Collaborative Project

GeoKnow - Making the Web an Exploratory Place for Geospatial Knowledge

Project Number: 318159 **Start Date of Project:** 2012/12/01 **Duration:** 36 months

Deliverable 6.3.1

Design and Specification of the Motive-Based Search Architecture

Dissemination Level	Public
Due Date of Deliverable	M18, 31/05/2014
Actual Submission Date	M19, 30/06/2014
Work Package	WP 6, GeoKnow for E-Commerce
Task	T6.3
Type	Report
Approval Status	Approved
Version	1.0
Number of Pages	19
Filename	D6.3.1_Design_Motive-Based_Search_Infrastructure.pdf

Abstract: This deliverable presents the design and specification of a search infrastructure specifically designed to support motive-based search. It focuses on data transformation into specialised data structures, query interpretation, and a federated search architecture.

Project funded by the European Commission within the Seventh Framework Programme (2007 - 2013)

History

Version	Date	Reason	Revised by
0.1	10/06/2014	Initial version, outline	Matthias Wauer
0.2	12/06/2014	Initial content	Didier Cherix
0.3	13/06/2014	Contributions to sections 3 and 4	Christiane Lemke
0.4	19/06/2014	Contributions to data aggregation and transformation	Matthias Wauer
0.5	24/06/2014	Consolidation of deliverable	Matthias Wauer
0.6	25/06/2014	Peer review	Daniel Hladky
0.7	25/06/2014	Internal review	Andreas Both
0.8	26/06/2014	Final changes w.r.t. review comments	Matthias Wauer
1.0	27/06/2014	Final version	Andreas Both

Author List

Organization	Name	Contact Information
Unister	Matthias Wauer	matthias.wauer@unister.de
Unister	Christiane Lemke	christiane.lemke@unister.de
Unister	Didier Cherix	didier.cherix@unister.de
Unister	Andreas Both	andreas.both@unister.de

Executive Summary

Providing linked geospatial data for the E-Commerce use case is only one aspect of work package 6, making best use of it is the second important part. For the E-Commerce application, we have so far described components and processes for generating an interlinked dataset in deliverables 6.1.1, 6.1.2 and 6.2.1. In this deliverable, we will focus on how to performantly search this data and how to best exploit all inherent information in order to provide a better search experience to our customers. We will show how to answer motive-based queries like "winter holiday with culture and mountains" with the technology currently available and describe future work in the scope of GeoKnow.

Table of Contents

1 Introduction

This deliverable describes the infrastructure of a motive-based search engine, allowing customers in the e-commerce domain to express their information need as freely as possible. It will describe three aspects of the system:

- The aggregation and transformation of a dataset, consisting of linked geospatial data and CRM data in the tourism domain), into search-optimized data structures
- The interpretation of the user query
- Architecture of the retrieval process

1.1 Motivation

In the e-commerce domain, simple text retrieval mechanisms are often not sufficient to provide the customer with the information he or she is looking for. Search scenarios like finding a suitable hotel for a vacation are often complex, with users often expressing their search queries based on vague feelings and ideas. Satisfactory results to queries such as "winter holiday with culture and mountains" could include hiking holidays as well as a tour of cultural highlights during the winter school holidays in the region of the customer. Answering these queries requires substantial spatial background knowledge as it can be provided by linked geospatial data generated in the scope of GeoKnow. For example, the internal datasets in the E-Commerce use case include general hotel attributes (including geospatial coordinates) and CRM related information, such as flight bookings. In order to answer the above query, we have to integrate linked open data for cultural activities and mountains, and interlink these individual datasets according to a previously defined ontology (discussed in deliverables D6.1.2 and D6.2.1).

An important challenge in retrieval systems based on big and complex data is response time. For a search engine in the travel domain, response times over 5 seconds are hardly acceptable, and the majority of the queries should be answered in under a second. A common and necessary step is to aggregate and transform the data set into search-optimized data structures. While the current state-of-the-art is to query such data statically, a lot of flexibility can be gained by allowing for dynamic information derivation from the data graph.

However, this adds another performance challenge. Take, for example, the static query "Hotel in Leipzig". Most likely this information is already stored in a structured way, but even if it is not and the search engine has to find matching geocoordinates, geospatial databases are capable of answering such queries quickly. In contrast, a more dynamic query like "Hotel with wireless LAN in a city near mountains with excursion destination Neuschwanstein Castle" is much more difficult to answer promptly. Each aspect of the query, if interpreted correctly, extends the search graph considerably and makes predictive query optimization very difficult. This can be tackled by combining three different research areas: ontology-based question answering, heuristic-based derivation and statistics-based Information Retrieval.

