Distrib Parallel Databases
DOI 10.1007/s10619-013-7133-7

Scalable entity-based summarization of web search results using MapReduce

Ioannis Kitsos · Kostas Magoutis · Yannis Tzitzikas

Abstract Although Web Search Engines index and provide access to huge amounts of documents, user queries typically return only a linear list of hits. While this is often satisfactory for focalized search, it does not provide an exploration or deeper analysis of the results. One way to achieve advanced exploration facilities exploiting the availability of structured (and semantic) data in Web search, is to enrich it with *entity mining* over the *full contents* of the search results. Such services provide the users with an initial overview of the information space, allowing them to gradually restrict it until locating the desired hits, even if they are low ranked. This is especially important in areas of professional search such as medical search, patent search, etc. In this paper we consider a general scenario of providing such services as *meta-services* (that is, layered over systems that support keywords search) without a-priori indexing of the underlying document collection(s). To make such services feasible for large amounts of data we use the *MapReduce* distributed computation model on a Cloud infrastructure (Amazon EC2). Specifically, we show how the required computational tasks can be factorized and expressed as MapReduce functions. A key contribution of our work is a thorough evaluation of platform configuration and tuning, an aspect that is often disregarded and inadequately addressed in prior work, but crucial for

Communicated by Feifei Li and Suman Nath.

I. Kitsos · K. Magoutis · Y. Tzitzikas (✉)
Institute of Computer Science, FORTH-ICS, Crete, Greece
e-mail: tzitzik@ics.forth.gr

I. Kitsos
e-mail: kitsos@ics.forth.gr

K. Magoutis
e-mail: magoutis@ics.forth.gr

I. Kitsos · K. Magoutis · Y. Tzitzikas
Computer Science Department, University of Crete, Heraklion, Greece

Published online: 14 August 2013

the efficient utilization of resources. Finally we report experimental results about the achieved speedup in various settings.

Keywords Text data analytics through summaries and synopses · Interactive data analysis through queryable summaries and indices · Information retrieval and named entity mining · MapReduce · Cloud computing

1 Introduction

Web searching is probably the most frequent user task in the web, during which users typically get back a linear list of hits. Several user studies [34, 48, 59] have shown that end-users see significant added value in services that analyze and group the results (e.g. in categories, clusters, etc.) of keyword-based search engines. Such services help them to easier locate the desired hits by initially providing them with an overview of the information space, which can be further explored gradually in a faceted search-like interaction scheme [50]. Our goal in this paper is to construct at query time, a *browsable summary* of the full contents of the search results using entity mining and external sources. To make such services feasible for large amounts of data we parallelize the entity-mining process by exploiting the *Map-Reduce* [17] distributed computation model. Our overall methodology falls in the general category of *big data analytics*.

The provision of such summaries is very important in *recall-oriented search*, which aims at satisfying information needs that require inspecting a set of resources (e.g. decide which car or vacation package to buy). In contrast, *precision-oriented* information needs can be satisfied by a single resource (e.g. find information about a particular smart phone). According to [58], the majority of information needs have exploratory nature, they are recall-oriented (e.g. bibliographic survey writing, medical information seeking, car buying), and aim at decision making (based on one or more criteria). According to Web search query analysis results reported in [7], 80 % of the submitted queries correspond to recall-oriented needs.

A recent approach that falls in this category attempting to bridge the gap between *documents* and *structured information*, is described in [19]. That work proposes a method for enriching the classical interaction scheme of search systems (keyword search over Web pages), with *(Named) Entity Mining* (for short *NEM*), over the *textual snippets* of the search results at *query time*, i.e., without any pre-processing. In both [19] and in this paper we consider the scenario where these services are provided as *meta-services*, i.e. on top of systems that support keywords search. In particular, Fig. 1 illustrates the process that we consider. The initial query is forwarded to the underlying search system(s)[1] and the results are retrieved; then the URIs of the hits are used for downloading the full contents of the hits, over which entity mining is performed. The named entities of interest can be specified by external sources (e.g. by querying SPARQL endpoints). Finally the identified entities are enriched with semantic descriptions derived by querying external SPARQL endpoints. The user can then

[1] In our implementation any system that supports OpenSearch [14] can straightforwardly be used.

