Social Semantic Emotion Analysis for Innovative Multilingual Big Data Analytics Markets

D5.4 Social Semantic Knowledge Graph Infrastructure and API, Final Version

<table>
<thead>
<tr>
<th>Project ref. no</th>
<th>H2020 644632</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project acronym</td>
<td>MixedEmotions</td>
</tr>
<tr>
<td>Start date of project (dur.)</td>
<td>01 April 2015 (24 Months)</td>
</tr>
<tr>
<td>Document due Date</td>
<td>31 Dec 2016 (Month 12)</td>
</tr>
<tr>
<td>Responsible for deliverable</td>
<td>SindiceTech</td>
</tr>
<tr>
<td>Reply to</td>
<td><a href="mailto:giovanni@siren.solutions">giovanni@siren.solutions</a></td>
</tr>
<tr>
<td>Document status</td>
<td>Final</td>
</tr>
<tr>
<td>Project reference no.</td>
<td>H2020 644632</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Project working name</td>
<td>MixedEmotions</td>
</tr>
<tr>
<td>Project full name</td>
<td>Social Semantic Emotion Analysis for Innovative Multilingual Big Data Analytics Markets</td>
</tr>
<tr>
<td>Security (distribution level)</td>
<td>PU</td>
</tr>
<tr>
<td>Contractual delivery date</td>
<td>31 Dec 2016</td>
</tr>
<tr>
<td>Deliverable number</td>
<td>D5.4</td>
</tr>
<tr>
<td>Deliverable name</td>
<td>Social Semantic Knowledge Graph Infrastructure and API, initial Version</td>
</tr>
<tr>
<td>Type</td>
<td>Other</td>
</tr>
<tr>
<td>Version</td>
<td>Final</td>
</tr>
<tr>
<td>WP / Task responsible</td>
<td>WP5 / SindiceTech</td>
</tr>
<tr>
<td>Contributors</td>
<td>Giovanni Tummarello, Renaud Delbru, Stephane Campinas, Alfredo Milani Comparetti, Fabio Corneti, Szymon Danielczyk</td>
</tr>
<tr>
<td>EC Project Officer</td>
<td>Martina Eydner</td>
</tr>
</tbody>
</table>
## Table of Contents

Executive summary.................................................................................................................................................. 6

1. The new Kibi Knowledge Graph Data Model: overview .................................................................................. 7
   1.1. Architectural Overview ......................................................................................................................... 7
   1.2. Logical Architecture ............................................................................................................................... 8
   1.3. Toward Consumption APIs infrastructure: SQL API - Calcite vs Anchormen .................................... 9
      Usage ............................................................................................................................................................. 10
      Extensibility ................................................................................................................................................ 10
      Model ......................................................................................................................................................... 10
      SQL Query .................................................................................................................................................. 11
      Conclusion .................................................................................................................................................. 12
   1.4. Native Databases support: federated query layers .................................................................................. 12
      OVERVIEW .................................................................................................................................................. 12
      SPECIFICATIONS ...................................................................................................................................... 13
      Mapping Relational Model to Document Model .......................................................................................... 13
      Advanced Search: Full-Text, Spatial, etc ...................................................................................................... 14
      Extending ElasticSearch API ....................................................................................................................... 14
      Action Filters .............................................................................................................................................. 14
      Registering of a Database Schema Alias ..................................................................................................... 15
      Handling Non Supported Actions .............................................................................................................. 15
      SQL Query Planner .................................................................................................................................... 15
      Parsing a SearchRequest .............................................................................................................................. 15
      Parsing the SearchSource ............................................................................................................................ 15
Convert SearchRequest into a Logical Model ......................................................... 16
Convert Logical to Physical .................................................................................... 16
Search Response .................................................................................................... 16

2. In action: the Emotion/News/Knowledge graph demo (Social TV) ..................... 16

3. The design choices of the high performance join component............................ 18
Caching of Join Computation .............................................................................. 19

OVERVIEW ........................................................................................................... 19
GOALS .................................................................................................................. 19
ANALYSIS ............................................................................................................ 19

Shard Request Cache .......................................................................................... 20
Index Query Cache ............................................................................................... 20
Basic Mechanism .................................................................................................. 20
The Caching in the Siren Platform: the Join Clause ............................................ 20
Distributed Cache Lookup ..................................................................................... 21
Segment Locking .................................................................................................... 21
Cache Entry Locking ............................................................................................. 21
Pseudo Algo ............................................................................................................ 21
Cache Refresh ......................................................................................................... 22
Advanced Distributed Cache Lookup .................................................................. 22

4. Social Network Extraction and Linked Data Publication .................................... 22
Architecture .......................................................................................................... 23
API .......................................................................................................................... 24
Metrics and analytics ............................................................................................ 27
Communities .......................................................................................................... 28
Community Detection ............................................................................................ 28
Sentiments and Emotions in Communities ........................................................................................................29

Step 1. Loading social-media ..........................................................................................................................30

Step 2. Retrieving data from Scaner ..............................................................................................................33

Step 3. Finding relevant tweets online ...........................................................................................................42

5. Semantic Knowledge Graph Infrastructure and API ..................................................................................44

Architecture ....................................................................................................................................................44

API .................................................................................................................................................................46

Kibi graph browser ........................................................................................................................................49

Summary .........................................................................................................................................................56

6. Conclusions ..................................................................................................................................................57
Executive summary

The present document is the final deliverable of WP5 describing the API and approach toward providing Semantic Graph capabilities and Socially derived data in the platform of MixedEmotions.

In the first part of the contribution, we describe how the original ideas that led to the first Kibi platform (described in the first deliverable) evolved into a more powerful and sophisticated system capable of providing big data streaming, semi-structured knowledge graph capabilities and APIs. The new stack includes the new Siren “Platform”, a sophisticated system that uses Relational Algebra technology to both be compatible with (virtualization) external SQL sources and to provide output SQL/Gremlin APIs. Secondly, we demonstrate this in action on DW’s data processed by Expert System in the context of the Social TV pilot demonstration. In the third part of the contribution we discuss the objective of the Social Context Analysis module, that is to provide a platform of analysis for social media and social context. The schemas used by this module follow the ontology defined in D5.2 Data Modelling for the Social Semantic Knowledge Graph, Final Version. We finally demonstrate in action the Knowledge Graph tools (link analysis browser) that are also included in the platform, and how they are used to interact with the underlying knowledge graph, which was created by enriching the data from external knowledge sources.
1. The new Kibi Knowledge Graph Data Model: overview

1.1. Architectural Overview

In the previous deliverable, we presented the version 1 one of the Knowledge Graph infrastructure. This was built as follow:

1. A basic model based on Tables and Relationships is defined
2. This model is interpreted in two ways:
   a. By a driver implementing the Gremlin knowledge graph language
   b. By the a Relational User Interface (Kibi) leveraging which would then use a backend Relational Engine (the Siren Join) which operated on top of ElasticSearch

The use of ElasticSearch (also mentioned as ES in this document) guaranteed that the final system can go across text and structured data, now with mindfulness of the structure (knowledge graph). This deliverable describes how we evolved from this first version into a new Kibi Knowledge Graph data model. The general model is depicted in the figure below.
The diagram illustrates the following components of the new “Siren Platform”:

1. The main part (described in Chapter 2) which has a completely new methodology to perform high scalability relational joins.
2. The SQL endpoint methodology - how the platform can offer its services via SQL interface. (described in 1.3)
3. Virtual ES/JDBC Alias - how the platform can make an external JDBC data source look like a virtual ElasticSearch Index (described in 1.4)

The graph also illustrates how the NodeJS backend uses the ES+Siren language but Kibi / Sentinl (the alerting component that Kibi now has https://github.com/sirensolutions/sentinl) can also use directly the new offered SQL APIs. Last, in the diagram it is shown that ElasticSearch can in fact be used also as primary storage for reindexing data. This is considered as a core feature when one has explorative scenarios and initially not well known datasets: reindexing is then often required.

