

# **VISUALL: A QUICKLY CUSTOMIZABLE LIBRARY FOR JUMPSTARTING VISUAL GRAPH ANALYSIS COMPONENTS**

A THESIS SUBMITTED TO  
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE  
OF BILKENT UNIVERSITY  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR  
THE DEGREE OF  
MASTER OF SCIENCE  
IN  
COMPUTER ENGINEERING

By  
Yusuf Sait Canbaz  
September 2021

VISUALL: A QUICKLY CUSTOMIZABLE LIBRARY FOR JUMP-  
STARTING VISUAL GRAPH ANALYSIS COMPONENTS

By Yusuf Sait Canbaz

September 2021

We certify that we have read this thesis and that in our opinion it is fully adequate,  
in scope and in quality, as a thesis for the degree of Master of Science.

---

Uğur Doğrusöz(Advisor)

---

Uğur Güdükbay

---

Osman Abul

Approved for the Graduate School of Engineering and Science:

---

Ezhan Karaşan  
Director of the Graduate School

## ABSTRACT

# VISUALL: A QUICKLY CUSTOMIZABLE LIBRARY FOR JUMPSTARTING VISUAL GRAPH ANALYSIS COMPONENTS

Yusuf Sait Canbaz

M.S. in Computer Engineering

Advisor: Uğur Doğrusöz

September 2021

Graph visualization is an area of information visualization, where relational data is depicted in the form of nodes (objects) and edges (links). Many people or organizations utilize graph visualization for insightful analysis and interpretation of relational data. In graph visualization, primary challenges include complexity management, efficient database querying, and customization for specific domains. *Visuall* aims to solve these problems by providing a generic, highly customizable, and easily configurable software component for building web-based visual graph analysis tools.

Essential functionalities needed by such visual analysis components include manually or automatically setting the layout of graph elements, support for nested or hierarchical drawings, efficient querying of the database or client-side data, emphasizing or highlighting graph elements of interest, customization of visuals and styles, clustering, calculating graph-theoretical properties, and time-based filtering of graph elements. Although *Visuall* provides all these functionalities out of the box for jumpstarting, customization of software for domain-specific needs is still unavoidable. Such software changes might result in complications due to unstructured code and code ignoring the invariants assumed by the original development team. To prevent these and to facilitate easily maintainable customization, *Visuall* provides a modular architecture. Furthermore, the developers straightforwardly upgrade the software so long as the *Visuall* developers and the users developing visual analysis components based on *Visuall* maintain the provided architecture.

We tested our database queries on a database that contains about half a million graph elements. We also examined our client-side operations up to a thousand

graph elements. In both client-side and database operations, we observe that operations take at most several seconds, making *Visuall* convenient for interactive exploration and analysis of networks.

*Keywords:* information visualization, graph visualization, software system, complexity management, visual analysis.

## ÖZET

# VISUALL: GÖRSEL ÇİZGE ANALİZİ BİLEŞENLERİNİ HIZLICA BAŞLATMAK İÇİN ÇABUK ÖZELLEŞTİRİLEBİLİR BİR KÜTÜPHANE

Yusuf Sait Canbaz

Bilgisayar Mühendisliği, Yüksek Lisans

Tez Danışmanı: Uğur Doğrusöz

Eylül 2021

Çizge görselleştirme, ilişkisel verilerin düğümler (nesneler) ve kenarlar (bağlantılar) şeklinde gösterildiği bir bilgi görselleştirme alanıdır. Birçok kişi veya kuruluş, ilişkisel verilerin detaylı analiz ve yorumlanması için çizge görselleştirmeden faydalanır. Çizge görselleştirmede, birincil zorluklar arasında karmaşıklık yönetimi, etkili veritabanı sorgulama ve belirli alanlar için özelleştirme yer alır. *Visuall*, web tabanlı görsel çizge analiz araçları oluşturmak için genel, son derece özelleştirilebilir ve kolayca yapılandırılabilir bir yazılım bileşeni sağlayarak bu sorunları çözmeyi amaçlar.

Bu tür görsel analiz bileşenlerinin ihtiyaç duyduğu temel işlevler, çizge öğelerinin düzenini manuel veya otomatik olarak ayarlama, iç içe veya hiyerarşik çizimler için destek, veritabanı veya istemci tarafı verilerinin etkili sorgulanması, ilgilenilen çizge öğelerinin vurgulanması veya ayırt edilmesi, görsellerin ve stillerin özelleştirilmesi, kümeleme, çizge-teorik özelliklerin hesaplanması ve çizge öğelerinin zamana dayalı filtrelenmesini içerir. *Visuall*, hızlı bir başlangıç için tüm bu işlevleri zaten sağlasa da, yazılımın alana özgü ihtiyaçlar için özelleştirilmesi hala kaçınılmazdır. Bu tür yazılım değişiklikleri, sistematik olmama ve kodun orijinal geliştirme ekibi tarafından varsayılan değişmezlerinin yok sayılması nedeniyle ihtilaflara neden olabilir. Bunları önlemek, anlaşılır ve sürdürülebilir özelleştirmeler yapmak için *Visuall* modüler bir mimari sağlar. Ayrıca, sağlanan mimari korunduğu sürece, *Visuall* geliştiricileri ve *Visuall* tabanlı bileşenleri geliştirenler yazılımlarını kolayca doğrudan günceller.

Veritabanı sorgularımızı yaklaşık yarım milyon çizge ögesi içeren bir veritabanında test ettik. İstemci tarafındaki operasyonlarımızı da bin civarı çizge ögeye

kadar inceledik. Hem istemci tarafında hem de veritabanı işlemlerinde, işlemlerin en fazla birkaç saniye sürdüğünü gözlemliyoruz. Bu gözlemler *Visuall*'un çizge görselleştirmelerin etkileşimli keşfi ve analizi için kullanışlı halde olduğunu gösteriyor.

*Anahtar sözcükler:* bilgi görselleştirme, çizge görselleştirme, yazılım sistemi, karmaşıklık yönetimi, görsel analiz.

## Acknowledgement

I would like to express my appreciation and sincere thankfulness to my supervisor Prof. Uğur Doğrusöz for letting me work with him. His expertise, guidance, and help are the backbone of this study. His patience, discipline, and meticulousness in work will be an example for me.

I also would like to thank Prof. Uğur Güdükbay and Prof. Osman Abul for accepting to be on the thesis committee. Their review and feedback are helpful for me.

I would like to thank my family for their immense support and love. I also would like to thank my supportive friends. I have to thank to alumni of i-Vis Research Lab for their valuable works. Also, I would like to thank current members of i-Vis Research Lab for their constructive ideas and efforts.

I would like to express my gratitude to the Scientific and Technological Research Council of Turkey (TÜBİTAK) for their extensive support (TÜBİTAK-Teydeb projects 5180088 and 5200049) during my studies.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	3
1.2	Contribution . . . . .	5
<b>2</b>	<b>Background and Related Work</b>	<b>8</b>
2.1	Graphs . . . . .	8
2.2	Graph Visualization . . . . .	9
2.3	Graph Visualization Tools . . . . .	12
2.4	Cytoscape.js . . . . .	14
2.5	Neo4j . . . . .	15
2.5.1	Labeled Property Graph Data Model . . . . .	16
2.5.2	Cypher . . . . .	17
2.5.3	User-Defined Procedures . . . . .	18
<b>3</b>	<b>Visual Architecture</b>	<b>19</b>

3.1	Overview . . . . .	19
3.2	Build and Maintenance . . . . .	20
3.3	Jump-Starting . . . . .	21
3.4	Application Description File . . . . .	22
3.4.1	General Information . . . . .	23
3.4.2	Objects . . . . .	23
3.4.3	Relations . . . . .	24
3.4.4	Timebar Data . . . . .	24
3.4.5	Enumeration Mapping . . . . .	25
3.4.6	App Preferences . . . . .	26
3.5	Customization . . . . .	26
3.5.1	Navigation bar and toolbar . . . . .	27
3.5.2	Context menus . . . . .	28
3.5.3	Tabs . . . . .	28
3.5.4	Database Service . . . . .	30
<b>4</b>	<b>Methodology and Use Cases</b>	<b>31</b>
4.1	Basic Visual Analysis & Filtering . . . . .	31
4.2	Persistency . . . . .	33
4.3	Automatic Layout . . . . .	34

4.4	Clustering . . . . .	37
4.4.1	Compound Nodes . . . . .	38
4.4.2	Compound Edges . . . . .	40
4.4.3	Clustering Algorithms . . . . .	42
4.5	Object Inspection . . . . .	43
4.6	Time-Based Filtering . . . . .	46
4.7	Querying Data . . . . .	49
4.7.1	Query By Rules . . . . .	49
4.7.2	Graph-Based Queries . . . . .	50
4.7.3	Table View . . . . .	58
<b>5</b>	<b>Testing &amp; Evaluation</b>	<b>61</b>
5.1	Implementation and Testing . . . . .	61
5.2	Evaluation . . . . .	64
5.2.1	Graph-Theoretical Properties . . . . .	65
5.2.2	Complexity Management Through Hide-Show . . . . .	66
5.2.3	Clustering . . . . .	67
5.2.4	Database Querying . . . . .	68
<b>6</b>	<b>Conclusion</b>	<b>72</b>

6.1 Limitations & Future Work . . . . .	72
---	----

# List of Figures

1.1	A sample graph visualization with a rich style . . . . .	2
1.2	Overview of <i>Visuall</i> User Interface . . . . .	6
2.1	A graph containing a compound nodes. The compounds have rectangular shapes. . . . .	9
2.2	The left drawing shows an undirected edge between two nodes. The right one shows a directed edge between two nodes . . . . .	9
2.3	The left drawing shows a <i>self-loop</i> between two nodes. The right one shows <i>multi-edges</i> between two nodes . . . . .	10
2.4	Spring analogy figure from [1] . . . . .	10
2.5	The graph on the left and right are exactly the same graph but their layouts are different . . . . .	11
2.6	Neural Muscle Signaling graph visualized with Cytospape.js and another open source tool <i>Newt</i> [2] [3] . . . . .	15
2.7	A sample labeled property graph model . . . . .	16
2.8	A sample Cypher query bringing all the <i>Person</i> nodes satisfying some conditions . . . . .	17

2.9	A sample Cypher query bringing a <i>Person</i> node, his co-actors and the movies they played together . . . . .	17
3.1	Overview of software architecture of <i>Visuall</i> . . . . .	19
3.2	A sample <i>Visuall</i> application user interface . . . . .	22
3.3	Sample <i>appInfo</i> section from a description file . . . . .	23
3.4	Sample <i>Objects</i> section from a description file . . . . .	24
3.5	Sample timebar data mapping section from a description file. Here production start and end dates of a title are taken as the two ends of the lifetime of a title. . . . .	25
3.6	On the left is the content of a sample <code>enums.json</code> file. On the right is the corresponding enumeration mapping section inside the application description file. . . . .	25
3.7	Query uses enumeration mapping for drop-downs and also in query generation. . . . .	26
3.8	Sample custom menu items . . . . .	27
3.9	Sample context menu items shown when clicked on an edge, a node, and the core . . . . .	28
3.10	Code snippet for custom context menu items to add actions to use the title's poster as node UI and to open the corresponding IMDB page in a new tab, respectively. . . . .	29
3.11	Sample custom queries in the sample movies graph. (left) Find actors playing in at least given number of titles, (right) Find titles of specified genre. . . . .	30

4.1	User interface to filter out certain elements (in this case node “Title” and edges “KNOWN_FOR” and “SELF”) based on type . . .	32
4.2	Sample graph with contains two selected nodes and one selected edge. . . . .	32
4.3	Sample graph with highlighted elements. Highlighted elements constitutes a path of length eight . . . . .	33
4.4	A sample query history . . . . .	34
4.5	Sample graph with six clusters. Circles represent clusters. There are no compound nodes in this graph. . . . .	35
4.6	(middle) A portion of the IMDB data showing a particular actor playing in three different movies. (left) Effect of placing new nodes using the mentioned extension before applying an incremental layout when other actors/actresses of a movie are displayed. (right) Resulting layout when newly added actors/actresses are assigned default positions of (0,0), where the previous layout is drastically changed potentially destroying the user’s mental map. . . . .	36
4.7	Effects of packing components with different aspect ratios . . . . .	37
4.8	The drawing on top shows a graph with two compound nodes, both expanded. The one below shows the same compound graph with one of the compound nodes in collapsed state. . . . .	39
4.9	An example graph showing many nested groups . . . . .	40
4.10	The drawing on the left shows multi-edges between two nodes. The one on the right shows the same graph after these multi-edges are collapsed and represented with a single compound edge . . . . .	41
4.11	Clustering choices for grouping nodes in the sample application . . . . .	42

4.12	Inspecting properties by selecting a node (left) or an edge (right) .	43
4.13	Inspecting properties by of multiple nodes (top) and multiple edges (bottom) . . . . .	44
4.14	<i>Statistics</i> sub-tab inside <i>Object</i> tab . . . . .	45
4.15	Whether or not a particular graph element passes the time filtering and is shown is determined by one of these three options . . . . .	46
4.16	An example view from the timebar . . . . .	47
4.17	A timebar statistic counting the number of “People” who satisfy defined conditions in Algorithm 1. . . . .	47
4.18	A timebar statistics condition for counting people who directed more than three movies . . . . .	48
4.19	A statistic that gives the total sum of ratings of titles that satisfy the specified conditions . . . . .	49
4.20	A sample screen for query by rule . . . . .	50
4.21	A sample neighborhood query and its results . . . . .	52
4.22	An example of manually selecting graph elements from the table view and adding them to the graph view . . . . .	59
4.23	Emphasize corresponding graph elements on graph . . . . .	60
5.1	Cypress script to test whether the user is be able to run a nested query by rule . . . . .	63
5.2	Cypress script to test whether the user is able to generate groupings and then remove them . . . . .	63

5.3	After all 31 <i>Visual</i> E2E tests are finished in Cypress . . . . .	64
5.4	Execution times of calculating various graph theoretical properties and then showing them on the graph . . . . .	65
5.5	Execution times of showing and hiding elements from a certain type	67
5.6	Execution times of clustering algorithms with circular and com- pound cluster representations . . . . .	68
5.7	“Complex rule 1” inside Table 5.1 . . . . .	69
5.8	“Complex rule 2” inside Table 5.1 . . . . .	70
5.9	“Complex rule 3” inside Table 5.1 . . . . .	70

# List of Tables

2.1	Comparison of functionalities of <i>Visuall</i> and similar tools . . . . .	13
5.1	Execution times of various database related operations in seconds	69

# Chapter 1

## Introduction

A *graph* is an abstract data type. It consists of a set of nodes (also called vertices, points or objects) and a set of edges (relationships or links between nodes).

A *compound graph* is an enriched form of a graph, where besides simple nodes and edges, it can contain compound nodes that might, in turn, contain sub-graphs.

Graph visualization depicts relational data in the form of nodes and edges. People use graphs to analyze and interpret social networks, computer networks, biological networks, and relational information in many other different areas.

Figure 1.1 shows a sample graph visualization. In the compound graph in this figure, there are compound nodes used for grouping movies and people associated with the movies. Compound nodes can contain nodes or other compound nodes. With enriched visualizations, nodes might have background images and dynamic sizes. There can be badges or annotations on the nodes. Colors or labels of the nodes and edges can be custom. Also, the positions of the nodes can be arbitrary. In short, the topology or style of a graph visualization can be vastly different.

Visualizing relational data as a graph is quite intuitive. It naturally depicts the data [4]. Even a layperson could understand the data. An observer with

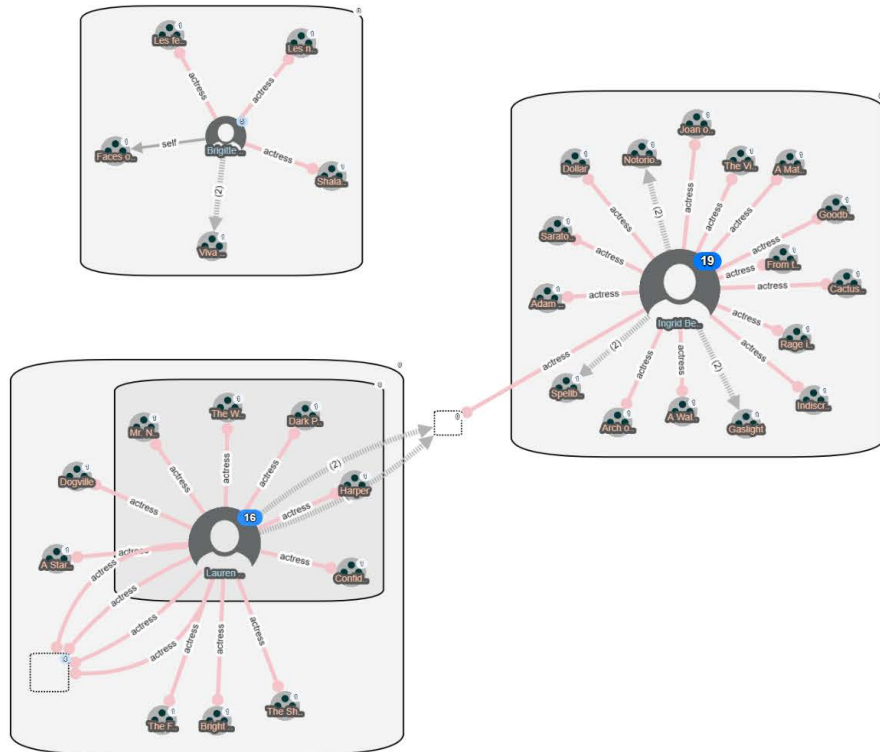


Figure 1.1: A sample graph visualization with a rich style

no domain knowledge can still make considerably better interpretations from fair graph visualization.

People use graph visualization in many domains such as finance, telecom, biology, social networks, transportation, recommendation systems, healthcare, and pharmaceutical domains. Visualization paves the way for interaction, interpretation, and analysis. It is used to extract knowledge from data. Mainly visualization aims for distilling knowledge with human cognition. It helps human intelligence using graph algorithms, user interaction, and varying drawings/layouts. Also, as machine learning algorithms can extract insight in different ways, in a sense, such algorithms can derive knowledge without human cognition.

Relational database management systems are used very frequently for data storage. PostgreSQL, MySQL, and Microsoft SQL Server are popular systems [5]. In these systems, a table represents a type of entity. A row of the table corresponds to an actual entity. A column corresponds to an attribute of the

corresponding entity. If there is a connection between two different types of entities, a column is used as a pointer. The column refers to a row in another table. Usually, these columns are called *foreign keys*.

When we make complex queries that require utilizing connections, we should use foreign keys to refer to other tables and *join* tables. Since a table might contain too much data, joining tables is a costly operation. That is particularly significant if the data is highly connected. There are numerous efforts to optimize the performance of *join* operations [6]. Although it depends on the data model, usually we represent a connection between two entities as an edge (relationship). So if we need to make a query that involves many connections, a graph database will make a traversal. That might give significantly higher performance compared to a join-heavy SQL query [7]. In some domains such as fraud detection, taking action in real-time is necessary. So usage of a graph database system might be unavoidable. Since graph databases provide such conventions, the interest in graph databases has increased [8].

