Faceted Exploration of Multiple RDF Data Sources Using SPARQL

by

Hanief Bastian
(2217547)

Submitted in partial fulfillment
of the requirements
for the degree of

Master of Science in Computer Engineering

Supervisors:
Prof. Dr.-Ing. Jürgen Ziegler
Dipl.-Inform. Philipp Heim

University of Duisburg-Essen
Faculty of Engineering
Department Computer and Cognitive Sciences
Institute of Interactive Systems and Interaction Design
6 May 2009
Aufgabe der Abschlussarbeit im
ISE Bachelor-Masterstudiengang

für : Hanief Bastian
gestellt von : Prof. Jürgen Ziegler
Thema : Faceted Exploration of Multiple RDF Data Sources Using SPARQL

Aufgabenstellung:

This thesis is aimed to develop an application for faceted exploration of multiple RDF data sources in an efficient way.

Tasks are:

1) Develop prototype that enables faceted exploration of dbpedia.
2) Optimize the query performance (SPARQL).
3) Add other data sources (multiple).

Aufgabensteller/Betreuer

Zweitgutachter der Abschlussarbeit: 

Erklärung:
Ich declare that I have performed the work independently and without the use of foreign help.

The source and the used tools are fully mentioned. The text and images are made in each individual case.

Duisburg, den __________

Unterschrift des/der Studierenden
Abstract

Many applications that access the Semantic Web are structured in Three-Tier Architecture consisting of Client Tier, Server Tier, and Data Tier. With the growing number of SPARQL endpoints, parts of the data access logic have moved to the Data Tier. This allows the query building process to be shifted to the Client Tier and therewith ease the resource and the performance cost to access information contained in the Semantic Web.

In this thesis, we describe the transformation from a Three-Tier Architecture to a Two-Tier Architecture using the example of gFacet, a tool for graph based faceted access to the Semantic Web and we support the abilities of gFacet tool by generating efficient SPARQL queries on the client-side. The former Three-Tier Architecture of gFacet did not efficiently access the Semantic Web via SPARQL endpoint, mainly because the intermediate processing in the Server Tier could increase the total execution time. This was the reason to reconstruct the architecture as well as the whole query building process of the gFacet tool by moving all the functionalities of the application server into the Client Tier and improve the performance of the queries in order to support an efficient and client-side access to any SPARQL endpoint and thereby to various information contained in the Semantic Web.

We provide all the queries that allow faceted exploration on a large RDF dataset. In this thesis, we use an RDF dataset released by DBpedia. All the queries that support gFacet to search a certain concept, to retrieve and filter the information, and to change the information point-of-view are described in detail and evaluated regarding their performance. We implement two different approaches to retrieve a large amount of instances that enable paging through these instances; by retrieving all instances at once to the client using standard SPARQL and by retrieving a subset of the possible instances using SPARQL extensions.

We facilitate the functionality of gFacet by providing the opportunity to explore more than one RDF source. As an additional RDF dataset, we use RDF data released by MusicBrainz. With this feature, gFacet can search for more information by exploring the additional data source.

**Keywords:** SPARQL, gFacet, RDF, DBpedia, faceted exploration, Semantic Web, SPARQL query optimization, multiple SPARQL endpoints access
Acknowledgements

First I am so grateful to Allah SWT for the health, ideas, and everything that make this thesis accomplished. I also offer my sincerest gratitude to Prof. Dr.-Ing. Jürgen Ziegler and Dipl.-Inform. Philipp Heim. Without the guidance, the great efforts, and great ideas they have been given, this thesis would not have been completed or written. I simply could not wish for more supportive and friendlier supervisors.

I wish to express my warm thanks to my brothers at home Mas Andy, Dicky, and Evan. The encouragement they always give me is so meaningful to me.

Lastly, and most importantly, I wish to thank my parents, Papa Eddy Purwanto and Mama Latifah. They bore me, raise me, support me, and love me. Thank you for the great opportunity you gave me, so I can be here standing even though away from you both. To them I dedicate this thesis.

Hanief Bastian
May 2009
Duisburg-Germany
Contents

Chapter 1: Introduction ........................................................................................................... 1
   1.1. Motivation .................................................................................................................. 1
   1.2. Starting Point ............................................................................................................. 2
       1.2.1. Semantic Web ...................................................................................................... 2
       1.2.2. Resource Description Language (RDF) ............................................................ 3
       1.2.3. SPARQL ............................................................................................................... 4
       1.2.4. DBpedia Project ................................................................................................. 10
       1.2.5. MusicBrainz ....................................................................................................... 12
       1.2.6. DB Tune ............................................................................................................... 12
       1.2.7. Faceted Navigation ............................................................................................ 13
       1.2.8. gFacet Project .................................................................................................... 13
   1.3. Task Description ......................................................................................................... 1
   1.4. Related Work .............................................................................................................. 16
   1.5. Thesis Outline ........................................................................................................... 17

Chapter 2: The Architecture ................................................................................................. 18
   2.1. The Strategy ............................................................................................................... 18
   2.2. Client-side gFacet Architecture .............................................................................. 19
       2.2.1. SPARQL Query Dispatcher .............................................................................. 20
       2.2.2. SPARQL Result Parser .................................................................................... 21
       2.2.3. SPARQL Query Builder ................................................................................... 22

Chapter 3: Browsing DBpedia ............................................................................................. 24
   3.1. Dispatching a Query to DBpedia .............................................................................. 24
   3.2. Exploring DBpedia with gFacet ............................................................................. 25
       3.2.1. Searching for Concepts .................................................................................... 26
       3.2.2. Selecting the Initial Node .................................................................................. 28
       3.2.3. Expanding the Graph ........................................................................................ 34
       3.2.4. Filtering .............................................................................................................. 37
       3.2.5. Result Set Pivoting ............................................................................................ 43
   3.3. Evaluation .................................................................................................................. 45
3.4. Time Measurements ................................................................. 45
3.5. Correctness Measurements .................................................. 48

Chapter 4: Multiple Sources .............................................................. 52
4.1. The Strategy ............................................................................. 53
4.2. The Obstacles .......................................................................... 55
4.3. Finding the Equivalent Data .................................................... 55
4.4. Transforming the URIs ........................................................... 57
  4.4.1. Zitgist URI to DBTune URI Conversion .......................... 57
  4.4.2. MusicBrainz Scheme to Music Ontology Mapping .......... 58
4.5. Implementation ....................................................................... 59

Chapter 5: Conclusions & Future Works ........................................... 64
5.1. Conclusions ............................................................................ 64
5.2. Future Works .......................................................................... 65
  5.2.1. Autocompletion Text Search ......................................... 65
  5.2.2. Searching for Instances .................................................. 66
  5.2.3. Automatic Data Interlinking ............................................ 66
List of Figures

1.1. RDF graph representation .......................................................... 4
1.2. RDF graph example ................................................................. 5
1.3. Linking Open Data cloud ........................................................... 11
1.4. gFacet Architecture ................................................................. 14
1.5. gFacet user interface ................................................................. 15

2.1. gFacet Two-Tier Architecture ..................................................... 19
2.2. SPARQLQuery (the dispatcher) class diagram ............................... 20
2.3. RDFTerm class diagram ............................................................. 21
2.4. gFacet Data Flow Diagram ......................................................... 23

3.1. User action’s flow while browsing DBpedia ................................... 26
3.2. Searching a concept ................................................................. 27
3.3. Opening the initial node ............................................................. 29
3.4. The Relation List ........................................................................ 32
3.5. Constructing a pair of predicate and related concept ....................... 33
3.6. Opening a new node by selecting a relation ................................... 34
3.7. A chain of 4 nodes is created after user gradually expanding the nodes ... 35
3.8. Model a chain of 4 nodes describing parent-child characteristic .......... 36
3.9. A constraint selected in direct child of result set triggers a basic filtering . 38
3.10. Hierarchical filtering .................................................................. 39
3.11. Union filtering .......................................................................... 41
3.12. Intersection filtering .................................................................. 42
3.13. A chain of 4 nodes after pivoting ............................................... 43
3.15. Average elapsed time for nodes with certain amount of instances ....... 46
3.16. A chain of 4 nodes for evaluation .............................................. 47
3.17. Sample dataset: A chain of 4 nodes with Britpop Musical Group as the result set ................................................................. 48
3.18. Relationships diagram for the sample dataset ........................................ 49

4.1. Data sets that have been published and interlinked by Linking Open Data project (March 2009) ................................................................. 53
4.2. gFacet model of multi sources exploration .......................................... 54
4.3. RDF graph of a subject with three equivalent resources ....................... 56
4.4. Two equivalent resources from (a) dbPedia, (b) MusicBrainz ............... 57
4.5. Extracting the class and UUID of MusicBrainz to be mapped to DBTune URI scheme ............................................................. 58
4.6. Data flow diagram of gFacet multi sources exploration ....................... 59
4.7. Relation list with sameAs:musicBrainz element .................................. 61
4.8. A chain of two nodes from distinct sources........................................ 61
List of Listings

<table>
<thead>
<tr>
<th>Listing</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.</td>
<td>N3 statements</td>
<td>3</td>
</tr>
<tr>
<td>1.2.</td>
<td>RDF/XML serialization</td>
<td>4</td>
</tr>
<tr>
<td>1.3.</td>
<td>Simple query obtaining a name and mbox of a FOAF profile</td>
<td>5</td>
</tr>
<tr>
<td>1.4.</td>
<td>A query obtaining a name and mbox of a FOAF profile with value constraint</td>
<td>6</td>
</tr>
<tr>
<td>1.5.</td>
<td>A query obtaining a name and mbox of a FOAF profile with OPTIONAL clause</td>
<td>6</td>
</tr>
<tr>
<td>1.6.</td>
<td>Simple query getting a name and mbox of a FOAF profile with group graph pattern</td>
<td>7</td>
</tr>
<tr>
<td>1.7.</td>
<td>Example of COUNT clause</td>
<td>8</td>
</tr>
<tr>
<td>1.8.</td>
<td>Example of subquery</td>
<td>8</td>
</tr>
<tr>
<td>1.9.</td>
<td>SPARQL query results in XML format</td>
<td>8</td>
</tr>
<tr>
<td>2.1.</td>
<td>HTTP trace of query dispatching</td>
<td>21</td>
</tr>
<tr>
<td>2.2.</td>
<td>Native Actionscript’s datatypes representing the query results</td>
<td>22</td>
</tr>
<tr>
<td>3.1.</td>
<td>Dispatching a SPARQL query to DBpedia</td>
<td>25</td>
</tr>
<tr>
<td>3.2.</td>
<td>Query for concept searching</td>
<td>28</td>
</tr>
<tr>
<td>3.3.</td>
<td>Query to retrieve all instances</td>
<td>30</td>
</tr>
<tr>
<td>3.4.</td>
<td>Query to retrieve a subset of instances</td>
<td>31</td>
</tr>
<tr>
<td>3.5.</td>
<td>Query for obtaining a total number of possible solutions</td>
<td>31</td>
</tr>
<tr>
<td>3.6.</td>
<td>Query for obtaining all the pairs of predicate and concept</td>
<td>33</td>
</tr>
<tr>
<td>3.7.</td>
<td>Query for a chain of 4 nodes; The result set is the initial node</td>
<td>36</td>
</tr>
<tr>
<td>3.8.</td>
<td>Query for basic filtering. Constraint : footballer Thomas Hitzlsperger</td>
<td>38</td>
</tr>
<tr>
<td>3.9.</td>
<td>Query for hierarchical filtering. Constraint : English Club C4</td>
<td>40</td>
</tr>
<tr>
<td>3.10.</td>
<td>Query for union filtering. Constraint : English Club C2 and C4</td>
<td>41</td>
</tr>
<tr>
<td>3.11.</td>
<td>Query for intersection filtering. Constraint : English Club D1 and C4</td>
<td>42</td>
</tr>
<tr>
<td>3.12.</td>
<td>Query for result set pivoting. Result set : node C</td>
<td>44</td>
</tr>
</tbody>
</table>
4.1. Query to get all relations and count the interlinked data of a concept ........ 60
4.2. Dispatching a SPARQL query to DBTune for MusicBrainz dataset ........ 62
4.3. Query to get the interlinked data from DBtune for MusicBrainz dataset ... 62

5.1. Query to search for instances ................................................................. 66
List of Tables

1.1. SPARQL result for simple query ......................................................... 6
1.2. SPARQL result with OPTIONAL clause .................................................. 7

3.1. The instances of each evaluation node .................................................... 46
3.2. Average of elapsed time between HTTP request and response for a chain
     of 4 nodes .......................................................................................... 47
3.3. Result of OR-operation test cases ........................................................... 50
3.4. Result of AND-operation test cases ......................................................... 51

4.1. Comparison MusicBrainz entity’s type and Music Ontology class .............. 58
Chapter 1

Introduction

1.1. Motivation

The Semantic Web [1] was introduced to extend the power of the current Web by making its content understandable to machines and thus allow machines to perform automated information gathering and to obtain more meaningful results. This requires the semantics of content in the Web to be described in a machine-readable form by using formal languages like RDF [14] and OWL [20]. These languages allows web content to be assigned to semantically defined concepts and related to one another by semantically defined relationships. That way annotated, information can be found more efficiently and with more certainty.