### 1.2. Logical Architecture

In 1.1 we discussed the component diagram forming the new Knowledge Graph infrastructure. In this section we will see how these reflect into the user experience.

The following diagram illustrates this:
Here we can see that the Siren Platform does query federation from SQL SPARK (via SQL) and ElasticSearch.

The relational capabilities are then *Ontologically Mapped* (simple table to table relational model turned into *entity+relations*), via the relational configuration.

At the same time one can then defined “saved searches” which are similar to views and these are in turn shown in Dashboards.

Last but not least the entire knowledge graph is offered for Consumption by SQL/Gremlin or core siren Platform APIs as described in the next chapter.

### 1.3. Toward Consumption APIS infrastructure: SQL API - Calcite vs Anchormen

In this section we discuss the work done in order to determine the infrastructure to use to implement the “consumption APIs” that is the ability to offer a unified SQL interface for the platform.
The need is to map the internal data model (which is similar to ElasticSearch Indexes) to SQL. In the previous deliverable we covered the mapping to Gremlin (via the unipop infrastructure).

This section presents an analysis of the functionalities of Anchormen/sql4es\(^1\) and of Calcite\(^2\) in relation to the implementation of an SQL API that gives access to all the Kibi Semantic Knowledge Graph.

### Usage

<table>
<thead>
<tr>
<th></th>
<th>Calcite</th>
<th>Anchormen</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Programmatic</td>
<td>Yes</td>
<td>Not really</td>
</tr>
</tbody>
</table>

- Both Calcite and Anchormen provides a JDBC API. However, it looks like it will be easier in Calcite to reuse programmatically components (e.g., SQL parser) outside the JDBC interface.
- Sql4es is based on the Presto sql parser. Calcite provides its own SQL parser, which is used in various projects: Drill, Hive, Samsa, Storm, Solr, etc.

### Extensibility

- Extensibility is important especially when we have to implement join and multi-backend functionalities.
- Anchormen (and NLPChina) are custom made libraries which might be more time consuming to extend.
- On the other hand, Calcite has been designed to be extended in many places (sql functions, schema, adapters, etc.).
- From our experience, it will be less time-consuming to extend Calcite with these functionalities than extending Anchormen or NLPChina.

### Model

- The index is mapped to a database, the type is mapped to a table.
- It is designed to execute one query against one index/alias. It is not possible to query multiple indices in the same query:
  - “Only types part of the active index or alias can be addressed in the FROM clause.”

\(^1\) https://github.com/Anchormen/sql4es
\(^2\) https://calcite.apache.org/docs/algebra.html
- Does not support alias for types
- Does not support index patterns and type patterns
- A Calcite’s extension as part of the Siren platform has been written to register index and type patterns as database and table aliases. This can be reused in our SQL API.
- Anchormen is more limited with respect to the data model, and it will require extensions to support query with multiple indexes, and with index and type patterns.

**SQL Query**

We compare the SQL query functionalities of sql4es and Calcite. Calcite provides an ElasticSearch adapter which supports basic functionalities.

<table>
<thead>
<tr>
<th></th>
<th>Calcite</th>
<th>Anchormen</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>WHERE Conditions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>OFFSET</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>LIMIT</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Advanced Search</td>
<td>Partial</td>
<td>No</td>
</tr>
<tr>
<td>Join</td>
<td>Partial</td>
<td>No</td>
</tr>
</tbody>
</table>

- The difference in functionalities between Calcite and Anchormen are small. GROUP BY operator in Calcite can be implemented.
- Anchormen supports also write operations, e.g., UPDATE, CREATE, etc., which is out of scope for typical read-only scenarios.
- NLPChina provides SQL functions to write advanced search queries, e.g., a match query or range, and special aggregations, e.g., data histogram. This could be used as example for extending the SQL syntax and supports advanced functionalities of ElasticSearch.
We extended Calcite with SQL functions to write advanced search queries. This can be reused.

We extended Calcite with a Semi-Join operation for ElasticSearch. This can be reused.

Conclusion

- While Anchormen and NLPChina might provide at first look a better support of basic SQL query operators, the difference in functionalities with Calcite is not that important. On the other hand, Calcite provides better extensibility and existing work done in the Siren platform could be already reused. As a conclusion the amount of effort to work on Calcite seems lower and as such it is the preferred infrastructure.

- Anchormen and NLPChina provide good examples of possible SQL to ElasticSearch mapping that can be reused in the Calcite implementation.

- Calcite is also a more mature project, backed by numerous large projects (Hive, Solr, Drill, etc.)

- Moreover, we could envision contributing our ElasticSearch adapter to Calcite so that it is maintained by the community.

1.4. Native Databases support: federated query layers

One of the results of the work in 2016 was the strong demand for direct federated knowledge graph operations vs the ETL that is required in Kibi 4.x.

This led to the conclusion that the new system had to allow direct federation of external sources, typically SQL systems (and in turn systems like Spark). This section deals with the design of the SQL to drivers which allow the Siren Platform (See next section) to use external databases as virtual tables.

OVERVIEW

We will analyze how to create a SQL driver plugin for ElasticSearch. The SQL driver should give the ability to access a SQL table as an index in ElasticSearch, so that clients such as Kibi can access it and issue search requests against it.

The admin and search requests issued by Kibi should be converted, when possible, into a SQL request against the SQL backend. The response of the SQL backend should be translated back into an ElasticSearch response.

The goal is to reuse as much as possible of the ElasticSearch code base to handle REST actions, parse requests and queries, format responses.

The SQL plugin can be seen as a module of the Siren platform which can be used independently. The query planner of the Siren platform that is responsible for planning, and executing joins will rely on this module by executing search requests against it. The SQL plugin will have its own calcite engine to build, optimize and execute SQL queries.
SPECIFICATIONS

This section analyses the mapping between the ElasticSearch and SQL model, and describes how the SQL plugin will extend the main API components of ElasticSearch.

Mapping Relational Model to Document Model

ElasticSearch model is based on a document model where a document is uniquely identified and is composed of one or more fields. Each field can have various data types, from simple (numeric, text) to complex (array, object).

A straightforward mapping between relational to document model can map “record” to “document” and “column” to “field”. Mapping complex datatypes such as array or object will be dependent on the SQL backend (for example, postgresql supports such complex datatypes).

In a relational database, records are organised into tables and tables into schemas, while in ElasticSearch the documents are organised into types, and types into indices. We can map “schema” to “index” and “table” to “type”.