## 1.1 Motivation

Complexity management is one of the primary challenges in graph visualization. If the graph has medium-large size, it can easily turn into an *hairball* [9]. Layout algorithms can help the user to mine information value. Layout algorithms place the nodes such that visual patterns are more visible. Even if a layout algorithm produces a decent layout, the visualization can be still too complex or too large for human cognition. If there are too many edges between two nodes, the drawing will be complex and costly. If a node has too many neighbors, it will seem too crowded. To reduce complexity, the user might want to focus on a certain part. The user might want to remove a certain part and then bring it back. The user might want to mark certain elements. Grouping or clustering elements can also help to reduce complexity [10]. Finding a suitable clustering algorithm is a serious issue. To make visualization interactive, the execution time of all the operations should be less than a few seconds. Depending on the size of the graph, reaching

low execution times can be hard. At the same time, the previous visualization and the new one should resemble as much as possible. The user should be able to follow the changes easily. Thus, the operations should work incrementally. Especially the layout algorithm should be able to work incrementally. Using animations in the visualization is also essential to grasp the changes.

Customization is another problem in graph visualization. Every company has its own data schema and different concerns in visualization. There can be lots of variety in the drawing. Nodes can have background images, background colors, border colors. Edges can have different thicknesses, colors, arrow shapes. Nodes and edges can have labels and the labels can be placed in different positions. Apart from visualization, companies also want to customize the user interface (UI). For graph visualization, the UI is important because interaction with the visualization is provided through the UI. To serve multiple companies or users, the software should stick to multi-tenant architecture. Since there are extensive customizations in many places, there will be some customer-specific source codes. Thus, maintenance of such projects can be cumbersome due to shared codes and customer-specific codes.

Efficient database querying might be difficult. A query should respond in few seconds. Queries should consider the resulting data size. If a query will result in large amounts of data, it will take too much time. Also, visualization of big data will be a problem. Since it is not useful to bring large chunks of data at once, all database queries should have some kind of pagination strategy. Queries should bring the data page by page. Bringing a graph in multiple parts is challenging because the pagination strategy affects the connectivity of the graph. Often applications might need data to fill tables. So queries should be able to bring both tabular and graph data.

## 1.2 Contribution

We propose a software library that forms a base for developing domain-specific graph visualization and analysis software. By using *Visuall*, developers will get a jump-start, with many tools already in their pockets. These tools can be used to address generic problems in graph visualization and analysis. At its core, *Visuall* is a visualization software. It has a database-agnostic architecture. In theory, it can visualize any kind of data source.

Since the user is going to define a graph model and might want to change the styles of nodes, edges, or other visual components of visualization software, *Visuall* provides a so-called application description file. By using this file, the user will first define the topology of their graphs. Then, they might change the styles of the nodes and edges as desired. Also, the user may change the font size and style used throughout the entire user interface of the software. Making such changes does not require any programming experience nor a change in the source codes of the software.

No matter how generalized and flexible software is, changes in the source code of software will still be unavoidable. Thus, a developer should be allowed to change the source codes of the software. For this reason, we provide a modular software architecture to make maintainable customizations as well as software updates. The changes in the source code of *Visuall* should not affect such customizations. There might be bug fixes or performance improvements in the base. To prevent possible source code conflicts, we reserve some of the files solely for the developer. We guarantee that improvements or new features in *Visuall* base will not change anything in these files and the developer is free to change these files as they like.

For complexity management in big graph visualizations, *Visuall* provides many functionalities such as showing/hiding nodes or edges, highlighting, searching labels, doing manual layout, using automatic layout algorithms, panning and zooming to a region in the graph, compound nodes, compound edges, support

for graph-theoretical properties, clustering, and expanding/collapsing compound elements. It also provides time-based filtering through an independent software component. By using this component, the users can perform time-based filtering and display the changes in their graph-based data over time through animation.

Often the data to be visualized is in a server-side database in an ample amount. A typical user will be interested in only a small part of the data in the database. *Visuall* provides a way for building complex database queries based on the properties of nodes and edges. The user can construct these queries just by using the user interface. *Visuall* also provides some graph algorithms implementing commonly required database queries.

As a result, *Visuall* provides a software library for quickly building domain-specific visual analysis components for relational information.

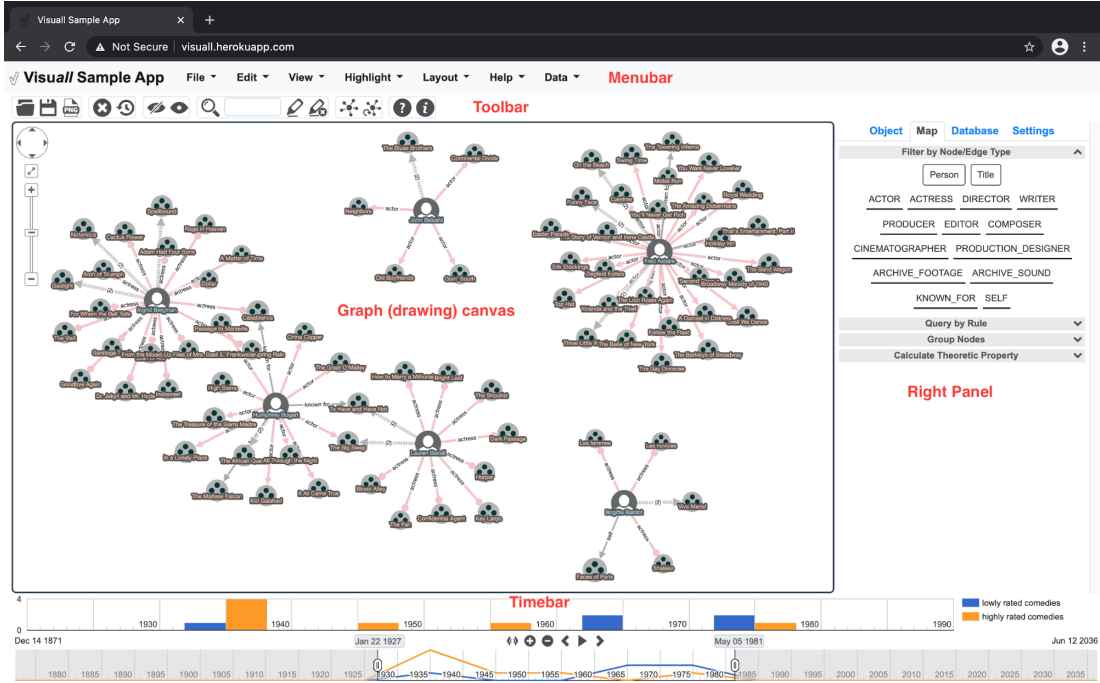


Figure 1.2: Overview of *Visuall* User Interface

In summary the work done within the scope of this thesis can be listed as follows:

- Design and implementation of a software library that can be easily/quickly customized for domain-specific needs, and
- Adaptation of the pathway database query algorithms to be used in generic graph database querying as well as performance improvements to some of these queries.

# Chapter 2

## Background and Related Work

### 2.1 Graphs

A *graph*  $G$  can be defined with a set of *vertices* (or *nodes*)  $V$ , representing objects, and a set of *edges*  $E$ , representing links or relations between the objects. An edge  $e \in E$  is a pair of vertices  $\{u, v\}$  such that  $u \in V$  and  $v \in V$ .

An edge  $e = (u, v)$  with a direction has a dedicated source node  $u$  and a dedicated target node  $v$ . If a graph consists of directed edges, it is called a *directed graph* or a *digraph*. A graph with undirected edges is called an *undirected graph* or simply a *graph*.

If there are multiple edges (two or more) between two vertices, these edges are called *multiple edges* or *multi-edges* or *parallel edges*. If an edge connects a vertex to itself, it is called a *self-loop* or simply a *loop*. A *subgraph* is a subset of a graph's vertices and edges defined between these vertices.

A node is called a *compound* node if it contains other nodes. The nodes inside the compound are called the children of the compound. The compound node is called the parent. Usually, the edges inside the compound do not count as children. Compound nodes are useful to represent hierarchical relations, groups

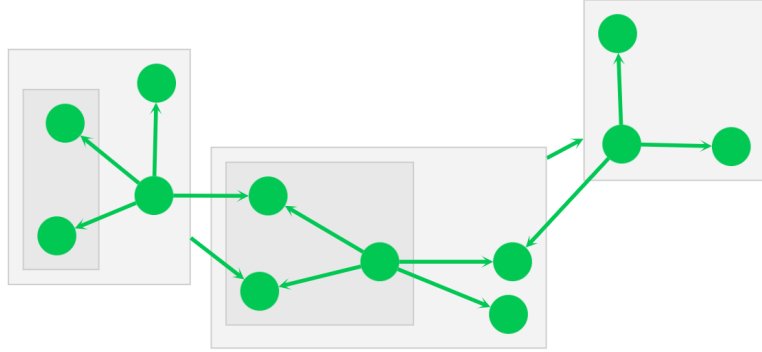


Figure 2.1: A graph containing a compound nodes. The compounds have rectangular shapes.



Figure 2.2: The left drawing shows an undirected edge between two nodes. The right one shows a directed edge between two nodes

and, clusters. Since a *compound* node can also contain other compound nodes, graphs can visualize nested structures conventionally.

A node is *incident* to an edge if the node is one of the two nodes the edge connects. The *degree* of a node is the number of edges incident to the node. The *in-degree* of a node is the number of incoming edges to the node. Similarly, the *out-degree* of a node is the number of outgoing edges from the node. A *path* is defined with a sequence of nodes  $\langle n_0, n_1, \dots, n_k \rangle$  and a sequence of edges  $\langle n_0n_1, n_1n_2, \dots, n_{k-1}n_k \rangle$  such that  $n_i$  are distinct and  $k \geq 0$ . The length of a path  $k$  is the number of edges it contains.

## 2.2 Graph Visualization

One main operation required in graph visualization is the automatic layout of the graph. Typically, graphs are visualized in two-dimensional space. So layout



Figure 2.3: The left drawing shows a *self-loop* between two nodes. The right one shows *multi-edges* between two nodes

algorithm sets positions (x and y coordinates) of the vertices. The layout is critical because the same graph can be visualized in many different ways.

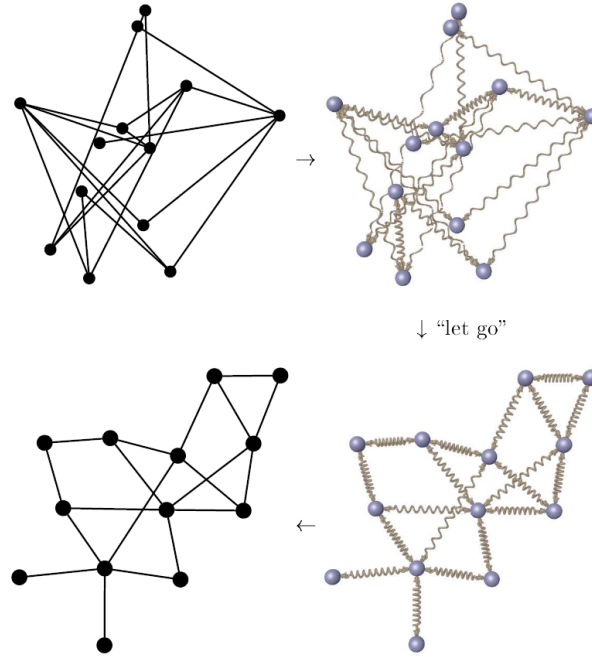


Figure 2.4: Spring analogy figure from [1]

Even though there are many metrics to measure the quality of layout such as edge-edge crossing number and total area of the drawing [11], there are also aesthetic concerns. In a drawing, sizes, the ratio of sizes, colors, shadows, transparency, borders, border colors, border sizes and shapes of nodes as well as edge arrow shapes, line styles, curve styles are critically important. Such design decisions are important in terms of human perception, and also for the performance of rendering. If the styles are heavy, it might be costly to render big graphs.

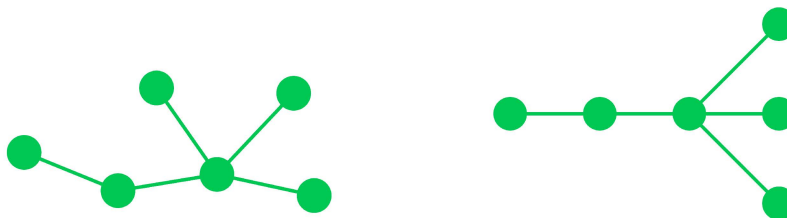


Figure 2.5: The graph on the left and right are exactly the same graph but their layouts are different

There are several different types of layout algorithms in the literature. Force-directed algorithms (aka spring embedders) are one of the most common [1]. The main idea is to represent nodes as electric charges and edges as physical springs. Thus based on Coulomb’s electric force law and Hooke’s law about spring force, nodes will apply repulsion and attraction forces. The algorithms perform calculations until the system is stable or close to stable (Figure 2.4).

Briefly, the compound spring embedder (CoSE) layout algorithm is an extended force-directed layout algorithm that supports compound nodes with arbitrary levels of nesting [12]. Fast compound spring embedder (fCoSE) builds on CoSE and uses the spectral graph drawing technique for quickly producing a draft layout, succeeded by phases in which user-defined constraints are enforced while layout is polished. So, fCoSE can satisfy user-defined placement constraints and it is faster than CoSE [13].

A layout algorithm can run in two different manners. It can run in an incremental and non-incremental (randomized) manner. We call it incremental when current node positions are taken into account and layout is calculated with the aim to minimally distract the user’s mental map (current respective positions of graph elements). The non-incremental (randomized) layout ignores the existing positions of the nodes.

## 2.3 Graph Visualization Tools

Most open source and/or free graph visualization tools are end-user tools. GraphViz [14], Gephi [15], and Neo4j Bloom [16] are some examples. There are also some free tools that are designed for customization with a fair bit of programming such as Cytoscape.js [17] and D3.js [18].

On the commercial side, some tools target specific domains. For instance, Ravelin [19] is for fraud detection, IBM i2 is for threat intelligence [20], whereas others are more generic such as yFiles [21], Tom Sawyer Perspectives [22], KeyLines [23], and Linkurious [24]. Most commercial tools allow many different ways of customization, but all charge high licensing fees.

*Visuall* is a software library intended for developers for customization with little effort. The customization step is necessary, because information visualization is purely based on data. *Visuall*'s architecture and design can reduce customization time to be as short as a couple of days.

*Visuall* provides many built-in functionalities in addition to an advanced and clear way for customization. It has support for compound nodes, complex and custom queries. It uses some advanced layout algorithms for doing fast and aesthetic layouts. It has a component called *Timebar* for time-based filtering. It supports calculation and display of various graph-theoretical properties of graph elements. Table 2.1 summarizes a comparison of functionalities of *Visuall* with some other tools.

	Customization	Compound Nodes	Complex & Custom Queries	Fast Layout	Time-based filtering	Graph Theoretical Properties
<i>Visuall</i>	✓	✓	✓	✓	✓	✓
IBM i2 [20]	✓	x	x	partial	x	partial
Ravelin [19]	partial	x	x	✓	x	x
Tom Sawyer [22]	✓	✓	partial	✓	x	✓
yWorks [21]	partial	✓	partial	✓	x	✓
Key Lines [23]	partial	partial	partial	partial	✓	✓
Linkurious [24]	partial	x	partial	partial	x	✓
GraphViz [14]	partial	✓	x	partial	x	x
Gephi [15]	partial	x	x	✓	x	✓
Cytoscape.js [17]	✓	✓	x	✓	x	✓

Table 2.1: Comparison of functionalities of *Visuall* and similar tools

## 2.4 Cytoscape.js

Cytoscape.js [17] is an open-source software library for visualizing graphs (networks) written in JavaScript. It provides a comprehensive API for interacting with graphs and visualization styles. Cytoscape.js is widely used in biological domains. Visualizing graphs in the biological domain is usually complex, requiring drawings with lots of different shapes, varieties in topology, and models. To provide such flexibility in graph visualization, Cytoscape.js expects to get some styles similar to CSS styles [25]. For convention, we will call these styles simply Cytoscape.js styles. Figure 2.6 is a relatively complex sample drawing using Cytoscape.js.

Graphs with up to a thousand elements can be comfortably visualized, where Cytoscape.js renders such graphs responsively. This depends on the hardware and software of the client system because processing is done on the client-side through a web browser. The performance also depends on the styles used in visualization. When no heavy styling is used, we observe that tens of thousands of elements can be visualized in an interactive manner responsively.

*Visuall* was explicitly built on Cytoscape.js with the aim of easy customization and minimal coding.

Furthermore, Cytoscape.js provides an extension mechanism so that developers can implement their functionalities. Extensions can be used to implement layout algorithms, user interface components, and also for extending the Cytoscape.js API. *Visuall* is powered with lots of such useful extensions.

There is also a desktop application of Cytoscape [26]. Cytoscape.js is a software library for developers whereas Cytoscape is a desktop application for end-users.

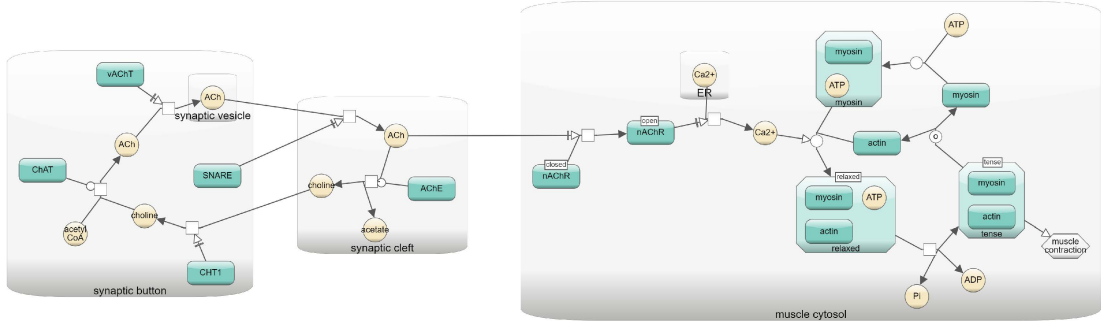


Figure 2.6: Neural Muscle Signaling graph visualized with Cytoscape.js and another open source tool *Newton* [2] [3]

## 2.5 Neo4j

Neo4j is a graph database platform implemented in Java [27]. It is convenient for quick starts because the setup is very easy. Both the data and database system itself are inside a single folder. It can be imported and exported simply by moving that root folder. It is a database management system but it stores data as a graph different from commonly used relational database systems. Because of this significant difference at its core, it might be more suitable for highly connected data. Since it is a database system, it enables users to perform standard database operations like create, read, update, and delete. To perform these operations, Neo4j has its query language called *Cypher*. Using Cypher, the users can easily get answers to simple queries. Of course, the users will need advanced queries. For advanced queries, Neo4j provides a mechanism to install *User-defined procedures*. User-defined procedures are very similar to *stored procedures* in relational database systems. In relational database systems, usually stored procedures and queries are written in the same query language such as SQL. In Neo4j, queries are written in Cypher but User-defined procedures are written in Java.

*Visual* has a built-in support for Neo4j. A Neo4j database can be visualized using *Visual* simply by providing credentials such as username, password, and URL of the Neo4j database.