1.2 High-Level Architectural Overview

In order to be able to solve such complex queries, a search infrastructure has to support the various required search functionalities, which can be formulated as requirements towards a service-oriented architecture. In order to achieve the required scalability, we define the following architectural properties:

Requirement 1. The search architecture defines an interface for search services. The definition is comprised of a summarization of all needed attributes for controlling the execution of (sub-)queries.

Requirement 2. The communication with the search services has to be stateless and transparent.

Requirement 3. The search query has to be modeled as a structured representation which encapsulates information about all relevant aspects of the search query.

Requirement 4. The structured search query representation is independent from any particular search service.

Different data types are present in the data set in the e-commerce use case:

- logical properties, for example "has shuttle transfer to ski lift" or "suitable for vegetarians"
- geospatial properties, for example "north of London", "close to skiing resort"
- text information that are not semantically captured, such as hotel descriptions and reviews

No single data management system does, at the time of writing, offer functionality and performance to search all three types of data at the level required by the application (see deliverable D1.1.1). There are, however, stores that specialize in one of the data types:

- searches for logical properties are implemented by triple stores, for example the Openlink Virtuoso Server [**?**];
- searches for geospatial properties are supported by database management systems with extensions for geographic information system (GIS), e.g. PostGIS [**?**], an open source software program that adds support for geographic objects to the PostgreSQL object-relational database [**?**];
- searches within text documents are supported by index-based data stores, for example the scalable search solution Elasticsearch [**?**] based on Apache Lucene [**?**].

The initial motive-based search architecture introduced in this deliverable consists of loosely coupled search services. Each of them provides search functionality tailored to the capabilities of a powerful backend: currently, the stores OpenLink Virtuoso, Elasticsearch and PostGIS are used, they can however be exchanged to alternative implementations. The services can be queried using a generalized interface inspired by RDF. An application layer on top of these services allows the implementation of a distributed search engine identifying the optimal search service for each query and subquery. In this way, we are able to implement a scalable search solution capable of conducting semantic search with geospatial aspects as well as information retrieval from text documents.

Hence, the components of a high-level architecture can be distinguished between two stages. A *search query analyzer* takes an unstructured search query and returns the most likely semantic interpretation in a structured form. Using this structured search query, *search query interpreters* handle the query aspects which can be answered by their underlying search engine using a federated search approach. The search process is visualized in Figure 1.

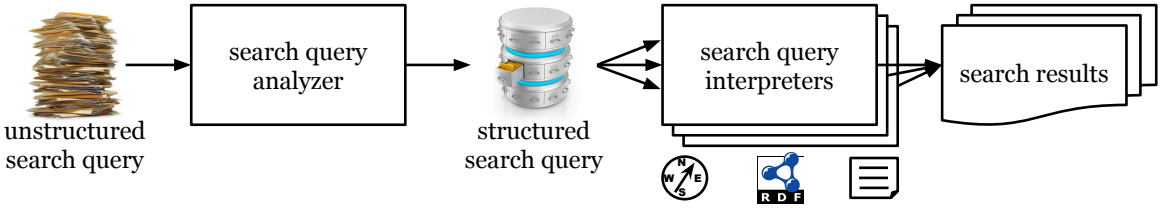

Figure 1: High-level Search Infrastructure Architecture

The initial federated search process is handled by a component not built into any of the stores, but as the semantic geospatial data management is improving (see GeoKnow work package 2) we plan to integrate the federated search management more closely with the triple store[1].

Still, this process can benefit massively from preprocessed information. For example, precomputing excursion destinations based on certain parameters such as minimum and maximum distance can reduce the number of required search query interpreters, which in turn accelerates the search result merging process and also reduces the risk of delaying search query interpreters. This preliminary step of preprocessing data is discussed in the following section 2, while the components for interpreting and processing the search query are described in sections 3 and 4.

[1]By this means, we might also be able to reduce the number of required stores as more functionality of the specialized stores is integrated with the triple store and meets the performance requirements of our use case.

