Master Thesis

MovieSearch

Building semantic search queries with suggestions

Tobias Sommer

November 16, 2016

First Reviewer:	Prof. Dr. Hannah Bast
Second Reviewer:	Prof. Dr. Peter Fischer
Supervisors:	Prof. Dr. Hannah Bast, Björn Buchhold

Albert-Ludwigs-University Freiburg
Faculty of Engineering

Chair of Algorithms and Data Structures
Prof. Dr. Hannah Bast

Declaration I hereby declare that I am the sole author and composer of this thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare that this thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg, November 16, 2016

Tobias Sommer

Contents

Abstract

MovieSearch helps with building semantic queries for tasks like *"find movies with Arnold Schwarzenegger where he fights with a sword"*. The suggestions by MovieSearch aim to enable the usage of semantic queries. Usability of semantic queries is the main contribution of MovieSearch. The tasks of our user study could be successfully answered in 94% of the cases.

MovieSearch provides a user-friendly and effective semantic search system for non-experts. Semantic search shall become available for users who have no knowledge about database languages like SPARQL. The interactive query building delivers traceable results in contrast to natural-language-based approaches with their hidden query building.

We build queries by stepwise adding conditions to specify what we are looking for. The conditions consist of facts and plot. MovieSearch implements an approach with those two input categories. Our example task looks for a movie with actor *Arnold Schwarzenegger* as a fact, and the plot description *fights with a sword*. MovieSearch enriches the fact options by 3-ary relations. This allows to connect certain fact parts, for example to look for a movie where the actor *Elijah Wood* played the character *Frodo*.

The results for the queries will be answered by the SPARQL Server provided by [**?**].

Zusammenfassung: Abstract in German

MovieSearch (FilmSuche) hilft dabei semantische Anfragen zu erstellen für Aufgabestellungen wie *"finde Filme mit Arnold Schwarzenegger, in denen er mit einem Schwert kämpft"*. Die Vorschläge, die MovieSearch macht, zielen darauf ab semantische Anfragen zugänglich zu machen. Die Benutzbarkeit von semantischen Anfragen ist der wesentliche Beitrag von MovieSearch. Die Aufgaben unseres Feldversuchs konnten in 94% der Fälle erfolgreich beantwortet werden.

MovieSearch bietet ein nutzerfreundliches und effektives System zur semantischen Suche auch für Laien. Semantische Suche soll nutzbar werden, ohne Kenntnisse von Datenbank-Anfragesprachen wie SPARQL vorauszusetzen. Das interaktive Zusammenstellen von Anfragen führt zu nachvollziehbaren Ergebnissen im Gegensatz zu Ansätzen, die natürliche Sprache benutzen und die Anfragen ohne Interaktion von selbst bilden.

Wir erstellen Anfragen durch schrittweises Hinzufügen von Bedingungen, die bestimmen wonach wir suchen. Die Bedingungen setzen sich zusammen aus Fakten und Handlungstexten. MovieSearch besitzt diese beiden Eingabekategorien. In unserer Beispielaufgabenstellung suchen wir nach einem Film mit Schauspieler *Arnold Schwarzenegger* als Fakt und der Handlungsbeschreibung *kämpft mit einem Schwert*. MovieSearch bereichert die Faktenoptionen mit 3-er Relationen. Damit können wir Teile von Fakten verbinden. Ein Beispiel ist, wenn wir nach einem Film suchen, in dem der Schauspieler *Elijah Wood* die Rolle des *Frodo* spielt.

Die Ergebnisse zu den Anfragen werden von dem SPARQL Server beantwortet, der zur Verfügung gestellt wurde von [**?**].

Why didn't we do this sooner?

WHAT EVERYONE SAYS AT SOME POINT DURING THE FIRST USABILITY TEST OF THEIR WEBSITE

– STEVE KRUG [**?** , page 111]

Chapter 1

Introduction

1.1 Goals

The goal of this thesis is to build a user interface for semantic search in the domain of movies that allows inexperienced users to explore the underlying data.
The narrow domain is used in particular to support 3-ary relations.
The whole system will be called MovieSearch and aims for high usability.

The semantic search engine Broccoli [**?**] is the predecessor of the whole work. To retrieve answers for your questions you have to know how to ask them. The User Interface should help users to formulate their question to the search engine. Formulating the question is in this context building a corresponding query. There are different approaches on how to do this. We will take a look at some of them in section 2.4. Unfortunately, the user interface of Broccoli is difficult to use for inexperienced users on their own [**?** , Section 3, Target Users]. Hence, there is improvement possible in helping users to formulate their questions. We narrow down the domain to create a system to search for common things about movies as well as plots in a more intuitive manner. The domain makes the different fact possibilities manageable for a preview on how the fact suggestions perform. Thus, enabling semantic search to a broader audience.

Consider the task *"find movies with Arnold Schwarzenegger where he fights with a sword"*. The task is a good candidate for a semantic search since it contains a fact and some describing text. But, how can we get a semantic query from the task? The goal of MovieSearch is to help users to answer this question and build a query. Existing approaches are often hard to use or at least have room for improvement concerning usability. MovieSearch helped the participants of our user study to answer their tasks correctly in 94% of the cases.

The underlying data consists of RDF facts about movies and text pieces like plots, quotes and similar. We will explain RDF in section 3.1. For now it is enough to know that this is a way of storing facts. An example fact is: the director of *Braveheart* is *Mel Gibson*. The idea of semantic search is connecting facts with text to improve the quality and relevance of results. We will call knowledge stored in the RDF data a fact and reference a text piece as a document, regardless if the text piece is a plot or a quote or of some

other kind.

What is done to achieve the goals Before we can answer any queries we need data. First we retrieve data sources and reformat them for our purposes. This is the topic of chapter 3. Second we want to handle the process from a given task to building a query and afterwards finding results for the query. The part we focus on is building a query from a task.

Consider our example task *"find movies with Arnold Schwarzenegger where he fights with a sword"* from a user perspective. How can we build a query from it to find the results we are interested in? The first observation is that the movie has something to do with Arnold Schwarzenegger. Especially, we mean to search for movies in which Arnold Schwarzenegger played a role. How can we represent that fact in the query? For this we need to figure out how such facts are stored in the underlying data. This is not trivial since we have a priori no idea how facts are described in the data. However, we input *"starring Arnold Schwarzen"* into the fact description input field and see what happens. The suggestions that are displayed for that input are grouped by some relation names with different values. The very first suggestion is ***in movie as actor*** *Schwarzenegger, Arnold*. This seems to to be the fact from our task, therefore we use it as a condition for the current query. Besides, we know what happens in the plot of the movie. Therefore, we input *"fights with a sword"* into the plot description as another condition.

Conditions specify what we are looking for. The conditions consist of the two categories facts and plot. Plot is inputted as a text and facts will be a relation with a value. The data is explorable with the suggestions for the given fact description. The suggestions offer guidance in specifying facts in an effective manner. Entities or relations from the data will be found in the fact description and respectively suggested. Furthermore, we add all relations to found entities that are relevant in the current scenario. This makes the possibilities of stating facts in the data explorable. The query is built by stepwise adding facts that determine the thing that the user looks for. The search can be set to find either a movie or a person. A person can be a real life human like *Ridley Scott* or a character like *Maximus*. A fact is roughly added with the following steps.

1. The user inputs text that describes a fact.
2. Suggestions of relations with entities that match the fact description are presented. The user selects a suggestion as a candidate fact.
3. The user checks how the candidate fact affects the results and decides if the candidate fact should be used for the current query.

Let us consider the parts in more detail.

Fact Suggestion For a given text that describes the fact, the Fact Suggestion Server (see chapter 4) will find matching facts. Remember the *"Arnold"* example from above

and the steps with which we found a fact. The facts in our data consist of entities together with relations. Our data about the movie facts is stored in a RDF graph. The RDF graph will be presented in chapter 3 which also specifies relations and entities in detail. The facts consist of a pair of a relation and an entity. These pairs are presented as suggestions.

If there is no word in the given text that matches any relation, the text will be used to match entities. Afterwards, all ingoing relations of the matched entities will be found, grouped and included to this fact suggestions. An example for this behaviour can be illustrated with the fact description just *"Arnold Schwarzen"* without the *"starring"*. Here, we find the entity *Schwarzenegger, Arnold* with a couple of matching relations. The ***in movie as actor*** is one amongst others like ***directed by***. Hence, even if the user has no knowledge about the relation names he can find all relations that are relevant for his input text.

All relations have types to define what kinds of objects are connected through them, stored in a file. The types are discussed in detail in section 3.5.1. For example there is a relation *directed-by* that always connects on the left side a movie with a person on the right side, for example *Gladiator directed-by Scott, Ridley*. The fact suggestion process finds only relations that connect to the given searchtype. The searchtype is the type of the object we are looking for. In our *"Arnold"* example the searchtype is *movie* since we want to find a movie. The suggestions will be clarified in detail in chapter 4.

Querying All the current facts will be transformed to a query. Our queries are SPARQL queries. Clarifying what exactly constitutes a SPARQL query is postponed to section 3.1. If there is text inputted that should occur in a corresponding document, it is added to the query. The results will show found movies (or persons) that are satisfying the conditions. The conditions are satisfied if all facts apply to the movie, and the stated plot description is contained in a document that is connected to the movie, too. The backend that answers the queries is introduced in section 2.2.

Candidate fact The candidate functionality of the web client allows the user to explore the data. The result of the current query can be seen with the inclusion of the current candidate fact as well as without the current candidate. This way the user can see how the results change. Regarding the changes helps with deciding if the built fact condition represents the fact from the task description. Therefore, the user can consider the addition of the candidate fact to the current query or if he wants to check other suggestions.

3-ary relations Most facts are binary relations, like *Gladiator is directed by Ridley Scott* or *Gladiator has a runtime of 155 min*. We have found no tool that allowed to ask for a movie with *actor Russell Crowe as character Maximus* directly linked together. Here, we want to use the 3-ary relation of a cast for a movie like *Russell Crowe acted in movie Gladiator as character Maximus*. Our user interface and the fact suggestions support these kind of facts which is meant to be a starting point for allowing to query for n-ary relations in future. Note, that the above example is different from the use of the two

binary subrelations. Searching a movie with *actor Russell Crowe* and *character Maximus* is different. Because with binary subrelations, the connection between the actor and the character is lost. For more about this see section 2.6.

1.2 Contributions

An overview about the contributions of this work is summarized in the following list. The list is sorted by relevance.

- User Interface for semantic search about movies
- Fact suggestions for easier query building
- Support for 3-ary relations
- Data integration of movie facts and movie documents (plots, quotes, trivia, ...)
- First version of modified PageRank scores

1.3 Structure of this thesis

After considering related work, we regard the underlying data and its preparation. Thereby, we introduce the data structure that is used by the Fact Suggestion Server which is the next topic. Afterwards, we see an overview of the User Interface. Summarizing theses parts together to the architecture leads to the evaluation. Here, we discuss the usability of the system. The conclusions finish the work. Details about the evaluation experiments or the used data files can be found in the appendix.

1.3.1 Notation conventions

Important abbreviations and notations that will occur are clarified in the following section or are defined at their first occurrence.

- Names are stored URL encoded in our data. In examples of this work we will nevertheless use plain names for a better readability.
- To simplify reading, only the masculine form is used in this thesis. All references to the male gender shall be deemed and construed to include the female gender.

1.4 Structure of the implementation

There are three parts working together after the initialisation. The web client with the user interface will build a query in an interactive manner with a user. For each new fact the Fact Suggestion Server will provide the client with suggestions and other

information. Whenever the results for the current query are requested, the SPARQL Backend [**?** , SPARQL Backend by Buchhold] will provide the results. The client will send the query after changes. The user can send the query at any time on purpose. For an overview of the parts of MovieSearch and their interactions see figure 6.1 in chapter 6. A list of all used modules, secondary software, et cetera is located in the appendix in section C.

Let us begin with regarding related work in the next chapter.

Chapter 2

Related Work

For an overview about different user interface choices for semantic search engines we take a look at other work. Furthermore, we glance at fields of interest in related research areas to locate this works contribution.

2.1 Predecessor: Broccoli

The semantic search engine Broccoli [**?**] connects two sources of knowledge that are accessible for the computer world. On the one hand there is natural language text that is widespread in the web and on the other hand there is structured data that describes facts for example any RDF Store. Combining these two sources is the main idea of semantic search in general to expand the mere text search by facts about entities that live in the text. Building queries that enable this combination is part of the system Broccoli. However, its target users are expert searchers and researchers that are familiar with the backgrounds of semantic search as specified in [**?**]. Whereas here, we build a search interface for non-experts. As non-experts we comprehend people who have few or no knowledge about the domain (e.g. relation names) or have few or no knowledge about backgrounds like RDF or SPARQL.

2.2 Semantic SPARQL Backend

The built SPARQL Queries are answered by the SparqlEngineDraft Backend presented in [**?**]. Querying RDF facts in combination with semantic search needs a special SPARQL Server. RDF triples have to be connected with text. Text is included via context sensitive integration of words and entities.

2.3 Storing facts in a RDF graph

For the fact suggestion we store the facts in a graph. The main goal with this is enabling fast suggestions in the later applications. We compare our reasoning for this choice with similar other work.

2.3.1 Social network

A system that integrates social network data with an own query language is described in [**?**]. The most interesting part is that Martín et al. store the RDF data likewise in a graph structure. For social networks this is quite obvious even if it is not the only way to store a relational model per se. Their transformation language is a dialect of SPARQL tailored to their data model. Since we want to abstract from SPARQL queries the part about the transformation language is not directly relevant for our purpose. But to compare how their fact data is stored surely is.

The social network data model from [**?** , Ch. 2 Representing Social Networks over RDF] is a directed graph with labels. Relations are stored as nodes. This has two reasons: First relations have properties which therefore can be connected with arcs to be included in the graph. Second this should integrate *n*-ary relations seamlessly.
MovieSearch does not have such relation properties. The 3-ary relations are stored via a link node and supplemented with shortcuts to model its contained binary relations. Section 3.5 will explain this in more detail. Thus, we do not have the same motivation for storing relations as nodes. Therefore, MovieSearch stores relations as directed arcs. This results in a graph that has a more similar structure to fact triples. Where a triple consists of two nodes that are connected with an arc. Thus, nodes are entities that are connected by arcs as relations. This should lead to a lightweight graph that answers fact suggestions quickly what we will evaluate in section 7.1.

2.3.2 Temporary RDF graph

If in the future MovieSearch is adapted to another domain in which facts only hold temporary, the following work is of interest [**?**]. The domain movie in our instance has only facts that are independent of time. For example the fact that defines the director of a movie holds without time limitation. In contrast a possible fact like *Obama is president* only holds in some years.

Gao et al. store their facts likewise in a graph while aiming for a user-friendly interface. But they focus on the temporal scope of facts. In-depth information about the background of their system can be found in [**?**].

2.4 User interface for semantic search engines

There are many approaches to designing semantic search access. They range from inputting pure SPARQL into an input field over graph-based definitions of queries to natural-language-based query building. The approaches differ in how much control a user has of the exact question that he asks. A pure SPARQL query gives total control of the query, but has the drawback of the necessary knowledge of SPARQL and relation names. Graph-based queries are also called tree-based because of the structure of the query in a way that reminds of a small graph or a tree where the root is the object that

is looked for and the paths describe facts. Broccoli is such a graph-based query building tool. They vary in how much help is given to the user in building the graph. But, they give in general a notable amount of control about how the query is built. Natural-language-based approaches are on the contrary more like black boxes. You provide a piece of text that describes your question and somehow it is transformed into a query. This process allows rather less control and has especially problems with more complex queries.

Since MovieSearch is graph-based, we first look at other graph-based user interfaces. Afterwards, other possibilities are presented.

2.4.1 Graph-based query representation

An overview of **intuitive** semantic search engine user interfaces can be found in [**?**]. Styperek et al. observe that despite the different approaches the practical value of the interfaces are quite limited in realizing intuitive use. They conclude that the most satisfying results are given by graph-based approaches. MovieSearch has a graph-based approach.

In their work they allow to define types to the variables. Which is done by internally adding a *is-a* relation to correspondent variable. In MovieSearch the type is handled differently. The user only has to select if he wants to search for a person or a movie. The rest of the types are only used implicitly by the Fact Suggestion Server. For each relation there is a definition that tells from which type this relation maps to which other type. For example *directed-by* connects left-hand *Movie* with right-hand *Person*. This is useful for suggesting only relations that are connectable and therefore currently relevant. As another benefit there is no need for a *is-a* relation triple per variable. Suggestions have to match to requested types therefore we ensure the types of the variables.

2.4.2 Natural-language-based

The natural-language-based movie search system [**?**] is based on conditional random fields. It presents movies that are described by one textish or spoken input. Text parts are being labeled to parse the request. Query details are hidden from the user. The whole usage of the tool should be like a conversation. Their example question is "*show me a funny movie about bridesmaids starring Kera Nightley*" [**?** , Introduction]. Note that this misspelling of *Keira Knightley* is recognised. In this example the labeled input parts are: *GENRE: funny*, *PLOT: bridesmaids* and *ACTOR: Keira Knightley*. The label sequence is chosen by conditional probability maximization. The system must be initialised with training data to predict these label sequences in a preprocessing step. The training data must be tagged by humans. Mapping language terms to relations and objects of the data is difficult because of the indeterminism and ambiguity of natural language.

MovieSearch is additionally able to search for persons and allows more control over the queries. In a sense MovieSearch lives between such natural-language-based and existing

graph-based approaches because it aims to use the best from both worlds. Especially the understanding of given text snippets as input is hard to understand for natural-language-based approaches. To decide when a piece of input text is meant to be a snippet, that defines a search property, is difficult. The aforementioned tags should not be applied to parts of the input that are inside such a text snippet. This is the main reason for us to use a separate input field for text snippets, hence there the system knows what to expect.

The text input per fact is reduced by our suggestions. One global input for a natural-language-based approach, is now one input per fact. However, these inputs are not as complex as in some graph-based interfaces, where you have to input a full relation triple. The displayed current query on the other hand utilizes the visualizing feature of graph-based approaches.

2.4.3 SPARQL input

An user interface for pure SPARQL queries into a text field is created in [**?**] by the name of YASGUI. The goal of this work is to advance interfaces for SPARQL to the level of state-of-the-art IDEs. This is done by autocompletion, syntax highlighting and syntax checking. It is independent of a single endpoint to answer its queries.

MovieSearch is single endpoint reliant. However, using YASGUI naturally demands knowledge about SPARQL which we want to avoid. But, its concept could be of interest in advancing the SPARQL answering backend. Besides, features that improve usability like autocompletion are of interest for MovieSearch.

The next two sections illustrate two alternative approaches for semantic search interfaces. These interfaces are made for a special cause and did not quite match the terms for reaching our goals.

2.4.4 Editable temporal fields

We take a look at the user interface to the work from [**?**]. This is the temporal fact graph from section 2.3.2. The UI allows to filter for time-spans in which facts are valid. This is done by using the Wikipedia Infoboxes and editable fields at every part of the Wikipedia page that is modifiable. Time constraints can be inputted into these fields which will filter the displayed results.

The various different places where users can input data is probably only suited for expert users. Though, a usability test for its interface is owing.

2.4.5 Faceted Search

One example for *"a lightweight browser-based faceted search engine tool"* (see [**?**]) is regarded at this point. Data is divided into hierarchic categories. Users choose categories and add constraints to them to narrow the results down.

The SPARQL Faceter [**?**] user interface has many input fields. The time needed for displaying results varies between a few seconds up to more than ten seconds. These two points are drawbacks MovieSearch aims to avoid. Too many input fields make it harder to orientate yourself on the website. Too much delay for the reactions of the client are reducing the interactivity. Similar to the aforementioned categories are the types of the entities in the fact RDF graph. The types are as well hierarchic and group the facts. This illustrates an instance of how too many input fields cloud user-friendliness.

2.5 Evaluation of user interfaces

The evaluation of semantic search user interfaces is considered in [**?**]. The main goal of Styperek et. al is to decouple the evaluation process from subjective test-user opinions to create a repeatable objective methodology for evaluating such interfaces. The big challenge of these user interfaces is to make them intuitive and allow to unfold the expressiveness of semantic queries.

They describe a repeatable methodology to measure interfaces not only based on users' opinions. The measure is counting how many text inputs have to be performed to answer a query. The assumption is the fewer things you have to insert, the simpler the interface is to use.

This measure is meant as an objective enhancement to the subjective evaluation through usability tests with users that base on their opinion. Styperek et. al focus on graph-based systems which is suitable for our system.

For the evaluation of MovieSearch we will regard quality of our results as well as the usability. The usability will be tested by performing test with participants and see how they build queries. Therefore, we can analyze if they built the expected queries for a task and compare how many text inputs they needed to do so.

2.6 *n*-ary relations

Aspects of modelling *n*-ary relation are discussed in [**?**]. Although, binary relations are the most common one, there is sometimes a need for connecting more objects. For example we want to connect actors with movies and with their character in that movie in a 3-ary relation. Note, that you can not simply split the 3-ary relation into 3 binary relations. To proof this, it suffices to build an example of 3-ary relations and split it. Than we show that the split binary relations express another 3-ary relation that was not part of the start. For detailed proof see B.1 which is similar to the one in [**?**]. In a visual way it is comprehensible that information is lost by breaking up the 3-ary relation. MovieSearch models 3-ary relations by introducing an artificial connecting entity that interlinks the 3 objects (see 3.5.2.3).

2.7 Ranking in RDF graphs

Ranking is an important part for finding relevant answers for queries. If for example the fact suggestion of MovieSearch found many possible entity hits for a given text we have to decide which are the most relevant hits and order the fact suggestion accordingly. For ranking nodes in a RDF graph it is natural to consider shortest paths like in [**?**]. For each node there is a score computed that than will be used for ranking. The score of a node is defined by how costly it is to reach all other nodes where the number of edges and the sum of the weight of the edges influence the resulting score. Thus, central nodes are identified because they connect most likely with shorter paths to all other nodes than a node that is not central. Like transit nodes in route processing these central nodes will be nodes that are part of many shortest paths.

In our domain we would want to enforce centrality of the main type of nodes which are movies and actors. However, computing all shortest paths is costly which is why we use a flow-based approach instead of a shortest-path-based ranking. How beneficial our scores are is part of the evaluation.

The main ranking part of the entities in the RDF graph for MovieSearch is done by a slightly modified version of PageRank [**?** , pages 107-113]. This is an iterative procedure. Each node starts with the same score. In one iteration each node distributes its current score to its neighbors. The iteration stops if the score differences from before and after the iteration drop under a defined threshold.

2.8 User recommendations

Another work with a similar goal than MovieSearch is [**?**]. They suggest combining informational retrieval with a recommender system. Their focus is using browser history per user to influence the personalized recommendations for them. Park et. al consider mainly users' ratings on movies. Whereas, MovieSearch mainly uses the graph structure of the underlying facts. However, both approaches aim to improve user experience in the same domain. They identify the same problems, we try to solve. If a user does not exactly know what he is looking for the search system he uses should aid him.

They identify querying for character names to find a movie as poorly supported, too. For example querying for *Frodo* or *Gandalf* should lead to *The Lord of the Rings*.

The following chapter will introduce the underlying data of MovieSearch and how we utilize it.

Chapter 3

Data Preparation

In this chapter we will regard the source data and how we process it for further use. After this, we present the data structure that is used by the Suggestion Servers. However, the first section will discuss some basic concepts regarding the data and the querying.

3.1 Fundamentals – Knowledge base

For our system we want to use facts. The facts have to be stored for our purposes in a suitable way. We use a knowledge base with triples to represent the real world facts. Knowledge bases aim to answer specific questions. Furthermore, they have an object-oriented touch. Those two things reason our choice for this data structure.

To be more specific, for us of interest are:

RDF **R**esource **D**escription **F**ramework: data model, based on relation triples to model entity relationships (see `https://www.w3.org/RDF`)

SPARQL **SP**ARQL **A**nd **R**DF **Q**uery **L**anguage
(see `https://www.w3.org/2001/sw/wiki/SPARQL`).

RDF is the way we store the facts. SPARQL retrieves answers from the RDF triples. We only regard briefly their components that are relevant for us. The RDF and SPARQL are parts of the Semantic Web. The Semantic Web is a framework that has much more to offer than we will discuss for our purposes.

The next sections clarify: what are entities, what are those triples and how does SPARQL work.

3.1.1 Entities

We are most interested in the entity-relationship aspect of RDF. With this relationship we will model the facts. For this we first have to clarify what an entity is. An entity is a thing that has an independent existence and can be distinguished to other things. An entity may appear at possibly various places in the data while referencing to the same thing. For example the person *Brad Pitt* or the movie *Inception (2010)* are entities. The

person *Brad Pitt* is part of many facts. For example a movie where he played a role or another movie where he was director. All those occurrences relate to the very same person.

We denote the set of all entities in our knowledge base E.

3.1.2 RDF triples

We store the facts with triples. An example for a triple in our data is *(Inception (2010), directed by, Christopher Nolan)*. With the relation *directed by* we describe the relationship of the two entities in the triple.

A triple is of the form (e,R,v) with an entity $e \in E$ and an value $v \in E$. Hence, it holds $R \subset E \times E$. Because of that we call the middle part of the triple a relation. The R specifies which entities are part of this relation by all its fact triples. The triple (e,R,v) is analog to the *(subject, predicate, object)* form of statements like *(Inception (2010), has genre, Action)*. In other words, we have an *(entity, attribute, value)* triple in an object-oriented sense. From the view of the RDF triples the values are entities, too. Look at the example *Action* which is one kind of a genre. It appears in various places in the facts at different movies. It describes always the same thing and *Action* is distinguishable from i.e. *Drama*.

The origin of RDF lives in Semantic Web. RDF inherits the URIs from the web background. An URI allows to access a part of a triple. For example a value of a rating could be `"4.2"^^<http://www.w3.org/2001/XMLSchema#float>`. The value is the part before the `^^` token. The URI contains information about the data type, here we have the *float 4.2*.

Graph of the triples The fact triples with the entity relationship form a graph in the following way. The entities have the same properties like nodes. They are accessible from different other places. Hence, we use all entities as nodes in the graph. We interpret the triple (a,R,b) as an graph arc $a \xrightarrow{R} b$. Depending if the actual relation R describes a fact that holds both ways, we additionally can add the arc $a \xleftarrow{R} b$.

3.1.3 SPARQL – Queries for RDF triples

SPARQL is a query language for RDF stores. SPARQL is similar to SQL which is used to query relational databases.

```
<Inception (2010)> <directed-by> <Christopher Nolan> .
<Inception (2010)> <has-genre> <Action> .
<Interstellar (2014)> <directed-by> <Christopher Nolan> .
```

Listing 3.1: Small example RDF triple store.

