Distributed Semantic Indexing Infrastructure

Deliverable D4.1: Public

Keywords: Indexing, Solr, Siren, Distributed

Linked Open Data for environment protection in Smart Regions
Table of Contents

Table of Contents ....................................................................................................................... 2
List of Figures .............................................................................................................................. 6
1 Introduction and Overview ......................................................................................................... 9
2 The Siren indexing infrastructure ............................................................................................. 11
  2.1 SIREn Highlights .................................................................................................................. 11
  2.2 Basic Concepts ..................................................................................................................... 12
    2.2.1 Document ..................................................................................................................... 12
    2.2.2 Nested Data .................................................................................................................. 12
    2.2.3 Tree Model ................................................................................................................... 12
    2.2.4 Comparing Indexing Models .......................................................................................... 13
      2.2.4.1 Field-Based Indexing ............................................................................................ 13
    2.2.5 Blockjoin ....................................................................................................................... 14
    2.2.6 The SIREn data model .................................................................................................. 15
    2.2.7 Schemaless Indexing ..................................................................................................... 16
  2.3 SIREn plugin Architecture ................................................................................................. 17
  2.4 Solr Plugin .......................................................................................................................... 18
    2.4.1 Downloading the SIREn/Solr Distribution .................................................................. 20
    2.4.2 Indexing a document ..................................................................................................... 21
    2.4.3 Searching a document .................................................................................................. 21
    2.4.4 Running the demo ......................................................................................................... 23
    2.4.5 Elasticsearch plugin ....................................................................................................... 23
      2.4.5.1 Creating an index .................................................................................................... 25
      2.4.5.2 Indexing a document ............................................................................................ 26
      2.4.5.3 Searching a document .......................................................................................... 26
    2.4.6 Running the demos (Elasticsearch) .............................................................................. 29
  2.5 Performance Evaluation ......................................................................................................... 29
    2.5.1 Introducing SIREn vs Blockjoin .................................................................................... 29
  2.5.2 The Test Dataset: US Patent Grant XML Corpus ............................................................ 30
  2.5.3 Indexing with SIREn ........................................................................................................ 30
2.5.4 Indexing with Blockjoin ........................................................................................................ 30
2.5.5 Test Environment .................................................................................................................. 31
2.5.6 Index Size ............................................................................................................................. 31
2.5.7 Index Merging Time ................................................................................................................. 32
2.6 Query Performance .................................................................................................................... 32
   2.6.1 Query Patterns ...................................................................................................................... 32
   2.6.2 Experimental Design ............................................................................................................ 33
2.7 Query Throughput ..................................................................................................................... 34
   2.7.1 Query Memory Requirements ............................................................................................... 34
   2.7.2 Evaluating Nested queries .................................................................................................. 35
   2.7.3 Evaluating Filter queries ..................................................................................................... 35
   2.7.4 Evaluating Facet queries ..................................................................................................... 36
   2.7.5 Conclusions ......................................................................................................................... 37
3 Scaling the relationships: “relational faceted browser” .............................................................. 38
   3.1 Introduction: relational faceted browser .................................................................................. 38
   3.2 Additional background on the System .................................................................................... 41
      3.2.1 On Faceted Browsing ....................................................................................................... 41
      3.2.2 Relational Faceted Browsing (AKA Pivoted Faceted Browsing) ...................................... 41
      3.2.3 Typical, state of the art, implementation ........................................................................... 44
   3.3 Description of the System ........................................................................................................ 44
      3.3.1 Method for performing a facet synthesis on a domain of information ............................. 45
         3.3.1.1 Tree-Based Synthesis ............................................................................................... 45
         3.3.1.2 Reachability-Based Synthesis .................................................................................. 47
      3.3.2 Method for encoding facet synthesis into an inverted index ............................................ 48
         3.3.2.1 Multiple inverted index embodiment ....................................................................... 49
         3.3.2.2 Single inverted index embodiment ............................................................................ 50
      3.3.3 Method for composing index retrieval queries in response to user actions ................. 50
         3.3.3.1 Navigation State and Pivoting .................................................................................. 50
         3.3.3.2 Single inverted index embodiment ............................................................................ 53
         3.3.3.3 Related Books and Scientific Publications ............................................................... 54
4 Scaling data quantity and availability: cross data center replication ........................................ 54
   4.1 Introduction, large scale cross datacentre replication .......................................................... 55
7 Preview: applications in Pilots: KnowledgeBrowser for Tragsa and ARPA ............... 76
7.1 Tragsa Data ........................................................................................................... 76
7.2 ARPA data ............................................................................................................. 76
7.3 The goal .................................................................................................................... 76
List of Figures

Figure 1 Visual summary of the interrelationships between the components along with the corresponding chapters of this deliverable ................................................................. 11
Figure 2 Flat model representation of a JSON object ........................................................................ 14
Figure 3 Blockjoin representation of a JSON object ......................................................................... 15
Figure 4 SIREn representation of a JSON object ................................................................................ 15
Figure 5 An overview of the SIREn Plugin Architecture ................................................................. 18
Figure 6 Relational Faceted Browser, full data workflow .............................................................. 40
Figure 7 Example of dataset ........................................................................................................... 42
Figure 8 Example of relationship .................................................................................................. 42
Figure 9 Example of said systems are Freebase Parallax also ........................................................ 43
Figure 10 Example of data tree associated to a few data records of different kind ..................... 46
Figure 11 Example of reachable data records .................................................................................. 47
Figure 12 CDCR mechanism ........................................................................................................ 56
Figure 13 Workflow with SolrCloud and CDCR .......................................................................... 57
Figure 14 The complete data workflow in the "KnowledgeBroser" navigator .............................. 64
Figure 15 Example of a page in the Relational Entity Set Browser .............................................. 71
Figure 16 Example of a mockup where a “company detail” page is visualized .............................. 72
Figure 17 Example of a whiteboard mockup .................................................................................. 73
Figure 18 Example of a Relational Entity Set Browser mockup .................................................... 74
Figure 19 Preliminary version of the KnowledgeBrowser ........................................................... 75
The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Communities. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

Copyright © 2014, SmartOpenData Consortium.
Executive Summary

This deliverable reports on the task, led by SINDICE, for the providing of a distributed semantic indexing infrastructure ultimately leading to the software that will serve the use cases – specifically the ARPA and the TRAGSA Use cases. The platform provides ETL workflows for transformation of the environmental data produced by the partners according to the vocabulary defined in WP3. The transformed data is then indexed by Sindice distributed technology “Siren”, then underpinned by Solr or Elasticsearch and made available as a cloud solution for the exploitation plan. The platform unique point is the integration of large scale structured data handling (e.g. database) with powerful textual capabilities, allowing for the integration of all sort of non-structured or semistructured source (articles, reports, legislations, social content) alongside the datasets. Finally, the task goes end to end and demonstrates a first prototype of Software-as-a-Service platform for search based data analytics, which will be then used in the case studies of Tragsa and ARPA.
1 Introduction and Overview

For this project, the challenge is that to push state of the art big data “information retrieval” technique (typically associated with “search” problems) to satisfy a problem which has a large amount of “Structure” in the data.

While information retrieval systems have been enormously successful in delivering real time experience to users when collections of unstructured documents are involved, less usual has been the experience of seeing these techniques operating seamlessly on content that is both unstructured and structured (databases, records etc.) – in a way that truly leverages the relational information across the records.

Systems like Sort/Elasticsearch have acquired over the years enormous popularity as enterprise grade platforms that can deliver near real time results over large collections of documents.

The challenge for us has been: can we provide the same interactive experience when relational data aspects are involved? For example not only “when searching documents that contain the work “Quercus Coccifera” (the Mediterranean Oak) but instead” “find legislation documents that deal with areas in which the Quercus Coccifera is naturally found”. The name “Quercus Coccifera” might not at all show in those documents, but because of the “relationship between an entity mentioned in the document and the database knowledge then the document becomes relevant and must be found and explained for its relevancy by the system.

On top of the above “relational search” core feature, we also want features like:

- real time analytics – allowing us to have nice visual interactive information about “aggregates” of the data we’re looking at –
- powerful integration with aggregate Map tools and
- the ability to operate quickly on very diverse data – that is as much as possible in a “schemaless” fashion.

The structure of this deliverable follows the work that was done in the corresponding work package task. The chapter describe the fundamental concept and the corresponding implementing software infrastructure. Specifically:

Chapter 2 The Siren indexing infrastructure

To achieve the above, the core technical achievement has been allowing structured data to be efficiently indexed by information retrieval systems such as Solr and Elasticsearch. This is possible by the advances we made in the “Semantic Information Retrieval Engine” (or SIREn). These advances, along with explanation, examples and benchmarks and are illustrated in this chapter.

Chapter 3 Scaling the relationships: “relational faceted browser”
This chapter illustrates how the SIREn system is used in order to create the end user experience that allows real-time restrictions on relational data sets.

This methodology, using considerable level of precomputation – or denormalization, creates a number of indexes which the user interface coordinates seamlessly by rewriting queries as the users navigates across “Entities”.

**Chapter 4** Scaling data quantity and availability: cross data center replication

One of the goals of this project was to demonstrate that the infrastructure could scale in a multinational/cross location wide manner. For this it is paramount that the data can be distributed not only among more computers, in the same physical location but also in different physical locations e.g. across datacenters. For this, we have extended the SolrCloud capabilities to obtain “cross data centre replication”.

**Chapter 5** End to End ETL

In this chapter, we discuss the software components that have been implemented to produce the Relational Faceted Browser “effect”. These range from the ETL pipeline, to special component that have been created to perform the “data materialization”.

**Chapter 6** Preview: User interface and interaction

In this preview chapter we give a preliminary feel of the software which will be delivered as part of the pilots: the KnowledgeBrowser. This is a highly interactive search/ analytics user interface powered siren and the relational faceted browser methodology.

**Chapter 7** Preview: applications in Pilots: KnowledgeBrowser for Tragsa and ARPA

In this preview chapter we anticipate how we expect to be able to use the KnowledgeBrowser together with Spaziodati ARPA and TRASA for the resolution of their pilots.

For clarity, the following diagram illustrates the relationship between these chapters.
2 The Siren indexing infrastructure

SIREn is a highly scalable open-source full-text search engine especially suited for nested and schemaless data. It can be used standalone, similar to Lucene, and it comes in the form of Elasticsearch and Solr plugins. SIREn has its own data model which is better suited for nested documents such as JSON and XML. The next sections give an overview of SIREn’s architecture, its model and the model’s implications on querying.

2.1 SIREn Highlights

SIREn is a plugin for Solr and Elasticsearch for searching nested and schemaless data at scale. Based on state-of-the-art Information Retrieval techniques, SIREn has been designed from the ground up to scale to billions of documents and to cope with nested and schemaless data while retaining low latency queries and low memory footprint.

SIREn offers the following features:

- **Nested Data**: search and analyse nested data at scale
- **Schemaless**: search and analyse data without significant modelling effort upfront
- **High Scalability**: scale-out through partitioning and replication
- **Low Latency**: search your data in sub-second response time
• **Low Memory Footprint**: complex nested data models do not impact memory consumption.

• **Advanced Search Operators**: augment standard search operators, e.g., term, phrase, Boolean, etc., with nested data search operators including parent-child and ancestor-descendant relationships and nested object proximity.

• **Plug & Play**: integrated with Solr and Elasticsearch’s REST API.

SIREn is based on a generic data model that is compatible with a broad variety of nested data models: JSON, XML, Avro, etc. The data model has been carefully designed for scalability and high performance.

SIREn is schema-agnostic. It does not require any schema definition to index and search data. The schema definition can even change across records. You do not have to invest significant modelling effort to clean, transform and flatten the data prior to indexing.