2 Data Aggregation and Transformation

While many queries can be processed using the originally available information, such a search would be highly inefficient in most cases. For example, the query "Hotel in Europe named like America" contains two search aspects:

1. a geospatial search, returning the subset of hotel coordinates intersecting the Europe polygon.
2. a index-based search, selecting the entities whose label contains "America".

Since certain geospatial properties occur frequently in a typical tourism-oriented search infrastructure, it makes sense to transform this implicit knowledge into explicit statements, which can typically be evaluated much faster. In the case of a federated search infrastructure as discussed here, it can further accelerate the search process by skipping the potentially costly result merging process, along with providing a much more predictable query plan. The downside of this is the time-consuming preprocessing task and increased memory requirements for storing the explicit statements.

Consequentially, we focus on five general means of data aggregation and transformation in order to speed up the query process:

1. extraction and explicit storage of common geospatial properties,
2. text indexing, including commonly found variations,
3. aggregation of certain properties into new properties frequently requested,
4. calculating a score for each resource,
5. export of parts of the dataset into specialized search structures.

With regards to the extraction of geospatial properties, we focus on the common relationships *located in* and *nearby*. While the inverse *contains* relationship of *located in* typically does not need any parameterization, defining a threshold distance is required for the *nearby* relationship. In contrast to a static definition of this threshold, we assume that a type-based threshold, along with a few different proximity properties, can sufficiently represent this relationship. Other geospatial properties, such as *in the north of*, can be represented similarly, but initial query analysis suggests that such queries are rare in our use case. Consequently, preprocessing these properties would not have much benefits.

Users often ask for certain places or use sub-labels (incomplete representations of the actual entity label) to address things, so a well-defined text index is very important for achieving acceptable response times. As a first step, we can compute *label variations* using both generic and type-specific methods. For every label regardless of its type, we modify:

- umlauts, e.g., for "München" we create the variation "Muenchen"
- diacritics, e.g., for "El Niño" we generate "El Nino"
- parentheses and punctuation marks, e.g., "Leipzig/Halle (Airport)" we generate "Leipzig Halle", "Leipzig Halle Airport", etc.
- stopwords, e.g., for "Weiden in der Oberpfalz" we create the variation "Weiden Oberpfalz"

- geospatial references, e.g., "Hotel Adlon Berlin" leads to "Hotel Adlon" if the hotel has property *located in* "Berlin".

Depending on the resource type, we additionally perform type-specific modifications:

- adding or removing type-related labels, e.g., "Hotel Adlon Berlin" becomes "Adlon Berlin" or "Jena Paradies" turns into "Bahnhof Jena Paradies"
- type-specific predicates, e.g., adding or removing sub-labels for geospatial references such as *serves* of airports etc.

These label variants should be stored distinct from the original labels. In RDF, they can be stored using the SKOS properties *alternativeLabel* and *hiddenLabel*, so they can be distinguished later on. While users often ask for specific properties, such as whether a hotel offers wireless LAN or a shuttle service, other attributes of a resource are indicated less clearly. For example, some place suitable for cultural activities can either be nearby theaters, operas, cinemas, festival venues or similar resources. Consequently, it can make sense to capture these many logically disjunct features into a single property in order to reduce the search space.

When we have fused all information in one place, we can also precompute certain ranking metrics. For example, a resource with a large area, a higher population, or relatively many metadata attributes in general should be more important to the majority of users. Furthermore, for resources with a DBpedia equivalent resource (`owl:sameAs`) we can compute a pagerank of the related Wikipedia page and use this value for a generic resource score. Certain types can benefit from specific scoring. For example, each hotel in our hotel dataset contains a *popularity* attribute, which can be used along with the city of the hotel to compute a general query-independent score for each hotel.

Finally, when we have transformed the data set according to the methods above, we can further export them into index structures specialized for certain types of search, which can then be used by different search query interpreters. Currently, we expect to use the following types of indexes:

Named Entity Index returns primary information (labels, geospatially superordinate entities, geocoordinate, equivalent resource identifiers, score) for each resource. It can be used for searching all resources for a specific label and optionally specifying a type and/or region (geospatially superordinate resource) of the resource. For example, the JSON[2] representation of the index entry of resource Leipzig is shown below.

```
{
    Uri: http://data.unister.geoknow.eu/resource/geonames/2879139
    Label: Leipzig
    Score: 39.9704
    Type: [
        http://data.unister.geoknow.eu/ontology#City
        http://data.unister.geoknow.eu/ontology#ExtendedPlace
        http://data.unister.geoknow.eu/ontology#Place
        http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing
        http://data.unister.geoknow.eu/ontology#PopulatedPlace
    ]
    LatLong: {
        lat: 51.344742186880374
        lon: 12.374882105988624
```

[2] JavaScript Object Notation

```
        }
    }
```

Using specialized technologies such as Elasticsearch, we expect to get improved query response times.