Given a RDF store containing the three triples from listing 3.1, we regard a small SPARQL example that accesses this RDF store.

```
SELECT ?x WHERE {
  ?x <directed-by> <Christopher Nolan> .
}
```

Listing 3.2: Example SPARQL 1.

The small example SPARQL query in listing 3.2 will retrieve the movies *Inception (2010)* and *Interstellar (2014)*. The conditions in the SPARQL query are mapped to triples of the same form in the underlying RDF store from listing 3.1. Whereas, for the variables like $?x$ there are all entities or values allowed. The only entities e that have a triple of the form of the given conditions are the two movies which we found.

What is returned, is defined by the variables that follow the `SELECT` clause. Other keywords help with sorting the results or making the results unique, et cetera.

More relation conditions can be specified like in the SPARQL query from listing 3.3. In this case, we only retrieve entities that have an appropriate RDF triple for all the conditions.

```
SELECT DISTINCT ?x WHERE {
  ?x <directed-by> <Christopher Nolan> .
  ?x <has-genre> <Action> .
}
```

Listing 3.3: Example SPARQL 2.

Hence, the result for the query in listing 3.3 is only *Inception (2010)*. The `DISTINCT` removes duplicates from the results.

Now, we are prepared for continuing with the next section about the source data.

3.2 Source data

We use two sources for the information about movies. First there are plain data text files from IMDB [**?**]. This is the subset of the complete IMDB data that is available for download. For a list of all files we included in the data structure see B.2.1 on page 108. The IMDB files contain facts and text. The kinds of text we consider are plots, taglines and various trivia. We generate RDF triples and documents from the files accordingly to their content. One weakness of the IMDB subset is that some famous actors are not contained. For example *Brad Pitt* occurs only as a producer, etc. but never as an actor in any of his movies. This has to be fixed for building a useful application on that data.

The second source is the API of The Movie Database [**?**]. With the TMDB data we address the problem of the missing topactors. The Movie Database is used to add cast information of popular actors and actresses that are not contained in the subset of the IMDB data. Their popularity is measured by a popularity score that is accessible via the

TMDB API. For the most popular actors we access all their movies and which character they played in it. We will later on discuss how to integrate this cast data of the topactors into the IMDB data. The movie posters and the profile pictures for persons are loaded once from The Movie Database and are than stored on the server. Such that if they are requested again we load them from our server.

The next section regards in what way we want to store the given source data.

3.3 Target data format

How to store the source data in a way that is suitable for our purposes? This section describes the format of the data we want to have for running MovieSearch on it. There are two types of data the running system has access to. We have fact data and text data.

The fact data is needed for the Fact Suggestion Server which we will regard in chapter 4 and for the SPARQL Server. The facts will be stored in a RDF triple store like we discussed in section 3.1.2.

The words are stored URL encoded for better communication between client and servers. For once we show here the actual RDF format used in the files. Afterwards, we show the easier readable format we use in this work instead.

```
<Gladiator%20%282000%29> <has-genre> <Action> .
<Gladiator%20%282000%29> <directed-by> <Scott%2C%20Ridley> .
<Gladiator%20%282000%29> <from-year> "2000-00-00"[...] .
```

Listing 3.4: RDF triples – Implementation format.

For better readability we use the following format for the above RDF triples. Table 3.1 shows the version of listing 3.4, which we will use from here on to display fact triples.

Gladiator (2000)	*has-genre*	Action
Gladiator (2000)	*directed-by*	Scott, Ridley
Gladiator (2000)	*from-year*	2000

Table 3.1: RDF triples – Format used in this work.

In general each part of a triple is one of the following: an entity, a literal a relation or a number-based value. A literal is a piece of text without an object instance. A number-based value can be an integer, a float or a date. The number-based values will be used with a comparison later on in this chapter from section 3.5.2.2 on. For all the RDF triples we are going to build, it holds that they have the form (e,R,v) with $e \in \{\text{entities}\}$, $R \in \{\text{relations}\}$ and $v \in \{\text{entities, literals, number-based}\}$.

The text data is stored on the one hand as the full text pieces and on the other hand as lists of words from each text data piece. The full text pieces are stored in a docsfile and the list of words per text piece are stored in a wordsfile. The docsfile and wordsfile

will be discussed in section 3.6.2. These two files are needed for the SPARQL Server to process answers that request plot.

Now let us take a detailed look at the preparation.

3.4 Preparation of the data

The full preparation process consists of:

1. Build RDF triple from the IMDB sources.
2. Store text that should be available as a document with its associated movie title.
3. Integrate TMDB casts into the existing RDF triples from IMDB.
4. Compute scores for the entities in the RDF graph. Afterwards, append the scores in RDF triple form to the RDF data. That way we can use these scores to sort the results from the SPARQL Server.
5. Perform entity recognition in the documents. The entities to recognise are from the RDF triples.

In step 1, we use Python scripts to reformat the various file formats to unified RDF triples. The appearing entities are stored URL encoded. This contributes to a smoother communication of the server and client about the RDF data which we will see in later chapters. Thereby, problems with special characters should be prevented. In examples of this work we will nevertheless use plain names for entities for a better readability.

We use relations for RDF triples that are supported by our data base and are likely interesting for users. Note, that the selection and naming of the relations is somewhat indiscriminately. Concerning the selection of the relations, we deliberately added many of the relations from the data. This increases the chance of having a relation ready for a desired fact description. Concerning the naming of the relations, we will see the role of alternative names for the relations in section 4.1.1.

The RDF triples define facts that are either binary or a 3-ary relation. An example for a binary relation is *directed-by* which is generated from the file *directors.list*. Whereas the file *actors.list* produces cast triples of an actor with its character in a movie. For example from the line *Crowe, Russell in Gladiator as Maximus* we generate the triples:

castconnector	*actor*	Crowe, Russell
castconnector	*in-movie*	Gladiator
castconnector	*character*	Maximus.

Besides, numbers will be formatted for the SPARQL Backend and currencies are exchanged to US-Dollar. The RDF triples are used to build the graph for movie facts which we will cover in section 3.5.

To achieve step 3, we use the Match Suggestion Server we will consider in detail in chapter 4. For example, for a given actor with movie the Match Suggestion finds all those occurrences in the given RDF graph which in this case is just the IMDB-based graph. This way we can directly merge actors from the IMDB data with topactors from TMDB if they have similar names and occur in the same contexts. By merging we add missing character names for cast information. Not found topactors will be added instead. More about this will be regarded in 3.5.4.

For step 5, we consider the movie context of each movie. The movie context consists of all neighbors of that movie that are a person or a movie or describe a text property (e.g. genre Action). So for each document and its movie, we check if we identify any entity from the context of that movie in the document. For details see 3.6.

As a result of the preparation we have the full RDF file with a documentfile and a wordsfile. The documentfile stores each full text with an ID and the wordsfile contains all words and entities that occurred in the corresponding document. This is the data base that is used for the Suggestion Servers and for the SPARQL Engine.

In the following sections we consider the graph-building steps in more detail.

3.5 RDF graph for storing Movie facts

This section elucidates the graph we build from the prepared RDF triples. Our overall goal is simple query building for non-experts. That includes helping them to find the relations they want to use to describe a fact they want to state. This is achieved by having a type that is searched like *movie* together with the input for a fact that represents a value. The types of entities and the definition of what types are connected by each relation help to get a suggestion. Suggestions are regarded in detail in chapter 4. The suggestion consists of all relations that match the current searchtype and have an entity which matches the inputted fact description. Thus, the user will be able to select from possible and relevant relations for his scenario rather than to have to state the relation he probably does not know the exact name for.

The review of the data resulted into the relations we want to build. Those are the relations that are used for exploring the movie data via the web client. At the same time we defined which information should be stored as a connected 3-ary relation. Besides, we defined the types that determine what kind of entities are connected by the relations.

3.5.1 Types of entities

To begin with the more detailed look at the graph we talk about the difference of nodes and entities. A node is a part of the graph that is connected with arcs to other nodes. An entity is an object in the data that is referenced by potentially many triples that describe the same thing with different facts. For example the actor *Crowe, Russell* is an entity

in the data who appears in many fact triples that define in which movies he acted and more. In the graph there will be a node with the same name, which models this entity. We want to distinguish this from for example dates or text descriptions of a property. The genre name *Action* is referenced in different facts in the data too, but we want to elevate types of nodes that are searchable above such properties. The term searchable type describes the things we will be able to query for with the user interface which we will discuss in chapter 5. In our domain the goal is to build queries that find movies and persons. Hence, if we speak of searchable types in our domain, we mean the types *movie* and *person*.

The following examples illustrate the types of entities. From the type perspective nodes and entities are equal. Thus, this part applies for both of them. The *word* type describes everything we want to be able to set as a value for a fact condition later on. In table 3.2 we mark everything that is a valid *word* value with the gray background. The *word* type is different from the *number* type. The *number* is marked yellow. We will see in chapter 4 how they are treated differently.

Gladiator (2000)	*has-genre*	Action
Gladiator (2000)	*directed-by*	Scott, Ridley
Gladiator (2000)	*from-year*	2000

Table 3.2: RDF data – Focus on difference of *word* and *number* values.

Now some *word* entities have even more use than as a mere value. Some of them that occur at specific places in the relations have also more purpose. The types *movie* and *person* are subtypes of *word*. Additionally, we want to be able to search for them. We identify them by their positions at certain relations. The more refined view of the RDF data from table 3.2 with more types can be regarded in table 3.3.

Gladiator (2000)	*has-genre*	Action
Gladiator (2000)	*directed-by*	Scott, Ridley
Gladiator (2000)	*from-year*	2000

Table 3.3: RDF data – Example of binary relations and entity types.

A further outlook: the subtypes allow us to find a relation with a specific value that also has a given searchtype as a part.

The overview of all types and their relationship is depicted in figure 3.1. The type *any* represents all types and is used by relations that do not apply for a fixed type.

The whole system wants to be able to search for movies and persons by specifying constraints. Based on that we build types for those searchable kind of entities and for the complementing parts of the data. The other entities describe a fact value with a word, a number or with a linkconnector that leads to a couple of other fact parts. The linkconnector models 3-ary relations. The relationship of all possible types of entities is illustrated in figure 3.1. The arrows in the figure describe the subtypes. For example

we see that both *movie* and *person* are subtypes of *word*. The *number* type is only an abstract type because there was no use for it in the implementation. *Number*-based types are not stored in the graph because they are not relevant for the entity matching which we will see later in section 4.2.

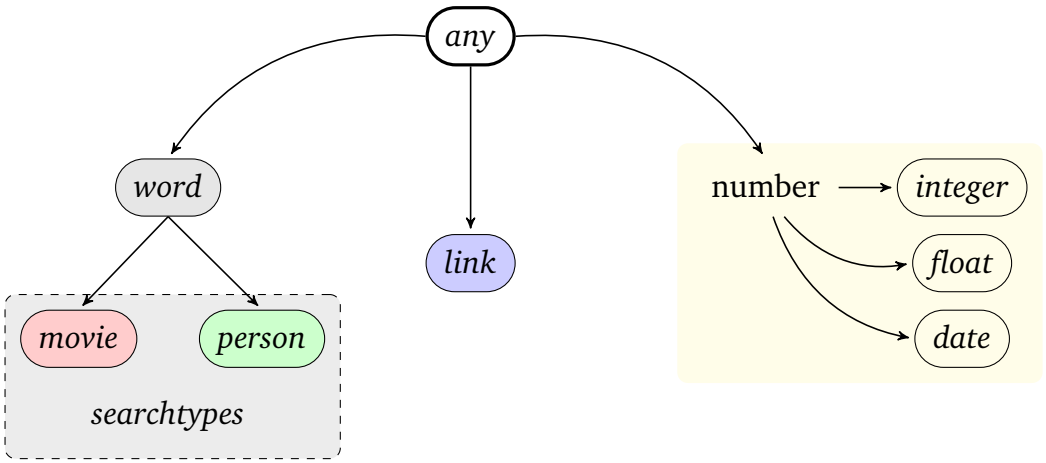

Figure 3.1: The relationship of all different types of entities. (The type of a node correlates to the type of the appropriate entity).

Figure 3.1 shows us that the *number* in the table 3.2 and table 3.3 can be even further identified as a *date*. An example that uses the *link* type is postponed to section 3.5.2.3.

Next, we will regard the relations that connect entities.

3.5.2 Relations

All the relations from the type settings (see B.2.2) have to be modeled into the graph. We will consider them grouped by common characteristics in the following parts and see how they are inserted. First of all we discuss the implication of the entity types for the relations.

3.5.2.1 Entity types and relations

The entity types are needed for defining which types of entities are connected by the graph arcs. The arcs of the graph are related to the RDF relations and inherit their preimage and image types for entities. The implication for the graph can be illustrated by visualizing a colored graph. The directed arcs in the graph receive a color depending on their preimage type. For example all arcs that leave a *movie* entity will be colored red, while arcs leaving *persons* will become green and arcs leaving *links* will become blue. The following outlook shows how this becomes useful for our fact suggestions. At a given entity that matches a searched property and a given searchtype we are trying to find, we can find ingoing arcs at the given entity with the color of the searchtype which is desired. This way we can find matching relations that are able to form a RDF triple with the given entity on the one side and the fitting searchtype on the other side.

Moreover, we will find all relations that exist in the data between the searchtype and the given entity.

Note that for changing the behavior of the relations only the settings file that defines the preimage and image types of the relations has to be changed. We continue with the groups of relations and how they influence the graph.

3.5.2.2 Binary relations

The most common relation is a binary connection of two entities. A single RDF fact consists exactly of such a triple therefore this is no surprise. Binary relations are directly added as an arc into the graph. Thereby, we want to have an arc for each searchable type which are *movie* and *person*. Thus, if both types of the relation are of searchable type we add two arcs with both directions to the graph. In table 3.4 all binary relations that define text properties are displayed. Note, that the types of *movie* and *person* are subtypes of the type *word* with higher resolution (remember figure 3.1). Examples for binary relations in the RDF data can be regarded in table 3.3.

movie	***directed-by***	*person*
movie	***produced-by***	*person*
movie	***written-by***	*person*
movie	***camera-by***	*person*
movie	***edited-by***	*person*
movie	***music-composed-by***	*person*
movie	***costumes-designed-by***	*person*
movie	***production-designed-by***	*person*
movie	***has-genre***	*word*
movie	***has-keyword***	*word*
movie	***distributed-by***	*word*
movie	***certified-by***	*word*
movie	***has-mpaa-rating-reason***	*word*
movie	***special-effects-by***	*word*
movie	***in-studio***	*word*
movie	***based-on***	*word*
movie	***from-country***	*word*
movie	***followed-by***	*movie*
movie	***follows***	*movie*
movie	***is-version-of***	*movie*
movie	***alternate-language-version-of***	*movie*
any	***is-a***	*word*

Table 3.4: Binary word-based relations.

The other kind of binary relations does connect a *movie* node with a number-based property. The difference to the text-based properties is that number-based ones can be compared with lesser or greater comparisons. These relations are not inserted into

the graph of the Entity Suggestion Matcher but instead play their role in the Relation Matcher in section 4.1 and in the SPARQL Server.

The scenario of 3.5.2.1 where we want to find all relations for a node does not quite apply to compare-based relations. For *word* nodes we actually want to consider its exact instance and its ingoing arcs for the relation suggestion. But, for building a compare-based fact we preferably want to detect such a fact by the type of the value that is used. For example for finding all relations that connect to the node that contains the words *Brad, Pitt* is different from the node that contains *Ridley, Scott* because their relations are instance dependent. Wheres, for a number-based search property like *2002* we want to treat it in the same way than *1999*. The way we want to match numbers to relations depends more on their format than on their instance. In the example of a four digit integer like *2002* we most likely should suggest date-based relations like *from-year*. All compare-based binary relations are combined in table 3.5.

movie	***from-year***	*date*
movie	***released***	*date*
movie	***has-production-date-end***	*date*
movie	***has-production-date-start***	*date*
movie	***has-budget***	*float*
movie	***has-sold-tickets***	*integer*
movie	***has-runtime-in-min***	*integer*
movie	***has-box-office-income***	*float*
movie	***has-rentals***	*float*

Table 3.5: Binary compare-based relations.

3.5.2.3 3-ary relations

First we regard an example of a 3-ary relation. An example in our RDF data for a 3-ary relation is depicted in figure 3.2.

L-c_connect_1	*in-movie*	Braveheart (1995)
L-c_connect_1	*actor*	Mel Gibson
L-c_connect_1	*character*	William Wallace

Figure 3.2: RDF data – Example of a 3-ary relation for a cast. Results into the graph clipping in figure 3.3.

Figure 3.3 depicts all components we bundled to a cast form the RDF data in figure 3.2. By structure of RDF all individual relations are binary on their own. To model 3-ary relations we introduce a connecting entity that acts as an anchor for the separate parts of the relations we want to combine. The center of the 3-ary relation is such an artificial connect entity. In our example the connect entity is named L-c_connect_1. The colors in figure 3.3 represent the different types of the nodes. The entity types determine the

type of their outgoing arcs. This is a premise for the type sensitive suggestions we will encounter in chapter 4.

Building the graph for the RDF data from figure 3.2 will result in the RDF graph clipping from figure 3.3.

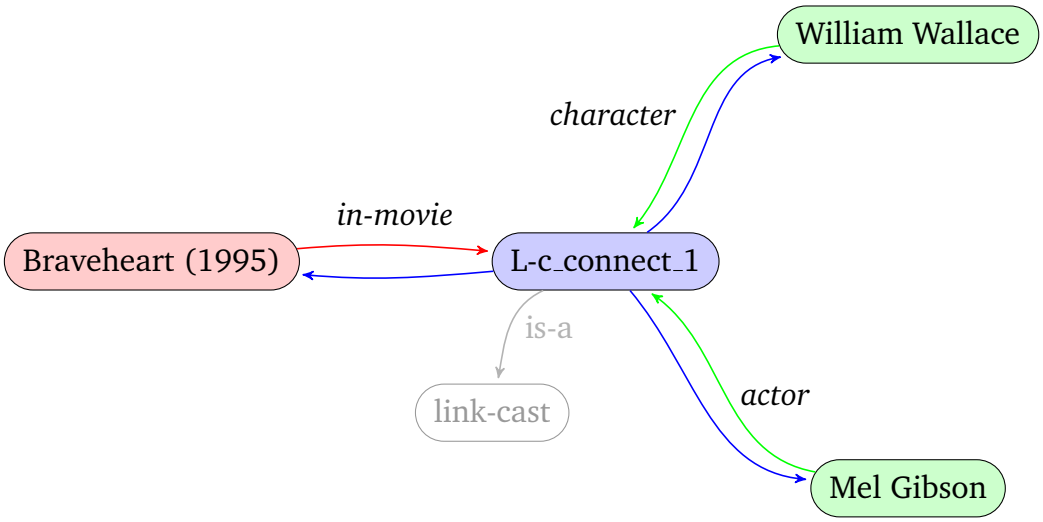

Figure 3.3: Example of a 3-ary relation for a cast in the RDF graph. (Based on RDF data from figure 3.2.)

Note, that at this point the 3-ary relations are given by the format of the RDF triples. Their form depends on rules that are used for reformatting the IMDB raw data to the RDF triples. For the sole graph we do not need the abstract associations of relations to their 3-ary relations that are depicted in table 3.6. To utilize the information which relations combine to 3-ary relations in a abstract way we will use a specification file containing this information in section 4.1. Stating these rules separately from the graph will allow us to use their dependency without an instance of the graph.

Cast		
link	***in-movie***	*movie*
link	***actor***	*person*
link	***character***	*person*
Rating		
link	***in-movie***	*movie*
link	***has-rating***	*float*
link	***has-nr-of-votes***	*integer*
Location		
link	***in-movie***	*movie*
link	***at-location***	*word*
link	***at-scene***	*word*
Language		
link	***in-movie***	*movie*
link	***language-details***	*word*
link	***has-language***	*word*

Table 3.6: All 3-ary relations.

It is always valid and possible that a fact in the data consists of only a subrelation of such a 3-ary relation as long as the subrelation contains the *in-movie* part. For example a binary subrelation for our actor *Brad Pitt* could look like
Troy (2004) ***in-movie:actor*** *Brad Pitt*.

Those binary subrelations will be handy for later applications in chapter 4. We even emphasize the subrelations by the construction of shortcuts, which is the next topic.

3.5.2.4 Shortcuts

For finding ingoing 3-ary relations of a node we add shortcuts for 3-ary relations. The goal is to find these relations by only checking the directly ingoing arcs of a node. The problem with the connecting link entity is its type. The link connector has type *link* which can further connect to *movie* and *person*. Hence, if we find an ingoing arc at a node that starts at a link we do not know if this link meets our searchtype. To verify that we have to traverse another arc further for all node neighbors of the link connector node. But, the fewer arcs we have to traverse to find the information we need the faster we will finish the current task.

Besides, we want to be able to quickly find the binary subrelations of the 3-ary relation because the user not always wants to state all parts as a fact definition. For example we want to be able to search movies with actor *Brad Pitt* without naming the character he played in it. Moreover, this enables the movie context to hit all information about a movie by just its adjacent nodes. The movie context is used for integrating the TMDB data which will be discussed in section 3.5.4.

The next topic is the construction of shortcuts. For a given 3-ary relation like the one depicted in figure 3.3 we add an arc from each searchable type to all properties of that

linked triples. After adding the shortcuts for the movie of the cast 3-ary relation from the figure this results into the graph clipping in figure 3.4.

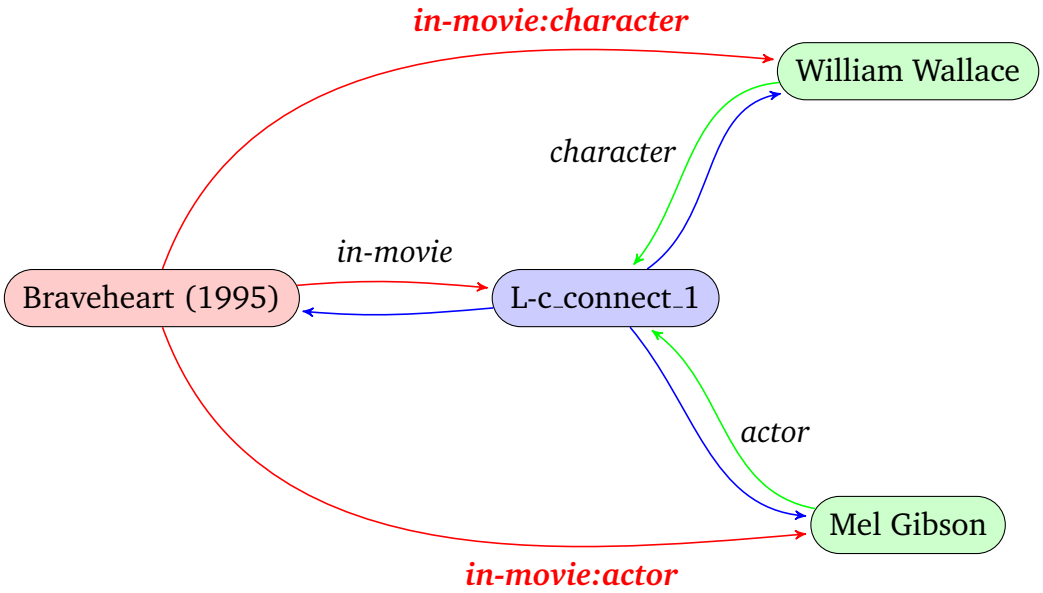

Figure 3.4: Example of a 3-ary relation for a cast after adding movie shortcuts.

Afterwards, the construction process for this 3-ary relation continues. The next step in this case is adding the two shortcuts for each person. For example for *William Wallace* there will be added a shortcut to *Braveheart (1995)* with relation-composite *in-movie:character* and the other shortcut will target *Mel Gibson* with relation-composite *actor:character*.

In the example the targets of the shortcuts are persons. Remember the type *person* is a sub-type of *word*. Thus, the construction is analog if the 3-ary relation contains *word* properties instead. The only difference in this case is that we do not add shortcuts from *word* typed nodes because this is not a searchable type.

3.5.2.5 Special relations: *has-alias* and *score*

The relations from table 3.7 are special. Aliases for entities are stored besides the graph and are not connected with arcs. This way the algorithms do not have to check at each node if it is an alias and if so, get all its adjacent nodes. The only difference in processing matches via an alias is in how the score of this match depends on the score of the main node. If the match happens because of an alias the resulting score is reduced. Besides that, we do not want to enforce any special behavior related to aliases. Hence, they are not part of the graph arc structure.

any	***has-alias***	*alias*
any	***score***	*float*

Table 3.7: Special RDF relations for the graph.

The *score* relation is solely for the SPARQL Engine and is ignored while building the RDF graph. Though after building the RDF graph the computed scores are stored and outputted as RDF triples with the relation *score*. This way the SPARQL Engine is able to use the scores for ordering. These are the scores we generate with the ranking from section 3.5.5.

Now we have all the pieces for building a RDF graph which is the topic of the next section.

3.5.3 Building a RDF graph

With the completed RDF triples and the types we have everything what we need to build a RDF graph. Here, we consider the version of a RDF graph that is used by the Entity Matcher which we will regard in section 4.2. All entities and relations from the RDF triple file are stored if the fact triple does not belong to a number- or date-based relation or if it is one of the special relations, see 3.5.2.5. The relations are stored as directed arcs. Since the arcs are directed, we will be able to find all ingoing relations for an entity. This way we can filter for relations which start at a node which type matches the type we are currently looking for. This will be considered in more detail in section 4.2.

Given an arc from node x to a node y that for example exists because of a fact triple of the form x *has-genre* y. Then this arc represents that the thing it starts from is of searchable type (or a link) and that the target thing is a property that can be set with a relation. Hence each arc means the start node has the property value at the target whit the property name defined by the relation name of the arc.

A small example with binary relations is depicted in figure 3.5. The colors represent the node types from figure 3.1. The node types are defined by the relation preimage and image types which we defined in the settings file. Note, how we ignore the number-based triple, since we will not need those for the Fact Suggestion Server. The alias names are connected to an entity with a *has-alias* triple.