3 as it is currently done in the internal relational model of the siren platform
Advanced Search: Full-Text, Spatial, etc.

ElasticSearch provides advanced query capabilities such as full-text search and spatial search. Traditional relational databases do not support such advanced queries. Trying to map full-text search query to a SQL query using regexp and the likes is not recommended for various reasons: increase of complexity of the query rewriting, performance, potential unexpected results, etc.

Therefore, such features should be mapped only if the SQL backend supports them (likely with user defined functions). This does not have really impact on Kibi, since most of the queries generated by Kibi do not rely on full-text search apart from the search bar. But in this case, if a user tries to write a full-text search query, we can simply display a warning if such feature is not supported by their SQL backend.

Extending ElasticSearch API

A request is processed by ElasticSearch as described in the figure below. The rest request is received by the RestController and delegated to the appropriate RestHandler. The RestHandler then executes the associated Rest*Action which itself executes the appropriate Transport*Action through the Client. The TransportAction applies a chain of ActionFilter to the request before its execution. An ActionFilter allows to filter the request by modifying it or to abort the execution chain.

The ActionFilter will be used to inject our custom logic for the SQL plugin. The rest of the workflow can be reused as it is. For example, if a SearchRequest is sent to ElasticSearch, the request will be processed by the previous workflow and will lead to the execution of a RestSearchAction. Let’s assume we have added a SearchActionFilter with our custom logic. The SearchRequest will first go through all the ActionFilter, and reach our SearchActionFilter. The SearchActionFilter will check if the index specified in the SearchRequest maps to a registered database schema. If it is the case, then the SearchRequest is delegated to our SQL engine and the execution of the chain is aborted. The SQL engine will then return a SearchResponse through the ActionListener that will be propagated back to the user.

Action Filters

We need to implement an ActionFilter for each ElasticSearch Action we want to extend. For example:

- SearchAction
- GetMappingAction
- GetIndexAction
- IndicesExistsAction
- TypesExistsAction
- Etc.

Each ActionFilter will contain the logic to:
- convert the ActionRequest into a sql query,
- execute the query against the database using jdbc,
- convert the database response into an ActionResponse

Registering of a Database Schema Alias

We need to provide an ElasticSearch service that will be responsible in:
- Managing (add, delete, get) sql index aliases, where an index alias is a mapping to a jdbc url and a database schema.

We might want to add a convention to the index alias created by the SQL plugin, e.g., a prefix ‘sql:’ so that ActionFilter can easily detect that one of the specified indices is a sql alias.

Handling Non Supported Actions

In case an action is not supported, i.e., we haven’t implemented its ActionFilter, then it is likely that ElasticSearch will just return an “index not found” error message, since the sql index alias will be unknown.

SQL Query Planner

The SearchActionFilter will rely on a SqlQueryPlanner to:
- Parse the search request and the search query and convert them into a logical model.
- Optimise and convert the logical model into a physical model
- Execute the physical model using JDBC
- Convert the JDBC result set into a SearchResponse

Parsing a SearchRequest

The parsing of the json SearchRequest is performed by the RestSearchAction. The SearchActionFilter will receive a SearchRequest object.

Parsing the SearchSource

The parsing of the SearchSource (i.e., the search query) is performed by the RestSearchAction too. The SearchRequest object contains a SearchSourceBuilder object which is a representation of the search query. It is composed of various query elements objects: QueryBuilder, AggregateBuilder, etc.
Convert SearchRequest into a Logical Model

Converting a SearchRequest into a logical model will consist in traversing the search request and the search source builder, convert each query element into a logical operator, and build a logical model that is equivalent with the SearchRequest. To keep the Calcite class name convention, we will call this class EsToRelConverter.

To traverse the SearchRequest and the SearchSourceBuilder, the siren platform provides a SearchRequestVisitor that implements the basic logic to traverse these two objects using a DFS approach.

The EsToRelConverter will implement such a visitor interface, and build a tree of RelNode during the traversal. The RelNode in calcite is the abstract class for all the logical operators.

The SearchRequest object contains information regarding the indices and types, and will therefore be mapped to a LogicalScan.

The SearchSourceBuilder contains all information regarding the query, such as size, sort, query, aggregate, etc. The sort parameter will be mapped to a LogicalSort, the query to a LogicalFilter and a list of conditions (RexNode), the aggregate to a LogicalAggregate, etc.

Convert Logical to Physical

To optimise and convert the logical model into a physical one, we can probably reuse the existing Calcite planner engine used for executing SQL queries. We just need to create a version of this engine that accepts a RelNode object as input instead of a SQL query (i.e., we skip the sql parsing phase).

Calcite provides a JDBC Convention that will implement a logical model into a physical one based on JDBC.

Search Response

Once the SQL query has been executed, we need to convert the result set into a SearchResponse, e.g., converting records into a SearchHits object, or converting aggregate into an Aggregations object. Most of these objects can be reused and we can build them manually while streaming the result set.

2. In action: the Emotion/News/Knowledge graph demo (Social TV)

The ability to interpret text and structure as well as to make big data relational joins inside the search engine is witnessed by the Social TV Backend Demo where the stream of news from DW is annotated by the Expert System infrastructure and then visualized and browsed inside Kibi.

In the first Dashboard below we can see an analysis of the tweets. An operator can restrict along any analytics dimension from time to the use of full text capabilities (e.g. searching in the search box). The
results of the initial emotional analysis as well as the tags associated with each category is shown in the large histogram, to the left.

The relational capabilities are demonstrated by the button which allows pivoting from the current set of documents (Tweet) to the connected entities in the knowledge graph in this case “entities” which are extracted from the tweet.

We then proceed with the demonstration as follow. We select “Joy” as an emotion. This restricts to a lower number of tweet and entities. We then pivot to these and restrict to Product. The result is the list of products which are associated with tweets that are showing a feeling of “Joy”.

![Image of demonstration]

**Image Description:**
- The image shows a demonstration of how to select and analyze emotions using mixed emotions software.
- It includes a large histogram with emotions on the y-axis and count on the x-axis, showing the distribution of emotions.
- A table is also visible with columns for time, tweet, emotion/category, and hashtags.
- A button is highlighted for pivoting from the current set of documents to connected entities in the knowledge graph.

**Text:**
We then proceed with the demonstration as follow. We select “Joy” as an emotion. This restricts to a lower number of tweet and entities. We then pivot to these and restrict to Product. The result is the list of products which are associated with tweets that are showing a feeling of “Joy”.

---

D5.3 Social Semantic Knowledge Graph Infrastructure and API, final Version
3. The design choices of the high performance join component

During the last half of 2016 we have reimplemented from scratch the joined capabilities within Kibi which are Key to the semantic relational capabilities. The new Infrastructure is called “Siren Platform”

The Siren platform is an in-memory distributed computing engine that augments ElasticSearch with Relational capabilities. It is deployed as an ElasticSearch plugin and it has been built for computing low latency analytical queries that can also join ElasticSearch indices whenever possible.

Siren is highly optimized. Filters are pushed down to ElasticSearch indices prior to computing the joins. Values involved in the join conditions are read from the filtered subset of documents and projected in an in-memory distributed file-system. A partitioning and shuffling phase is distributing values across all the available nodes of the cluster. The joins are then computed in a fully distributed manner and in memory across all the available nodes. This solution scales horizontally: adding more computing nodes will reduce the processing time.

