ASAP: **A**daptive, highly **S**calable **A**nalytics **P**latform)

EC project: FP7-ICT-2013-11 ASAP 619706

Description of the initial dataset

Deliverable no.: D8.1

Date: March 2014

Table of Contents

Deliverable Title: Description of the initial dataset

Filename: D8.1_V2

Author(s): Philippe Rigaux, France Lasfargues

Date: March 2014

Start of the project: 01/03/2014

Duration: 3 years

Project coordinator organization: FORTH

Deliverable title: Description of the Initial dataset

Deliverable no.: 8.1

Due date of deliverable: M1

Actual submission date: March 2014

Dissemination Level

x	PU	Public
	PP	Restricted to other programme participants (including the Commission Services)
	RE	Restricted to a group specified by the consortium (including the Commission Services)
	CO	Confidential, only for members of the consortium (including the Commission Services)

Deliverable status version control

Version	Date	Author
0.1	24.03.2014	France Lasfargues
0.2	27.03.2014	Philippe Rigaux
v1.0		

Abstract

The present deliverable describes the initial dataset provided by Internet Memory Research (IMR) at the beginning of the project. We also briefly introduce the infrastructure used for data storage, replication, and access, as well as the IMR API.

Keywords

Web data, Hadoop/HBase, Java API.

1 The Internet Memory Infrastructure

Our infrastructure relies on a data center hosted by Internet Memory. It contains a continuously expanding set of servers designed for reliable long-term storage of big datasets. The software components chosen to manage these datasets is the well-known Hadoop suite, with two prominent layers:

1. HDFS (Hadoop Distributed File System, http://hadoop.apache.org), a distributed file system natively equipped with replication, load balancing and fault tolerance features;
2. HBase (http://hbase.apache.org), a persistent distributed data structure with indexing capabilities, designed for very large datasets (TeraBytes or even PetaBytes).

1.1 Architecture

On top of this infrastructure, Internet Memory provides a platform called **Mignify**, supporting the execution of analytic workflows on Big Data. Figure 1 shows the main components of the platform.

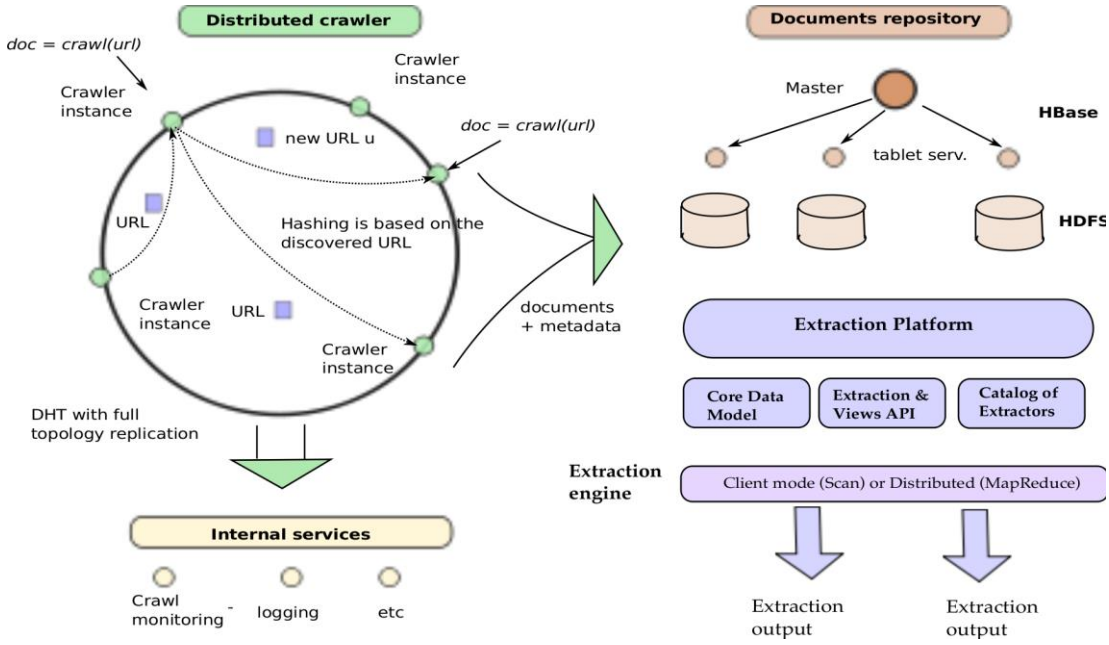


Figure 1: Mignify platform schema

1. the left part is a crawler, MemoryBot, designed to operate at large scale;
2. the top-right part is the Hadoop/HBase repository where datasets are persistently stored;
3. the bottom-right part is the Mignify extraction platform.