## 2.5.1 Labeled Property Graph Data Model

In Neo4j's graph/data model, there are four components: *nodes*, *relationships*, *properties* and, *labels*. *Nodes* represent entities, objects, or vertices in the graph. *Relationships* represent edges, connections, or links between *nodes*. So far, there is no difference with the definition of *Graph* as an abstract data type. The difference starts with *properties*. In Neo4j, *nodes* and *relationships* can have *properties*. *Properties* are simply key-value pairs. They are like metadata. They store some data about the node or the edge it belongs to. Also, in this model, *relationships* must have a direction. When undirected edges/relations are needed, the user can simply ignore direction information.

In Neo4j model, *labels* are used to tag nodes. Basically, a *label* corresponds to a *Class* or an *Interface* in Object-Oriented Programming terminology. A *label* can be used to express a type or a role. A *node* can have multiple *labels* but typically it is recommended to have a single *label*. In Neo4j, *relationships* have no *labels* but instead have *types*. A *relationship* can have a single *type*.

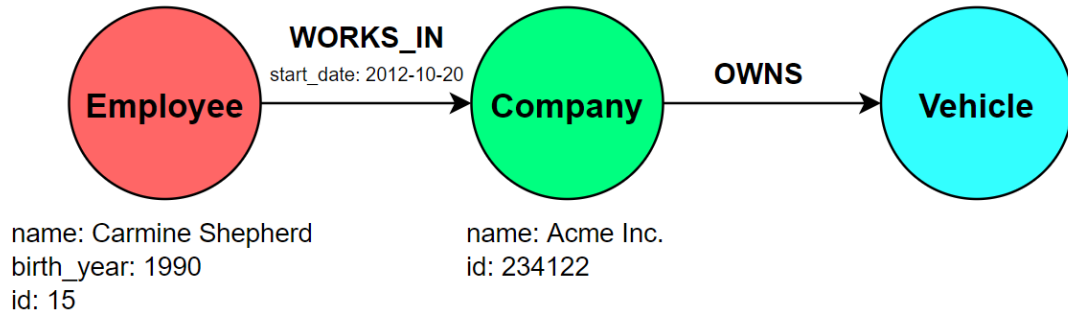


Figure 2.7: A sample labeled property graph model

In Figure 2.7, there are three types of nodes and two types of relationships (edges). The nodes respectively from left to right have *labels* 'Employee', 'Company' and 'Vehicle'. The edges respectively from left to right have *types* 'WORKS\_IN' and 'OWNS'. As can be seen, the 'Employee' and 'Company' nodes have some *properties* like 'name' and 'id'. The 'WORKS\_IN' edge also has a *property* named "start\_date". Usually, edges are named using verbs or actions for better readability.

In *Visuall*, we assume that a node has exactly one label and an edge has exactly one type. Hence, the graph model provided in *Visuall* follows the labeled property graph model of Neo4j.

### 2.5.2 Cypher

*Cypher* is the query language of Neo4j, specifically developed by Neo4j for this purpose. It is a declarative language that can be used like SQL to create, read, update or delete records. These kinds of queries are directly based on the properties of records. Figure 2.8 shows a query that brings some nodes that have label *Person* and their *primary\_name* contains the string ‘Tom’ and *birth\_year* greater than 1980.

```
1 MATCH (x:Person)
2 WHERE (x.primary_name Contains 'Tom' AND x.birth_year > 1980)
3 RETURN x
```

Figure 2.8: A sample Cypher query bringing all the *Person* nodes satisfying some conditions

At the same time, the users can make queries to describe paths or patterns related to the topology of the graph. Figure 2.9 shows such a query. Parenthesis are used to represent nodes. Square braces are used to represent edges. After the *MATCH* statement, we are essentially describing a path. This path is used to find co-actors and common movies they played in. Making such queries with SQL might require costly join operations. Also, SQL code for the same query is typically more complex/longer than that in Cypher.

```
1 MATCH p=(x:Person)-[:ACTOR]->(y:Title)<-[:ACTOR|:ACTRESS]-(x2:Person)
2 WHERE x.primary_name = 'Eddie Redmayne'
3 RETURN p
```

Figure 2.9: A sample Cypher query bringing a *Person* node, his co-actors and the movies they played together

### 2.5.3 User-Defined Procedures

*User-defined procedures* are very similar to stored procedures in relational database management systems. They are used to extend the capabilities of Cypher. They should be written in Java using Neo4j's official Java driver. After the codes are ready, a JAR file should be exported. And then the JAR file should be placed inside the *Plugins* folder. After that, the procedures can be called from Cypher codes.

## Chapter 3

# Visuall Architecture

### 3.1 Overview

Visuall has a web-based user interface, written in JavaScript. *Angular* [28] was used as the web application framework of Visuall. This choice is mainly due to *Angular* being typescript-based. Enforcing types makes applications more robust, making management of code more convenient during development. Visuall was primarily designed to jump-start visual analysis software components up and running. Mainly, it consists of three *Angular* modules named *AppModule*, *CustomizationModule*, and *SharedModule*. You can see the basic architecture in Figure 3.1, where the directed edges show the dependency ( $A \rightarrow B$ : B imports / depends on A) between the modules.

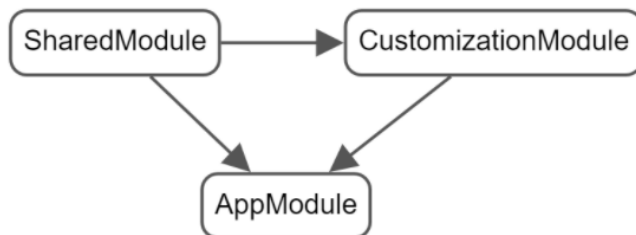


Figure 3.1: Overview of software architecture of Visuall

## 3.2 Build and Maintenance

As mentioned earlier, *Visuall* is a software library. First and foremost, it should be configured and customized by developers. Firstly, a developer needs to make their own copy (by forking on GitHub where sources are maintained) of *Visuall*. If the developer creates a fork, it becomes easier to get new commits for improvements or new features from the original *Visuall* repository. The original remote repository should be added with a command similar to:

```
git remote add base https://github.com/...
```

In previous command, the original repository is named as *base*. The developers could fetch the new commits from the base with command:

```
git fetch base
```

Then the developers could apply a new commit with a *cherry-pick* command such as:

```
git cherry-pick -n 12ae
```

They could alternatively merge the remote branch to get the updates. A command such as

```
git merge base/master
```

will merge “master” branch from the remote repository with alias “base” to the current branch.

Ideally, the developers should only need to change the files under the `/src/app/custom` folder. Changing other files might result in conflicts during

updates if the original *Visuall* repository has also changed for improvements or new features. All things under this custom folder and *CustomizationModule* are reserved for the developers. The developers can generate and use their Angular components by declaring them under the *CustomizationModule*. Since *AppModule* depends on *CustomizationModule*, the developers should be careful not to break the existing dependencies.

### 3.3 Jump-Starting

Many things such as the type and style of nodes and edges, the name and logo of your application, default values of many settings are defined in a so-called *application description file*. We will explain this file in more detail in the upcoming sections.

Once this file is ready, the developer should execute the `style-generator.js` script. This will read the *application description file* and then modify the `index.html`, `styles.css`, `properties.json`, and `stylesheet.json` files according to the descriptions provided in this file. These files will basically constitute the skeleton of the visual analysis component's code base.

In the end, the user is expected to see a web application similar to the one in Figure 3.2. Here, you can see that at the top, there are drop-down menus for various actions. We call this part the *navigation bar*. Right below the navigation bar is the *toolbar*. The toolbar is a shortcut for reaching commonly used menu items rather than using the drop-down menus. At the center is the *graph canvas* area. This is basically where the user is expected to focus most of the time. At the bottom of the page is where you see two charts and some action buttons between them for the *timebar* component. On the right, adjacent to the graph canvas are various tabs with sub-tabs for inspection, queries and configuration of the visual component.

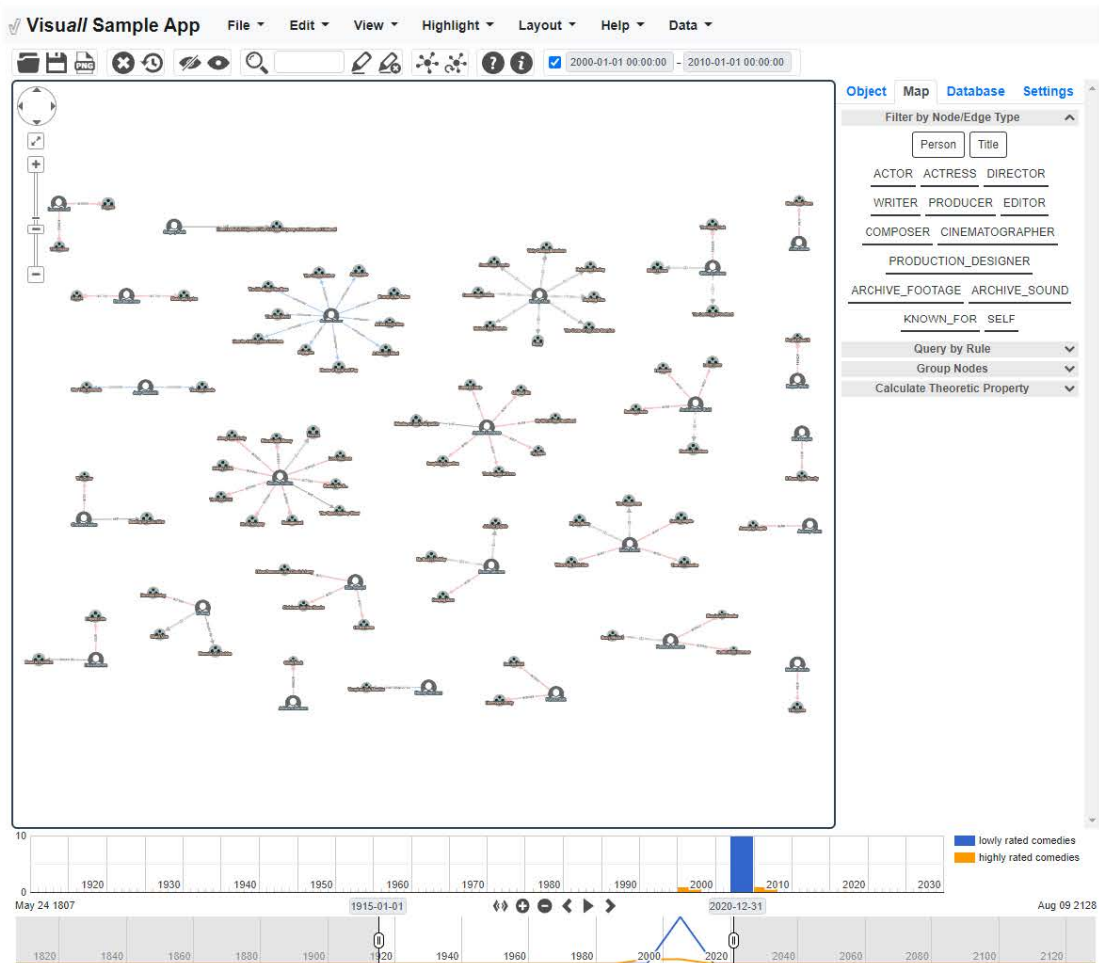


Figure 3.2: A sample VisualI application user interface

### 3.4 Application Description File

If a user wants to start building their own specialized visualization component, this file is where they start. Firstly, in this file, the user defines the data schema or the graph model. Then comes the definition of the Cytoscape.js styles to be used for each type of node and each type of edge. These styles will be used in rendering the associated graph elements. They basically specify how the corresponding objects and relationships will look.

### 3.4.1 General Information

The first section in the *application description file* is *appInfo*. This section contains metadata about the application such as its name, version, and logo. Figure 3.3 shows a sample.

```
"appInfo": {  
  "name": "Visu<i>all</i> Sample App",  
  "html_header": "Visuall Sample App",  
  "icon": "app/custom/assets/logo.png",  
  "version": "1.0.0 beta",  
  "company_name": "i-Vis at Bilkent",  
  "company_contact": "ivis@cs.bilkent.edu.tr"  
},
```

Figure 3.3: Sample *appInfo* section from a description file

### 3.4.2 Objects

This section describes the objects in the data schema or the graph model. Here for each node type, we expect to have *properties* and *style*. Figure 3.4 shows a sample *objects* section. There are two types of nodes called *Person* and *Title* in this example. Each type of node has data properties and Cytoscape.js styles to be applied during rendering. In this example, a *Person* has properties such as *primary\_name* and *birth\_year*. Each property has an associated data type. These data types are defined inside *Visuall*. Available data types are **string**, **int**, **float**, **datetime**, **list**, and **enum**. **string**, **int**, and **float** are primitive data types as in most programming languages. **datetime** is actually the same as **int** but it used to represent a date and time in Unix timestamp in milliseconds. Here, the **list** data type simply corresponds to an array of values. **enum** is a special type used to represent a predefined finite set of values. For example, a status code can be represented as an **enum** type. In the database, status codes might be stored with integers such as 0, 1, and 2 but these codes might be presented to the users using strings like “Success”, “Error”, and “Pending”. Since this is a mapping from integers, it should have type "**enum,int**". These finite sets of values should be defined inside a designated file named **enums.json**.

```

"objects": {
  "Person": {
    "properties": {
      "primary_name": "string",
      "primary_profession": "enum,string",
      "birth_year": "int",
      "death_year": "int"
    },
    "style": {
      "color": "#a7d6eb",
      "shape": "ellipse",
      "width": "32px",
      "height": "32px"
    }
  },
  "Title": { ...
}
},

```

Figure 3.4: Sample *Objects* section from a description file

### 3.4.3 Relations

This section describes the data schema of edges (connections) in the graph. Just like the *Objects* section, for each type, we expect to have *properties* and *style*. Additionally, for an edge type, the user should also express its *source* and *target* as well as whether or not it is bidirectional (*isBidirectional*). *source* expresses the type of source node for this type of edge, while *target* expresses the type of target node for this type of edge. *isBidirectional* states whether the edge type is bidirectional or unidirectional.

### 3.4.4 Timebar Data

This section is only needed for the timebar. Ideally, each node and each edge should have some properties that correspond to the lifetime of the element. For each node type and each edge type, the names of these properties might be different. So this section describes for each type which property should indicate the beginning or end lifetime of that element type. When beginning (end) lifetime is not provided, we assume an element exists from the beginning (until the end)

of the time range. Figure 3.5 shows a sample.

```
"timebarDataMapping": {
  "Person": {
    "begin_datetime": "start_t",
    "end_datetime": "end_t"
  },
  "Title": {
    "begin_datetime": "production_start_date",
    "end_datetime": "production_end_date"
  },
  "ACTOR": {
    "begin_datetime": "act_begin",
    "end_datetime": "act_end"
  },
}
```

Figure 3.5: Sample timebar data mapping section from a description file. Here production start and end dates of a title are taken as the two ends of the lifetime of a title.

### 3.4.5 Enumeration Mapping

This section is used to map enumeration typed properties to their actual values. For example, if we have an *Object* type named *MoneyTransfer* (could be an edge type or node type) and it has an enumeration property named *Status*, we can simply map status codes to string values by defining the enumeration mapping in the description file. Figure 3.6 shows a defined enumeration and its mapping.

```
{
  "MONEY_TRANSFER_STATUS": {
    "0": "Success",
    "1": "Pending",
    "2": "Fail"
  }
}

"enumMapping": {
  "MoneyTransfer": {
    "status": "MONEY_TRANSFER_STATUS"
  }
},
```

Figure 3.6: On the left is the content of a sample `enums.json` file. On the right is the corresponding enumeration mapping section inside the application description file.

These enumeration mappings are used to map values. At the same time, *VisualI* uses these values to show drop-downs for example for query generation. Figure 3.7 shows how enumeration mapping is used to form the user interface in queries.

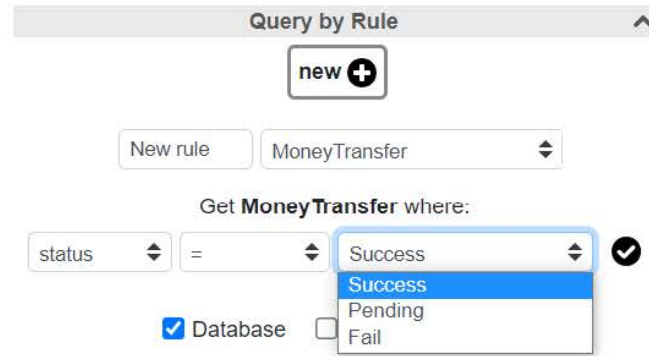


Figure 3.7: Query uses enumeration mapping for drop-downs and also in query generation.

### 3.4.6 App Preferences

This section simply stores all optional behaviors throughout *Visuall*. For instance, some users might prefer not to display the timebar but some other might want to show it by default.

## 3.5 Customization

Modification of the application description file will provide the user with bare minimum customization. This does not require any coding but mere formatting (in JSON) of the description and configuration of the visual analysis component. Advanced users however will like to change the source code to provide advanced customization and implement domain specific functionality. As *Visuall* continues to evolve, our changes and changes of the user may cause conflicts. To eliminate the conflicts, we reserved files under `/src/app/custom` folder for the user. A developer can add or change all the files under this folder by adding their components and implementing their business logic. Specifically, the developer is able to inject their components into many places such as the navigation bar, toolbar, context menu, and tabs. Lastly, the developer can also write their own database service. *Visuall* already has a database service for Neo4j databases but they might want to use a different database provider.

### 3.5.1 Navigation bar and toolbar

The developers can inject their custom menu items into both the toolbar and navigation bar. Figure 3.8 shows a new drop-down menu named “Custom DropDown 1”. The developer can also add menu items to existing drop-downs. “Custom Action 1” is such an operation in this example. Below the drop-down items, you can see that there is a toolbar. The toolbar is simply a shortcut to trigger popular actions conveniently. This example also contains custom items.

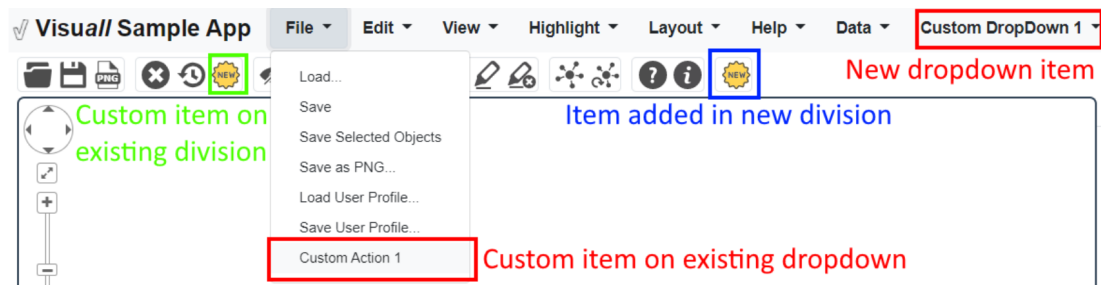


Figure 3.8: Sample custom menu items

To achieve this, the developer should provide an array in the source codes. The array contains names and corresponding functions to call. Here the functions should have no parameter. Below code snippet shows a sample array.

```
[{
  dropdown: 'File',
  actions: [
    { txt: 'Custom Action 1', id: '', fn: 'fn1', isStd: false }
  ]
},
{
  dropdown: 'Custom DropDown 1',
  actions: [
    { txt: 'Custom Action 2', id: '', fn: 'fn2', isStd: false }
  ]
}]
```

### 3.5.2 Context menus

The context menu shows up when the user right-clicks a graph element or the background of the graph. To provide this functionality, *Visuall* uses a Cytoscape.js extension named *cytoscape.js-context-menus* [29]. Inside `context-menu-customization.service.ts` file, the developer should put their context menu items. It might be preferred displaying varying menu items based on the right-clicked element type. For example, right-clicking on a node and on an edge could show different operations. Here, right-clicking on an empty place on the graph canvas can also show a menu. We call this *the core context menu*. Figure 3.9 shows three different context menus shown for an edge, a node, and the core, respectively from left to right.