With the steady growth of the Semantic Web, more and more annotated information is published on the Web, leading to a growing number of RDF datasets. Even though RDF data is originally meant to be read by machines, information about the meaning of Web content and its interrelations can be highly valuable for humans, too. However, there is no defined method to render RDF data in a way that can be easily understood by humans, in comparison to, for example, HTML [15], where markups are used for a proper presentation. In order to let also humans benefit from information contained in RDF datasets, methods are needed to access and to render this information in an appropriate way.

One promising way to access information that is contained in RDF data is offered by the tool gFacet [11]. It combines graph-based visualization with faceted filtering functionalities to build up queries and thus control what information is displayed on the screen. The queries are formulated in SPARQL [9], the W3C recommendation to access RDF data. SPARQL has a SQL-like syntax and can be used to express queries across diverse data sources.

With the existing SPARQL endpoints around the Web that allow RDF dataset to be queried in order to get the results, gFacet can be the bridge between user and the RDF dataset. Efficient and accurate queries need to be implemented into gFacet, so that gFacet can be a powerful faceted RDF browser. But in doing this, we also have to find
the correct architecture for gFacet. The architecture has to be simple but efficient and supports direct connection with SPARQL endpoint in order to enable active accessing to any SPARQL endpoint.

In general, the main focus of this thesis is given as follows

- Building an efficient architecture for gFacet. We find that Two Tier architecture is suitable for gFacet. With Two-Tier Architecture, gFacet tool in the Client Tier can directly communicate with a SPARQL endpoint.

- Optimize the query performance of the current gFacet by building new efficient and accurate queries in order to support faceted exploration over a large RDF dataset.

- Allowing multi RDF sources exploration using gFacet. With this feature gFacet can browse an entity from a source to the same entity to the other sources.

### 1.2. Starting Point

This section gives brief description about the platforms, technologies, or projects that are used as the foundation of the thesis. We adapt this foundation in order to make it applicable to our work.

#### 1.2.1. Semantic Web

Since the WWW began in the early 1990s, WWW has given a great impact for mankind in information, education, business, and even social life. From time to time the number of websites on the web keeps growing. According to Google Blog\(^1\), the Google index reached 1 trillion unique URLs on the web by the end of July 2008, for comparison that the first Google index in 1998 had 26 million pages. However, most of web pages currently are still in the form what we called the Syntactic Web. The syntactic web focuses only on the visual presentation of the content. Once the content been displayed, it is up to the user to interpret the meaning.

But this trend has already begun to change since Sir Berners-Lee introduced the term of Semantic Web in his article *Semantic Web Roadmap* in 1998 [1] and the following *The Semantic Web* in 2001 [2]. He was wondering what if machines can “talk” and change information available around the Web to each other. This idea can be done by making the semantic of the information understandable to the machines — which is the main goal of the Semantic Web. The Semantic Web will enable machines to comprehend semantic documents and data, not human speech and writings [2].

\(^1\) [http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html](http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html)
However Semantic Web is not designed to replace the Web of today but to improve it, so the next generation of Web is accessible both to human and machines.

### 1.2.2. Resource Description Language (RDF)

The Resource Description Language (RDF) is a general-purpose language for representing information about resources in the Web. It is particularly intended for representing metadata about Web resources, but it can also be used to represent information about objects that can be identified on the Web, even when they cannot be directly retrieved from the Web [3].

RDF allows semantics to be expressed in a way, so that the information can be processed by applications also, rather than being only displayed to the users. Basically, RDF defines a data model for describing machine-processable semantics of data [4]. The basic data model consists of three objects:

- **Resources.** A resource may be an entire Web page, a part of a Web page, a whole collection of pages, or an object that is not accessible via the Web (e.g., a printed book). Resources are always named by URIs.
- **Properties.** A property is a specific aspect, characteristic, attribute, or relation used to describe a resource.
- **Statements.** A specific resource, together with a named property plus the value of that property for that resource, constitutes an RDF statement. These three individual parts of a statement are called, respectively, the *subject*, the *predicate*, and the *object* of the statement.

RDF statements can be represented in a triple notation called N3, in RDF/XML serialization, and as a graph of triples.

Let us start with an example of statements in natural language

*Kaka plays for AC Milan.*

*Kaka has jersey number 22.*

*AC Milan has a website accessible at http://www.acmilan.com/*.

In N3, the statements are presented as follows

```plaintext

<http://www.example.org/Kaka> <http://www.example.org/hasJerseynumber> "22" .

```

Listing 1.1. N3 statements
1. Introductions

Listing 1.2 shows the same example in RDF/XML

```xml
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ex="http://www.example.org/">
    <rdf:Description rdf:about="http://www.example.org/Kaka">
        <ex:playsFor rdf:resource="http://www.example.org/ACMilan"/>
        <ex:hasJerseynumber>22</ex:hasJerseynumber>
    </rdf:Description>
    <rdf:Description rdf:about="http://www.example.org/ACMilan">
        <ex:hasWebsite rdf:resource="http://www.acmilan.com/"/>
    </rdf:Description>
</rdf:RDF>
```

Listing 1.2. RDF/XML serialization

Figure 1.1 presents the example in graph

The resource subjects and objects are drawn as ellipses, the literal object as a square, and the properties as labeled-directed arcs.

RDF Schema is a simple set of standard RDF resources and properties to enable people to create their own RDF vocabularies. The data model expressed by RDF Schema is the same data model used by object-oriented programming languages like Java. The data model for RDF Schema allows you to create classes of data [5].

1.2.3. SPARQL

RDF is the foundation of the Semantic Web. It is expected that in the future more and more open RDF datasets are released. Allowing easy access to these collections requires a query language that able to execute against RDF data.

Since January 2008, RDF Data Access Working Group (DAWG) of the World Wide Web Consortium (W3C) has released a query language for RDF called SPARQL [9]; a recursive acronym stands for **SPARQL Protocol and RDF Query Language**.
SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions [9]. SPARQL is also considered as a component of the Semantic Web.

Making Simple Query

Most of SPARQL queries contain a set of triple pattern called *basic graph pattern* (BGP). Each part of the triples acts as a subject, a predicate, or an object which may be explicitly defined to a resource or literal or as a variable. SPARQL variables are prefixed either with ? or $.

An example of RDF data presented as an RDF graph is shown below

![RDF graph example](image)

A query to find person with a given name and email address is executed against the given RDF data

```sparql
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE
  { ?x foaf:name ?name .
    ?x foaf:mbox ?mbox
  }
```

Listing 1.3. Simple query obtaining a name and mbox of a FOAF profile

The PREFIX clause defines a namespace for the FOAF\(^2\) location. The SELECT clause specifies what the query should return in this case variable name and mbox. WHERE clause provides the basic graph pattern to match against the data.

\(^{2}\)\url{http://xmlns.com/foaf/spec/}
The query matches the graph pattern of the query to the data model. The result of this query is shown in Table 1.1. A solution sequence consists of one or multiple solution if there is a match between the graph pattern and the model, or zero solution if there is no matching pair.

<table>
<thead>
<tr>
<th>name</th>
<th>mbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Bobby Iceman&quot;</td>
<td><a href="mailto:boobyiceman@example.org">mailto:boobyiceman@example.org</a></td>
</tr>
<tr>
<td>&quot;Tony Ironman&quot;</td>
<td><a href="mailto:tonyironman@example.org">mailto:tonyironman@example.org</a></td>
</tr>
</tbody>
</table>

Table 1.1. SPARQL result for simple query

**Value Constraints**

FILTER is an optional clause to restrict solutions to those for which the filter expression evaluates to TRUE.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE
  { ?x foaf:name ?name .
    FILTER regex(?name, "^Bobby")
  }
```

Listing 1.4. A query obtaining a name and mbox of a FOAF profile with value constraint

The query above will give “Bobby Iceman” and his mailbox <mailto:bobbyiceman@example.org> as the solution.

**Including Optional Values**

SPARQL query will give a non-empty solution sequence only if every query pattern matches to the data model. Unfortunately, if at least a query pattern fails to match the model, then the entire query will give an empty solution sequence. So, it is useful to have query patterns that still allow the query to provide bindings even if a part of the query pattern fails to match the data model. OPTIONAL clause gives this feature: even if the optional part does not create any binding, it does not eliminate the solution.

```
SELECT ?name ?mbox
WHERE
  {
    OPTIONAL ( ?x foaf:name ?name )
  }
```

Listing 1.5. A query obtaining a name and mbox of a FOAF profile with OPTIONAL clause
The query above tries to find all the email address no matter it has the person’s name or not, as shown in Table 1.2.

<table>
<thead>
<tr>
<th>name</th>
<th>mbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Bobby Iceman&quot;</td>
<td><a href="mailto:boobyiceman@example.org">mailto:boobyiceman@example.org</a></td>
</tr>
<tr>
<td>&quot;Tony Ironman&quot;</td>
<td><a href="mailto:tonyironman@example.org">mailto:tonyironman@example.org</a></td>
</tr>
<tr>
<td></td>
<td><a href="mailto:ororostorm@example.org">mailto:ororostorm@example.org</a></td>
</tr>
</tbody>
</table>

Table 1.2. SPARQL result with OPTIONAL clause

**Group Graph Pattern**

Group graph pattern can consist zero, one, or multiple basic graph patterns. Group graph pattern is delimited with curly braces {}. The query in Listing 1.6 can be rewritten into a query in Listing 1.3 that groups the triple patterns into two basic graph patterns. Even both of queries have different structure; they give the same solution sequence.

```sparql
SELECT ?name ?mbox
WHERE
{ { ?x foaf:name ?name . }
  { ?x foaf:mbox ?mbox . }
}
```

Listing 1.6. Simple query getting a name and mbox of a FOAF profile with group graph pattern

An extensive explanation of SPARQL syntax and semantics can be found on SPARQL Query Language for RDF document [9].

**SPARQL Extensions**

There are a number of limitations in current SPARQL version, such as SPARQL is read-only and cannot modify RDF dataset, it does not support subqueries and aggregate functions, and so on. However, Openlink Virtuoso\(^3\) provides some extensions for SPARQL in order to overcome the limitations above.

In this thesis only SPARQL extension for subqueries and aggregate function COUNT will be explained, because these extensions are intensively used in the thesis.

**COUNT function:** COUNT function provides a function to count the number of the solutions satisfying the criteria specified in the WHERE clause. With the count aggregate the argument may be either * that means counting all rows, or a variable name that means counting all the rows where this variable is bound. There can be an

\[^3\](http://www.openlinksw.com/virtuoso/)
optional `distinct` keyword before the variable that is the argument of an aggregate. An example can be seen in Listing 1.7. The example returns the count the amount of variable `o` for each distinct `p`.

```
select ?p count (?o)
from <http://mygraph.com>
where {?s ?p ?o};
```

Listing 1.7. Example of COUNT clause

**Subquery extension**: Subquery or Inner query or Nested query is a query inside a query. It is usually used for a complex computation that cannot be done by using only one query. In SPARQL, subquery is added inside the WHERE clause of the query.

For example, one use case was taking all the teams in the database and for all with over 5 members, add the big_team class and a property for member count.