Lookup Table only stores the highest scoring entity for each label, thus enabling a quick initial interpretation of query terms with the most likely entity. For example, since the representation above is the most common understanding of the term Leipzig, the lookup table would contain:

"Leipzig" $\Longrightarrow$ `http://data.unister.geoknow.eu/resource/geonames/2879139`

Polygon Store contains the geometries and related metadata relevant for the search engine, along with custom functions. After the export, the mapping between polygons and their respective resource representation in the triple store is annotated. Thus, a search on the polygon data store also yields respective resource identifiers. An example of polygon store information for Leipzig is shown below.

```
osm_id: 62649
landuse: administrative
name: Leipzig
natural:
way: POLYGON ((12.236209048692674 51.32321761110662, ... ))
uri: http://data.unister.geoknow.eu/resource/geonames/2879139
admin_level: 6
score: 39.9704
lastmodified: 2014-06-20
artificial: f
area_in_square_meters: 3.1610301E8
```

Bit Set Index contains a bit matrix for a limited set of common combinations of predicate-object columns (keys), and the (boolean) availability of this information for a set of resources, e.g., a all resources of a certain type (such as hotels in the tourism domain). A conjunction of featured keys in the interpreted query could be answered very quickly by such an index (or, if there are additional query aspects not contained in the bitset, the bit set would yield a candidate set), with additional columns for typical ranking criteria (score, stars, average price, etc.). The following table shows a few selected entries of a bit set index, indicating that `urn:hotel-1` is located in Germany and Leipzig, but does not provide wireless LAN.

	locatedIn:Germany	locatedIn:Leipzig	hasFeature:Wifi	price
`urn:hotel-1`	1	1	0	53.00
`urn:hotel-2`	1	0	1	65.00

Some statistics on the size of the resulting data sets can be found in appendix 5. In the following section, we will describe how the initial unstructured query has to be processed to enable a federated search across these different search indexes.

3 Query Interpretation

This section briefly goes through the prerequisites and steps needed to be done for transforming the original user query into a structured representation that can be used as input for the search engine.

Understanding what the user is looking for is the starting point in the retrieval process. In the e-commerce domain, the most common search pattern customer includes (1) a subject of interest, for example entities of type 'hotel', 'point of interest' or 'winter holiday' and (2) an idea about which properties this subject should fulfill, for example 'nearby mountains', 'with free wireless internet' or 'suitable for vegetarians'.

To achieve this level of understanding of the user intention, the initial user query is structured as a graph consisting of statements in the form of triples similar to plain RDF, with each node representing entities and an edge represents the relation between two nodes. For example, the user query "winter holiday with culture and mountains in southern Germany" is supposed to result in the a structured query as shown in figure 2.

Using such a structured query enables us to federate different aspects of the query to specialized search services, e.g., let polygon search handle the aspect "Hotel nearby mountains" (see section 4.1 and following for details on the search process) This section describes the process of generating such a structured query starting from a plain text user search query.

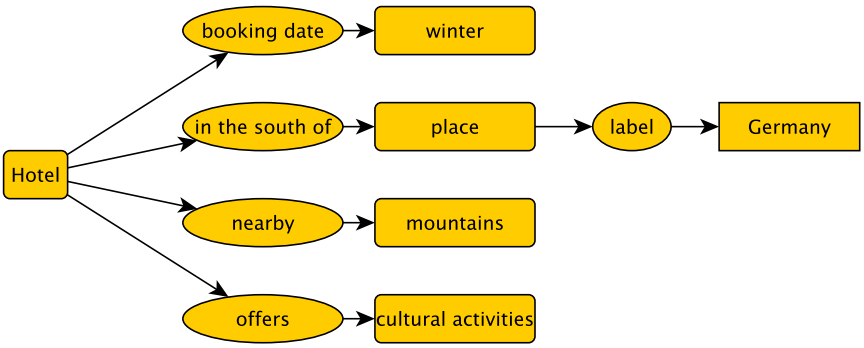

Figure 2: Example: structured query, represented as graph

3.1 Interpretation Domain

For the current system, a custom developed touristic ontology (TBox schema) is used to define knowledge on terminology, a set of concepts and their properties. One of the results of the data import and aggregation process described in section 2 results in data following this conceptualization (ABox schema). The result of the query interpretation process is a structured graph defining resources and relations, which contains resources and relations found in the entire ontology, including both TBox and ABox concepts. The analysis process is generic, so that exchanging the underlying TBox & ABox schematas to address other use cases is possible.

