The Challenge of using Map-reduce to Query Open Data

Mauro Pelucchi[1], Giuseppe Psaila[2] and Maurizio Toccu[3]
[1]*University of Bergamo, Italy (current affiliation: Tabulaex srl, Milan, Italy)*
[2]*University of Bergamo, Italy*
[3]*University of Milano Bicocca, Italy*

Keywords: Retrieval of Open Data, Single Item Extraction, Map-reduce.

Abstract: For transparency and democracy reasons, a few years ago Public Administrations started publishing data sets concerning public services and territories. These data sets are called *open*, because they are publicly available through many web sites.
Due to the rapid growth of *open data corpora*, both in terms of number of corpora and in terms of open data sets available in each single corpus, the need for a *centralized query engine* arises, able to select single data items from within a mess of heterogeneous open data sets. We gave a first answer to this need in (Pelucchi et al., 2017), where we defined a technique for blindly querying a corpus of open data. In this paper, we face the challenge of implementing this technique on top of the *Map-Reduce* approach, the most famous solution to parallelize computational tasks in the *Big Data* world.

1 INTRODUCTION

Public Administrations, like central and local governments, are publishing more and more data sets concerning their activities and citizens. Since these data sets are open, i.e., publicly available for any interested people and organizations, they are called *Open Data*. The motivation for such an effort is transparency, a key factor in democracy. However, in (Carrara et al., 2015) the notion of *Open Data Value Chain* has been introduced: the idea is that by making data sets publicly available, citizens and organizations can make decisions concerning economical activities that generate new value (increasing the local or global GDP), taxes increase and largely compensate the effort to publish Open Data. This is the ROI (Return of Investment) of publishing Open Data (see Figure 1).

Analysts and journalists (e.g.) that need to find the right data for their research or article have to visit many portals and spend a lot of time in searching data sets, downloading them and looking into them for the desired pieces of information.

Here is our view of the problem: if there were a central system that provides a centralized and global view of open data sets, users could query this system to retrieve open data sets that better fit their needs.

In (Pelucchi et al., 2017) we addressed the first part of the problem, i.e., defining a technique for querying an open data corpus in a blind way. In fact, users (e.g., analysts and journalists) do not actually know names and structures of data sets; in contrast, they make hypotheses that the query engine should solve w.r.t. real data sets. For this reason, the technique mixes a query expansion mechanism, which relies on string similarity matching, and the classical *VSM Vector Space Model* approach of information retrieval.

In this paper, we face the challenge of developing the *Hammer* prototype, the testbed built for validating our technique, by exploiting the *Map-Reduce* approach, in several parts of the computation. This approach is very popular in the area of *Big Data*, since it parallelizes the computation in a native way and easily scales when the number of nodes involved in the computation increases. As shown in (Dean and Ghemawat, 2008), *Map-Reduce* is a programming model for processing large and complex data sets based on two primitive functions: a *map* function, that processes source data and generates a set of intermediate *key/value* pairs, and a *reduce* function, that takes the intermediate set and merges all *values* with the same *key*. We will show how the prototype behaves in different configurations of the testbed, showing that the *Map-Reduce* approach is promising, but requires very efficient implementations.

The paper is organized as follows. Section 2 shows the concept of *blind way* to querying Open

Pelucchi, M., Psaila, G. and Toccu, M.
The Challenge of using Map-reduce to Query Open Data.
DOI: 10.5220/0006487803310342
In *Proceedings of the 6th International Conference on Data Science, Technology and Applications (DATA 2017)*, pages 331-342
ISBN: 978-989-758-255-4

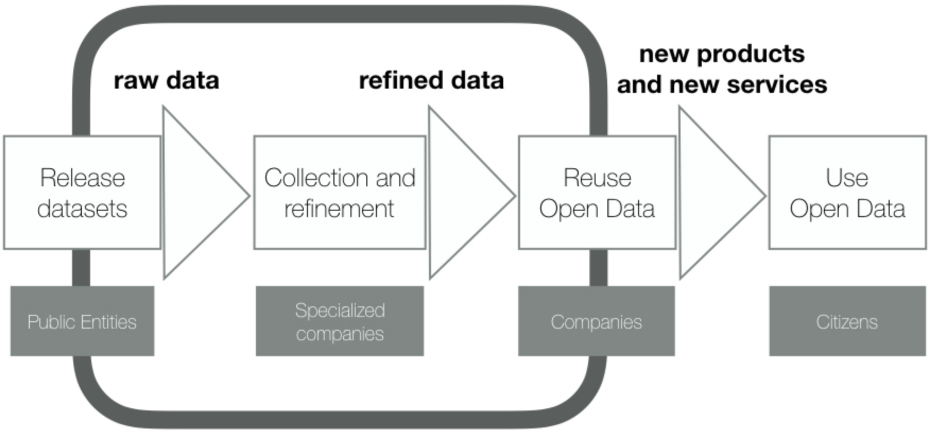

Figure 1: The Open Data Value Chain.

Data, defines the problem (Section 2.1) and summarizes the retrieval technique (2.2). Section 3 describes the *Hammer* prototype and how the *Map-Reduce* approach is exploited. Section 4 presents the experimental settings and the results in terms of execution times. Section 5 discusses some related work and Section 6 draws conclusions and future work.

2 BLINDLY QUERYING OPEN DATA PORTALS

"Every day I wake up and ask, 'how can I flow data better, manage data better, analyze data better?"' says Rollin Ford, the CIO of Wal-Mart, like reported by (Cukier, 2010). Data are everywhere and now they are not only within information systems of organizations, but the web provides a mess of data: Open Data are a meaningful and valuable part.

Users of Open Data Portals wishing to retrieve useful pieces of information from within the mess of published data sets, have to face several issues.