The result of the join (a set of document ids) is then used to filter the target ElasticSearch index and to return the answer of the complex query in the form of an ElasticSearch search response. The result of a join can be cached very efficiently using bit set (a list of doc ids) on an index segment level: in case of an index update, only the new index segments will be joined, reusing previous computation.
The platform includes a query planner based on Apache Calcite. It includes a cost-based optimizer that is able for example to select the optimal implementation for a join operation or to reorder joins in order to speed up processing. Finally, the Siren platform implements optimised join operator that operates directly on binary data located in the distributed file system in order to avoid unnecessary data copying and serialisation.

These are some of the key design features for achieving low latency, flexible join queries. Based on these, Siren will deliver the required performance and also provide ample flexibility for future requirements.

Caching of Join Computation

OVERVIEW

The computation of joins is one of the most expensive operations in the siren platform. Being able to cache and reuse the computation of joins across queries is critical for ensuring low response time. In addition, in the context of Kibi, it is very common to have identical intermediate joins between queries in a same user session. This is due to the specific data access patterns that are generated when the user is navigating the dashboards. The user is usually exploring and restricting the data collection step by step. Each step is usually based on the state from the previous step, which means that it is based on the same query but with additional restrictions. Being able to reuse the computation of the previous step will allow providing a fast interactive navigation to the user.

GOALS

The goal is to create an efficient mechanism for caching the computation of a join. The core functional requirements are:

1. Low memory footprint: the cache entry for a join must be as lightweight as possible. Caching mbytes or more for a single entry is not optimal.
2. Aware of index changes: if an index changes, all the cache entries related to this index should become invalidated.
3. Work in a shared-nothing infrastructure: each node is independent from the others and each node has its own cache.

ANALYSIS

There are two main approaches to implement a caching layer (as implemented in ElasticSearch):

- Caching on a shard request level
- Caching on an index query level
Shard Request Cache

The shard request caching is not really appropriate because it would mean that we will have to cache the projection of a join. The projection is in its base form a collection of tuples where each tuple is composed of one or more numeric columns. The data generated by a projection is generally in the order of mbytes and can easily reach hundreds of mbytes.

Index Query Cache

The index query caching is more appropriate in term of memory footprint as only the doc ids (bitset) resulting of the join query is cached. The index query cache is based on a per-segment query cache, i.e., each segment will have its own cache. This has an important consequence on the cache refresh against index updates: we need to recompute the cache of the segments that have been updated. This is a property we can use in the future to improve cache refresh (e.g., by performing a join between the segments that have changed, and not the full index). However, we will assume for the moment a simple cache refresh policy that will perform the full join in case of an index update.

Basic Mechanism

The execution of a query (being a join or not) can be decomposed in two phases: search/select and fetch/project. In a standard search query as in ElasticSearch, the phases are called search and fetch. However, a join query as in siren, the phases are called select and project. There are similar in essence but have different outputs. The fetch phase will fetch the stored documents and will return them in the search response. The project phase will scan certain fields of the documents and send their values in the form of a tuple to the siren’s Data Staging.

In the search/select phase, the query clauses are evaluated and a set of doc ids representing the set of documents matching the query is computed. In the fetch/project phase, the set of doc ids are used to read the content of the matching documents. The query cache is in fact caching the set of doc ids from the search/select phase. In case of a cache hit, the search/select phase is skipped. In practice, when computing join, the select phase is the most expensive one to compute as this is the one that will evaluate the join clause. Therefore, the query cache level is the most optimal one since it will help skipping heavy computation for a very low memory footprint.

The Caching in the Siren Platform: the Join Clause

When evaluating a query, we can cache each query clause individually including the join clause. This has the advantage that the same join clause can be reused in a different query, therefore increasing the cache hits.

However, evaluating the full join clause individually without leveraging the restrictions from the other query clauses (i.e., leveraging a smaller search space) will be generally be more expensive, since it is
similar to evaluating the join clause on the full index. This overhead can be reduced once we have the more advanced cache refresh mechanism based on segment cache refresh.

We can propose an option to activate or not the caching of the individual join clause. For certain relations where the indices are not huge, it could be beneficial.

**Distributed Cache Lookup**

Given that we are in a shared nothing architecture, the query caches on each shard are independent from each other, i.e., their state is not synchronized among themselves and therefore can be different at my point in time. For example, a cache entry can be evicted in one cache and not in the other for various reasons (e.g., a different replica was previously queried) or a node was restarted and its query cache is empty.

In this context, the coordinator needs to first check that the cache entry exists on all the shards. If one of the shards returns a cache miss, then it knows that it needs to execute the join computation (which means computing the projection of the sub-request and probably trigger other join computation downstream). Otherwise, it knows that it can optimise the query execution by reusing the cache entry for the select phase. Given that the cache lookup is performed in two phases, a first request to check the existence of the cache entry then a second request to execute the query using the cache entry, we need to “lock” two elements:

- the cache entry so that it does not get evicted between the two requests;
- the segments in case of a segment merge, segment creation, etc.

**Segment Locking**

Such a lock concept exists in ElasticSearch and it is used by the scroll API. When using the scroll API, a search context is kept alive for a certain time. Subsequent scroll requests use the scroll id to refer to the search context that is kept alive. Internally, a search context keeps a reference to the index reader. An index reader provides a fixed view over an index (until the reader is reopen) and keeps a lock on the segments, so that those segments are not deleted as long as the reader is open.

**Cache Entry Locking**

We could use the search context to store the reference of the cache entry and ensure that even if it is evicted from the main query cache, they will be available in the search context.

**Pseudo Algo**

The coordinator traverses the query plan using a DFS approach. For each query node:

1. Send request to check existence of cache entry and acquire a search context lock
2. If one of the shard response indicates a cache miss
   a. send request to release the search context lock
b. resume the traversal of the query plan.

3. If all the shards acknowledge a cache hit
   a. send request to execute the optimised query (query that will release automatically the
      search context lock at the end of the processing)

Cache Refresh

The cache refresh policy for a join query must take into account two elements: index updates on both the left and right relations of the join. The index updates on the left relation will be implicitly taken into account by the index query cache itself. In order to implicitly take into account the index updates on the right relation, we can use the same technique than in siren join, i.e., the cache id must be based on the index version of the right relation.

Given that the cache id is based on the join query, and the join query is based on a data input id, this means that the data id created by the left relation must be based itself on the index version.

Advanced Distributed Cache Lookup

We can optimise the distributed cache lookup by not considering a cache miss whenever we encounter a segment cache miss on a shard. Instead we could return to the coordinator the ids of the segments where the cache entry is missing, and the coordinator could launch a smaller join by joining only the missing segments with the right relation.

This would mean that we will likely have to keep the search context lock longer, since we will have to lock it for the duration of the join.

Also, in this scenario it will be difficult to ensure that the right relation being joined with the missing segments is the same than when it was joined with the other segments. In between, the index of the right relation might have been updated. But this does not really have impact on the final responses (a part of the query will be more up to date than the rest).