SIREn offers a Plug & Play integration with major open-source search systems such as Apache Solr and Elasticsearch. You can install the plugin in existing Solr or Elasticsearch deployment and start searching nested data alongside Solr and Elasticsearch features.

### 2.2 Basic Concepts

SIREn has a slightly different concept of Document than Solr and Elasticsearch. The difference allows for richer indexing and querying of nested data. This section introduces few concepts that are core to SIREn and that are commonly used throughout this manual.

#### 2.2.1 Document

As for Solr and Elasticsearch, a document is the basic unit of information that can be indexed. In SIREn, a document is expected to be a serialization of a nested data object expressed in JSON.

#### 2.2.2 Nested Data

Nested data follows a tree-like structure, in which you arbitrarily can nest objects within objects. The nested data model is quite popular with data format such as JSON, XML, Avro, Protocol Buffer. These formats allow deeply nested and large structure, and do not enforce strict schemas. Each document can have a separate schema.

#### 2.2.3 Tree Model

SIREn adopts an ordered tree data model to encode nested data. Nested data such as JSON is mapped to a tree model prior to indexing. A tree is composed of nodes, where each node can have one or more child nodes. The child nodes of a given node form an ordered sequence, i.e., there is an ordering relationship between them. A root node represents the
beginning of the tree. A node of the tree can contain arbitrary data such as text or numeric values. This generic tree data model enables SIREn to be compatible with a wide numbers of nested data models such as JSON and XML, while offering advanced search operations on the tree content and structure.

2.2.4 Comparing Indexing Models

Search systems such as Solr and Elasticsearch were originally designed for searching flat data, e.g., documents with a set of attribute-value pairs, and not for searching nested data. A common fallback strategy is to rely on flattening. However, flattening the data is error-prone and time-consuming. For this reason, Solr and Elasticsearch have recently introduced a new feature, called Blockjoin in Solr and Nested Type in Elasticsearch, to enable searching nested data. However, this feature is rooted in the traditional flat model design, which results in limitations, in performance and flexibility. SIREn on the contrary has been designed from the ground up for searching schemaless nested data and does not suffer from these limitations. In this section we will give a high level overview on how SIREn compares to other indexing models.

2.2.4.1 Field-Based Indexing

The field-based indexing model is the traditional model implemented in Lucene. To index nested data using this model, it is necessary to first flatten the data prior to indexing. The flattening consists of creating a document field for each path of attributes. For example, the json object below will produce the document as shown in Figure 2.

```
{  
  "name": "Radar Networks",
  "location": {
    "city": "San Francisco",
    "state": "California"
  },
  "fundings": [
    {  
      "round": "a",
      "investors": "NY Angels"
    },
    {  
      "round": "b",
      "investors": "Sequoia"
    }
  ]
}
```
This approach works fine with nested objects with a one-to-one relationship with the parent document, for example with the location nested object. The limitation of this approach occurs with one-to-many relationships between the parent document and the nested objects. For example, if you try to search for documents which contains a nested object matching `fundings.round: a AND fundings.investors: Sequoia`, the previous document will be returned which is a false-positive match. Due to the flat representation, the system is not able to differentiate attribute matches between nested objects and lose precision in the matching.

### 2.2.5 Blockjoin

The Blockjoin approach does not rely on flattening nested data. Instead, it internally creates one document per nested objects and index them within the same "index block". For example, the previous JSON object will be represented by four documents as shown in Figure 3 Blockjoin representation of a JSON object. Documents within the same index block are related to each other through a parent-child relationship. An index block enables to join documents with their parent document in a very efficient manner. This approach solves the precision problem encountered with one to many relationships. However, a side-effect of Blockjoin is that it artificially increases the number of documents in the index. The number of documents being generated by Blockjoin is proportional to the number of nested objects. This has a direct consequence on the memory consumption and scalability of the system since the memory requirement of various caching strategies is correlated with the number of documents in the index.

As we can see in Figure 3 Blockjoin representation of a JSON object, Blockjoin needs to tag each document with an identifier, usually the attribute path that leads to the nested object. This enables Blockjoin to differentiate between nested objects from different paths and to avoid a loss of precision in the matching. This tag will be used at query time to filter out nested objects, and it introduces a loss of performance during query processing. Another
side-effect is that it forces the user to know the exact path to the nested objects he wants to query, limiting the ability to query for nested objects without knowing their path.

There are crucial differences in performance and scalability between the Blockjoin and the Siren indexing method, which will be illustrated in the following sections.

![Figure 3 Blockjoin representation of a JSON object](image)

### 2.2.6 The SIREn data model

SIREn adopts a generic tree model to represent nested data and semi-structured information. Each document in SIREn is represented as a tree, as pictured in Figure 4 SIREn representation of a JSON object. The root node of the tree represents the top-level object. The child nodes of the root represent the attributes associated to the top-level object. The child nodes of an attribute node represent the values associated to this attribute. A value can either be a primitive value (string, numeric or boolean), a nested object or an array of values.

![Figure 4 SIREn representation of a JSON object](image)

Compared to Blockjoin, SIREn does not artificially increase the number of documents in the index, since the full tree is encoded within one single document. As a result, the memory
consumption and scalability of the system is not affected by the number of nested objects in the data. To achieve this, SIREn uses its own index format with very efficient compression and encoding schemes to store and index the tree structure within a single document.

SIREn indexing model enables more advanced search capabilities than Blockjoin. Given that the encoded tree model keeps the ordering of the nodes, it becomes possible to design a search query that will take into account the proximity of the nodes, e.g., searching for nested objects that occurs next to each other or at two positions apart. It is also possible to search for nested objects without knowing their full path from the root node, i.e., a wildcard path search.

There are also other advantages such as the support for precise search of multi-valued attributes, or the ability to analyze and search attribute names as any other textual content.

### 2.2.7 Schemaless Indexing

SIREn’s schemaless indexing approach allows you to index a wide variety of JSON data without the need for any upfront configuration.

Let’s take a look at a truncated sample record of the NCPR dataset that is used in Siren the Getting Started guides.

```json
{
    "ChargeDeviceId": "4d61996cb5a40f214058d84fc7aba126",
    "ChargeDeviceName": "4c Design Limited, Glasgow (1)",
    "ChargeDeviceRef": "SCOT1",
    "Accessible24Hours": false,
    ...

    "DeviceController": {
        "ContactName": null,
        "OrganisationName": "Part of the Charge Your Car network",
        "TelephoneNo": "0191 26 50 500",
        "Website": "http://chargeyourcar.org.uk/ev-driver/"
    },

    "ChargeDeviceLocation": {
        "LocationShortDescription": "4c Design Limited, Glasgow",
        "Latitude": 55.875192,
        "Longitude": -4.249105,

        "Address": {
            "BuildingNumber": "100",
            "Street": "Borron Street",
            "PostCode": "G4 9XG",
            "PostTown": "Glasgow",
            "Country": "gb"
        }
    }
}
```
You do not have to write a single line of configuration to index this complex object. Schemaless indexing allows SIREn to automatically infer and index:

- A diverse set of native JSON datatypes - string, int, float and boolean
- Simple flat attributes like `ChargeDeviceRef`
- Nested objects like `DeviceController`
- Multiple levels of nesting like `ChargeDeviceLocation : Address`
- Array of objects like the `Connector` attribute

What’s more, this schemaless approach also applies to documents that are further added with a different schema. SIREn will transparently handle any changes when you add documents with:

- Additional fields
- Additional nested objects
- Fields with varying types
- Objects with varying nesting structure
- An array of heterogeneous objects

### 2.3 SIREn plugin Architecture

SIREn extends the Lucene framework on multiple levels as shown in the figure below.

SIREn provides a JSON analyzer to interpret incoming JSON data and maps it to our generic tree data model. This tree data representation of the JSON object is sent to the Lucene’s Codec API. The tree indexing model is implemented on top of the Lucene’s Codec API by
adding our own posting format that is optimized for encoding tree-structured data. This component is in charge of encoding and decoding the tree data model to/from the file system.

When a query comes in, the query is interpreted by our JSON-based query parser. The query is converted into a Node Query object, which is an extension of the Lucene’s Query API. Given that the underlying data model in SIREn is quite different than the one found in Lucene, we had to implement a new query processing framework that is compatible with our data model. Finally the node query is executed against our Codec API to compute the result set.

On a top-level, SIREn provides two plugins, one for Solr and one for Elasticsearch, that integrates these functionality into their respective API.

![Diagram of SIREn Plugin Architecture](image)

**Figure 5 An overview of the SIREn Plugin Architecture**

### 2.4 Solr Plugin

Siren is distributed in Solr and Elasticsearch plugins. These sections (Solr and Elasticsearch) are not comprehensive but exclusively reported to give an idea of how SIREn can be installed and configured. For the full documentation please refer to the [http://siren.solutions](http://siren.solutions) website.
In these examples, will see how you can index a collection of richly structured JSON data and query them using SIREn within the Solr environment.

We will use the National Charge Point Registry dataset in JSON format available here. It contains charge points for electric vehicles with information like geographical location, address connectors types, opening hours and so on. There are over a 1000 charge points in the dataset. We have modified the dataset to ensure that the values are correctly typed with native JSON types. You can see a truncated sample record below.

```
{
  "ChargeDeviceId": "885b2c7a6deb4fea10f319c4ce993e02",
  "ChargeDeviceName": "All Eco Centre Car Park",
  "ChargeDeviceRef": "CM765",
  "Accessible24Hours": false,
  ...

  "DeviceController": {
    "ContactName": null,
    "OrganisationName": "Source East",
    "TelephoneNo": "08455198676",
    "Website": "www.sourceeast.net"
  },

  "ChargeDeviceLocation": {
    "Latitude": 52.5744,
    "Longitude": -0.2396,
    "Address": {
      "Street": "City Road",
      "PostCode": "PE1 1SA",
      "PostTown": "Peterborough",
      "Country": "gb"
    },
  },

  "Connector": [
    {
      "ConnectorId": "CM765a",
      "ChargeMode": 1,
      "ChargeMethod": "Single Phase AC",
      "ChargePointStatus": "In service",
      "ConnectorType": "Domestic plug/socket type G (BS 1363)",
      "RatedOutputCurrent": 13
    },
    {
      "ConnectorId": "CM765b",
      "ChargeMode": "3",
      "ChargeMethod": "Single Phase AC",
      ...
    }
  ]
}``
2.4.1 Downloading the SIREn/Solr Distribution

The SIREn/Solr distribution can be downloaded from this link. Let’s assume to extract it in a directory which we will call ${SOLR_HOME} from now on. The directory should contain:

- **1** SIREn jars required by the SIREn/Solr plugin
- **2** the SIREn/Solr plugin
- **3** the SIREn documentation
- **4** a Solr instance with the SIREn plugin pre-installed, including one demo

You can start the Solr instance with the following commands:

```
$ cd $SOLR_HOME/example
$ java -jar start.jar
```

In the output, you should see a line like the following which indicates that the SIREn plugin is installed and running:

```
1814 [main] INFO org.apache.solr.core.SolrResourceLoader— Adding
'SOLR_HOME/example/solr/lib/siren-core-${version}.jar' to classloader
1815 [main] INFO org.apache.solr.core.SolrResourceLoader— Adding
'SOLR_HOME/example/solr/lib/siren-qparser-${version}.jar' to classloader
1816 [main] INFO org.apache.solr.core.SolrResourceLoader— Adding
'SOLR_HOME/example/solr/lib/siren-solr-${version}.jar' to classloader
```
2.4.2 Indexing a document

To index the NCPR dataset, just open a new terminal window, navigate to the example folder within the SIREn installation and use the post.jar to load the data:

```bash
```

If all is fine, you should see the count of documents loaded into SIREn (1078). You can also go to the Solr Admin UI and see the count of docs in stats page for the default collection ("collection1").

2.4.3 Searching a document

SIREn uses a JSON based query syntax. You can find more about the query syntax of SIREn in the chapter [Querying Data](#). We will now show you some query examples you can execute on the NCPR index. You can use either the Solr Admin UI or the command line to query SIREn. You could copy/paste and execute the queries from the Solr Admin UI.

The first search query is a **node query** that matches all documents with an attribute “ChargeDeviceName” associated to a value matching the wildcard search query “SCOT*”.