- *Only keyword-based search.* Search engines provided by open data portals usually provide users with a simple keyword-based retrieval, thinking about data sets as documents without structure. However, data sets are usually structured data sets, provided as CSV (comma separated values) files (often, the first row reports the names of fields and the other rows are the actual data set) or JSON files.
- *Large Data Sets.* Usually, data sets contain hundreds or thousands of items; often, users are interested in a few of them, those with certain characteristics. Traditional search engines provided by portals return the entire data set, instead of returning only the items of interest.
- *Heterogeneity of data sets.* The structure of data sets is not coherent, even though two data sets concern the same topic. In fact, different fields, different names for similar fields and, even, different languages are often used. For an analyst, it becomes very difficult to get the desired items sparse in such a mess of data sets.
- *Schema Unawareness.* Finally, the user cannot be aware of the real schema of thousands of data sets, as well as he/she cannot knows the names of thousands of data sets in advance.

The consequence of these considerations is that analysts must query open data corpora in a *blind way*, i.e., they do not know names and structures of data sets, but should formulate a query by means of which they express what they are looking for. At this point, it is responsibility of the query mechanism to address the query to the data sets that possibly match the user's wishes.

2.1 Problem Definition

We now define the problem, by reporting some formal definitions.

Definition 1: **Data Set.** An *Open Data Set ods* is described by a tuple

ods: <ods_id, dataset_name, schema, metadata>

where *ods_id* is a unique identifier; *dataset_name* is the name of the data set (not unique); *schema* is a set of field names; *metadata* is a set of pairs *(label, value)*, which are additional meta-data associated with the data set. □

Definition 2: **Instance and Items.** With *Instance*, we denote the actual content of data sets. It is a set of *items*, where an item is a row of a CSV file, as well as a flat object in a JSON vector. □

Definition 3: **Corpus and Catalog.** With $C = \{ods_1, ods_2, \ldots\}$, we denote the corpus of *Open Data sets*. The *catalog* of the corpus is the list of descriptions, i.e., meta-data, of data sets in C (see Definition 1). □

Now, we introduce the concept of *query*, by explaining the way we intend the notion of *Blind Querying*.

Definition 4: **Query.** Given a *Data Set Name dn*, a set P of field names (properties) of interest $P = \{ pn_1, pn_2, \ldots\}$ and a selection condition sc on field values, a query q is a triple $q\!: <dn, P, sc>$. □

As an example, suppose a journalist wants to get information about high schools located in a given city named "My City". The query could be

q :$<dn=$`Schools`,
  $P=$ {`Name, Address, Reputation`},
  $sc=$ (`City="My City" AND`
    `Type="High School"`) $>$.

However, there could not exist a data set with name `Schools`, as well as desired schools could be in another data set named `SchoolInstitutes`. So, items of interest could be obtained by a different query, i.e.,

nq_1 :$<dn=$`SchoolInstitutes`,
  $P=$ {`Name, Address, Reputation`},
  $sc=$ (`City="My City" AND`
    `Type="High School"`) $>$.

Query nq_1 is obtained by rewriting query q: the data set name $q.dn$ =`Schools` is substituted with `SchoolInstitutes`, becoming $nq_1.dn$ =`SchoolInstitutes`.

Another possible rewritten query could be obtained by changing property `Type` with `SchoolType`. In this case, we obtain the following rewritten query nq_2.

nq_2 :$<dn=$`Schools`,
  $P=$ {`Name, Address, Reputation`},
  $sc=$ (`City="My City" AND`
    `SchoolType="High School"`) $>$.

In practice, the substitution of a term with a similar one originates a new query that could find out data sets containing items of interest for the user. Nevertheless, a third query nq_3 could be obtained by substituting two terms.

nq_3 :$<dn=$`SchoolInstitutes`,
  $P=$ {`Name, Address, Reputation`},
  $sc=$ (`City="My City" AND`
    `SchoolType="High School"`) $>$.

Query nq_3 looks for a data set named `SchoolInstitutes` and selects items with field `SchoolType` having value `"High School"`.

Queries nq_1, nq_2 and nq_3, similar to the original one q, constitute the *neighborhood* of q; for this reason, they are called *Neighbour Queries*.

The concept of neighbour queries explains our idea of *blind querying*. Since users cannot know names and structure of thousands of data sets in advance , they formulate a query q, making some hypotheses about data set name and property names. At this point, the system generates a pool of *neighbour queries*, by looking for similar terms in the catalog of the corpus. Then, for each neighbour query, the system looks for data sets that match the query and contains items satisfying the selection condition. Hereafter, we summarize the problem.

Problem 1: Given an open data corpus C and a query q, return the result set $RS = \{o_1, o_2, \ldots\}$, that contains items (rows or JSON objects) o_i retrieved in data sets $ods_j \in C$ ($o_i \in Instance(ods_j)$) such that o_i satisfies query q or a neighbour query nq obtained by rewriting query q. □

The above problem is formulated in a generic way. In the next Section, we present the technique we developed to solve the problem.

2.2 Retrieval Technique

The technique we developed to blindly query a corpus of Open Data is fully explained in (Pelucchi et al., 2017). Here, we report a synthetic description, in order to let the reader understand the rest of the paper.

The proposed technique is built around a query mechanism based on the *Vector Space Model* (Manning et al., 2008), encompassed in a multi-step process devised to deal with the high heterogeneity of open data sets and the blindly query approach (the user does not know the actual schema of data sets in the corpus).

- Step 1: *Query Assessment.* Terms in the query are searched within the catalog. If some term t is missing, t is replaced with term $\bar{t}$, the term in the meta-data catalog with the highest similarity score w.r.t. t. We obtain the *assessed query* $\overline{q}$.
- Step 2: *Keyword Selection.* Keywords are selected from terms in the assessed query $\overline{q}$, in order to find the most representative/informative terms for finding potentially relevant data sets.