4. Social Network Extraction and Linked Data Publication

The objective of the Social Context Analysis module, which we refer to as Scaner for short, is to provide a platform of analysis for social media and social context. Scaner receives as input social media content (e.g. tweets) from an external source. It stores this content in a graph database, extract relevant information related to this content and the users related from social networks, and process all that information in order to analyze the social context of the social media content provided. The service offers

4 If the execution of the optimised query fails, we could resume the traversal of the query plan but this might lead to computing twice some operations. Or we could just return as a response the failure which is anyway probably due to overload issue on the cluster.
a REST API that exposes the results of the social analysis and allows getting information about individual users or content, groups, topics, etc. The schemas used in the responses and the contexts used in the JSON-LD documents follow the ontology defined in D5.2 Data Modelling for the Social Semantic Knowledge Graph, Final Version.

Architecture

The architecture of the Scaner module is composed of four main parts: the Graph Database, the Processing module, the Scraper/Crawler, and a web server that exposes a REST interface, as shown in the figure below:

- **Graph Database.** A Repository where all the information will be stored, using graph structures. After evaluating two of the most popular ones, Neo4j and OrientDB, OrientDB was selected because of its friendly licensing. In addition, OrientDB provides a hybrid graph/document-oriented database that is suitable for storing both user and social media entities as well as their social network.

- **Processing module.** This module is responsible for accessing and processing the information stored in the database. It is responsible for running the different metric calculation algorithms, as well as making calls to the Crawler when additional information is needed.

- **Crawler.** This module is responsible for accessing to social networks to respond to requests for information from the processing module. The downloaded data is stored in the database for later use.

- **Web Server.** This module offers a web server that makes the data accessible from the outside through a REST API. The interface has been defined with OpenAPI [https://openapis.org/]. OpenAPI provides an API definition language that is independent from the programming language, as well as many tools that integrate with this definition and provide features such as automatic web server generation or validation. In particular, the tools used in our Web Server provide a layer of request validation for all the calls and responses from the server, which ensures that the interface of the web service adheres to the specification. This tool also provides an interactive web interface which allows making custom calls to try the API.
The REST API has been structured around five resources:

- **Topics**: subject or topic to classify social media (e.g. set of hashtags, special scenario).
- **Users**: online accounts that publish social media in a topic.
- **Tweets**: tweets posted by the users in the social network.
- **Tasks**: processes queued or in execution in the service.
- **Communities**: users that publish within a given topic are split into communities based on their relationships and interactions.

The table below contains a summary of the methods in the API. The full definition, the schemas used for validation and usage examples, are available at the API demo endpoint:


<table>
<thead>
<tr>
<th>Description</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topics</td>
<td></td>
</tr>
<tr>
<td>Obtain information of a particular topic</td>
<td>GET /topics/{topicID}</td>
</tr>
<tr>
<td>Obtain list of available topics</td>
<td>GET /topics</td>
</tr>
<tr>
<td>Obtain social network of a topic</td>
<td>GET /topics/{topicID}/network</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td><strong>Users</strong></td>
<td></td>
</tr>
<tr>
<td>Obtain list of available users</td>
<td>GET /users</td>
</tr>
<tr>
<td>Obtain information of a particular user</td>
<td>GET /users/{userId}</td>
</tr>
<tr>
<td>Obtain social network of a user</td>
<td>GET /users/{userId}/network</td>
</tr>
<tr>
<td>Obtain the emotion of a user</td>
<td>GET /users/{userId}/emotion</td>
</tr>
<tr>
<td>Obtain the sentiment of a user</td>
<td>GET /user/{userId}/sentiment</td>
</tr>
<tr>
<td>Obtain the metrics of a user</td>
<td>GET /user/{userId}/metrics</td>
</tr>
<tr>
<td><strong>Tweets</strong></td>
<td></td>
</tr>
<tr>
<td>Obtain list of available tweets</td>
<td>GET /tweets</td>
</tr>
<tr>
<td>Obtain information of a particular tweet</td>
<td>GET /tweet/{tweetId}</td>
</tr>
<tr>
<td>Obtain the history of a particular tweet</td>
<td>GET /tweets/{tweetId}/history</td>
</tr>
<tr>
<td>Add a tweet to the database</td>
<td>POST /tweets</td>
</tr>
<tr>
<td>Delete a tweet from the database</td>
<td>DELETE /tweets/{tweetId}</td>
</tr>
<tr>
<td>Obtain the emotion of a tweet</td>
<td>GET /tweets/{tweetId}/emotion</td>
</tr>
<tr>
<td>Obtain the sentiment of a tweet</td>
<td>GET /tweets/{tweetId}/sentiment</td>
</tr>
<tr>
<td>Obtain the metrics of a tweet</td>
<td>GET /tweets/{tweetId}/metrics</td>
</tr>
</tbody>
</table>
### Tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
<th>Endpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obtain the list of tasks</td>
<td>GET</td>
<td>/tasks</td>
</tr>
<tr>
<td>Obtain the status of a particular task</td>
<td>GET</td>
<td>/task/{taskId}</td>
</tr>
</tbody>
</table>

### Communities

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
<th>Endpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obtain the list of calculated communities</td>
<td>GET</td>
<td>/communities</td>
</tr>
<tr>
<td>Obtain information of a particular community</td>
<td>GET</td>
<td>/communities/{communityId}</td>
</tr>
<tr>
<td>Obtain the emotion of a community</td>
<td>GET</td>
<td>/communities/{communityId}/emotion</td>
</tr>
<tr>
<td>Obtain the sentiment of a community</td>
<td>GET</td>
<td>/communities/{communityId}/sentiment</td>
</tr>
<tr>
<td>Obtain the users that belongs to a particular community</td>
<td>GET</td>
<td>/community/{communityId}/users</td>
</tr>
</tbody>
</table>

The figure below contains an example of a request to the module, using the web interface provided. In particular, it is a query for a particular tweet in the system. Additionally, by using the “fields” parameter, it instructs the server to only provide the metrics property of the tweet. An alternative to this query would be to use the /tweets/{tweetId}/metrics method.
Metrics and analytics

Internally, metrics are classified in two different types: direct and indirect metrics. Direct metrics are obtainable from the extracted data, such as the number of followers a user has. The Social Context Analysis module obtains direct metrics as soon as new social media content is stored in the database, and these metrics are updated when new information arrives. For instance, the Social Context module is configured to refetch general information about users periodically, so these metrics would be updated as well. Indirect metrics are obtained through data processing. The mechanisms to obtain these metrics are detailed in D4.9 Social Context Analysis for Emotion Recognition, initial version. These metrics are calculated periodically, as they have a high calculation cost and require accessing all the information in the database.
The user also needs to have real time information about the social context (e.g. the impact of a Donald Trump tweet talking about military industry). Because of this, it is important to separate Scaner performance in two phases:

1. Batch phase: This phase consists on the calculation of indirect metrics that requires accessing all the information in the database and a lot of calculations for the data stored. In this phase, Scaner creates relations in the topics for the tweets and users that belong to them.
2. Online phase: After processing batch phase, this phase is available. In this phase the Social Context Analysis module obtains direct metrics as soon as new social media content is stored in the database, and these metrics are updated when new information arrive. Social Context module is prepared to retrieve online information. Scaner processes new tweets posted or retweeted by the topic-related users and retweets of the topic-related users’ tweets.