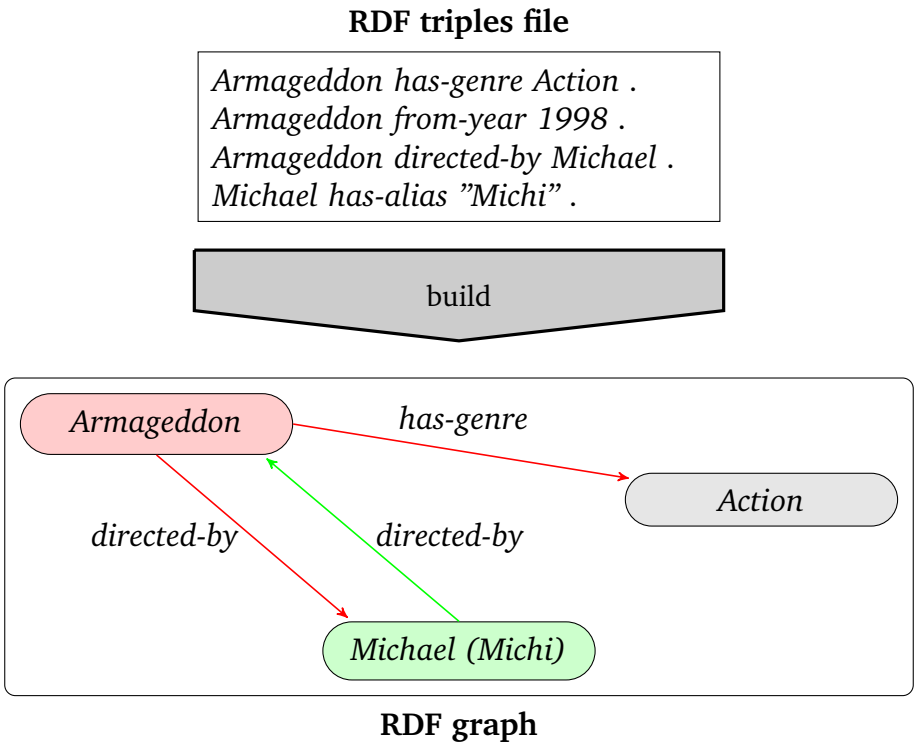

Figure 3.5: Building a *word*-based RDF graph for the given file.

The next topic is completing the graph facts with the topactors. For this we integrate the TMDB casts in the following section.

3.5.4 Integrating TMDB Cast facts

The problem is that in the downloadable subset of IMDB data popular actors with their roles in movies are removed from the data. We want to integrate the TMDB casts into the incomplete IMDB RDF data. This will resolve the missing casts of topactors in the IMDB data. The two outcomes are either we add a cast that was not in the data at all or we add a character to a cast that lacked that piece of information.

The cast is a 3-ary relation that contains an actor, the character that was played and the movie in which the person occurred. We allow occurrences of subrelations of this. In other words, it is possible that we also have just an actor and a movie in which he acted. This is a valid cast, too.

The integration is done with the following steps. For a TMDB cast we try to find a similar cast in the IMDB data. This is done by the 3-ary Relation Matcher from section 4.2.3. We check for a given TMDB actor and its TMDB movie from a TMDB cast if there exists an IMDB pendant that contains the words of the TMDB properties. The Matcher finds the third part of a 3-ary relation for the other two parts fixed. So for the cast context and a given actor and movie we will find a character if there is a matching cast for that three properties. Now we either add the TMDB cast or we amend the character. The

TMDB casts will be reformatted such that their properties are consistent with the IMDB way of stating properties.

We now have gathered all entities and relations we want to store in the graph. Therefore, we can consider in the next section how we rank the entities in the graph.

3.5.5 Ranking

For ranking the nodes in our RDF graph we use a version of PageRank [**?** , pages 107-113]. In each iteration we distribute the current rank of a node evenly along all outgoing arcs. If the node has no outgoing arcs we distribute it to all other nodes. The iteration stops if the rank differences from before and after the current iteration come under a defined threshold. So far this is just basic PageRank. With this we happened to get somewhat strange movies and actors with best scores. In our domain we expect certain nodes to have very high scores. For example the most popular actors we loaded from TMDB should receive as topactors a very high score. Since we added them later on in the integration process, they are likely to have rather few connections in the graph which results in a low score. To fix this we modify their scores after the PageRank iterations has stopped. We add a bonus to each topactor. The bonus is based on the maximum score that was found for a node in the graph and based on its popularity value we know from the TMDB data:

```
score[topactor] += maxScore + (popularity[topactor] / maxScore);
```

This moves all topactors to the top of the score chart and preserves their order influenced by their graph scores and their popularity.

The topic of the next section is the connection of the graph and its facts with the documents. This is vital for unleashing the full potential of semantic search.

3.6 Connect Movie facts and Text

Text that belongs to a movie should be connected to it. Broccoli [**?**] does not always link a plot to its movie if the movie title is not in the plot description itself. In our domain and given data, for each text occurrence there is the movie defined that belongs to this text. Hence, it is simple to store each movie entity to every text document we find in the data. To identify all other entities that are somewhat close to this movie and occur in the document is not that obvious. First we have to set how close an entity has to be to act as a candidate we are looking for in a document. Second we have to find phrases of the documents we can compare to the entity candidates. Comparing all possible subparts of each document was tried but was not feasible for the full dataset.

By this connection of movies with their context and their documents we can issue queries that find movies and persons that have a given text description with them. Adding these entities to the documents enables the context search of the SPARQL Backend [**?** , SPARQL Backend by Buchhold]. Hence, we can find movies that have *Maximus*

in one of their documents. Otherwise, we can find persons that occur in documents which also contain *the Stark Tower*. For a person search together with a plot we will find a document that contains the person entity and the plot. The result of such a search will display the document and the movie that belongs to this document.

We only consider the movie context for a document as entity candidates to avoid cross references that are not based on the content. For example in a description to the movie *Gladiator (2000)* could be a sentence like ... *Maximus played by Russell Crowe, who also played the Robin Hood in another Ridley Scott movie a couple of years later* ... . Here, the character *Robin Hood* is identified, if we consider all entities. But, this character has nothing to do with the movie *Gladiator (2000)* this document is about in the first place. By restricting the entity candidate to neighbors of the movie that is connected to the document we focus on relevant entities.

The text parts that can be inputted in the web client are called plot. The search for plots is the main appliance that is intended. Nevertheless, more types of text from the data are added for a broader text data base. Besides the plots, we process also quotes, trivia, taglines and goofs to documents. The complete list of source files that are used for this can be regarded in table B.6.

The following section considers more details about the identification of entities in a text for a movie.

3.6.1 Connecting text with entities from the RDF graph

The first part of connecting text with entities from the RDF graph was just identified as defining a movie context as a candidate pool for entities. Since for each document there is exactly one movie correlated, we only have to consider the movie context of this one movie. The term movie context in this work is defined as the set of direct neighbors that are either of type *movie, person* or *word*. Each fact we built for a movie is a direct neighbor of it because the majority of relations is binary. Note that we likewise hit all 3-ary relations because of its shortcuts (the binary subrelations of a 3-ary relation, seen in section 3.5.2.4). Given that all 3-ary relations contain the relation *in-movie* for connecting them to a movie, it holds that for all movies all facts triple per 3-ary relation are contained in a corresponding shortcut. (Compare to table 3.6).

Second we find noun phrases with the help of the Stanford CoreNLP [**?**] and its Part-of-Speech-Tagger in each document with a Java application. This tool identifies parts of speech for a word in the context of the sentence. For example in a grammatical sense the word can be tagged as a noun or verb, et cetera. This kit of natural language processing tools implements the Part-of-Speech-Tagger of [**?**]. For each sentence we store all its noun phrases. Examples for found noun phrases in our data set are *Derek Stephen Prince* and *Assistant Director Pollock*. For all found noun phrases we will check if they match any of the entities in the movie context of the movie that belongs to the current document. If not all words of the noun phrase matched any entity we continue to check if a subset of words matches an entity. If we found a match, we store its entity name for the current document.

3.6.2 Words- and Docsfile

The documents together with the results of the connecting process are stored in a wordsfile and in a docsfile. The docsfile contains each document with its ID. The wordsfile contains for each document ID all the words that occur in that document, and the corresponding movie as well as all entities which we found after checking the movie context with the Stanford candidate noun phrases. The split texts and identified entities with its associated movie are used in the context search of the SPARQL Engine [**?** , SPARQL Backend by Buchhold].

With the complete RDF graph we now have everything we need to explain the Fact Suggestion Server in the next chapter.

Chapter 4

Fact Suggestion Server

The Fact Suggestion Server is the main server-side contribution of this work. It enables the exploration of the data for inexperienced users. This is achieved by different suggestions that aim to guide the user. The generation of the suggestions and the therefore needed gathering of information from the underlying RDF graph is the main task of the tools presented in the following sections.

Working together: Before considering in detail what different kinds of suggestions are made, we overview how the server and the user interface interact. Besides, we regard how the different Matchers are interlocked.

The server applications in the section of this chapter apply for fact input on the user interface which we will regard in section 5.1.3. A general round for a fact suggestion starts with a given searchtype and the fact input text. During fact input the autocompletion offers options to search for. The autocompletion itself uses the methods of foremost the Relation Matcher and if there are no matches found, afterwards the Entity Matcher is called. Results found by the autocompletion are suggested. On sending a fact suggestion request from the web client this order of events repeats. First, the Relation Matcher checks for any relation clues in the given input. If something was found the result is offered as a suggestion list. If the Relation Matcher did not find any relation, it is the Entity Matchers turn. The Entity Matcher will find all entities that match the given fact input and all ingoing relations with the searchtype as preimage type. Again all found results will be presented as suggestions to choose from.

The reloading is not directly dependent on the fact input. Reloading happens if a part of the result should be shown that was not already sent to the web client.

Next, we start by regarding the different Matchers and their scope of duties.

4.1 Relation Matcher

The main task of the Relation Matcher is to identify relations and their values in a requested list of words. For example for the request *"from 2000"* the relation *fromyear* with the value *2000* will be found. Next, we consider an example with a 3-ary

relation: the request text *"Pitt as Mr. Smith"* will be matched to the first relation *actor* with value *Pitt* and as the second part with the second relation *character* and its value *Mr. Smith*. This illustrates that parts of 3-ary relations that are found will be completed.

First we regard what components are needed for the Relation Matcher.

4.1.1 Composition

The Relation Matcher does not need to know the graph structure, but the names of the relations. Besides, we have to provide the information which relations are part of a 3-ary relation.

3-ary relation bundles A specification file defines what relations are bundled to 3-ary relations. This file contains the same information as depicted in table 3.6 on page 26.

Alternative relation signifier For improving the range of hits, we specify alternative names for relations. The example from the introduction to this section is the word *as* that is an alternative for the relation *character*. Alternatives do not have to be exclusive for one relation. As an example alternative that belongs to more than one relation we name *made*. The verb *made* can mean both *directed-by* and *produced-by*.

Our goal of enabling a smooth exploration of the data, despite few knowledge about it, needs a broad range of alternative relation names. This way we guide users to the actual used relation names they want to apply, instead of requiring from them to name the relation exactly on their own.

Next, we consider the actual process of finding relations in a request.

4.1.2 Finding relation matches

We either allow all relations for matching or only a part of them that has a number-based type as image. First, we check if there is a number with at least two digits in the request. If this is the case the Relation Matcher assumes the relation that is searched has to map to a number-based type. Hence, we restrict relation matches to number-based types, which is done by simply using the defined preimage and image types of the relations.
Moreover, each request to the Relation Matcher states a searchtype (*movie* or *person*, see 3.5.1). This restricts the pool of candidate relations even further by again utilizing the defined types around each relation.

The general matching process tries to find a relation name or relation name beginning for each given request word. If a part of a 3-ary relation was found we check for a candidate in the remaining request-words that matches another 3-ary relation part. If there is no such secondary part identified, we append the name of the relation to show the possibility. This is illustrated by a couple of examples in table 4.1.

Searchtype	Request words		Result
movie	*"from 1999"*	→	**from-year** *1999*
movie	*"role Maximus"*	→	**character** *Maximus* **actor** ...
person	*"film Gladiator starring Crowe"*	→	**in-movie** *Gladiator* **actor** *Crowe*
movie	*"Gibson as Wallace"*	→	**actor** *Gibson* **character** *Wallace*

Table 4.1: Examples for relation matching.

Another part of information that is needed for the web client to process suggested 3-ary relations is the name of the connecting third relation of the 3-ary one if the other two are found. For example the result from table 4.1 that has searchtype *person* is the following: **in-movie** *Gladiator* **actor** *Crowe*. A 3-ary relation has to be connected to the searchtyped object we look for. In this case we connect the search object with the relation **character** to the result. The resulting query will therefore find all characters that occur in movies which title contains *Gladiator* and have an actor *Crowe* connected with the same linking entity. If the Relation Matcher found already both parts of a 3-ary relation there is only one connecting relation possible. Even if only a subrelation is found there are at most two possibilities as connecting relations. Thus, the few possible connectors will be added to the request answer and suggested separately.

The following section covers the Entity Matcher which is the most integral fact suggestion tool for exploring entities and related relations.

4.2 Entity Matcher

The Entity Matcher finds entity candidates for a given list of words and all their ingoing relations. The result is grouped by the found relations with their correspondent entities. A searchtype is part of the request as well. Therefore, only relations with appropriate preimage type are found. A small example with a mini graph, a request and its resulting fact suggestion is depicted in figure 4.1. In the example the searchtype is *movie*. The fact description is "peter" and matches the three entities in the graph that are marked bold. Since our searchtype is *movie*, only the two persons are relevant hits. Because, the two persons have an ingoing arc from a *movie* entity. This is not the case for *Peter Pan (2003)*. The two persons which we found, will be grouped together, since they are found with the same relation. The results correspond to two possible facts. One fact for the movie we are looking for could be selected. Assuming we select the *directed-by Peterelli*, this fact specifies that we are looking for a movie that is directed by someone called *Peterelli*. The searchtype corresponds to the type of the object we are searching. The suggestions represent relations with values that can serve as a fact definition.

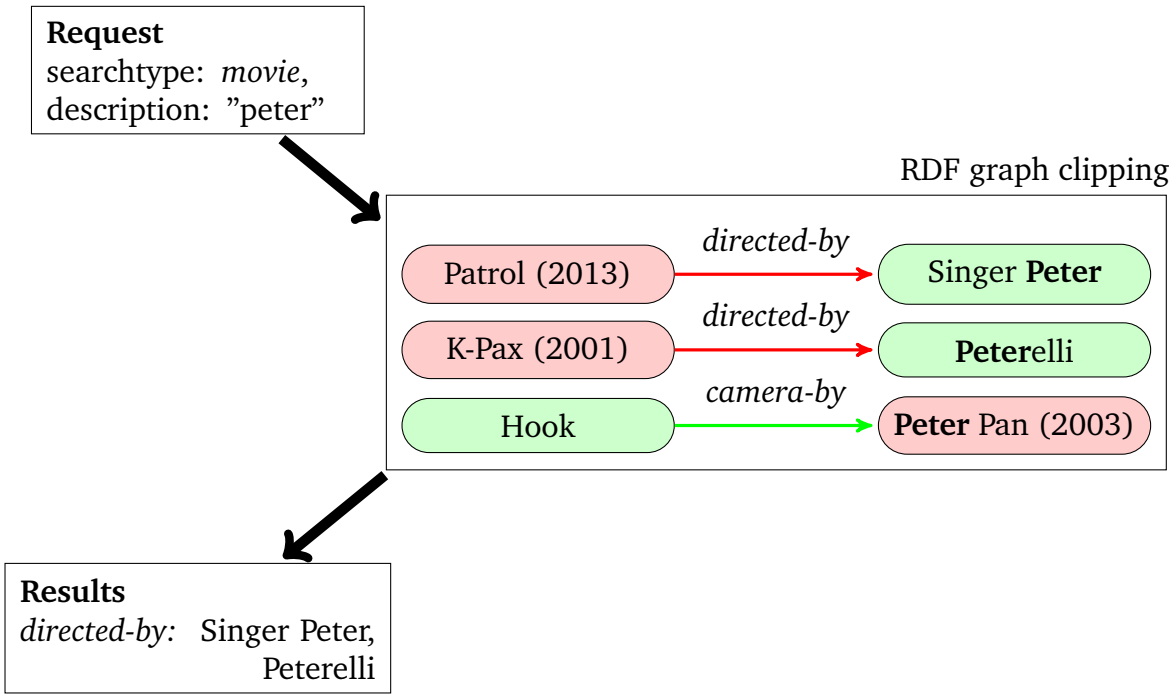

Figure 4.1: Example for entity matching. Entities that contain the description are marked bold. But, only hits that have an ingoing arc of the searchtype *movie* are relevant for the request.

In the following sections we will regard how the single steps will be performed to achieve the fact suggestions.

4.2.1 Composition

The Entity Matcher uses the following components to fulfill its task. First we have to find matching entities in 4.2.1.1. Second we gather all the related relations in 4.2.1.2.

4.2.1.1 Matching entity names with an inverted index

Finding entity matches will be done by an inverted index. The following segment will state how the inverted index will be build and what it contains.

Entities are stored and identified by their ID. For each word of the entity we consider all prefixes until a maximum size. The maximum size is stated on server startup. (We most commonly used three or four.) Next, we build an inverted index. The inverted index maps prefixes to all entities which contain a word that has this prefix as a beginning. Note, that all those entity ID lists will be sorted since we iterate through the entities in increasing order of their IDs to map their prefixes to IDs.

Let us consider a small example for an inverted index for entities. The example in table 4.2 shows how we build an inverted index of prefix size 2 for the given two entity

names. Each of the entity names is split by the specified separator. Here, we split the entity name ”Bill Berg” into the two words ”Bill” and ”Berg”. For each of the words of the split we add the corresponding entity ID to all possible prefixes until a defined maximum prefix size. As prefixes we store only lower cases.

ID	entity names
0	”Barbara”
1	”Bill Berg”

$\Longrightarrow$

$\Longrightarrow$

Prefix		list of entity IDs
[b]	$\rightarrow$	0, 1
[ba]	$\rightarrow$	0
[be]	$\rightarrow$	1
[bi]	$\rightarrow$	1

Table 4.2: Inverted index for entities with prefixsize 2 and *space* as word separator.

For matching a set of given request words $\{w_i\}$ to entities, we use the inverted index. Per given word w_i we retrieve the entity ID list for the prefix of the given word. The prefix of w_i which we consider is the substring of w_i from the beginning until we fetch the maximum-prefix-size amount of letters. Prefix of w_i as a pseudo definition $pre(w_i) := w_i[0 : \text{maxprefixsize}]$. Next, we get the entities that contain all given prefixes $\{pre(w_i)\}$. We perform a list intersection of the found entity ID lists for each $pre(w_i)$. Since the lists are sorted this is fast. We look through all lists at once. If the current value of all lists is equal, we add it to the intersection result. Otherwise, in the list with the minimum value we check the next value in the list and compare it to all other current values. We repeat this until one list reaches its end.

Afterwards, we filter the intersection properly by the full given request words $\{w_i\}$. Our filter method checks if all the request words from $\{w_i\}$ are contained from the beginning in any entity name word. The remaining entities are the matches for the given words. For our example from table 4.2 we can consider the request ”Bil Bergson”. Both of the request prefixes are $pre(\text{”Bil”}) = \text{”bi”}$ and $pre(\text{”Bergson”}) = \text{”be”}$. Intersecting the appropriate lists leads to the entity ID 1 as prefix candidate. Now we filter for the full request words. ”Bil” is contained in the entity name word ”Bill”. So far this entity remains a candidate for the request. But, the next request word ”Bergson” is not a beginning of any of the word parts of the entity name ”Bill Berg”. Hence, we filter out the candidate entity ID 1 and our resulting match is empty.

Returning all matches with the filter method contains from start helps in finding all possibly interesting suggestions later on. Even, if a letter is missing at the end of a request word, we can provide suggestions.

Aliases are added to the inverted index, too. They have their own alias IDs. An ID can be recognised as an alias by its size. The first alias ID is the next bigger ID after the biggest entity ID. For the name matching they are treated equally, both for inserting into the inverted index and matching.

The next component of the Entity Matcher stores the fact data.

4.2.1.2 Gathering information form the RDF graph

The RDF graph is an important component of the Entity Matcher because it has knowledge about the structure. For found entities we want to gather all ingoing arcs and their corresponding relations. This way we can suggest entities with their possible connections in the graph.

The next topic is the computation of suggestions of the Entity Matcher.

4.2.2 Computing entity matches

As stated in the introduction we want to find suggestions for a searchtype and some given words.

Precomputed Matches First we check for a single given short word if we have a precomputed result for it. Those words will most likely find many matches therefore we save computation effort by paying with some storage space. The intention for this is to make the server responsive by avoiding computation for short inputs repeatedly.

Next, we retrieve all matching entities and aliases for the given words by using the inverted index we build in section 4.2.1.1.

With the entity matches we now can gather all relations from the graph that are part of the RDF graph instance. Again we only consider relations that have the searchtype as preimage type. Additionally, for all entity matches that are an alias we replace this alias with all its correlated entities. The next step groups by the relations we found. As a result we have all relations that connect the searchtype to an entity match for the given request words. Besides, per relation we have all entity matches that occur with this relation in the graph.

The smallest suggestion unit of the result is therefore a found relation with an entity that matches the given words. This way every suggestion relates to an actual segment of the graph structure of the data. The searchtype with a found relation and an entity match directly specifies a SPARQL condition. Such a unit we call a fact. The intention of the suggestion is to specify facts that will specify a SPARQL query which is an integral topic of chapter 5.

The results are sorted by their graph score. Entity matches that only were found by an alias receive a reduced score. First all entities per relation are sorted. Afterwards, we sort the relations by the maximum score of their entities.

The first part of the resulting suggestion will be sent as a JSON to the client. While we keep the complete suggestion for further reloading requests which we will view in section 4.2.4.

Entity Matcher and Shortcuts At this point we emphasize the impact of the shortcuts which we added to the graph in section 3.5.2.4. Only because of the shortcuts the procedure we just described finds subrelations of the 3-ary relations. Without the shortcuts we would have to perform more checks on the graph and even traverse the graph deeper to identify 3-ary relations. Hence, the shortcuts reduce the computation effort for the Entity Matcher. For example a popular subrelation of the Cast 3-ary relation is the construct that matches movies to actors. This is done by connecting the searchtype *movie* with the connecting relation *in-movie* with a connector entity which is further connected by the relation *actor* to an *person* entity we found.
For the client to be able to process those subrelations the connector relation is part of the result JSON. The connector relation from the example is in this case *in-movie*.

Finding the entity matches for the request words is at maximum linear in the maximum size of prefix lists that are intersected for the request words. Filtering needs at most n comparisons with $n := (\text{nr. of entity candidates}) \cdot (\text{nr. of request words})$. Gathering the relations for all entity matches depends mostly on the graph structure that determines how many matches are found in average. Sorting of the entities per relation, regrouping and than sorting the relations by their score is dominated by the sorting complexities of $O(s \log s)$ for the different sizes s.

A different task we want the Entity Matcher to perform is the topic of the next section. After regarding the subrelations we now consider the complete 3-ary relations.

4.2.3 3-ary relation matching

The Entity Matcher is able to search for 3-ary relations by a separate call.
The input for this task is a bit different than the inputs we considered so far in this chapter. The 3-ary relation matching happens after a suggestion from the Relation Matcher is selected that contains already two relations from a 3-ary relation. For both selected relations there is a separate input text as a value. The searchtype is given, too.

The procedure starts analog to the simple entity matching by finding all entity IDs that match the inputted text. This is done separately for both given relations with their values. Afterwards we check for those two ID sets if they are part of a 3-ary relation in the graph that also has a relation to connect to the searchtype.

The whole 3-ary relation finding process uses an array (std::vector) for the smaller set of entities and a hashset for the bigger ID set. Than for all IDs in the small set we check ingoing arcs. Here, we only want to consider *link* typed relations (relations inherit a type from their preimage node). Since 3-ary relations have at most three relations bundled together, the connecting entity has a fix maximum number of outgoing arcs.

Now consider an entity that is connected by a *link* relation with the current small entity ID as target. Thus, this is a connector entity we introduced in the RDF triple preparation. Let us name this connector entity L.

The outgoing arcs of L are the ones we have to compare to the relation correlated to the big entity ID set. Since L is the connector entity, it bundles the entities of the current 3-ary relation together. Arcs from L to a neighbor imply this neighbor is part of this 3-ary

relation. If the relation of the big entity ID hashset is not one of the outgoing arcs we are done with L because it either is not the kind of 3-ary relation we are currently looking for or it does not have the amount of information we desired. Whereas, if we found an outgoing arc from L that represents the relation that correlates to the big entity ID hits the check if this is one of our matches is fast. The check if this arc targets a desired entity from the big entity ID hits is done in approximately constant time thanks to the hashset we use for the big entity-IDs. Hence, the 3-ary relation gathering is linear in the smaller entity hit size. At each link there is a fix amount of out arcs to check for which we each need constant time. Additionally, we have to perform two entity matcher runs with their complexity from the previous section.

The case of a found 3-ary relation in the graph is illustrated in figure 4.2 in the following scenario. Given the two requested relations r_1, r_2, the small entity hits as $\{e_1\}$ for relation r_1 and the big entity hits as $\{e_3, e_4\}$ for relation r_2, a complete match in the graph could look like this:

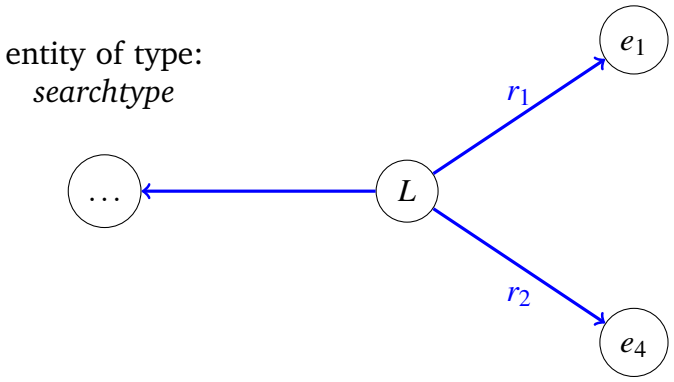

Figure 4.2: Example of a 3-ary relation match in the graph.

The computation time for the important Matcher methods will be evaluated in detail in section 7.1. They are essential for acceptable response times. In the next section we take a look at a part of the utility functionality of the server.

4.2.4 Reloading Entity Matches

After each fact suggestion a match object is stored at the server from which we can request more parts. In general only the top-n entities for the first relation are sent to the client and are there displayed. Additional entities or the result entities for another found relation will be sent dynamically by the server on demand. This reduces the size of the answer JSONs per request.

The following section considers in more detail the matching of words. How we define when two words match is an important part for entity matching. Finding entities that are somehow described is vital for utilizing the fact data.

4.3 When does a given word match a name?

One important question for the accuracy of the suggestions is when way say a given input word matches an entity name or a relation name from the graph.

The basic theme for words matching so far considered if two sets of words match. We split text into list of words, separated by a token depending on the case. In the course of the whole system tokens are for example space " " or the space URL pendant *%20* or a comma. This is done for the names of entities and relations in the building of the appropriate Matcher and for the given request text for each request. The Matchers state that the given request words match a name if the set of the request words is contained in the set of all word parts of that name.
In a more formal manner, we define when a finite set of request words $\{w_i\}$ matches an entity name e with a separator token s. Consider the split of e by s named $\{e_j\}$. For example the split of the entity name "Russell%20Crowe" by the token "%20" is {Russell, Crowe}. Now, $\{e_j\}$ matches $\{w_i\}$, if $\forall w_i \in \{w_i\}\ \exists k,l \in \mathbb{N}$ such that $e_k[0:l]$ is equal to w_i.