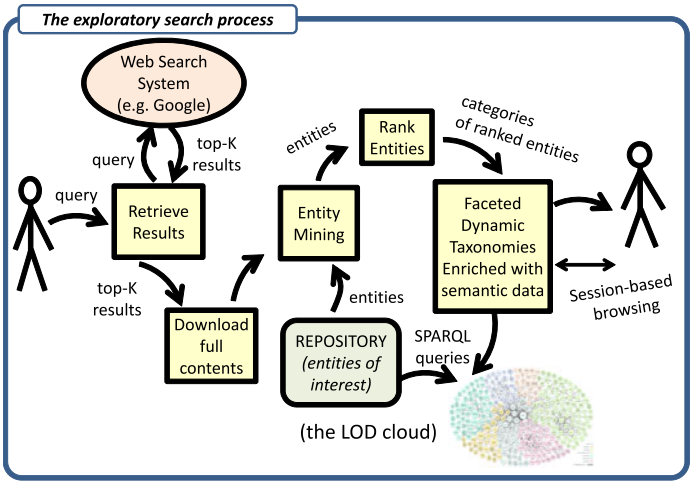

Fig. 1 The exploratory search process

gradually restrict the derived answer by clicking on the entities. The above process is fully configurable and dynamic: the user can set up the desired underlying search system, the desired kind of entities and the desired entity list. For example, in one instance the user may request the analysis of results coming from Google, where the entities of interest are person names, companies, and locations. In another instance however, the user may request the analysis of hits coming from bibliographic sources about the marine domain, where the entities of interest are water areas, species, industries, and names of politicians. Figure 2 shows an indicative screendump of the results from such a meta-service. The right bar contains various frames for different entity categories (e.g. `Person`, `Organization`, etc.) and in each of them the identified entities are shown along with their number of occurrences. By clicking on an entity the user may restrict the answer to those hits that contain that entity. The restricted answer can be further restricted by clicking on another entity, and so on. By clicking the icon to the right of each entity the system shows a popup window containing semantic information fetched and assembled from the Linked-Open Data (LOD) cloud. Figure 19 (in the Appendix) shows two screendumps from search engines focusing on a specific segments of online content (also known as *vertical search* applications) specializing on *marine entity* and *patent* search.

Applying *NEM* over the textual snippets of the top hits, where the snippet of a hit consists of 10–20 words, has been shown to produce results in real time [19]. However, mining over snippets does not provide any guarantee about completeness of the returned entities; moreover, not all search systems return textual snippets. Extending *NEM* (without pre-processing) to the *full contents* of the top-hits is demanding in a number of ways: First, it requires transferring (downloading) large amounts of data. Second, it is resource-intensive both computationally (scanning and processing the downloaded contents) and in terms of memory consumed. Performing entity mining on several thousand hits (the scale of queries considered in this paper) using the sequential *NEM* procedure exceeds the capabilities of any single compute node (e.g.,

Fig. 2 An application for general-purpose searching

a Cloud VM) eventually leading to a crash. Even for just a few hundreds of hits, an *NEM* job on a single node may crash or take several hours to complete.

To tackle these challenges, in this paper we propose two ways for distributing the entity-mining process onto MapReduce tasks and examine ways to efficiently execute those tasks on the Apache Hadoop MapReduce platform on Amazon EC2. We provide the required algorithms, we discuss their requirements (e.g. in terms of exchanged data), and establish analogies with other tasks (e.g. inverted index construction). While significant attention must be paid to the specification of the MapReduce algorithms that address the problem at hand, the complexity of appropriately configuring and tuning the platform for efficient utilization of resources is often disregarded and inadequately addressed in prior work. To this end we analyze the factors that affect performance and how to tune them for optimal resource utilization. In fact, one of our key contributions is to thoroughly evaluate the parameter space of the underlying platform and to explain how to best tune it for optimal execution. Finally we report extensive and comparative experimental results. We believe that the methods and results of our work are applicable to the parallelization and efficient execution of other related applications.