Communities

There is another aspect in the social context of a user which Scaner now considers, the community to which a user belongs. In this document, we present how Scaner detects user communities and retrieves information about them. Moreover, due to Senpy, Scaner can now retrieve information of the communities’ sentiment analysis and emotion recognition.

Community Detection

The information extracted from a social network can be represented as a graph, where the vertices represent the users and the edges represent the relationships among them. This graph can be clustered into user groups, or communities, based on the topology information of the graph. Each community should include strongly interconnected vertices and few connections with the rest of graph vertices. The detection of these communities is handled using the Louvain community detection algorithm. This algorithm can automatically organize a set of users from a social network into similar communities to acquire knowledge about their common behaviours.

Scaner detects user communities using the algorithm mentioned before. To do so, uses information that relates to users, i.e. ‘follows’ in Twitter, as this is the information that defines the communities. An example of this is illustrated in the figure below. In this example it can be seen the relations among users that defines the community and the relations between an user and its community. This schema is made with the graph editor of OrientDB studio.
Sentiments and Emotions in Communities

Also, Scaner tags the sentiment and emotion of the communities, based on the sentiments and emotions of the users that compound the community.

Sentiment analysis is made by Senpy tool. This tool has an implemented algorithm which obtains the tweet’s polarity prevailing value. In order to get user polarity, Scaner obtains all users’ tweets polarity mean value. User sentiment is tagged according to this polarity. Community predominant sentiment is acquired likewise user sentiment (i.e. Community polarity is based on user’s belonging to each community polarity mean).

Emotion analysis is also made by Senpy, this case uses a different algorithm. This algorithm calculates tweet emotion prevailing centroids. On the other hand, each emotion has a predominant centroid. Each tweet is tagged with the nearest emotion. Users emotion is calculated based on each user created tweets, the emotion is determined using user’s centroids. Community predominant emotion is acquired likewise user emotion.
Scaner retrieves information of the communities detected: the sentiment and emotion of the community and the users that compounds the community. This information is available in the public Scaner API. In this way, the user can exploit this to find communities related to a topic that show negative emotions or a negative sentiment.

Example of usage
In order to use the Social Content Analysis service, users should follow three steps:
1. Load social-media into the module
2. Request social network entities, metrics and communities with the REST API
3. Post new tweets related to a known topic in order to find its relevance online

Step 1. Loading social-media
The following figure shows an example of the process of insertion of Twitter data into the service. In first place, the user inserts tweets via REST API, using the POST method. When the parameters provided in the request are invalid, the web server will issue a validation error:

_validation error: the id is missing_
In this case, the field “id” of the tweet is missing, which causes the validation error.

On the other hand, when the provided tweet is in the correct format, it will be stored in the database and the web interface will show a success message:

**Storing tweets in the database**
At this moment, the database contains just one tweet, basic information about its author, the emotion of the tweet, the empty objects of their metrics and their relationships. The relationships and metrics objects are empty because more tweets and users are necessary to calculate the different metrics implemented.

*Entities, metrics and relations in the database*
When the service gets more information, it calculates all the metrics and updates the different values. This allows keeping the information updated and also keeping the records of older metrics.

**Step 2. Retrieving data from Scanner**
Here is an example of the process of retrieving Twitter data from the service. In this scenario there are tweets already loaded in the social context analysis service. The service starts extracting the necessary information from Twitter and then calculates the metrics and communities.

Users can control the status of the extraction and processing tasks consulting a REST service that provides the status of the analysis task (FINISHED, RUNNING, ERROR).
Once the task has finished, users can use the rest of methods of the REST API.

*Querying with OrientDB Studio*
After the tweets are added, the service creates the different relationships between the elements of the database in order to create the different networks and context of tweets and users. Those are needed for the metrics and communities calculation. These relationships can be visualized with gephi directly from the database.

When the service has all the networks created, it starts the periodical process of calculating the metrics and the communities, storing the results in the database. Each iteration of the metrics has a timestamp, so the user can have a history of the metrics and track their evolution through time. Once the service has all the data, the user can access it through the API.

The user can get information about a specific tweet or user, their context, a list of tweets that belong to a certain topic, a list of tweets or users ordered by one of their attributes or metrics, a history of the metrics of certain tweet or user and so on. For example, the user can get the list of tweets with ordered by their “id”, but only showing their “id” and the text.

*Getting tweets list ordered by Id from Scanner API*
Other options could be to get the list of tweets that belong to a certain topic, filtering some of their fields, in order to get information about the status of the topic, such as activity, amount of tweets, tweet rate, etc.

The user can also retrieve metrics from the service in json format. An example of tweet metrics would be the following:

**Example of tweet metrics**

```json
{
    "tweet_metrics": {
```
Lastly, the following is an example of user metrics:

**Example of user metrics**

{  
  "user_metrics": { 
    "time_stamp": "Sun Feb 07 07:00:59 +0000 2016",  
    "popularity": 0.877778,  
    "followers": 41,  
    "following": 364,  
    "user_relevance": 0.42111,  
    "impact": 0.82,  
    "voice_r": 0.60035,  
    "replied_ratio": 0.0588235294,  
    "follow_ratio": 0.01587301587,  
    "reply_ratio": 0.03225806451,  
    "UI_score": 1,  
    "UI_unnormalized": 0.17830723,  
    "FR_score": 0.833333,  
    "TR_score": 0.677778  
  }  
}
Once the tweet and user data is stored in the database as detailed in the step 1, Scaner periodically calculates and tags sentiments and emotions of the tweets, users and communities.

After this, the user can obtain information about the sentiment and emotion of the users in the database, as it is shown in the pictures below.

**Getting User Sentiment from Scaner API**

This user expresses positive sentiments which concern a specific topic.
Getting User Emotion from Scaner API

Curl


Request URL

http://localhost:5000/api/v1/users/135192333/emotion

Response Body

```
{
    "metadata": {
        "parameters": {},
        "url": "http://localhost:5000/api/v1/users/135192333/emotion"
    },
    "result": {
        "emotion": {
            "emotion": "joy"
        },
        "id": 135192333
    }
}
```

Response Code

200

Response Headers

```
{
    "date": "Wed, 21 Dec 2016 10:49:52 GMT",
    "server": "Werkzeug/0.11.11 Python/3.4.5",
    "content-length": "198",
    "content-type": "application/json"
}
```

The emotion recognition shows how this user expresses joy in its posted tweets.