```json
{
    "node": {
        "attribute": "ChargeDeviceName",
        "query": "SCOT*"
    }
}
```

The next query is a **twig query** that demonstrates how to search nested objects.

```json
{
    "twig": {
        "root": "DeviceOwner",
        "child": [{
            "node": {
                "attribute": "Website",
                "query": "uri(www.sourcelondon.net)"
            }
        }]
    }
}
```

The next query demonstrates how to search multiple level of nested objects.

```json
{
    "twig": {
        "root": "ChargeDeviceLocation",
        "child": [{
            "node": {
                "attribute": "LocationName",
                "query": "London"
            }
        }]
    }
}
```
The next query demonstrates how to search among an array of nested objects.

```
{  
  "twig": {  
    "root": "Connector",  
    "child": [{  
      "node": {  
        "attribute": "RatedOutputCurrent",  
        "query": "xsd:long(13)"  
      },  
      "attribute": "RatedOutputVoltage",  
      "query": "xsd:long(230)"  
    }]  
  }
}
```

The next query demonstrates how to perform a numerical range search.

```
{  
  "twig": {  
    "root": "ChargeDeviceLocation",  
    "child": [{  
      "occur": "MUST",  
      "node": {  
        "attribute": "Latitude",  
        "query": "xsd:float(50.0, 51.0)"  
      }  
    }]  
  }
}
```
2.4.4 Running the demo

The SIREn/Solr distribution contains a demo based on the NCPR (National Charge Point Registry) dataset. To execute the demo, go to the $SOLR_HOME/example directory:

$ cd $SOLR_HOME/example

To index the NCPR dataset, execute the following command:

$ bin/load-ncpr.sh

You can then query the index using the following command:

$ bin/query-ncpr.sh

The script executes a list of queries. It will display each of the query and the response header returned by Solr.

2.4.5 Elasticsearch plugin

In this section we will install the SIREn plugin within the Elasticsearch environment and execute some initial queries.

We will use the National Charge Point Registry dataset in JSON format available [here](#). It contains charge points for electric vehicles with information like geographical location, address connectors types, opening hours and so on. There are over a 1000 charge points in the dataset. We have modified the dataset to ensure that the values are correctly typed with native JSON types. You can see a truncated sample record below.

```json
{
    "ChargeDeviceId": "885b2c7a6deb4fea10f319c4ce993e02",
    "ChargeDeviceName": "All Eco Centre Car Park",
    "ChargeDeviceRef": "CM765",
    "Accessible24Hours": false,
    ...
    "DeviceController": {
        "query": "xsd:double([-3.2 TO -2.8])"
    }
}
```
“ContactName”: null,
“OrganisationName”: “Source East”,
“TelephoneNo”: “08455198676”,
“Website”: “www.sourceeast.net”
},

“ChargeDeviceLocation”: {
  “Latitude”: 52.5744,
  “Longitude”: -0.2396,

  “Address”: {
    “Street”: “City Road”,
    “PostCode”: “PE1 1SA”,
    “PostTown”: “Peterborough”,
    “Country”: “gb”
  }
},

“Connector”: [
  {
    “ConnectorId”: “CM765a”,
    “ChargeMode”: 1,
    “ChargeMethod”: “Single Phase AC”,
    “ChargePointStatus”: “In service”,
    “ConnectorType”: “Domestic plug/socket type G (BS 1363)”,
    “RatedOutputCurrent”: 13
  },
  {
    “ConnectorId”: “CM765b”,
    “ChargeMode”: “3”,
    “ChargeMethod”: “Single Phase AC”,
    ...
  }
]

Downloading the SIREn/Elasticsearch Distribution can be done at this address. Extracting it to a directory which we will call $\{ES\_HOME\}$ from now on, it should contain:

```
├── dist
│   ├── siren-core-${version}.jar ❶
│   ├── siren-qparser-${version}.jar
│   └── siren-elasticsearch-${version}.jar ❷
├── docs
└── example ❹
```

1. SIREn jars: bundle them with your Java application to use SIREn in embedded mode
2. the SIREn/Elasticsearch plugin
3. the SIREn documentation
4. an Elasticsearch instance with the SIREn plugin pre-installed, including three demos

You can start the Elasticsearch instance with the following commands:

$ cd $ES_HOME/example
$ bin/elasticsearch

In the output, you should see a line like the following which indicates that the SIREn plugin is installed and running:

[2014-07-02 14:40:04,008][INFO ] [plugins ] [Basilisk] loaded [siren-plugin], sites []

Now you can create an index, set an initial mapping, index and query documents via the Elasticsearch REST API as usual. You might also want to skip ahead and just run the provided demo scripts.

2.4.5.1 Creating an index

First, create an index called “ncpr” with the following command:

$ curl -XPOST "localhost:9200/ncpr/"

The following command registers a mapping that will setup a SIREn field for the the “chargepoint” type under the “ncpr” index:

$ curl -XPUT "http://localhost:9200/ncpr/chargepoint/_mapping" -d ' 

```json
{
  "chargepoint" : {
    "properties" : {
      "_siren_source" : {
        "analyzer" : "concise",
        "postings_format" : "Siren10AFor",
        "store" : "no",
        "type" : "string"
      },
      "_siren" : {}
    }
  }
}
```

From now on, any document indexed into the “ncpr” index under the “chargepoint” type will be indexed by SIREn too.
2.4.5.2 Indexing a document

The following command inserts a JSON document into the “ncpr” index under the “chargepoint” type with an identifier equal to 1.

```
$ curl -XPUT "http://localhost:9200/ncpr/chargepoint/1" -d ' 
{
  "ChargeDeviceName": "4c Design Limited, Glasgow (1)",
  "Accessible24Hours": false
}
```

Next, you can execute a bash script to load the full NCPR dataset into the index:

```
$ bin/ncpr-index.sh
```

If all is fine, you should see the count of documents loaded into SIREn (1078).

2.4.5.3 Searching a document

SIREn uses a JSON based query syntax. You can find more about the query syntax of SIREn in the chapter Querying Data. We will now show you some query examples you can execute on the NCPR index. The following commands execute various search queries and should get back the previously indexed documents.

The first search query is a node query that matches all documents with an attribute “ChargeDeviceName” associated to a value matching the wildcard search query “SCOT*”.

```
$ curl -XPOST "http://localhost:9200/ncpr/_search?pretty" -d ' 
{
  "query": {
    "tree": {
      "node": {
        "attribute": "ChargeDeviceName",
        "query": "SCOT*"
      }
    }
  }
}
```

The next query is a twig query that demonstrates how to search nested objects.

```
$ curl -XPOST "http://localhost:9200/ncpr/_search?pretty" -d ' 
{
  "query": {
    "tree": {
      "twig": {
        "root": "DeviceOwner",
        "attribute": "ChargeDeviceName",
        "query": "SCOT*"
      }
    }
  }
}
```
The next query demonstrates how to search multiple level of nested objects.

$ curl -XPOST "http://localhost:9200/ncpr/_search?pretty" -d ' 

```
{
  "query": {
    "tree": {
      "twig": {
        "root": "ChargeDeviceLocation",
        "child": [{
          "twig": {
            "root": "Address",
            "child": [{
              "node": {
                "attribute": "PostTown",
                "query": "Norwich"
              }
            }, {
              "node": {
                "attribute": "Country",
                "query": "gb"
              }
            }
          }
        }
      }
    }
  }
}
```

The next query demonstrates how to search among an array of nested objects.

$ curl -XPOST "http://localhost:9200/ncpr/_search? pretty" -d ' 

```
{
  "query": {
    ...  
  }
}
```


The next query demonstrates how to perform a numerical range search.


{
  "query": {
    "tree": {
      "twig": {
        "root": "ChargeDeviceLocation",
        "child": [{
          "occur": "MUST",
          "node": {
            "attribute": "Latitude",
            "query": "xsd:double([55.6 TO 56.0])"
          }
        },
        {
          "occur": "MUST",
          "node": {
            "attribute": "Longitude",
            "query": "xsd:double([-3.2 TO -2.8])"
          }
        }
      }
    }
  }
}
2.4.6 Running the demos (Elasticsearch)

The SIREn/Elasticsearch distribution contains three demos on three different datasets: NCPR (National Charge Point Registry), BNB (British National Bibliography) and a small movie dataset. To execute the demos, go to the $ES_HOME/example directory:

```
$ cd $ES_HOME/example
```

To index the small movie dataset, execute the following command:

```
$ bin/movies-index.sh
```

The script creates an index called “movies”, sets a mapping as shown earlier for the movie type, so that all documents sent to that index are indexed by SIREn too and then indexes a couple of movie documents which reside in the datasets/movies/docs/ directory.

You can then query the index using the following command:

```
$ bin/movies-query.sh
```

The script takes you through a couple of queries and always hints at what results are expected.

There are two more demos that index and query a slightly larger number of documents — BNB:

```
$ bin/bnb-index.sh
$ bin/bnb-query.sh
```

And NCPR (National Charge Point Registry):