1.2 Crawler

The crawler is an internal tool and its design and operational features is not described in detail within this deliverable. It is important, however, to note that (i) MemoryBot is a distributed software, and its performance (number of resources collected per time unit) is proportional to the number of machines (the “cluster”) assigned to a crawl; (ii) MemoryBot does not directly insert in the repository, but rather produces compressed archive files which are subsequently “ingested” in Hadoop/HBase.

1.3 Data ingestion

The ingestion process is a distributed process composed of a set of tasks operating in parallel. Each task takes a compressed archive file, extracts Web resources, and inserts these resources in a HBase table. During ingestion, analytic operations can be applied either to a single resource or to group of resources. They produce feature (e.g., language of the document) which are stored as meta-data in HBase along with the web resource itself.

The representation of a *resource* (or document) in Mignify is complex. First, they are identified by their URL (the *rowkey* in HBase). Second, we store and preserve several *versions* of a resource, each tagged by a timestamp. MemoryBot indeed repeatedly scans the Web, searching for updates of resource. The temporal stack of versions is a distinctive feature of Mignify. Finally, we store not only the mere content of a resource, but also associated features, which are either obtained from the Web server during the crawl (e.g., MIME type, length, HTTP fields) or *extracted* during the ingestion. A standard feature is for instance the MIME type of the document. More sophisticated extractors may produce for instance, for a textual document, features like entity references or sentiment annotations.

At the HBase level, collections of Web resources are organized as tables with two mandatory column families, *content* and *meta*, and optional column families containing extracted information, defined on a per-collection basis.

In order to properly represent the complex object, which constituted a resource, Mignify encapsulates its representation with an *Object-Document Model (ODM)*. The Java API maps the HBase row storage to the ODM, which presents a consistent and simplified representation of resources. We provide to our ASAP partners the Java API.

2 Web collections

We briefly describe the organization of collections in Mignify, and then give the figures of collections provided to the ASAP partners.

2.1 The Content DB and Source DB

Mignify maintains two distinct databases: the *Content* database and the *Source* database.

1. The *Content DB* stores individual Web resources, identified by their URL and timestamp (time of capture).
2. The *Source DB* stores a qualified list of *Web sources*, i.e., sites, Blogs, news sites and other Web spots which have been identified as periodic publishers of user-defined content.

Currently, a source is identified by a RSS feed, which is investigated by Mignify to check that it does correspond to an actual publication source (most RSS on the Web do not produce anything). Mignify may enlarge this definition in the future.

The two databases are not independent from one another (Figure 2). Actually, sources are detected during ingestion of resources in the content database. Therefore, a *content* collection is always associated to one and only one *source* collection. A source *s* in the collection refers to Web resources published by *s*, and each such Web resource may in turn refers to the source it belongs to. The Figure shows that source *s1* for instance has published Web resources *u1* and *u4*. On the other hand, the Web resource *u2* in the *ContentDB* does not belong to any source. It might be a static web page, not part of a so-called active source.