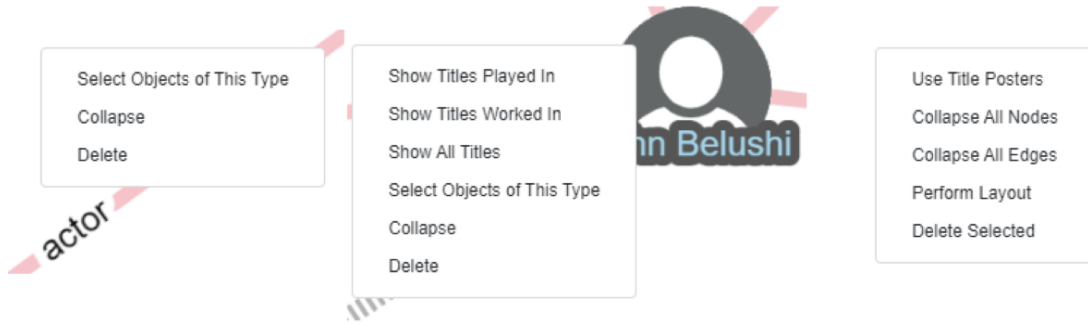


Figure 3.9: Sample context menu items shown when clicked on an edge, a node, and the core

For custom context menu items, the developer should provide an array of objects. Each object corresponds to an item. Figure 3.10 shows a sample array.

### 3.5.3 Tabs

*Visuall* contains the following tabs: “Object”, “Map”, “Database”, and “Settings”. Each tab contains subsections. We call them *sub-tabs* (implemented using *accordion* GUI components). The developers can add their own tabs and/or sub-tabs. For adding a tab or sub-tab, the developer should provide their Angular component. We simply inject the provided component into the application.

```
[{
  id: 'getPoster',
  content: 'Use Title Poster',
  selector: 'node.Title',
  onClickFunction: this.getPoster.bind(this)
},
{
  id: 'go to IMDB',
  content: 'Go to IMDB',
  selector: 'node.Title',
  onClickFunction: this.goInfo.bind(this)
}]
```

Figure 3.10: Code snippet for custom context menu items to add actions to use the title’s poster as node UI and to open the corresponding IMDB page in a new tab, respectively.

Here, the component should not have any input or output like a function with no parameters. These components should be placed inside the `/src/app/custom` folder as usual. The developer can import and use Angular components defined in `SharedModule` (e.g., `TableViewComponent`) inside their own Angular components.

*Database* tab contains a particularly specialized sub-tab called *Custom Queries*. This is where the developer can implement their own custom database queries. *Visuall* provides built-in rule based (SQL-like) queries to the database through the user interface. This might be enough up to a level, but the user might need more advanced custom queries based on advanced graph traversals. Figure 3.11 shows two queries named “Get actors by title counts” and “Get titles by genre” implemented in this manner.

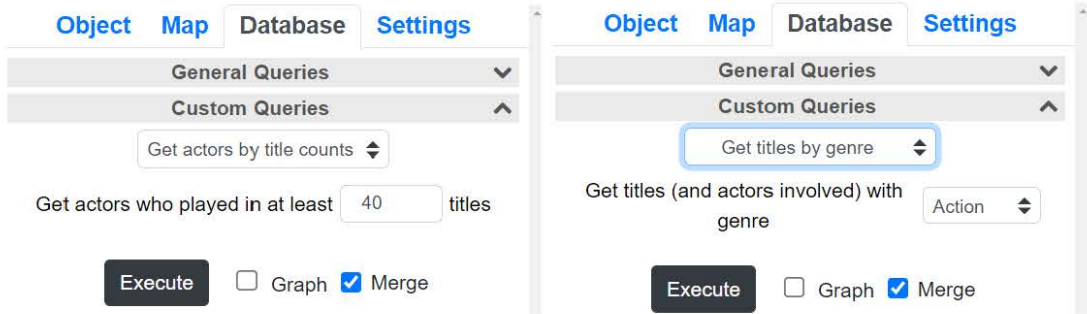


Figure 3.11: Sample custom queries in the sample movies graph. (left) Find actors playing in at least given number of titles, (right) Find titles of specified genre.

### 3.5.4 Database Service

As we mentioned before, *Visuall* is database agnostic. Any database can be integrated for visualization, provided that there is a database service that implements our interface named *DbService* for communication with the database. This interface defines a contract between the database and *Visuall*. The interface requires implementation of certain functions. *Visuall* will call these functions with some parameters. In the end, the functions should call the callback function with the necessary data parameters. The callback function will simply parse the data passed to itself and show the graph.

The built-in Neo4j database service needs a database URL, username, and password. By default, *Visuall* uses the HTTP API of Neo4j to connect with the database instance.

# Chapter 4

## Methodology and Use Cases

This chapter presents the methodology and modules within *Visuall* to make it a full-fledged visual analysis tool that is highly customizable.

### 4.1 Basic Visual Analysis & Filtering

*Visuall* provides lots of instruments for visual analysis of your relational data. It displays these relations based on your current focus. The user can additionally filter or show/hide elements. Hidden elements can be later shown on demand, whereas when the user deletes a graph element, it will be removed from the graph model on the client side (unless a subsequent query brings them back from the server). Hiding and showing can be done based on type or by manual selection. Figure 4.1 shows all the node types (“Person” and “Title”) and edge types (e.g., “Actor”) are listed in the sample application. Rounded rectangular shapes represent node types in this dialog, whereas straight lines represent edge types. That gives the user an intuition about whether the corresponding element is a node or edge.

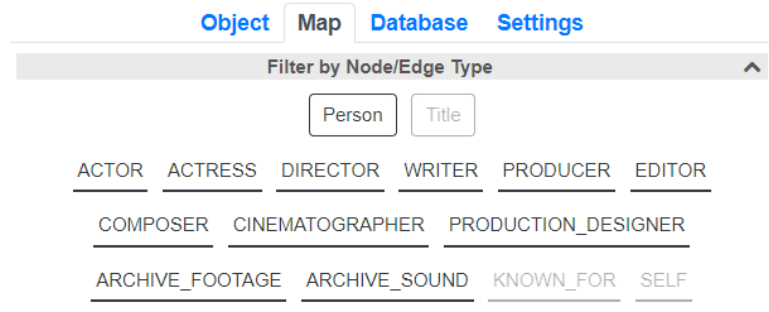


Figure 4.1: User interface to filter out certain elements (in this case node “Title” and edges “KNOWN\_FOR” and “SELF”) based on type

The user can select nodes or edges simply by clicking on them. Selection of multiple items can be achieved by additionally holding the shift key. To signify selected items, we use an overlay whose color can be customized by the user. Figure 4.2 shows the nodes “Rowan Atkinson” and “Keeping Mum” are selected nodes. The edge “actor” is also selected. For selection, we can use different Cytoscape.js styles instead of overlay. For example, we can change node and edge colors. However, we do not recommend change of such basic functionality for keeping things consistent.

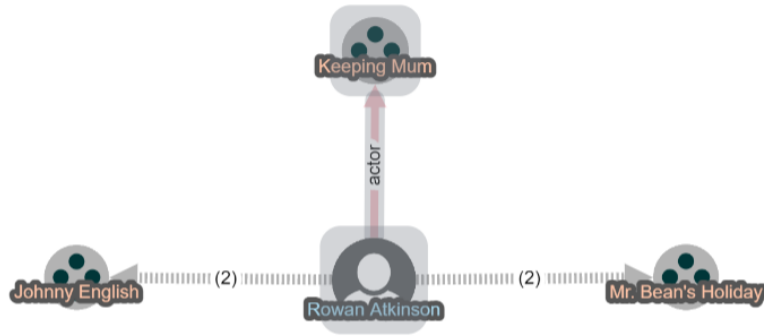
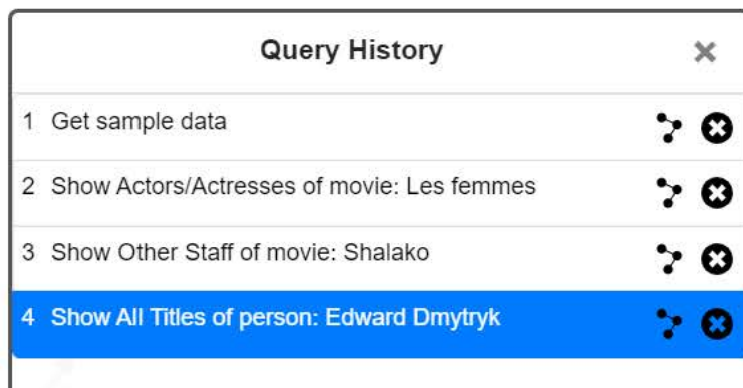


Figure 4.2: Sample graph with contains two selected nodes and one selected edge.

The user can also *highlight* graph elements. We use overlays with custom colors for this as well. The user can also search object labels with a keyword by using the search bar on the toolbar. Visuall will highlight the founded elements. Figure 4.3 shows a path with length eight is highlighted among 120 graph elements.





Query History		
1	Get sample data	🔗 ✕
2	Show Actors/Actresses of movie: Les femmes	🔗 ✕
3	Show Other Staff of movie: Shalako	🔗 ✕
4	Show All Titles of person: Edward Dmytryk	🔗 ✕

Figure 4.4: A sample query history

*Visuall* also persists various metadata using the browser’s local storage. It stores data without an expiration date. Preferences related to *Visuall*, timebar statistics, and ruled queries are saved to local storage. So even if the user restarts the computer or browser, they can continue to work from where they left off.

### 4.3 Automatic Layout

For automatic layout, *Visuall* uses *fCoSE* [13, 12] and *CiSE* [30] layout algorithms. Both of these algorithms can run incrementally. *Visuall* will apply an incremental layout on changes to the topology of the graph being visualized (for example, as things are filtered or new content is merged into the graph). It also allows the user to re-calculate layout from scratch however (especially useful when you are not happy with the current layout). *Visuall* uses *CiSE* only when there are clusters represented using circles. Figure 4.5 is an example a graph using the *CiSE* algorithm to lay out a clustered graph, where each cluster is laid out on a separate circle. When cluster structure is instead shown as compounds (nested drawings), then *fCoSE* is preferred as *CiSE* cannot handle compound structures.

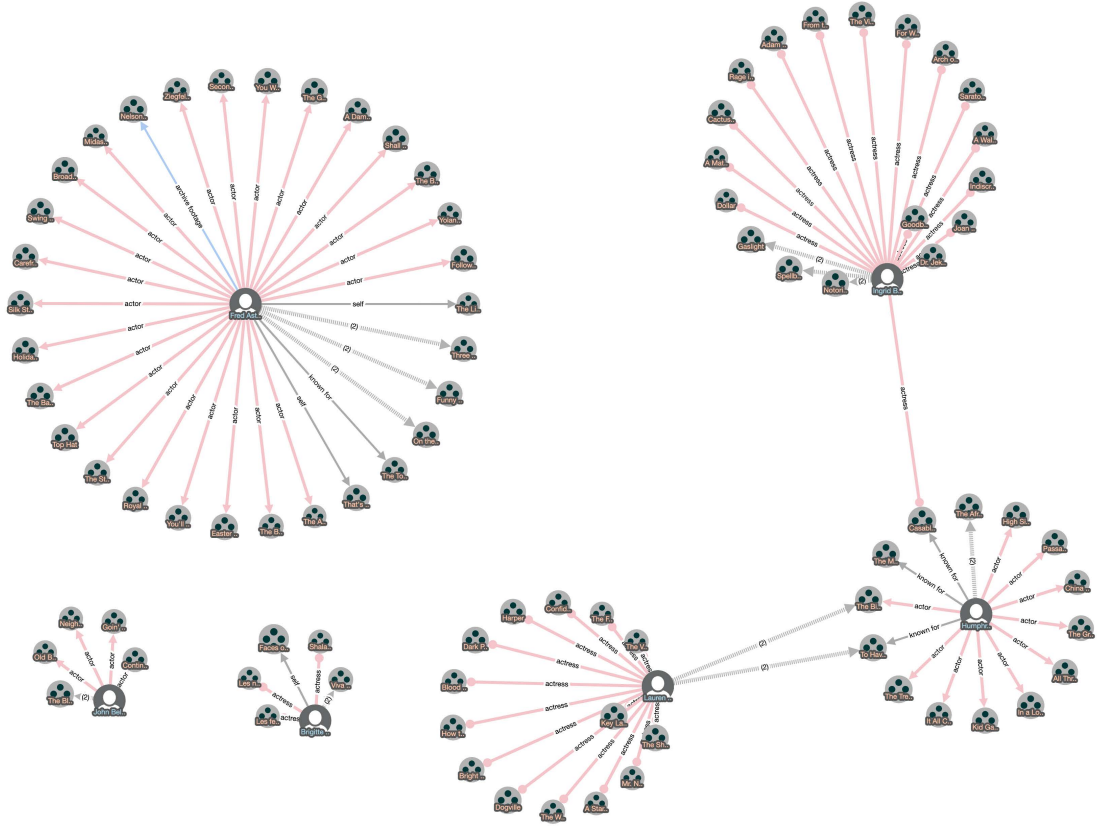


Figure 4.5: Sample graph with six clusters. Circles represent clusters. There are no compound nodes in this graph.

To better position any new nodes on an incremental layout, *Visuall* uses another Cytoscape.js extension called *cytoscape.js-layout-utilities* [29, 31]. When *Visuall* adds a new node without a position to the graph, *Visuall* calls the `placeNewNodes` method of this extension to position the newly added node. After new nodes have a position, the incremental layout works more effectively. Figure 4.6 shows the effect of placing new nodes in a smart manner using the extension. To mark the newly added graph elements, *Visuall* uses overlays (similar to selection but using a different color).

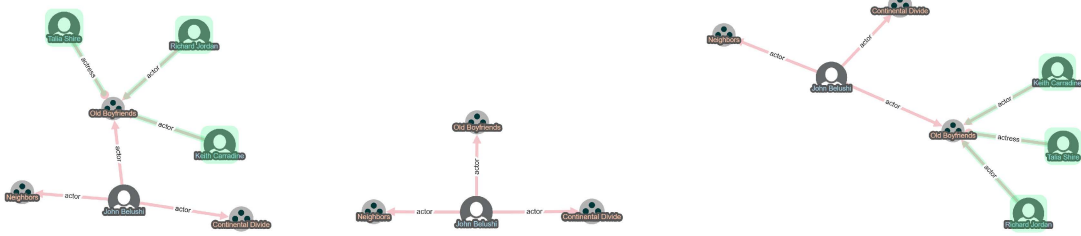


Figure 4.6: (middle) A portion of the IMDB data showing a particular actor playing in three different movies. (left) Effect of placing new nodes using the mentioned extension before applying an incremental layout when other actors/actresses of a movie are displayed. (right) Resulting layout when newly added actors/actresses are assigned default positions of (0,0), where the previous layout is drastically changed potentially destroying the user’s mental map.

For disconnected graphs, both CiSE and fCoSE extensions use the cytoscape.js-layout-utilities extension for packing components of the graph. Figure 4.7 shows the effects of packing components in the fCoSE algorithm. The drawing on the left shows the results when the layout packs the components to an aspect ratio of width/height of the canvas. Here, you see that the components span the whole drawing area nicely. The drawing on the right shows when it packs components to an undesired aspect ratio (i.e., 5). Here, the effect of packing components to aspect ratio is more evident since the aspect ratio is highly different from the aspect ratio of the graph canvas. Components are horizontally lined up to satisfy the aspect ratio requirement.

Lastly, *Visualall* also lets the user play with the layout by hand. The user can simply drag and drop nodes to change their position.

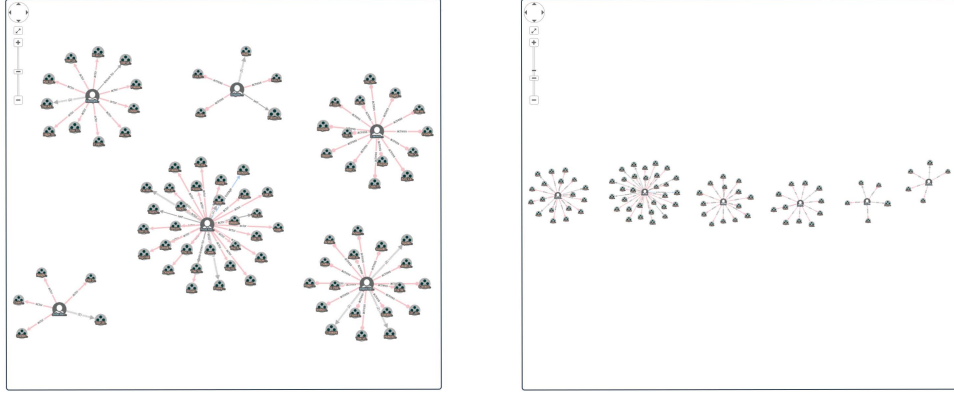


Figure 4.7: Effects of packing components with different aspect ratios

## 4.4 Clustering

When a graph is dense (high number of edges per node), visualization becomes too complex for a human. These kinds of graph visualizations are called “hairball” [9]. To handle such visual complexity, clustering the nodes can be helpful. Clustering or grouping aims to partition the nodes so that similar nodes will be in the same group. Mainly, the similarity of two nodes can be defined using two different approaches. Firstly, we can describe similarity based on data properties. For example, if the graph contains *Person* nodes, we can describe similarity based on names. We can say if the labels are similar, the nodes are also similar. This approach is using properties defined in *Property Graph Data Model*.

Secondly, one can cluster without considering data properties. In this case, the focus is on the topology of the graph. The connections between the nodes will simply decide the clustering. There are many algorithms in the literature. Louvain community detection [32] and Markov cluster algorithm [33] are such algorithms.

### 4.4.1 Compound Nodes

*Visuall* represents compound nodes with barrel shapes. *Visuall* uses a Cytoscape.js extension called *cytoscape.js-expand-collapse* [29] for handling related operations. To decrease the complexity of large graphs, the user might want to remove certain elements temporarily. To achieve this, *Visuall* enables users to *collapse* compound nodes. When the user collapses a compound node, the expand-collapse extension will remove the children of the compound node. We call this state *collapsed*. In this state, the children will not be in the graph but will stay in the memory. Later, the user can bring back the children by *expanding* the collapsed node. Figure 4.8 shows there are two compound nodes on the image above. On the image below, again you see two compound nodes. But this time, the compound node on the right does not have any children as that compound node is collapsed. The user can expand this collapsed compound node and see its children on demand. In this way, the user can interactively manage complexity.