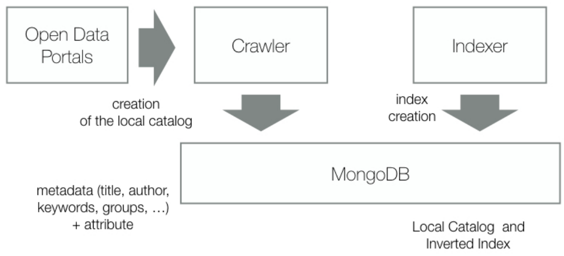

Figure 2: Crawling and Indexing Remote Open Data Portals.

- Step 3: *Neighbour Queries.* For each selected keyword, a pool of similar terms which are present in the meta-data catalog are identified, in order to derive, from the assessed query $\overline{q}$, the *Neighbour Queries*, i.e., queries which are similar to $\overline{q}$. For a neighbour query, the set of relevant keywords is obtained from the set of relevant keywords for $\overline{q}$ by replacing original terms with similar terms. Both $\overline{q}$ and the derived neighbour queries are in the set Q of queries to process.
- Step 4: *VSM Data Set Retrieval.* For each query in Q, the selected keywords are used to retrieve data sets based on the Vector Space Model: in this way, the set of possibly relevant data sets is obtained. The *Keyword Relevance Measure krm* is the cosine similarity measure.
- Step 5: *Schema Fitting.* The full set of field names in each query in Q is compared with the schema of each selected data set: the *Schema Fitting Degree* is higher for data sets whose schema better fits the query.
  The general *Relevance Measure rm* for a data set is a combination of *krm* and sfd: only data sets with *rm* greater than or equal to a minimum threshold th are selected (relevant data sets).
- Step 6: *Instance Filtering.* Instances of relevant data sets are processed in order to filter out and keep only the items (rows or JSON objects) that satisfy the selection condition.

As a result, at the end of the process the user is provided with the set of items (CSV records or JSON objects) that satisfy the initial query and the rewritten ones.

In this paper, we do not focus on the capability of the technique to retrieve the desired items, which is discussed in (Pelucchi et al., 2017) in terms of recall and precision. Here, we discuss the challenge of implementing such a technique by exploiting the *Map-Reduce* approach.

3 MAP-REDUCE IN THE HAMMER PROTOTYPE

In this section, we present the *Hammer* prototype, that is the testbed for the technique shortly presented in Section 2.2 and deeply presented in (Pelucchi et al., 2017). The final perspective application of this technique is to build a centralized query engine for many open data portals, in order to further simplify the work of open data users. This perspective is simple and attracting. However, open data corpora are becoming larger and larger: we can say that we are going towards the *Big Open Data World*. The consequence is that it is not possible to deal with queries from scratch, but it is necessary to perform preliminary computation in order to get acceptable execution time during query processing. For this reason, three main services are provided by the *Hammer* prototype:

1. *Open Data Portal Crawling*: a portal is queried to get the list of open data sets they publish, getting their schema and meta-data, so that a local catalog of data sets is locally built;
2. *Indexing*: the inverted index to perform step 4 of the technique (VSM Dataset Retrieval) is built;
3. *Querying*: queries are actually executed and evaluated on the actual content of data sets.

However, the possible large number of data sets in a corpus could make the process highly inefficient, if executed on one single computer. Also because, in our vision, the *Hammer* prototype should be able to deal with several open data portals, previously configured. The final goal of our project is to build a centralized query engine for open data portals.

Therefore, the challenge of our work is to make possible to parallelize time consuming operations, on the basis of the *Map-Reduce* approach. In fact, this approach could easily scale, by increasing the number of nodes involved in the execution of the process. Consequently, it could be beneficial for implementing our query technique.

Having in mind the perspective of *Big Open Data*, in the *Hammer* prototype we used modern technology for Big Data management. In particular, we adopted *MongoDB* as DBMS to manage local storage, in that it is schema less and allows us to store heterogeneous JSON collections. Another tool we exploited is *Apache Hadoop*, that supports the Map-Reduce technique, in order to parallelize long computational tasks. In the following, we present the two processing phases and the devised components in details.

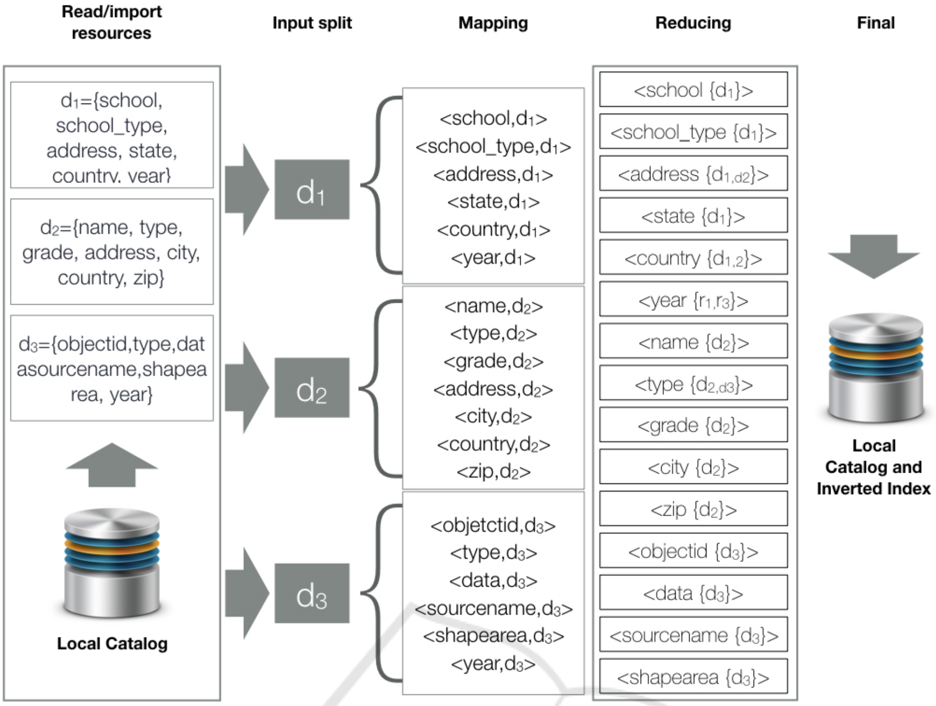

Figure 3: Building the Inverted Index for three open data sets by applying the *Map-Reduce* approach.