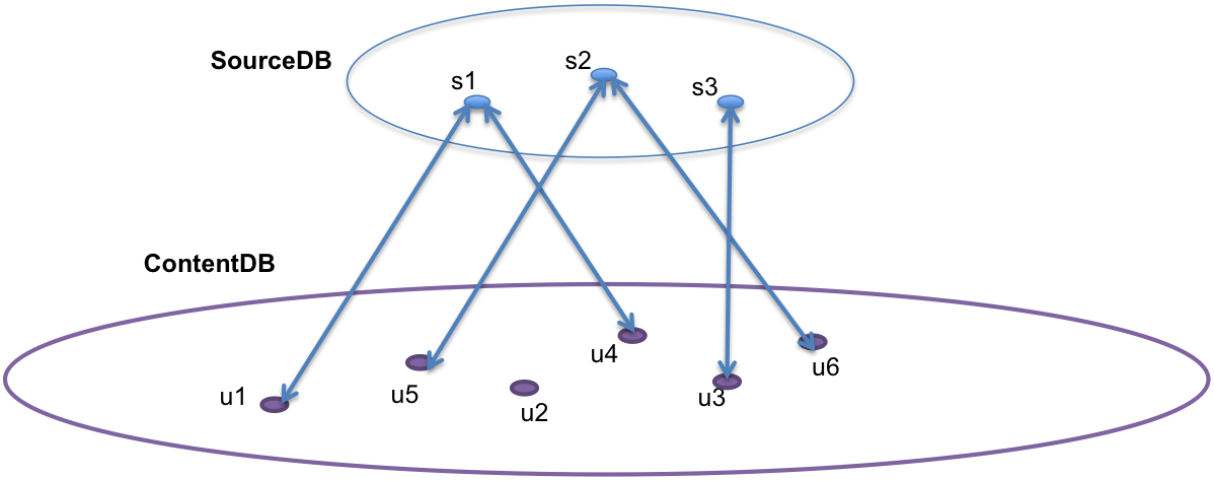


Figure 2: Mignify databases

Resources can be accessed thanks to their unique identifier (combination of a URL and a timestamp), or through *queries* expressing criteria over the features produced by the baseline view. It is possible for instance to query a collection by specifying a combination of language code, MIME type and time interval ("Give me all PDF documents in French collected during this period").

A *source* is a Web place which produces content on a regular basis, possibly with a more or less constant periodicity. It can be a Blog, a news site, a Forum, and in general any Web site where any number of users publish content.

Sources are detected during ingestion in a *ContentDB* collection. Typically, a RSS document is found, and considered as a candidate of a source. Mignify further examines this candidate by fetching several Web pages referred to by the feed, and decides, based on the characteristics of the pages, whether the feed can be validated as an original source of content. Several features are then produced for the source:

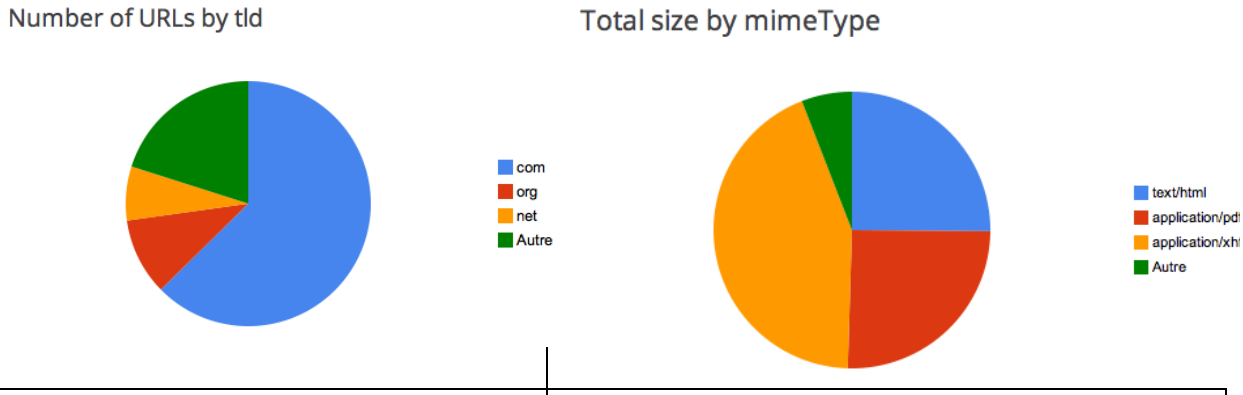
1. its main language;
2. a representative sub set of the entities found in the source's pages;
3. the refreshment rate of the source; this feature determines how often Mignify will inspect the source to fetch newly published content;
4. a template analysis which finds where in a source page the main content can be found; this allows to extract *articles* from the page, ignoring boilerplate information (menus, advertisements, images, etc.)