```
construct { ?team a big_team . ?team member_count ?ct } 
where {
  ?team a team .
  { select ?team2 count (*) as ?ct
    where { ?m member_of ?team2 }.
    filter (?team = ?team2 and ?ct > 5)
  }
}
```

Listing 1.8. Example of subquery

**SPARQL Query Results XML Format**

Most of SPARQL processors provide the SPARQL query result in a various document format, so it allows programmers to choose the most convenient format for their application. To make the result serializable to any application, W3C recommends SPARQL Query Results XML Format [10], so that the returned result set is written as an XML document.

The SPARQL results in XML document of the query in Listing 1.3 is shown below:

```
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="name"/>
    <variable name="mbox"/>
  </head>
  <results ordered="false" distinct="false">
    <result>
      <binding name="name">
```
Listing 1.9. SPARQL query results in XML format

SPARQL results document begins with `<sparql>` document definition and a namespace -- http://www.w3.org/2005/sparql-results# -- where all of the key elements belong to. Inside the `<sparql>` element there are two sub-elements, `<head>` and a results element which can be `<results>` for SELECT queries or `<boolean>` for ASK queries. The `<head>` element declares all the variables returned on the result set. These variables are the same like the variables declared in the SELECT clause of the query. If we see back to Table 1.1, the variables are equivalent to the column heading.

The `<results>` section contains solution sequence (a set of query solutions). Each query solution is stored in the sub-element `<result>`. Every `<result>` element corresponds to every row in Table 1.1. Every `<result>` element contains one or more `<binding>` element with a name element property defining the bound variable.

The value of a query variable binding, which may be a resource/URI, a string literal, a typed literal, or a blank node, is included as the content of the `<binding>` as follows:

RDF URI Reference U

```xml
<binding><uri>U</uri></binding>
```

RDF Literal S

```xml
<binding><literal>S</literal></binding>
```

RDF Literal S with language L

```xml
<binding><literal xml:lang="L">S</literal></binding>
```

RDF Typed Literal S with datatype URI D

```xml
<binding><literal datatype="D">S</literal></binding>
```
1. Introductions

Blank Node label I

<binding><bnode>I</bnode></binding>

If a variable is unbound, there will be no <binding> element for that variable included in the <result> element.

1.2.4. DBpedia Project

The DBpedia project [6] is a community effort to extract structured information from Wikipedia and to make this information available on the Web. DBpedia allows you to ask sophisticated queries against Wikipedia and to link other datasets on the Web to Wikipedia data. The goals of DBpedia projects are to convert Wikipedia content to a large, multi-domain RDF dataset, which can be used further in Semantic Web applications, to interlink DBpedia dataset with other open datasets creating a large Web of open data, and to develop interfaces so Web services can make use of DBpedia dataset.

The DBpedia project extracts various kinds of structured information from Wikipedia, such as infobox templates, categorization information, images, geo-coordinates, and links to external websites [7]. Since DBpedia 3.2, the new infobox extraction method was introduced to create DBpedia ontology.

The DBpedia dataset currently consists of around 274 million RDF triples, which have been extracted from Wikipedia editions in 14 languages. The DBpedia knowledge base currently describes more than 2.6 million things, including at least 213,000 persons, 328,000 places, 57,000 music albums, 36,000 films, 20,000 companies. It features labels and short abstracts for these things in 14 different languages; 609,000 links to images and 3,150,000 links to external web pages; 4,878,100 external links into other RDF datasets, 415,000 Wikipedia categories, and 75,000 YAGO categories.

The DBpedia dataset can be accessed online by querying via a public SPARQL query endpoint at http://dbpedia.org/sparql, hosted by Virtuoso, or by browsing as Linked Data using Semantic Web browsers like Disco, Tabulator, or Marbles.

The DBpedia dataset is interlinked with various open dataset on the Web using RDF links, this enable DBpedia users to discover information starting from a resource in DBpedia dataset to related data within other sources. RDF links utilization between related resources creates a giant Web-of-Data which within September 2008 consists of approximately 2 billion RDF triples. Figure 1.3 gives an overview of the Web of interlinked data.

The Web-of-Data enables users to navigate for example from a resource of a musical band in DBpedia dataset to a list of their songs in Musicbrainz or to a list of reviews of the band in Revyu.
1. Introductions

The example RDF link below connects a URI of Portishead in DBpedia to a related URI in Musicbrainz:

```
<http://dbpedia.org/resource/Portishead> owl:sameAs <http://zitgist.com/music/artist/8f6bd1e4-fbe1-4f50-aa9b-94c450ec0f11>
```

![Linking Open Data cloud](image)

**Figure 1.3. Linking Open Data cloud**

DBpedia provides three different classification schemata for things.

1. **Wikipedia Categories** are represented using the SKOS vocabulary[^4].

2. The **YAGO Classification**[^5] is derived from the Wikipedia category system using Word Net.

3. **Word Net Synset Links** were generated by manually relating Wikipedia infobox templates and Word Net synsets, and adding a corresponding link to each thing that uses a specific template. In theory, this classification should be more precise than the Wikipedia category system.

[^4]: http://www.w3.org/2004/02/skos/
[^5]: http://www.mpi-inf.mpg.de/yago-naga/yago/

---

[^4]: http://www.w3.org/2004/02/skos/
[^5]: http://www.mpi-inf.mpg.de/yago-naga/yago/
These classifications enable users to select a specific thing mentioned in the executed SPARQL queries.

### 1.2.5. MusicBrainz

MusicBrainz\(^6\) is a community music metadatabase that attempts to create a comprehensive music information site. MusicBrainz collects this information about recordings and makes it available to the public. Users can contribute their knowledge about music which then can be shared with others.

The music metadata can consist of all data about Artists, Releases, Tracks, Labels, and advance relationship among them. This metadata is stored in a Postgresql relational database engine.

MusicBrainz has URI schemes to identify their entities, such as

- http://musicbrainz.org/artist/UUID
- http://musicbrainz.org/release/UUID
- http://musicbrainz.org/track/UUID
- http://musicbrainz.org/label/UUID

where UUID is a Universally Unique Identifier\(^7\) in its 36 character ASCII representation.

### 1.2.6. DBTune

DBTune\(^8\) hosts a number of servers, providing access to music-related structured data, in a Linked Data fashion. DBTune provides all the data based on open Web standards such as RDF and SPARQL.

Various datasets has been provided by DBTune, including MusicBrainz data. DBTune maps this MusicBrainz data based on Music Ontology [28]. And now MusicBrainz data is available via SPARQL endpoint, http://dbtune.org/musicbrainz/sparql, powered by D2R server. The basic graph of MusicBrainz located at http://musicbrainz.org/.

---

\(^6\) [http://musicbrainz.org/](http://musicbrainz.org/)


\(^8\) [http://dbtune.org/](http://dbtune.org/)
1.2.7. Faceted Navigation

Data on the Semantic Web is semi-structured and does not follow one fixed schema [8]. Faceted navigation [12] is an exploratory interface suitable for such data. Facets refer to categories used to characterize information items in a collection [27]. By categorizing data into facets, the exploration takes place when the user selects any restriction values of the facets in order to filter the result set.

There are two possible methods to map subject-predicate-object RDF triples into facets. First, facets are computed by the predicate that connects two resources; information elements are RDF subjects, facets are RDF predicates and restriction-values are RDF objects [8]. Second, facets are computed from all resources that related to resources in the result list and grouped by their specific characteristics or concepts; using predicate rdf:type or skos:subject for example. [13] implemented the latest method in their browser interface. An example implementing these methods is explained below.

A result list of people related to Formula 1 racing can have predicates such as first team, current team, former team, or last team that connect to a group of racing teams, and predicates such as lives in, born in, or died in that connect to a group of countries. With the first method, there will be 7 facets constructed from the predicates. But, only facets racing team and country would appear if the second method is applied.

A faceted interface has several advantages over keyword search or explicit queries: it allows exploration of an unknown dataset since the system suggests restriction values at each step; it is a visual interface, removing the need to write explicit queries; and it prevents dead-end queries, by only offering restriction values that do not lead to empty results [9].

1.2.8. gFacet Project

gFacet [11] is a browsing approach that supports the exploration of RDF datasets by combining graph-based visualization with faceted filtering functionalities. With this combination, gFacet facilitates to explore of large and highly interrelated RDF datasets. The major aims of the approach are:

1. **Prevention of an over-cluttered graph:** The facet-based visualization groups the instances of data into separate facets according to their characteristics. Rather than visualizing each relation of an instance to any single instance by an edge, the facet-based visualization allows to visualize the relationship between an instance of a facet and one or more instances in the other facet only with a single edge, and this relation between instances only indirectly visible when certain facet in an instance get selected by the user in order to filter the result set.
2. **Representation of relations between facets:** Visualizing the information as a graph – facets as nodes and the relations between them as labeled edges – can make the hierarchy of the information more understandable.

3. **Single coherent visualization:** A graph-based visualization prevents the users from getting “lost in hyperspace” by displaying all the information in a single visualization instead of being visualized over several screens or windows.

**The Architecture**

gFacet is built on a Three-Tier Architecture consists of client tier, server tier, and data tier.

![Figure 1.4. gFacet Architecture](image)

The first tier of Three-Tier Architecture is the Client Tier in which gFacet user interface is displayed. gFacet is implemented using Adobe Flex\(^9\) – a framework for creating Rich Internet Applications (RIAs) based on Adobe Flash\(^10\) platform. RIAs created with Flex can run in every browser installed with Adobe Flash player. So this makes gFacet to be an interactive RDF data browser that can run in every operating system as long as it has a browser and Adobe Flash player installed.

The Server Tier is the application server – also called the middleware – where the application logic and server software are stored. The middleware is implemented in PHP. It provides the logic of query generation. It generates queries according to user tasks and sends back the query result to the client tier. Because gFacet is a Flash-based user interface, AMFPHP\(^11\) is used to serialize the communication between gFacet and the PHP class objects on the server. To be able to get the data from a relational database and transform it into an RDF model, RAP (RDF API for PHP)\(^12\) package is used. RAP

---

1. Introductions

is a software package for parsing, querying, manipulating, serializing and serving RDF models. Then the available RDF models are manipulated using RAP’s SPARQL package.

The Data Tier is where the physical data is served for the application. The RDF data is stored in a relational database.

The Prototype

gFacet can be accessed via the internet using a browser with a Flash player installed. In the prototype version, gFacet uses a sample of dataset form the field of music.

Initially, a node contains a list of songs is displayed. The node can be expanded to other related node by selecting a pair of a relationship and a facet to which it refers, from a dropdown menu on the bottom of the node. If a user selects a pair from the dropdown menu, a new node is opened and gets connected to the original node by a labeled edge. The instances of the new node can act as a filter for the instances of the connected node. Expanding the nodes gradually can create a collection of hierarchical facets as illustrated in Figure 1.5.

Figure 1.5. gFacet user interface
1.3. Task Description

In this Thesis, we focus on topics of accessing and querying RDF data on an arbitrary RDF sources with SPARQL and applying it in a faceted RDF browser. We use gFacet as the platform to display the information to the users after the data manipulation in faceted manner has been performed. But in order to make gFacet performance better in the case of data access and availability, we need to do some modifications on the gFacet architecture and then rebuild the SPARQL queries being used.

The Three-Tier Architecture of gFacet has drawbacks in the case of resource utilization and execution time. The idea to prevent these issues is to make gFacet does all computation of its logic full in the client side, so gFacet can directly access any SPARQL endpoint without requires any server in between. We will move all the functionalities of AMFPHP, the application logics, and RAP packages by rebuilding similar functionalities in the Client Tier.

The second task is to build new queries that will improve the gFacet performance. The queries should be efficient and accurate so it can support gFacet accessing a large RDF dataset in faceted manner.

The main SPARQL endpoint and RDF dataset we use in this thesis is the one that has been released by DBpedia Project. Because DBpedia is a large, multi domain RDF dataset extracted from Wikipedia, so it makes DBpedia to be a good source of information. We need to build queries that allow user to browse DBpedia in effective and efficient way. And then we should evaluate the performane of the queries by measuring the time required to execute the queries and measuring the accuracy.

Since DBpedia is also interlinked to various RDF sources around the Web, it allows the user to jump from a resource in DBpedia to a related resource to an RDF dataset in another source by following the given RDF link. A nice example is that DBpedia is interlinked to the MusicBrainz datasets. For the third task in this thesis, we try to adjust gFacet to be able to follow RDF links from DBpedia to MusicBrainz, so that the users can explore the data from both sources like they were browsing from only one huge dataset.

1.4. Related Works

SPARQL has been recommended by W3C as a standard language for querying RDF datasets. Many approaches has been investigated how to query RDF data using SPARQL.

Erling [21] investigated sample of SPARQL queries to be executed against large datasets using Openlink Virtuoso SPARQL engine. The queries consist of SPARQL extensions that work with Virtuoso at the back-end.
Many other researches try to optimize the performance of SPARQL. [23], [26], and [24] implemented different approaches to optimize the query, but these researches suggested query reordering in order to get an optimized query execution plans.

Even though more and more data available over SPARQL endpoints, however it is still difficult to integrate data from multiple data sources. RDF data integration is often done by loading all data into single repository and querying the merged data locally [22]. Tabulator [16] uses this approach. Tabulator collects all the information by following the related resources indicated by owl:sameAs or rdfs:seeAlso predicate and stores it in the local repository. Tabulator allows user to query against the locally-stored data.

Quilitz and Leser [22] built DARQ\textsuperscript{13}, a query engine for federated SPARQL queries. It provides transparent query access to multiple endpoints. The implementation introduces service description that provides the declarative descriptions of the data available from each endpoint, which will be used to determine the endpoint a query should be sent to.

### 1.5. Thesis Outline

This thesis is organized as follows

**Chapter 2** explains how to make gFacet into a full client-side application. It starts with the main strategy and then explains the new gFacet architecture and new components that have been built during the thesis.

**Chapter 3** explains in details how the generated queries work but it will give explanation step by step according to the user action while browsing DBpedia dataset using gFacet. At the end, this chapter will show the evaluation result based on time measurement and accuracy measurement.

**Chapter 4** describes how to make gFacet able to use more than one RDF source. In here, gFacet will be set to be able to execute dataset both from DBpedia and MusicBrainz.

**Chapter 5** will give some short summary of the implementation and evaluation of this thesis. This chapter also introduces some ideas that can be foundation for improvements of gFacet in the future.

\textsuperscript{13} Distributed ARQ, as an extension to ARQ (http://jena.sourceforge.net/ARQ)
Chapter 2

The Architecture

The current gFacet works on a Three Tier Architecture consisting the Client Tier, The Server Tier, and Data Tier. The Client Tier is where the user interface and the presentation logic reside. The AMFPHP\(^1\), the application logic written in PHP\(^2\), and the RAP\(^3\) packages are located in the Middle Tier. And the physical relational database is placed in the Data Tier.

With this kind of architecture, there are two issues to be considered.

- The Three Tier Architecture is resource expensive. Using middleware means more resources, such as more dedicated machines, more space or working memory usages, is necessary in addition to a database server.

- It is relatively time consuming. Instead of directly communicate with the Data Tier, an additional processing in the Server Tier needs to be done. This means the total execution time will increase eventually.

These issues trigger us to make gFacet has better performance by making gFacet to be a client-side application. This chapter explains the strategy to make it possible. The structure of this chapter is as follows. Section 2.1 describes the strategy in general. Section 2.2 explains about the new architecture and the implementation.

2.1 The Strategy

The problems of the gFacet architecture are located in the Server Tier. There are too much intermediate processing before user’s commands can be processed in the Data Tier. The main focus of the strategy is the PHP applications logic and the RAP packages in the Server tier. Since the PHP applications logic plays the important role that it is where the queries are built. And the RAP packages play the role as the query dispatcher

\(^1\) http://www.amfphp.org/

\(^2\) http://www.php.net/

\(^3\) http://www.seasr.org/wp-content/plugins/meandre/rdfapi-php/doc/
and as a parser for the result from the database server. We can put aside the AMFPHP because this package is only used to serialize the communication between gFacet and the middleware.

Mainly, there are three steps that have to be done to make gFacet a full client-side application.

1. We have to rebuild the query builder into the Client Tier; this will replace the PHP application logic in the server.
2. In order to replace the RAP packages, we have to build a query dispatcher that will send the query to SPARQL endpoint; and
3. Build a parser for the result returned by the SPARQL endpoint.

Since gFacet is implemented using Adobe Flex\(^4\), all the new components will be built in a client-side scripting language, Actionscript\(^5\), as the core scripting language of Flex. By this implementation, the application logic will fully run in the client.

### 2.2 Client-side gFacet Architecture

By accomplishing all the three steps, mentioned in the previous section, we have simplified the architecture of gFacet. We have moved all the necessary functionalities such as query building and dispatching, and result parsing into the Client Tier. So now the new gFacet will completely run in the client-side. And this means that gFacet is now built on a Two-Tier Architecture consisting Client Tier and Data Tier as illustrated in Figure 2.1. With this architecture user’s request can be invoked to the SPARQL engine without any intermediate processing in between.

![gFacet Two-Tier Architecture](image)

Figure 2.1. gFacet Two-Tier Architecture


\(^5\) [http://www.actionscript.org/](http://www.actionscript.org/)
In the next subsections, we will explain briefly the three architecture components which we develop during the thesis; the query builder, the query dispatcher, and the SPARQL result parser. The development of presentation handler, the action listener, and also the SPARQL endpoint is out of scope of this thesis, because they have been established even before the thesis started.

### 2.2.1 SPARQL Query Dispatcher

A SPARQL endpoint allows SPARQL query to be conveyed as a HTTP request over the Web using a GET or POST method. This HTTP request is assembled and sent to the SPARQL endpoint by the query dispatcher. The request package contains some parameters that required by the SPARQL endpoint. The parameters are given as follows:

1. **query** specifies the query pattern will be executed.

2. **default-graph-uri** which specifies the graph to be used to form the default graph. Specifying this parameter will overwrite the defined default graph in the query pattern using FROM clause.

3. **output** which specifies the result format to be returned. In this application we expect an XML document of the SPARQL query result.

The query dispatcher is derived from the `HTTPService` class of the Actionscript, as shown in Figure 2.2. The `send()` method of the `HTTPService` object is able to send a HTTP request to a host specified by the `url` variable, and an HTTP response is returned. The method is also able to pass parameters to the specified `url`. Hence, inside the `execute()` method of `SPARQLQuery` object, the required SPARQL parameters explained before are packaged together into one object variable called `parameters` which will be sent within the HTTP request by calling the `send()` method.

![Figure 2.2. SPARQLQuery (the dispatcher) class diagram](image-url)
The abstract HTTP trace example in Listing 2.1 illustrates the invocation of the SPARQL query in http://example.org/sparql/ SPARQL endpoint with a GET sending method. The EncodedGraphURI and EncodedQuery are equivalent representation of the graph URI and the query pattern that have been encoded.

GET /sparql?default-graph-uri=EncodedGraphURI&query=EncodedQuery&output=xml HTTP/1.0
Host: example.org
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.0.8) Gecko/2009032609 Firefox/3.0.8

Listing 2.1. HTTP trace of query dispatching

### 2.2.2 SPARQL Result Parser

W3C recommends a standardized XML document as an optional SPARQL result format to be easily serialized by any applications. We build a parser for this XML document in order to understand the document structure and then to parse the elements into Actionscript native datatype, which are required for further processing.

All variable names found inside `<variable>` will be stored in an array datatype. All the results of a SELECT query will be stored in a multidimensional array, which each row represent one query solution.

One thing to be considered, that each binding variable inside a solution could be a value of a resource/URI, literal, or blank node. This RDF Term is defined in `<uri>`, `<literal>`, or `<bnode>`, and each of it has different behaviors as modeled in Actionscript classes shown in Figure 2.3.

![Figure 2.3. RDFTerm class diagram](image-url)
2. The Architecture

So, the array representation of the variable names and the query solution given by an example XML document in the Listing 1.7 will be represented in Listing 2.2 as follows:

```javascript
arrVariables = ["name", "mbox"];
arrResult = [
    [ "name" => new Literal("Bobby Iceman", null, "http://www.w3.org/2001/XMLSchema#string"),
      "mbox" => new Resource("mailto:bobbyiceman@example.org")
    ],
    [ "name" => new Literal("Tony Ironman", null, "http://www.w3.org/2001/XMLSchema#string"),
      "mbox" => new Resource("mailto:tonyironman@example.org")
    ]
]
```

Listing 2.2. Native Actionscript’s datatypes representing the query results

The first query solution denoted by `arrResult[0]` consists of variable binding `name` and `mbox`. The value of variable binding `name` is a Literal object with a label “Bobby Iceman” as a string datatype. And the value of `mbox` is a Resource object with a URI of `mailto:bobbyiceman@example.org`. And so on with the second query solution.

A `<boolean>` element is a result of `ASK` query. The value of this element could be `TRUE` or `FALSE`. This value will be stored as a string Actionscript datatype.

### 2.2.3 SPARQL Query Builder

We implement the query builder by mainly rebuilding the PHP application logic located in the server into the Client Tier. The query builder is responsible to generate SPARQL queries and send the query patterns to the query dispatcher.

The queries are generated according to the received commands from a caller function. The queries are then sent to a SPARQL endpoint by the query dispatcher. Eventually, after the execution is accomplished; the endpoint will send back the result as an XML document. Then the XML document will be parsed and the query results will be converted into a multidimensional array datatype. This conversion is necessary to be done, in order to make further processing easier. Then the array representation will be manipulated to extract the required data, before it is sent back to the caller function. Finally the presentation handler is responsible to display the data to the user. The dataflow is illustrated in Figure 2.4.
2. The Architecture

Figure 2.4. gFacet Data Flow Diagram
Chapter 3

Browsing DBpedia

The previous chapter explained about the new architecture of gFacet. Now the application logic resides in the Client Tier including the Query Builder class where all the queries according to the user’s commands are built. This chapter will look deeper into the queries which are constructed while user interacts with gFacet. The Query Builder will interpret every user’s action into queries that will be sent to the SPARQL endpoint.

Instead of describing the queries in abstraction, we will explain the query case by case according to user’s action while exploring an RDF dataset with gFacet. In this chapter, we will use dataset released by DBpedia\(^1\) project. DBpedia dataset contains multi domains information which is extracted from structured information from Wikipedia\(^2\).

In this chapter, we will also evaluate the queries execution time and its consistency to give the correct result.

This chapter is structured as follows. Section 3.1 gives short explanation about dispatching a query to DBpedia. Section 3.2 explains query implementation while browsing the DBpedia. And then we run the evaluation in Section 3.3.

3.1. Dispatching a Query to DBpedia

DBpedia released its dataset in order to make it available over the Web. The dataset can be accessed as a Linked Data or via a SPARQL query endpoint. In this thesis, we focus on accessing DBpedia dataset by executing queries to the DBpedia SPARQL endpoint in order to get the result to be viewed using gFacet.

\(^1\) [http://dbpedia.org/](http://dbpedia.org/)
3. Browsing DBpedia

The queries can be sent via HTTP request to the endpoint located at http://dbpedia.org/sparql. The endpoint is provided using OpenLink Virtuoso\(^3\) virtual database engine.

As explained in Section 2.2.1, the endpoint location and the sending method need to be predefined and then the query can be sent along with other parameters required by the endpoint, such as the default graph and the result format. In gFacet, dispatching a query is done as follows:

```
1: var endpoint:String = "http://dbpedia.org/sparql";
2: var aQuery:SPARQLQuery = new SPARQLQuery(endpoint);
3: aQuery.defaultGraphURI = "http://dbpedia.org";
4: aQuery.method = "GET";
5: aQuery.resultFormat = "xml";
6: aQuery.execute();
```

Listing 3.1. Dispatching a SPARQL query to DBpedia

Listing 3.1 describes query dispatching to DBpedia endpoint as a GET method. The query is executed against the DBpedia default graph which located at http://dbpedia.org. We expect the result as an XML document.

The queries that have been built in this thesis are not standard SPARQL queries like the W3C recommendation. There are some cases where the standard SPARQL cannot carry out the tasks. In this thesis we use SPARQL extension such as free-text searching, and aggregating \texttt{COUNT()} function proposed by OpenLink Virtuoso. So, the queries we built may not work in other SPARQL query service except for OpenLink Virtuoso.

3.2. Exploring DBpedia with gFacet

In this section we will describe the detail implementation of the queries generated by gFacet tool. We use DBpedia dataset as the source of our information.

The explanation in this section is given step by step according to the user’s goals. Let’s make an example case and then we will explain how the queries and the gFacet user interface achieve this goal from such a large interlinked dataset like DBpedia. The case is given as follows:

A user is very interested in German Football Clubs. He needs to see all the relevant clubs. He keeps exploring the dataset by looking for information of German Footballers that plays for the German clubs, then English Football Clubs that ever hired the German footballers, and Football Venues where the English clubs reside. And then he needs to find the German football club for which a German Footballer named Thomas Hitzlsperger plays. But after that, he is no longer interested in German Football Clubs.

\(^3\) http://virtuoso.openlinksw.com/
He is changing his point of view. Now he is more interested in English Football Clubs and continues the exploration then.

Let’s formulate the case in simple way

**Goal 1**: User is interested in information about German Football Clubs and he wants to see all the list of the clubs

**Goal 2**: User expands the graph by looking for

(a) All the names of German Footballers that play the for the German Clubs,

(b) All the English Clubs for which the footballers in (a) had ever played, and

(c) The venues where the English Clubs in (b) reside.

**Goal 3**: User wants to see the German Football Club where Thomas Hitzlsperger plays for. He selects the player’s name in order to filter the clubs and receive the information he needed.

**Goal 4**: While exploring the user decides to be more interested in English football clubs than in German ones. Now he changes his perspective on the information.

All the steps to achieve this goal are shown in Figure 3.1 and all the details of every steps will be described in the several next sections.

Figure 3.1. User action’s flow while browsing DBpedia

### 3.2.1. Searching for Concepts

One of the gFacet features is a capability to search for a concept in order to define the initial node to begin with the exploration. User can specify a string of keyword to be matched with the concept name. A list of concepts and the amount of instances, which each concept has, will be displayed as a return (see Figure 3.2).
According to **Goal 1**, the user is interested in information about German football clubs. A screenshot of gFacet interface, Figure 3.2, shows the user specifying his idea by typing the keyword “german football”. Then a list of concepts that contains “german football” text will be displayed also with the number of instances that belong to each concept.

![Search for concepts and add them to the graph.](image)

**Figure 3.2. Searching a concept**

Free-text searching within DBpedia texts can be performed using `bif:contains` predicate which has been proposed for SPARQL extension by Openlink Virtuoso. Since Virtuoso 5.0, it is possible to declare RDF object of triples with a given predicate or object get indexed [25]. Using `bif:contains`, the triples that have been indexed can be found.

Actually, there is a more-generic way of searching texts; by using standardized SPARQL function `regex()`. But we prefer to use `bif:contains` predicate rather than `regex()` function. `bif:contains` looks for the objects from the indexing table, instead of searching in the whole dataset like `regex()` does. This makes the `bif:contains` works faster especially if the query is executed against a large dataset. So, for a large dataset like DBpedia, using `bif:contains` is more reasonable.
The query in Listing 3.2 is the generated query from the example illustrated in Figure 3.2.