3.1 Crawler

This component gets information about the data sets contained in the corpus published by an open data portal and builds the local catalog. Figure 2 is useful to clarify.

The crawler has been designed to connect with several platforms usually adopted to publish open data portals. Depending on the platform, the crawler relies on a specific connector. Several connectors were built, but, in particular, the most useful ones are the following: the *CKAN Connector*, for the standard platform named *Comprehensive Knowledge Archive Network*; the *Socrata Connector*, for *Socrata Open Data Portals*. Since these platforms are the ones mostly used to build open data publishing services, our prototype is ready to connect with a large variety of different open data services. Nevertheless, it is easy to extend the crawler with new connectors.

3.2 Indexer

The *Indexer* builds the Inverted Index *IN*. This is exploited by the technique presented in Section 2.2, to perform the *VSM Dataset Retrieval*. In the *Hammer* prototype, the *Indexer* component extracts terms and labels from schema and meta-data of each open data set in the corpus *C*. The indexer creates an inverted index of terms and labels, i.e., a data structure that implements a function

$$TermWeighForDataSets : t \rightarrow \{ods\_id, w\} \quad (1)$$

that, given a term t, returns a set of pairs denoting the *ods_id* of the data set and the weight w (number between 0 and 1)

Within *Indexer* we adopted the *Map-Reduce* technique; for each data set and for each label in the schema and in the meta-data, the *map* primitive generates a pair $<t, ods\_id>$. Then, the *reduce* primitive transforms the set of pairs into a nested structure

$$IN = \{ <t, ods\_list: \{ ods\_id_1, \ldots, ods\_id_n \}> \}$$

where *ods_list* is the list of *ids* of data sets for which t is a label in schema or in meta-data. This structure is the inverted index *IN*.

Finally, the inverted index *IN* is stored into the local database.

The advantage of using the *Map-Reduce* approach is that it natively implements *map* and *reduce* primitives in a distributed environment, thus permitting to distribute the computation among several servers.

Figure 3 shows how the process works. Suppose three open data sets labeled d_1, d_2 and d_3 are in the corpus C. Their schema appear in the upper left side of the figure. The *map* primitive generates the set of

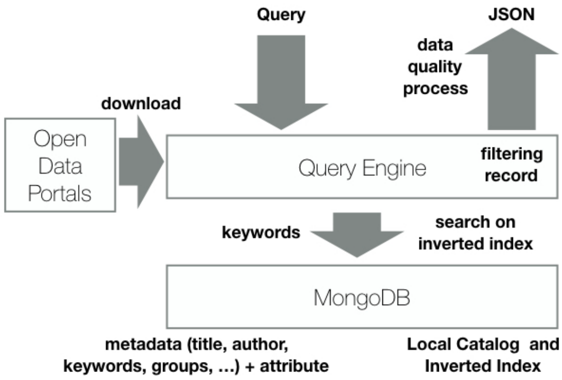

Figure 4: The Retrieval Process.

tuples that describe each data sets. These tuples are reported in the central part of the figure. Recall that each tuple associates a term, in this case a property name, with the data set identifier. Finally, the *reduce* primitive fuses all tuples with the same label: we obtain the nested structures reported on the right hand side of the figure, i.e., the inverted index *IN*.

3.3 Query Engine

At query time, the *Query Engine* exploits the inverted index *IN* to retrieve the list of data sets relevant for the query (it is better to say: for each neighbor query $nq \in Q$).

Figure 4 depicts how the query engine works: it receives the query and generates the set *Q* of queries to process. Then, by exploiting the catalog and the inverted index *IN*, it determines relevant data sets and then provides (as output) the items within those data sets that satisfy at least one selection condition *nq.sc* of a query $nq \in Q$. Thus, the *Query Engine* must implement all the steps of the retrieval technique introduced in Section 2.2. However, steps from 1 to 3, i.e., from query assessment to query expansion, are not suitable for parallelization with *Map-Reduce*, because they strongly rely on the computation of the *Jaro-Winkler* string similarity measure (Cohen, 1998). These steps are implemented by a single-thread process.

In contrast, step 4 (*VSM Dataset Retrieval*), Step 5 and Step 6 can be implemented by exploiting again the *Map-Reduce* approach.

- Step *VSM Dataset Retrieval* and Step *Schema Fitting* determine, for each neighbour query $nq \in Q$, the list of relevant data sets, i.e., those data sets which items of interest for the user are likely to be extracted from. To this end, a relevance measure *rm* for a data set *ods* w.r.t. a neighbour query *nq* is defined as:

  $$rm(ods,\ nq) = (1-\alpha) \times krm(ods,\ nq) + \alpha \times sfd(ods,\ nq)$$

  krm(ods, nq) is the *Keyword-based Relevance Measure*, i.e., the relevance of a data set on the basis of relevant keywords in query *nq*; *sfd* is the *Schema Fitting Degree*, i.e., a measure between 0 and 1 that evaluates if the schema of the data set *ods* is suitable for the query *nq*. $\alpha \in [0,1]$ is a parameter that permits to balance the contribution of both *krm* and *sfd* to the overall relevance measure. The outcome of this step is the set $RD = \{ods_i\}$ of relevant data sets.
- For each $ods_i \in RD$, $Instance(rd_i)$ is collected from the open data portal and temporarily stored into the local storage.
- The result set *RS* of relevant items is obtained by evaluating the selection condition provided with the neighbour query *nq* for which the data set was retrieved. Again, the *Map-Reduce* approach is a good solution to deal with heterogeneous items in a parallel way.