Now, we regard the basis for the set compare. The basis is the question if a given word matches a word from the entity name.

The current implementation of the server checks if the given word is the beginning of the name word. Or in other words: if the given word is contained from the start on in the name word. The part from the above definition we consider here, is if it holds that $e_k[0:l]$ is equal to w_i. This check is performed for example in the inverted index from section 4.2.1.1. The filtering by complete words after the prefix intersection does the contained from start on check, too. For a given set of request words, we say it matches an entity name, if each of the request words is the beginning of one of the words from the entity name.

The Relation Matcher performs this check also for deciding if the current word of the input is a relation name. The Relation Matcher is even stricter since it only accepts two words as a match if they have furthermore a length difference of at most one. Without the further restriction there will be often misinterpreted entities as relations. This happens because we allow prefixes of relation names as alternatives which is handy for the autocompletion. An example where we want the strictness of the Relation Matcher is depicted in table 4.3. Without the length compare we only check if the input contains the prefix *fro* for the relation *from-year* and we would accordingly match *frodo* to this relation. This would be unlikely the desired behavior.

Input		Result
fro	→	*from-year*
frod	→	*from-year*
frodo	→	∅

Table 4.3: Example that illustrates the strictness of relation matching.

Let us consider this approach. The check is performed fast but the approach has further

implications. Next, we consider a drawback of the contains from start check.

Spell checking: The drawback of the contains from start check is its poor reaction to typos. Most typos result in finding nothing or the wrong names in the graph. The autocompletion helps to prevent typos with the selection suggestions. However, adding a form of spell checking is definitely a future project. Spell checking is yet another way of improving usability. Some possibilities would be allowing matches with some tolerated edit distance or utilizing phonetic similarities. Another approach could integrate a k-gram index and than checking for a minimum amount of common k-grams. For more details see [**?** , pages 197-203].

Yet, we have to account of cases with many possible corrections. Hence, after adding a spellchecker, we have to resolve correction possibilities by a user selection, a meaningful ranking, a context dependent scheme or a combination of some of them.

For the Relation Matcher we could add more alternatives that contain common typos in the prefixes for the relation names. This would not require any code changes but merely additions to the appropriate settings file that states the alternatives for the relation names. Relation names are stored in an inverted index of prefixes similar like in section 4.2.1.1, hence with the defective alternatives we would perform a kind of spell checking and even hit relation names with typos. The task to enable this is tracing such common typos.

Now, we regard a way to increase the odds for finding matching names that is actually used by MovieSearch.

Filtering stopwords: If a Matcher finds no results for the given words, we retry the suitable process after filtering out all stopwords. A stopword is a common word in natural language that most likely does not carry a lot of distinguishable meaning. The list of stopwords we use can be regarded in section B.2.3. Repeating the matching with only relevant words helps because a smaller set of words only increases the chance of finding a match. If we want to cut some words, it is most likely best to start by cutting stopwords. Note, that there are examples that consist mostly of stopwords and are nevertheless meaningful like for example *"To be or not to be"*. We address such cases because before stopwords are removed we try finding a match with them.

The next chapter presents the user interface. The user interface will issue the suggestion requests we encountered in this chapter.

Chapter 5

User Interface

The topic of this chapter is the user interface of MovieSearch. First we regard the separate elements of the MovieSearch user interface and their purpose in detail. Second we state the scope of use cases that we want to be able to process. Third the user interface is compared to a couple of alternatives.
The interface allows the user to specify conditions. A condition is either a fact or a plot. Suggestions for fact descriptions are generated dependent on the given searchtype (*movie, person*). From the suggestions we select or find an actual fact from the RDF graph. We will regard a couple of examples for the suggestions in section 5.2 while presenting use cases for MovieSearch. The plot is a text that has to occur in a document which is connected to the search object.

Now we begin with the introduction of the user interface of MovieSearch.

5.1 MovieSearch UI – components in detail

Overview First of all we take a quick overview of the MovieSearch website. The website can be seen in figure 5.1. On the left side is the major section in which the user can build the conditions for the query. The middle contains the current query and the results. The current query area is the dark gray box that contains **E**, **F** and **G**. Without any conditions the current query area is hidden. It should not distract from the left section where the first input has to be performed. If the current query is empty, some popular movies or actors are shown. The populars are a preview that visualizes the movie theme of the website. At the top right corner various settings can be adjusted.
After the first condition is given, the display of the current query fades in at the top in the middle. Figure 5.1 depicts the state of the user interface after the use case from section 5.2.3 is completely inputted by its conditions.

In details view Figure 5.1 depicts the user interface with red letters marking important interface parts we will discuss in the following sections.

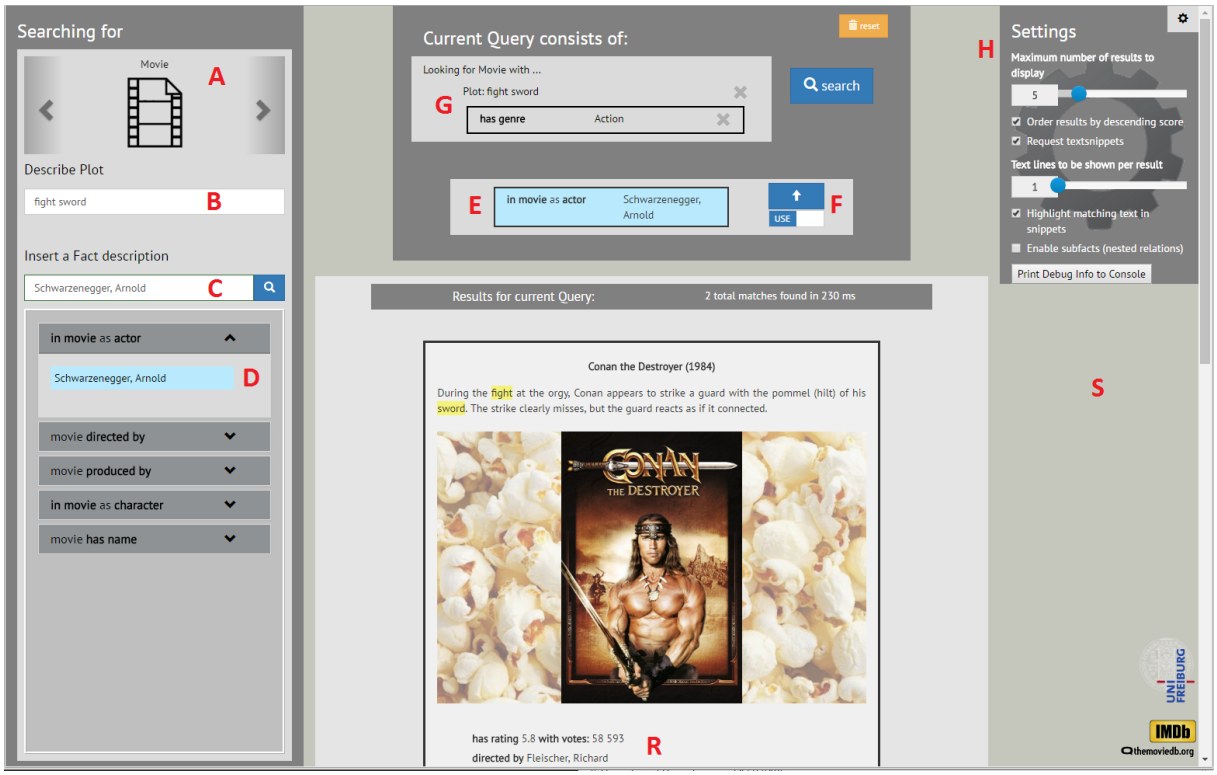

Figure 5.1: MovieSearch user interface with markings for a detailed description.

5.1.1 A: Searchtype

The searchtype specifies what we are currently looking for. Possible types are Movie or Person. This is the searchtype in the Fact Suggestion Server requests we use in chapter 4 at various places. It defines the preimage type of the relations in the suggestions. Hence, the searchtype influences implicitly what the current query looks for by the type restrictions on the suggestions. Besides, the current query uses the searchtype directly if no facts are given.

5.1.2 B: Plot input

This is the input field for plot text.

5.1.2.1 Improving plot words

If the SPARQL query with the previous plot did not find any results the server tries to improve the plot words. Stopwords are filtered out, some words are reduced to their stem and wildcard search will be used in the next query. Here, the same stopwords from section B.2.3 are used. The stemming is performed rule-based. Wildcard search

is a feature of the SPARQL Engine [**?**] where, if a word has as last character a $*$, also words with the same beginning match. Improving the plot increases the odds of finding results.

5.1.3 C,D: Fact Suggestion

C marks the fact description input. This equates to the request text we considered in chapter 4.
D marks the area where the suggestions of the Entity Matcher or of the Relation Matcher are displayed and selected. Here, we suggest possible facts for the inputted fact description (**C**) and the searchtype (**A**). The whole fact suggestion aids in finding facts to specify a condition. The conditions define what the query looks for.

In the fact description input field we use the autocompletion from the Fact Suggestion Server. This is the topic of the next section.

5.1.3.1 Autocompletion

The autocompletion of the fact description input (**C**) has two intentions. First we want to help finding entity names and relation names. Second we want to hint at the possibility of a 3-ary relation input.

To achieve this we utilize the Relation Matcher and the Entity Matcher. Similar to a fact suggestion round, we first test for relations in the input. If we found some relations they will be shown as autocompletion suggestions. If none were found we present the top-*n* matched entities ordered by their score as suggestions.

After the autocompletion the Relation Matcher or the Entity Matcher are called for the fact input (**C**). Their resulting suggestions lead to the selection possibilities which we regard in the following section.

5.1.3.2 Selection possibilities

There are different types of suggestions depending on the searchtype and the format of the fact input. Suggestions have word-based entity facts, compare-based relations or 3-ary relations as rough categories. Next, we consider suggestions grouped by which Matcher made them.

Suggestions of Entity Matcher Figure 5.1 at marking **D** shows the fact suggestion for a binary fact with the type *word*. The header of the boxes contain the relation that is represented by this box. Inside the accordion box are all loaded entities that were found for the relation in the corresponding header. The user selects an entity in one of the boxes. This sets the condition with the relation and entity as the candidate fact at **E**. We will regard the candidate in section 5.1.4. More entities can be loaded if there

are more matches than currently displayed. Entities per relation are loaded on demand respectively on the first click on that relation header.

Selection boxes of Relation Matcher If the Relation Matcher found some relations each of them is displayed in a selection box. 3-ary relations are displayed as its binary subrelation and as a complete 3-ary relation. Depending on the fact object type the selection box has a different appearance. All boxes are suggested in the area around **D**.

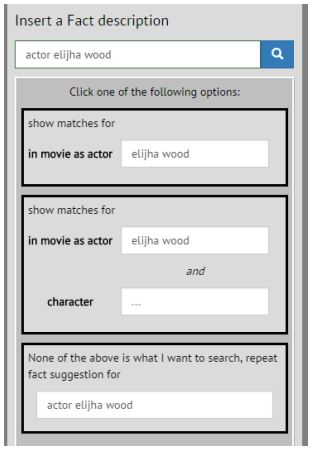

Figure 5.2: Word.

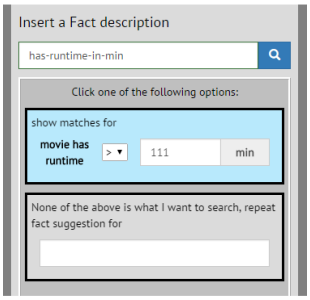

Figure 5.3: Number.

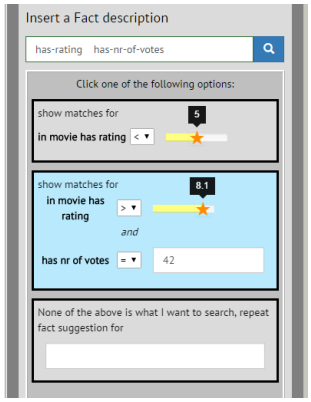

Figure 5.4: Rating.

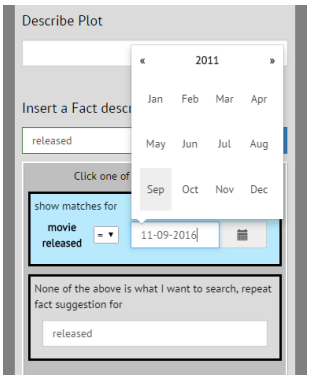

Figure 5.5: Date.

Figure 5.2 shows word-based selection boxes. The first box uses a binary relation, that issues an Entity Matcher run with only this relation allowed. The second box uses the displayed 3-ary relation for an 3-ary relation matching of the Entity Matcher.

The other boxes and their relations state a condition on their own without a call to the Entity Matcher. They are all compare-based. That means we have to specify a compare type and a value to define a condition. Compare types are *smaller, equal* or *bigger*. Figure 5.3 is compare- and number-based. The values could be a quantity, for example of votes or number of minutes, or an amount of money. What value is desired depends on the actual relation.

The rating from figure 5.4 is a special case of the number-based case because for the rating we know predefined values as boundaries. The rating is the IMDB rating which ranges from zero to ten. To enforce this we use a slider instead of a number input.
Figure 5.5 is compare- and date-based. The values are a date. To avoid struggling with input format we included a datepicker to make the date selection smoother.

The bottom box in all the figures can be selected if none of the above boxes quite matches the desired fact description. This box will directly issue an Entity Matcher call without any fixed relations. Hence, all relation possibilities will be displayed.

5.1.4 E,F: Candidate Fact

E presents the current selected fact as a candidate. The blue highlighting indicates which fact suggestion is selected as the current candidate. If the current candidate is added as a condition to the current query, can be toggled on and off. This way a user can see how the results change with this condition. The user can decide if he wants to add the condition to the current query or if he prefers to change the candidate fact. After some suggestions are found, the first suggestion that defines a condition that further specifies the current query is automatically selected as current candidate.

F has a toggle button that either uses the candidate for the SPARQL query or not. This way the user can see how the results change with this condition. Now he can decide if he prefers to search for another condition or if he wants to add it to the current query.

F arrow button adds the candidate fact to the current query. This way we add stepwise more conditions to the current query. A fact will only be added if it defines a condition that is not yet contained in the current query.

5.1.5 G: Current Query and SPARQL

The current query gathers and displays all conditions. These are the plot words and the facts. Conditions may be removed at this place. From the current query and the candidate fact below at **E** the SPARQL queries are built and answered by the SparqlEngineDraft Backend [**?**].

5.1.6 H: Settings

Area where a couple of settings can be adjusted:

- Maximum number of results to display for current query.
- Should the results be ordered.
  If a plot word is given we order by textmatchscore. Otherwise, we use the RDF graph score to order.
- Should we display document snippets where plot matches occur.
- Textlines to show, SPARQL Server property.

- Should the plot words be highlighted in the document snippets.
- If subfacts (nested relations) should be enabled.

5.1.7 R: Results

Here we display all found results. Bonus information is loaded and if something was found it will be added to the appropriate result.

5.1.8 S: Subfacts – nested relations

A subfact is a relation whose fact object is not an entity but further specified by a relation and its value. An example subfact condition is *movie directed-by X* together with *X actor:in-movie Gladiator*. Here, the director is not specified by an entity but by another relation. This is more an advanced feature for experts and is therefore not part of the evaluation. Currently only word-based relations can participate in a subfact.

The subfacts have to be enabled in the settings. Than the Relation Matcher will add an appropriate selection box for relations that suit as a first subfact part. After a first subfact part is selected, a fact suggestion for the second subfact part opens at **S**. It has the same properties than the fact suggestions at **C** and **D** but only specifies second parts for subfacts. For connecting subfact relations in a valid way we again utilize the relation preimage and image types.

Figure 5.6 shows the selection of the second part at **S** with enabled subfacts.

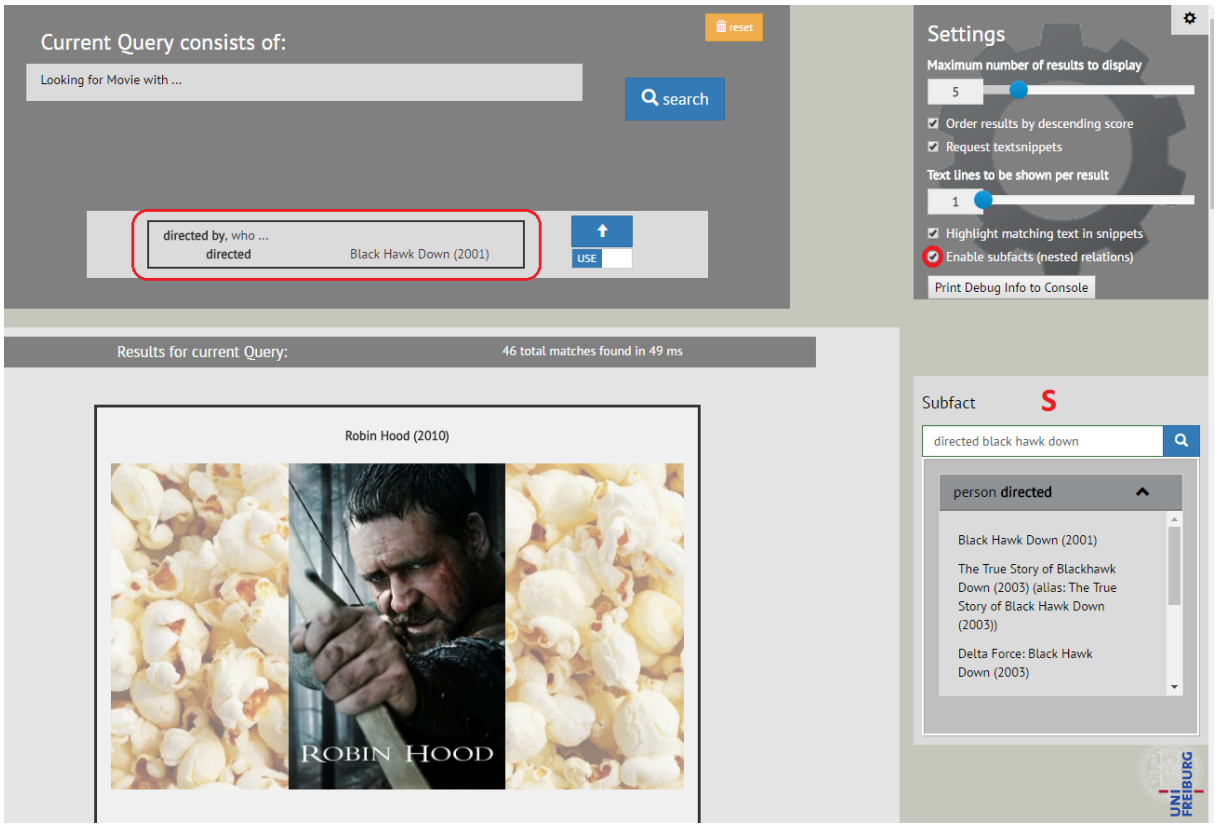

Figure 5.6: Finding a movie that is directed by the director of the movie *Black Hawk Down*.

Instead of the new fact suggestion area, we could reuse the existing one on the left-hand side. But, it was rather confusing using the original fact suggestion area for this because for example the type from **A** is misleading and it would overwrite the current suggestions.

5.1.9 History

We store the following history state changes ADD, REMOVE, RESET and PARSE. On firing the history.back() the last state change is undone.

ADD = adding a fact to current Query
REMOVE = removing a fact from current Query
RESET = clear current Query and inputs
PARSE = perform a fact suggestion round on current fact suggestion input

The next section integrates a couple of use cases into the user interface. They will serve as examples for the suggestions and query building. Furthermore, the use cases

illustrate the scope of tasks for which we want to be able to specify conditions. These conditions will form a semantic query. Altogether, by drafting which queries we can build, we also set the scope of appropriate results that shall be possibly given.

5.2 Use cases

This section covers the topic of what use cases MovieSearch aims to fulfill. The building of semantic search queries for non-experts has the following three aspects we want to emphasize.

5.2.1 Exploring – without knowing exact relation names

The data should be explorable through the user interface. That means even for not exactly known relation names the user interfaces guides the user to a suggestion. Flexible suggestions are important for specifying conditions.

An example for the exploring aspect is the use case:
Find movies made by Jerry Bruckheimer.

A suggestion relation that correlates to the fact description *made by* has to be found. In this case this would be the relations *directed-by* and *produced-by*. These are the autocomplete suggestions in the drop-down menu of the fact description input field depicted in figure 5.7.a).

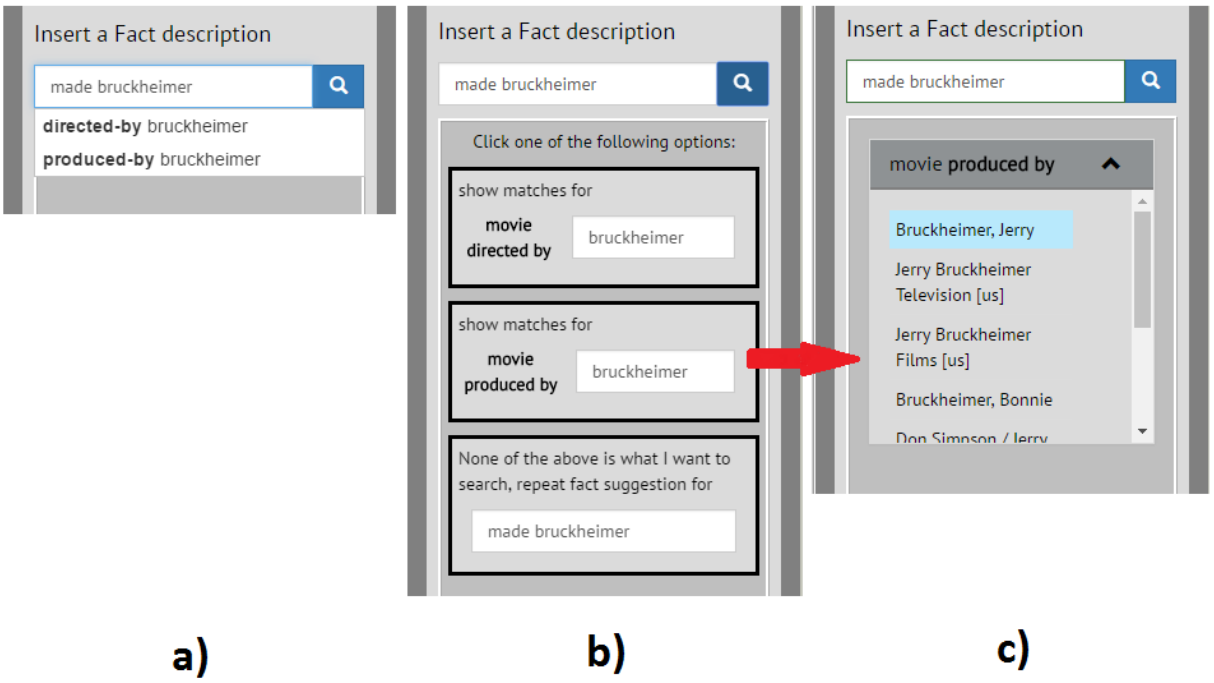

Figure 5.7: Suggestions for the exploring use case: *Find movies made by Jerry Bruckheimer.*

If we do not select an autocomplete suggestion and click the search button instead, the results are displayed in figure 5.7.b). Here, all found relations to the input are displayed with the found value from the input. We select ***produced by*** since we think it suites the task best. Note, that we could change the input for each of the relation suggestion boxes. Our box selection leads us to the fact suggestions depicted in figure 5.7.c) where we are satisfied with the first entry. The header of this accordion box represents the relation. All the table entries are possible values for this relation. Our current selection ***produced by*** *Bruckheimer, Jerry* is a fitting condition for our current use case.

As an alternative we would have reached a similar fact suggestion field by only inputting *Jerry Bruckheimer*. In this case there would not only be one accordion box, but one for each found relation.

5.2.2 3-ary relations – connecting fact parts

We specifically want to be able to search for 3-ary relations. In the domain movie it is of interest to search for movies where a certain character was played by a given actor.

An example for such a cast use case is:
Find movies where Frodo was played by Elijah Wood.

The phrasing of the use case implies that we only want movies where the given character has to be actually played by the given actor. Most likely, if one fact description contains an actor and a character the fact intends on connecting those two parts. We provide this connection of the actor with a character by a 3-ary relation.

If we would build two separate facts to model this use case as a query, we would search for movies where a character exists of the name *Frodo* and where the actor *Elijah Wood* appears as any character. But, with both fact parts in one fact description this is unlikely the desired condition.

The input process for such a 3-ary relation is depicted in the series of figures 5.8. The autocompletion in part 5.8.a) detects a keyword for a cast relation ... *as* ..., hence it suggests the relation parts of a cast. In figure 5.8.b) the selection boxes for using the binary subrelation or the full cast are selectable. Since we have a full cast information, we select the middle box. This leads to the 3-ary matches in the RDF graph which are depicted in figure 5.8.c). Here, we select the fact that we want to specify as a condition.

This example illustrates a current problem depicted in figure 5.8.c). We find two different character names from at least two different movies. One character name lives in *The Hobbit: An Unexpected Journey (2012)*. The other character name lives in each of the movies of the *Lord of the Ring* trilogy. But, all of those movies are from the same universe and therefore ideally the character names for the same role in all movies should be named equal. Chapter 8 will regard this in more detail and offer possibilities to tackle such problems.