```sql
1: SELECT DISTINCT ?concept ?label COUNT(?instance) AS ?numOfInstances
2: WHERE {
6:   FILTER (lang(?label) = "en")
8: }
9: ORDER BY DESC(?numOfInstances) LIMIT 30
```

Listing 3.2. Query for concept searching

In Line 3, we are looking for resources that are identified as a concept. We bind these resources as variable `concept`. Line 4 searches for instances of the concept, we bind to variable `instance`. In Line 5, we use predicate `rdfs:label` to get a human-readable version of a resource’s name. Here we want to get a label of the concept and we need only the categories which has label presented in English as we specified in Line 6. In Line 7, we apply the `bif:contains` into the label of the concept by specifying a string of keyword. Back in Line 1, we define the variable `concept`, `label`, `numOfInstances` to be returned in the solution sequence. We calculate the number of instances using SPARQL extension function, `COUNT()`.

Due to so many possibilities of concepts to be found, we ask the endpoint to order the result in descending (line 9) according to the number of instances each concept has, so the concept at the top of the list might be the most relevant concept for the user.

To make the queries look simpler for explanation, we assume that all the resources’ label will be returned only in English version. So for the next queries, we skip the query patterns like in line 5 and line 6.

### 3.2.2 Selecting the Initial Node

In the list of concepts, the user sees the concept he is looking for, the German football clubs. So he selects the concept and then the initial node of this concept will be opened, as shown in Figure 3.3.
3. Browsing DBpedia

The initial node is set as the result set by default. Result set is the information space that the user interested to. This is the perspective of the user when he looks at the information. All the things that he searches for are displayed in this node. Result set node always appears with a dark grey color.

In gFacet, a node will have a list of the instances of a certain concept that can be paged through, a pull-down menu of relations, and a button to set the node as the result set (see Figure 3.3). The information that is required by the user will be displayed on the instances list. The user can navigate from page to page to explore these instances. By opening the initial node of German Football Clubs, Goal 1 has been achieved.

In the next subsections we will explain the details of the generated queries to retrieve all the instances, the paging mechanism, and to obtain all the relations.

**Retrieving the instances**

The instances will be displayed in a list consists of label and description of the instances. In order to avoid extensive scrolling when there are so many instances displayed at once, gFacet provides a paging mechanism.

By using a paging mechanism, there are at least two tasks need to be done by the application.

1. The application should be able to count the amount of all possible instances. This value is very important to determine how many page buttons should be made.

2. The application should be able to produce just a subset of instances to be displayed according to the page selected by the user.
In this thesis we try two approaches to retrieve the data and apply the paging mechanism; the all-at-once querying and step-by-step querying.

**All-at-once querying:** In this first approach, we solve the tasks by using a standard SPARQL query in order to make the query more generic. In simple words, the query retrieves all the possible instances from the endpoint into the Client Tier and then the client application accomplish the paging mechanism by grouping the instances into several pages and producing a subset of instances to be displayed. The query for this approach is given as follows

```
1: SELECT DISTINCT ?insOfresultSet ?comment_resultSet
2: WHERE {
  3: <URIofConcept> rdf:type skos:Concept .
  5: OPTIONAL {
  6:   ?insOfresultSet rdfs:comment ?comment_resultSet
  7:     FILTER (lang(?comment_insOfresultSet) = "en")
  8:   }
  9: }
```

Listing 3.3. Query to retrieve all instances

The query in Listing 3.3 is just simply asking if the URI of the selected concept is really defined as a concept (Line 3). And if it is really a concept, then Line 4 searches for the instances of it, by applying the `skos:subject` predicate.

In the Line 6 and Line 7, the query requires for an English description of the concept, but we put this requirement into an `OPTIONAL` clause (Line 5), which means that the patterns are not necessarily to have a binding result.

The advantage of this approach is that once all the data has been retrieved, exploring the pages will be comfortably fast, because the application does not have to execute queries anymore. However, the drawback of this approach is that it tends to take a long time for the query engine to query all existing instances of a certain concept especially if there are a lot of instances available.

The second drawback of this approach is caused by the result limitation for every query execution. To protect service from overload, the SPARQL endpoint truncates query results into only 1000 rows every execution [6]. This makes gFacet cannot get the rest of instances if the concept has more than 1000 possible instances.

**Step-by-step querying:** To solve the drawbacks of the first approach, we introduce the Step-by-step querying approach. We try to accomplish the tasks by executing two queries; one query of each task. The first query is to ask the query engine to return only a subset of instances, instead of asking the whole query solutions to be returned. This task is possible to do by using standard SPARQL clauses, OFFSET and LIMIT (see Listing 3.4).
In general the query is the same with the one in Listing 3.3, the difference, which is the important thing of the query, is located in line 10. First, we ask the results to be ordered by the label of the concept in ascending using ORDER BY ASC() clause. And then the main focus of this approach is done by specifying OFFSET index and LIMIT clauses to get the subset of the available query solutions. We predefine a limit of 10 instances to be returned at a time, starting from the defined value in OFFSET clause. These three clauses are needed for paging mechanism; by ordering the solutions before OFFSET-ing we will get a consistent and meaningful order. For example, if OFFSET is set to 0, the query returns instance #1 to instance #10; if the OFFSET is set to 10 we will get instance #11 to #20, and so on.

The second query in this approach is then asking the amount of the possible solutions. But there is no function of standardized SPARQL capable to do this task. That is why we need to use COUNT() function, which is also a SPARQL extension function provided by Openlink Virtuoso, to calculate the amount of instances. We execute the similar query to the first one, but we change slightly what to be returned by the SELECT clause. In line 1, we can see that the query only needs the overall amount of possible instances by specifying COUNT() function inside the SELECT clause (see Listing 3.5).

```sparql
1: SELECT COUNT(DISTINCT(?insOfresultSet)) AS ?totalNumber_concept
2: WHERE {
3:   <URIofConcept> rdf:type skos:Concept .
5:   OPTIONAL {
6:     ?insOfresultSet rdfs:comment ?comment_resultSet
7:     FILTER (lang(?comment_insOfresultSet) = "en")
8:   } 
9: }
10: ORDER BY ASC(?label_insOfresultSet) OFFSET 0 LIMIT 10
```

Listing 3.5. Query for obtaining a total number of possible solutions

Even though we generate two independent queries for this approach, the application works significantly faster by fetching just a subset of the instances, rather than using the
All-at-once querying approach that fetches all instances into the client. And with Step-by-step approach, gFacet does not have a problem with concepts that have more than 1000 instances. gFacet can explore all the instances from the first to the last instance without any limitation.

The drawback of this approach is that it still has to send queries if the user moving from page to page, thus in this case this approach runs less fast than the All-at-once querying approach. Despite of this drawback, the two advantages mentioned before state that the Step-by-step querying approach is more suitable for gFacet.

We will also omit similar patterns to get the description of an instance, like Line 6 and Line 7, in the next sections to make the query looks simpler for explanation.

**Retrieving the Relations**

Each node in graph will have a relations list that gives all the available relations to any nodes related to the current one. These relations are given as a list in a drop-down menu (see Figure 3.4). The relations are presented as pairs of RDF predicate and the related concept name (predicate:conceptName). The amount of the related instances of the new concepts is also displayed in the list of relations. By selecting any of these relations, a new node related to the current one will be opened.

As an example, we take the second row of the relation list. A relation of name:German_footballers means that one or more instances of the node German Football Clubs are related to an arbitrary number of resources by the RDF predicate name. From these resources, there are 302 resources which are instances of
concept German Footballers. The connection between these resources with concept German Footballers are defined by the predicated skos:subject. The RDF graph for this case is illustrated in Figure 3.5.

![RDF graph](image)

**Figure 3.5. Constructing a pair of predicate and related concept**

In Listing 3.6, we provide a query to fill the relations list. The query searches all the predicates that semantically connecting the instances of the current concept with any other resources. Then the query will search all the new concepts to which these resources belong.

The important part of the query is on the Line 6 and Line 7. In Line 6, once the instances of the current concept have been found, the query looks for any related resources of these instances based on certain RDF predicates. And in Line 7, the query searches the new concepts to which these resources belong.

**Line 2 calculates the number of instances for each new concept.**

```sparql
1: SELECT DISTINCT ?prop ?newConcept
2: COUNT(DISTINCT ?instNewCat) AS ?numOfInstances
3: WHERE {
4:   <URIofGermanFootbalClubs> rdf:type skos:Concept .
8:   ?newConcept rdf:type skos:Concept .
9:   ) ORDER BY DESC(?instNewConcept) ?prop ?newConcept LIMIT 40
```

**Listing 3.6. Query for obtaining all the pairs of predicate and concept**

Using combination of RDF predicate and concept name for constructing a facet brings into a relatively time-consuming query execution in the SPARQL engine, especially if the current node has a large amount of possible instances. This is because every instance could have a lot of predicates referring to other resources and then the query should look for the concepts each of these resources belongs to. It is hard to handle a huge
number of combinations of both. This issue makes the execution in SPARQL endpoint takes a long time to complete and in the worst case, execution time limit is exceeded.

To prevent the problem above, we set a limit number of 40 relations should be returned for this query. We order the relation list by the number of related instances in the new concept in descending way (line 9). We realize that displaying only 40 relations is not enough to generalize all the possible combination of relations, however by ordering the number of related instances in descending way, then the most-likely important relations for users will be viewed at the top of the list.

3.2.3. Expanding the Graph

Now we move to the Goal 2, which is to expand the graph by adding more nodes into it.

A new node is opened if user selects a certain relation from the drop-down relation list. An edge will be created and labeled as the predicate selected by the user, as shown in Figure 3.6. This edge will relate the current node and the new node, and indicate the semantic relation between both of them. Gradually expanding the nodes will create a chain of nodes which represents hierarchical facets. This new nodes act as the constraints directly or indirectly for the result set.

In our case, the Goal 2a is to see all the German footballers and our starting node is the initial node German Football Clubs. So from the relation list in the initial node, the user looks for a relation that might be appropriate for his requirement. So he selects the relation name:German_footballers and the new node will be open as shown in Figure 3.6.

Figure 3.6. Opening a new node by selecting a relation
At this point, **Goal 2a** has been done. The **Goal 2b** and **Goal 2c** are done similarly. To get the English football clubs that are related to the German footballers, user selects a relation `clubs:English_football_clubs` from the list in node German Footballers. Then after the node English Football Club is opened, user can select relation `ground:Football_venues_in_England` to see all the stadium where the related English clubs reside. A screenshot of a chain of 4 nodes is presented in Figure 3.7.

Figure 3.7. A chain of 4 nodes is created after user gradually expanding the nodes

In building a query for a chain of nodes, a special characteristic between a child node and its parent has to be considered. An instance of a child node will not be displayed if the instance is not related to any visible instance of its direct parent node. Figure 3.8 will demonstrate how the parent node and child node interact.
We can see in Figure 3.8, that there is no German club that hires a player with a name B4, so that is why player B4 is not visible in the node German Footballers. And so because B4 is not visible, instance C1 in node English Football Clubs will be not displayed also.

This characteristic is intentionally meant so that only relevant instances of a child node can be used to filter instances of its direct parent or to filter indirectly the instances of the result set.

Our approach to express this characteristic is by using nested OPTIONAL clauses. Each child node has to be written inside an OPTIONAL clause. By using this clause, each visible instance of parent node does not necessary to have a related instance in its child node. But, each instance that is visible in the child node must have at least a relation with visible instance in its parent node. The generated query for the chain shown in Figure 3.8 is described in Listing 3.7.