The rest of this paper is organized as follows. In Sect. 2 we discuss the motivation, context, and related work. In Sect. 3 we describe the centralized task, and in Sect. 4 we show how it can be logically decomposed and expressed as MapReduce functions. In Sect. 5 we detail the implementation of the MapReduce tasks and in Sect. 6 we report experimental results. Finally, in Sect. 7 we provide our conclusions and discuss future work.

2 Background and related work

2.1 Analysis of search results

2.1.1 Why is it useful? Evidence from user studies

The *analysis of search results* is a useful feature as it has been shown by several user studies. For instance, the results in [31] show that categorizing the search results improves the search speed and increases the accuracy of the selected results. A user study [30] shows that categories are successfully used as part of users' search habits. Specifically, users are able to access results that are located far in the rank order list and formulate simpler queries in order to find the needed results. In addition, the categories are beneficial when more than one result is needed like in an *exploratory* or *undirected search* task. According to [34] and [59], recall-oriented information can play an important role not only in understanding an information space, but also in helping users select promising sub-topics for further exploration.

Recognizing entities and grouping hits with respect to entities is not only useful to public web search, but is also particularly useful in *professional search* that is, search in the workplace, e.g. in industrial research and development [33]. The user study [48] indicated that categorizing dynamically the results of a search process in a *medical* search system provides an organization of the results that is clearer, easier to use, more precise, and in general more helpful than simple relevance ranking. As another example, in professional *patent search*, in many cases one has to look beyond keywords to find and analyze patents based on a more sophisticated understanding of the patent's content and meaning [29]. We should also stress that professional search sometimes requires a long time. For instance, in the domain of patent search, the persons working in patent offices spend days for a particular patent search request. The same happens in bibliographic and medical search.

Technologies such as entity identification and analysis could become a significant aid to such searches and can be seen, together with other text analysis technologies, as becoming the cutting edge of information retrieval science [6]. Analogous results have been reported for search over collections of *structured* artifacts, e.g. ontologies. For instance, [1] showed that making explicit the relationships between ontologies and using them to structure (or categorize) the results of a Semantic Web Search Engine led to a more efficient ontology search process.

Finally, the usefulness of the various analysis services (over search results) is subject of current research, e.g. [11] comparatively evaluates *clustering* versus *diversification* services.

2.1.2 Past work on entity mining over search results

Recent work [19] proposed the enrichment of the classical web searching with entity mining performed at query time as a flexible way to integrate the results of keyword search with structured knowledge (e.g. expressed using RDF/S). The results of entity mining (entities grouped by their categories) complement the query answers with useful information for the user which can be further exploited in a faceted search-like interaction scheme [50]. In addition, [20] describes an application of the same

approach in the domain of *patent search*, where the underlying search systems are patent search systems and entity mining is applied over some the text-valued metadata of the patents.

As regards efficiency, [19] showed that the application of entity mining over the (small in size) textual snippets of the top-hits of the answers, can be performed in real-time. However, mining over the snippets returns less entities than mining over the full contents of the hits, and comparative results for these two scenarios were reported in [19]. Specifically, mining over the contents returns around 20 times more entities than mining over just the snippets. This happens because a snippet is actually an excerpt of the document containing the maximum possible number of words of the submitted query (usually 10–20). Furthermore, the Jaccard Similarity index (a statistic used for comparing the similarity and diversity of sample sets [26]) between the top-10 entities from mining over the snippets vs. mining over full contents, is 0 % (i.e., there are no common entities between the two entity sets) for about 60 % of the queries. The main difference of [19] to our current work is that we apply entity mining over the full contents and the required computation is distributed to various nodes.