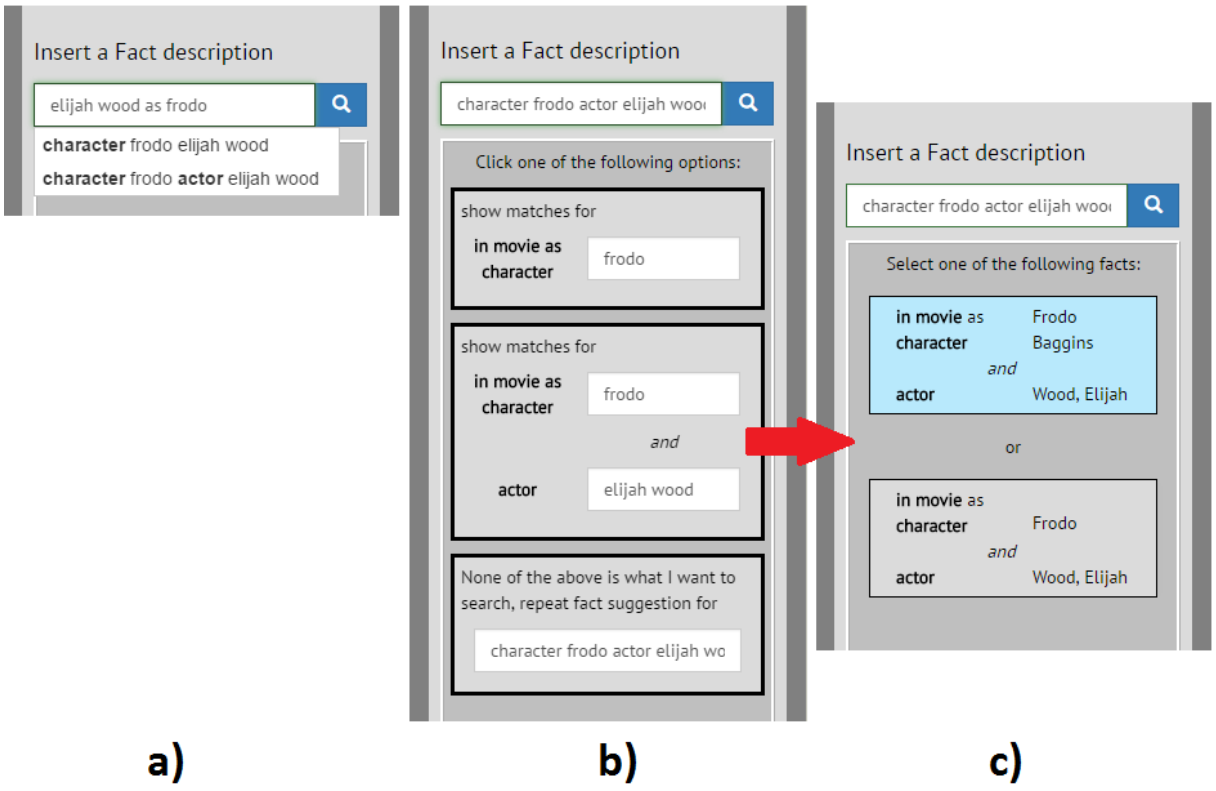

Figure 5.8: Suggestions for the 3-ary use case: *Find movies where Frodo was played by Elijah Wood.*

5.2.3 Facts and Plot

The complete process of building a semantic query consists of facts and plot.

An example use case that utilizes both facts and plot is:
Find an action movie with Arnold Schwarzenegger where he fights with a sword.

This use case illustrates the scenario where a user knows some things about a movie for which title he is looking for. Some parts of the request will be best used as a plot description while other parts define facts.

Figure 5.9 displays the suggestions for the three conditions in this use case. Figure 5.9.a) shows, that we search for a movie with the inputted plot description. The other two parts show how we select the genre relation with our given input in figure 5.9.b) and the actor we demand in figure 5.9.c). For the actor we use the binary subrelation of a cast that connects a movie with an actor. We only have an actor given so there is no need to use a full cast relation. Figure 5.1 on page 44 contains the resulting current query for this use case.

The three use cases we presented are also part of the user study. In the evaluation in chapter 7 we will see if users can build the expected semantic queries or at least expected results.

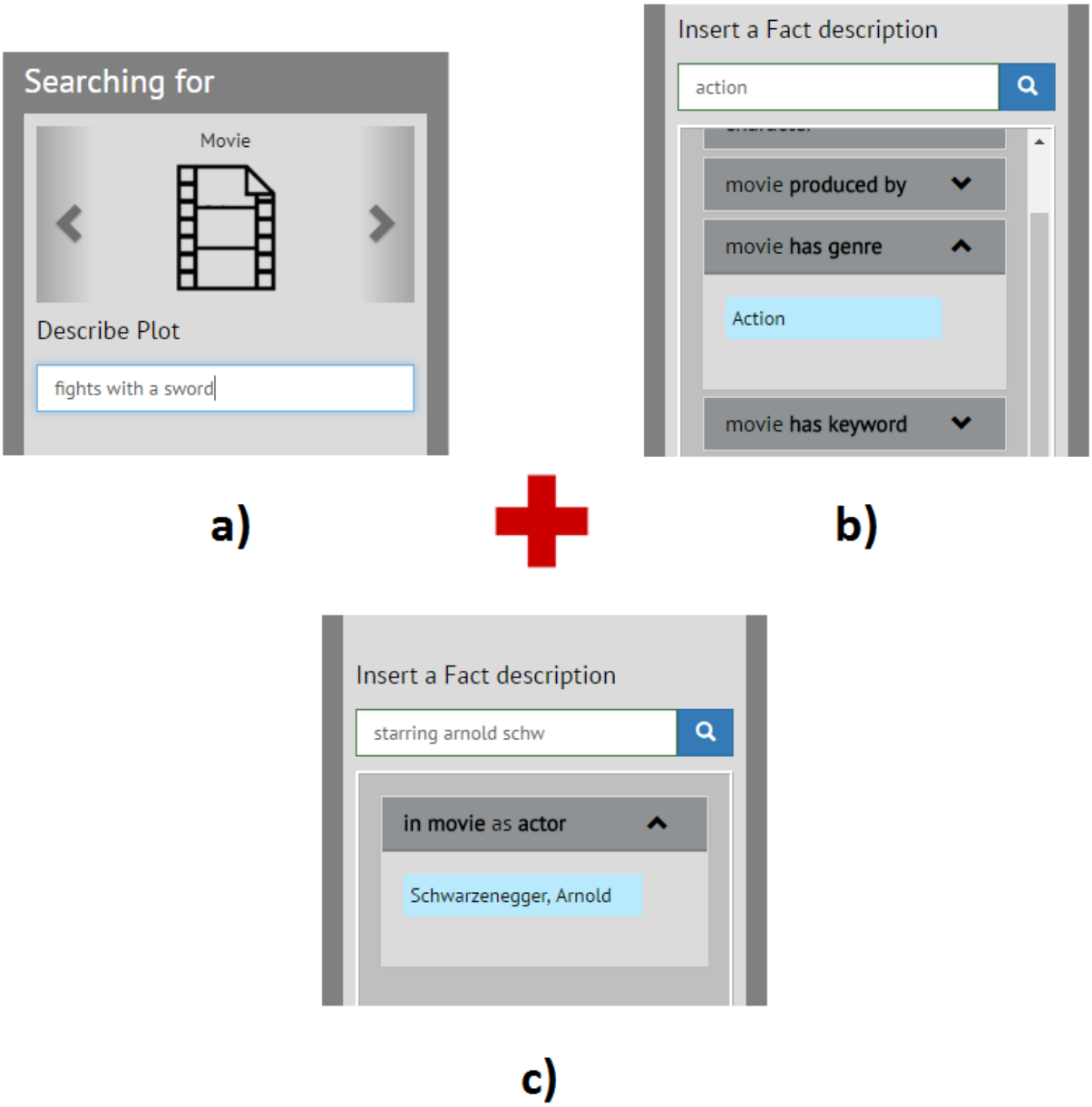

Figure 5.9: Combined suggestions for all conditions from the use case: *Find an action movie with Arnold Schwarzenegger where he fights with a sword.*

We summarized the parameters of MovieSearch and regarded it in detail. The adding of condition after condition describes the stepwise semantic query building. The suggestions and appropriate selections make the query building interactive. We will now overview a comparison of the interface of MovieSearch and some other interfaces that apply to similar use cases.

5.3 MovieSearch user interface in context

We begin with an overview of the predecessor Broccoli [**?**].

5.3.1 Broccoli

MovieSearch and Broccoli have a similar placement of the general components. Those are: on the left side is the condition specification and in the middle is the current query with its results underneath.

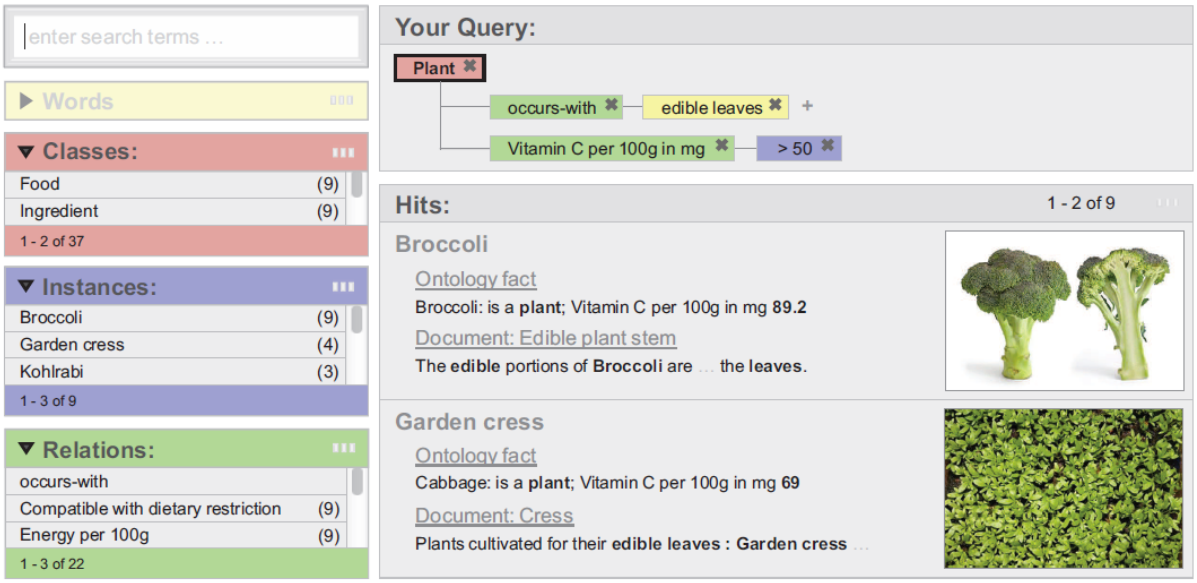

Figure 5.10: Origin web client – Broccoli [**?**].

The major differences are how the queries are build and the splitting of conditions into plot and facts with appropriate input sections at MovieSearch. MovieSearch aims to enrich the usability by the suggestions. Furthermore, MovieSearch adds the support of 3-ary relations to the features.

Next, we regard the user interface for detailed queries about facts that specify a movie title of the website where our main underlying data comes from.

5.3.2 IMDB advanced search

> *"Welcome to IMDb's most powerful title search. Using the options below you can combine a variety of the types of information we catalog to create extremely specific searches. Want Canadian horror movies of the 1970s that at least 100 IMDb users have given an average rating above a 6? You can find them here."* (see [**?** , introduction text])

This website aims at similar use cases: the specification of facts for finding movies. Regarding usability through the tool from [**?**] is a deterrent example as illustrated by figure 5.11. Here, basically every possible relation in the data has its own input field hard-coded. Our use cases can not all be answered with the IMDB advanced search. The main reason for this is that in the IMDB advanced search you only can specify one actor or crew member per query. Hence, all use cases that use a relation at most once can be answered. However, this is a very tedious process.

Figure 5.11: IMDB advanced search – User interface.

This advanced search illustrates the problem of the ”good” amount of input fields.

5.3.2.1 Amount of input fields

The problem we want to discuss here is the ”good” amount of input fields. How many input fields balance feature with usability in a desired way. Most likely, there is no one good number for all applications. However, the IMDB advanced search illustrates that there can be too many input fields for a smooth usage. Too many input fields make it harder to find what you can input and where you can input the facts you want to specify. Especially inexperienced users will have a hard time orienting and figuring out their options. In section 2.4.5 we regard a faceted search [**?**] with the same problems of too many input fields for our goal.

MovieSearch tries an approach with two input fields. One for plots and one for facts because those are two categories we want to treat differently. However, we therefore rely on the user to implicitly split his conditions into plot and facts correctly.

A different approach that uses only one input field could be natural language based like the one we regard in section 2.4.2. Handling text input like the plot parts with natural-language-based approaches can be hard because the tool has to identify the text as a text part without parsing its content. This is the reason why we preferred the approach from MovieSearch. We will see in the evaluation in chapter 7 how well both approaches perform on the use cases. The one input field and the hidden automatic query building compete there against our two-input-field approach and iterative conditions adding from MovieSearch.

Since we identified the natural-language-based approach as a competitor in realizing requests in user-friendly manner, we picked such a tool to compare it with our results. The next section will introduce the competitor for MovieSearch.

5.3.3 Competitor – Valossa Video Search

From a couple of search engines for movies in the web, we selected the one with the best results on queries that are similar to our use cases. This results in our competitor Valossa Video Search [**?**]. Valossa offers a descriptive movie search engine at the URL `whatismymovie.com` that is depicted in figure 5.12. Their data set contains 40 000 movie titles with English movies from 1900 to 2016. Valossa is a offspring company of the University of Oulu. Their search engine is based on multimedia retrieval research at the University of Oulu.

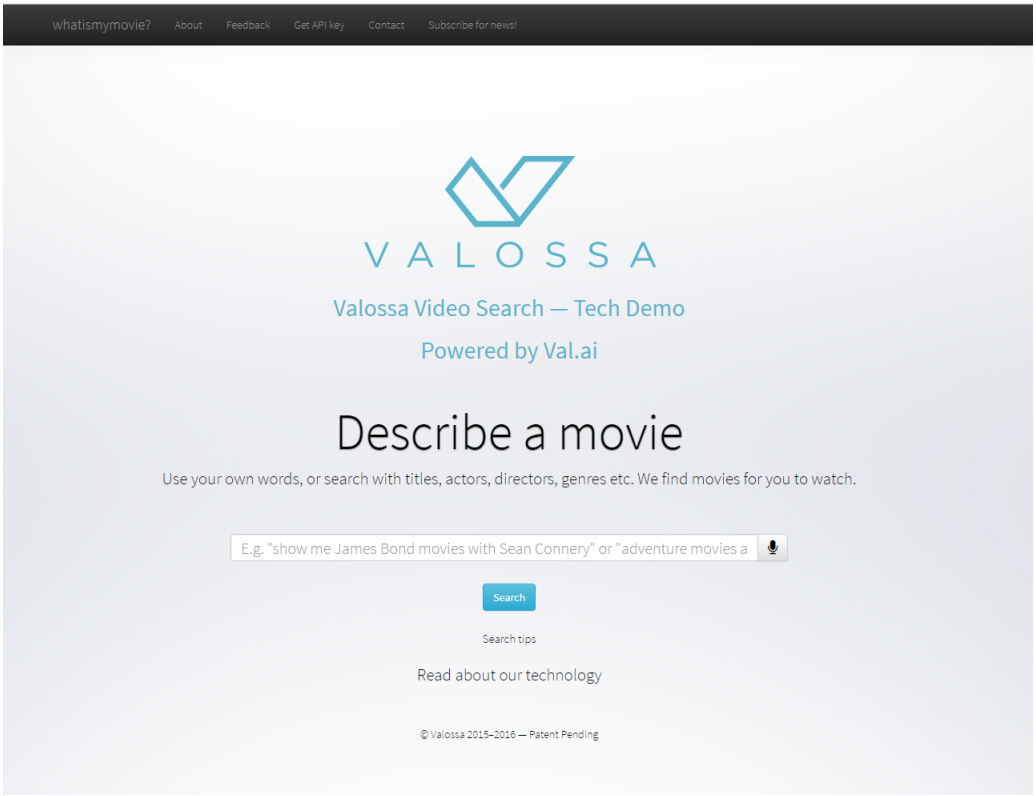

Figure 5.12: `whatismymovie.com` – Natural-language-based user interface by Valossa.

The Valossa website has basically two elements. The input field for the full query and the search button. In contrast, MovieSearch has more elements. The query building at Valossa happens internally without interaction to the user except the input. For performing the condition adding at MovieSearch and other tasks a couple of user interface elements are inbuilt.

Next, we will regard the global architecture of MovieSearch with all the different components we encountered so far. On the one hand we summarize the different parts and on the other hand we visualize the relationships of the parts that build MovieSearch.

Chapter 6

Architecture

This chapter combines the different components we so far encountered to the whole system. Figure 6.1 depicts an overview of the main parts and how they interact.

The data we prepared in chapter 3 is used at three places:

- First as the RDF graph of the Fact Suggestion Server.
- Second as the RDF triples with documents as a base for the SPARQL Server.
- Third by the Python Poster Server which manages the posters. (While retrieving the topactors from the TMDB API at [**?**], we also store the occurring posters. These posters are the starting pool for our Poster Server.)

The Fact Suggestions from chapter 4 are performed by the Fact Suggestion Server which is the part at the bottom left of figure 6.1 marked by the green background. The various types of suggestions all apply for a given fact description input.

The User Interface with its two input categories plot and facts is marked by the blue background. We present the UI in chapter 5. The architecture overview illustrates how the current query consists of conditions that are facts and plot.

The final results for the current query are displayed with the help of the SPARQL Server by [**?**]. We find the movies (or persons) that are specified by the current query. Furthermore, per searchtype we have a fix set of bonus information relations that are loaded. For example for *movie* results we will try to load amongst others the director, the running time of the movie, its IMDB rating, et cetera. It can happen that a result movie (or person) does not have all of the bonus relations set with a value. If this is the case we just exclude this bonus information from the result display.

The Python Poster Server loads posters for the results. The first time a poster is requested we retrieve it from the TMDB API or if there nothing is found with a google custom search by [**?**]. After we found a poster for a result once, we store it on our server. Hence, the second time this poster will be requested the poster answer will be fast. Our poster pool dynamically gets more complete this way.

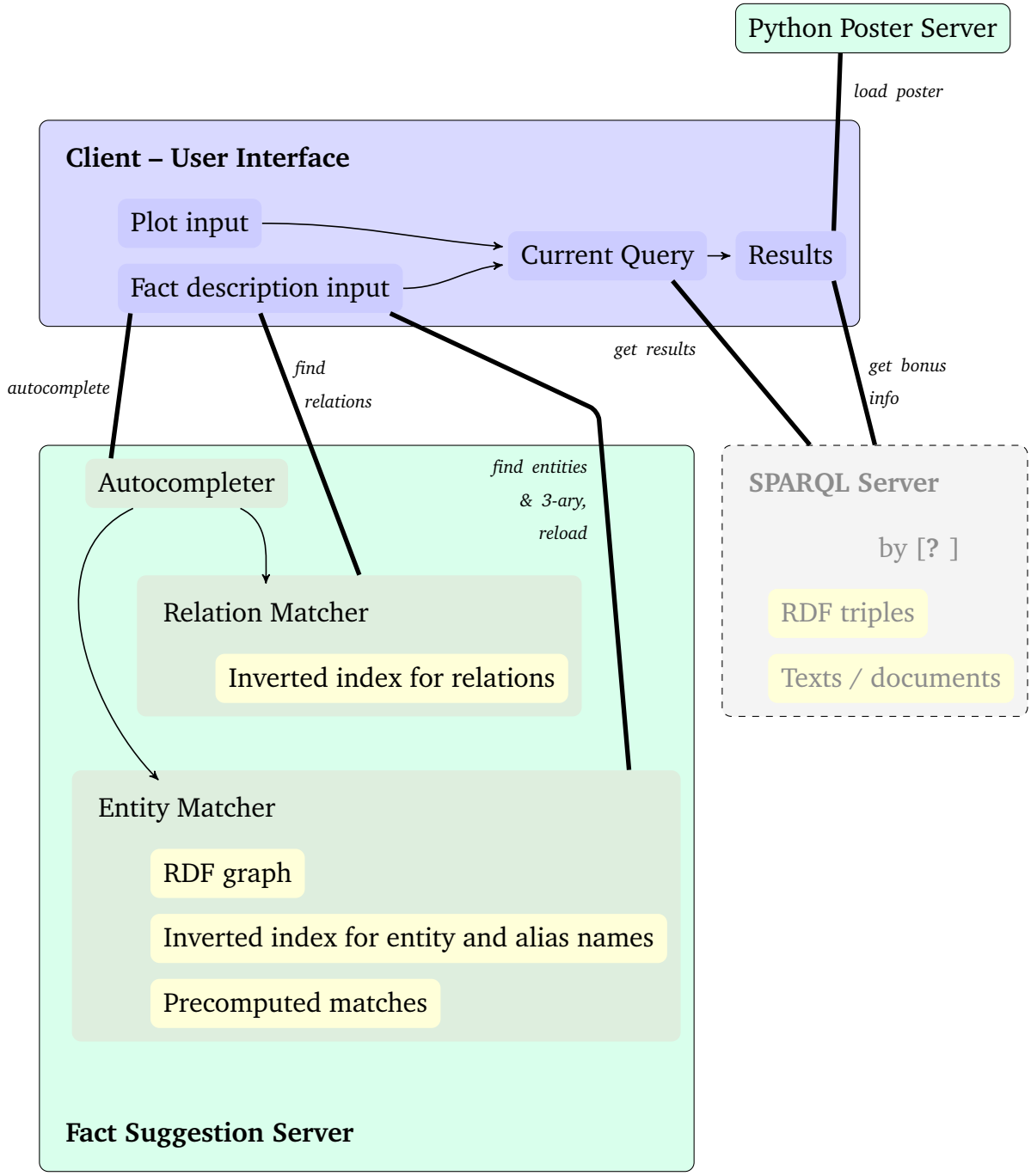

Figure 6.1: Main architecture of MovieSearch.

The next chapter is the evaluation of MovieSearch. We evaluate the system MovieSearch from a usability perspective. Hence, we are interested in the response times of the Suggestions Server. Furthermore, how good are the results and their ranking which are found by MovieSearch? Are they better than the results of the representative of the natural-language-based approach? Is the query building effective? Does the query building lead to satisfying built queries and to relevant results?

Chapter 7

Evaluation

The topic of this chapter is the evaluation of MovieSearch. Our main perspective is usability. We will see how well the separate components perform their task. The Suggestion Server is tested for acceptable response times as a requirement for a smooth query building. Afterwards, MovieSearch is compared with the natural-language-based competitor Valossa. Here, we check the relevance of the results and the quality of the order of the results. The last segment discusses the user study. The user study measures the quality of the query building process.

7.1 Efficiency Tests of Fact Suggestion Server

We want to ensure efficient query building. For the user side the most vital part is a short response time for the suggestions to make the query building interactive. User-friendliness is improved by fast answers to conciliate the users with their natural impatience. The part of the response time we influence the most is the processing time of the request at the server. At this point we focus on the time component of the methods because we regard the system from a usability perspective. A proper space consumption analysis is left open for future work.

For testing if the Fact Suggestion system that is used by the server is efficient, we will regard the processing times for the three most used methods of the Suggestion Server. The critical methods are finding matches and gathering graph information for both entities and for 3-ary relations.

Matching given request words to lists of candidate entities is performed by almost every call to the server. Therefore, the first method we consider is the retrieving of matching entity IDs. But before, we will introduce the setup of our experiments.

7.1.1 Experimental Setup

The experiments for the algorithm performance tests were run on an
Intel Xeon E5640 @2.67GHz machine with 16 CPUs and 65 GB main memory. The operating system is Ubuntu 14.04.

Table 7.1 shows the involved RDF graph with its statistics. The graph is built only with *word* based entities. This graph is the same which is used by the Suggestion Server in full activity. The graph contains all things we specified in section 3.5 for example shortcuts or the integrated TMDB casts. The number-based entities that are ignored are especially not relevant for the methods we want to evaluate here. Because, the name matching and gathering only apply for words as values and not for numbers as values.

Amount of entities	29 531 552
thereof Movies	11.1%
thereof Persons	21.5%
Amount of aliases	1 519 666
entities with aliases	2.9%
Amount of relations	28
Amount of arcs	278 967 994
avg. out/in-arcs per entity	9.4

Table 7.1: RDF graph statistics for the graph that is used for the performance tests of matching and gathering.

Procedure of testing For matching words to entities we need a pool of words. From this pool we will select uniformly random words that will serve as an input for the entity matcher. The gathering information tests will afterwards run on the matched entities.

We regard two different pools of words. The first pool is all words that occur in an entity name, in an alias or in any *word*-based value. Especially for the 3-ary relation gathering test with casts, we want another pool of only person-name words and appropriate aliases. This way we increase the chance of finding matches, because we already need to find two names that belong to the same cast which is unlikely enough with only names.

For gathering 3-ary we have additionally a couple of predefined cast pairs of actor with character that exist in the graph. This way we at least find some successful gather requests if we do not find any with the random name-word selection. Each predefined cast consists of an actor with two words and an character with two words, for example actor *Liam Neeson* and character *Bill Marks*. We than try to find matches and gather information for all combinations of its subsets. As a couple examples we would test for the given cast example: actor *Liam* as character *Marks*, or actor *Neeson* as character *Bill Marks*.

Test variations For testing the computation times we regard different prefix sizes of the inverted index for the entity matches and one alternative filtering method.

The filtering method of contains from start is compared to a fuzzier approach with the Levenshtein edit distance [**?** , p. 199]. For this edit distance the transformation from one string to another string counts the minimum needed amount of operations. Allowed operations for the Levenshtein edit distance that alter one letter are *replace*, *insert* and *delete*. For an example see the edit distance of ”flat” and ”lite” in figure 7.1.

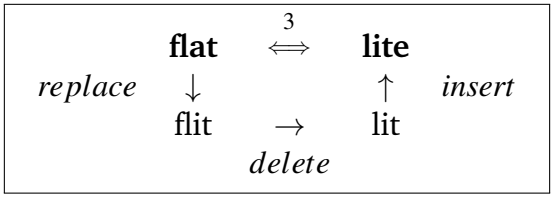

Figure 7.1: Example for Levenshtein edit distance and its operations.

While using edit distance we filter out words that have an edit distance greater than two. This way we tolerate typos that for example swap two letters or forgot a letter.

All following sections are in the scenario where we have given a searchtype and a list of request words. The first method we look at, is finding matching entities. This is the most basic method of those we evaluate, because it is called by many different server components.

7.1.2 Entity Matcher – finding matching entities

Finding matching entities depends on how many search parts are given. The search parts are the words that we try to match. The search parts correlate with a list of request words. Here, we find entities that match the request like we discuss in section 4.2.

We will compare different prefix sizes and two different filter methods.

7.1.2.1 Measurements

C_i represents a test with **C**ontains from start and ED_i with **E**dit **D**istance smaller or equal than two. Where i is the prefix size that is used. Here, the Levenshtein edit distance [**?** , p. 199] for testing a fuzzier string match version is used. For the C_i we performed 10,000 runs of matching and gathering and for the ED_1 we performed 1,000 tests.

Originally we intended testing for different amount of search parts but for at least two random parts there were no results left after filtering. Than with an empty entity match list the gathering will return nothing, too. Returning nothing for an empty input will always be fast, hence we will not consider those experiments.

Next, we take a look at how the prefix matching performs with different search parts amounts in table 7.2. This is basically the list intersection of the inverted indexes. For the other parts of the test we are going to restrict ourselves to one randomly selected search part.

Search-parts	avg. **matching prefixes** in [ms/calls]			
	C_3	C_6	C_9	ED_1
1	1.931	0.169	0.044	10.712
2	3.374	0.217	0.039	–
3	4.422	0.241	0.055	–

Table 7.2: Matching prefixes, average computation times.