3.2 Syntactic Analysis of the Query

The first step splits a user's plain string query into tokens representing units of meaning such as named entities, verbs or adjectives. The major difficulty is to not separate words belonging to the label of the same named entity like "New York City" or "Adlon Kempinski Berlin". In order to do this, all possible segmentations are filtered using word ngram statistics to identify words frequently appearing next to each other. Additionally,

knowledge and pattern-based filters identify patterns such as dates, for example "in August 2014" or "Christmas next year".

The example query "winter holiday with culture and mountains in southern Germany" will be split into the following candidata segmentations:

- winter holiday | with | culture | and | mountains | in | southern | Germany
- winter holiday | with | culture and mountains | in | southern Germany
- winter holiday | with | culture | and | mountains in | southern Germany
- winter holiday | with | culture | and | mountains in southern Germany
- ...
- winter holiday with culture and mountains in southern Germany

For each segmentation, a first raw sequence of triples is generated by detecting relational words like "in" and "with". These structures are called "context" composed of "context statements". Each "context statement" is composed of exactly three tokens named "subject", "predicate" and "object" similar to plain RDF. For the example "winter holiday | with | culture | and | mountains | in | southern Germany" will be transformed into:

- Subject: winter holiday; Predicate: with; Object: culture
- Subject: *null*; Predicate: and; Object: mountains
- Subject: *null*; Predicate: in Object: southern Germany

Afterwards, ontology resources are mapped to each token of each context statement by fuzzy matching the query text to resource labels. An adapted full-text search index based on Elasticsearch is used, the similarity measure is the overlap of query words and words of entity labels as similarity measure. Due to the ambiguity of a query token, the number of candidates after this step might increase. For example "southern Germany" could match to a number of hotels with the same name, but also to the region in Germany.

In each step of this syntax-based query analysis, the number of candidates increases continuously: Each of the most likely segmentations can produce multiple contexts and each context can be multiplied by different matching ontology resources due to the ambiguity of entities.

3.3 Semantic Interpretation

The output of the last step consists of possible entities and relations the user could be looking for, however still only semi-structured. This section explains how to connect these imperfect and unconnected contexts.

3.3.1 Relation Detection

Relations the user has explicitly included in his query are connected to neighbouring entities. In this step, it is also considered that for some properties in sentences, their subject is not always obvious, as there can be more than one candidate complying with the syntax of the language and, in several cases, also being semantically plausible.

Example: The user types "Hotel nearby a museum with internet connection". It is not quite clear it is the hotel or the museum which should have an internet connection.

Additionally, it is made sure that a connection between two resources does not violate the TBox constraints (domain & range).

3.3.2 Relation Injection

Users are commonly used to search for information with keyword-based queries, meaning all key concepts are given, but they are not necessarily well-structured or connected with specific relations. For this reason, identifying explicit relations written by the user as described in the previous paragraph does not suffice for the analyzing process.

In order to properly interpret keyword-based queries, injecting plausible relations between concepts not explicitly connected is necessary. This can potentially lead to multiple interpretation candidates. Two concepts in the query defining concepts or instances are connected to each other if any of the properties defined in the TBox would fit, i.e. if the property is valid (see section 3.3.1)

3.4 Interpretation Candidate Scoring

The different steps described so far have produced a number of contexts, with several points in the process potentially increasing the number of candidates available:

- string query tokenization
- possible multiple plausible matches for each query tokenization
- multiple relation connection possibilities
- multiple plausible relations injected to connect entities

Therefore, a decision process is necessary to determine which interpretation is considered to be the correct one, or at least the most likely one . A classifier based on a set of scoring features, is responsible for this decision. Each score is defined as a specific characteristic of the resulting interpretation. Most of them are based on the structure of the resulting graph (e.g., number of connected components, shortest longest path, etc.) others consider the some aspects of the semantic of the query (e.g., the user looks for entities of the type Hotel). With this set of features, a manually annotated and continuously extended training and test set (or gold standard), a model is trained using a Weka implementation of an ensemble of boosted regression trees. This model is later applied to the real user input, identifying the candidate that is hopefully the best interpretation for the query.