```
1 : SELECT ?instResultSet
2 : WHERE {
3 :   <URIofA> rdf:type skos:Concept .
5 :   OPTIONAL
6 :   {
7 :     <URIofB> rdf:type skos:Concept .
```
3. Browsing DBpedia

Listing 3.7. Query for a chain of 4 nodes; The result set is the initial node

```
10:    OPTIONAL
11:    {
12:        <URIofC> rdf:type skos:Concept .
15:        OPTIONAL
16:        {
14:            <URIofD> rdf:type skos:Concept .
17:        }
18:    }
19: }
20: ORDER BY ASC(?label_instResultSet) OFFSET 0 LIMIT 10
```

We can see that every child is written inside a nested OPTIONAL clause. The indentations in Listing 3.7 show the level of the nodes in the graph. In line 3 – 4, the result set is defined as the node A. Node A has a child which is node B, and the query patterns for B are written in an OPTIONAL clause. In line 7 – 8 the query searches for all resources that are instance of node B. In line 9, here we declare the dependency between node A and node B which is defined by the predicate dbpedia2:name. And so with Line 12 – 14 and Line 14 – 16 for class C and D. Because node C is a child of node B, then all query patterns for C are written inside OPTIONAL clause too.

3.2.4. Filtering

The idea of exploration with gFacet is to restrict the available instances in the result set by selecting arbitrary restriction values so that the user can find the relevant information. Exploring data with gFacet eases the user by constructing the selection queries automatically every time the user adds a constraint. First, user can only select a filter instance at once and then gFacet will display the intermediate results in the result set before user applying more selections.

We now describe closely the filtering operations that can be done with gFacet. There are four filtering operations in gFacet: basic filtering, hierarchical filtering, union filtering, and intersection filtering. gFacet allows a combination of operations as desired by the user.

Basically, the filtering is propagated upward from the selected node until the result set. While propagating upward, there might be intermediate nodes in between that will be filtered before the propagation reaches the result set. After the result set is filtered, every other node has to update its visible instances based on the changes in the result set.
3. Browsing DBpedia

**Basic Filtering** The basic filtering is the simplest operation in gFacet. It restricts the result set by selecting an instance of its direct child node. The selected instance will act as the restriction value for the result set.

Now, as we realize that the **Goal 3** is to filter the German Football Clubs by a German footballer named Thomas Hitzlsperger. This is done by selecting the player in the node German Footballers and then the result will be given in the node German Football Clubs (see Figure 3.9). The result set shows club Vfb Stuttgart as the result.

![Figure 3.9. A constraint selected in direct child of result set triggers a basic filtering](image)

The query to express the task is given as follows

```sql
1: SELECT DISTINCT ?resultSet
2: WHERE {
6:   ?resultSet dbpedia2:name <URIofThomas_Hitzlsperger> .
8:     ?facet1 skos:subject <URIofC> .
9:     OPTIONAL { ?facet1 dbpedia2:ground ?facet0 .
10:    ?facet0 skos:subject <URIofD> .
11:  }
12: }
13: }
14: ORDER BY ASC(?label_resultSet) OFFSET 0 LIMIT 10
```

Listing 3.8. Query for basic filtering. Constraint : footballer Thomas Hitzlsperger
There are two main parts in the query in Listing 3.8, which each part executing different task. The first part is the Filter Patterns (Line 3 – 6). All nodes which are influenced during the filtering are written in these lines. We can see the result set is restricted by the specified constraint <URIofThomas_Hitzlsperger> in Line 6.

Since the filtering part only influence the selected node, the result set, and might be some intermediate nodes, so we write all the query patterns for the uninfluenced nodes below the Filter Patterns inside OPTIONAL clause. This is the second part of the query which is the Update Patterns (line 7 – 13). In this part, all nodes that are not influenced due to filtering will be updated based on the result form the filter part.

Now we will explain the rest of filtering functionalities that gFacet able to do. In order to understand how the functionalities work, we will not use gFacet to illustrate the case; but we will use the relationship model given in Figure 3.8 instead.

Hierarchical Filtering gFacet allows user to add a constraint that not even directly related to the result set. For example user is looking for “all German football clubs that has a player, who had also played for an English club, C4”.

We can see in Figure 3.10 that the node English football clubs is not directly related to the result set German Football Clubs. First the intermediate node German Footballers gets filtered by the specified constraint, in this case only B2 that directly related to C4. And then B2 will act as a constraint for the result set. So the result set will only display instances that have relation to B2. After the filtering, each child node has to update its instances. While B2 is also related to C2, then C2 is now displayed and its related instance, D1, is also displayed.

![Figure 3.10. Hierarchical filtering](image-url)
The Listing 3.9 explains an example query of hierarchical filtering

Listing 3.9. Query for hierarchical filtering. Constraint: English Club C4

```sql
1: SELECT DISTINCT ?resultSet
2: WHERE {
7:     ?facet1 skos:subject <URIofC> .
9:     OPTIONAL { ?facet1 dbpedia2:ground ?facet0 .
10:        ?facet0 skos:subject <URIofD> .
11: } }
12: ORDER BY ASC(?label_resultSet) OFFSET 0 LIMIT 10
```

The selection influences three the nodes which are the selected node C, the intermediate node B, and the result set. All the dependencies between these nodes can be seen in the Filter Patterns (line 3 – 7). And the constraint is defined in line 8. And there is only one uninfluenced node which is node D (line 9 – 11).

**Union Filtering** gFacet allows user to select multiple instances from the same node in order to broaden the result. This functionality will combine the result from each selection.

Figure 3.11 shows the user selects two constraints for the result set. First he selects C4 (indicated with green arrow) and this will return only instance A1. And then he selects another constraint C2, then the result set now is displaying another relevance instance A3. Thus, the result of this functionality is the combination of A1 and A3. And after the filtering, the updating mechanism will take place. While instance D1 is related to C2, then D1 will be displayed.
This operation is expressed in the query using UNION clause between the constraints (see Listing 3.10 line 8 - 9). Using this clause we can combine the result from the first selection with the second one.

```
1: SELECT DISTINCT ?resultSet
2: WHERE {
7:   ?facet1 skos:subject <URIofC> .
8:   {?facet2 dbpedia-owl:clubs <URIofInstanceC4> } UNION
9:   {?facet2 dbpedia-owl:clubs <URIofInstanceC2> } OPTIONAL { ?facet1 dbpedia2:ground ?facet0 .
10:   ?facet0 skos:subject <URIofD> .
11:   }
12: ORDER BY ASC(?label_resultSet) OFFSET 0 LIMIT 10
```

Listing 3.10. Query for union filtering. Constraint : English Club C2 and C4

**Intersection Filtering** Intersection filtering is triggered when the user selects two or more constraints from distinct nodes in order to narrow down the result set. This functionality will intersect between two consecutive results.
In Figure 3.12, user selects D1 as the first constraint. This selection will return A1 and A3 as the intermediate result. And then the user chooses C4 for the second constraint. Because C4 is indirectly related with A1, then this functionality will result only A1.

The generated query for this case can be seen in Listing 3.11. In this case, all the nodes will get influenced by the filtering. All the nodes dependencies are written in the Filter Patterns and also the selections (line 11 – 12)

```
1: SELECT DISTINCT ?resultSet
2: WHERE { 
7:   ?facet1 skos:subject <URIofC> .
8:   ?facet1 dbpedia2:ground ?facet0 .
10:  ?facet1 dbpedia2:ground <URIofInstanceD1> .
12:  }
13: ORDER BY ASC(?label_resultSet) OFFSET 0 LIMIT 10
```

Listing 3.11. Query for intersection filtering. Constraint: English Club D1 and C4
3.2.5. Result Set Pivoting

When we first begin the exploration, the initial node is set up as the result set by default. So the relevant information is displayed in this node. However, gFacet allows the user to change the current result set and thereby to change his/her point of view. By allowing user to change the result set, gFacet gives the freedom to the user to easily change the perspective in which way he/she look at the information. The user is not pushed to have the same perspective from the beginning until the end of exploration.

Now, **Goal 4** is to change the result set to English Football Clubs. Changing result set is done by selecting the result set button in the node English Football Clubs. A screenshot which the English Football clubs as the new result set is given in Figure 3.13.

![Figure 3.13. A chain of 4 nodes after pivoting. The new result set is node C](image)

Changing the result set means rebuilding the graph. The node that has been set up for the new result set is now the root of the graph. The node C as the result set now is displaying all the possible instances, without any restriction from the other nodes. Then the children nodes are updating their visible instances based on the instances in the result set (see Figure 3.14). In the children nodes, the instances that have at least a relation with instance in the result set are displayed, like instance B2, B4, D1, and D2. And instances like B1 and B3 are made invisible due to no relation with the parent node. These changes are propagated too to the next children. Node A has to updates his instances by making instance A1 becomes visible and then instance A2 and A3 become invisible.
Figure 3.14. A model of a chain of 4 nodes after pivoting. The new result set is node C. All the children instances are updated due to this changes.

It is exactly having the same way to build a chain of nodes explained earlier. The important thing in generating query for pivoting is to put the result set query patterns directly inside the WHERE clause and then putting each child nodes inside an OPTIONAL clause based on each level on the graph structure. The query generated for building new graph is given as follows