Finally, Mignify records with a source a list of its most recently published pages. These pages can be found in the *ContentDB* collection.

2.2 The Web resources collection

For ASAP, we provide an initial collection with 600 Million pages, that correspond to 20 Terabytes of storage in Mignify. This collection is a snapshot on international news and websites. It was started on May 2013 on top 1 Million URLs ranked by Alexa. From the websites crawled, we extracted the RSS. For each RSS, a refresh rate is calculated to refresh the collection by getting new content. All the RSS are recorded in the sources DB. Thus, the collection is extended with new Web resources by continuous crawling. Using the Mignify API, users can access that data.

Furthermore, all the content archived is available through a full text index. By using the dashboard, users can browse the collection through the index and can access some collection's statistics. Figure 3 shows an example of the statistics on the collection, as they can obtained from the Web interface of Mignify, open to our partners.



Distribution per PLD	Distribution per Mine Type

Figure 3: Example collection statistics

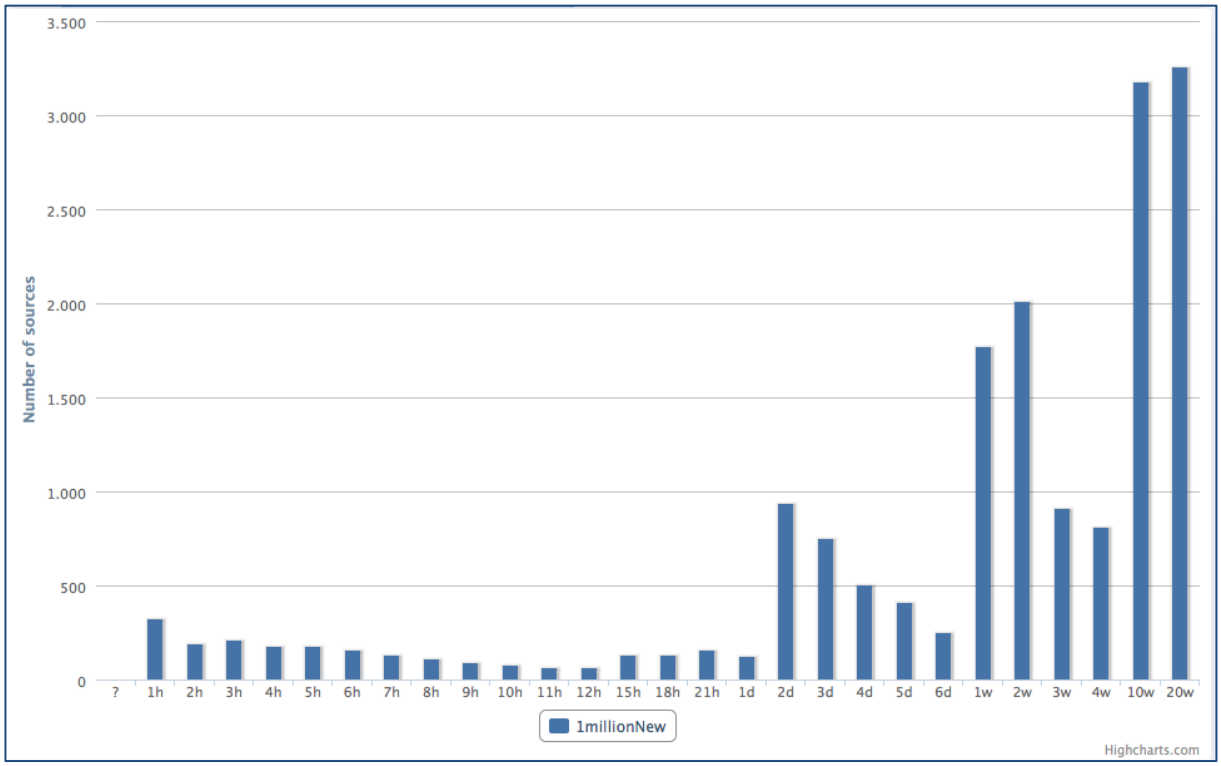


Figure 4: Dataset distribution

2.3 The RSS collection

The source database, initially open to ASAP, contains 600,000 RSS feeds, associated to 11 million resources (the RSS item pages) and 50 Gigabytes of data.