The keyword-based relevance measure $krm(ods, nq)$ is computed by adopting the vector space model approach.

Consider a neighbour query $nq \in Q$ (where *Q* is the set of queries to process): its set of keywords *K(nq)* can be represented as a vector $v = <k_1, \ldots, k_n>$. The vector $\overline{w}(nq) = <\overline{w}_1, \ldots, \overline{w}_n>$ represents the weight $\overline{w}_i$ of each term k_i in the set of keywords *K(nq)*; this weight is conventionally set to 1, i.e., $\overline{w}_i = 1$ (but different settings could be defined, depending on the role of each term in the query).

Given a data set identifier *ods_id*, vector

$$W(ods\_id) = <w_1, \ldots, w_n> \}$$

is the vector of weights of each keyword k_i for data set identified by *ods_id*, as obtained by evaluating function *TermWehighForDataSetrs* through the inverted index *IN*.

The *Keyword-based Relevance Measure krm(ods, nq)* for the data set *ods* (identified by *ods_id*) w.r.t. query *nq* is the cosine of the angle between vectors $\overline{w}(nq)$ and *W(ods_id)*. The maximum value of rm_i is 1, obtained when the data set is associated with all the keywords, while values less than 1 but greater than 0 denote partial matching. A minimum threshold *th* is used to discard less relevant data sets and keep only those such that $rm(ods,\ nq) \geq th$.

For each relevant data set, the *Query Engine* downloads its instance from the Open Data Portal. Instances are stored into a collection within the

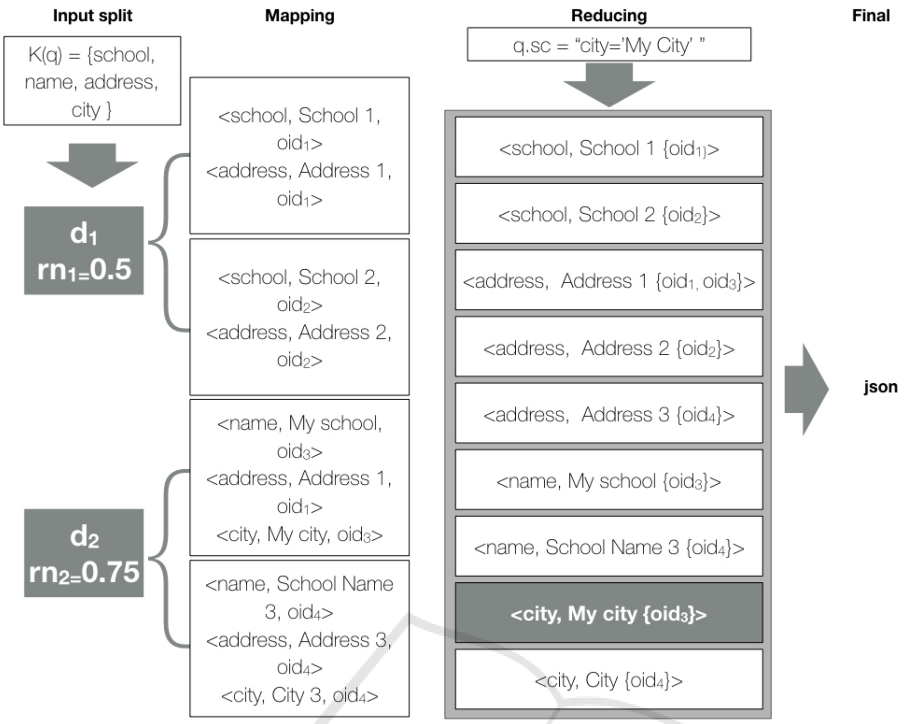

Figure 5: Retrieving relevant items by applying *Map-Reduce*.

NoSql MongoDB database. This way, we exploit the schema-less feature of *MongoDB*, since the actual schema of each retrieved data set can be very different from each other. Instead, *MongoDB* is very useful to manage heterogeneity.

Map-Reduce is exploited to select items as well. First of all, the *Query Engine* assigns a unique identifier to each item within the downloaded data sets. In order to deal with heterogeneity of these objects and easily select them, for each item and each field (property) in the item, the *map* primitive generates a triple $<oid,\ field\_value>$. Then, the *reduce* primitive transforms the set of triples into a nested structure

$$\{ <field,\ value,\ oid\_list\text{:}\ \{ oid_1,\ \ldots,\ od_n \}>$$

in order to aggregate all items having the same value for a given field.

To illustrate, consider a query q. Figure 5 illustrates the process. Based on the set of keywords *K(q)*, we match two data sets d_1 and d_2 that have sufficient relevance w.r.t. the minimum threshold *th=0.3* (in Figure 5, $rm_1 = rm(d_1, nq) = 0.5$ and $rm_2 = rm(d_2, nq) = 0.75$)). Their instances are downloaded and items are processed by the *map* primitive (see Figure 5). Then, the *reduce* primitive aggregates triples based on the equality of field name and value. Finally, the items that match the selection condition (in Figure 5, the gray item on the right-hand side) are selected. In the example, only object with oid_3 is selected.