```sql
1: SELECT ?instResultSet
2: WHERE {
3:    <URIofC> rdf:type rdfs:Class .
5:    OPTIONAL {
6:        <URIofB> rdf:type skos:Concept .
9:    }
10:    OPTIONAL {
11:        <URIofA> rdf:type skos:Concept .
14:    }
15:    OPTIONAL {
16:        <URIofD> rdf:type skos:Concept .
19:    }
20: ORDER BY ASC(?label_instResultSet) OFFSET 0 LIMIT 10
```

Listing 3.12. Query for result set pivoting. Result set : node C

There are slight changes in the query Listing 3.12 compared to query in Listing 3.7 (result set is node A). In Line 3 – 4 we now define that node C is the new result set.
Line 5 – 19 tells that the structure of the graph has changed. In Line 9 – 14, a nested OPTIONAL clause of node A means that A now is a child of node B. And D is a direct child of C (Line 15 – 19).

3.3. Evaluation

In this section we will evaluate the performance of gFacet while accessing DBpedia. The first approach is to measure how fast the generated queries are executed and the second method is to prove the correctness of the queries in returning the result.

3.3.1. Time Measurements

In this evaluation we will measure the execution time of queries using the Step-by-step querying approach of instances retrieval (see section 3.2.2), which are the most suitable approach for gFacet in this thesis. In measuring how fast the queries are executed, we only focus on the elapsed time between HTTP request and the response, instead of measuring the time starting from user’s action until the information is displayed. With this approach we can neglect the user-centric parameters such as the machine performance and the browser used by the user, which is irrelevant for our thesis because the queries are fully executed in the SPARQL endpoint.

Influencing Parameters

The parameters which might influence this measurement are

1. The number of nodes in the chain. As we already explained before, to construct every single node, three queries are sent to the endpoint. The instances retrieval query, the instances calculation query, and query for getting the relations. The more nodes a chain has, the more requests have to be send, and also the more complex the queries will be, due to the dependencies within the chain.

2. The amount of possible instances within each node will influence the execution time in the endpoint. A large amount of instances will require more time to be processed.

3. And the speed internet connection is also need to be considered in this measurement. In this thesis, we try to measure the elapsed time on an internet connection with 12.74 Mbps download speed and 0.89 Mbps upload speed.

Time Measurement A

In this measurement, we calculate the time required to execute queries for a single node. We will test independently nodes with various amounts of possible instances. We repeat
the calculation for 10 times for each node. We analyze on how significant the total of possible instances will affect the query performance. Table 3.1 describes the test nodes and the amount of possible instance of each.

<table>
<thead>
<tr>
<th>Node</th>
<th># possible instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. German Football Clubs</td>
<td>384</td>
</tr>
<tr>
<td>B. German Footballers</td>
<td>1302</td>
</tr>
<tr>
<td>C. English Football Clubs</td>
<td>1127</td>
</tr>
<tr>
<td>D. Football venues in England</td>
<td>224</td>
</tr>
</tbody>
</table>

Table 3.1. The instances of each evaluation node

The result of this measurement can be seen in detail in Appendix 1. Figure 3.15 describes the average time of the execution time after 10 repetitions. As we can wee, that the more possible instances a certain node has, the more time required to execute all the queries for it. Node B, which has 1302 possible instances, requires 8.258 second to retrieve the instances and fill the relation list.

![Figure 3.15. Average elapsed time for nodes with certain amount of instances](image)

Time Measurement B

In this measurement, we calculate the time required to execute all queries to construct a chain of certain length. We investigate how significant the dependencies between the nodes will affect the elapsed time.

A. Testing Environment

In this measurement, we use all the 4 nodes we used in Measurement A. But instead of evaluate the node independently; we construct the nodes into a chain as shown in Figure 3.16
B. Testing Scenario
At first user searches a concept, he continues by choose a concept to be the initial node, and then he expands the graph by opening the second, third, and fourth node. The time calculation is done for every user’s action and it is not accumulative, means that the time is reset before the next action takes place. By doing this, we can evaluate the time needed to search a concept, to open the initial node, to expand the chain with certain length.

We repeat the same procedure for 10 times so we can do the overall analysis from the measurement.

C. Test Result
After we do the repetitions we get all the elapsed time (see Appendix 2) and we calculate the average time for every action as shown in the Table 3.2

<table>
<thead>
<tr>
<th></th>
<th>Average Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search concept</td>
<td>0.4699</td>
</tr>
<tr>
<td>Open the initial node</td>
<td>1.503</td>
</tr>
<tr>
<td>Expand with 2ⁿᵈ node</td>
<td>8.086</td>
</tr>
<tr>
<td>Expand with 3ʳᵈ node</td>
<td>4.222</td>
</tr>
<tr>
<td>Expand with 4ᵗʰ node</td>
<td>2.582</td>
</tr>
</tbody>
</table>

Table 3.2. Average of elapsed time between HTTP request and response for a chain of 4 nodes

When user submits the keyword “german football clubs”, gFacet get the results in 0.4699 seconds. And after he selects the concept and opens the initial node, all the
queries required to open a single node with 384 possible instances (see Table 3.1) are executed in 1.503 seconds.

There is an interesting point if we look at the result carefully. We can see that for a chain of 2 nodes, the elapsed time between the HTTP request and response is significantly increasing. This is obvious, because the node B has more possible instances than any node in the chain.

But, after expanding with the 2\textsuperscript{nd} node; expanding the chain with 3\textsuperscript{rd} and 4\textsuperscript{th} node require less execution time. The reason behind this occurrence is because the effect of a cache. OpenLink Virtuoso provides a cache to store the most recent query execution plans. Because query execution plans for 1\textsuperscript{st}, 2\textsuperscript{nd}, and 3\textsuperscript{rd} node already exist in the cache, expanding the chain with the 4\textsuperscript{th} node only requires 2.582 second.

3.3.2. Correctness Measurement

The correctness measures are meant to prove whether the information returned to the user is consistent with the actual relationship within the instances of the dataset. We will test the queries with simple test cases. For these test cases we need a simple and controlled dataset, so we can priory estimate the result already. After we apply the test cases to gFacet, then we can check whether gFacet returns the result as expected or not.

We take a sample from DBpedia consist of 4 concepts; British Musical Groups as the result set, Britpop Songs, British Record Labels, and Living People. We construct the chain as shown in Figure 3.17

![Sample dataset: A chain of 4 nodes with Britpop Musical Group as the result set](image-url)
For this evaluation, we are only interested to small piece of data from the dataset above. After we examine the data, we can extract the relationship between instances as shown in Figure 3.18

![Figure 3.18. Relationships diagram for the sample dataset](image)

**Test Case 1: Basic operation and Union operation**

Union operation is triggered when there is more than one selection in a node. In this test case we iteratively filter the result set by selecting the instances in British Record Labels; first select Food Record (Test case 1.1), then select Parlophone (Test case 1.2), and then Virgin Record (Test case 1.3). We will compare the result of gFacet with the result we have estimated by examining the relationship diagram in Figure 3.18.

We can see in Table 3.3, when the user selects instance Food Records in concept British Record Labels, the only instance get filtered in the result set is Blur. Then the user broadens his search by also selecting instance Parlophone, here the UNION operation is applied, thus instance Supergrass then added in the result set. This works also when the user select the third instance, gFacet now returns instance Blur, Supergrass, The Verve, and These Animal Men. So for all the OR-operation test cases, gFacet has return correctly as expected.
3. Browsing DBpedia

Table 3.3. Result of OR-operation test cases. All objects influenced by the selections are indicated by Magenta outlines.

**Test Case 2: Intersection and Hierarchical operation**

For the test case 2 we will check how well gFacet solves intersection and hierarchical operations. Intersection operation is triggered when a user select more than one instances in multiple nodes. We will test gFacet with two cases as follows

1. Multiple selections on sibling nodes (Test Case 2.1): Select instance Virgin Records in concept British Record Labels and select also a song with a title of Bitter Sweet Symphony in concept Britpop Songs.

2. Multiple selections on hierarchical nodes (Test Case 2.2): Select instance Virgin Records in concept British Record Labels and select also a person with a name of David Balfe in concept Living People.
3. Browsing DBpedia

Table 3.4. Result of AND-operation test cases. The end result is the object that has outline with two colors

In the Table 3.4, for intersection operation test cases gFacet acts like the way we expect it to be. In test case 2.1, when user select Virgin Record first, gFacet returns Blur, The Verve, and These Animal Men. But after user specifies his second selection which is a song titled Bitter Sweet Symphonie, we filter the previous result of gFacet with this song. So after the second selection gFacet will return only musical group The Verve.

In the test case 2.2, user narrows his search by select David Balfe in concept Living People, which is not directly related to result set. If we trace the relation arrow, the arrow for this instance ends at instance Blur in the result set. So by applying intersection operation from the result of first selection and the second selection, we will have Blur as the result. And gFacet gives the same result as expected.

So intersection and hierarchical operation test cases have been accomplished by gFacet also with correct results.
Chapter 4

Multiple Sources

We have seen the capabilities of gFacet in the previous chapter. However, the demonstration in the previous chapter is limited to the exploration of a single RDF data source. Since the DBpedia\(^1\) dataset is interlinked with various other data sources on the Web using RDF Links [6] (see Figure 4.1), we try to make this advantage applicable to gFacet in order to make gFacet allows multi sources exploration.

What we mean about the multi sources exploration is that the user can browse a thing from a starting source, and then he can continue browsing related resources which might be located in another data source available on the Web. This multi sources exploration could help user to gain more relevant data as he desires.

In this thesis we implemented a way to achieve this goal. Even though this feature has not fully implemented yet into gFacet, but we will explain some ideas so that gFacet will be able to gather resources from more than one source.

This chapter is structured as follows: 4.1 explains the strategy in general, 4.2 explains some issues that can be the obstacles to the goal, 4.3 explains how to get equivalence resources, 4.4 describes mapping mechanism of the obtained RDF links, 4.5 explains the current implementation of the goal.

\(^1\) http://dbpedia.org/
4. Multiple Sources

Figure 4.1. Data sets that have been published and interlinked by Linking Open Data\(^2\) project (March 2009)

### 4.1. The Strategy

In our implementation, the starting data source is the DBpedia data sets. User browse from these data sets at first, and then he can open a new node that fetches information from an additional data source.

In this thesis, we are interested to explore more musical data gained from other source around the Web. One project that released a large amount of musical data is MusicBrainz\(^3\). However, MusicBrainz data is built on a relational database. In order to make the data available as a part of the Semantic Web, Zitgist\(^4\) and DBtune\(^5\) have successfully mapped the MusicBrainz relational database into RDF dataset using Music Ontology.

\(^2\) http://esw.w3.org/topic/SweolG/TaskForces/CommunityProjects/LinkingOpenData  
\(^3\) http://musicbrainz.org/  
\(^4\) http://zitgist.com/  
\(^5\) http://dbtune.org/
Since MusicBrainz RDF dataset released by Zitgist has been interlinked directly with DBpedia dataset, we will use the interlinked resources as our anchor to connect between the two datasets. According to DBpedia webpage\(^6\), there are about 23,000 links of resources in MusicBrainz RDF dataset hosted by Zitgist that have been interlinked from DBpedia dataset. It can be used to gain more information about something related to music.

Every data source might have a different URI scheme to identify each entity within the dataset. So, in order to be able to browse from DBpedia and Zitgist’s MusicBrainz dataset, we have to identify in which way two URIs from both sources are explaining the same entity.

However, Zitgist does not provide any SPARQL endpoint that allows querying to the MusicBrainz RDF dataset. But DBTune provides this service by making it available via a SPARQL query service. So we will access the MusicBrainz dataset by executing the query against DBtune endpoint. From this endpoint we will gain more information about the requested resources. Once we able to do this, we can obtain a large collection of information, by looking for information in DBpedia and DBtune’s Musicbrainz dataset separately and then gather it together.

The queries are sent to both DBpedia and DBtune. However, since DBtune maps the MusicBrainz scheme into Music Ontology, the definition of each class might be different between the two schemes. So, we have to find out first in which class an entity is mapped to Music Ontology from the original class in MusicBrainz scheme. This task can be done by comparing the terminology of the class in MusicBrainz scheme with the class in the Music Ontology.

An illustration of multi sources exploration is given in the following Figure 4.2

![gFacet model of multi sources exploration](http://wiki.dbpedia.org/Interlinking)

\(^6\) [http://wiki.dbpedia.org/Interlinking](http://wiki.dbpedia.org/Interlinking)
Figure 4.2 describes a model of gFacet that can access two datasets, DBpedia and DBtune’s MusicBrainz dataset. In the model, only 3 of the instances of node “Music Band” are interlinked with MusicBrainz dataset. By comparing the terminology of the instances’ class in MusicBrainz scheme and Music Ontology, we can find out that the instances belong to class “musicArtist” in the Music Ontology. And then user can explore both from DBpedia or MusicBrainz dataset separately.

4.2. The Obstacles

There are three obstacles that might complicate multi sources exploration to be applied in gFacet for a moment:

1. No current technology that enables invoking a SPARQL query to multiple sources at once. Since current SPARQL version does not support for query federation to execute queries to multiple sources at once, we will have to separately execute queries to DBpedia endpoint and to DBTune which provides MusicBrainz data accessible via a SPARQL endpoint.

2. The difference scheme between DBpedia SKOS categorization and Music Ontology makes the queries explained in Chapter 3 have to be adjusted in order to applicable for MusicBrainz RDF dataset.

3. DBTune SPARQL endpoint is using D2R server to map the MusicBrainz relational database into RDF dataset. While mostly our queries are tailored in order to browse DBpedia efficiently by using Openlink Virtuoso’s SPARQL extensions, some of these features are not applicable to DBtune endpoint.

Despite of these obstacles, we try to achieve the goal, which we will explain in the next several sections.

4.3. Finding the Equivalent Data

The very first thing to do to allow exploration between distinct data sources is to find the equivalent resources of the resource requested by the user. Two resources that explaining the same thing are indicated to be equivalent by using owl:sameAs predicate. This predicate allows us to find all the equivalent resources so that gFacet can collect information even more and not restricted within one source.
The RDF graph in Figure 4.3 describes that a resource in DBpedia is equivalent with three resources in the other sources; Zitgist, Freebase\(^7\), and OpenCyc\(^8\). All the resources are explaining about musical band Portishead.

![Figure 4.3. RDF graph of a subject with three equivalent resources](image)

Due to so many possibilities of equivalent data from other sources, we are only interested to the resource from MusicBrainz dataset. The equivalent MusicBrainz entity, which is hosted by Zitgist, is identified by the text “zitgist.com/music/” contained in the URIs.

The URIs from other data sources may be not relevant for our approach, because they might not provide SPARQL endpoint which is essential for our approach to be able to control the information retrieval.

If we open the left-side URI in the browser we can see all information about Portishead collected from DBpedia dataset (see Figure 4.4 (a)). The information is displayed using DBpedia resource viewer. The Zitgist URI will give us the information from MusicBrainz dataset, which will be rendered automatically using Zitgist data viewer (see Figure 4.4 (b)).

We can see that both URI shows us the same music band.

\(^7\) [http://www.freebase.com/](http://www.freebase.com/)

\(^8\) [http://www.opencyc.org/](http://www.opencyc.org/)
4.4. Transforming the URIs

As we can see, a resource in DBpedia is interlinked with MusicBrainz dataset which hosted by Zitgist. Since the SPARQL query service for MusicBrainz dataset is hosted by DBtune, we need to know the location of every entity using DBtune URI scheme to be able to browse the dataset. So, that is why the given Zitgist URI above is no use for us without any manipulation done first. We have to transform the Zitgist URI into DBtune URI first, in order to know the location of the entity in DBtune.

Since gFacet is a faceted data viewer, all information about resource id or the concept in which the entities belong to is really necessary. Obtaining this resource id and its concept from a given Zitgist URIs requires a mapping mechanism.

4.4.1. Zitgist URI to DBTune URI Conversion

Every entity in MusicBrainz dataset has a unique resource identifier, which is an absolute URI. In general, it has the form

http://musicbrainz.org/class/UUID

where class can be of Artist, Release, Track, or Label, and a Universally Unique Identifier\(^9\) in its 36 character ASCII representation.

Since, Zitgist and DBTune is originally mapped from MusicBrainz metadatabase, they have the similar URI scheme as MusicBrainz. For Zitgist, the scheme looks like

http://zitgist.com/music/class/UUID

---

and for DBTune

http://dbtune.org/musicbrainz/resource/class/UUID.

Because we have the Zitgist URI already, given by DBpedia, mapping this into DBTune resource identifier for MusicBrainz data can be done by extracting the class and UUID and then append it with text “http://dbtune.org/musicbrainz/resource/” (see Figure 4.5). The DBTune identifier is very important to us, since we want to gain more data from MusicBrainz dataset hosted by DBTune.

Figure 4.5. Extracting the class and UUID of MusicBrainz to be mapped to DBTune URI scheme

### 4.4.2. MusicBrainz Scheme to Music Ontology Mapping

The URI schemes provided by MusicBrainz ease us to extract the class of each entity. However, since DBTune maps the MusicBrain metadatabase into Music Ontology, we need to map this class released by MusicBrainz into the class of Music Ontology.

In this thesis we manually compare the terminology of each entity class in MusicBrainz and a class in Music Ontology, and then decide whether both have similar terminology or not. We will map the type in MusicBrainz with a class of Music Ontology if they both share a similar terminology. Table 4.1 shows the mapping of class in MusicBrainz and a class in Music Ontology along with the terminology of each.

<table>
<thead>
<tr>
<th>MusicBrainz Class</th>
<th>Music Ontology Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artist</td>
<td>mo:MusicArtist</td>
</tr>
<tr>
<td>“A musician, group of musicians, a collaboration of multiple musicians or other music professional”</td>
<td>“A person or a group of people (or a computer ), whose musical creative work shows sensitivity and imagination”</td>
</tr>
<tr>
<td>Release</td>
<td>mo:Record</td>
</tr>
<tr>
<td>“An entry in the album table of the database”</td>
<td>“A published record (manifestation which first aim is to render the product of a recording)”</td>
</tr>
</tbody>
</table>
4. Multiple Sources

<table>
<thead>
<tr>
<th>Track</th>
<th>mo:Track</th>
</tr>
</thead>
<tbody>
<tr>
<td>“A track is an entry in the track table of the database”</td>
<td>“A track on a particular record”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Label</th>
<th>mo:Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>- “An imprint is (strictly, and nothing more than) a brand (and trademark) associated with the marketing of sound recordings”</td>
<td>“Trade name of a company that produces musical works or expression of musical works”</td>
</tr>
<tr>
<td>- “A record company typically manages imprints, and coordinates the production / manufacturing / promotion / relations with artist / PR / distribution of sound recordings”</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1. Comparison MusicBrainz entity’s class and Music Ontology class

4.5. Implementation

Now we will explain the first implementation of the idea above in gFacet. In general the data flow is illustrated in Figure 4.6

Since MusicBrainz contains data related to music, so we are only interested to such of data released by DBpedia as the starting dataset. In this case we set concept “English Electronic Music Groups” as the result set.

The main strategy is to collects all equivalent data by searching binding results of owl:sameAs predicate for every DBpedia resource, as explained in section 4.3. This task is done inside the query for filling the relation list of a node (see Listing 4.1). The query asks for all the available relations and also to return all the Zitgist links found and calculate how many of them available.
4. Multiple Sources

Listing 4.1. Query to get all relations and count the interlinked data of a concept

The important things in the query located in Line 8 – 22. We ask optionally for all instances that have equivalent resources in the other nodes and filter these resources so that only resources containing text “http://zitgist.com/music/” are accepted (see Line 10 – 11). While these two lines ask for the explicit Zitgist URIs, Lines 13 – 22 only calculates how many instances of concept “English Electronic Music Groups” are interlinked with Zitgist’s MusicBrainz RDF dataset. The calculation is applied by using a subquery inside the main query.

If there is at least one Zitgist URI found, a relation sameAs:MusicBrainz will be added in the relation list, as shown in Figure 4.7. This relation indicates that there is available equivalent data in MusicBrainz dataset and the number on the left-side shows how many instances of the concept “English electronic Music Groups” are interlinked with MusicBrainz, which are 6 intances in our case.
4. Multiple Sources

Figure 4.7. Relation list with sameAs:MusicBrainz element to open node from MusicBrainz data

Then user can select the sameAs:MusicBrainz relation if he want to gain more information released by MusicBrainz. A new node of MusicBrainz data will be opened (see Figure 4.8).

Figure 4.8. A chain of two nodes from distinct sources. The left node retrieves data from DBpedia and the right node retrieves data from DBtune’s MusicBrainz dataset

Then the equivalent Zitgist URIs are mapped into DBTune URI class (see Section 4.4.1). By doing this we will obtain the actual DBTune identifier of each instance and the class of it. This is necessary to be done so we know in which class we have to search the instances in DBTune dataset.
4. Multiple Sources

After we extract the class and UUID of every Zitgist URI, we find that all the instances in our example belong to class Artist, and then all these instances will be mapped into mo:MusicArtist class of Music Ontology by comparing the terminology of both class (see Section 4.4.2).

Since we have the actual DBTune URI for each instance and their class in Music Ontology, the instances of the new node are searched using this information. The query in Listing 4.3 will be dispatched to the DBTune SPARQL endpoint (see Listing 4.2).

