal*, 3(2):161–210, Apr. 1994.

[52] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases.* 2013.

[53] M. A. Rodriguez. Graph Pattern Matching with Gremlin 1.1. http://markorodriguez.com/2011/06/15/graph-pattern-matching-with-gremlin-1-1/, 2011.

[54] M. A. Rodriguez. SPARQL vs. Gremlin. http://www.tinkerpop.com/docs/wikidocs/gremlin/2.5.0/SPARQL-vs.-Gremlin.html, 2014.

[55] M. A. Rodriguez and P. Neubauer. Constructions from Dots and Lines. *Bulletin of the American Society for Information Science and Technology, American Society for Information Science and Technology*, 36(6):35–41, June 2010.

[56] M. A. Rodriguez and P. Neubauer. The Graph Traversal Pattern. *Chapter in Graph Data Management: Techniques and Applications*, pages 1–18, Apr. 2010.

[57] R. Schaefer. On the limits of visual programming languages. *ACM SIGSOFT Software Engineering Notes*, 36(2):7, Mar. 2011.

[58] H. A. Simon. *The Sciences of the Artificial.* The MIT Press, 3rd edition, 1996.

[59] W. Slany. Catroid: a mobile visual programming system for children. In *Proceedings of the 11th International Conference on Interaction Design and Children - IDC '12*, page 300, New York, New York, USA, 2012. ACM Press.

[60] R. Spence. *Information Visualization: Design for Interaction.* Prentice Hall, 2nd edition, 2007.

[61] B. Stehno and M. Haidacher. Rapid Visualization Development based on Visual Programming Developing a Visualization Prototyping Language. 2012.

[62] I. Tanase, Y. Xia, L. Nai, Y. Liu, W. Tan, J. Crawford, and C.-y. Lin. A Highly Efficient Runtime and Graph Library for Large Scale Graph Analytics. pages 1–6.

[63] Tinkerpop. Tinkerpop Stack. http://www.tinkerpop.com, 2014.

[64] M. Tory and T. Moller. Evaluating Visualizations: Do Expert Reviews Work? *IEEE Computer Graphics and Applications*, 25(5):8–11, Sept. 2005.

[65] W3C. SPARQL 1.1 Query Language. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/, 2013.

[66] P. T. Wood. Query languages for graph databases. *ACM SIGMOD Record*, 41(1):50, Apr. 2012.

[67] L. Yang, B. Geng, Y. Cai, A. Hanjalic, and X.-S. Hua. Object Retrieval Using Visual Query Context. *IEEE Transactions on Multimedia*, 13(6):1295–1307, Dec. 2011.

Glossary

JPA Java Persistence API. 13

OLAP Offline Analytical Processing. 40

OLTP Online Transaction Processing. 40

RDBMS Relational Data Base Management Systems. 10

RDF Resource Description Framework. 8

SQL Structured Query Language. 1, 11, 24, 79, 81

SRT System Response Time. 28

TSQL Transactional Structured Query Language. 23

UXD User Experience Design. 17

VDFPL Visual data flow programming language. 23

VPL Visual programming language. 21

VQL Visual Query Language. 3, 23–28, 30, 31, 33, 58, 64, 71, 73, 75, 80, 81, 83