4 EXPERIMENTS

We implemented our technique within the *Hammer* prototype, so as to evaluate the effectiveness of the technique and study performance. In this section, we discuss execution times, to evaluate the effectiveness of the *Map-Reduce* approach. Instead, the effectiveness of the retrieval technique in terms of recall and precision is reported in (Pelucchi et al., 2017).

4.1 Testbed

We assembled a testbed for evaluating the *Hammer* prototype. It is a cluster of *Hadoop* nodes. We tested it in two different environments: the first one was on local PCs and it was used for development and tuning; the second environment was configured on *Google Cloud Platform* and it was used to evaluate performance in an industrial settings (i.e., with networking connections having low latency and high throughput during the download of datasets). Table 1 summarize the different configurations of our tests.

Notice the three different configurations of the *Hadoop* ecosystem: with 1 single node for storage and computation; with 1 node for storage and 3 for

Table 1: Configurations of the Testbed.

Configuration	Storage	Cluster
1. Single Node (Development CLuster)	1 Node (1 virtual CPU and 3 GB of memory)	1 Node (1 virtual CPU and 3 GB of memory)
2. Multiple Nodes (Development Cluster)	1 Node (1 virtual CPU and 3 GB of memory)	3 Node (1 virtual CPU and 3 GB of memory)
3. Multiple Nodes (Google Cloud Platform)	2 Node (1 virtual CPU and 3.75 GB of memory)	3 Node (1 virtual CPU and 3.75 GB of memory)

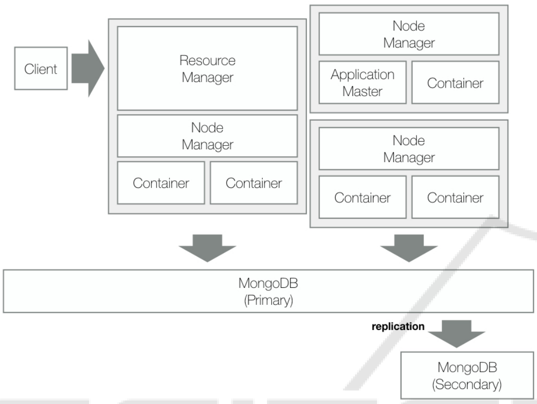

Figure 6: Architecture of the Prototype.

computation; (on Google Cloud Platform) with 5 nodes, 3 devoted to perform crawling, inverted index creation and query execution, 2 devoted to storage (one node for the main instance of MongoDB and one node hosting a replica). Figure 6 shows the architecture.

Experiments were performed on a set of 2301 open data sets published by the *Africa Open Data* portal[1]. These open data sets are prepared in a non homogeneous way, as far as field names and conventions are concerned: *Africa Open Data* aggregates data sets from a lot of different organizations. This fact makes the search more difficult than with an homogeneous corpus: data sets are different by format, type and argument.

4.2 Crawling and Indexing

First, we tested crawling and indexing. The results are reported in Table 2, for the three different configurations.

First of all, notice the execution times for crawling. A dramatic improvement is obtained by passing from configuration 1 (one node) to configuration 2 (4 nodes), even in the development environment. Then, configuration 3 (on Google Cloud Platforms) further reduces execution times. The overall gain is from more than 2 hours to 40 minutes.

Table 2: Response times (in sec) for crawling and indexing.

Configuration	Crawling	Indexing
1	7524	972
2	3276	792
3	2448	774

As far as indexing is concerned, increasing the number of nodes is not particularly effective. In fact, passing from configuration 1 (one node) to configuration 2 (4 nodes) we save 180 secs, i.e., the 18%. In contrast, moving to configuration 3 (Google Cloud Platform), just a few seconds are saved. This is due to the *reduce* primitives, that must aggregate sparse tuples coming from the *map* primitive.

4.3 Querying

We run 6 different queries. They are reported in Figure 7. Hereafter, we describe them to help the reader to understand what kind of queries it is possible to perform.

- Information about nutrition in Kenya. In query q_1, we look for analytical information (number of cases of *underweight*, *stunting* and *wasting*) about malnutrition in the regions of Monbasa, Nairobi and Turkana in Kenya.
- Civil unions statistics. In query q_2, we look for information about civil unions since 2012, where the age of both spouses is at least 35. The attribute extract are `year`, `month`, `spouse1age` and `spouse2age`.
- Availability of water. Query q_3 retrieves information about availability of water, a serious problem in Africa; it looks for distribution points that are functional (note the condition,

[1] https://africaopendata.org/

Table 3: Time of execution in seconds for each steps of the querying process.

Configuration 1

	Query Assessment	Keyword Selection	Neighbour Queries	VSM Data Set Retrieval	Schema Fitting	Instance Filtering	Total time
q_1	107	10	529	1535	7740	6	9927
q_2	114	11	257	576	8453	2410	11821
q_3	82	10	179	746	8156	5615	14788
q_4	61	14	313	609	8969	2414	12380
q_5	60	19	35	93	455	218	880
q_6	45	9	28	29	262	37	410

Configuration 2

	Query Assessment	Keyword Selection	Neighbour Queries	VSM Data Set Retrieval	Schema Fitting	Instance Filtering	Total time
q_1	105	12	534	990	4967	3	6611
q_2	117	11	257	456	5624	1099	7564
q_3	91	13	165	707	4591	2577	8144
q_4	64	12	321	439	4086	1475	6397
q_5	61	21	32	47	203	95	459
q_6	34	11	37	20	139	21	262

Configuration 3

	Query Assessment	Keyword Selection	Neighbour Queries	VSM Data Set Retrieval	Schema Fitting	Instance Filtering	Total time
q_1	98	9	433	895	2345	2	3782
q_2	99	7	212	432	2478	725	3953
q_3	87	9	134	654	1572	759	3215
q_4	59	7	291	321	804	1077	2559
q_5	48	11	21	23	159	48	310
q_6	21	6	23	15	82	12	159

where we consider two possible values for field `functional-status` (`functional` and `yes`).