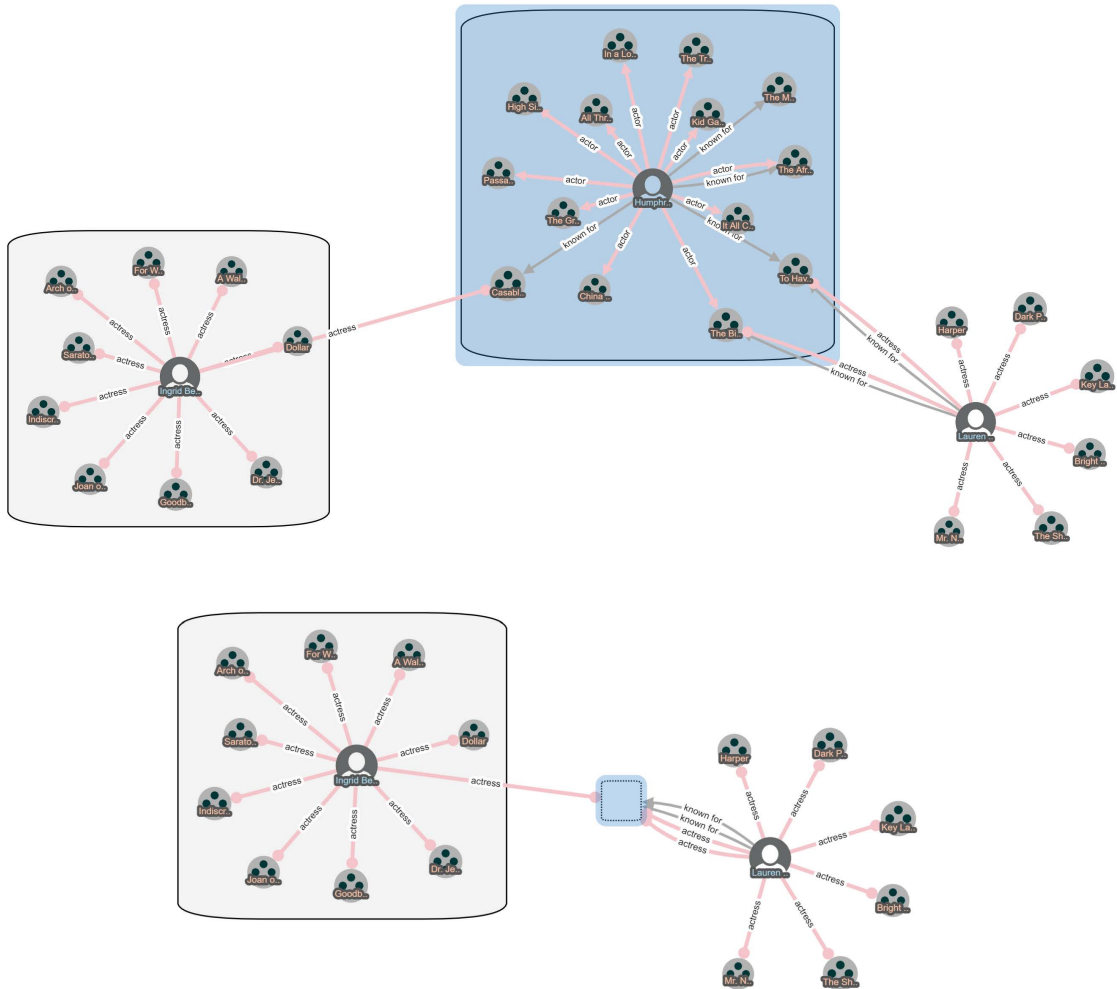


Figure 4.8: The drawing on top shows a graph with two compound nodes, both expanded. The one below shows the same compound graph with one of the compound nodes in collapsed state.

As can be seen in the drawing on top, of Figure 4.8, the compound nodes are not connected to any edges. But in the drawing at the bottom, you can see that some so-called *meta* edges are pointing to the collapsed compound node. When the user removes a node Cytoscape.js removes all the edges incident to the removed node. That changes the topology of the graph and might harm the mental map of the user. So to preserve the mental map of the user, the expand-collapse extension removes the nodes, then it modifies the graph so that removed edges point to the collapsed parent so that the user can understand there were

some edges connected to collapsed children.

*Visuall* not only allows programmatically adding/removing compound nodes with certain operations but also allows the user to add or remove compound nodes manually. The user can do so by selecting the nodes from the map. Note that the selected nodes can be compound as well. So, a compound node can contain other compound nodes. In this way, *Visuall* can express arbitrarily deeply nested groups. Figure 4.9 is a sample graph showing such nested groupings. After some nodes are selected, the user can add a compound node that will be the parent of selected nodes. Also, the user can select compound nodes and then remove them. That will not remove the children of the compound node. They will still exist in the graph, but *Visuall* will change their parents. Their parents will be the parent of the removed compound node.

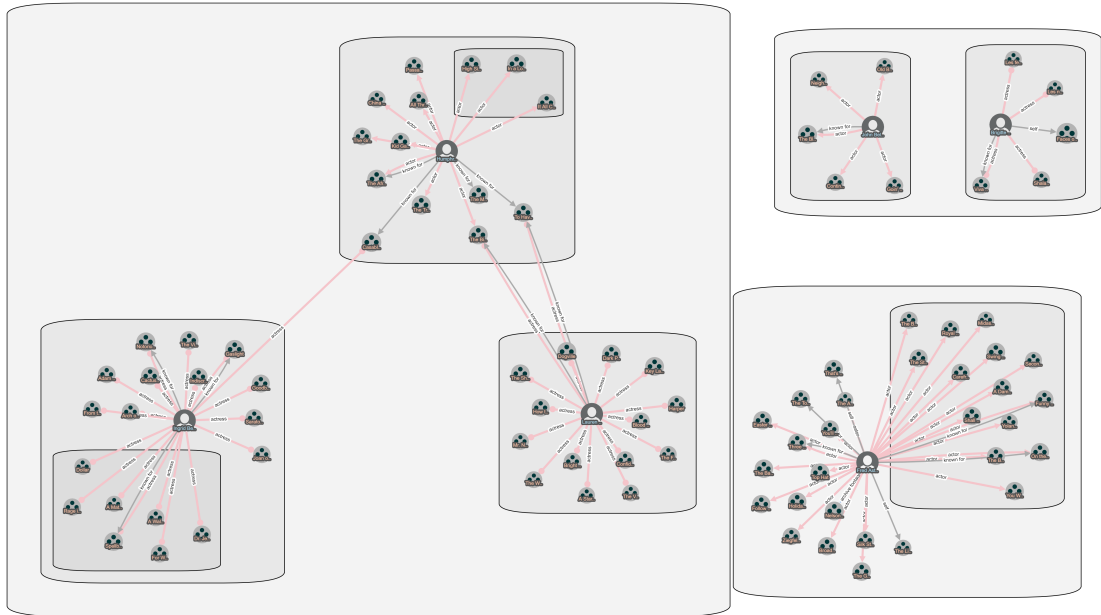


Figure 4.9: An example graph showing many nested groups

#### 4.4.2 Compound Edges

Sometimes there could be multi-edges or parallel edges between two nodes. If there are many edges between two nodes, displaying them can be computationally

expensive. So the user might like to represent all such multi-edges using a single, *compound* or *meta* edge as exemplified in Figure 4.10. When the user right-clicks to any of these edges, they will see a context menu. If the user chooses “Collapse”, the expand-collapse extension will represent the parallel edges with a single edge. When the user right-clicks to a compound edge and then chooses “Expand”, we will remove the compound edge and then bring back the parallel edges as before. Compound edge lines are styled as dotted for distinction. The compound edge line width is thicker than normal edges. In fact, we define the thickness with a simple logarithmic function  $3 + \log_2(n)$ , where  $n$  is the number of edges it contains. The label of the compound edge is the number of multi-edges it represents. This dynamic labeling signifies what to expect when the compound edge is expanded. Also, the user could find out the count of parallel edges by temporarily collapsing them.

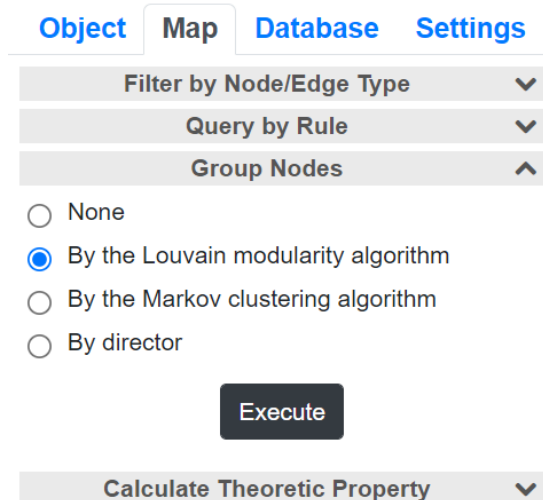


Figure 4.10: The drawing on the left shows multi-edges between two nodes. The one on the right shows the same graph after these multi-edges are collapsed and represented with a single compound edge

The expand-collapse extension also allows us to run an incremental layout after expand or collapse operations. Since expanding and collapsing changes the topology of the graph, we do not want to disturb the mental map of the user but we also want to position the new elements without cluttering the drawing. This extension uses specialized layout algorithms [34, 29] inspired by fish eye techniques to make room for new content before expanding.

### 4.4.3 Clustering Algorithms

Compound nodes and compound edges are tools for clustering and complexity management, which can be done manually or computationally. *Visuall* provides Louvain [32] and Markov Clustering [33] algorithms by default for topology based clustering of graph objects. These algorithms ignore properties of graph elements. For example, our movies data set contains movies with data properties like rating, number of votes, and run-time minutes. Algorithms considering data properties can be more effective and handy for a specific domain. So, *Visuall* lets developers inject their clustering algorithms by implementing them under *GroupCustomizationService*. Figure 4.11 shows the options for grouping as: ‘None’, ‘Louvain’, ‘Markov clustering’ and ‘By director’. ‘None’ will remove all clusters. ‘Louvain’ and ‘Markov clustering’ will generate groups using the topology of the graph. ‘By director’ is an example of a custom clustering algorithm. It generates groups using the director and the movies directed by the director. For different domains, different properties can be meaningful. For effective grouping, domain-specific algorithms might be necessary. These can be implemented by following the recommendations of domain experts.



The screenshot shows the 'Settings' tab of an application interface. At the top, there are four tabs: 'Object', 'Map', 'Database', and 'Settings', with 'Settings' being the active tab. Below the tabs, there are three expandable sections: 'Filter by Node/Edge Type' (collapsed), 'Query by Rule' (collapsed), and 'Group Nodes' (expanded). Under the 'Group Nodes' section, there are four radio button options: 'None', 'By the Louvain modularity algorithm' (which is selected), 'By the Markov clustering algorithm', and 'By director'. Below these options is a dark 'Execute' button. At the bottom of the settings panel, there is a 'Calculate Theoretic Property' section, which is currently collapsed.

Figure 4.11: Clustering choices for grouping nodes in the sample application

## 4.5 Object Inspection

When the user selects a graph element by tapping on it, we open the *Object* tab. In this tab, the user can see the properties of the selected element. Properties are like metadata for the element. Figure 4.12 shows an example, where, on the left, a node with the label “The Matrix” is selected and under inspection. The type of the node is “Title”. It has a number of properties such as “rating” which has a value of 8.7. On the right is an edge with label “actor”, which has only four properties. One property is named “characters” which has the value “Neo”.

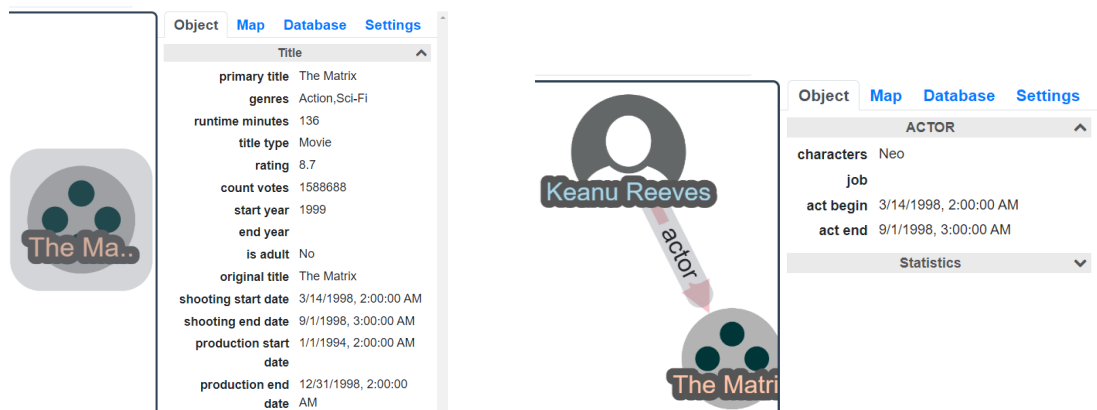


Figure 4.12: Inspecting properties by selecting a node (left) or an edge (right)

When multiple graph objects are selected, *Visuall* shows a table instead. In the drawing at the top of Figure 4.13, three nodes are selected. Hence, the inspector shows a table with three rows. Each row corresponds to one of the selected graph objects (nodes in this case). In the drawing at the bottom, three edges are selected, but the table contains four rows since one of the selected edges is a compound edge representing two edges.

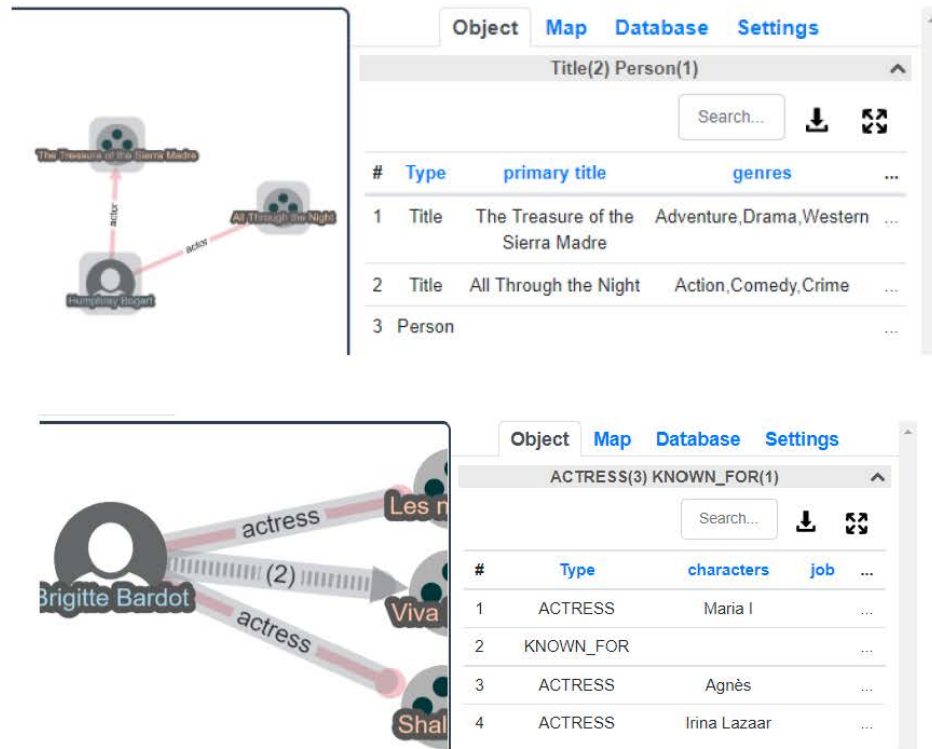


Figure 4.13: Inspecting properties by of multiple nodes (top) and multiple edges (bottom)

Inside the *Object* tab, there is a sub-tab named *Statistics*. The Statistics sub-tab shows the statistics about the currently loaded graph as a table. For example, Figure 4.14 lists each type of node and edge in the graph. A graph element can be selected or hidden. So it also shows the count for these states. At the bottom of the table, the last two rows show total node and edge counts. Statistics table provides the overall picture of the graph in terms of its size.

Object <b>Map</b> Database Settings				
Statistics ^				
<div>Search...</div> <div>↓ ↺</div>				
#	Type	Count	Selected	Hidden
1	Person	6	1	1
2	Title	86	8	14
3	SELF	3		2
4	ACTOR	40		26
5	ARCHIVE_FOOTAGE	1		1
6	Meta edge	8	3	
7	KNOWN_FOR	7		4
8	ACTRESS	33	4	
9	Node	92	9	15
10	Edge	92	7	33

Figure 4.14: *Statistics* sub-tab inside *Object* tab

Lastly, all the tables use the same Angular component with a common code base. This component has many available operations. For example, the user can search inside the table. They can import the table as a CSV (comma separated values) file. In addition, the table can be detached from its position and can be moved on the screen by dragging and dropping within the boundaries of the associated browser tab. Also, when you hover over a graph element, *Visuall* can emphasize the corresponding table row if such a row exists. This could also function in reverse. When the user hovers over a table row, *Visuall* can highlight corresponding graph elements if it exists.

## 4.6 Time-Based Filtering

For time-based filtering, we built a Cytoscape.js extension called “cytoscape.js-timebar”. The extension facilitates time-based filtering on the current graph. It does not add or remove any data but it shows or hides graph elements, based on their time properties. We assume that each node or edge should have some properties representing its lifespan called *object range*. If they do not exist, we use default values of  $-\infty$  and  $\infty$ , respectively for the begin and end datetimes of graph elements.

To make time-based filtering on the graph, the user should specify a time range. Since this range decides what is shown on the graph, we call this range *graph range*. The filtering uses the start and end datetimes of the graph range. Filtering will be based on the relation between graph range and object range of each graph element. Figure 4.15 shows three different possibilities in which these two ranges are related. An object can pass from filtering if its lifetime overlaps with or contained by or contains the filtering range. This is a user configurable setting.



Figure 4.15: Whether or not a particular graph element passes the time filtering and is shown is determined by one of these three options

Apart from showing or hiding graph elements based on graph range, the timebar also lets the user generate custom statistics. The statistics are calculated for the current unit time such as decades, years, and months. The time unit

will automatically change based on the current graph range. For example, if the graph range is 200 years, the time unit will be a decade. In Figure 4.16, there is a bar chart and a line chart. On the line chart, you can see a slider with two thumbs. The thumbs set the graph range. The bar chart shows statistics on the graph range. The line chart shows the statistics on a wider range.

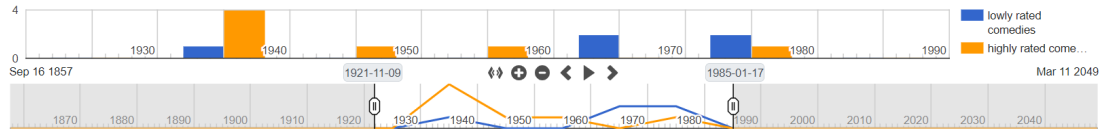


Figure 4.16: An example view from the timebar

Figure 4.16 shows two statistics. A statistic is a function that returns a number. *Visuall* gives users the ability to write their own statistics. A statistic could be a count of a certain type of element that satisfies specified conditions. Figure 4.17 shows the statistic counting the number of people who satisfy Algorithm 1.