```
$ bin/ncpr-index.sh
$ bin/ncpr-query.sh
```

2.5 Performance Evaluation

There are a lot of use cases that require indexing of nested documents. One only needs to scan the Lucene/Solr user forums to discover examples of this need.

In this section we’ll show how SIREn is by far the only approach that can scale to real world scenarios while delivering the advanced search capabilities on structured data allow.

2.5.1 Introducing SIREn vs Blockjoin

Both SIREn and Lucene’s Blockjoin feature offer a solution for working with nested documents. However, SIREn is based on a conceptual model that is fundamentally different from the Blockjoin approach. As such, for anyone deciding between these two solutions, it is important to understand their relative scalability and performance metrics.
In this document, we describe the results of some benchmark tests we conducted on both these approaches. We show how SIREn v1.2 offers enormously improved scalability and enhanced performance.

While the focus of this document is Lucene/Solr, the results are identically applicable to ElasticSearch which, under the hood, uses Lucene’s Blockjoin to support nested documents.

### 2.5.2 The Test Dataset: US Patent Grant XML Corpus

One of the core strengths of Lucene/SIREn is their ability to query over structured and unstructured content. As such, we wanted to pick a dataset that provides a combination of both.

The [US Patent and Trademark Office (USPTO) dataset](https://www.uspto.gov) is a good example of complex XML data and as such it was used for the test. Each patent document contains structured data about inventors, classifications, etc., as well as textual content such as the abstract, the claims being made, and even down to the single paragraph of text. All of this data is described in a XML document with multiple levels of nesting.

#### Preparing the Data for Indexing

The XML data is transformed into the equivalent JSON for this test. The transformation is based on the [json-lib library](https://code.google.com/p/json-lib) with some extensions to remove bold, italic and other tags in text values. You can find an example of such a JSON document in the Appendix.

#### Statistics about the USPTO dataset

<table>
<thead>
<tr>
<th>Number of Documents</th>
<th>Uncompressed JSON</th>
<th>Compressed JSON</th>
</tr>
</thead>
<tbody>
<tr>
<td>44369</td>
<td>5 GB</td>
<td>715 MB</td>
</tr>
</tbody>
</table>

#### 2.5.3 Indexing with SIREn

SIREn natively supports JSON structured data. It can index the JSON documents without any additional data preparation. Each JSON document is mapped to a Lucene’s document with the following fields:

- **id** : indexed and stored field to identify the document
- **json** : indexed, non-stored field configured to use SIREn’s hybrid indexing model

#### 2.5.4 Indexing with Blockjoin

The quickest way to test the Blockjoin approach would be to use Elasticsearch, as it too does not require transformation of the data – the only gotcha being that one has to manually [modify the schema](https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules.html) mapping to ensure that the engine does not “flatten” the data.
nested documents and uses Blockjoin. In contrast, SIREn is truly “schemaless” and does not require any configuration or modification.

For these tests however we will use Lucene directly for greater control and precision in the results.

2.5.5 Test Environment

The configuration of the machine used for the running the benchmark test is described below:

- Processor: Intel(R) Core(TM) i5 CPU M 580 @ 2.67GHz
- Memory: 8GB DDR3 Synchronous 1334 MHz
- Disk: Intel SSDSA1M160 – Ext4
- OS: Ubuntu 14.04 64-Bit
- Java: 1.7.0_45 64-Bit

SIREn v1.2 was used for this test. Lucene/Solr v4.7 was used for testing the Blockjoin’s functionality.

2.5.6 Index Size

For the creation of the index, we have disabled auto flush and the compound format, and we have increased the ram buffer size to 256 MB. We finally perform a merge to create one single segment.

For the test dataset, the Blockjoin’s approach leads to indexes that are almost twice as large as that for SIREn. This is due to the artificial increase in the number of documents for Blockjoin.

There are an average of 1833 nested objects per patent document in the test dataset. Given that the dataset covers just the first two month of the year, we can expect 6 times more documents for indexing the dataset for a full year. This would result in 486 million documents for Blockjoin.

**Comparison of index size in gigabytes between SIREn and Blockjoin**

<table>
<thead>
<tr>
<th>Number of Documents in Index</th>
<th>Index Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIREn</td>
<td>44369</td>
</tr>
<tr>
<td>Blockjoin</td>
<td>81350921</td>
</tr>
</tbody>
</table>
2.5.7 Index Merging Time

During the creation of the index, we perform a merge to create one single segment, and we record the time to perform the merge. Reported time has been averaged over 5 runs.

**Comparison of index merge time in milliseconds between SIREn and Blockjoin**

<table>
<thead>
<tr>
<th></th>
<th>Merge Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIREn</td>
<td>91312</td>
</tr>
<tr>
<td>Blockjoin</td>
<td>339002</td>
</tr>
</tbody>
</table>

We can see that SIREn is 3 to 4 times faster than the Blockjoin’s approach.

2.6 Query Performance

2.6.1 Query Patterns

The query patterns cover common operators, i.e., conjunction, disjunction and phrase, on the text but also on the structure of the document. The query patterns are restricted to match in a paragraph. A paragraph is a nested object of a description section, the description section being itself a nested object of the parent document. This ‘path’ of nested objects is quite common in the USPTO dataset (there are more than 3.7 millions paragraph objects). Querying a paragraph requires two joins: the first one being the join with the description objects, and the second one being the join between the description objects and the parent documents.

- **lookup** lookup of a high frequency term in a paragraph
- **conjunction** conjunction (AND) of two high frequency terms in a paragraph
- **disjunction** disjunction (OR) of two high frequency terms in a paragraph
- **phrase** phrase of two terms in a paragraph
- **conjunction-paragraph** conjunction of two paragraphs within the same description object
- **disjunction-paragraph** disjunction of two paragraphs within the same description object

The paths for the lookup, conjunction, disjunction and phrase queries are shown below.

```json
{
  "description": {
    ...
  }
```
The paths for conjunction and disjunction paragraph queries are shown below.