```
1: var aQuery:SPARQLQuery = new SPARQLQuery
   ("http://dbtune.org/musicbrainz/sparql");
2: aQuery.defaultGraphURI = "http://musicbrainz.org";
3: aQuery.method = "GET";
4: aQuery.resultFormat = "xml";
5: aQuery.execute();
```

Listing 4.2: Dispatching a SPARQL query to DBTune for MusicBrainz dataset

The query ask for all the instances of class MusicArtist and then gives constraints in order to get only the instances that are interlinked with DBpedia.

```
1 : SELECT ?facet0 ?label_facet0
2 : WHERE {
3 :   ?facet0 rdf:type mo:MusicArtist .
4 :   ?facet0 rdfs:label ?label_facet0 .
5 :   FILTER (((?facet0 = <DBTuneURIforInstance1>) ||
6 :     (?facet0 = <DBTuneURIforInstance2>) ||
7 :     (?facet0 = <DBTuneURIforInstance3>) ||
8 :     (?facet0 = <DBTuneURIforInstance4>) ||
9 :     (?facet0 = <DBTuneURIforInstance5>) ||
10 :    (?facet0 = <DBTuneURIforInstance6>) ))
11 : }
```

Listing 4.3. Query to get the interlinked data from DBtune for MusicBrainz dataset

At this point we have successfully implemented two data sources for gFacet. Yet the Figure 4.5 looks simple, we have made these two nodes having different data sources.

Further implementation such as filtering from both source and result set pivoting will require more deep research. As stated before, that DBtune uses D2R server for the SPARQL query service. Since all the queries we have built for efficiently browsing DBpedia is using SPARQL extension from Openlink Virtuoso, these queries might not applicable in D2R server used by DBtune. We have to rebuild new queries that can be executed in D2R server.
And the other issue is because of the difference categorization scheme between DBpedia and Music Ontology; it will need more research to be able to gain more musical data from Musicbrainz dataset.
Chapter 5

Conclusions & Future Works

5.1. Conclusions

This thesis mainly focuses on accessing and manipulating RDF dataset using SPARQL. We also use gFacet faceted RDF browser as a platform to view the data. Since the latest gFacet version worked on a Three-Tier Architecture, in this thesis we have successfully reorganized and simplified the architecture of gFacet. All the intermediated processing inside the Server Tier is now moved into the client. So now gFacet can run completely in the Client Tier.

The new architecture of gFacet allows direct communication with a SPARQL query service. This makes gFacet can easily access any endpoint around the Web and to browse the available dataset. One dataset that in this thesis used is the one that been released by DBpedia. Via the provided SPARQL endpoint, we browse DBpedia using gFacet and investigate how the queries we have built work on such a large dataset like DBpedia. We have also tried to build queries with some standard and extended SPARQL functions.

One new feature of gFacet is to allow user to search any concept he needed. And this feature is supported by a query using bif:contains predicate. This predicate can search from an index table of triple patterns, so text searching can run efficiently rather than using a standard regex() SPARQL function.

To retrieve the data from DBpedia we have implemented two approaches. The first approach is using a standard query, we cakk it All-at-once querying approach. It asks for all instances of a concept at once. The extensive execution time and the limitation of 1000 rows per execution pushed us to build another approach. And the second approach, Step-by-step querying approach, uses the OFFSET and LIMIT clauses to get only a subset of all possible instances, and this brings into a significantly faster
5. Conclusions

execution. And using the OFFSET and LIMIT allows user to browse unlimited amount of instances.

In this thesis, we also have implemented some filtering operations like basic, hierarchical, union, and intersection filtering to allow user to broaden or narrow down the result while exploring. The result set pivoting mechanism is also well implemented. Now user can actively change the perspective of the information he is looking for.

From the evaluation of time parameter, we can see how a large concept can significantly influence the execution time. A concept with the most amounts of instances will take more time than the other concepts. However, the capability of Virtuoso to store the recent query plans in the cache, makes execution of the same query in some time after become faster.

For the evaluation of correctness measure, we have tried to evaluate all the filtering operations with some test case. And gFacet can give results as expected.

In the chapter 4, we have proved that it is possible to make gFacet able to browse more than one RDF source. We make gFacet able to send query to both DBpedia and DBTune SPARQL endpoint. We have already implemented this feature for two simple nodes currently. More functionality to be applied will need more complex queries to be built. Since we tailored the queries for efficiently browsing DBpedia which is supported by Virtuoso at the back-end, these queries might not applicable to DBTune SPARQL endpoint since DBTune uses D2R server as the database engine. This will need much adjustment to the queries we have built. And the other issue is because of the difference categorization scheme between DBpedia and Music Ontology; it will need more research to be able to gain more musical data from Musicbrainz.

5.2. Future Works

The topics that are covered in this thesis, such as Semantic Web, RDF, SPARQL, faceted exploration, DBpedia, MusicBrainz, are still relatively fresh topics. There are still many researches and innovations in these topics that might be applicable to gFacet for improvement in the future.

5.2.1. Autocompletion Text Search

When user is searching for concept, he has to specify at least one complete keyword. Assuming that the user mistakenly type a false letter, the submitted text can result to no concept found. So for this reason, it is nice to have an autocompletion text searching.
While user typing the keyword, after every keystroke the application dynamically suggests a list of concepts that matches the prefix currently typed so far. By providing this feature, it will help the user to access the data without necessary knowing the structure and concepts in the dataset.

[18] has implemented such kind of feature. Their implementation is capable of faceted searching with text autocompletion.

5.2.2. Searching for Instances

While in our current gFacet allows free text searching, however the text submitted by the user will be matched only with the concept’s name but not directly to the instance’s label. In our opinion, instances searching will give more benefits to the user, since users tend to start from something in his mind and usually it is not a concept, but the instance itself. For example, if a user wants to know about “Michael Schumacher”, he could search directly by that keyword, rather than guessing a concept “List of Formula 1 Drivers”.

The query below might be useful for further development of instance searching

1: SELECT DISTINCT ?concept ?instance
2: WHERE {
7:   FILTER (lang(?label) = "en")
8:   ?labelOfInstance bif:contains “Schumacher” .
9: }

Listing 5.1. Query to search for instances

The query will search keyword “Schumacher” directly to the instance’s label. And the SELECT query will return the instance itself and the concept it belongs to.

5.2.3. Automatic Data Interlinking

In Chapter 4 we have try a naive approach to find the interlinked data from DBpedia and MusicBrainz. Unfortunately, due to enormous resources available in DBpedia, not all instances are interlinked. In our implementation previously for example, from 13 instances of English Electronic Music Groups, there are only 6 of them that interlinked to MusicBrainz dataset. Thus, it would be very useful if the not-interlinked instances
can be automatically interlinked using some specific algorithm like in [19]. They provide three approaches for automatically interlinking of music datasets on the Semantic Web.
Bibliography


[22] Quilitz, B., and Leser, U. Querying Distributed Data Sources with SPARQL.


Appendix 1: Results of Time Measurement A (Section 3.3.1)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>node A</th>
<th>node B</th>
<th>node C</th>
<th>node D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.61</td>
<td>8.26</td>
<td>3.81</td>
<td>1.05</td>
</tr>
<tr>
<td>2</td>
<td>1.5</td>
<td>8.14</td>
<td>3.35</td>
<td>1.05</td>
</tr>
<tr>
<td>3</td>
<td>1.48</td>
<td>8.36</td>
<td>3.44</td>
<td>0.968</td>
</tr>
<tr>
<td>4</td>
<td>1.54</td>
<td>8.14</td>
<td>3.3</td>
<td>1.19</td>
</tr>
<tr>
<td>5</td>
<td>1.48</td>
<td>8.33</td>
<td>3.26</td>
<td>1.02</td>
</tr>
<tr>
<td>6</td>
<td>1.52</td>
<td>8.31</td>
<td>3.34</td>
<td>1.03</td>
</tr>
<tr>
<td>7</td>
<td>1.5</td>
<td>8.14</td>
<td>3.28</td>
<td>1.02</td>
</tr>
<tr>
<td>8</td>
<td>1.55</td>
<td>8.13</td>
<td>3.36</td>
<td>0.975</td>
</tr>
<tr>
<td>9</td>
<td>1.57</td>
<td>8.46</td>
<td>3.31</td>
<td>0.987</td>
</tr>
<tr>
<td>10</td>
<td>1.44</td>
<td>8.31</td>
<td>3.33</td>
<td>1.1</td>
</tr>
<tr>
<td>Average</td>
<td>1.519</td>
<td>8.258</td>
<td>3.378</td>
<td>1.039</td>
</tr>
</tbody>
</table>
### Appendix 2 : Results of Time Measurement B (Section 3.3.1)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Elapsed Time (s) for Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Search concept</td>
</tr>
<tr>
<td>1</td>
<td>0.441</td>
</tr>
<tr>
<td>2</td>
<td>0.464</td>
</tr>
<tr>
<td>3</td>
<td>0.478</td>
</tr>
<tr>
<td>4</td>
<td>0.495</td>
</tr>
<tr>
<td>5</td>
<td>0.471</td>
</tr>
<tr>
<td>6</td>
<td>0.453</td>
</tr>
<tr>
<td>7</td>
<td>0.479</td>
</tr>
<tr>
<td>8</td>
<td>0.478</td>
</tr>
<tr>
<td>9</td>
<td>0.476</td>
</tr>
<tr>
<td>10</td>
<td>0.464</td>
</tr>
<tr>
<td>Average</td>
<td>0.4699</td>
</tr>
</tbody>
</table>