---

**Algorithm 1** A sample timebar statistic combining a number of conditions

---

```

1: procedure SAMPLESTATISTIC( $x$ )
2:   return  $x.birth\_year > 1990$  AND  $x.death\_year < 2020$  AND
3:     ( $'a'$  in  $x.primary\_name$  OR  $'b'$  in  $x.primary\_name$ )
4: end procedure

```

---

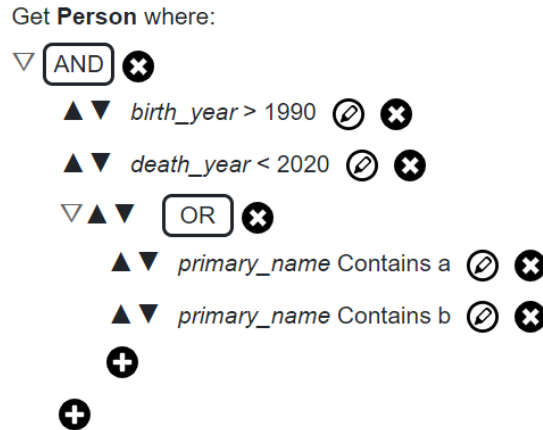


Figure 4.17: A timebar statistic counting the number of “People” who satisfy defined conditions in Algorithm 1.

As exemplified in Figure 4.17, a tree structure combining conditions with AND and OR logical operators can be used to generate arbitrarily deeply nested statistics. The user does not need to know any kind of programming knowledge for this, and can use the provided simple user interface components such as drop-down and input.

In the example of Figure 4.17, all the conditions are on data properties, which is rather standard. Here, the user can also define conditions related to some graph topological properties. For example, the user can write a condition based on the degree of nodes. Figure 4.18 shows a condition for people who directed more than three movies. *Visuall* counts the number of “DIRECTOR” edges connected to a node. This is the *degree* of a node considering only a certain type of edge. So a statistic can be a combination of both data properties and graph topological properties.



Get **Person** where:  
*DIRECTOR* > 3  

Figure 4.18: A timebar statistics condition for counting people who directed more than three movies

In addition to counting elements, *Visuall* can also find an aggregated value of a specific property. Figure 4.19 shows a statistic that gives the total sum of ratings of titles that have more than 10 actors and have more than 100 votes. The condition related to counting the number of actors is related to graph topology but the condition related to votes is related to data properties. We combine two conditions and then we aggregate the ratings of fulfilling objects.

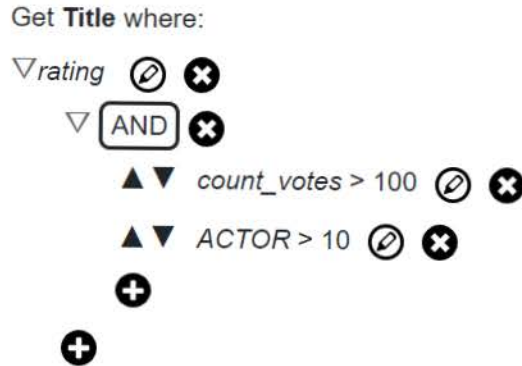


Figure 4.19: A statistic that gives the total sum of ratings of titles that satisfy the specified conditions

Additionally, *Visuall* gives the ability to export statistics as JSON files. So the user can save statistics, reuse them later, and also share them with their colleagues.

## 4.7 Querying Data

As relational data on a server side is typically too big, the user needs to initially form a graph of interest by performing a query. The constructed graph as a result of the query can be too complex or too big. The user might also need to query the existing client-side graph to see the graph elements of interest. So *Visuall* can query both the client-side data and database.

### 4.7.1 Query By Rules

This component is expected to be one of the most used components in *Visuall*. Using this component, the user can query both the database and client-side. *Visuall* uses the same Angular component to define timebar statistics and ruled queries. Their logic is very similar. Timebar statistics works only on the client-side but ruled queries can also work on the database. Timebar statistics generate

JavaScript code that will run on the client-side. Ruled queries can generate JavaScript code and also Cypher code that will run on the Neo4j database side.

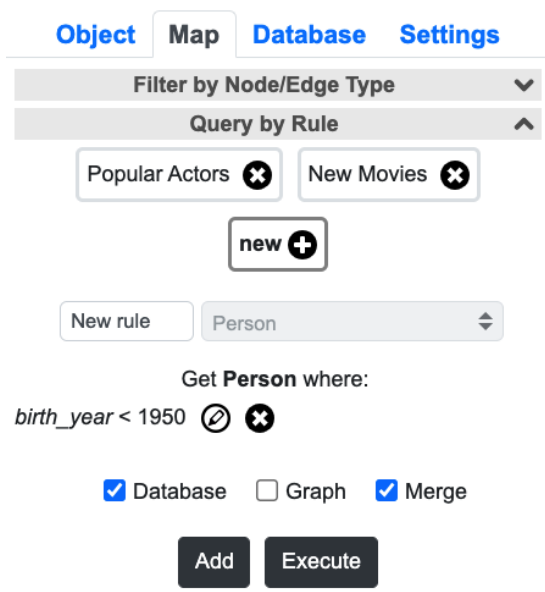


Figure 4.20: A sample screen for query by rule

As shown in Figure 4.20, the user can see previously saved queries (e.g., “Popular Actors” and “New Movies”). By clicking on the name of the saved query, the user can view, update or execute the query. If the “Database” option is checked, *Visuall* will execute the query on the database side. In any case, we will show the data as a table. If the “Graph” option is checked, we will also show the data as a graph in the drawing canvas. So, the resulting relational data can be shown as a table and as a graph simultaneously. If the “Merge” option is checked, we will merge the new data into the existing graph. Otherwise, the existing graph is replaced with the query result.

### 4.7.2 Graph-Based Queries

*Visuall* provides three types of graph or traversal based queries: *neighborhood*, *graph of interest* (GoI), and *common target/regulators* (CTR) based on biological pathway queries in [35]. We adapted them to be domain-independent, generic

graph queries and implemented them as Neo4j *user-defined procedures*. Fundamentally each query is a function defined in Java. Original algorithms are graph traversal based, solely working on/with the topology of the graph. To give more flexibility, we modified these traversals to use data properties. Hence, the traversals will check the data properties of elements and proceed accordingly, leaving out node/edge types that are not desired during traversals.

Figure 4.21 shows a sample neighborhood query and its results. Neighborhood query will bring the limited neighborhood of a given node set. It is a breadth-first search (BFS) with a length limit. The “length limit” parameter will determine the number of hops we can go further. The “Directed” parameter is used to decide whether to consider the direction of edges or not. From “Object Types to Consider”, the user can indicate the graph object types to consider. By default, all the types are included. To make the graph simpler, the user might want to ignore certain types. At the bottom right, results are shown in a tabular manner. We always show the query results as a table.

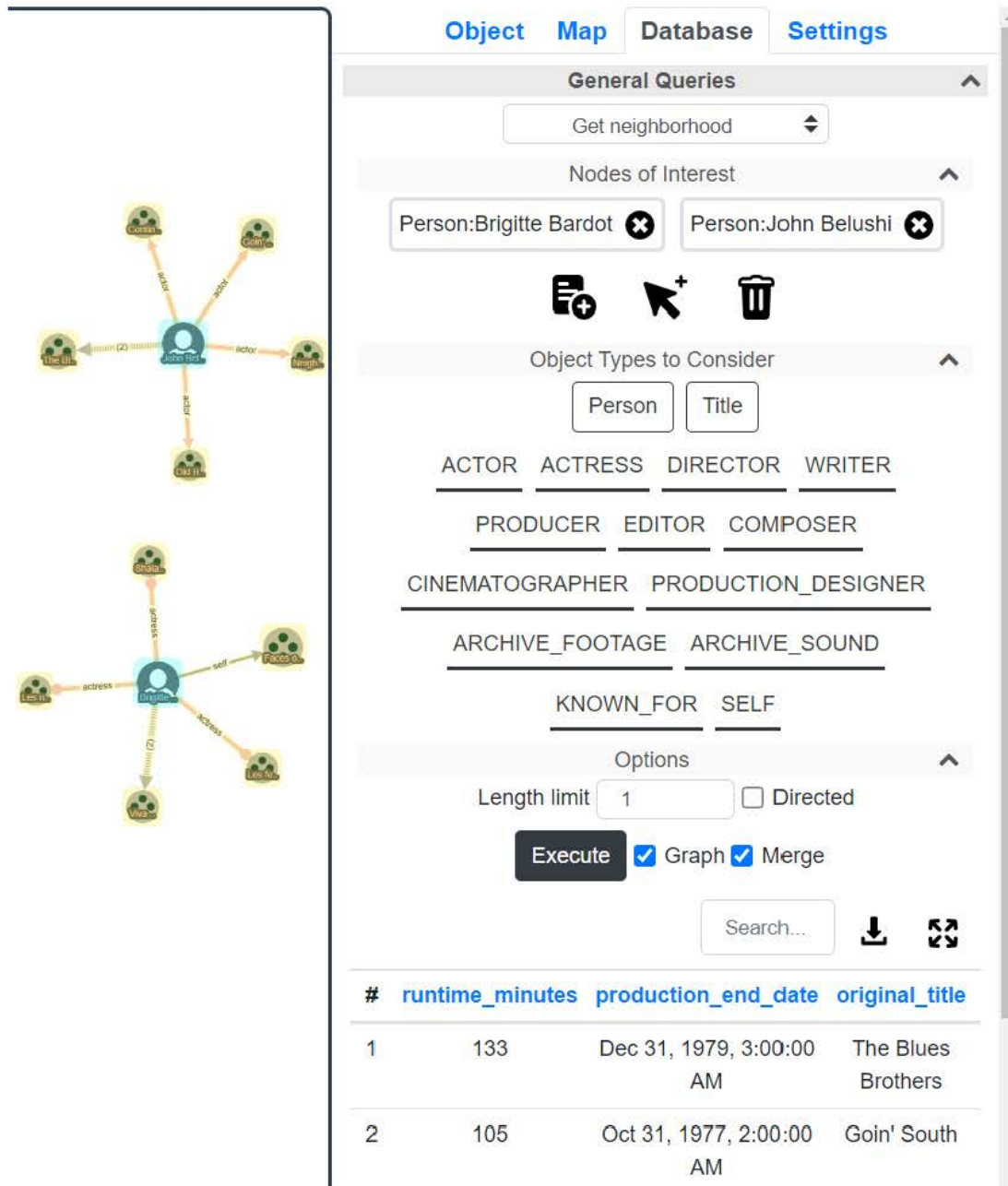


Figure 4.21: A sample neighborhood query and its results

GoI and CTR queries are more complex algorithms. These algorithms were originally designed by Dogrusoz et al. [35] to query compound graph-based pathway databases. The original algorithms are developed to work on compound graphs. Neo4j does not store data as compound graphs. So the algorithms were

adapted to work with simple graphs where compound relations are represented with special inclusion edges/relations. In addition, we made some optimizations in the CTR algorithm. These optimizations make CTR query significantly faster on some practical examples.

Fundamentally, the GoI query will take a set of nodes and a length limit as parameters. The nodes will be used as sources to initiate the traversal. The algorithm will find a minimal sub-graph that contains the source nodes and the paths between. The length of a path between two source nodes should be less than or equal to the length limit parameter. Similar to the neighborhood query, this query also takes “Directed” and “Object Types to Consider” as parameters.

To find the minimal sub-graph, GoI makes breadth-first searches from the source nodes. One BFS is executed for incoming direction, another BFS is executed for outgoing direction. BFS calls store distances for each direction. In the end, GoI will check if the sum of two distances passes the length limit.

---

**Algorithm 2** Graph of Interest (GoI) algorithm

---

```

1: procedure GoI(ids, ignoredTypes, lengthLimit, dir)
2:   Initialize all nodeLabels and edgeLabels as 0
3:   params  $\leftarrow$  nodeLabels, edgeLabels, ids, ignoredTypes, lengthLimit, dir
4:   C = GoI_BFS(params, OUTGOING)  $\cup$  GoI_BFS(params, INCOMING)
5:   for q  $\in$  C do
6:     if q.labelForward + q.labelReverse  $\leq$  lengthLimit then
7:       R  $\leftarrow$  R  $\cup$  q
8:     end if
9:   end for
10:  removeOrphanEdges(R)
11:  purify(R)
12:  removeOrphanEdges(R)
13:  return R
14: end procedure

```

---

Algorithm 2 summarizes the GoI algorithm. It takes the union of two GoI\_BFS calls and then filters based on length limit, similar to the way it is done in [35].

Algorithm 3 summarizes the GoI\_BFS algorithm. The algorithm is a special BFS that keeps *forward* and *reverse* labels for each node and edge. A label

---

**Algorithm 3** Graph of Interest breadth-first search (GoI-BFS) algorithm

---

```
1: procedure GoI-BFS(nodeLabels, edgeLabels, ids, ignoredTypes,  
   lengthLimit, direction, isDirected, isFollowLabeled, unignorable)  
2:   nodeSet  $\leftarrow$  edgeSet  $\leftarrow$  visitedEdges  $\leftarrow$   $\emptyset$   
3:   Add all node ids in ids to queue Q  
4:   while Q  $\neq \emptyset$  do  
5:     n1  $\leftarrow$  Q.remove()  
6:     for e  $\in$  n1.UnignoredEdges(direction, isDirected, ignoredTypes) do  
7:       n2  $\leftarrow$  e.getOtherNode(n1)  
8:       b1  $\leftarrow$  isIgnoreNode(n2, unignorable, ids, ignoredTypes)  
9:       b2  $\leftarrow$  isIgnoreEdge(e, visitedEdges, isFollowLabeled)  
10:      if b1 OR b2 then  
11:        continue  
12:      end if  
13:      visitedEdges.add(e)  
14:      if edgeLabels.get(e) does not exist then  
15:        labelEdge  $\leftarrow$  (lengthLimit + 1, lengthLimit + 1)  
16:      end if  
17:      if nodeLabels.get(n1) does not exist then  
18:        labelN1  $\leftarrow$  (lengthLimit + 1, lengthLimit + 1)  
19:      end if  
20:      if direction is OUTGOING then  
21:        labelEdge.fwd  $\leftarrow$  labelN1.fwd + 1)  
22:      else if direction is INCOMING then  
23:        labelEdge.rev  $\leftarrow$  labelN1.rev)  
24:      end if  
25:      edgeLabels.put(labelEdge), nodeLabels.put(labelN1)  
26:      nodeSet.add(n2), edgeSet.add(e)  
27:      if nodeLabels.get(n2) does not exist then  
28:        labelN2  $\leftarrow$  (lengthLimit + 1, lengthLimit + 1)  
29:      end if  
30:      if labelN2 > labelN1 + 1 on direction then  
31:        set labelN2 to labelN1 + 1 on direction  
32:        if labelN2 < lengthLimit on direction AND ids  $\not\subset$  n2 then  
33:          Q.add(n2)  
34:        end if  
35:      end if  
36:      nodeLabels.put(labelN2)  
37:    end for  
38:  end while  
39:  return nodeSet, edgeSet  
40: end procedure
```

---

represents the minimum length to any of the source nodes (given as `ids` in the pseudo-code) in forward direction or reverse direction.

Similar to GoI, CTR aims to find common downstream or common upstream of a set of source nodes (i.e., whether the traversals follow the direction of edges or go in reserve order, respectively). CTR also uses the same parameters such as “Directed” and “Object Types to Consider”. CTR additionally takes another parameter to denote whether it should work downstream or upstream. If it works downstream (upstream), it will find common targets (regulators). In CTR, the length limit parameter represents the maximum length between a source node and a target/regulator. So in CTR, the length of a path between two source nodes should be less than or equal to two times the length limit parameter.

In the original CTR algorithm, the traversal starts from the source nodes. First, target/regulator nodes are found. Then the paths between the target/regulators and the source nodes are found with two other BFS. Two other breadth-first searches keep some metadata for visited nodes and edges. In each BFS, the nodes and edges are marked as they are visited. In the first BFS, the traversal starts from the source nodes. In the second BFS, on the other hand, the traversal starts from the target/regulator nodes. Each BFS forms a sub-graph of reached graph elements followed by calculation of their intersection.

As an optimization in the original CTR algorithm, we basically base the second BFS on the previously marked sub-graph, rather than the entire original graph. In the end, we need to find the intersection of these two sub-graphs. So we can limit our scope to the firstly constructed sub-graph on the second BFS. This may potentially significantly reduce the search space, drastically decreasing the execution time.

Here we also observed that the size of the first sub-graph is critical. We find a sub-graph by making a BFS that starts from some nodes. The size of a sub-graph depends on the number of starting nodes. So the number of start nodes affects the size of a founded sub-graph. In the original algorithm, the first BFS was starting from the source nodes, the second BFS was starting from the target/regulator

nodes. We know that changing the order does not change the results. So we think we can start the first BFS with the node set with fewer nodes. The number of source nodes is a parameter given by the user. Typically it could be as low as two but it could be a much bigger set. In the end, to keep the size of the first sub-graph minimum, we check the number of target/regulator nodes and the number of source nodes. If the number of source nodes is smaller, we construct the first sub-graph from the source nodes, else we use the target/regulator for the first BFS.

Algorithm 4 summarizes the CTR algorithm (options related parameters are left out for brevity). It starts traversals from *ids* (the source nodes). It finds a candidate set *C* and result set *R*. *R* is a set of nodes that are common targets/regulators. Then it should find the paths to *R* from *ids*. To find the paths, it performs two `GoI_BFS` calls. The last parameter in these calls represents *isFollowLabeled*. In the first call, it is `false`, whereas in the second call it is `true`. The second `GoI_BFS` call will follow the elements labeled in the first call. In the last part, it applies filtering based on labels and returns the founded paths.

The time complexity of all three queries depends on the number of neighbors a node has. The worst-case time complexity of the algorithms is exponential in the number of source nodes. Because of this exponential nature, these algorithms should be carefully used with a few number of source nodes and a low limit in practice. In some practical examples, we observed that our optimization in the CTR algorithm provides nearly a tenfold faster execution times.

Note that the implementation of these three general queries (neighborhood, CTR, and GoI) were done in Java specifically for Neo4j. If the user wants to use some other database provider, they should be able to quickly adapt these code for the specific provider.

Although the users can build complex and custom queries using “Query By Rule”, this will not be sufficient for most domain experts (such as fraud detection and cybersecurity operators) and will require dynamic traversal-based queries

---

**Algorithm 4** Common Target Regulator (CTR) algorithm

---

```
1: procedure CTR(ids, ignoredTypes, lengthLimit, direction)
2:    $R \leftarrow C \leftarrow \emptyset$  ▷ R is result set, C is candidate set
3:   for  $id \in ids$  do
4:      $C \leftarrow C \cup CS\_BFS(id, ignoredTypes, lengthLimit, direction)$ 
5:   end for
6:   for  $q \in C$  do
7:     if  $q.labelReached = |ids|$  then
8:        $R \leftarrow R \cup q$ 
9:     end if
10:  end for
11:  set all edgeLabels to (0,0)
12:  if direction is OUTGOING then
13:    set nodeLabels to (0, lengthLimit + 1) for ids
14:    set nodeLabels to (lengthLimit + 1, 0) for R
15:    if  $|ids| < |R|$  then
16:       $o1 \leftarrow GoLBFS(..., ids, OUTGOING, false)$ 
17:       $o2 \leftarrow GoLBFS(..., R, INCOMING, true)$ 
18:    else
19:       $o1 \leftarrow GoLBFS(..., R, INCOMING, false)$ 
20:       $o2 \leftarrow GoLBFS(..., ids, OUTGOING, true)$ 
21:    end if
22:  else if direction is INCOMING then
23:    set nodeLabels to (lengthLimit + 1, 0) for ids
24:    set nodeLabels to (0, lengthLimit + 1) for R
25:    if  $|ids| < |R|$  then
26:       $o1 \leftarrow GoLBFS(..., ids, INCOMING, false)$ 
27:       $o2 \leftarrow GoLBFS(..., R, OUTGOING, true)$ 
28:    else
29:       $o1 \leftarrow GoLBFS(..., R, OUTGOING, false)$ 
30:       $o2 \leftarrow GoLBFS(..., ids, INCOMING, true)$ 
31:    end if
32:  end if ▷ Else part handled similar to INCOMING with undirected
33:  for  $q \in o1 \cup o2$  do
34:    if  $q.labelForward + q.labelReverse \leq lengthLimit$  then
35:       $R2 \leftarrow R2 \cup q$ 
36:    end if
37:  end for
38:   $R2 = R2 \cup ids \cup R$ 
39:  purify(R2)
40:  removeOrphanEdges(R2)
41:  return R2
42: end procedure
```

---

with user specified parameters. Hence, we let the users write their own custom queries. We expect the developer to provide a separate Angular component for each new query. They can, however, import and use some shared components such as tables for such custom queries.