```
{
    "description": {
        "p": [
            {
                "#text": "TERM1",
            }
        AND / OR
            {
                "#text": "TERM2",
            }
        ]
    }
}
```

### 2.6.2 Experimental Design

The queries are generated based on the frequency of the words composing them. The word frequency determines how many entities match a given keyword. The words are grouped into three frequency ranges: high, medium, and low. We first order the words by their descending frequency, and then take the first \( k \) words whose cumulative frequency is 40% of all word occurrences as high range. The medium range accounts for the next 30%, and the low range is composed of all the remaining words. For the phrase queries, we follow a similar technique. We first extract all the 2-gram from the data collection. We then compute their frequency and sort them by descending frequency. We finally create the three ranges as explained above.

For each type of query, we (1) flush the OS cache; (2) initialize a new JVM; (3) generate a set of 50 random queries; (4) warm the JVM by executing a certain number of times the set of queries, and (5) perform 60 measurements. Each measurement is made by performing \( n \) times the query execution of the 50 random queries, with \( n \) chosen so that the runtime is long enough (10 seconds) to minimize the time precision error of the OS and machine. The measurement time is the time used by the current thread to process the 50 random queries. We report the query rate (query per second) based on the mean of the measurements.

With respect to Blockjoin’s queries, the parent document filters are created prior to the measurements, and are reused across measurements.
SIREn and Blockjoin are configured to compute the score of a document based on the average score of its children. This means that we iterate over all the child node matches.

## 2.7 Query Throughput

The results for the query performance are shown below. The query rates for SIREn and Blockjoin is shown along with the percentage boost offered by the former relative to the later.

### Query rates (queries per second) per query type for SIREn and Blockjoin

<table>
<thead>
<tr>
<th>Query</th>
<th>Blockjoin Query Rate</th>
<th>SIREn Query Rate</th>
<th>Rate Ratio %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Lookup</td>
<td>52.89</td>
<td>140.12</td>
<td>265%</td>
</tr>
<tr>
<td>2. Conjunction</td>
<td>307.66</td>
<td>192.42</td>
<td>63%</td>
</tr>
<tr>
<td>3. Disjunction</td>
<td>28.17</td>
<td>74.04</td>
<td>263%</td>
</tr>
<tr>
<td>4. Phrase</td>
<td>170.29</td>
<td>47.68</td>
<td>28%</td>
</tr>
<tr>
<td>5. Conjunction Paragraph</td>
<td>48.30</td>
<td>163.92</td>
<td>339%</td>
</tr>
<tr>
<td>6. Disjunction Paragraph</td>
<td>26.39</td>
<td>74.41</td>
<td>282%</td>
</tr>
</tbody>
</table>

For query 1,3,5,6 SIREn performs an average of 3 times faster than Blockjoin. This is due to Blockjoin suffering when the child query is not highly selective as intersecting the tag filter with the child query results is costly. The necessity of Blockjoin to use a tag filter for each parent and child query incurs a performance penalty.

SIREn on the other hand suffers penalty for query 2 and 4, when the child query is highly sensitive. A specific solution for this is on its way, targeting SIREn release 1.4. We will update this benchmark as soon as available.

### 2.7.1 Query Memory Requirements

While query throughputs are comparable for SIREn and Blockjoin, the issues with the later surface when we consider memory requirements.

To speed up the processing of certain types of queries (Blockjoin queries, facet queries, filter queries), Solr uses various caching strategies, including bit arrays (or bitsets). The memory
requirement for caching a query is proportional to the number of documents in the index. For example for an index containing 10m docs, a bitset would need 10m bits of memory (about 1.2MB).

Given that the Blockjoin approach artificially inflates the number of documents in the index, queries that require bitsets significantly impact memory requirements. For the test dataset, a bitset in SIREn would require \(0.005\) MB \((44,369 / 8 / 1024 / 1024)\) whereas Blockjoin would require \(9.7\) MB \((81,350,921 / 8 / 1024 / 1024)\).

SIREn on the other hand has stable memory usage which is not dependent with the number of nested objects. This leaves more memory for caching additional information, e.g., query filters, facets, and for the operating system to cache index files, which is very important for good performance.

We can look at the impact this has on the various types of queries.

### 2.7.2 Evaluating Nested queries

The Blockjoin approach requires a bitset to be cached for each parent child relationship that is part of the query. The test queries for instance require two bitsets.

Given the large number of nested object types (as explained in the Appendix), thousands of bitsets would be required for querying over all possible paths, each of 10mb size. SIREn on the other hand needs 0 bitsets for nested querying.

The maximum memory utilization during execution of the benchmark queries was measured. With respect to Blockjoin, this measures the impact of caching the two parent filters that are necessary for the execution of the nested queries.

#### Comparison of memory usage during the execution of the query benchmark

<table>
<thead>
<tr>
<th>Memory Usage</th>
<th>Memory Usage per Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blockjoin</td>
<td>20 MB</td>
</tr>
<tr>
<td></td>
<td>9.69 MB</td>
</tr>
<tr>
<td>SIREn</td>
<td>3 MB</td>
</tr>
<tr>
<td></td>
<td>0 MB</td>
</tr>
</tbody>
</table>

SIREn does not need memory per filter

### 2.7.3 Evaluating Filter queries

Filter queries are used for caching query results and reusing them in subsequent queries. They are extensively used for supporting high volumes. Any filter query over the data would require the creation of a docset (either a BitDocSet or a SortedIntDocSet). This would be in
addition to the parent filter bitsets required by Blockjoin for nested queries, as described above.

The memory utilization of Solr was measured (via a profiler) after warming up Solr with 200 random queries (non Blockjoin queries) with a random filter picked from a pool of 50 different filters on object types.

**Comparison of memory increase after the execution of the queries with random filters**

<table>
<thead>
<tr>
<th>Memory Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blockjoin</td>
</tr>
<tr>
<td>SIREn</td>
</tr>
</tbody>
</table>

SIREn requires 4.85 times less memory

### 2.7.4 Evaluating Facet queries

Facet queries are required to support interactive faceting. Lucene supports two distinct modes of faceting: enum and fieldcache (fc).

Enum requires a docset to be cached for each facet value. This is used when the number of unique facet values is low. The docset can be a BitDocSet (i.e., using a bitset described above) or a SortedIntDocSet based on whether the frequency of facet values across documents is high or low respectively.

The Fieldcache mode (which is usually the default one) uses an uninverted field data structure. This mode is recommended when the number of unique facet values is large.

Due to the difference in the index size between Blockjoin and SIREn (81m vs 40k docs), the memory requirement to support faceting is very high for Blockjoin for either of the two methods above.

If for instance, the enum mode is used for the ‘country’ field and if the BitDocSet is used for all the facet values, the Blockjoin would need 853.6MB (88 * 9.7) whereas SIREn would only need 0.44MB (88 * 0.005). However, if the frequency of a facet value is low, the SortedIntDocSet will be used and the total memory requirement will be reduced.

The memory utilization of Solr was measured after requesting a facet query for three fields: main-classification (14k values), country (88 values) and city (12k values). We also measure the average response time to answer the facet query (averaged over 50 runs after warmup).
Comparison of the memory increase and the query time for the execution of the facet query with the enum mode

<table>
<thead>
<tr>
<th></th>
<th>Memory Increase</th>
<th>Query Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blockjoin</td>
<td>289 MB</td>
<td>668.36 ms</td>
</tr>
<tr>
<td>SIREn</td>
<td>75 MB</td>
<td>23.34 ms</td>
</tr>
</tbody>
</table>

SIREn requires 3.85 times less memory while being 28.6 times faster

Comparison of the memory increase and the query time for the execution of the facet query with the fieldcache mode

<table>
<thead>
<tr>
<th></th>
<th>Memory Increase</th>
<th>Query Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blockjoin</td>
<td>3077 MB</td>
<td>90.96 ms</td>
</tr>
<tr>
<td>SIREn</td>
<td>126 MB</td>
<td>8.36 ms</td>
</tr>
</tbody>
</table>

SIREn requires 24.42 times less memory while being 10.88 times faster

Solr was configured with the default filter cache size (512), and with a maximum memory heap size of 4GB.

The fieldcache mode uses the “uninverted field” data structure per field. Underneath, it creates a single int[maxDoc()] per field. We can see from the results that it becomes problematic for Blockjoin to use this mode for facet queries due to a significant increase of memory requirement. Instead, the enum mode must be used. However, caching facet values using filters is suboptimal for the filter cache, especially for fields with medium to large number of distinct values, as it will reduce the cache hit rates. As a consequence, the filters will not be cached and have to be created at each request, significantly increasing the response time.

To summarize, the Blockjoin approach would require extraordinary amounts of memory for relatively tiny index sizes. A typical application would require multiple facet queries, nested queries and filter queries.

2.7.5 Conclusions

As can be seen from the results above, a fundamental issue with Blockjoin is memory requirements.
The test dataset indexes just 44,369 patent grants which have been issued over two months. If you want to index the 2 million or so grants over the last 10 years, the Blockjoin approach would require an index of 3.6 billion docs. Projecting based on a linear increase in memory requirements, you would need 135GB of memory to facet over the three fields used in the test compared to SIREn’s requirement of about 6GB.

Beyond the much smaller memory requirements, SIREn’s indexing model provides better performance for nested queries, greater query flexibility (e.g., support for nested arrays), true schemaless operations along with more advanced query operations such as wildcard path query and positional operators on structural elements.

3 Scaling the relationships: “relational faceted browser”

In this section we’ll introduce the key feature of the SmartOpenData Knowledge Explorer Project, the ability to perform real-time relational faceted browsing.

To get a good initial understanding of what we are talking about it is recommended to first watch the screencast of an early system: the “PivotBrowser”.

An ancestor of the “KnowledgeBrowser” developed for SmartOpenData, the “PivotBrowser” is demonstrated in a screencast here: https://vimeo.com/43331683.

3.1 Introduction: relational faceted browser

The present system generally relates to information navigation and retrieval systems. More specifically, the system relates to an efficient way to perform an advanced “faceted searching” which we call “Relational Faceted Searching”.

Traditional faceted navigation systems are useful for searching a collection of data records where each data record is described by a set of independent facet categories. However, they fail to address the need of searching and exploring datasets with relational structure where (1) users constraints must apply to more than one related collection of data records and (2) the set of matching data records depends on the relationships between those data records and the data records in other collections.

To accomplish this task in a relational faceted navigation system, the state-of-the-art approach relies on a combination of inverted index and relational database technologies. Inverted index technology is used to maps facet values to records, and enables traditional

---

faceted search on the collection of records. Relational database technology is used to index relationships between records and to create a query execution plan that joins the record tables to produce the expected result sets. Joining tables enables to check the existence of relationships (or paths) between multiple related collections of data records, and filters out data records that do not satisfy such constraint.

The problem is that this approach is complex in term of computational requirements. Joining tables as well as enumerating and aggregating facet values are expensive operations both in term of space (i.e., memory) and time (i.e, cpu), limiting the scalability and performance of the system. The problem increases in complexity with the number of data record types and relation types present in the dataset.

Instead of computing the relationships between data records at query time via RDBMS, our method is based on a specific data preprocessing materialization which turns a relational data model into a tree data model. The first step is the “relational facet synthesis”, that is the process of creating a materialized view for each of the records which is composed of both the record itself and of all the other records that are related to it possibly also at multiple levels of relationship. Such materialization takes the shape of a tree. The relational facet synthesis can be computed using graph searching algorithms, e.g., breadth-first search, iterative deepening depth-first search, or using transitive closure algorithms.

The result of the relational facet synthesis is much larger than the original model. However, tree models can be encoded, compressed and searched very efficiently with an appropriate inverted index data structures, for example as in the SIREn methodology..

Once this is done, the desired relational faceted browsing is performed by turning user actions into appropriate Tree queries (which at these point have nothing to do with those that would have been issued to a RDBMS but instead reflect the data in its denormalized tree format) which can be executed very efficiently thanks to the use of the underlying Inverted Index.

Graphically speaking, the difference between the way this task has been usually performed (as known in literature and patents today) and our approach can be seen as in the diagram below:
In this diagram, a relational database composed of Artists, Artworks and Museums is processed to be browsed in a Relational Faceted Browser. The path above shows the classic approach, simply via relational indexing and querying. The PivotBrowser method is shown in the path below. First a “Faceted Materialization” is performed, with full relation trees built for each entity and each relation direct or inferred inverse - then encoding and querying of the trees happens efficiently thanks to specific encoding and query translation based on an underlying inverted index.

The present system, a data-driven information navigation system and method, enables search and analysis of a set of data records by certain common attributes that characterize the data records as well as by relationships among the data records. The system includes:

- a method for performing a relational facet synthesis on a domain of information
- a method for encoding said relational facet synthesis into an inverted index
- a method for selecting and retrieving information from said relational facet synthesis index
3.2 Additional background on the System

3.2.1 On Faceted Browsing

An exploratory interface allows users to find information without a-priori knowledge of its schema. Especially when the structure or schema of the data is unknown, an exploration technique is necessary. Faceted browsing is an exploration technique for structured datasets based on the facet theory.

In faceted browsing the information space is partitioned using orthogonal conceptual dimensions of the data. These dimensions are called facets and represent important characteristics of the data records. Each facet has multiple restriction values and the user selects a restriction value to constrain relevant records in the information space. The values in a facet may be organized:

- simply listed for the user to select, e.g. from a list allowing single or multiple choices
- hierarchically with more general topics at the higher levels of the hierarchy and more specific topics towards the leaves;
- on a timeline if the values represent time information;
- on a map if the values represent geo-localisation information;
- or other visual concepts depending on their types.

For example, a collection of art works can have facets such as type of work, time periods, artist names and geographical locations. Users are able to constrain each facet to a restriction value, such as “created in the 20th century”, to limit the visible collection to a subset. Step by step other restrictions can be applied to further constrain the information space. A faceted browser might allow also other restrictions e.g. based on keyword search across all or some of the fields.

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

3.2.2 Relational Faceted Browsing (AKA Pivoted Faceted Browsing)

Faceted navigation systems are useful for searching a collection of data records where each data record is described by a set of independent facet categories. However, it is needed to search datasets with more complex structure where users constraints must apply to more than one related collection of data records and the set of matching data records depends on the relationships between those data records and the data records in other collections.

Consider a dataset about Museums, Art Works and Artists as pictured below.
At schema level, we can zoom on the relationship between ArtWork and Artist and Museum. Art Works are related to one or more Artists and described by facets such as Type, Period, and then related to one or more Museums which then have facets like Name and Location.

While a typical faceted browser would allow restrict ArtWorks by properties it has directly associated to it, e.g. type, Period and Location, a relational faceted navigation system could be used to search the set of Art Works based on the facets associated to one or more Artist related to the artwork as well as on the facets related to Museums.

Also, a in a relational faceted browser, the focus of exploration can typically change from a type to another, e.g. the user would start browsing artworks, restrict them using some of their facets (e.g. just those in the impressionist period) and then pivot to the set of artist associated to those artworks which have been selected. This can happen iteratively, e.g. once a set of artists is selected, the user can decide to focus on Museums and see only those that are, relationally via Artworks, connected to the artists that were previously visualized. At
each step the system can enumerate, aggregate and count the facet values that are associated with data records in the current constrained information space.

A typical user interface for a relational faceted browser (in this case pivoting between classes of items related to Life Sciences e.g. Compounds, Assays, Activities etc) is displayed below:

![User interface example](image)

**Figure 9** Example of said systems are Freebase Parallax\(^2\) also \(^3\)\(^,\)\(^4\).


3.2.3 Typical, state of the art, implementation

To accomplish this task in a faceted navigation system, a typical approach is to rely on the direct use of database technologies to create a query execution plan that joins the tables and produces the expected result sets. The problem with this approach is that joining tables are an expensive operation either in term of space (i.e., memory) or time (i.e., cpu), limiting the scalability or performance of the system. Furthermore, this operation becomes even more complex since the faceted navigation system needs to enumerate, aggregate and count the facet values that are associated with data records in the current constrained information space.

The problem increases in complexity with the number of relation types present in the dataset. For example, consider a dataset about Art Works, their Artists, and the Museums where these Art Works has been exposed. To locate all Museums that have exposed Art Works from French Artists, the system would have to join three different tables, creating a query execution plan composed of two joins. This approach makes faceted navigation intractable with even a modest number of data records and data record types.

A more advance system, which we consider the current state of the art and the patent closest to this system relies on a combination of inverted index and relational database technologies. Inverted index technology is used to maps facet values to records, and enables traditional faceted search on the collection of records. Relational database technology is used to index relationships between records and to create a query execution plan that joins the record tables to produce the expected result sets. Joining tables enables to check the existence of relationships (or paths) between multiple related collections of data records, and filters out data records that do not satisfy such constraint.

The problem is that this approach is complex in term of computational requirements as mentioned in [1]. Joining tables as well as enumerating and aggregating facet values are expensive operations both in term of space (i.e., memory) and time (i.e., cpu), limiting the scalability and performance of the system. The problem increases in complexity with the number of data record types and relation types present in the dataset.

3.3 Description of the System

The state of the art technique for relational faceted browsing very quickly shows its limitations in terms of performance which force the user to wait long times even on
performing systems and moderately sized datasets. For this reason, relational faceted browsers have not, to the best of our knowledge, been seen outside academic demonstrators.

The present system, instead, allows relational faceted browsing at real time speed, typically with just a few milliseconds between user action and updated user interface.

This is obtained thanks to multiple chained steps which ultimately precompute a specific index which can be queried in response to user actions in a way fundamentally different than in the state of the art but ultimately achieving the same goal.

In short the steps are:

- Facet synthesis
- Encoding facet synthesis into an inverted index
- Inverted index querying in response to user action

### 3.3.1 Method for performing a facet synthesis on a domain of information

In this step - facet synthesis - we will create a materialized structure which is specifically suitable to both fulfill the final task, relational faceted browsing, and to match the high performance capabilities of the specific kind of index we will use in the next step.

We present two methods for facet synthesis, which achieve slightly different results with different costs. The first method is based on a tree structure. This method is the exact method as it does not lead to false positives results among those that are shown to the user. This method suffers from high complexity in certain cases. To overcome this limitation, we present a second method based on the graph reachability concept which trades precision for a much lower complexity.

#### 3.3.1.1 Tree-Based Synthesis

This synthesis can be seen as a materialization step of the data graph that will precompute for each data record all the existing paths to data records in other collections. After synthesis, each data record, of each possible types, will be associated to the root of a tree where each branch of the tree encodes one path linking it to the other records. For example, the figure below depicts the data tree associated to a few data records of different kind.
In the case of many-to-many relationship between two data record types, such as Museum and Art Works, the system materializes the data into a one-to-one relationship form. The consequences is that certain data records are duplicated even within a single tree across multiple branches, such as for example the Museum Louvre in the Artist Cezanne rooted tree in the previous diagram. At this stage, also, any relationship is given an “inverse” counterpart and this inverse relationship is also materialized. For example in the previous picture, the relationship between L’Estaque and Cezanne is materialized as that of “inverse has created”.

Figure 10 Example of data tree associated to a few data records of different kind
In some embodiments, this materialised view can be computed using database technologies, e.g., query execution planning joining data record tables, or graph searching algorithms, e.g., breadth-first search. In other embodiments this can be computed very scalably by distributed computing such as the MapReduce paradigm.

This first approach, however, can be problematic as the materialised view of a data record can grow exponentially with the number of many-to-many relationships between data record types. For example, if there were $K$ data records of type Art Works, and $M$ data records of type Museum, the worst case complexity (i.e., when every Art Works are linked to all the Museums) is $O(K \cdot M)$. If another data record collection of size $N$ would be related to Museum with many-to-many relationships, then the worst case complexity would become $O(K \cdot M \cdot N)$.

### 3.3.1.2 Reachability-Based Synthesis

This method is based on the reachability concept in graph theory and has a considerably lower (computational) complexity than the previous method.

Instead of computing a tree-based materialized view composed of the paths from one data record to all the data records from other collections, this method computes a materialized view composed of all data records from other collections that are reachable from one data record without relation but also without duplication of records within a single record view. After synthesis, each data record will be associated to a list of other reachable data records, i.e., data records that are connected by a path. For example, the figure below depicts the reachable data records associated to the data records from the previous example.

![Figure 11 Example of reachable data records](image)

With respect to the size of a materialized view of a data records, the worst case complexity is linear with the number of data records in the other collections, instead of being exponential. With respect to the previous example, the worst case complexity becomes
$O(K + M + N)$ instead of $O(K \times M \times N)$. However, compared to the tree-based synthesis, information is lost as relations between data records are not kept.

Due to this, by synthesizing the facets using this method however, one obtains a different end result for the end user. While the system will apparently looks and behaves identically as before, there will be difference in the results provided to the users at each step.

The easiest way to explain these differences is that of “precision”: the system will provide all the results that were previously available (no false negatives) but could also be “less precise” as it could include some false positives. We envision that in case the system is made to operate with this kind of facet synthesis the user will be informed.

In some embodiments, this materialized view can be computed using graph searching algorithms, e.g., breadth-first search or iterative deepening depth-first search, or using transitive closure algorithms using database or distributed computing technologies.

### 3.3.2 Method for encoding facet synthesis into an inverted index

Inverted index data structures are commonly used to efficiently retrieve data records which have given values in certain attributes. It is less common, although known in literature, that inverted index can also be used efficiently to retrieve records given nested data structures (undirected, unlabeled, acyclic graphs), the main topic of works such as.

In this section we will show how the previously presented facet synthesis can be mapped as said nested structures which can then be effectively searched by said inverted index systems.

In some embodiments, the nodes of the tree can represent records, attribute names associated to the records, and values associated to these attributes names. Such a tree is depicted below.

---

6 Transitive Closure problem has a worst case complexity of $O(|V^3|)$ with naive algorithm. Transitive Closure can be parallelized efficiently on a computing cluster with a relatively low space requirement.


In other embodiments, the nodes of the tree can represent records and attribute-value pairs. Such a tree is depicted below. In other embodiments, the tree model can be a combination of these two models and/or variations of these models.

Either of these embodiments can then be indexed efficiently using a node-labelled tree approach as reported in state of the art and Appendix A.