3.5 Final Representation

The final output of the query analysis process is one context with the structured representation of the user query we consider the most likely. For our sample query, this would correspond to the following representation:

t_1	`urn:placeholder`	`rdf:type`	`urn:type-hotel`
t_2	`urn:placeholder`	`urn:season`	`urn:season-winter`
t_3	`urn:placeholder`	`urn:in-the-south-of`	`urn:placeholder2`
t_4	`urn:placeholder`	`urn:nearby`	`urn:type-mountain`
t_5	`urn:placeholder2`	`urn:sublabel`	"Germany"

3.6 Future Work

3.6.1 Alternate Interpretations

Due to the nature of the analyzing process, which iteratively adds more plausible candidates, the resulting set of candidates contains interpretations which are plain wrong or would not make a lot of sense from a semantic point of view. The interpretation classifier does a good job on sorting the candidates on plausibility, but there is currently no mechanism to determine which other ones (aside the winner candidate) would lead to a meaningful search. This queries could then be considered as alternate interpretations and would facilitate presenting the user with a choice if the winning interpretation was not what he had in mind. However, showing bad interpretations would break the purpose of the functionality due to bad quality. A future task is to keep increasing the quality of the classifier, so that meaningful interpretations can be easily differentiated from the ones that are not and to identify the correct threshold in order to not just consider the first interpretation, but a winning set of them.

Another limitation of the interpretation quality is the incompleteness of the knowledge base. In the syntactic part of the query analysis, we map tokens to entities. If no entity matches a token, we mark this token as unknown to search this in a different way like full-text search or give the user feedback that this part of the query is unknown. However, with a growing knowledge base, the possibility of false positives grows as well.

For example the query "hotel with cat" will not be correctly understood if "cat" as concept is not in the ontology. There is, however, at least one hotel with the name "cat". The best interpretation will be a hotel offering the feature "hotel cat". But this context is not valid and therefore the complete interpretation will be rejected. To mark an already identified token as unkown in an invalid context could lead to an imperfect but correct interpretation. We have to find proper rules which token of which candidate context is to mark and methods to validate also incomplete contexts.

4 Search Architecture

Now that we have a sufficiently interpreted query representation, we discuss the search architecture outlined in section 1.2 in more detail.

4.1 Motive-Based Search System Architecture

A more detailed overview of the search architecture outlined in section 1.2 is depicted in Figure 3. With a structured query as derived from the query interpretation process explained above, the *Semantic Search* service invokes required *Search Service* components, which return results according to the query aspects (context statements) they cover. Then, the results are merged by a *Chunk Reduction* component and returned to Semantic Search, who forwards it to the search application.

Not every Search Service has to be invoked, as explained below. Furthermore, with a bitset index in place it is also possible to learn context statement results not initially covered by the exported bitsets (see section 2), which improves the query response time on future similar queries[3].

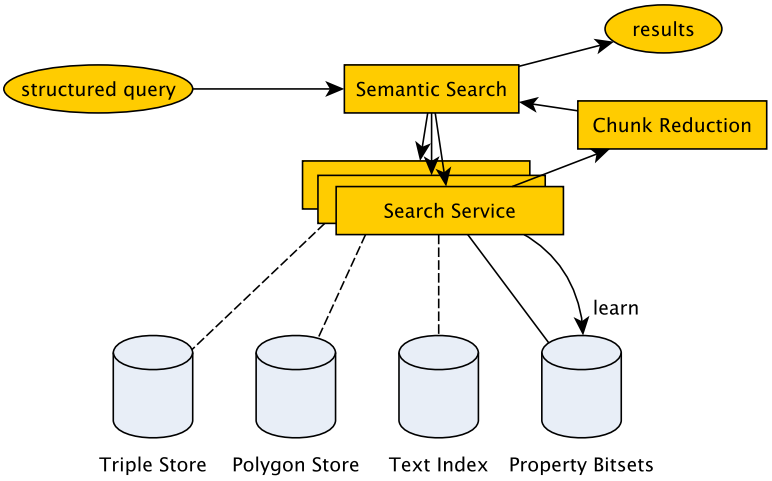

Figure 3: Search System Architecture

4.2 Search Architecture & Execution

In this section, we explain how the *Semantic Search* service utilizes the different search services.