### 4.7.3 Table View

In data visualization, one of the most commonly used methods to display information is via tables. *Visuall* uses an Angular component to provide table views in many places such as the “Query By Rule”, “Object” tab, and “General Queries”. Visualizing data as a graph and also showing the same data as a table can give users a more comprehensive viewpoint.

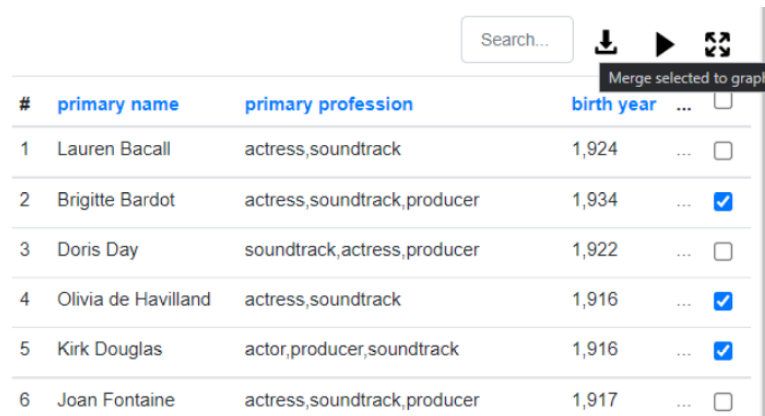
If done with some type of pagination, table views are especially useful as a way to first show query results to the user without having to automatically lay out and render large graphs. *Visuall* provides support for pagination for both server-side and client-side. In server-side pagination, we only bring one page of data from the server. The user is allowed to set a maximum page size. With server-side pagination, we also bring the total count of available records so that the user will know how many pages exist in the database. When the user wants to bring the next page, *Visuall* will query the database again and bring the next set of records to fill the next page.

In situations where querying the database too frequently takes a long time, the user might set the page size to a higher value such as a 100. Then, the user will not need to make frequent database queries but suddenly showing 100 records as a table or graph might be overwhelming. To give users an alternative, *Visuall* also provides a client-side pagination strategy. In client-side pagination, in addition to page size, there is another parameter called *page limit*. This time, the number of records *Visuall* brings from the database is up to the multiplication of page size and page limit. *Visuall* will only bring all the data from the database at once but shows it one page at a time. If the user wants to bring the next page, *Visuall* will not query the database again but instead it will just bring the next slice from

the previously saved data in the memory. This strategy is faster at the cost of increased memory utilization. To give the user utmost flexibility, *Visuall* allows the user to dynamically switch between client-side and server-side pagination.

Searching and sorting tables are common features in many applications. The tables in *Visuall* let the user search by typing a text and sort by a column of the table. Search and sort operations might result in database actions. If server-side pagination is used, search and sort operations in “Query By Rule” and “General Queries” need data from the database. In this case, the operations are handled on the database side. On the other hand, the table inside the “Object” tab uses client-side data. Since it does not need data from the database, the operations are completely handled on the client-side.

The user can also manage the complexity of graphs manually with tables. For example, the user might be already working with a complex graph. In this case, the user might want to bring new graph elements manually by selecting. The user can select certain rows by checking the check-boxes, and then clicking on the “Merge selected to graph” icon. You can see this use case in Figure 4.22.



#	primary name	primary profession	birth year	...	
1	Lauren Bacall	actress,soundtrack	1,924	...	<input type="checkbox"/>
2	Brigitte Bardot	actress,soundtrack,producer	1,934	...	<input checked="" type="checkbox"/>
3	Doris Day	soundtrack,actress,producer	1,922	...	<input type="checkbox"/>
4	Olivia de Havilland	actress,soundtrack	1,916	...	<input checked="" type="checkbox"/>
5	Kirk Douglas	actor,producer,soundtrack	1,916	...	<input checked="" type="checkbox"/>
6	Joan Fontaine	actress,soundtrack,producer	1,917	...	<input type="checkbox"/>

Figure 4.22: An example of manually selecting graph elements from the table view and adding them to the graph view

When the same data is being shown both as a graph and a table at the same time, the user might like to observe the mapping between table rows and graph elements dynamically. *Visuall* will emphasize this mapping by highlighting graph

elements or table rows, should the setting named “Emphasize on hover” is enabled. If it is enabled, hovering on a table row will emphasize the corresponding graph element, and also hovering a graph element will emphasize the corresponding table row. Additionally, this also works for the “Statistics” table. For instance, as shown in Figure 4.23 if you hover on the “Person” type which is a node type in the sample graph, you will see all the “Person” nodes highlighted.

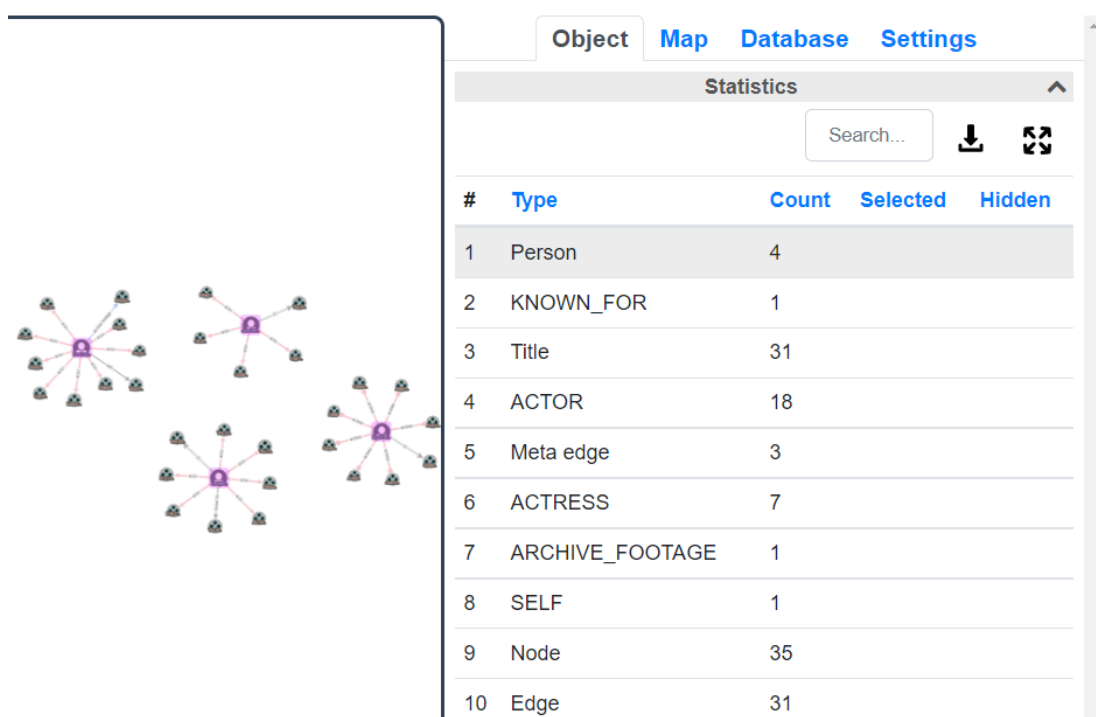


Figure 4.23: Emphasize corresponding graph elements on graph

# Chapter 5

## Testing & Evaluation

In order to make *Visuall* more robust and useful, its functionality has been tested and performance was evaluated. To test functionality, end-to-end (E2E) tests were written. For performance, we measure the execution times of certain use cases with graphs of varying sizes.

### 5.1 Implementation and Testing

*Visuall* is a web-based project. It consists of codes written in HTML, JavaScript, Typescript, and CSS. Since it uses Angular, the majority of the codes are written in Typescript. To handle code complexity, we embraced a modular architecture. We did not exceed 1100 lines in a code file. We aim to minimize code duplication as much as possible. We give great importance to the readability of code so that the codes are formatted and variables have explanatory names.

The need to make changes in software is unavoidable due to bug fixes and new features. As the software gets complicated, fixing a bug might cause another new bug or addition of new features might introduce new problems [36]. Sometimes, a change might cause a bug in a seemingly irrelevant place. Hence, to ensure functionality and robustness, the software should be tested thoroughly. There

are various types of tests such as unit tests, integration tests, and end-to-end tests [37]. We implemented E2E tests to ensure the functionality of the software as the whole focus is on the end user’s point of view [38]. Although they are useful and valuable, writing E2E tests can be time-consuming and cumbersome [39]. To make implementation of E2E tests easier, we used an E2E testing framework called *Cypress* [40]. Cypress enables us to programmatically control the browser. By this means, we test *Visuall* as if an end-user clicks on buttons and keyboard.

For *Visuall*, we implemented 31 E2E tests. Some tests are simple scenarios such as checking if a graph can be saved as an image, whereas others are more complex such as checking if a nested query rule can be created.

An example test case is checking whether the user can create and run a nested rule inside the “Query By Rule” screen. Figure 5.1 shows Cypress script for this test. The script creates a rule which can be summarized with expression

```
x.primary_name contains ‘Jo’ AND (x.ACTRESS > 3 OR x.ACTOR > 3)
```

It means the elements whose “primary\_name” contains the string ‘Jo’ and who have more than three “ACTOR” incident edges or more than three “ACTRESS” incident edges. In the end, the script checks whether the results comply with the executed rule.

Another example test case is checking whether the user can make a grouping with Louvain community detection and Markov clustering algorithms. Figure 5.2 shows Cypress script for this test. The script fetches some sample data. Then it calculates groups using the Louvain clustering algorithm. After that it deletes the groups. It then calculates groups using the Markov clustering algorithm. It finally deletes the groups again.

```

it('TC6: Should be able to run a nested rule', () => {
  beginQueryByRule();
  cy.get('button:visible').contains('AND').click();
  addPropertyRule('primary_name', 'contains', 'Jo');

  // start inner OR
  cy.get('img[title="Add"]:visible').click();
  cy.get('button:visible').contains('OR').click();
  cy.get('img[title="Add"]:visible').eq(0).click();
  cy.get('button:visible').contains('Condition').click();
  addPropertyRule('ACTRESS', '>', '3');

  // second rule of inner OR
  cy.get('img[title="Add"]:visible').eq(0).click();
  cy.get('button:visible').contains('Condition').click();
  addPropertyRule('ACTOR', '>', '3');
  click2Execute(true);

  cy.window().then((win) => {
    const canGetAllJos = win.cy.$("[.Person][primary_name *='Jo']")
    expect(canGetAllJos).to.eq(true);
  });
});

```

Figure 5.1: Cypress script to test whether the user is be able to run a nested query by rule

```

it('TC1: Can group with Louvain & Markov using compounds', () => {
  navbarAction('Data', 'Sample Data');
  openSubTab('Group Nodes');

  groupBy('By the Louvain modularity algorithm', true);
  groupBy('None', false);
  groupBy('By the Markov clustering algorithm', true);
  groupBy('None', false);
});

```

Figure 5.2: Cypress script to test whether the user is able to generate groupings and then remove them

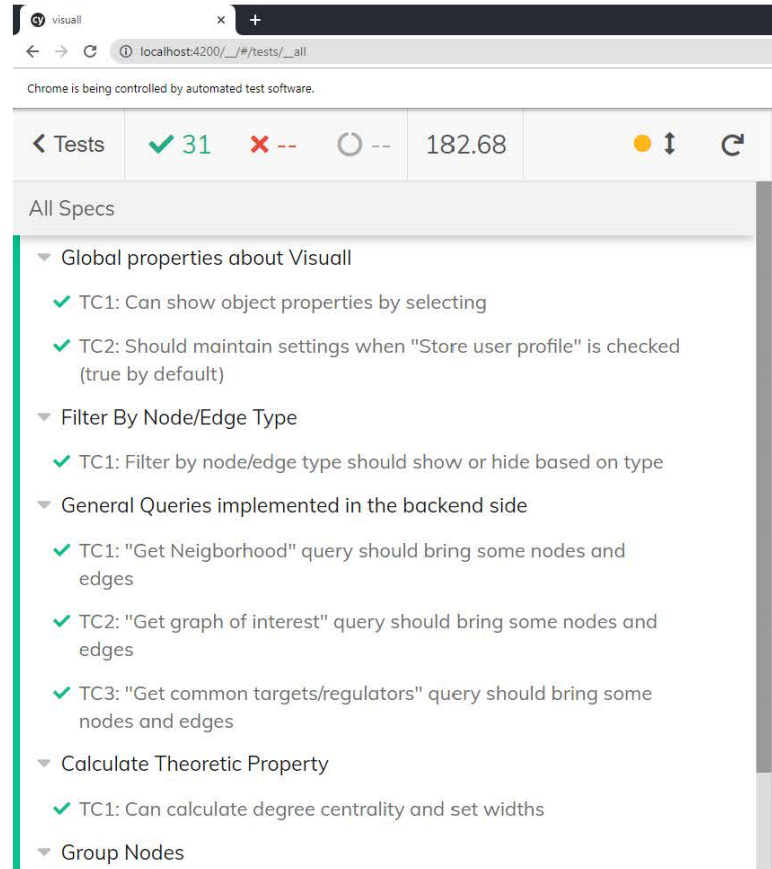


Figure 5.3: After all 31 *Visuall* E2E tests are finished in Cypress

## 5.2 Evaluation

*Visuall* is meant to be used in an interactive manner by the end-users. To achieve that, operations should take at most a few seconds to avoid the user giving up on the application. Depending on the operation, *Visuall* might take up some time on the database side and/or on the client-side. Database operations typically take more time, also subject to the particular database management system as well as the operation itself.

For the client-side operations, we measured execution times on graphs with 50, 100, 200, 400, and 800 nodes. For larger graphs, we assume the complexity will be managed by either tabulating results in text first or by applying other complexity

management operations such as clustering and collapsing [29]. In each graph, the number of edges is approximately the same as the number of nodes as expected with most real-life graphs. To make time measurements error-prone, we take the average of three measurements.

Below are the details of specific performance evaluation for varying operations.

### 5.2.1 Graph-Theoretical Properties

Calculating graph-theoretical properties might be one of the most time-consuming operations. It depends on graph size and the theoretical property. Firstly, calculations and then showing the results of calculations on the graph takes time. We evaluated four different theoretic properties: degree centrality, closeness centrality, betweenness centrality, and page rank. Figure 5.4 shows execution times for various graphs. Except for closeness centrality, all the execution times are approximately less than ten seconds.

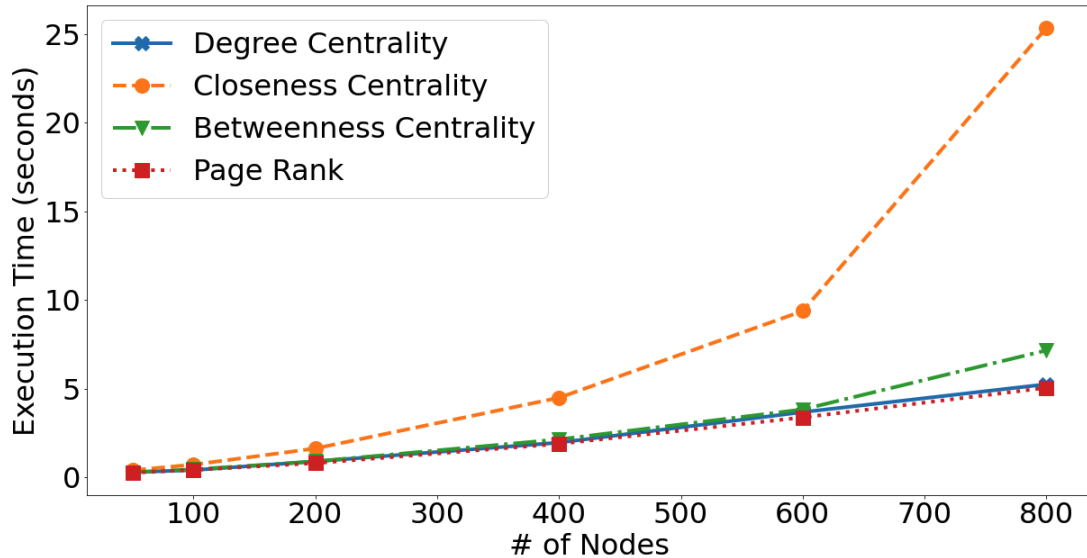


Figure 5.4: Execution times of calculating various graph theoretical properties and then showing them on the graph

For calculations of theoretical properties, we use Cytoscape.js [17] library functions. For degree centrality, Cytoscape.js uses the algorithm of Opsahl et al. [41]. For betweenness centrality, it uses the algorithm of Brandes et al. [42]. The time complexity of the degree centrality algorithm is  $O(|V| + |E|)$ . Here  $|V|$  and  $|E|$  denotes the number of vertices and number of edges, respectively. The time complexity of betweenness centrality and closeness centrality are  $O(|V| \cdot |E|)$ . The time complexity of the implementation of page rank inside the Cytoscape.js is  $O(k \cdot |V| \cdot |E|)$ , where  $k$  is the number of iterations.

Figure 5.4 shows the test results mostly comply with theoretic estimation except for page rank. Even though the page rank algorithm has quadratic theoretical run-time complexity, experiments show a linear behavior. Since our focus is on small to medium graphs, we do not consider large graphs. Due to the overhead of the algorithm and not considering large graphs, quadratic behaviour is not observed. In addition, we executed page rank with default parameter values. So default values might cause faster executions.

### 5.2.2 Complexity Management Through Hide-Show

Showing or hiding elements based on type might be useful to make fast filtering. After elements from a specific type are shown or hidden, *Visual* will make an automatic layout by default. Showing or hiding elements and then doing layout takes time. We measured four different use cases: hiding a node type, showing a previously hidden node type, hiding an edge type, and showing a previously hidden edge type. We used the ‘Person’ node type and ‘ACTOR’ edge type inside the sample movies graph. Figure 5.5 shows the executions times. Even on the largest graphs, the operations take less than four seconds.