3.3.2.1 Multiple inverted index embodiment

In one of the embodiment of system, an index exists per record type indexing all the record views about this record type that have been materialised during the facet synthesis step. In our example, the faceted browser will then have 3 inverted indexes: the artist-index, the artwork-index and the museum-index.
3.3.2.2 Single inverted index embodiment

In another of the embodiment of system, a single inverted index can be used as opposed to one per record type. In this case all the record views materialised during the facet synthesis step are stored together in the same index but are distinguished from each other with a specific “type value”, seen as an extra tree branch materialized in each record view, allowing the selection of only the relevant records from a particular type.

3.3.3 Method for composing index retrieval queries in response to user actions.

An inverted index encoded as in the previous steps is capable of efficiently answering boolean and containment relationship (Parent-Child and Ancestor-Descendant) queries on nested data structures.

The relational faceted browsing can be then obtained as resulting of user actions by composing a query on the multiple inverted indexes (or on the single inverted index) as follows:

3.3.3.1 Navigation State and Pivoting

A navigation state of the faceted navigation system is composed of:

- a set of constraints applied by the user to the information space;
- a focus on a particular data record collection (typically the type of the record e.g. Museums vs Artworks).

First of all the focus on a particular data record type (e.g., now we are looking at “Artists”) determines which inverted index is used for the query (e.g., the Artist-index in this case).

Then it lets consider a set of example constraints “The name of the artist must be Cezanne, and the artwork must be located in New York”.

In logical notation this constraint query becomes:

\[(?\text{Artist name} = \text{Cezanne}) \text{ AND } (?\text{Artist has created} = ?\text{ArtWork}) \text{ AND } (?\text{ArtWork location} = \text{New-York})\].

If the focus of the faceted browser is “Artists”, we obtain the content of the view by selecting the Artist-index and casting the above query as tree query following the view model that has been materialised during the facet synthesis.

This query is shown, in graphical notation, in the Figure below, the node labels “?Artist” and “?ArtWork” represent variables associated to their respective data record collection. The variable that is retrieved by the system is the root node “?Artist”.

The query composition is performed automatically in a way which consider the view model that has been materialised during the facet synthesis. The query will therefore be tree shaped itself, with the tree possibly being as deep and wide as the corresponding view model.

For example, if the user was to focus on “ArtWorks” the same identical constraint query must be written to be executed on top of the ArtWork-index - which reflects the view model of the facet synthesis for the ArtWork record collection. Following the algorithm below, the query in graphical notation would then look as follows:

The query tree rewriting is performed automatically to form the new query tree as follows, which can be seen as a rotation of the root node of the query tree:

Find the variable that is relative to the new focus using tree search algorithms (e.g., ?ArtWork in the previous example)
1. Set such a variable as root of the query tree

2. Change the direction of the left-hand side edges connecting the root node to the previous root node (e.g., the edge connecting ?Artist to hasCreated and the edge connecting hasCreated to ?ArtWork)
3. Replace the relationship connecting the root node to the previous root node by its inverse equivalent. For example, the relationship “has created” between “?Artist” and “?ArtWork” is rewritten into the relationship “has created $^{-1}$” between “?Artwork” and “?Artist”.

3.3.3.2 Single inverted index embodiment

In another of the embodiment of system, a single inverted index can be used as opposed to one per record type as explained previously. In this case, we show in the following diagram an example materialization of a tree query that encodes the same constraint query as above but adds the additional entity type branch allowing the selection of only the relevant entities.
3.3.3.3 Related Books and Scientific Publications

- Overview of the current state of the art in faceted search.
- Model for relational faceted search
- /facet: A Browser for Heterogeneous Semantic Web Repositories by: Michiel Hildebrand, Jacco van Ossenbruggen, Lynda Hardman
- A relational faceted browser for RDF data using SPARQL as backend.
- A relational faceted browser for RDF data using SPARQL as backend.
- gFacet: A Browser for the Web of Data by: Philipp Heim, Jürgen Ziegler, Steffen Lohmann In International Workshop on Interacting with Multimedia Content in the Social Semantic Web, Vol. 417 (3 December 2008), pp. 49-58
- A relational faceted browser for RDF data using SPARQL as backend.
- Overview of different faceted browser systems

4 Scaling data quantity and availability: cross data center replication

One of the goals of the Smartopendata project was that of creating a distributed infrastructure for publishing and operating on environmental linked data at a very large scale, possibly with the ambition of covering the European scale and more.

For this reason it is not thinkable that a full, in production system would operate effectively in production if not in a full distributed way.

Solr already offers solution for distributing indexes across machine, a popular solution called SolrCloud. In this section we discuss however how we extended Solr to support multiple distributed datacentre

The Cross Data Center Replication project (CDCR) created an extension to Solr Cloud which is currently being pushed into the Apache Solr community so that it will benefit everyone at large.
4.1 Introduction, large scale cross datacentre replication

The goal of the project is to replicate Solr and SIREn data to multiple Data Centers. The initial version of the solution will cover the active-passive scenario where data updates are replicated from a source Data Center to a target Data Center, where data updates include adding and deleting documents. Three or more Data Centers can be configured in a daisy-chain, where a target Data Center is configured to be the source Data Center of a third Data Center.

Data changes on the source Data Center are replicated to the target Data Center only after they are persisted to disk. The data changes can be replicated in real-time (with a small delay) or could be scheduled in intervals to the target Data Center.

This replication model is designed to tolerate some degradation in connectivity, accommodate limited bandwidth, and support batch updates to optimise communication.

The design described in this document reflect the current status (February 2015) but might change in the future, e.g., based on the recommendation of the open-source community.

4.2 Glossary

In this section, we cover some key concepts and terms that are used in the document.

- **Node**: A JVM instance running Solr; a server.
- **Cluster**: A cluster is a set of Solr nodes managed as a single unit.
- **Data Center**: A group of networked servers hosting a Solr cluster. In this document, the terms Cluster and Data Center are interchangeable as we assume that each Solr cluster is hosted in a different group of networked servers.
- **Shard**: In Solr, a logical section of a single collection. This may be spread across multiple nodes of the cluster. Each shard can have as many replicas as needed.
- **Leader**: Each shard has one node identified as its leader. All the writes for documents belonging to a shard are routed through the leader.
- **Replica**: A copy of a shard or single logical index, for use in failover or load balancing.
- **Collection**: Multiple documents that make up one logical index. A cluster can have multiple collections.
- **Updates Log**: An append-only log of write operations maintained by each node.

4.3 Design Overview

The following figure gives a reference for the discussion of the CDCR mechanism:
Two kinds of replications exist in the full scenario: Intra-cluster replication, taking place within the Data Center, and cross data center replication (CDC Replication). CDC Replication is performed on a per-shard basis. Each shard leader in the source Data Center will be responsible for replicating its updates to the appropriate shard leader in the target Data Center. When receiving updates from the source Data Center, shard leaders in the target Data Center will replicate the changes to its own replicas.

### 4.3.1 Updates Workflow

The figure below depicts the update workflow with SolrCloud and CDCR.
In detail:

1. A shard leader receives a new data update that is processed by its Update Processor.
2. The data update is first applied to the local index.
3. Upon successful application of the data update on the local index, the data update is added to the Updates Log queue.
4. After the data update is persisted to disk, the data update is sent to the replicas within the Data Center.
5. Only after Step 4 is successful, CDCR reads the data update from the Updates Log and pushes it to the target Data Center. This is necessary in order to ensure consistency between the source and target Data Centers.
   - i.e. - We want to avoid scenarios where a data update is sent to the target Data Center and fails on the source Data Center.

Steps 1, 2, 3 and 4 are performed synchronously by SolrCloud; Step 5 is performed asynchronously by a background thread. Given that CDCR replication is performed asynchronously, it becomes possible to push batch updates in order to minimize network communication. Also, if CDCR is unable to push the update at a given time -- for example, due to a degradation in connectivity -- it can retry later without any impact on the source Data Center.

**4.4 Architecture**

There are a number of key features and components in CDCR’s architecture:
4.4.1 CDCR Configuration

In order to configure CDCR, the source Data Center requires the host address of the Zookeeper cluster (the component responsible for the coordination of distributed machines) associated to the target Data Center. The Zookeeper host address is the only information needed by CDCR to instantiate the communication with the target Solr cluster. The CDCR configuration file on the source cluster will therefore contain a list of Zookeeper hosts. The CDCR configuration file might also contain secondary/optional configuration, such as the number of CDC Replicator threads, batch updates related settings, etc.

CDCR will by default try to replicate all existing collections from the source Solr cluster to the target cluster with the following requirements:

- the collections on the source and target clusters must be both configured to have the same number of shards. Replication of collections where the number of shards differ across clusters is not supported.
- the names of the collections must be identical on both clusters. If a collection does not exist on the target cluster, it will not be replicated. It is therefore required to manually create collections on the target cluster before starting the CDCR.

4.4.2 CDCR Initialization

In a bulk load scenario, where the index needs to be fully rebuilt, the push-based replication will be suboptimal as the Updates Log on the source Data Center will likely grow quickly, as the ingestion speed of the source Data Center will be much higher than the ingestion speed of the target Data Center. In this case, it is preferable to first load the index separately on the source and target Data Centers, and then initialize CDCR to start replicating updates between the two pre-built indexes.

As a result, one requirement is to be able to initialize the replication both on a new empty index and on a pre-built index. In the scenario where the replication is setup on a pre-built index, CDCR will ensure consistency of the replication of the updates, but cannot ensure consistency on the full index. Consistency of the full index will depend of the procedure to pre-build the index.

4.4.3 Inter-Data Center Communication

Communication between Data Centers will occur exclusively between shard leaders and will be one-directional (from source to target) as explained earlier. To initiate a communication, the shard leader on the source Data Center will have to first discover the related shard leader on the target Data Center. This discovery process will be performed by contacting the Zookeeper cluster associated to the target Data Center.
Communication between Data Centers will be achieved through HTTP and the Solr REST API using the SolrJ client. The SolrJ client will be instantiated with the Zookeeper host of the target Data Center. SolrJ will manage the shard leader discovery process.

### 4.4.4 Updates Tracking & Pushing

CDCR replicates data updates from the source to the target Data Center by leveraging the Updates Log.

A background thread regularly checks the Updates Log for new entries, and then forwards them to the target Data Center. The thread therefore needs to keep a checkpoint in the form of a pointer to the last update successfully processed in the Updates Log. Upon acknowledgement from the target Data Center those updates have been successfully processed, the Updates Log pointer is updated to reflect the current checkpoint.

This pointer must be synchronized across all the replicas. In the case where the leader goes down and a new leader is elected, the new leader will be able to resume replication to the last update by using this synchronized pointer. The strategy to synchronize such a pointer across replicas will be explained next.

If for some reason, the target Data Center is offline or fails to process the updates, the thread will periodically try to contact the target Data Center and push the updates.

### 4.4.5 Synchronization of Update Checkpoints

A reliable synchronization of the update checkpoints between the shard leader and shard replicas is critical to avoid introducing inconsistency between the source and target Data Centers. Another important requirement is that the synchronization must be performed with minimal network traffic to maximize scalability.

In order to achieve this, the strategy is to:

- Uniquely identify each update operation. This unique identifier will serve as pointer.
- Rely on two storages: an ephemeral storage on the source shard leader, and a persistent storage on the target cluster.

The shard leader in the source cluster will be in charge of generating a unique identifier for each update operation, and will keep a copy of the identifier of the last processed updates in memory. The identifier will be sent to the target cluster as part of the update request. On the target Data Center side, the shard leader will receive the update request, store it along with the unique identifier in the Updates Log, and replicate it to the other shards.

SolrCloud is already providing a unique identifier for each update operation, i.e., a “version” number. This version number is generated using a time-based lamport clock which is incremented for each update operation sent. This provides an “happened-before” ordering of the update operations that will be leveraged in (1) the initialisation of the update checkpoint on the source cluster, and in (2) the maintenance strategy of the Updates Log.
The persistent storage on the target cluster is used only during the election of a new shard leader on the source cluster. If a shard leader goes down on the source cluster and a new leader is elected, the new leader will contact the target cluster to retrieve the last update checkpoint and instantiate its ephemeral pointer. On such a request, the target cluster will retrieve the latest identifier received across all the shards, and send it back to the source cluster. To retrieve the latest identifier, every shard leader will look up the identifier of the first entry in its Update Logs and sent it back to a coordinator. The coordinator will have to select the highest among them.

This strategy does not require any additional network traffic and ensures reliable pointer synchronization. Consistency is principally achieved by leveraging SolrCloud. The update workflow of SolrCloud ensures that every update is applied to the leader but also to any of the replicas. If the leader goes down, a new leader is elected. During the leader election, a synchronization is performed between the new leader and the other replicas. As a result, this ensures that the new leader has a consistent Update Logs with the previous leader. Having a consistent Updates Log means that:

- On the source cluster, the update checkpoint can be reused by the new leader.
- On the target cluster, the update checkpoint will be consistent between the previous and new leader. This ensures the correctness of the update checkpoint sent by a newly elected leader from the target cluster.

### 4.4.5.1 Impact of Solr’s Update Reordering

The Updates Log can differ between the leader and the replicas, but not in an inconsistent way. During leader to replica synchronisation, Solr’s Distributed Update Processor will take care of reordering the update operations based on their version number, and will drop any operations that are duplicate or could cause inconsistency. One of the consequence is that the target cluster can send back to the source cluster identifiers that do not exist anymore. However, given that the identifier is an incremental version number, the update checkpoint on the source cluster can be set to the next existing version number without introducing inconsistency.

### 4.4.5.2 Replication Between Clusters with Different Topology

The current design can work also in scenarios where replication is performed between clusters with a different topology, e.g., one source cluster with two shards and a target cluster with four shards. However, there is one limitation due to a clock skew (version number) problem across shards.

In such a scenario, a target shard can receive updates from multiple source shards (as the document ids will be redistributed across shards due to the different cluster topology). This means that the version number generated by the source cluster must be global to the cluster in order to keep partial ordering of the updates. However, the version number is local to a
shard. Given that it is likely to have a clock skew across shards, a target shard will receive updates with duplicate or non-ordered version numbers. This does not really cause problems for add and delete-by-id operations, since the local version number replicated to the target cluster will be able to keep partial ordering for a given document identifier. However, this causes issues for delete-by-query operations:

- When a cluster receives a delete-by-query, it is forwarded to each shard leader.
- Each shard leader will assign a version number (which can end up being different between shard leaders) to its delete-by-query, and replicate the delete-by-query to all the target shard leaders.
- A target shard leader will receive more than one delete-by-query, with possibly different version numbers. It will not be possible to duplicate and reorder properly these delete-by-query.

One way to solve the problem of delete-by-query would be to have a clock synchronisation procedure when a delete-by-query is received, which would happen before the leader forwards the delete-by-query to the other leaders. The workflow would look like the following:

- A leader receives a delete-by-query
- This (primary) leader requests a clock synchronisation across the cluster (i.e., among the other leaders). The clock is synchronised by using the highest version numbers across all the leaders.
- At this stage, the primary leader can assign a version number to the delete-by-query, and forwards it to the other leaders.
- The secondary leaders does not overwrite and reuse the version number attached to the delete-by-query.

The delete-by-query command will have the same version number across all the leaders. When the leaders will replicate the commands to the target data center, it then becomes possible to keep the partial ordering, since the source leaders have been synchronised and the delete-by-query commands have all the same version.

Therefore, the problem boils down on how to implement a clock synchronisation procedure. Here is an initial proposal for a future option. Given that a synchronisation will be done rarely (only in the case of a delete-by-query), performance might be not critical for its implementation. A possible solution would be a 2-phase communication approach, where the primary leader will initiate the clock sync protocol, and will request the secondary leaders to:

- block/buffer updates
- send its latest version number to the primary leader
- await the answer of the primary leader with the new clock
- synchronise its clock
It is far from being perfect, as things might become tricky if there are network or communication problems, but this is an initial idea to start discussion.

### 4.4.5.3 Replication of Version Number

In Solr, the version numbers are information that is local to a cluster and are automatically generated by the Distributed Update Processor for each update request. It is currently not possible to force Solr to use a predefined version number. To implement the above strategy, we will have to extend Solr so that version numbers generated by the source cluster and provided as part of the update request are processed and reused internally by the target cluster.

### 4.4.6 Maintenance of Updates Log

The CDCR replication logic requires modification to the maintenance logic of the Updates Log on the source Data Center. Initially, the Updates Log acts as a fixed size queue, limited to 100 update entries. In the CDCR scenario, the Update Logs must act as a queue of variable size as they need to keep track of all the updates up through the last processed update by the target Data Center. Entries in the Update Logs are removed only when all pointers (one pointer per target Data Center) are after them.

If the communication with one of the target Data Center is slow, the Updates Log on the source Data Center can grow to a substantial size. In such a scenario, it is necessary for the Updates Log to be able to efficiently find a given update operation given its identifier. Given that its identifier is an incremental number, it is possible to implement efficient search strategy.

### 4.4.7 Monitoring

CDCR provides the following monitoring capabilities over the replication operations:

- Monitoring of the outgoing and incoming replications, with information such as the source and target nodes, their status, etc.
- Statistics about the replication, with information such as operations (add/delete) per second, number of documents in the queue, etc.

Information about the lifecycle and statistics is provided on a per-shard basis by the CDC Replicator thread. The CDCR API can then aggregate this information at the cluster level.

### 4.4.8 CDC Replicator

The CDC Replicator is a background thread that is responsible for replicating updates from a source Data Center to one or more target Data Centers. It will also be responsible in providing monitoring information on a per-shard basis. As there can be a large number of
collections and shards in a cluster, we will use a fixed-size pool of CDC Replicator threads that will be shared across shards.

### 4.4.9 CDCR API

The CDCR API will be used to control the lifecycle of the replication, communicate specific information across Data Centers, and monitor the replication. Here is a list of the commands understood by the API:

- **Lifecycle**
  - enable(Collection): Enable CDCR on the given collection
  - enableAll: Enable CDCR on all the collections
  - disable(Collection): Disable CDCR on the given collection
  - disableAll: Disable CDCR on all the collections
  - getStatus: Return the status of the CDCR on the current shard - Stop, Running, Error

- **Checkpoint**
  - getLastCheckpoint: Retrieve the last command id that has been successfully processed by the shard

- **Monitoring**
  - getTargets: Return the list of target peers
  - getSource: Return the source peer
  - getQueueSize(Target): Return the queue size for a given target
  - getOperationsPerSecond: Return the number of operations per second
  - getAddsPerSecond: Return the number of add operations per second
  - getDeletesPerSecond: Return the number of delete operations per second

### 4.5 First feedback and conclusions

The CDCR Solr Cloud replication extension has been very favourably welcomed by the Apache community. In fact, the work is being chaperoned into the main branch of Solr by notable members of the community at this point.

At the time of the writing, the system is being tested and improvements are being made. It is expected that the final push of this extension into the core codebase of Apache Solar will happen by April 2015.
5 End to End ETL

To be able to provide the end user with a realtime relational faceted browser interface – something we will describe more in detail in the next 3 chapters – the data needs to be Extracted Transformed and Loaded (ETLed) into the final search indexed (with or without SIREn, according to the configurations).

Our reference ETL architecture is as follows:

![Diagram](image)

**Figure 14 The complete data workflow in the "KnowledgeBroser" navigator**

In detail:

- A state of the art traditional Open Source ETL platform is typically used to transform from the original sources
- ..to a staging DB, either SQL or – as it is the case of SmartOpenData an RDF/Graphstore
- The JSONBuilder component is then in charge of fetching data from the relational or graph/relational sources and creating a set of “materialized documents”.
- The SIREn/Elasticsearch-Solr – Solr/Multidaceter system is used for indexing and backend, described in the previous two chapters
• The frontend queries both the search system backend and the original data source (SQL/SPARQL) to create powerful navigation experience as described in the last chapters of this deliverable.

5.1.1 The JSONBuilder

The JSONBuilder is a component which is used to create materialized documents out of a relational domain.

It operates by:
• An Entity-Relation map (ERmap) is a document that specifies the entities and their inter-connections. The goal is to materialise the entities into a tree.
• One or More JSON Templates, which define how exactly the materialized documents are constructed, for each entity type

5.1.1.1 The ERMAP

The ERmap is a JSON document consisting of two objects:
• “entities”: specifies the entities and their attributes;
• “relations”: specifies the relations between the entities.

5.1.1.2 Entity

Required fields:
• “enumerate”: the query to get all the entities of a class. This query must project one attribute. The project value is then considered as the identifier of the entity. The values must all be distinct.
• “output”: a list of IDs
• “attributes”: the query to get the attributes for a given entity
• “input”: an entity ID which will substitute SOURCEID in the query.
• “output”: a table with attributes about that ID

This query may return more than one row in case of multi-valued attributes. In such a case, the value in the JSON of the multi-valued attribute is an array, in which all elements are distinct. Any duplicates are removed automatically.

The name of a class cannot contain a dot because it is used to separate the class name from an attribute of that class. If it is really needed we can take the approach in SQL where the name is wrapped in brackets.

For example, the following object defines the Project class:
5.2 Relation

The relation is an array which contains a list of relation queries. A relation query links two classes together. A relation query is defined as an array in which both directions of the relation are outlined.

Required fields:

- "source": the source class of the relation
- "target": the target class of the relation
- "query": the query to retrieve the identifiers of that target class.
  - input: an entity ID which will substitute the SOURCEID in the query.
  - output: a list of entity IDs of the target class.
- "label": (OPTIONAL) the name of the relation. This is useful when there are multiple relations between the same source and target classes. If not specified, the name of the target class is used to label the nested objects in the materialised entity.

For example, the following defines the relation between a Project and a Document. The table “projdoc” is another table that contains the pairs of classes within the fields “ida” and “idb”.

The first element of the array defines how to go from a Project to a Document. The other element defines the inverse, i.e., how to go from a Document to a Project.
5.3 Multiple Relations between Two Classes

In the example ERmap below, there are two classes, i.e., Person and Company, which are connected via two relations. Each relation is identified with a different label, i.e., “owns” and “works for”.