4.2.1 Distributed Services

Searching for graph-structured information performs well on a triple store such as Virtuoso. Geospatial search and text search are, however, still not as performant or functional as in other systems. For the first prototype, we therefore decided to divide the search into 3 services:

- Triple store search, with a Virtuoso triplestore as backend

[3]This learning feature is not directly related to relevance feedback, which is discussed in section 4.5.2.

- Geospatial search, with a PostGIS enabled PostgreSQL database as backend
- Text search, with an Elasticsearch index as backend

With improvements to the Virtuoso triplestore (GeoKnow WP2), we plan to replace at least the geospatial search component in later revisions of the search infrastructure[4]. The basic architecture of the prototype search system consists of a central search executor receiving the structured query interpretation and a set of search services which are used by the central search executor to retrieve information from their respective endpoints (data stores).

4.2.2 Distributed Services Search

A query interpretation consists of statements (triples) forming a graph. Whenever there are several statements referring to the same subject, the search can be split over several services, with each service delivering its results for a subset of statements. The final result set consists of the intersection of the result sets provided by the search executions on the statement subsets.

The search engine builds a query plan by checking statement compatibility with the different search services, considering the possible split points and the selectiveness of the subset. The engine is then responsible to merge the results accordingly.

4.2.3 Chunk Reduction

To avoid sending a lot of information from the datastores to the search executor when splitting the search, a chunk-reduce approach was developed. The approach is loading a result chunk, a subset of the complete result size, from one of the services (called search service in this scenario) and then use the remaining services (called reduce services) to filter out the results which do not fit the statement subset assigned to the service. If not enough results were found in this first iteration, the next chunk is loaded. In the worst case scenario, all the chunks of the result set provided by the search service must be loaded and processed.

The search engine query planner is responsible of generating an optimal query plan with the aim of achieving a good balance between chunk loading time and maximal chunk numbers to load.

4.3 Property Bitsets

One of the most recurrent query interpretation types from a structural point of view is the star-shaped interpretation, in which all statements share the same object. An example of this could be an interpretation consisting of a concept denoting the type of searched entities (e.g. a Hotel) with a list of properties for this entity, e.g. 'in Barcelona, with Internet connection and swimming pool'. In order to boost the performance of such queries, bitset structures were used.

Having a well-known finite set of elements of the desired type (using the previous example, hotels), a matrix with the following characteristics can be built:

- each column represents entity of the supported type. A mapping index allows mapping the URL of an entity to its colum (bitset index)

[4]Some features, such as multipolygon support and specific functions like st_area, currently require us to handle these geometries in PostGIS. We are also working on prototypes integrating Elasticsearch-based text search more closely with Virtuoso in collaboration with OpenLink.

- for a given property, where the domain is equivalent to the type in question if the range is also a well-known finite set, a row (bitvector) is generated for each value
- if for all entities (bitset indexes) where a property value applies, '1' is set in the corresponding propertyValue bitset

With this matrix, searching for entities for a star-shaped interpretation looks like this:

- for each searched property-value tuple, retrieve the corresponding bitset
- intersect all retrieved bitsets, (logical AND operation on all bits from every bitset on position i -> position i in new bitset)
- the resulting bitset contains the desired entities and by extracting the ids from the mapping index, the search is finished
- if the entities were assigned to columns following some order, this order will be preserved after a search

With this approach, applying reduction (as explained in section 4.2.3) on a given set of hotels is trivial, since the only additional necessary logic is to generate a bitset containing the wished entities and add it to the set of bitsets to be intersected.

4.4 Scoring and Ranking

In order to show the potentially most relevant search results in the first positions, applying sorting is necessary. Otherwise, the user could become irrelevant results before relevant ones.

Supporting multiple sort criteria (possibility of sorting on different values) gives the user further flexibility to find his wished information, but it supposes an additional complexity to the described search process, since the information is split into different search services (see section 4.2.2), which can potentially support different sorting criteria, even depending on the type of the query. For example, a search for cities could be sorted by population, a search for hotels could be sorted by stars, but neither can cities be sorted by stars nor hotels by population.