The following information is available for each source:

- Source Name
- Current Strategy: refresh cycle and refresh rate
- Last Refresh Date

- Last Validation Date
- List of Last Updates

The analysis on active sources comprises the *extraction of features* from the source (i.e., Web feed) and from the Web pages that are dynamically discovered to be linked to this source.

Figure 4 shows the refreshment rate distribution of this dataset.

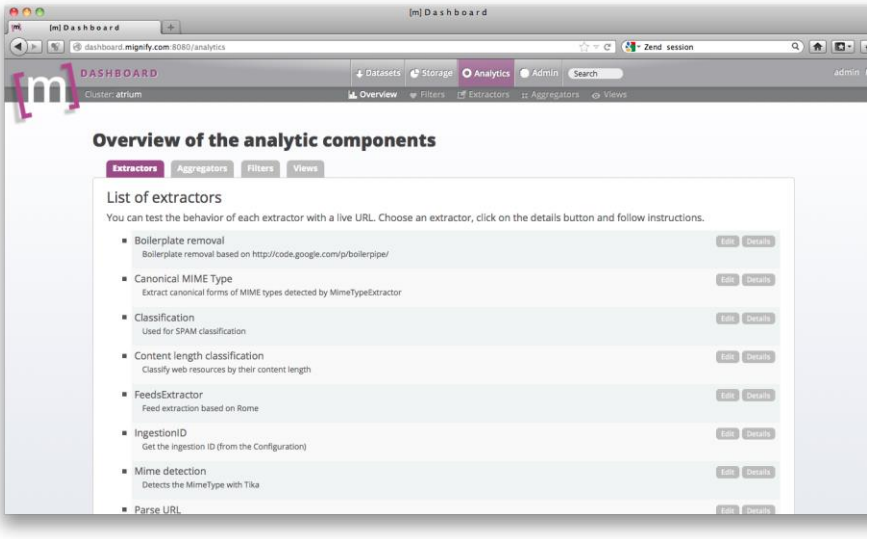
3 Accessing collections

ASAP partners can access the collections by three means:

1. The Mignify dashboard, a Web application which allows users to browse, search and inspect the collection content, and consult statistics.
2. A Java API which extends the standard HBase API.
3. A REST API, which is not described in this deliverable as it is not convenient for fast access, as required by ASAP.

3.1 The Mignify dashboard

The following screenshot shows the Web interface of Mignify. An account has been created for ASAP personnel.



3.2 The Java API

Our collections can directly be accessed with he HBase API. For convenience, we also propose a specialized API tailored to the specific structure of our data.

Tables and rows are wrapped in the API as, respectively, “Collection” and “Row” objects. Both “Collection” and “Row” are abstract classes which must be sub-typed according to the properties of the HBase specific objects. Both classes are also closely related: defining a new subclass of “Collection” necessarily comes with a companion “Row” subclass.

A “Row” is made of *versions*, instances of a subclass of “RowVersion”. Versions are managed as first-class citizens in our model and the API takes care of preserving the consistency of versions in HBase.

3.2.1 Collections

The “Collection” class is parameterized by two types: “KEY_TYPE” and “ROW_TYPE”, the latter being a subclass of “Row”:

```
public abstract class Collection<KEY_TYPE, ROW_TYPE extends Row<KEY_TYPE>>
```

An instance of “Collection” is essentially a container for “Row” objects and provides services at the “Row” set level. One of the most important aspects of the persistence management is that “Row” instances can be stored and retrieved from an associated HBase table. This table is specified by the “useTable” method:

```
public final void useTable(String tbName) throws IOException
```

No persistence-related method can be successfully called if a target table has not been defined.