```json
{
  "entities": {
    "Person": {
      "enumerate": "SELECT id FROM persons",
      "attributes": "SELECT * FROM persons WHERE id='<SOURCEID>'"
    },
    "Company": {
      "enumerate": "SELECT id FROM companies",
      "attributes": "SELECT * FROM companies WHERE id='<SOURCEID>'"
    }
  },
  "relations": [
    {
      "query": "SELECT ido FROM person WHERE id='<SOURCEID>'",
      "source": "Person",
      "target": "Company",
      "label": "owns"
    },
    {
      "query": "SELECT idw FROM person WHERE id='<SOURCEID>'",
      "source": "Person",
      "target": "Company",
      "label": "works for"
    }
  ]
}
```

5.3.1 Cache Warming

An optional cache warming statement “precaching” can be added to entities or relationships. If this is present, the system will execute those queries and warm the cache.
accordingly. By calculating the size of the objects while warming the cache, they can stop warming the cache after the desired memory max allocation.

5.3.2 The JSON Template

The Json template is 1 document per entity (actually per entity type to be created in the elastic search index) usually the same number as the entities in the ER map. It defines how the data is materialized for that entity in the index. This allows dashboards to be connected to each other e.g. via filter joins or other kinds of joins.

The Json template indicates:
- The root class, e.g. “Person”
- inside the block it indicates either
  - values:* to indicate all the values from the KV map should be materialized
  - values: key1,key2,key3 .. : this indicates which “keys” to materialize here e.g. “first name” “last name” “user id”
  - values: -key1,-key2,key3 : this removes the keys from the final materialization. This is useful in conjunction with values:* for example
    - Article
      - values:*  
      - values:-fulltext
- It then indicates the names of a RELATIONSHIP (not a class) that connects to another class.
  - Inside these relationships other values are nested.

This is then used in the JSON template as follows:

```json
{
   “Person”: {
       “owns” : { “onlyvalues”:”companyid,companyname”),
       “wrotearticle” : { “lessvalues”:”fulltext”}
   }
}
```

5.3.2.1 Functionalities of the component

The JSONbuilder can from an ERMap, create a set of initial JSONTemplates that the user can customize.

This is made with a command line functionality that allows one to generate “level one JSONTemplates” (default) or more levels (E.g. 2-3) which starting from each class will encompass more and more other classes according to the existing relationships.

The JSONbuilder will check at run time:
• if one adds or removes a key value from the template that is not the result of a KV query (it could happen as result of misconfiguration)
• that the resulting Json is not bigger than a predefined limit (will drop the document and generate a big error in the log)

6 Preview: User interface and interaction

Prehamble: this work describes work in progress. It was decided during the execution of the project that Sindicetech would go beyond the simple “indexing infrastructure” (the original scope of D4.1) to also do additional work on the delivery of the user interface.

This section shows the work that is currently being performed on the user interface – some of which is still at mockup/whiteboarding phase.

The final version of this application will be described in the final pilot deliverables (M24).

6.1 The SmartOpenData use cases

Within the SmartOpenData project, the KnowledgeBrowser system described so far will be used to fulfil two use cases:

• Tragsa dataset : a datasets of
  ○ Parcels (for which we know the kind of soil and numerous other indicators)
  ○ Naturalistic reports, of animal and plants and other relevant aspects which can be relationally connected to Parcels
  ○ Other information about the said animals and plants, e.g. from DBPeda
  ○ (optional news about the territory)
• ARPA dataset (similar to Tragsa) has
  ○ Water and Air quality measurements, in territory parcels
  ○ Naturalistic reports (same as above)
  ○ Other information about said animals and plants.

6.2 Shared Prerequisites:

• Scalability, in the area of at least 100x million news and 100x knowledge graph nodes.
• Real time operations. Most if not all the operations are real time, searching, browsing, faceting, feeds are mostly subseconds operations.
• Traceability. The system will allow knowledge sources to be highlighted and filtered out.
• Applicable across multiple scenarios with diverse knowledge structure.
• Configurability by Analysis or Domain expert. Domain experts and analysts must be able to load and configure the browser themselves, by the mean of relatively easy configurations and templates. Configurability must serve to adapt the system to different domains both ad data and at UI level.
• [eventually] quasi real time updates

From the analysis of the above use cases, we’ll define here a Knowledge Data Browser experience.

● **Relational Entity Set Browser**

● **Concept detail page and ad hoc queries** - powered by knowledge graph.

The end user sees these integrated together approximately as follows:

• From the list of entities (e.g. companies/news/parcels/species) one get the “detail page” by clicking on it
• From the detail page a list of “related” entities can be obtained which is then used to generate a list of views that is used in the Browser

### 6.3 Relational Entity Set Browser

The Relational Entity Set Browser is meant to navigate all the entity types in the knowledge graph.

It can be used for example to navigate, in example *News, Species, people, Companies, Locations,* etc.

The Relational Entity Set Browser, can be conceptually seen as a mix between the [PivotBrowser](#) and Elasticsearch [Kibana](#) – a search/analytic application for Elasticsearch which gives widgets and configurable dashboard.

The result is, in other words, a high performance, interactive, highly scalable searchable entity browser, with relational and analytic capabilities. In this system, each one of the types of data is handled by its own “page”, roughly corresponding to the concept of “tab” in Pivotbrowser, or “dashboard” in Kibana.

This is an example of a possible visualization of the “Companies” page in the Relational Entity Set Browser in our use case:
We can see classic Kibana like widgets (e.g. histograms) classic facets. We see in particular a special widget: a “Relational Filter” (top right box). This allows restriction of the news based on properties of the companies that are mentioned in it. This effectively implements the core of the PivotBrowser by adding a “Siren JSON” part to queries using trees stored in the documents.

Other notes:

- This UI mockup is just an example, in general the UI is configurable with widgets that should be the same at least as Kibana e.g. a quantity histogram is shown, this is useful.

**Rotating to other concepts (a la Pivotbrowser)**

Rotating to other concepts “a la Pivotbrowser” is possible as each entity type has its own dashboard. E.g.

- News (depicted above)
- Companies (geo widget, breakdown by sector widget)
- People (breakdown by position)

In the PivotBrowser, rotating happens via Tabs. Kibana does not have these tabs. A possibility is that the same “relational filter” widget, used to show restrictions that have been imposed on the set by filters in the other page, also includes the counts for the other entities.
6.3.1 Concept Detail Page

The following is a mockup where a “company detail” page is visualized:

![Mockup of a company detail page]

We see 5 areas:

- **A Top part**, shows general information available in the graph (owner, area etc...)
- **A “Sources” area**, where the sources of information below are reported, with checkbox. This is a feature that works in a twofold way, one the one hand one can select or deselect the wanted sources, in the other by hovering on information, one or more sources highlight to show the origin of the data. (Similar to the project [sigma](https)).
- **Tab 1: News/Documents**
  - This reports simply the most relevant news (or documents, anything textual in general) directly connected to the entity “Direct News”. This is a quite simple “news” form, distinct from what’s possible in the other tab (see below). One can click on the “All Direct News” to go to the [Relational Entity Set Browser](https) for those news (which then allows filtering by all other filers).
○ Notice that one sees the sentiment indicator (+ or -) per type of news.
● Tab 3: Other Info
○ This simply contains more ad hoc widgets. E.g. can show any information coming from the knowledge graph or other sources.
● Tab 2: Related Entities. This reports a list of related entities.
○ E.g. in the case of “companies” this could be Employees, Competitors, Locations, and Products. These are visualized in a meaningful way (e.g. with links, pictures, etc).
○ Each list has a “Read Related News” button and a prominent “All Related News” is also provided. The following is a whiteboard mockup of the idea:

![Whiteboard Mockup](image)

**Figure 17 Example of a whiteboard mockup**

Functionally, the News button and the All Related news work by again going to the Relational Entity Set Browser, and visualizing all the related news (E.g. to all the entities which are “Related”) as a browsable list.

It is to be notice that at business level all sort of restrictions and queries can be used to generate “related entities”. E.g. in case of Acme, related entities might be “Top 10 performing Shoemakers in Italy for year 2014”.

When rotating to the “Relational Entity Set” to see “news about these”, it will likely not be possible to map these sort of “entity descriptions” to filers, e.g. to show these restrictions in the “Relational Filter’ Instead these will appear as extra “Facets/Filters”.
The following whiteboard mockup depicts the Relational Entity Set Browser with the additional “facets” injected by the above process.

Note also that the result will also be ranked by “relevance” according to mechanisms that will be described later.

6.3.1.1 Preliminary actual screenshots.

At the time of writing, a preliminary version of the KnowledgeBrowser is available. Loaded with Spaziodati data (which was available prior to the Tragsa and ARPA data) it looks as follows:
Figure 19 Preliminary version of the KnowledgeBrowser
7 Preview: applications in Pilots: KnowledgeBrowser for Tragsa and ARPA

In this section we will quickly go over the work that is currently being done in conjunction with Spaziodati ARPA and Tragsa for the application of the above infrastructure and application to their data.

7.1 Tragsa Data

Tragsa has data about “Parcels” of land. Natura 2000 DB has data about which species are in each area. KnowledgeBrowser (or pivotbrowser) will allow correlating parcels data with species data.

For example, most species that are endangered are in parcels that are small or big, that are cultivated using this or that crop, etc.

Example use case:

- Pick an endangered species (e.g. blue bird of the mountain) see what are the most common cultivations that are in these parcels.
- Enlarge to similar species (e.g. same category “bird” and “endangered”) see how the above changes.
- See the interested parcels on a map.

7.2 ARPA data

ARPA has data about pollution of water and air. Similarly Natura 2000 DB has data about species in which air. The intersections of these 2 databases will allow us to zoom in and pivot on Species and Pollutants and the like.

Example use case:

- Select a pollutant.
- See which species are the most endangered due to their presence in areas where that pollutant is prominent.
- Click on a species, enlarge it (e.g. via the “Related species”) and then see which areas it is present in (e.g., on a map).

7.3 The goal

The goal is that the KnowledgeBrowser will be able to answer the questions of the above use cases. Sindicetech will deliver the tool to Spaziodati which can then configure it for the use cases.