In this case, the scheduler has to consider the wished sort criterion when deciding which service will execute the initial search (section 4.2.3). This can lead to situations where a user query cannot be answered with the wished sort criterion. A pragmatic solution to this problem is to select one of the supported sort criteria automatically when the query is introduced and give the user afterwards the possibility of repeating the search, but with another sort criterion, since the executor can provide the alternative supported sort criteria together with the initial result set.

4.5 Future Work

4.5.1 Single Backend Search

Using several search services (section 4.2.2) has the advantage of being flexible. Adding another datastore to the search means building a service to provide communication with the search executor. The process however is very ineffective, since a lot of intersection logic has to be done in the executor, leading to massive data flow in several cases. Therefore, better integrating the different search approaches is necessary in order to build a search executor able to search on big amounts of data fast.

In the area of geographical search, OpenLink is committed to keep improving the geo search tools of the Virtuoso triplestore in the GeoKnow project. Success in this area would give us the possibility of eliminating the geographical search service.

Text search is an area which is also of interest of other partners of OpenLink, therefore index integration is already in progress. In addition, Virtuoso provides ways to integrate own indexes in the system, so this would be a viable alternative in case of wanting to integrate the existing Elasticsearch implementation. In the same way could the property bitsets be integrated in the triplestore.

Once all this points are finished, the process could be simplified by removing the query planning and service query logic and just using a datastore, in this case the extended Virtuoso server.

4.5.2 Relevance Feedback

Users of the motive-based search provide implicit relevance feedback by their actions. By clicking on a search result, they indicate that this search result is relevant for them. Rephrasing the query shows that the results don't match their expectations, while query extension (i.e., adding more criteria) indicates that the objective of a motive-based search (leading the users through a large amount of data, iteratively refining the search query until the results match their needs) is successful.

Such relevance feedback should be fed back into the system for improving the search results. We plan to record result clicks, which indicate a high relevance of the result, and feed the click statistics back into the preprocessed and runtime scoring and ranking process (sections 2 and 4.4). As a result, a resource which has more clicks per impression than other resources will receive a higher score in the next iteration of preparing the data set.

5 Summary

In this deliverable we discussed a search architecture suitable for motive-based search in the e-commerce domain. Although the architecture and related services have been designed particularly with regards to the work package 6 usage case and the tourism industry, most of the components are generic and perfectly applicable in other domains.

We presented a motivation for motive-based search and a high-level architectural overview of the workflow. We further elaborated data transformation and aggregation tasks required to prepare optimized search structures, which include preprocessing of geospatial properties and label variations for accelerating query response time and improving result accuracy with regards to the recall.

Next, we discussed our approach towards transforming an unstructured query into a structured graph-like query model, closely related to the RDF data model. Finally, we presented details on the search process and the interaction between the search services, including future work on the architecture with regards to expected ongoing improvements in other GeoKnow work packages.

Based on this architecture, we plan to implement an initial prototype and evaluate it based on component and usability test metrics to be defined in task 6.4.

Data Set Statistics

The following metrics have been generated from current prototype datasets in the E-Commerce use case as of 2014/06/26.

Triple Store

Database required for import, aggregation and transformation:

Triples: 643.818.537[5]

Resources: 637.115

Virtuoso database file size: 48GB

Database exported for search infrastructure:

Triples: 45.349.668[6]

Resources: 1.446.037

Virtuoso database file size: 9.1GB

Polygon Store

Database required for import, aggregation and transformation:

Resources: 1.123.615

PostgreSQL database file size: 21GB

Database exported for search infrastructure:

Resources: 1.187.830

PostgreSQL database file size: 3.9GB

Named Entity Index / Lookup Table

Resources: 636.188

Resources in Lookup Table: 32.670.014

Elasticsearch database file size: 3.9GB

MongoDB file size: 13.9GB

[5]This metric neither includes larger source graphs already in RDF (DBpedia, Geonames, LinkedGeoData) nor geometries except latitude and longitude geocoordinates, but different versions of the same resource.

[6]This metric does not include geometries except latitude and longitude geocoordinates.

References

[1] Orri Erling and Ivan Mikhailov. Rdf support in the virtuoso dbms. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.

[2] Erik Hatcher, Otis Gospodnetic, and Michael McCandless. Lucene in action, 2004.

[3] Rafal Kuc and Marek Rogozinski. *Elasticsearch Server*. Packt Publishing Ltd, 2013.

[4] Bruce Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

[5] Regina Obe and Leo Hsu. *PostGIS in action*. Manning Publications Co., 2011.