- Public schools metrics. Query q_4 looks for information about the number of teachers in public schools, for each county. The query returns `county`, `schooltype` and `noofteachers`.
- Water consumptions. In query q_5, we look for data about per capita water consumptions at the date of December, 31 2013 (the date is written as `"2013-12-31t00:00:00"`).
- National Export statistics of Orchids. Query q_6 retrieves data about orchids export in terms of weight (field `kg`) and `money`.

Tables 3 summarizes execution times in secs, reporting the details for each step described in Section 2.2. The table is divided in three groups, one for each configuration.

The *Query Engine* was configured with the following minimum thresholds: *th*= 0.3 is the minimum threshold for data set relevance measure, *th_sim=0.8* is the minimum threshold for *Jaro-Winkler* (Cohen, 1998) string similarity metric (to build neighbour queries) with up to three alternative terms for each substituted term. This setting is the one (see (Pelucchi et al., 2017)) that obtained the best performance in terms of recall and precision, but it is also the one that retrieves the highest number of data sets. This is why we adopted this setting to test execution times.

The reader can see that, at the current stage of development, execution times are very long, in some cases several hours (when thousands of neighbour queries must be processed). In particular, a not efficient step is *Schema Fitting*, because the schema of a data set retrieved by the *VSM Dataset Retrieval* must be fitted against all the neighbour queries. Currently, this step is not optimized: we plan to do that in the near future.[2]

Instead, the execution time of step 6 *Instance Filtering* strongly depends on the actual size of downloaded data set instances.

[2]Number of neighbour queries generated for each original query: 243 for q_1, 562 for q_2, 81 for q_3, 91 for q_4, 263 for q_5 and 9 for q_6.

q_1:<dn=`nutrition`,
  P= {`county, underweight, stuting, wasting`},
  sc= (`county = "Monbasa" OR county = "Nairobi" OR county = "Turkana"`) >
q_2 :<dn=`civilunions`,
  P= {`year, month, spouse1age, spouse2age`},
  sc= (`year = 2012 AND (spouse1age >= 35 OR spouse2age >= 35`) >
q_3 :<dn=`wateravailability`,
  P= {`district, location, position, wateravailability`} ,
  sc= (`functional-status = "functional" OR functional-status = "yes"`) >
q_4 :<dn=`teachers`,
  P= {`county, schooltype, noofteachers`} ,
  sc= (`schooltype="public"`) >
q_5 :<dn=`waterconsumption`,
  P= {`city, date, description, consumption_per_capita`} ,
  sc= (`date = "2013-12-31t00:00:00"`) >
q_6 :<dn=`national-export`,
  P= {`commodity, kg, money`} ,
  sc= (`commodity = "Orchids"`) >

Figure 7: Queries for the Experimental Evaluation.

Anyway, as far as the effectiveness of *Map-Reduce*, the results are promising: both step 4 *VSM Dataset Retrieval* and Step 6 *Instance Filtering* strongly reduce their execution time. Thus, we can expect that a configuration with a larger number of computing nodes will better parallelize the execution and will obtain faster response times. Finally, the reader can notice that the configuration on *Google Cloud Platform* is always the oe with better performance: a better virtual environment and an excellent network infrastructure speed up the prototype.

5 RELATED WORKS

The world of Open Data is becoming more and more important for many human activities. Just to cite some areas that can get benefits, we cite Neuro-Sciences (Wiener et al., 2016), prediction of tourists' response (Pantano et al., 2016) and improvements to digital cartography (Davies et al., 2017). These works are proving the concept of Open Data Value Chain: the wide adoption of Open Data by researchers and analysts shows that the effort of public administrations is motivated and must be carried on.

We now focus on Open Data Management research. This area is young and in progress. In particular, at the best of our knowledge, our approach is novel and no similar systems are available, at the moment. Anyway, some works have been done.

An interesting paper is (Braunschweig et al., 2012), where the authors observe characteristics of fifty Open Data repositories. As a result, they sketch our vision of a central search engine.

In (Liu et al., 2006), the authors note that there is a growing number of applications that require access to both structured and unstructured data. Such collections of data have been referred to as dataspaces, and *Dataspace Support Platforms* (DSSP) were proposed to offer several services over dataspaces. One of the key services of a DSSP is seamless querying on the data. The *Hammer* prototype can be seen as DSSP of Open Data, while (Liu et al., 2006) proposes a DSSP of web pages.

In (Kononenko et al., 2014), the authors reported their experience in using *Elasticsearch* (distributed full-text search engine) to resolve the problem of heterogeneity of Open data sets from federated sources. Lucene-based frameworks, like *Elasticsearch* and *Apache Solr* (see (Nagi, 2015)), are alternatives to the *Hammer* prototype, but they usually don't retrieve single items satisfying a selection condition.

In (Schwarte et al., 2011), the authors describe their approach and their idea to build a pool of federated Open Data Corpora with *SPARQL* as query language. However, we considered *SPARQL* only for people that are highly skilled in computer science: our query technique is very easy to use and it is designed for analysts with medium-level or low-level skills in computer science.

The benefit of *Map-Reduce* approach are described in (Dean and Ghemawat, 2010). In few words, *Map-Reduce* automatically parallelizes and executes the program on a large cluster of commodity machines. In (Vavilapalli et al., 2013), the authors present *YARN Yet Another Resource Negotiator* and they provide experimental evidence demonstrating the improvements of use YARN on production environments. YARN is the basic component within *Hadoop* that handles the actual execution of the *Map-Reduce* tasks.