Another work that uses similar functionality is *Google's Knowledge Graph* (as announced in May 2012), which tries to understand the user's query and to present (on the fly) a semantic description of what the user is probably searching, actually information about *one* entity. In comparison to our approach, Google's Knowledge Graph is *not* appropriate for recall-oriented search since it shows only one entity instead of identifying and showing entities in the search results. Furthermore if the user's query is not a known entity, the user does *not* get any entity or semantic description.

We should clarify that the various *Entity Search Engines* (e.g. [13, 23, 56]) are not directly related to our work, since they aim at providing the user only with *entities* and *relationships* between these entities (not links to web pages); instead we focus on enriching classical web searching with entity mining. In addition, works on *query expansion* using lexical resources (thesauri, ontologies, etc.), or other methods that exploit named entities for improving search (e.g. [10, 18]), are out of the scope of this work, since we focus on (meta-)services that can be applied on top of search results.

From an *information integration* perspective, we can say that entity names are used as the *glue* for automatically connecting documents with data (and knowledge). This approach does not require designing or deciding on an integrated schema/view (e.g. [54]), nor mappings between concepts as in knowledge bases (e.g. [28, 53]), or mappings in the form of queries as in the case of databases (e.g. [22]). Entities can be identified in documents, data, database cells, metadata attributes and knowledge bases. Another important point is that the exploitation of LOD is more affordable and feasible, than an approach that requires each search system to keep stored and maintain its own knowledge base of entities and facts.

To the best of our knowledge, our paper is the first work that enriches Web searching with entity mining over the full content of the search hits at query time, by exploiting the scalability of the MapReduce framework over Cloud resources.

2.2 MapReduce and summarization of big data

MapReduce [12, 17, 21, 45] is a popular distributed computation framework widely applied to large scale data-intensive processing, primarily in the so-called *big-data* domain. Big-data applications analyzing data of the order of terabytes are fairly common today. In MapReduce, processing is carried out in two phases, a *map* followed by a *reduce* phase. For each phase, a set of tasks executing user-defined *map* and *reduce* functions are executed in parallel. The former perform a user-defined operation over an arbitrary part of the input and partition the data, while the latter perform a user-defined operation on each partition. MapReduce is designed to operate over *key/value pairs*. Specifically, each Map function receives a key/value pair and emits a set of key/value pairs. Subsequently, all key/value pairs produced during the map phase are grouped by their key and passed (shuffled to the appropriate tasks and sorted) to the reduce phase. During the reduce phase, a reduce function is called for each unique key, processing the corresponding set of values.

Recently, several works have been proposed for exploiting the advantages of this programming model [5, 24, 35, 41, 55], impacting a wide spectrum of areas like information retrieval [9, 40, 41], scientific simulation [41], image processing [12], distributed co-clustering [44], latent Direchlet allocation [61], nearest neighbors queries [62], and the Semantic Web [42] (e.g. from storing/retrieving the increasing amount of RDF data[2] [24] to distributed querying [35] and reasoning [5, 55]).

Previous work by Li et al. [36] on optimally tuning MapReduce platforms contributed an analytical model of I/O overheads for MapReduce jobs performing incremental one-pass analytics. Although their model does not predict total execution time, it is useful in identifying three key parameters for tuning performance: chunk size (amount of work assigned to each map task); external sort-merge behavior; number of reducers. An important difference with our work is that their model does not capture resource requirements of the mapper function, a key concern for us due to the high memory requirements of our NEM engine. Additionally, Li et al. assume that the input chunk size is known a-priori and thus they can predict mapper memory requirements, whereas in our case it is not. Another difference is that Li et al. do not address JVM configuration parameters (such as heap size, reusability across task executions) that are of critical importance: our evaluation shows that incorrectly sizing JVM heap size (such as using default values) leads to a crash; reusing JVMs across task executions can improve execution time by a factor up to 3.3. Our work thus contributes to the state of the art in MapReduce platform tuning by focusing on resource-intensive map tasks whose input requirements are not a-priori known.