When implementing a “Collection” subclass, we must specify the “Row” subclass whose instances will populate the collection. This is done by providing a concrete implementation of the following abstract method:

```
public abstract Class<ROW_TYPE> setRowClass();
```

The “Row” class encapsulates the *schema* of a row (see below). This schema defines, among others, the column families of an HBase table apt to store the rows. At the “Collection” level, we are therefore able to create a table if needed:

```
public void createTable(String tbName);
```

When a table is created, it automatically becomes the current table, and rows can be inserted immediately after. We refer to the JavaDoc API documentation for the list of methods supported at the “Collection” level.

3.2.2 Rows and versions

A “Row” represents a Java object which can be smoothly integrated in an HBase and MapReduce environment.

- A “Row” object can be inserted in a “Collection”, which takes care of storing the row in HBase.
- Class “Row” also implements the “Writable” interface and can therefore be used as a value in a MapReduce job.
- A “Row” object has a one-to-many relationship with “RowVersion”, each version being indexed by a timestamp.

“Row” is an abstract class:

```
public abstract class Row<KEY_TYPE> implements Writable
```

The main method to implement a concrete subclass is “createSchema”:

```
protected abstract void createSchema();
```

The method instantiates a “RowSchema” object, containing the list of columns (qualifiers), their association to column families, and their types. Here is an example, extracted from the “Resource” subclass implementation. We add to the schema (an internal property of “Row”, instance of “RowSchema”) a column "mime", to be stored in the "metadata" column family, with type “String”:

```
schema.addColumn("mime", "metadata", String.class);
```

Next, any subclass of “Row” must define the type of its version instances. This is done by overriding the following abstract method:

```
protected abstract Class<? extends RowVersion> getRowVersionClass();
```

Versions can then be added to a “Row” instance as follows:

```
public <T extends RowVersion> T createRowVersion(long timestamp)
```

Values are actually stored and managed at version level. The “RowVersion” implementation takes in charge the coding and decoding of values in “bytes” prior to any insertion or retrieval from HBase. Implementing subclasses of “RowVersion” involves the definition of an appropriate getter / setter interface. For instance, “ResourceVersion” features the following getter for the MIME type:

```
public String getMime() {
    return (String) getQualifierValue("metadata", "mime");
}
```

Note the “getQualifierValue()” method provided by “RowVersion”. The setter is similar:

```
public void setMime(String mime) {
    setQualifierValue("metadata", "mime", mime);
}
```

This defines a typed interface over HBase rows content. Any logic applied to a row must be implemented as a method at subclass level.

3.2.3 Example: “ResourceColl”, “Resource”, and “ResourceVersion”

Classes “ResourceColl”, “Resource”, and “ResourceVersion” are respectively extensions of “Collection”, “Row”, and “RowVersion”, for collections of Web data. “Resource” is defined as:

```
public class Resource extends Row<String>
```

First, let us look at the implementation of the “createSchema()” abstract method in “Resource”:

```
@Override
public void createSchema() {
    schema = new RowSchema<String, Resource>();

    schema.addColumn("content", CONTENT_CF, byte[].class);

    schema.addColumn("ip", METADATA_CF, String.class);
    schema.addColumn("header", METADATA_CF, String.class);
    schema.addColumn("length", METADATA_CF, Long.class);
    schema.addColumn("statusCode", METADATA_CF, String.class);
    schema.addColumn("digest", METADATA_CF, String.class);
    schema.addColumn("mime", METADATA_CF, String.class);
}
```

From this specification, the API gets the list of columns, their types, and knows the two column families (resp. “CONTENT_CF” and “METADATA_CF”, constants defined in “Resource”).

Versions of a “Resource” object are represented with “ResourceVersion”:

```
@Override
  protected Class<? extends RowVersion> getRowVersionClass() {
            return ResourceVersion.class;
  }
```

The “ResourceVersion” implementation is essentially an interface with getters, setters, and ad-hoc method related to Web data content management. (See the Javadoc for more information).

We now turn to the “ResourceColl” implementation:

```
public class ResourceColl extends Collection<String, Resource>
```

Objects in this class manage collections of “Resource” objects:

```
@Override
   public Class<Resource> setRowClass() {
      return Resource.class;
   }
```

“ResourceColl” implements specific methods for managing sets of Web resources.

Concluding, this summarizes the description of the data set, collection method and data access API regarding the data used in the IMR Mignify web analytics application.