6 CONCLUSION

In this paper, we present the current state of development of the *Hammer* prototype, a testbed for a novel technique to query corpora of Open Data sets. In particular, in this work we faced the challenge of parallelizing many parts of the prototype by applying the *Map-Reduce* approach, in order to evaluate the feasibility of this *Big Data* approach in the area of *Big Open Data*. Specifically, the *Hammer* prototype implements the retrieval technique presented in (Pelucchi et al., 2017). We demonstrated that the adoption of modern standard technology specifically designed for big data management, such as *Apache Hadoop* and *MongoDB*, can be effective in this context, in particular increasing the number of nodes in the *Apache Hadoop* ecosystem, even though the retrieval technique produces a large number of rewritten queries (neighbour queries) and several parts of the prototype are currently not optimized.

As far as the effectiveness of the query technique is concerned, i.e., evaluation of the capability to retrieve what users want, we refer to our previous work (Pelucchi et al., 2017), in which the technique has been extensively introduced and evaluated on a corpus of open data sets. In that paper we showed that the technique is actually effective, in particular in comparison with a tool like *Apache Solr*, that is a stand-alone search engine. We discovered that, although *Apache Solr* behaves quite well, our technique is capable of better focusing on data sets of interest; furthermore, it extracts only items of interest (while *Apache Solr* does not in a classic configuration).

In the future work, we will optimize the implementation of many components of the *Hammer* prototype, in order to get near real time response times. In particular, we plan to replace the native *Hadoop* implementation with *Spark* on a *Hadoop Cluster* to obtain dramatic improvement of performance (according to (Zaharia et al., 2010), *Spark* is 10x faster then *Hadoop*).

Finally, we will extend queries to provide complex features such as join and spatial joins ((Bordogna and Psaila, 2004)) of retrieved data sets. In particular, we are considering, as a starting point, the concept of query disambiguation, in order to improve the generation of neighbour queries; a work we are considering as a starting point is (Bordogna et al., 2012). Furthermore, we think that a post processing of results is necessary, in particular when thousands of items are retrieved. We think that useful operators could be defined, similar to those introduced in (Bordogna et al., 2008).

Similarly, the adoption of *NoSQL* databases for persistent storage of retrieved results could be useful, since the *Hammer* prototype provides collections of heterogeneous JSON objects, possibly georeferenced. A good idea could be to integrate the concept of blind querying and the *Hammer* engine as part of the *J-CO-QL* query language (Bordogna et al., 2017), which is able to query heterogeneous collections of possibly geo-tagged JSON objects, providing high-level operators which natively deal with spatial representation and properties.

REFERENCES

Bordogna, G., Campi, A., Psaila, G., and Ronchi, S. (2008). A language for manipulating clustered web documents results. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 23–32. ACM.

Bordogna, G., Campi, A., Psaila, G., and Ronchi, S. (2012). Disambiguated query suggestions and personalized content-similarity and novelty ranking of clustered results to optimize web searches. *Information Processing & Management*, 48(3):419–437.

Bordogna, G., Capelli, S., and Psaila, G. (2017). A big geo data query framework to correlate open data with social network geotagged posts. In *International Conference on Geographic Information Science*, pages 185–203. Springer, Cham.

Bordogna, G. and Psaila, G. (2004). Fuzzy-spatial sql. In *International Conference on Flexible Query Answering Systems*, pages 307–319. Springer.

Braunschweig, K., Eberius, J., Thiele, M., and Lehner, W. (2012). The state of open data. In *WWW2012*.

Carrara, W., Chan, W. S., Fischer, S., and van Steenbergen, E. (2015). *Creating Value through Open Data*. European Union.

Cohen, W. W. (1998). Integration of heterogeneous databases without common domains using queries based on textual similarity. In *ACM SIGMOD Record*, volume 27, pages 201–212. ACM.

Cukier, K. (2010). *Data, data everywhere: A special report on managing information*. Economist Newspaper.

Davies, T. G., Rahman, I. A., Lautenschlager, S., Cunningham, J. A., Asher, R. J., Barrett, P. M., Bates, K. T., Bengtson, S., Benson, R. B., Boyer, D. M., et al. (2017). Open data and digital morphology. In *Proc. R. Soc. B*, volume 284, page 20170194. The Royal Society.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Dean, J. and Ghemawat, S. (2010). Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77.

Kononenko, O., Baysal, O., Holmes, R., , and Godfrey, M. (2014). Mining modern repositories with elasticsearch. In *MSR. June 29-30 2014, Hyderabad, India*.

Liu, J., Dong, X., and Halevy, A. Y. (2006). Answering structured queries on unstructured data. In *WebDB. 2006, Chicago, Illinois, USA*, volume 6, pages 25–30. Citeseer.

Manning, C. D., Raghavan, P., Schütze, H., et al. (2008). *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge.

Nagi, K. (2015). Bringing search engines to the cloud using open source components. In *Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K), 2015 7th International Joint Conference on*, volume 1, pages 116–126. IEEE.

Pantano, E., Priporas, C.-V., and Stylos, N. (2016). You will like it!using open data to predict tourists responses to a tourist attraction. *Tourism Management*.

Pelucchi, M., Psaila, G., and Toccu, M. (2017). Building a query engine for a corpus of open data. In *Proceedings of the 13th International Conference on Web Information Systems and Technologies - Volume 1: WEBIST,*, pages 126–136. INSTICC, ScitePress.

Schwarte, A., Haase, P., Hose, K., Schenkel, R., and Schmidt, M. (2011). Fedx: a federation layer for distributed query processing on linked open data. In *Extended Semantic Web Conference*, pages 481–486. Springer.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. (2013). Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM.

Wiener, M., Sommer, F. T., Ives, Z. G., Poldrack, R. A., and Litt, B. (2016). Enabling an open data ecosystem for the neurosciences. *Neuron*, 92(3):617–621.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95.