The space needed for the full Suggestion Server in main memory which contains the RDF graph, precomputed results, inverted indexes for entities and relations is summarized in table 7.3.

Test	virtual memory		reserved memory	
C_3	45.2 GB		33.2 GB	
	↓	+18%	↓	+25%
C_6	53.6 GB		41.5 GB	
	↓	+12%	↓	+16%
C_9	60.4 GB		48.2 GB	

Table 7.3: Space consumption of the Fact Suggestion Server.

Now, we only regard results for one randomly selected search part to have enough results for the complete matching of entities. For the complete matching we first find matching prefix candidates and second filter those by the given search part and a filter method that compares the candidate with the search part. The results in table 7.4 are grouped by the two different pools of words for the random selection from the introduction.

Pool	Test	avg. matching prefixes	avg. filtering	**total avg.**	
words	C_3	1.9	382.6	**384.5**	ms/calls
	C_6	0.2	21.7	**21.9**	ms/calls
	C_9	0.04	4.3	**4.34**	ms/calls
	ED_1	10.7	15356.3	**15367.0**	ms/calls
names	C_3	0.7	91.7	**92.4**	ms/calls
	C_6	0.02	1.4	**1.42**	ms/calls
	C_9	0.01	0.18	**0.19**	ms/calls
	ED_1	9.0	11001.9	**11010.9**	ms/calls

Table 7.4: Finding matching entity IDs, average computation times.

7.1.2.2 Interpretation

The absolute time needed for matching the entities from table 7.4 for the C_i are small enough that they should not attract much attention of the user. Especially for name-based words which will probably be a majority of the inputs.

The prefix matches computation time gets a magnitude faster for each 3 further length in prefix while increasing memory consumption by 12-25%. The filtering profits from the prefix length increase, too. Because a finer granulation by prefixes results in smaller intersections that we have to filter. In the same way, the overall matching gets faster by a magnitude per prefix length increase.

Otherwise, we see that it does not work to just swap out the filter method for fuzzier string matches. Prefix match does not help much with only length 1 for the ED_1. How-

ever, the very first letter is very unlikely for a typo, but each letter afterwards could be swapped. Hence, we want a prefix size of one for a good typo detection with edit distance. But with this, we get few and long prefix lists which explains the long prefix match duration. The result list will be larger and we use a filter method that is costlier than for C_i so the new filtering time gets way increased. The measured times for ED_1 show clearly that it is not a usable alternative for our scenario. This indicates for supporting a more flexible fuzzy search than contains from start we have to change something for the prefix lists, too. The pre-selection before filtering has to be altered for using edit distance. We could utilize different k-grams as inverted index keys. But, this is open for future work.

The entity matching is a method that does not depend much on our fact data structure. Therefore, we can consider an alternative and compare how they perform. For the gathering methods this would require an alternative data structure to gain meaningful results. Hence, a proper evaluation of different data structures and appropriate gathering methods is a bigger future project. But for the entity matching, we now regard an alternative approach that uses permutations and binary search of the words.

7.1.2.3 Comparison to an alternative entity matcher – Binary search in sorted permutations *BiSo*

As a baseline we built another Matcher that finds matching entities for a given set of request words. Here, we exchange the inverted index for a sorted list of word permutations. Using permutations is a common approach in information retrieval. For each name of an entity or an alias we split the name into all its words by the separator we specify. After the split, we pull each word from the split in front and concatenate this as the next permutation. This way each word from this name will be in the beginning once, with the rest of the name afterwards. As an example we regard the entity name *Sir Sean Connery*. From the example we create three permutations and sort them. Hence, the results are:

connery sir sean
sean sir connery
sir sean connery

Now we have a given request set of words $\{w_i\}$. For the first word w_1 we find a position where the permutation contains w_1 as a beginning. From there we find boundaries in both directions while the neighbors also contain w_1 as a beginning. For the resulting boundaries we perform a filtering for the other request words $\{w_j | j > 1\}$. If the permutation has all the request words as a beginning of a word part, we add the corresponding entity ID to the match results.

The filtering afterwards is similar to the method in our main Matcher. The only difference in the implementation is for this alternative we have to sort the remaining entity IDs after filtering to have the same properties of the results like with the main Matcher and its filtering. With the inverted indexes the sorted IDs are built in. Note, the given

candidates list do not have to be the same or even of similar length. It can be especially the case that we have duplicates in the candidates of this alternative version. This illustrates that with this alternative approach the candidates do not have as much useful properties as with an inverted index.

Now let us discuss the computation time results in comparison to the C_i in table 7.5.

Pool	Test	avg. matching prefixes	avg. filtering	**total avg.**	
words	C_3	1.9	382.6	**384.5**	ms/calls
	C_6	0.2	21.7	**21.9**	ms/calls
	C_9	0.04	4.3	**4.34**	ms/calls
		avg. binary search	avg. filtering		
	BiSo	0.04	0.3	**0.34**	ms/calls
names	C_3	0.7	91.7	**92.4**	ms/calls
	C_6	0.02	1.4	**1.42**	ms/calls
	C_9	0.01	0.18	**0.19**	ms/calls
		avg. binary search	avg. filtering		
	BiSo	0.03	0.22	**0.25**	ms/calls

Table 7.5: Finding matching entity IDs, average computation times – Comparison to *BiSo*, an alternative with binary boundaries search in a permutation list.

We observe, that only C_9 is on a par with *BiSo*. However, our experiment setup is skewed towards *BiSo*. When will the permutations of *BiSo* be fast? Especially, in the case of only one search part, because for this case we basically do one binary search. A more interesting comparison would be for two or three search parts. Only for more search parts the inverted index helps with the intersections. However, for a better comparison we have to handle the problem with the randomly selected words to find matches for. Currently we only regard one search part, because for more randomly selected words in almost all cases there are no matches left after filtering. This makes the results for more than one search part not very meaningful.

Another point to clarify is, how many search parts do we expect? Depending on that we will be able to make a better decision. For this we could measure the amount of inputs per fact description over a period of running time of MovieSearch. This way we could state a reasoned demand. However, the average amount of parts for all words in the fact graph is 3.5. If we only consider person names, they have an average of 2.4 words per name. The expected amount of search parts will probably be in $\{1,2,3\}$.

Summing up, *BiSo* seems to be a very fast approach. However, a meaningful experiment with two and three search parts has to be performed in future. The comparison of *BiSo* and C_i together with a determined expectation for an amount of search parts will lead to a reasoned decision which version is more suited. Another aspect is the memory consumption. The average amount of parts of an entity name in the graph is 3.5 which implies the additional memory consumption for the permutations. Hence, we have to store 3.5 times of the entity and alias name amount as the permutation strings. This has to be properly compared to the amount that is needed by the inverted index lists

accordingly. Especially the space consumption compare with C_9 will be interesting, because C_9 is on par with *BiSo* concerning the process time. Lesser space demands would be an argument for C_9 and the inverted index.

After finding matching entities, we want to find information that belongs to them which we can use as suggestions. The matched entities are internally stored by their entity IDs. The next section regards how fast we perform the information gathering for found entities.

7.1.3 Entity Matcher – gather info for matching entities

After finding entities that match the request we now want to gather corresponding graph information like we discuss in section 4.2.

The gathering only depends on the entity IDs. Hence, it is not really interesting how many search parts were given. Surely the search parts number implicitly influences the amount of matching entity IDs. But this is negligible, because more search parts will only further constrain the amount of possible hits by more conditions. Only one search part will have therefore the most possible matches and it suffices to consider this case.

7.1.3.1 Measurements

Gathering all the information that is needed for the suggestions we regard in section 4.2 is measured in the following table 7.6.

Pool	Test	**avg. gathering**	
words	C_3	0.546	ms/calls
	C_6	0.588	ms/calls
	C_9	0.511	ms/calls
	ED_1	23.293	ms/calls
names	C_3	0.360	ms/calls
	C_6	0.280	ms/calls
	C_9	0.196	ms/calls
	ED_1	20.499	ms/calls

Table 7.6: Gathering info for matching entity IDs, average computation times.

7.1.3.2 Interpretation

All the measured gathering times are acceptable since they are small enough that an user should not notice them in the client. All C_i times are similar because the gathering depends on the matching IDs amount and the graph structure. For all tests the graph structure is the same, but for ED_1 the matching IDs amount is larger since its matching method is fuzzier and allows way more differences.

The last method we evaluate is the matching of 3-ary relations which is one of the important components of MovieSearch. Similar to the gathering of information for the entities we have two sets of matching entities given. For 3-ary relations we now have to find connecting links that form a fact between the two entity sets and the given searchtype.

7.1.4 Entity Matcher – gathering information for 3-ary relations using the example of casts

For the suggestions of 3-ary relations the scenario requires more parameters than searchtype and given search parts. We also need two candidate relations that belong to a 3-ary relation.

We will test this for the special case of casts. We only allow words that occur in any person name or corresponding aliases by using the name pool for the selection of the one search part per relation. Also, we search for the cast while searching for a movie, which sets the two relations to be actor and character.

We will than measure the time in both combinations for the check if there exists a cast relation for a movie that matches those two words as actor and character. More words per relation make it very unlikely to find something for random words.

We observed very few runs that actually found results, hence we add some predefined cast tuples to the random process. This is discussed in section 7.1.1 in more detail.

7.1.4.1 Measurements

The times for gathering in table 7.7 are a bit misleading since there were so few calls that actually found a cast. A call that found a result is called successful. The share of successful calls is depicted amongst others in table 7.8.

Pool	Test	**avg. 3-ary gathering**	
names	C_3	0.140	ms/calls
	C_6	0.158	ms/calls
	C_9	0.107	ms/calls
	ED_1	18.497	ms/calls

Table 7.7: Gathering 3-ary relations for a given cast information, average computation times.

For this reason we will regard the gathering times of calls that actually found a cast in a separate table 7.8.

Pool	Test	**avg. 3-ary gathering**		successful calls	total calls
names	C_3	6.881	ms/calls	77	10000
	C_6	7.418	ms/calls	74	10000
	C_9	7.466	ms/calls	77	10000
	ED_1	68.719	ms/calls	336	1000

Table 7.8: Gathering 3-ary relations for a given cast information, averages only of successful calls.

7.1.4.2 Interpretation

We store 3-ary relations in the graph differently than for example in [**?** , section 2.3.1]. But, the gathering and therefore access times to those facts show, that they are accessible with an acceptable response time.

Again ED_1 finds more matches and therefore needs more time because its matching method is fuzzier than contains from start.

7.1.5 Summary

First we regard the C_i cases because for those the system is designed for. In total a response will combine some of the methods from above. An Entity Suggestion call will consist of matching entities and gathering info. Since the gathering is so cheap and the matching is tolerable for the most test variations, their combination sums up to an acceptable response time.

For 3-ary Suggestions we have to perform two matching entities and afterwards gather the 3-ary information. For C_3 the required two matching entities calls can need some time. For example for two words from the bigger word pool the sum of the separate matching procedures can be around 800 ms in average which already gets quite long and maybe noticeable. Remember we use the suggestion methods often. Sometimes even implicitly in the autocompletion. Hence, having them to be fast is vital. For the C_6, C_9 though the combination is harmless. Good enough response times could already be achieved by a prefix size of 4 or 5 if the space consumption increase forms a problem. We could test this in a finer grade in future. However, even for C_3 the space consumption of the Fact Suggestion Server is after all the half of the main memory of the given server machine. Having the whole *word*-based RDF graph, the inverted indexes and the precomputed results in main memory needs a sizeable amount of space. This is a drawback of the current Suggestion Server which we will address in the future work section 8.2 again.

Now for ED_1 we can sum up that its test variation does result in unacceptable response times and should not be used in the current setup that aims for an user-friendly environment.

The next section will introduce the queries that will serve as use cases for the following user interface analysis.

7.2 Use cases – Queries for the User Interface analysis

This section introduces the queries that are used for the usability tests and for the search quality comparison. Table 7.9 contains the text of the queries that was given to the participants of the user tests. The same texts were inputted into the search website of the competitor Valossa.

Type	Query	Task Description
E	Q_1	Movies from Christopher Nolan.
E	Q_2	Movies with songs from Hans Zimmer.
E	Q_3	Movies made by Jerry Bruckheimer.
E	Q_4	Funny Movies with action.
E	Q_5	Movie that is 111 minutes long.
E	Q_6	Movies released at 11.11.2011.
E	Q_7	Movie that is 111 minutes long and released at 11.11.2011.
E	Q_8	Movie by Martin Scorsese and stare role Leo DiCaprio.
3	Q_9	Movies where James Bond was played by Pierce Brosnan.
3	Q_{10}	Movies where Frodo was played by Elijah Wood.
E	Q_{11}	In which movies directed by Garry Marshall was Julia Roberts starring?
3	Q_{12}	Movie by Tim Burton with Johnny Depp as Todd.
E	Q_{13}	Movie with a (IMDB) rating worse than 2.0 .
E,3	Q_{14}	Movie directed by Ridley Scott from the year 2000 with a rating better than 8,0 by more than 80,000 users.
P&F	Q_{15}	Drama Movie about a car accident in Los Angeles directed by Paul Haggis.
P&F	Q_{16}	Movie where Di Caprio uses cocaine.
P&F	Q_{17}	Action Movie with Arnold Schwarzenegger where he fights with a sword.
P&F	Q_{18}	Movies where Keira Knightley sails on the Black Pearl.
P&F	Q_{19}	Movie from 2014 where Liam Neeson plays an air marshal on a plane.
P&F	Q_{20}	Movie with Angelina Jolie and Brad Pitt where they have secrets.
P&F	Q_{21}	Movie about lying and forgery in which the protagonist is sent to Rome (it is a new version of the movie Plein soleil 1960).

Table 7.9: Use case queries.

The type of the first column in table 7.9 hints to the intention of the appropriate query. Here, E represents queries that require exploring of the data. For example if using vague descriptions of the relations or alternative relation names can be overcome to find the results. By avoiding the relation names we use in the graph we will see if the suggestions can help the users to find the required relations. The type 3 marks queries that involve a 3-ary relation possibility. The last type P&F indicates that for this query the given conditions have to be split into plot and facts.

The next topic is comparing the result quality and ranking quality of the graph-based approach with the natural-language-based approach of query building.

7.3 Quality of results and ranking: MovieSearch vs. Valossa

We will regard the top-10 results for the two search engines MovieSearch and Valossa for each query. For comparing we use the measures Recall, Precision, F measure and Discounted Cumulative Gain (*DCG*). With these common informational retrieval methods we will analyze the quality of the query results. We now introduce these methods.
For a query Q_i there is $S \subset \{\text{movies}\}$ which are all the movies that are a solution for the query. May the request answer be $A \subset \{\text{movies}\}$. Note, since we only regard the top-10 results we know $|A| \leq 10$. With this circumstances the measures are defined by the following terms. Recall tells us how many of the possible solutions were found with the request answer:

$$\text{Recall } R := \frac{|S \cap A|}{|S|}.$$

Precision defines how many result movies of the answer are hits:

$$\text{Precision } P := \frac{|S \cap A|}{|A|}.$$

The F measure combines the previous two with the harmonic mean to a single number:

$$\text{F measure } F := \frac{2RP}{R+P}.$$

For taking the order of the results in the answer into account, we consider a measure that values hits higher that are near the start of the answer. The corresponding measure is DCG_{10} since we only regard the top-10 results:

$$DCG_{10} := w_1 + \sum_{i=2}^{10} \frac{w_i}{\log_2 i}.$$

Here, $w_i \in \{0,1\}$ is a weight per answer result that defines if it is relevant for the current query or not. For this our answers correspond to a sequence $(m_1,...,m_{10})$ with $m_i \in \{\text{movies}\}$ where we set $w_j = 1 \Leftrightarrow m_j \in S$ for the current query.

We relate the definitions of the measures with the result tables of MovieSearch 7.10 and Valossa 7.11 by the following. The columns have the meaning: found Hits $= |S \cap A|$, answers $= |A|$. Furthermore, for a better comparability of DCG_{10} in between the queries we normalize it by the DCG_{10} of the best possible answer per query. Therefore we use for a given answer sequence a and the best answer sequence top:

$$NDCG_{10}(a) := \frac{DCG_{10}(a)}{DCG_{10}(top)}.$$

Now, we compare these measures of the results from MovieSearch and Valossa.

7.3.1 Measurements

Our expected SPARQL query for each of the Q_i produces the results which are used for the relevance analysis in table 7.10. The overall quality of the results is displayed in the F measure while the quality of the order is part of the $NDCG_{10}$.

Query	found Hits	answers	Recall	Precision	F measure	$NDCG_{10}$
Q_1	10	10	83,33%	100,00%	90,91%	100,00%
Q_2	10	10	4,22%	100,00%	8,10%	100,00%
Q_3	10	10	0,55%	100,00%	1,09%	100,00%
Q_4	10	10	0,10%	100,00%	0,20%	100,00%
Q_5	10	10	0,58%	100,00%	1,16%	100,00%
Q_6	10	10	1,03%	100,00%	2,04%	100,00%
Q_7	1	1	100,00%	100,00%	100,00%	100,00%
Q_8	7	8	100,00%	87,50%	93,33%	98,76%
Q_9	4	10	100,00%	40,00%	57,14%	77,22%
Q_{10}	3	3	75,00%	100,00%	85,71%	84,03%
Q_{11}	4	4	100,00%	100,00%	100,00%	100,00%
Q_{12}	1	1	100,00%	100,00%	100,00%	100,00%
Q_{13}	10	10	0,42%	100,00%	0,83%	100,00%
Q_{14}	1	1	100,00%	100,00%	100,00%	100,00%
Q_{15}	1	1	100,00%	100,00%	100,00%	100,00%
Q_{16}	2	3	66,67%	66,67%	66,67%	76,02%
Q_{17}	2	2	66,67%	100,00%	80,00%	76,02%
Q_{18}	3	3	100,00%	100,00%	100,00%	100,00%
Q_{19}	1	1	100,00%	100,00%	100,00%	100,00%
Q_{20}	1	1	100,00%	100,00%	100,00%	100,00%
Q_{21}	1	1	100,00%	100,00%	100,00%	100,00%
Avg.			**66,60%**	**94,96%**	**66,06%**	**95,81%**

Table 7.10: MovieSearch quality of results and ranking.

The relevance of the answers was defined by how the result fulfilled the given properties of the task Q_i. The sources to determine if a movie applies to the properties was the

IMDB website and the downloaded data. We call a movie m relevant for a query Q_i if $m \in S_i$. Per query Q_i the total amount of movie matches $|S_i|$ and sets of solution matches S_i can be regarded in detail in the appendix tables A.1 and A.2. For some solution sets we are stricter than movies which fulfill the properties. An example for this is the solution set for query Q_9 for which we only accept the actual Bond movies as hits and not any documentary which has Pierce Brosnan appear as Bond.

The appropriate measurements of the results from the Valossa website are summarized in table 7.11. For issuing the query we inserted the exact query description into the one natural language input field. The averages are only formed for values which are defined. Q_{14} did not lead to any results.

Query	found Hits	answers	Recall	Precision	F measure	$NDCG_{10}$
Q_1	10	10	83,33%	100,00%	90,91%	100,00%
Q_2	8	10	3,38%	80,00%	6,48%	74,96%
Q_3	10	10	0,55%	100,00%	1,09%	100,00%
Q_4	10	10	0,10%	100,00%	0,20%	100,00%
Q_5	6	10	0,35%	60,00%	0,70%	71,05%
Q_6	3	10	0,31%	30,00%	0,61%	21,59%
Q_7	0	10	0,00%	0,00%	–	0,00%
Q_8	1	10	14,29%	10,00%	11,76%	8,99%
Q_9	4	10	100,00%	40,00%	57,14%	96,39%
Q_{10}	2	10	50,00%	20,00%	28,57%	36,12%
Q_{11}	4	10	100,00%	40,00%	57,14%	100,00%
Q_{12}	1	10	100,00%	10,00%	18,18%	100,00%
Q_{13}	0	10	0,00%	0,00%	–	0,00%
Q_{14}	–	–	–	–	–	–
Q_{15}	1	10	100,00%	10,00%	18,18%	100,00%
Q_{16}	3	10	100,00%	30,00%	46,15%	90,72%
Q_{17}	3	10	100,00%	30,00%	46,15%	43,99%
Q_{18}	3	10	100,00%	30,00%	46,15%	100,00%
Q_{19}	1	10	100,00%	10,00%	18,18%	100,00%
Q_{20}	0	10	0,00%	0,00%	–	0,00%
Q_{21}	0	10	0,00%	0,00%	–	0,00%
Avg.			**47,62%**	**35,00%**	**22,38%**	**62,19%**

Table 7.11: Valossa quality of results and ranking.

The detailed result lists for both MovieSearch and Valossa can be regarded in section A.2. For each query all the returned movies are stated and marked accordingly to their relevance. Let us continue with the interpretation of the measurements.

7.3.2 Interpretation

First we note that the natural-language-based approach could not answer all of our use cases. Some of the queries led to results which are not related to the query at all and for one query nothing was found. This illustrates the difficulties in automatic query building. Though if the results were found even when we searched for only a few movies their order was fine. However, this indicates the usefulness of MovieSearch and the graph-based query building which could form the informational queries and the plot relevant queries for all use cases.

Second the returning of only results that fulfill the given properties boosts the precision of MovieSearch compared to Valossa. Especially if there is exactly one movie that meets the conditions MovieSearch will only answer one movie whereas Valossa almost always fills the results with other movies. Even if the sole solution movie is Valossa's top-hit other movies are added to the results which harms the Precision of Valossa.

The Recall for a couple of the queries with a lot of possible results is very small because we restricted us to the top-10 results. But this affects both engines hence it does not harm the comparison. For example all the movies with music from Hans Zimmer (He was very diligent.) are way more than we could find with the top-10 results.

Let us compare the average values from table 7.12. These are the averages combined from table 7.10 and 7.11.

Search with	avg. Recall	avg. Precision	avg. F measure	avg. $NDCG_{10}$
MovieSearch	**66,60**%	**94,96**%	**66,06**%	**95,81**%
Valossa	47,62%	35,00%	22,38%	62,19%

Table 7.12: Comparing MovieSearch and Valossas averages.

In all fields MovieSearch beats Valossa. Especially the precision is high because of the property-based answers of MovieSearch. This affects the order-based $NDCG_{10}$ too because we have fewer noise answers in MovieSearch.

Decoupling our perspective from the comparison we notice that MovieSearch in the current instance is made to find movies that meet the given criteria in a very strict way. This boosts precision but naturally has the drawback that if one condition for a movie is only slightly missed this movie will not be in the answer. This sometimes leads to no or very few results.

A big possible improvement can be done in the ranking. For example Q_9 suffers from the current PageRank approach because there the top movie found is a documentary. Documentaries and talk shows gain a very high score due to the many different topactors that occur in them in the course of time or in short movie snippets. The PageRank score therefore boosts their scores. For example a documentary with the topic Oscar awards will have a lot of topactors appear and receive an appropriate high score. A solid score for relevant movies and topactors is therefore a field for future work.

Summing up, we have in general better results than the representative Valossa for

natural-language-based approaches. Next, we have to see if this advantage penalizes the usability. Valossa is of course very easy to use because you just type any text into the sole input field and are done. The following section presents the analysis of our usability tests for the query building with MovieSearch. The tasks we give to the user test participants are the same from table 7.9 on page 70 which displays the overview of all queries.

7.4 Quality of query building with the User Interface – User study

The user study is the usability test for the user interface. We want to see if users can find answers for the given tasks. This is the main goal of the whole system. For each task we have an expected query. We will compare the answers and the query that are found by the users to handle each task. Hence, we see if users can finish the tasks with relevant answers. But, for measuring the usability we want something that evaluates the process of building the queries as well. For this we count the amount of text input the participants need to build a query with results that satisfies them. We than will compare the average user inputs with the minimum amount to build the expected query. The assumption is: the fewer inputs the user needs, the more helpful and user-friendly the interface was.

This is a heuristic that tries to measure usability in an objective manner based on the work in [**?**]. Styperek et. al intended with this measure to gain a method that is easily repeatable and excludes the subjective opinions of the users. They only use it in the sense we will analyze the minimum needed amount of inputs for building the expected query. For our analysis we extend the usage of this measure to our user study by counting the number of inputs that is needed by the participants until they are satisfied with their query.

How many of the queries could be answered with relevant movies is the final part of the evaluation. From the user perspective the main goal is to find relevant answers. Hence, we will test if the found answers are the expected answers. If they are not the exact movies we expected, we see if they are at least relevant or in other words if they fulfill the query properties in a desired way. For our evaluation we count how many of the queries with relevant results were answered with the expected query. But this is mainly for our analysis. From the user perspective any query that returns the correct answers for a task is exchangeable. So we do not penalize tasks that are answered with the expected results but with another query than the one we expected. However, we will summarize all problems during the build process even if they were solved during it, too. These occurred difficulties are good starting points for future improvements.

Now, we begin with the consideration of the actual user study. General guidance for what we talked about and how to perform a do-it-yourself usability study were inspired by [**?** , chapter 9 - Usability testing on 10 cents a day].

7.4.1 Experimental setup – the participants of the user study

We performed the usability tests with eight participants. Some were in persona, others were carried out with a screen broadcast. The screen and the on-screen actions performed by the participant were recorded for a later in-depth analysis. Next, we consider the parameters regarding the participants.

Internet usage The stated average time per day a participant surfs the web is 6 hours containing 1 hour emailing.

MovieSearch website intentions Seven of the eight participants stated after a first glance on the website that they think this website allows them to search for movies by criteria, which is exactly true. The eighth participant regarded it as a review site for movies.

The next topic is the input counting during the build process.

7.4.2 Usability – count text inputs per query to achieve a result

Now, we count text inputs per query in the user study that they needed to achieve a result that satisfied them. We will use the text input count as a measure for the usability of the user interface. MovieSearch only needs some text input and some selection clicks. We only count text input because clicks are easy for the small selections of options.

> *"It doesn't matter how many times I have to click, as long as each click is a mindless, unambiguous choice."* [**?** , page 43].

To enable this, we introduced the fact suggestions in the first place. Now, we regard the measured text input amounts.

7.4.2.1 Measurements

The average of how many inputs were needed for each query until the participants were satisfied with the results or did not want to try further, is summarized in table 7.13. The minimum text input describes the amount of inputs that is needed to build the expected query in the shortest way.