It’s also possible to obtain information about the sentiment and emotion of the communities. In this example it can be seen the negative sentiment of a small community detected by Scaner.
### Getting Community Sentiment from Scaner API

#### Curl
```
curl -X GET --header 'Accept: application/json' 'http://localhost:5000/api/v1/communities/1/sentiment'
```

#### Request URL
```
http://localhost:5000/api/v1/communities/1/sentiment
```

#### Response Body
```
{
    "communities": {
        "id": 1,
        "polarity": "negative",
        "polarityValue": -0.5,
        "user_count": 2
    }
)
```

#### Response Code
```
200
```

#### Response Headers
```
{
    "date": "Tue, 13 Dec 2016 12:12:35 GMT",
    "server": "Werkzeug/0.11.11 Python/3.4.5",
    "content-length": "324",
    "content-type": "application/json"
}
```
Getting Community Emotion from Scaner API

Curl

```
curl -X GET --header 'Accept: application/json' 'http://localhost:5000/api/v1/communities/36/emotion'
```

Request URL

```
http://localhost:5000/api/v1/communities/36/emotion
```

Response Body

```
{
  "metadata": {
    "parameters": {},
    "url": "http://localhost:5000/api/v1/communities/36/emotion"
  },
  "result": {
    "emotion": {
      "neutral"
    },
    "id": 36,
    "user_count": 8
  }
}
```

Response Code

`200`

Response Headers

```
{
  "date": "Thu, 22 Dec 2016 11:33:39 GMT",
  "server": "Werkzeug/0.11.11 Python/3.4.5",
  "content-length": "215",
  "content-type": "application/json"
}
```

Finally, in this picture, made with the graph editor of OrientDB Studio, it can be seen how the communities and the user sentiments and emotions calculated are distributed.
Community sentiments graph

The communities are in red and its sentiment near them. In this example, there is four communities, one positive, one negative and two neutrals. Also the polarity value of each user is highlighted.
**Community emotions graph**

The communities are in red and its emotion near them. In this example, there is a community whose users express fear.

**Step 3. Finding relevant tweets online**
After Scanner computes batch phase, explained in the step 1, the user instantly get the relevance of a new added tweet to the system that belongs to the same topic and are posted by an user in the database.
Retrieving tweet relevance online

```json
{
  "user": {
    "protected": false,
    "screen_name": "moc1_3",
    "lang": "ja",
    "created_at": "Thu Feb 24 04:14:58 +0000 2011",
    "id_str": "255653564",
    "followers_count": 191,
    "id": "255653564",
    "friends_count": 23
  },
  "id_str": "302813352173180203",
  "created_at": "Wed Dec 17 21:43:38 +0000 2014",
  "lang": "ja",
  "metadata": {
    "iso_language_code": "ja",
    "result_type": "recent"
  },
  "http://localhost:5000/api/v1/tweets"
}
```

**Request URL**

http://localhost:5000/api/v1/tweets

**Response Body**

```json
{
  "metadata": {
    "parameters": {},
    "url": "http://localhost:5000/api/v1/tweets"
  },
  "result": {
    "status": "Tweet already in DB",
    "tweet_relevance": {
      "relevance": 0.000279989919,
      "topic": "0lgdata"
    }
  }
}
```

**Response Code**

200
5. Semantic Knowledge Graph Infrastructure and API

The objective of the Knowledge Graph module, later in text referred to as KG module, is to provide insights into relations between recognised entities using semantic knowledge from DBpedia. The KG module uses entities that are recognised by the Entity Extraction and Linking module, and extracts relationships between the entities from DBpedia. Once the relations are extracted and filtered to keep the relevant ones only, they are stored in an ElasticSearch database, where they are visualized using the Kibi platform. The KG module is managed by a REST API, and needs an index in the ElasticSearch database that contains both the source text and the entities extracted. For now, it only supports English language. Using the module requires as little as providing credentials for the ElasticSearch instance and the name of the index.

Architecture

The architecture of the KG module is composed of five main parts: the Database, the DBpedia crawler, the Processing module, the Web server that exposes a REST interface, and the Kibi graph browser, as shown in the figure below:

- **Database.** A Repository where the information that is processed by other modules is stored, and where the information processed by the KG module will be stored. We are using ElasticSearch database here as Kibi is used for visualizing and exploring the graph.
- **DBpedia crawler.** This module is responsible for crawling information from DBpedia, that is related entities in the ElasticSearch database that were identified by the *Entity Extraction and Linking* module. We investigated multiple approaches to extract information from DBpedia (i.e., SPARQL endpoint, RDF dump). Considering that thousands of entities can be identified, querying a SPARQL endpoint for each of the entities would take far too long. For this reason, we decided to select the RDF files that contain information that is relevant to us only (i.e. infobox_properties_en.ttl, instance_types_en.ttl, persondata_en.ttl). We use then the Linux built-in application *grep*, which allows searching multiple patterns in a text file, to get all the information about entities out of the files. The resulting extracted set of triples is stored on the filesystem.

- **Processing module.** This module is responsible for filtering the extracted information and splitting it by types. The *Entity Extraction and Linking* module assigns one of 3 types to the recognised entity (Person, Organization, Location). When processing information from DBpedia, each type is processed separately so they can be stored in separate indexes. As the extracted information is not always "clean" (it can falsely be classified as a certain type of entity), the module applies customized filters for each type of entities to reduce number of wrongly classified entities. Filters were created so that entities containing properties that should not be shared with the other entities (i.e. duplicates) are removed. Below are listed the filtered properties for each type:
  - **Location:** birthPlace, location, foundation, type, founder, predecessor, party, country, based, channel, relations, branch, placeOfBirth, spouse, network, chancellor, parent, parents, primeMinister, artist, athletics, title, successor, state.
  - **Organization:** type, successor, distributor, birthPlace, vicePresident, largestCity, residence, spouse, predecessor, state, party, children, parents, education, almaMater, president, leaderName, largestCity, officialLanguages, leaderTitle, capital, governmentType, influences, influenced.
  - **Person:** leaderName, leaderTitle, type, largestCity, governmentType, officialLanguages, capital, network, firstAired, channel, headquarters, sisterNames, broadcastArea, country, director, creator.

Apart from writing the extracted information to the ElasticSearch database, the KG module automatically defines links between entities, adds the mapping of relations to ElasticSearch, and creates dashboards for each type of entities as well as a dashboard for the graph browser.

- **Web Server.** This module offers a web server that allows controlling and monitoring the KG module externally through a REST API.

- **Kibi.** The aforementioned application that performs "on the fly" analytics on the collected entities and processed data stored in ElasticSearch. The most important part for the KG is the Kibi graph browser, as it provides the capability to visualize connections between entities and explore existing connections based on relations in DBpedia. If the *Emotion Recognition from Text* module has been run and data has been stored in the same index as the entities extracted by the *Entity
Extraction and Linking module, then additionally to DBpedia relations, the graph will display connections based on the co-occurrence of emotions and entity altogether in the text.