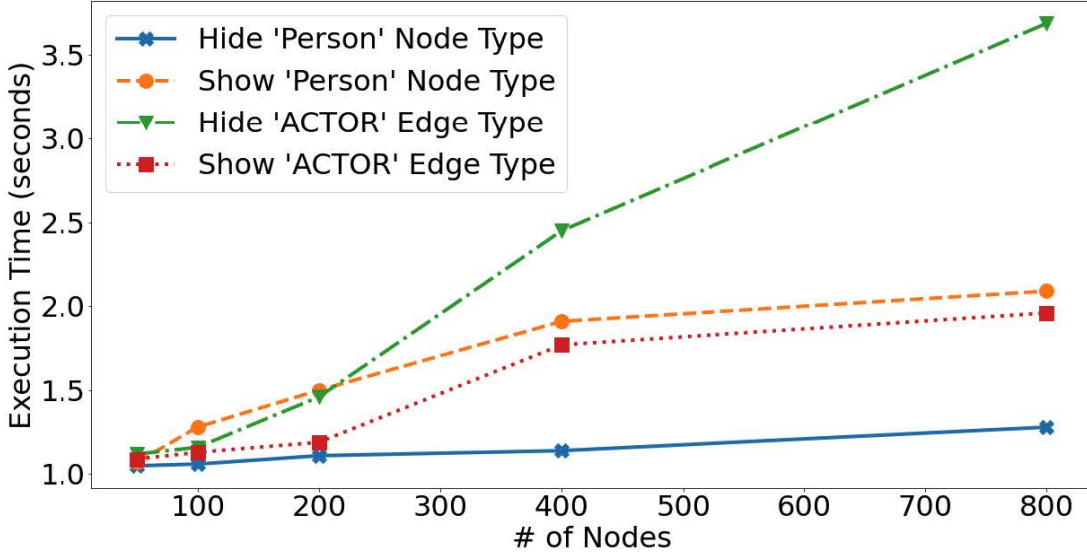


Figure 5.5: Execution times of showing and hiding elements from a certain type

Except for the layout, all the operations will be linear in the number of elements. The execution of the layout depends on the number of nodes and edges available. But as a result, it is satisfactory in terms of performance that the hide/show operations of the graphs finish in less than four seconds in these sizes.

### 5.2.3 Clustering

Clustering graph elements can be very costly in terms of time. It depends on the size of the graph and the clustering algorithm. It also depends on how the clusters are represented. In *Visuall*, clusters can be represented with compound nodes or circles. After the clustering algorithm is executed and the clusters are expressed, *Visuall* will run an automatic layout algorithm by default. We measured the sum of execution times of these operations. We make experiments using Louvain clustering and Markov clustering with both circular cluster representation and compound node representation. Figure 5.6 shows the execution times. For circular representations, the CiSE layout algorithm is used. For compound representations, the fCoSE layout algorithm is used. The fCoSE performs slightly faster than CiSE. Markov clustering takes at most 16 seconds. Louvain clustering

takes at most 6 seconds.

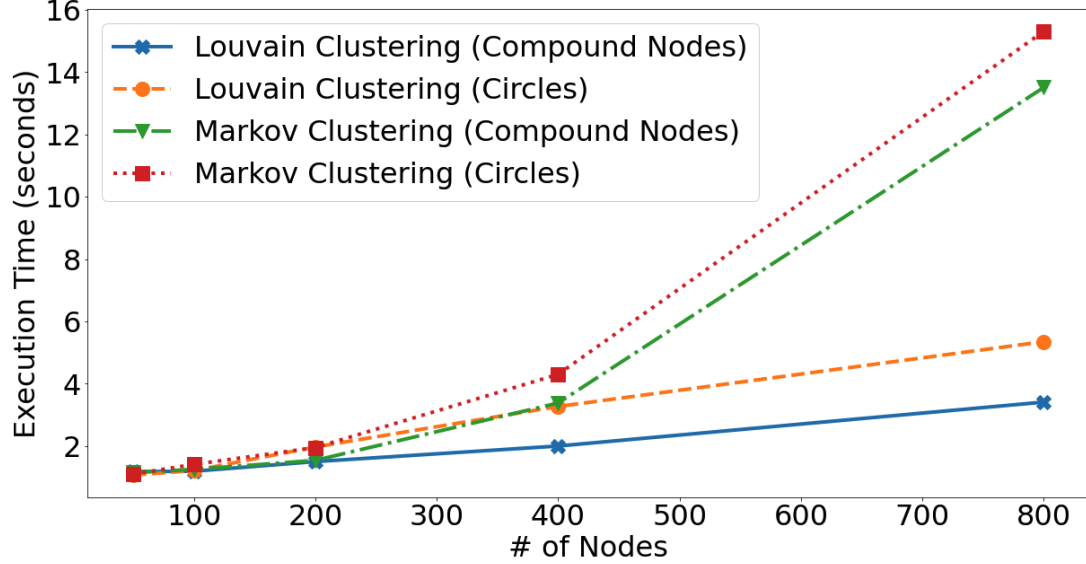


Figure 5.6: Execution times of clustering algorithms with circular and compound cluster representations

For Markov clustering algorithm [33], we used the available Cytoscape.js function. Its time complexity is  $O(|V|^3)$ . By utilizing sparse matrices, the original algorithm [33] can run faster. The implementation we used does not utilize sparse matrices. In the implementation, naive matrix multiplications cause cubic time complexity. We implemented Louvain clustering algorithm [32] ourselves. Its time complexity is  $O(|V| \cdot \log(|V|))$ . Figure 5.6 shows theoretical time complexity estimations are consistent with practical experiments.

## 5.2.4 Database Querying

Effective database querying is a critical part of visual analysis. For this reason, we tested some common use cases that involve database operations. We used a Neo4j database (version 3.5) consisting of 188 920 nodes and 406 316 edges. To prevent the effect of caching, we restarted the database for each experiment. Table 5.1 shows six use cases involving database operations. Three use cases are

fairly complex rules the other three are graph-based queries: neighborhood, GoI, and CTR.

Operation	Client-side Time (seconds)	Database-side Time (seconds)	Total Time (seconds)
Complex rule 1	1.58	0.85	2.43
Complex rule 2	1.08	1.34	2.42
Complex rule 3	1.68	2.54	4.42
Neighborhood	6.69	0.33	7.02
GoI	2.71	2.67	5.38
CTR	4.20	0.82	5.02

Table 5.1: Execution times of various database related operations in seconds

“Complex rule 1” is brings all people who acted at least once and have a name containing strings either ‘Jo’ or ‘Tom’. Figure 5.7 shows the executed rule.

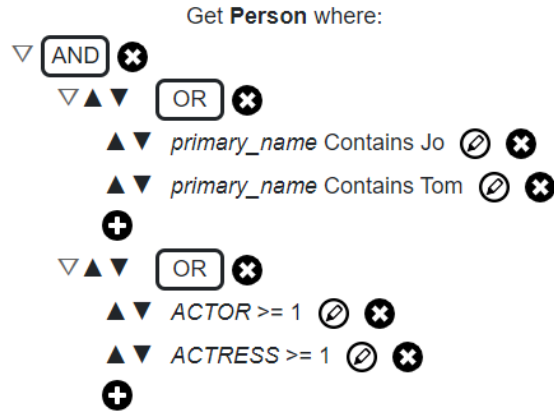


Figure 5.7: “Complex rule 1” inside Table 5.1

“Complex rule 2” brings all titles who have at least five actors or five actresses and genre is either comedy or action and type of title is either ‘movie’ or ‘TV movie’ and have more than 100 votes and rating greater than six. Figure 5.8 shows the executed rule.

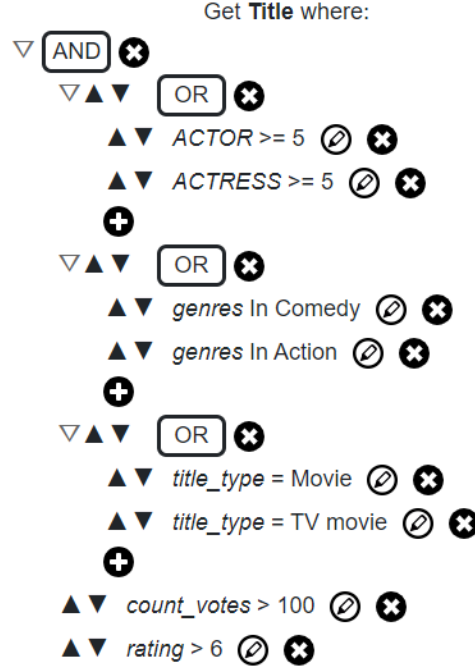


Figure 5.8: “Complex rule 2” inside Table 5.1

“Complex rule 3” is brings edges with type ‘Actor’ that have character ‘Tom’ or ‘James’ or ‘Alex’. Since an edge cannot be drawn without source and target nodes, edge type queries also bring source and targets of edges. Figure 5.9 shows the executed rule.

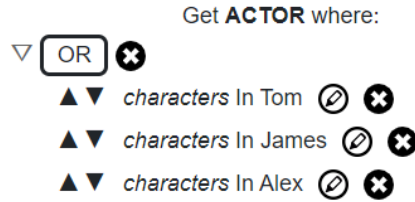


Figure 5.9: “Complex rule 3” inside Table 5.1

The last three rows in Table 5.1 are graph-based queries. We did not examine the performance of graph-based queries extensively as they were adapted from an original work [35]. Hence, we relied on analysis and experiments of that work. Experiments however confirm reasonable execution times on large databases as presented below.

The neighborhood query in this table brings degree-2 neighborhood of three people ('Federico Fellini', 'Richard Burton', and 'Bette Davis'). It brings 1 046 nodes and 216 edges. The GoI query finds a sub-graph involving two people ('Frank V. Phillips' and 'Bill Stewart') with a length limit of six. It brings 219 nodes and 343 edges. The CTR query finds a sub-graph with a length limit of three involving three people ('Georges Delerue', 'Richard Burton', and 'Bette Davis'). It brings 581 nodes and 537 edges. All the graph-based queries are undirected and apply a date filter which brings only the elements between the years 1970 and 2000.

In Table 5.1, total execution time is at most around seven seconds. To see the whole sub-graph at once, we set the page size to a large value in graph-based queries. So we bring more elements in graph-based queries compared to complex rules. For this reason, graph-based queries spend more time on the client side.

# Chapter 6

## Conclusion

In this study, we propose a software library that forms a basis for developing domain-specific graph visualization and analysis software. *Visuall* was designed to be modular and quickly customizable. It is equipped with various instruments for complexity management and visual analysis. For efficient database querying, we implemented some generic graph traversal algorithms and built appropriate UI components for constructing generic (SQL-like) queries.

For performance evaluation, we conducted experiments on common use cases and validated timely response for interactive use. On small to medium graphs, it not only works responsively but layout is done incrementally, respecting the mental map of the user. To ensure the robustness of the software, we also implemented many E2E tests.

### 6.1 Limitations & Future Work

*Visuall* is a software library but it is not packaged like a software library of a package-management system (e.g., NPM [43], Yarn [44]). Packaging *Visuall* might be useful to separate customizable parts from the base. Graph visualization requires deep customizations in many places. So providing customizations and

preparing *Visuall* to be used with a package-management system is challenging. On the other hand, packing *Visuall* might make it unnecessarily complex, hence over-engineering.

*Visuall* is a full-fledged software. It has its own UI components. People or organizations might want to integrate a graph canvas/visualization into existing software. In this case, *Visuall* can not be used directly. Packaging only the graph canvas part of *Visuall* can also still be useful, perhaps reducing the UI complexity.

*Visuall* uses Cytoscape.js to render graphs. Cytoscape.js works on the client-side without the need for any external computational power. Cytoscape.js however can not handle very large graphs though we think complexity management can be applied with such large graphs.

Our graph-based queries such as GoI and CTR are implemented on the database level. So if the user is using some other database provider, they should be implemented for the specific database provider. Since every database provider has its own query language and special mechanisms to inject algorithms, implementing queries can be tedious and compelling.

In the “Query By Rule” section, we build a hierarchy of conditionals. We might also let the user define graph patterns like in the Graph Studio of TigerGraph [45]. Visual patterns might be more explanatory way of querying a graph database. In this case, the user can build more graph-like queries instead of SQL-like queries. Users could be allowed to generate graph-based queries in this fashion.

fCoSE layout algorithm supports defining constraints. So we can let the user specify some layout constraints. In some sense, we can provide the user a way for customizing/defining their own layout algorithm. The user can define multiple constraints and choose which ones to apply on demand.

# Bibliography

- [1] U. Brandes, “Drawing on physical analogies,” in *Drawing graphs*, pp. 71–86, Springer, 2001.
- [2] H. Balci, M. C. Siper, N. Saleh, I. Safarli, L. Roy, M. Kilicarslan, R. Ozaydin, A. Mazein, C. Auffray, Ö. Babur, *et al.*, “Newt: a comprehensive web-based tool for viewing, constructing, and analyzing biological maps,” *Bioinformatics (Oxford, England)*, p. btaa850, 2020.
- [3] M. Sari, I. Bahceci, U. Dogrusoz, S. O. Sumer, B. A. Aksoy, Ö. Babur, and E. Demir, “SBGNViz: a tool for visualization and complexity management of sbgn process description maps,” *PloS one*, vol. 10, no. 6, p. e0128985, 2015.
- [4] M. Decuyper, “Visual network analysis: a qualitative method for researching sociomaterial practice,” *Qualitative research*, vol. 20, no. 1, pp. 73–90, 2020.
- [5] “Stackoverflow developer survey 2020.” <https://insights.stackoverflow.com/survey/2020#technology-databases>. (Accessed on 08/16/2021).
- [6] T. Kim, H. Chung, W. Choi, J. Choi, and J. Kim, “Cost-based join processing scheme in a hybrid rdbms and hive system,” in *2014 International Conference on Big Data and Smart Computing (BIGCOMP)*, pp. 160–164, IEEE, 2014.

- [7] J. Pokorný, “Graph databases: their power and limitations,” in *IFIP International Conference on Computer Information Systems and Industrial Management*, pp. 58–69, Springer, 2015.
- [8] I. Robinson, J. Webber, and E. Eifrem, *Graph databases: new opportunities for connected data.* ” O’Reilly Media, Inc.”, 2015.
- [9] H.-J. Schulz and C. Hurter, “Grooming the hairball-how to tidy up network visualizations?,” in *INFOVIS 2013, IEEE Information Visualization Conference*, 2013.
- [10] S. Noel and S. Jajodia, “Managing attack graph complexity through visual hierarchical aggregation,” in *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pp. 109–118, 2004.
- [11] H. C. Purchase, “Metrics for graph drawing aesthetics,” *Journal of Visual Languages & Computing*, vol. 13, no. 5, pp. 501–516, 2002.
- [12] U. Dogrusoz, E. Giral, A. Cetintas, A. Civril, and E. Demir, “A layout algorithm for undirected compound graphs,” *Information Sciences*, vol. 179, no. 7, pp. 980–994, 2009.
- [13] H. Balci and U. Dogrusoz, “fCoSE: a fast compound graph layout algorithm with constraint support,” *IEEE Transactions on Visualization and Computer Graphics*, 2021.
- [14] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, “Graphviz—open source graph drawing tools,” in *International Symposium on Graph Drawing*, pp. 483–484, Springer, 2001.
- [15] M. Bastian, S. Heymann, and M. Jacomy, “Gephi: an open source software for exploring and manipulating networks,” in *Third international AAAI conference on weblogs and social media*, 2009.
- [16] “Neo4j bloom.” <https://neo4j.com/product/bloom/>. (Accessed on 08/05/2021).

- [17] M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, and G. D. Bader, "Cytoscape.js: a graph theory library for visualisation and analysis," *Bioinformatics*, vol. 32, no. 2, pp. 309–311, 2016.
- [18] "D3.js." <https://d3js.org/>. (Accessed on 08/05/2021).
- [19] "Ravelin." <https://www.ravelin.com/>. (Accessed on 08/05/2021).
- [20] "IBM security i2 threat intelligence analysis platform." <https://www.ibm.com/security/intelligence-analysis/i2>. (Accessed on 08/05/2021).
- [21] "yFiles." <https://www.yworks.com/products/yfiles>. (Accessed on 08/05/2021).
- [22] "Tom Sawyer Perspectives." <https://www.tomsawyer.com/perspectives>. (Accessed on 08/05/2021).
- [23] "The KeyLines Toolkit." <https://cambridge-intelligence.com/keylines/>. (Accessed on 08/05/2021).
- [24] "Linkurious enterprise." <https://linkurio.us/>. (Accessed on 08/05/2021).
- [25] "CSS: Cascading Style Sheets." <https://developer.mozilla.org/en-US/docs/Web/CSS>. (Accessed on 06/09/2021).
- [26] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker, "Cytoscape: a software environment for integrated models of biomolecular interaction networks," *Genome research*, vol. 13, no. 11, pp. 2498–2504, 2003.
- [27] "Neo4j graph data platform." <https://neo4j.com/>. (Accessed on 08/05/2021).
- [28] "Angular." <https://angular.io/>. (Accessed on 08/16/2021).
- [29] U. Dogrusoz, A. Karacelik, I. Safarli, H. Balci, L. Dervishi, and M. C. Siper, "Efficient methods and readily customizable libraries for managing complexity of large networks," *Plos one*, vol. 13, no. 5, p. e0197238, 2018.

- [30] U. Dogrusoz, M. E. Belviranli, and A. Dilek, “CiSE: A circular spring embedder layout algorithm,” *IEEE transactions on visualization and computer graphics*, vol. 19, no. 6, pp. 953–966, 2012.
- [31] K. Freivalds, U. Dogrusoz, and P. Kikusts, “Disconnected graph layout and the polyomino packing approach,” in *International Symposium on Graph Drawing*, pp. 378–391, Springer, 2001.
- [32] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, “Fast unfolding of communities in large networks,” *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.
- [33] S. M. Van Dongen, *Graph clustering by flow simulation*. PhD thesis, Utrecht University, 2000.
- [34] M. Sarkar and M. H. Brown, “Graphical fisheye views,” *Communications of the ACM*, vol. 37, no. 12, pp. 73–83, 1994.
- [35] U. Dogrusoz, A. Cetintas, E. Demir, and O. Babur, “Algorithms for effective querying of compound graph-based pathway databases,” *BMC bioinformatics*, vol. 10, no. 1, pp. 1–16, 2009.
- [36] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta, “How changes affect software entropy: an empirical study,” *Empirical Software Engineering*, vol. 19, no. 1, pp. 1–38, 2014.
- [37] D. Spinellis, “State-of-the-art software testing,” *IEEE Software*, vol. 34, no. 5, pp. 4–6, 2017.
- [38] R. Paul, “End-to-end integration testing,” in *Proceedings Second Asia-Pacific Conference on Quality Software*, pp. 211–220, 2001.
- [39] W. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, “End-to-end integration testing design,” in *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, pp. 166–171, 2001.
- [40] “Cypress.” <https://www.cypress.io/>. (Accessed on 08/16/2021).

- [41] T. Opsahl, F. Agneessens, and J. Skvoretz, “Node centrality in weighted networks: Generalizing degree and shortest paths,” *Social networks*, vol. 32, no. 3, pp. 245–251, 2010.
- [42] U. Brandes, “On variants of shortest-path betweenness centrality and their generic computation,” *Social Networks*, vol. 30, no. 2, pp. 136–145, 2008.
- [43] “Npm.js.” <https://www.npmjs.com/>. (Accessed on 08/27/2021).
- [44] “Yarn.” <https://yarnpkg.com/>. (Accessed on 08/27/2021).
- [45] “TigerGraph.” <https://docs.tigergraph.com/ui/graphstudio/build-graph-patterns/visual-query-builder-overview>. (Accessed on 08/30/2021).