Query	avg. user text inputs	minimum text inputs	relative user extra input
Q_1	1.6	1	62.50%
Q_2	1.8	1	75.00%
Q_3	1.1	1	12.50%
Q_4	3.4	2	68.75%
Q_5	1.9	1	87.50%
Q_6	1.5	1	50.00%
Q_7	2.4	2	18.75%
Q_8	2.9	2	43.75%
Q_9	2.9	1	187.50%
Q_{10}	2.9	1	187.50%
Q_{11}	2.8	2	37.50%
Q_{12}	6.1	2	206.25%
Q_{13}	1.4	1	37.50%
Q_{14}	5.8	4	43.75%
Q_{15}	6.4	4	59.38%
Q_{16}	2.5	2	25.00%
Q_{17}	3.8	3	25.00%
Q_{18}	2.0	2	0.00%
Q_{19}	4.9	3	62.50%
Q_{20}	3.3	3	8.33%
Q_{21}	6.8	3	125.00%
Avg.	**3.2**	**2**	**67.81%**

Table 7.13: Counting user text inputs during their query building. Compared to the minimum amount needed that is possible to build the expected queries.

Furthermore, we compute the average extra input the users needed for the task compared to the expected query with its minimum amount.

7.4.2.2 Interpretation

The extra amount of 68% is in our opinion acceptable since it only correlates to a rough single extra input in average. This is even more convincing, after recognising that often for the first kind of a query there are a couple of extra inputs needed but for the following similar queries they were simple to answer. We will see more about this after comparing the quality of the final queries in section 7.4.3.

The next topic is classifying our results by comparing them to similar work.

7.4.2.3 Comparing with results from [?]

The work that inspired the usage of counting input fields is [**?**]. Their example query is Q_{11}: *In which films directed by Garry Marshall was Julia Roberts starring?*. Styperek et.

al did not include an evaluation with users but their values compare to our minimum needed input per query. Their own system is called SFC. The other two systems Styperek et. al compare their work with are GoR and NAGA. All three of them are graph-based which makes them a good comparison for MovieSearch, too.

System	Nr. of text inputs needed to build query Q_{11}
GoR (GoRelations)	9
NAGA (Not Another Google Answer)	6
SFC (Semantic Focused Crawler) [**?**]	5
MovieSearch	**2**

Table 7.14: Comparing minimum needed amounts of input with other systems for query Q_{11} : *"In which movies directed by Garry Marshall was Julia Roberts starring?"*.

The results in table 7.14 contain the information from [**?** , page. 197, Table 3]. The Fact Suggestion makes the input process smooth. MovieSearch only needs two text inputs which are the two names in the query. The needed relations are selected from the suggestions with two clicks. But clicking does not cost effort and the selection is straightforward for the query by the solely fitting relations that are suggested.

Styperek et. al also performed tests for a set of queries with the two other compare systems. We compare our minimum inputs and the average inputs of our test study participants with their results in table 7.15.

System	Nr of text inputs needed
GoR	5.7
NAGA	3.6
SFC [**?**]	2.8
MovieSearch, minimum avg.	**2.0**
MovieSearch, user test avg.	3.2

Table 7.15: Comparing needed number of text input in average with other systems for a set of queries.

The values in table 7.15 are included from [**?** , page 198, Table 5].

Summing up the comparison of MovieSearch text inputs with those in [**?**] we conclude that the Fact Suggestions reduce the amount of text inputs to a new minimum among the compared systems. Even the average inputs from our user study are competing with the overall minimum needed input values of the other systems. This result indicates the improvement of usability with MovieSearch. However, note the different set of queries in our work and those in [**?**]. A new comparison in future with the same queries for all systems would achieve more meaningful conclusions. But, for the query Q_{11} we can

definitely see the improvement by MovieSearch. For the overall queries we can at least deduce the tendency of improvement.

The topic of the next section is the quality of the query building in the user study.

7.4.3 Quality of built queries and results from the user study

Here, we want to evaluate the quality of the queries that were finalized by the participants for each task. For this we check if they found the expected results or relevant alternatives.

For comparing the expected query with one that was constructed by a participant we compare the sets of conditions of those two queries. If both contain the same conditions we call the queries equal. However, we are mainly interested in the difference of two queries by their conditions if they lead to different results. We think of a query as a representative of a set of queries that leads to the same results of the current query. For the usability perspective it suffices if the user can construct any query from the representative class that returns the desired answer. This is the reason for our classification of the user queries.

For each built query we first classify it as *expected*, *relevant* or *not relevant*. *Expected* means it returns the expected results (with the expected query or another query). *Relevant* means the answer consists of movies from which **at least the half** is relevant for the task. *Not relevant* means the answer consists of movies from which **fewer than the half** is relevant for the task.

An example for an expected query in the internally used SPARQL form for task Q_1 is displayed in listing 7.1.

```
SELECT DISTINCT ?Movie WHERE {
  ?Movie <directed-by> <Nolan, Christopher (I)> .
}
```

Listing 7.1: Expected query of the first task Q_1.

The SPARQL conditions correlate to the conditions of the abstract queries. After adding a condition for ordering the results, this is the form of the queries that are answered by the SPARQL Backend [**?**]. Next, we sum up the classification of the final queries by the participants of the user study.

7.4.3.1 Measurements

Most of the tasks for the participants were solved with the expected results like it can be regarded in table 7.16.

Total query answers	168	
with expected results	159	**94.64**%
at least half relevant hits	7	4.17%
fewer than half are relevant hits	2	1.19%
expected Result with expected Query	130	77.38%
expected Result with other Query	29	17.26%

Table 7.16: Classifying the final versions of the built queries in the user study.

Furthermore, 81% of the query answers with expected results used the expected queries. This can be deduced from table 7.16.

7.4.3.2 Interpretation

The share of queries that were answered with the expected results is high with 94%. Even both expected results and the expected query were built in three-quarter of the tests. These results indicate the usefulness of MovieSearch, since the goal of enabling users to search for movies by properties and find answers is in the majority of the cases fulfilled.

Of the two tests that did not find enough relevant hits, one was a version of Q_2 that tried to find the movies with songs from Hans Zimmer with the plot description. Which happened due to a misunderstanding of the word plot by the participant. Here, the majority of the results were awards where Hans Zimmer received a price which was part of a text part. The other problem case was a test for query Q_{15} where the task description was inputted into the plot field. It did not return any results whereupon the participant preferred to continue with the next task. The cases were only some relevant hits were found happened also due to a fact that was inputted as plot. Though this happened for the very vague query Q_4(*action and funny*) that was intended to be answered by facts about genre. The usage of *action* and *funny* as plot words did return some relevant results at least. Therefore, the only problem that influenced the finalized queries where facts that were inputted unexpectedly as plot. We will address this further in section 7.4.4.

Of course it would be preferable to have more participants than the eight. Because, more testers result in more tests what improves the validity of the test results. However, a large scale user study exceeded the scope of this work. But even a small amount of usability testers reveals many existing problems. Especially fundamental problems are hard to miss. Since our results are quite satisfiable at the end of each query building per test, we may say MovieSearch has no fundamental problems.

Nevertheless, the next topic considers a couple of difficulties during the building of the queries. Most of them got resolved by trying different things while building the query, but a look at them will help to detect points that can be improved in future.

7.4.4 Difficulties during the user study

Here, we will discuss all difficulties that occurred during the usability tests. Most of the problems could be resolved by the participants during the query building.

What kind of difficulties occurred? First we list all occurred difficulties and their abbreviations which we will use from here on.

PF: A participant inputted a condition as a **Plot**.
But, we expected this condition to be inputted as a **fact** instead.

FP: A participant inputted a condition as a **Fact**.
But, we expected this condition to be inputted as a **plot** instead.

R: Other **R**elation used than the expected one.

S: No usage of a 3-ary relation where it would be possible. The condition was added by two **S**eparate facts for the subrelations.

M: Given **M**any words as a plot description, the problem can be that there is no one document for the movie which contains all of the words. But, for each word there exists a document that at least contains one of the given words. Hence, the participants were confused that for the separate words the same movie is found, but for all the words no movie is found.

A: **A**ll fact conditions for the task were inputted at once into the fact description field. This is currently not supported. We expect an iterative query building fact by fact.

B: Candidate **B**utton misinterpreted as possibility to use the negated fact of the candidate fact.

D: **D**ata base handles equality other than expected by the participants. They expected for example that *Todd* as a value should lead to *Sweeney Todd* being in the results (suggestion and query result).

The only difficulty that influenced the measurement results of the quality directly is **PF**. The input of a fact description as a plot does not build a query with the strong properties of facts. For a few cases this led to fewer relevant answers.

Under what circumstances did the difficulties occur? Before we consider some examples for the difficulties, we regard table 7.17. This table summarizes at which queries what difficulty did occur at least once. Most of them were resolved by the participants during the tests.

Type	List with occurrences
PF	Q_1, Q_2, Q_3, Q_4, Q_{10}
FP	Q_{15}, Q_{16}, Q_{17}, Q_{21}
R	Q_1, Q_4, Q_8, Q_{12}, Q_{14}, Q_{19}
S	Q_9, Q_{10}, Q_{12},
M	Q_{15}, Q_{21}
A	Q_7, Q_{11}, Q_{14}, Q_{20}
B	Q_9, Q_{17}
D	Q_4 Q_9, Q_{10}, Q_{12}, Q_{17}

Table 7.17: Overview of at which query did the difficulties occur at least once.

There were often difficulties that could be resolved. This indicates that the overall layout is not quite optimal, but on the other hand we offer good help to resolve those difficulties.
Next, we regard some example occurrences of difficulties.

At the beginning of the user study, for two participants **PF** happened because they were not quite sure what plot means in their mother tongue. After this was clarified it did not state a problem any longer. Note, that **PF** does not necessarily have to lead to less relevant results. For example for Q_1 and *Christopher Nolan* the same results are found with the name as a plot like with the fact *directed-by Christopher Nolan*. This indicates the strength of semantic search in the documents we provided in the data.

Instead of a plot search for *sword* in Q_{17} a participant tried to find the movie by a keyword search as a fact. Generally this should be a valid approach, too. *Sword* and *sword-fighting* are keywords that exist in the data. But, somehow one of the *Conan* movies which is a solution movie amongst others, did not have the keyword *sword* set in the data. With a more complete data the answer for the alternative query would have been the expected result. This illustrates the strong properties and demands of the facts. The difficulties happen if the data is not as complete as we expect. Like in this example, missing a keyword in the data that applies to the movie for sure will lead to fewer relevant results.

An example for the occurrence of **R** is the confusion between the relations *from-year* and *released*. A movie can have a couple of *released* dates because they can differ for different countries. Whereas *from-year* only binds one year to the movie. The year in which it first was seen is the year in the IMDB title and the value of this relation. The main difficulty is that for specifying a *released* relation only a date is accepted by the user interface and not only a year like for *from-year*. This should be resolved by allowing only parts of a date as a value for released, too. If the participants could have entered a year only at *released* they would have found the movies with this relation easily. Another example for **R** is the exchange of *has-genre* with *has-keyword*. The cause for the difficulty here is the variety of different levels of completeness in keywords per movie.

Another difficulty was encountered in the following scenario. On the one hand the data has slight inconsistencies. Like, in *The Hobbit* the character is called just *Frodo* while in

the movies *Lord of the Rings* the same character is called *Frodo Baggins*. Those movies live in the same universe, therefore the same characters should have ideally the same name. The difficulty for this is currently only one of the character names can be selected. Therefore, the results only show the *The Hobbit* movie or the other ones depending on the character selection. Here, the solution could be a better consistency in the data. But on the other hand, a similar problem occurs with the character name *Todd* which occurs in some movies. While looking for the target character *Sweeney Todd* with only the part of the name *Todd* as search value it caused confusion that the target was not found via the partial name. Here, *Todd* is a valid independent character name and part of another character name. It would be desirable behavior to allow the part also find names that contain the part instead of mere full matches. By allowing this, we would additionally fix the problem with *Frodo* and other characters that appear only sometimes with surnames. Regarding the data inconsistency with *Frodo*, the reason for the different two names in first place are the missing topactors in the IMDB downloadable data. Since in *Lord of the Rings* he is a main character and therefore not contained in the available IMDB data set, we have to add him with the TMDB data. In *The Hobbit* he is only an underpart and therefore already contained in the IMDB Data. Detection of such variations of names in movies from the same universe is an open task for future work. Besides, allowing partial matches of values in the final queries could improve the range of hits.

Concerning the example of *Frodo* in the context of Q_{10} the difficulty regarding the awareness of 3-ary relations with **S** occurred, too. The main observation with **S** is, that after the participant first saw the possibility of inputting 3-ary relations it would be used efficiently and often. The difficulty of **S** happened because it sometimes needed a couple of queries until the participant found the 3-ary relation input. Currently, while only inputting names into the fact description there is no suggestion of 3-ary relations. Only if a tag word or a connectorword is found the 3-ary relation is suggested. For example the fact description *Johnny Depp as Todd* triggers a 3-ary relation suggestion while *Johnny Depp Todd* will not. Awareness for the 3-ary relation could be improved by suggesting it more often if the selected entity is part of one of those 3-ary relations.

Besides, only the half of the participants had the opinion that a 3-ary relation differs from the split subrelations. This should get more clearer. In the discussion after the user study some participants stated it would help to rename the connector phrase from *and* to *as* in the selection box of 3-ary relations (see figure 5.2). With a more intuitive understanding of the difference it will be used more often.

For **M** we have to ask what is the more interesting answer for the user of his query. He most likely is more interested in the movie that has all the plot words in the set of the documents for that movie instead of inside a single document. This is easily changeable by splitting the context part in the SPARQL query into separate contexts per word. By doing this we will search for a movie that has for each plot word at least one document containing it. This was the expected behavior of the participants and should therefore be used.

A should be implemented in future to provide a shorter input option.

To avoid **B** in future a clearer label of the slide button could be tested. Instead of USE

and NOT we could try something like USE and DON'T or use symbols for on and off.

Interpretation Some difficulties will be always there. This is part of exploring. It is important though that the participants can overcome the difficulties. This was the case for the majority of the user study.

The problems **PF** and **FP** occur only because we have two input fields for the two categories of plot and facts. On the one hand the two input fields make such problems possible in the first place. On the other hand, they help to specify queries with text in a way the user has much control about the query building. In the majority of the tests the partition of the tasks into plot and facts worked. Than the results were very good which we saw in the previous sections. Though, if the user does not manage a good partition of the task than the results can be quite lackluster. We conclude two things from this. First the general approach of two input fields is convenient for the query building. Second the partition is vital and therefore we need a better guidance from the user interface to create the partition. In future there should be a deeper inspection of the input. For each input field we should check how the input would perform in the other input field. The text score of the matches with the plot could be compared with the graph scores from the suggestions. For this we have to utilize a better ranking of the RDF graph at the Suggestion Server and the scores from the SPARQL engine for text matches. Having some form of verification of the partition into plot and fact will make the user interface more helpful.

During the user study we observed in general after any type of task was performed once successfully further repetitions of such a task were no problem. But, sometimes it needed a couple of tries or queries until the first task worked. An example for this is the 3-ary relations. For this it would be beneficial to provide a short tutorial. Besides, it indicates that there have to be better design choices that can improve the intuitive usage. In the future work we will present some ideas to improve the usability and the intuitive composition.

The next and last chapter will sum up our work and present possibilities for future work.

Chapter 8

Conclusions

The last chapter of this work discusses the summarized results of the different components. Furthermore, we will suggest different areas for future work.

8.1 Discussion

The goal was building a semantic search for movies with a high amount of usability while keeping the query building transparent.

First we regard the suggestions and the exploring of the data. They are relevant for the query building. The query building process did return satisfiable results in the user study. The number of inputs needed in average was in an expected range. This indicates that exploring the data is possible with the user interface and the suggestions. The autocompletion together with the suggestions allowed the users to find the facts they wanted to build in the majority of the tests.

The difficulties that are introduced due to demand of the splitting of tasks into plot and facts are acceptable. Because, the difficulties did not hinder the users to achieve good final queries. Furthermore, the plot and facts categories enable the transparent query building. There, the user has a good amount of influence over the question he constructs. Nevertheless, we found several points to improve the usability. Some of them will benefit the splitting of tasks into plot and facts by better guidance. More help from the user interface in this area will improve the quality of the split that is made by the user.

Another difficulty was the awareness about 3-ary relations. After they were found, the user could use them easily. But, the 3-ary relations should be discovered earlier. Maybe by a tutorial or the addition of the 3-ary relation suggestion to related entity suggestions. Then again, one question is if the 3-ary relations are really necessary. The cases were the 3-ary relation differs from the subrelations are slim after all. But, if we do not think they are important for the condition side they at least reduce the amount of text input amounts. Furthermore, we conclude that in future partial matches from current query to results should be allowed, since these are the expectations of many users from the user study. In our example we want that the condition that specifies the character *Todd*, finds also movies with character *Sweeney Todd*. This influences the

relevance of 3-ary relations. Given the case where we know the actor with last name *Pitt* played a character with the first name *John*. In the given case with partial matching allowed, we would find a lot of hits with the two subrelations. Here, the 3-ary relations become valuable again. They reduce the amount of irrelevant results and map the fact description knowledge with more complete information to the condition. However, a better understanding of 3-ary relations should be communicated, since the half of the users thought they make no difference to the split subrelations.

The suggestion response times are good enough that they provide interactivity for the user. Therefore, this requirement for usability is fulfilled. However, this comes at a sizeable demand for memory space on server side.

The quality of the query results by MovieSearch is very good. Precision and $NDCG_{10}$ were almost perfect and the few poor Recall values were caused by our restriction to the top-10 results. Especially the high precision occurred due to the fact-based queries. The power of the manual query building was indicated by the comparison to the representative of the natural-language-based approach. MovieSearch performed better in every measured aspect.

Concerning our underlying data we can say it was currently sufficient. Many queries could be answered. However, we have found a couple of things that can be improved. For example the distribution of keywords is very diverse. More sources for data (facts and plots) would definitely add more information to our pool. Many other sources are unfortunately liable to pay costs. A good start would be to gain access to a broader IMDB-Pro data with a paid membership. We did discover the missing topactors and fixed this, but maybe this is not the only incompleteness in our subset of IMDB data.

Overall, the approach of the two input categories plot and facts, together with good suggestions seems to be promising for balancing usability with powerful semantic queries.

The next section will consider further refinements and future areas of work.

8.2 Future Work

The ideas for future work are grouped into different fields with an own intention. We will visit different fields where some improvement can be done.

The future work suggestion are ordered by the amount of expected improvement.

Range of relevant results

Additions that improve the results by a broader result selection or a better relevance have highest priority for the users.

1. **Partial value matching:** the expectations of a couple of user study participants were that for example character *Todd* finds not only equal matches but also everything that contains the given value. In our example with partial value matching

also movies with the character *Sweeney Todd* would have been found, because this contains the given value. This would increase the range of the results and would be more forgiving for not perfectly selecting the exactly intended value from the suggestions. This would also help with finding values in the data like *Sweeney Todd / Jack Sparrow* which belong to a documentary were both were played by the same person. These could additionally be split into two cast relations in the data though.

The different candidate values on which we want to perform the partial matches are a subset of the found entity matches in the suggestions. Because by selecting a suggestion that matches the starting request text, a further specification has to be already contained in the suggestions. We could regard the union of all queries with the different candidates as character which are known from the suggestions already.

The estimated time is around a week to adapt all components.

2. **Scores and ranking:** we already saw the problems of the PageRank and for example talk shows. Since there many topactors participate this will get a higher score than it should. They have a lot of ingoing arcs hence PageRank gives them score.

   Especially for queries with a lot of results, the ranking is of utmost importance. An idea for general improvement of the score could not only consider the graph but also the documents. Than, if an actor occurs in a document to a movie this boosts his score. This should help topactor scores. It would be a much nicer approach than our manually boosting of the topactors. To handle the documentary problem we also could consider implementing more types for TV shows, documentaries or video games which all occur in the IMDB data. Currently they all get combined under the *movie* type. Building a stable ranking is a challenging piece of work. The estimated time could range from a month or more depending on how refined the rank should get.

Flexibility of input

The second important part for usability is the input. The more option to input, the better. A better guidance by the web site can ease the input process, too. The difficulty markings (like **PF**) are explained in section 7.4.4 on page 81.

1. **Guidance for splitting task into plot and facts:** the most important goal for improving the query building is avoiding the difficulties **PF** and **FP** which describe the swapping of a plot and a fact for a condition. The web site could internally test each input with the results from the other input category. Afterwards, if we detect way better scores relatively at the other input category, we suggest the user to consider the alternative category for his input. The better scores from the RDF graph will help to value the facts for comparing. The text scores from the SPARQL engine could be used for comparing this side. Another simple addition would be

to check for a plot word that did not find any documents to try it as a fact. Refining both scores and figuring out how to compare them in between the both categories should be done in a couple of weeks.

2. **Many fact inputs at once in fact description:** as a shortcut for a faster input of multiple facts we can add a check for more relations in the fact description field. If more than one are found and they do not form a 3-ary relation, we can add the best matching value each and create all facts. This will allow a faster input, but without the comparison of the different suggestions per relation. Providing this option was requested at difficulty **A**. The implementation will need one or two days at most.

3. **More options for Relation Matching:** the Relation Matcher currently detects numbers in the input. If a number is detected the Relation Matcher only returns number-based relations. This could be more refined. We could detect special numbers like dates and than only suggest date-based relations. This should be manageable in a day. Furthermore, inputs into fact suggestion could be tested on client side. If they only consist of a date we can request appropriate relations from the suggestion server. To better handle difficulty **R** while modifying the relation matching process, we could add more alternatives. Maybe even through a learning process that involves user feedback. With feedback we could gradually find all possible terms that describe a relation. Estimated time is probably a couple of days.

4. **Modifiable current query:** it would be nice if the entries in the current query would be editable. The compare-base values and the comparison type are no problem as changeable fields. But, for relations and word-based values we have to test for each change if an appropriate fact exists in the graph. This was an intention of the suggestions to only find relations with values that actually exist in the graph. Dropping this guarantee by just enabling all changes to the current query would be counterintuitive. But, we could realize something that allows to input a change and we insert the best entity or relation that was found in the graph. Nevertheless, this would be a nice feature that improves usability. For testing some different approaches and seeing what works best this will probably be a task for a week.

5. **Fuzzy date input:** allowing for example years only as a date would give the user more options. This would prevent problems like we saw with *from-year* and *released* (**R**). This would only need a separate user interface attendance. After the input this would result in a bit different FILTER-ing in the SPARQL query. Nevertheless, this could be integrated in a day.

6. **Autocompletion for suggestion boxes:** adding an autocompletion for the suggestion box inputs would be a natural addition to the autocomplete of the fact description input. For this we should make the autocompletion relation sensitive. For each suggestion box we know the corresponding relation. An autocompletion in a suggestion box should consider the relation of the box and only show entity

hits that match accordingly. The matches should be sensitive to the current relation that is part of the suggestion box. This is all possible with existing methods. They just have to be used by the Autocompleter. This should be doable within a day.

7. **Spell checking:** add a spellchecker for the inputs to tolerate small typos. We already talked about a change to the keys of the inverted indexes that use *k*-grams. This is a basic change and a lot of test cases have to be added likewise. The above at least holds if we also want to change the filter method that defines the matching rules in the graph methods. Easier would be to only use something like edit distance for the autocompletion, and continue to use the current filter method (contains from start) for the graph-based methods. The realization likely needs a couple of days or a week.

8. **Subfacts:** a better integration of specifying the second part of a subfact into the user interface is an open task.

3-ary relations – Awareness

A better awareness of 3-ary relations would be nice. Currently the 3-ary relations are only suggested if there is any tag word found that hints for a 3-ary relation. Hence, some participants of the user study needed a couple of queries until they found the option to insert 3-ary relations. To help this out we could add an appropriate suggestion of a relation if an entity is found that matches it. If we see this would spam the relation hints too much we could only suggest 3-ary relations if the input can be split into e.g. an actor and a character.

More utility

The changes discussed in this section are most likely combined with a rework of the display of the current query. Therefore the changes here could take a while including different design ideas testing. Estimated time is a couple of weeks.

1. **Current query logic:** if no results are found for the current query we could check if a subset of the conditions finds any results.

2. **Advanced conditions:** another goal could be more possibilities in how the conditions are connected in the current query. Currently we use logical AND to combine all conditions. A task could be to enable the option to add a condition with OR. Besides, interesting is the negation of a condition with an NOT. For example for the *James Bond* query it would be nice to add a condition that states that the movie has not the genre documentary.

3. **Order by:** enabling an order by a custom condition.

4. **Result movies and their main summary:** the preview plot should be changed to the main summary. At least the main summary should be loadable as a more

meaningful text to describe the movie instead of the snippet where the plot words were found.

Data and Preparation

1. **Evaluation of entity matchers:** the alternative entity matcher that uses binary search on a sorted list of permutations has to be evaluated for more search parts. We discussed this in section 7.1.2.3. To make a better experiment, we have to figure out a better experiment setup. A more detailed evaluation will help to decide if we want to swap our inverted index matcher with the binary search one. Selecting the option that balances speed and space while meeting our requirements will improve the quality of the system. Another thing to keep in mind is the impact on the future spell checking. With the inverted index we could use k-grams as keys, which would not be feasable for all permutations. The experiments will most likely take a week for finding good setups. A potential swap can be performed in half an hour plus server restart time.

2. **Split multi-values:** better results would be found if we split more thoroughly relations that have one value set in the raw data but with a value that combines a number of values. This way we could improve the relevance of the found hits. For example *Sweeney Todd / Himself* should be split and be added as two relations into the graph which would make finer fact conditions possible. Another example for a multi-value would be *Rome, Lazio, Italy*. The main problem here is the many different formats for the different value combinations. But, with a data analysis most of the variations could be found and than in the preparation be treated. This should not need longer than a couple of days to realize.

3. **Entity Recognition:** to speed up the precomputation for the entity recognition in the documents we could use multi threading. Since the process only needs the current document and the current movie context we could split the source files and perform the entity recognition for the separate parts simultaneously.

4. **Live database support:** in the movie domain we mainly have to consider adding new information. Common changes are a new movie, a new award won etc. which we could realize by adding new facts to the graph regularly with new information.

   Removing or changing old facts has more impact since it could render results from the entity recognition useless. Here, we need a new data preparation anyhow.

Loosen the domain restrictions

So far we considered mostly fine tuning of MovieSearch. Remember that this is only one possible domain for semantic search. The big step after MovieSearch has a satisfiable level reached, is applying the suggestions to a broader domain. For this we would have to review the data preparation and define types and relations dependent on the new domain and its data.

Two possible topics we can consider for this scheme are regarded in the following.

***n*-ary relations with $n > 3$:** Is there a need for such relations? For this we have to think how we store them on the server side and how we include them in the user interface. For big n building all shortcut possibilities can become quite many.

Space consumption: Table 7.3 on page 64 shows that the absolute space consumption of the Suggestion Server is quite a lot, since we have the full RDF graph in main memory and the inverted indexes. Especially, if we want to apply our results to a domain with much more data we have to think about saving space consumption.