API

The table below contains a summary of the methods accessible in the API. The full definition, the schemas used for validation, and some usage examples are available at the API demo endpoint: http://scaner.cluster.gsi.dit.upm.es/api/v1/ui/#/

<table>
<thead>
<tr>
<th>Description</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check default configuration</td>
<td>GET /configuration</td>
</tr>
<tr>
<td>Modify the configuration</td>
<td>POST /configuration</td>
</tr>
<tr>
<td>Reset back to default configuration</td>
<td>GET /reset</td>
</tr>
<tr>
<td>Get status of the module</td>
<td>GET /status</td>
</tr>
<tr>
<td>Create the Knowledge Graph</td>
<td>GET /start</td>
</tr>
</tbody>
</table>

Below is shown an example of default configuration. The configuration file contains two main parts: the credentials - containing information required to access ElasticSearch database - and variables - containing the name and type of the input index, as well as names of indexes that will be created and the filters that apply to them. This output is visible when REST’s call "GET /configuration" is made.

```json
{
  "credentials": {
    "elasticPort": 0,
    "elasticHost": "elasticIP",
    "elasticUsername": "elastic",
    "elasticPassword": "changeme"
  },
  "variables": {
```
Credentials are the minimal configuration required for the module to attempt to create the graph. They are submitted using REST’s call "POST /configuration". Below is an example of the request body.

```json
{
    "credentials": {
        "elasticHost": "localhost",
        "elasticPassword": "changeme",
        "elasticPort": 9220,
        "elasticUsername": "elastic"
    }
}
```

If everything is successful, the API will return the response below.

```json
{
    "status": "Updated"
}
```

Once the configuration is updated, the module can be started using the REST’s call "GET /start". If successful, the API will provide the response below. As can be seen, it points to "/status", where can be seen the status of the system.

```json
{
    "output": "check status on /status"
}
```
Checking the status is made using the “GET /status” REST’s call. It contains three main fields:
- **started**: time when the graph creation was started
- **status**: there can be three values for this field (running, finished, errors)
- **log**: contains the list of completed subtasks.

If the graph is successfully created, the API will provide the output below.

```json
{
  "log": [
    "deleting index location_unique {'acknowledged': True}'",
    "deleting index organization_unique {'acknowledged': True}'",
    "deleting index person_unique {'acknowledged': True}'",
    "Getting datasets",
    "got entities",
    "Running preset for location",
    "Running preset for location_unique",
    "Running preset for organization",
    "Running preset for organization_unique",
    "Running preset for person",
    "Running preset for person_unique",
    "created indexes successfully",
    "adding default indexes (6, [])",
    "Creating search location_unique (1, [])",
    "Creating search organization_unique (1, [])",
    "Creating search person_unique (1, [])",
    "Creating search trump_tweets (1, [])",
    "Creating search location (1, [])",
    "Creating search organization (1, [])",
    "Creating search person (1, [])",
    "Creating visualization (7, [])",
    "Creating dashboards location_unique (1, [])",
    "Creating dashboards organization_unique (1, [])",
    "Creating dashboards person_unique (1, [])",
    "Creating dashboards trump_tweets (1, [])",
    "Creating relations (1, [])",
    "Finished: 2017-04-08 01:18:31.665464"
  ],
  "started": "Sat, 08 Apr 2017 01:08:54 GMT",
  "status": "finished"
}
```
Kibi graph browser

Once the KG module has finished, the resulting graph can be explored with the Kibi instance. To better demonstrate the functionality of the Kibi graph browser, we will use as an example the "Trump demo". In this demo, we collected tweets containing a hashtag referring to Donald Trump, and then ran both the Entity Extraction and Linking and Emotion Recognition from Text modules.

In the beginning, the Graph dashboard is empty, and, as mentioned above, Kibi provides dynamic analytics "on the fly". As you can see there are in total 5 dashboards. All of them were created by the KG module. The prerequisites are having ElasticSearch running, and the Kibi instance and source index already added to Kibi (in our example, it is called "trump_tweets").

There are multiple options to start the graph discovery. We can start with a specific entry or one of four indexes:

- locations_unique: contains all recognised entities that are classified as locations
- organization_unique: contains all recognised entities that are classified as organizations
- person_unique: contains all recognised entities that are classified as person
- trump_tweets: contains the text from tweets, the recognised entities, and the emotions associated with each tweet

In the graph below you can see the result of selecting locations. In the beginning, the graph will show all locations as separate entities, but after a short while, connections between the entities become visible (see the second image).
At this stage, the visualisation is too chaotic to be useful. In order to fix this, we can select the "standard" layout. We can understand better how locations are interconnected (see below).
If we zoom in, we can clearly see connections between locations. The number in the green circle represents the number of connections the entity has to any other entity (not only the location entities).
We have then the choice either to add another type of entity to the visualization (e.g. organizations), or to select one of the visible entity to in turn discover its connections. In order to exemplify more fine grained results, we will select here the entity "Berkeley, California". Once we have selected the entity we can choose to "Expand" the selection.

We can thus see that the entity "Berkeley, California" is connected to 8 emotional tweets, one organization ("University_of_California,_Berkeley") and 3 other locations. One of the emotion nodes (negative-fear) has itself 3 connections, which means that this emotion was extracted from a tweet containing three entities. To learn more about what they are, we expand in turn this emotion node, and discover connections to one organization "United_States_Department_of_Justice", and two locations "San_Jose_California", "Berkeley, California".
In order to investigate how they are connected, we can select the emotion, create a filter and then check the "trump_tweets" dashboard to see more details displayed.

We can thus attest that the entities were detected correctly, including "DOJ" which stands for "United_States_Department_of_Justice", as well as the emotion of the tweet.

Now, we start again from an empty graph, but select persons this time. In the image below, we can see that the graph correctly shows that Trump family members are connected to each other.
If we want to find out what is the emotion distribution surrounding "Donald_Trump", we select the "Donald_Trump" node and create a filter. The display is then as follow.
In the “PercentageEmotions” graph, we can see emotions distribution about Trump evolved in time. In the “EmotionDistribution” graph we can see the overall emotion distribution in connection to Trump. Considering that the tweets are collected based on the mention of Trump, they cover a wide range of emotions and two of them are dominant, sadness and joy.

If we combine all 3 types of entities, we get a highly connected graph where the center is the United States. Some entities are not connected to this main graph, which is explained by the fact that they are not related to the United States (e.g., Netherland, Turkey, Sweden), but have been mentioned in correlation with Donald Trump.
Summary

We demonstrated with this use case that, with very little configuration or even background knowledge, it is possible to create and navigate through a Knowledge Graph, using the KG module and Kibi. In this example we only partially covered the capabilities of this technology. Depending on the use case and the user's knowledge of the domain, this technology can either provide quick insight into the data, or assist in more in-depth analysis, and help discovering connections and facts that might otherwise not have been detected.
6. Conclusions

In this document, we discussed the evolution of the Semantic Graph capabilities and socially derived data in MixedEmotions platform.

In the first part of the contribution we described how the original ideas that led to the first Kibi platform (described in the first deliverable) evolved into a more powerful and sophisticated system capable of providing big data streaming, semi-structured knowledge graph capabilities and APIs. The new stack includes the new Siren “Platform”, a sophisticated system that uses Relational Algebra technology to both be compatible with (virtualization) external SQL sources and to provide output SQL/Gremlin APIs. We then illustrated it in action on DW’s data processed by Expert System. In the third part of the contribution, we discussed the objectives of the Social Context Analysis module, that is to provide a platform of analysis for social media and social context using graph analytics. In the last part we illustrate the Knowledge Graph/Link Analysis tools and how they’re used to interact with the underlying knowledge graph, which has been created based on information extracted from external sources.

It is very worthy to notice that the great majority of this deliverable reflects real world used software infrastructure. The Kibi platform is currently in heavy use and commercialized across different sectors ranging from media, to life science and investigation. This is not dissimilar from the Social Context Analysis module. The fact that almost 100% of the described infrastructure is “commercially at work” is a remarkable fact and indication that the topic of this project has been of high general interest.