One straightforward space reduction could be done by compressing the ID lists in the inverted indexes. Currently, we store directly the IDs in the index lists. For example for a prefix ["bre"] we have a list of entity IDs $(3, 45, 60, 73)$. Instead of storing the IDs, we could do the common thing of storing only the differences to the next ID. This would result in the list $(3, +42, +15, +13)$. Here, we gain an entry at any position by the sum of all entries before this position. This is called *delta encoding*. *Delta encoding* works for us because our original lists are sorted increasingly. Various other compression methods can than further be used or instead.

On the one hand we could store the precomputed results in files. On the other hand we can change the structure in which we store the facts. For this we could use a large scale graph that only looks at graph clippings locally.

Acknowledgment

First of all I want to express my gratitude to my dear Caro. Her support fuels my strength and keeps me going through all difficult times.
I thank my supervisors for their guidance, especially Björn who answered my questions as best as he could even when I asked them again. His remarks helped me a lot in creating this work.
Furthermore, I want to thank my user study participants. It was not always fun for you but you all made it to the last query. Thanks to you Adrian, David, Carola, Ricarda, Jonas, Richard, Marlene and Patrick for giving me a piece of your time.

Besides, I want to thank the authors of my reference books for JavaScript [**?**], [**?**] for Python [**?**] and for C++ [**?**] for their help and textish advice. Always keeping in mind a clean coding style with the aided recall of [**?**] and my beloved Linter.

Appendix A

Experiments

A.1 Relevant results per query task

If there are very many movies with the property of the task than the total result count is meant to be a minimum value of relevant movies. There, for the total result count we count movies in our data with that properties. We consider all of them as matches.

Query	Total number of matching Movies	Matching Movies
Q_1	12	Batman Begins (2005) Doodlebug (1997) Following (1998) Inception (2010) Insomnia (2002) Interstellar (2014) Memento (2000) Quay (2015) The Dark Knight (2008) The Dark Knight Rises (2012) The Prestige (2006) Dunkirk (2017)
Q_2	237	*... too many movies with that property*
Q_3	1819	...
Q_4	10034	...
Q_5	1713	...
Q_6	971	...
Q_7	1	This Must Be the Place (2011)
Q_8	7	Gangs of New York (2002) *... continued in next table*

Table A.1: Relevant and valid results for the use case queries, Part 1.

Query	Total number of matching Movies	Matching Movies
Q_8	7	The Aviator (2004) The Departed (2006) Shutter Island (2010) The Wolf of Wall Street (2013) The Audition (2015/III) The Devil in the White City (????)
Q_9	4	GoldenEye Tomorrow Never Dies The World is not Enough Die Another Day
Q_{10}	4	The Hobbit: An Unexpected Journey (2012) The Lord of the Rings: The Return of the King (2003) The Lord of the Rings: The Two Towers (2002) The Lord of the Rings: The Fellowship of the Ring (2001)
Q_{11}	4	Valentine's Day (2010/I) Pretty Woman (1990) Runaway Bride (1999) Mother's Day (2016)
Q_{12}	1	Sweeney Todd: The Demon Barber of Fleet Street (2007)
Q_{13}	2401	...
Q_{14}	1	Gladiator (2000)
Q_{15}	1	Crash (2004/I)
Q_{16}	3	The Wolf of Wall Street (2013) The Departed (2006) The Basketball Diaries (1995)
Q_{17}	3	Conan the Barbarian (1982) Conan the Destroyer (1984) Red Sonja (1985)
Q_{18}	3	Pirates of the Caribbean: The Curse of the Black Pearl (2003) Pirates of the Caribbean: At World's End (2007) Pirates of the Caribbean: Dead Man's Chest (2006)
Q_{19}	1	Non-Stop (2014)
Q_{20}	1	Mr. & Mrs. Smith (2005)
Q_{21}	1	The Talented Mr. Ripley (1999)

Table A.2: Relevant and valid results for the use case queries, Part 2.

A.2 Results from MovieSearch and Valossa

This section lists all the results per web site for each of the queries. These tables are the basis for the evaluation in section 7.3. Movies deemed relevant for the current task are marked with ✔ otherwise with ×.

Q_1: *"Movies from Christopher Nolan."*

MovieSearch results	relevant	**Valossa** results	relevant
The Dark Knight Rises (2012)	✔	Dunkirk (2017)	✔
The Dark Knight (2008)	✔	Interstellar (2014)	✔
Batman Begins (2005)	✔	The Dark Knight Rises (2012)	✔
Inception (2010)	✔	Inception (2010)	✔
Interstellar (2014)	✔	The Dark Knight (2008)	✔
The Prestige (2006)	✔	The Prestige (2006)	✔
Insomnia (2002)	✔	Batman Begins (2005)	✔
Memento (2000)	✔	Insomnia (2002)	✔
Following (1998)	✔	Memento (2000)	✔
Untitled Christopher Nolan Project (2017) == *Dunkirk*	✔	Following (1998)	✔

Table A.3: Results and their relevance for query Q_1.

Q_2: *"Movies with songs from Hans Zimmer."*

MovieSearch results	relevant	**Valossa** results	relevant
The Dark Knight Rises (2012)	✔	The Lion King (1994)	✔
The Amazing Spider-Man 2 (2014)	✔	The Letters (2015)	×
Batman v Superman: Dawn of Justice (2016)	✔	The Prince of Egypt (1998)	✔
The Dark Knight (2008)	✔	Spirit: Stallion of the Cimarron (2002)	✔
Angels & Demons (2009)	✔	Shark Tale (2004)	✔
Rush (2013/I)	✔	Interstellar (2014)	✔
Pearl Harbor (2001)	✔	Muppet Treasure Island (1996)	✔
Batman Begins (2005)	✔	The Power of One (1992)	✔
Pirates of the Caribbean: At World's End (2007)	✔	Joseph: King of Dreams (2000)	×
How Do You Know (2010)	✔	Tears of the Sun (2003)	✔

Table A.4: Results and their relevance for query Q_2.

Q_3: *"Movies made by Jerry Bruckheimer."*

MovieSearch results	relevant	**Valossa** results	relevant
Without a Trace (2002) {Between the Cracks (#1.4)}	✔	Kangaroo Jack (2003)	✔
Without a Trace (2002) {Hang on to Me (#1.13)}	✔	Armageddon (1998)	✔
Dark Blue (2009) {Shell Game (#2.8)}	✔	Beverly Hills Cop II (1987)	✔
Cold Case (2003) {Torn (#4.21)}	✔	Pearl Harbor (2001)	✔
CSI: NY (2004) {Greater Good (#5.23)}	✔	Prince of Persia: The Sands of Time (2010)	✔
Confessions of a Shopaholic (2009)	✔	The Lone Ranger (2013)	✔
Glory Road (2006)	✔	The Sorcerer's Apprentice (2010)	✔
National Treasure (2004)	✔	Farewell, My Lovely (1975)	✔
Pearl Harbor (2001)	✔	Con Air (1997)	✔
National Treasure: Book of Secrets (2007)	✔	Pirates of the Caribbean: At Worlds's End (2007)	✔

Table A.5: Results and their relevance for query Q_3.

Q_4: *"Funny Movies with action."*

MovieSearch results	relevant	**Valossa** results	relevant
West Side Storytime (2014)	✔	Kung Fu Hustle (2004)	✔
Killerz (2015)	✔	Last Action Hero (1993)	✔
Waisa Bhi Hota Hai Part II (2003)	✔	Team America: World Police (2004)	✔
Guns, Drugs and Dirty Money (2010)	✔	Hot Fuzz (2007)	✔
Kolay para (2002)	✔	Action Jackson (1988)	✔
A Low Down Dirty Shame (1994)	✔	Kung Pow: Enter the Fist (2002)	✔
Crocodile´ Dundee II (1988)	✔	Black Dynamite (2009)	✔
Human Cargo (1936)	✔	Tropic Thunder (2008)	✔
Grand Theft Auto V (2013)	✔	The Heat (2013)	✔
Bad Ass (2012)	✔	The Interview (2014)	✔

Table A.6: Results and their relevance for query Q_4.

Q_5: *"Movie that is 111 minutes long."*

MovieSearch results	relevant	**Valossa** results	relevant
The Water Diviner (2014)	✔	Ratatouille (2007)	✔
Against the Ropes (2004)	✔	Grill Point (2002)	✔
Thelma (2011)	✔	The Choice (2016)	✔
Dead in 5 Heartbeats (2013)	✔	Audrey Rose (1977)	×
Kamienie na szaniec (2014)	✔	Hannah (2011)	✔
Çogunluk (2010)	✔	The War Game (1965)	×
Ruby Cairo (1992)	✔	RED (2010)	✔
Les caprices d´un fleuve (1996)	✔	In the Electric Mist (2009)	×
L´accompagnatrice (1992)	✔	Pan (2015)	✔
Populaire (2012)	✔	Police Story 2013 (2013)	×

Table A.7: Results and their relevance for query Q_5.

Q_6: *"Movies released at 11.11.2011."*

MovieSearch results	relevant	**Valossa** results	relevant
Real Steel (2011)	✔	11-11-11 (2011)	×
Casus belli (2010)	✔	11:11 (2011)	×
J. Edgar (2011)	✔	11/11/11 (2011)	×
Jack and Jill (2011/I)	✔	Immortals (2011)	✔
What´s Your Number? (2011)	✔	Dream House (2011)	×
Moneyball (2011)	✔	Rise of the Planet of the Apes (2011)	×
Mlyn i krzyz (2011)	✔	Urumi (2011)	×
Slacker 2011 (2011)	✔	Rockstar (2011)	✔
A Little Bit of Heaven (2011)	✔	Engeyum Kadhal (2011)	×
Tower Heist (2011)	✔	Anonymous (2011)	✔

Table A.8: Results and their relevance for query Q_6.

Q_7: *"Movie that is 111 minutes long and released at 11.11.2011."*

MovieSearch results	relevant	**Valossa** results	relevant
This Must Be the Place (2011)	✔	11-11-11 (2011)	✘
		Immortals (2011)	✘
		Anonymous (2011)	✘
		30 Minutes or Less (2011)	✘
		Dream House (2011)	✘
		247°CF (2011)	✘
		Gnomeo & Juliet (2011)	✘
		10 Years (2011)	✘
		Kill the Irishman (2011)	✘
		Urumi (2011)	✘

Table A.9: Results and their relevance for query Q_7.

Q_8: *"Movie by Martin Scorsese and stare role Leo DiCaprio."*

MovieSearch results	relevant	**Valossa** results	relevant
The Wolf of Wall Street (2013)	✔	Goodfellas (1990)	✘
The Departed (2006)	✔	Raging Bull (1980)	✘
Gangs of New York (2002)	✔	Taxi Driver (1976)	✘
The Aviator (2004)	✔	My Voyage to Italy (1999)	✘
Shutter Island (2010)	✔	A Personal Journey with Martin Scorsese Through American Movies (1995)	✘
The Concert for New York City (2001)	✘	The Departed (2006)	✔
The Audition (2015/III)	✔	The Last Waltz (1978)	✘
The Devil in the White City (????)	✔	Bob Dylan: No Direction Home (2005)	✘
		After Hours (1985)	✘
		Mean Streets (1973)	✘

Table A.10: Results and their relevance for query Q_8.

Q_9: *"Movies where James Bond was played by Pierce Brosnan."*

MovieSearch results	relevant	**Valossa** results	relevant
Oscar, que empiece el espectáculo (2008)	×	Die Another Day (2002)	✔
Die Another Day (2002)	✔	GodenEye (1995)	✔
The World Is Not Enough (1999)	✔	The World Is Not Enough (1999)	✔
Bond Girls Are Forever (2002)	×	Casino Royale (2006)	×
Tomorrow Never Dies (1997)	✔	The Thomas Crown Affair (1999)	×
True Bond (2007)	×	Tomorrow Never Dies (1997)	✔
GoldenEye (1995)	✔	The James Bond Story (1999)	×
Double-O Stunts (2000)	×	Spectre (2015)	×
The Bond Sound: The Music of 007 (2000)	×	Everything or Nothing: The Untold Story of 007 (2012)	×
Cubby Broccoli: The Man Behind Bond (2000)	×	The November Man (2014)	×

Table A.11: Results and their relevance for query Q_9.

Q_{10}: *"Movies where Frodo was played by Elijah Wood."*

MovieSearch results	relevant	**Valossa** results	relevant
The Lord of the Rings: The Return of the King (2003)	✔	Green Street Hooligans (2005)	×
The Lord of the Rings: The Two Towers (2002)	✔	Ringers - Lord of the Fans (2005)	×
The Lord of the Rings: The Fellowship of the Ring (2001)	✔	The Lord of the Rings: The Return of the King (2003)	✔
		The Hobbit: An Unexpected Journey (2012)	✔
		Everything is Illuminated (2005)	×
		The Trust (2016)	×
		Deep Impact (1998)	×
		Grand Piano (2013)	×
		The Faculty (1998)	×
		Frodo Is Great... Who Is That?!! (2004)	×

Table A.12: Results and their relevance for query Q_{10}.

Q_{11}: *"In which movies directed by Garry Marshall was Julia Roberts starring?"*

MovieSearch results	relevant	**Valossa** results	relevant
Valentine´s Day (2010/I)	✔	Mother´s Day (2016)	✔
Pretty Woman (1990)	✔	Pretty Woman (1990)	✔
Runaway Bride (1999)	✔	Runaway Bride (1999)	✔
Mother´s Day (2016)	✔	Valentine´s Day (2010)	✔
		Soapdish (1991)	×
		Erin Brockovich (2000)	×
		New Year´s Eve (2011)	×
		Frankie and Johnny (1991)	×
		Maps to the Stars (2014)	×
		My Best Friend´s Wedding (1997)	×

Table A.13: Results and their relevance for query Q_{11}.

Q_{12}: *"Movie by Tim Burton with Johnny Depp as Todd."*

MovieSearch results	relevant	**Valossa** results	relevant
Sweeney Todd: The Demon Barber of Fleet Street (2007)	✔	Sweeney Todd: The Demon Barber of Fleet Street (2007)	✔
		Corpse Bride (2005)	×
		Ed Wood (1994)	×
		Edward Scissorhands (1990)	×
		Frankenweenie (2012)	×
		Batman (1989)	×
		Beetlejuice (1988)	×
		Big Fish (2003)	×
		Charlie and the Chocolate Factory (2005)	×
		Hansel and Gretel (1982)	×

Table A.14: Results and their relevance for query Q_{12}.

Q_{13}: *"Movie with a (IMDB) rating worse than 2.0 ."*

MovieSearch results	relevant	**Valossa** results	relevant
The Obama Effect (2012)	✔	The Day the Earth Stood Still (2008)	×
Unutursun diye (1986)	✔	40 Carats (1973)	×
Pledge This! (2006)	✔	The Loft (2014)	×
The Dead Will Rise 2 (2013)	✔	Murder on the Moon (1989)	×
Jóban rosszban (2005)	✔	Network (1976)	×
The 31st Annual People´s Choice Awards (2005)	✔	Devil Seed (2012)	×
Herd (2012)	✔	The Lucky One (2012)	×
Pirates of Ghost Island (2007)	✔	Slap the Monster on Page One (1972)	×
Sleepaway Camp IV: The Survivor (2012)	✔	Neighbors 2: Sorority Rising (2016)	×
Who Wants to Marry a Multi-Millionaire? (2000)	✔	Tjoet Nja´Dhien (1989)	×

Table A.15: Results and their relevance for query Q_{13}.

Q_{14}: *"Movie directed by Ridley Scott from the year 2000 with a rating better than 8,0 by more than 80,000 users."*

MovieSearch results	relevant	**Valossa** results	relevant
Gladiator (2000)	✔	– nothing found	

Table A.16: Results and their relevance for query Q_{14}.

Q_{15}: *"Drama Movie about a car accident in Los Angeles directed by Paul Haggis."*

MovieSearch results	relevant	**Valossa** results	relevant
Crash (2004/I)	✔	Crash (2005)	✔
		Answers to Nothing (2011)	×
		In the Valley of Elah (2007)	×
		Mulholland Drive (2001)	×
		Rain Man (1988)	×
		Babel (2006)	×
		The Woman Chaser (1999)	×
		Ser It Off (1996)	×
		Short Cuts (1993)	×
		San Andreas (2015)	×

Table A.17: Results and their relevance for query Q_{15}.

Q_{16}: *"Movie where Di Caprio uses cocaine."*

MovieSearch results	relevant	**Valossa** results	relevant
The Wolf of Wall Street (2013)	✔	The Basketball Diaries (1995)	✔
The Departed (2006)	✔	The Wolf of Wall Street (2013)	✔
The 66th Annual Golden Globe Awards (2009)	×	Rulers of the City (1976)	×
		True Romance (1993)	×
		The Consequences of Love (2004)	×
		The Departed (2006)	✔
		American Pop (1981)	×
		Alpha Dog (2006)	×
		Pain & Gain (2013)	×
		In Too Deep (1999)	×

Table A.18: Results and their relevance for query Q_{16}.

Q_{17}: *"Action Movie with Arnold Schwarzenegger where he fights with a sword."*

MovieSearch results	relevant	**Valossa** results	relevant
Conan the Destroyer (1984)	✔	Highlander (1986)	×
Red Sonja (1985)	✔	Blind Fury (1990)	×
		Kull the Conqueror (1997)	×
		Conan the Barbarian (1982)	✔
		The Beastmaster (1982)	×
		Batman & Robin (1997)	×
		Red Sonja (1985)	✔
		I (2015)	×
		The 13th Warrior (1999)	×
		Conan the Destroyer (1984)	✔

Table A.19: Results and their relevance for query Q_{17}.

Q_{18}: *"Movies where Keira Knightley sails on the Black Pearl."*

MovieSearch results	relevant	**Valossa** results	relevant
Pirates of the Caribbean: The Curse of the Black Pearl (2003)	✔	Pirates of the Caibbean: The Curse of the Black Pearl (2003)	✔
Pirates of the Caribbean: At World´s End (2007)	✔	Pirates of the Caibbean: Dead Man´s Chest (2006)	✔
Pirates of the Caribbean: Dead Man´s Chest (2006)	✔	Pirates of the Caibbean: At World´s End (2007)	✔
		A Dangerous Method (2011)	×
		The Edge of Love (2008)	×
		London Boulevard (2010)	×
		Never Let Me Go (2010)	×
		In the Loop (2009)	×
		Chris Rock: Kill the Messenger (2008)	×
		Princess of Thieves (2001)	×

Table A.20: Results and their relevance for query Q_{18}.

Q_{19}: *"Movie from 2014 where Liam Neeson plays an air marshal on a plane."*

MovieSearch results	relevant	**Valossa** results	relevant
Non-Stop (2014)	✔	Non-Stop (2014)	✔
		Left Behind (2014)	✕
		How to Build a Better Boy (2014)	✕
		Sharknado 2: The Second One (2014)	✕
		Airplane vs Volcano (2014)	✕
		Unbroken (2014)	✕
		A Million Ways to Die in the West (2014)	✕
		A Walk Among the Tombstones (2014)	✕
		The Nut Job (2014)	✕
		The Lego Movie (2014)	✕

Table A.21: Results and their relevance for query Q_{19}.

Q_{20}: *"Movie with Angelina Jolie and Brad Pitt where they have secrets."*

MovieSearch results	relevant	**Valossa** results	relevant
Mr. & Mrs. Smith (2005)	✔	George Wallace (1997)	✕
		Kung Fu Panda (2008)	✕
		Gia (1998)	✕
		A Mighty Heart (2007)	✕
		The International Criminal Court (2013)	✕
		Kung Fu Panda 3 (2016)	✕
		Kung Fu Panda 2 (2011)	✕
		Changeling (2008)	✕
		Playing by Heart (1998)	✕
		True Woman (1997)	✕

Table A.22: Results and their relevance for query Q_{20}.

Q_{21}: *"Movie about lying and forgery in which the protagonist is sent to Rome (it is a new version of the movie Plein soleil 1960)."*

MovieSearch results	relevant	**Valossa** results	relevant
The Talented Mr. Ripley (1999)	✔	Inspector Palmu´s Error (1960)	×
		Butterfield 8 (1960)	×
		Purple Neon (1960)	×
		Siege of Syracuse (1960)	×
		The Passionate Thief (1960)	×
		The Battle of the Sexes (1960)	×
		Goliath and the Dragon (1960)	×
		L´Avventura (1960)	×
		Spartacus (1960)	×
		Classe Tous Risques (1960)	×

Table A.23: Results and their relevance for query Q_{21}.

Appendix B

Proof and Lists

B.1 A 3-ary relation is not equivalent to 3 split binary relations.

Given three sets A,B,C and a relation $R \subset A \times B \times C$,
R is not in general equivalent to the triple of subrelations:

$$R_1 := \{(a,b) \in A \times B \mid \exists c \text{ such that } (a,b,c) \in R\}$$
$$R_2 := \{(a,c) \in A \times C \mid \exists b \text{ such that } (a,b,c) \in R\}$$
$$R_3 := \{(b,c) \in B \times C \mid \exists a \text{ such that } (a,b,c) \in R\}.$$

Proof: It suffices to give an example that contradicts equivalence in general. Consider for $a_1,a_2 \in A$, $b_1,b_2 \in B$ and $c_1,c_2 \in C$
the 3-ary relation $R := \{(a_1,b_1,c_2),(a_1,b_2,c_1),(a_2,b_1,c_1)\}$.
Splitting R to its binary subrelations leads amongst others
to the following tuples $(a_1,\ b_1) \in R_1$, $(a_1,\ c_1) \in R_2$ and $(b_1,\ c_1) \in R_3$.
Recombining those three binary tuples to a 3-ary relation results in the 3-tuple (a_1,b_1,c_1).
But $(a_1,b_1,c_1) \notin R$.

□

B.2 Lists

B.2.1 IMDB Files that are used

Sources for RDF fact triples:

Files with variable columns amount, minimum 2 and often 3
directors.list
cinematographers.list
composers.list
costume-designers.list
editors.list
producers.list
production-designer.list
writers.list
miscellaneous-companies.list
miscellaneous.list
genres.list
keywords.list
movies.list
countries.list
release-dates.list
production-companies.list
special-effects-companies.list
distributors.list
certificates.list
running-times.list

Table B.1: Files with tab separated columns.

movie-links.list

Table B.2: Files for finding versions and followers of movies.

aka-names.list
aka-titles.list
german-aka-titles.list
iso-aka-titles.list

Table B.3: Files for finding aliases.

actors.list
actresses.list
language.list
locations.list
ratings.list

Table B.4: Files used for producing 3-ary relations.

Here, separated lines with tags per movie are stored.

filename	used tags
business.list	BT (budget), GR (box office income), RT (rentals), AD (sold tickets), PD (production dates), ST (studio)
literature.list	BOOK (book), NOVL (novel), ADPT (adaption), OTHR
mpaa-ratings-reasons.list	RE (rating reason)

Table B.5: Tag lined files.

Sources for text documents:

plot.list
taglines.list
trivia.list
quotes.list
goofs.list
soundtracks.list

Table B.6: Files for which we create documents per movie.

B.2.2 Full list of relations with their preimage and image type

The types define what kind of nodes are connected by the correspondant relation. Table B.7 is ordered by display relevance for Relation Matching for hits with equal score.

link	***in-movie***	*movie*
link	***actor***	*person*
link	***character***	*person*
link	***has-nr-of-votes***	*integer*
link	***has-rating***	*float*
link	***at-location***	*word*
link	***at-scene***	*word*
link	***language-details***	*word*
link	***has-language***	*word*
movie	***directed-by***	*person*
movie	***produced-by***	*person*
movie	***written-by***	*person*
movie	***camera-by***	*person*
movie	***edited-by***	*person*
movie	***music-composed-by***	*person*
movie	***costumes-designed-by***	*person*
movie	***production-designed-by***	*person*
movie	***has-genre***	*word*
movie	***has-keyword***	*word*
movie	***distributed-by***	*word*
movie	***certified-by***	*word*
movie	***has-mpaa-rating-reason***	*word*
movie	***special-effects-by***	*word*
movie	***in-studio***	*word*
movie	***based-on***	*word*
movie	***from-country***	*word*
movie	***followed-by***	*movie*
movie	***follows***	*movie*
movie	***is-version-of***	*movie*
movie	***alternate-language-version-of***	*movie*
movie	***from-year***	*date*
movie	***released***	*date*
movie	***has-production-date-end***	*date*
movie	***has-production-date-start***	*date*
movie	***has-budget***	*float*
movie	***has-sold-tickets***	*integer*
movie	***has-runtime-in-min***	*integer*
movie	***has-box-office-income***	*float*
movie	***has-rentals***	*float*
any	***has-alias***	*alias*
any	***score***	*float*
any	***is-a***	*word*

Table B.7: All relations with preimage and image types.

B.2.3 List of used Stopwords

All the stopwords we filter out in some text processing parts to restrict to meaningful words:

a, about, above, after, again, against, all, am, an, and, any, are, aren't, as, at, be, because, been, before, being, below, between, both, but, by, can't, cannot, could, couldn't, did, didn't, do, does, doesn't, doing, don't, down, during, each, few, for, from, further, had, hadn't, has, hasn't, have, haven't, having, he, he'd, he'll, he's, her, here, here's, hers, herself, him, himself, his, how, how's, i, i'd, i'll, i'm, i've, if, in, into, is, isn't, it, it's, its, itself, let's, me, more, most, mustn't, my, myself, no, nor, not, of, off, on, once, only, or, other, ought, our, ours, ourselves, out, over, own, same, shan't, she, she'd, she'll, she's, should, shouldn't, so, some, such, than, that, that's, the, their, theirs, them, themselves, then, there, there's, these, they, they'd, they'll, they're, they've, this, those, through, to, too, under, until, up, very, was, wasn't, we, we'd, we'll, we're, we've, were, weren't, what, what's, when, when's, where, where's, which, while, who, who's, whom, why, why's, with, won't, would, wouldn't, you, you'd, you'll, you're, you've, your, yours, yourself, yourselves.

Appendix C

Resources used for programs

Webclient

- JSON
- JQuery 1.12.0
- JQuery UI 1.12.0
- Bootstrap 3.3.6 & Bootstrap -switch, -datepicker, -slider
- hover.css files for hover effects from `http://ianlunn.github.io/Hover/`

Suggestion Server

- googletest library gtest,
  from `https://github.com/google/googletest`
- C++11 library, compiled with g++ version 4.8.5 while using -O3 flag
- IMDB Data [**?**], TMDB Data and pictures [**?**]
- google custom search API for more posters [**?**]
- Stanford CoreNLP [**?**]

Other

- SPARQL Backend [**?**]
  at `https://github.com/Buchhold/SparqlEngineDraft`
- bottle.py server for movie and actor poster providing
  at `http://bottlepy.org/docs/dev/index.html`
- OBS Studio – software for the screen recordings of the user study.
  at `https://obsproject.com/`.
  For further information about the recordings of the user study or access contact `tobias.sommer.wt@gmail.com`.

List of Figures

List of Tables

Listings