2.2.1 Summarization of big data

MapReduce has also been used for producing *summaries of big data* (such as histograms [47]) over which other data analytics tasks can be executed in a more scalable and efficient manner (e.g. see [27]).

[2] By September 2011, datasets from Linked Open Data (http://linkeddata.org/) had grown to 31 billion RDF triples, interlinked by around 504 million RDF links.

Our work relates to data summarization in two key aspects:
First, the output of our analysis over the full search results can be considered a summarization task over text data appropriate for exploration by human users. Text summarization has been investigated by the Natural Language Processing (NLP) community for nearly the last half century (see [16, 43] for a survey). Various techniques and methods have been derived for identifying the important words or phrases, either for single documents or for multiple documents. In the landscape of such techniques, the summarizations that we focus on are *entity-based* , concern *multiple* documents (not single document summarization), and are *topic-driven* with respect to ranking, and *generic* with respect to the set of identified entities. They are *topic-driven* since they are based on the search results of a submitted query (that expresses the desired topic) and the entities occurring in the first hits are promoted. However, since we process the entire contents (not only the short query-dependent snippets of the hits), the produced summary for each document is generic (not query-oriented). The extra information that is mined in this way gives a better overview and can be used from the users for further exploration. Moreover, as we will see later on Sect. 3.3, we identify various levels of functionality each characterizing the analyzed content at different levels of detail, and consequently enable different post-processing tasks by the users (just overviews versus the ability to also explore and restrict the answer based on the produced summary/index). To the best of our knowledge, such summaries have not been studied in the past. Moreover, the fact that they are configurable (one can specify the desired set of categories, entity lists, etc.), allows adapting them to the needs of the task at hand; this is important since there is not any universal strategy for evaluating automatically produced summaries of documents [16].

Second, our implementations perform a first summarization pass over the full search results to (i) analyze a small sample of the documents and provide the end-user a quick preview of the complete analysis; and (ii) collect the sizes of all files and use them in the second (full) pass to better partition that data set achieving better load balancing.

2.3 Cloud computing

MapReduce is often associated with another important trend in distributed computing, the model of *Cloud computing* [4, 32]. Cloud computing refers to a service-oriented utility-based model of resource allocation and use. It leverages virtualization technologies to improve the utilization of a private or publicly-accessible datacenter infrastructure. Large Cloud providers (such as Amazon Web Services, used in the evaluation of our systems) operate out of several large-scale datacenters and thus can offer applications the illusion of infinite resource availability. Cloud-based applications typically feature *elasticity* mechanisms, namely the ability to scale-up or down their resource use depending on user demand. MapReduce fits well this model since it is highly parametrized and can be configured to use as many resources as an administrator deems cost-effective for a particular job. Given the importance of Cloud computing for large-scale MapReduce implementations, we deploy and evaluate our system on a commercial Cloud provider so that our results are representative of those in a real-world deployment of our service.

3 The centralized process

Recalling the exploratory search process described at a high level and depicted in Figs. 1 and 2, we will now describe a centralized (non-parallel) version of the process in more detail. The process consists of the following general steps:

1. Get the top HK (e.g. $HK = 200$) results of a (keyword) search query
2. Download the contents of the hits and mine their entities
3. Group the entities according to their categories and rank them
4. Exploit Semantic Data (the LOD cloud) for semantically enriching the top-EK (e.g. $EK = 50$) entities of each category (also for configuring step 2), and
5. Exploit the entities in faceted search-like (session-based) interaction scheme with the user.

This process can also support classical metadata-based faceted search and exploration [50] by considering categories that correspond to *metadata* attributes (e.g. date, type, language, etc.). Loading such metadata is not expensive (in contrast to applying *NEM* over the full contents) as they are either ready (and external) to the document or the embedded metadata can be extracted fast (e.g. as in [37]). Therefore we do not further consider them in this paper.

In what follows, we introduce notation (Sect. 3.1), we detail the steps of the above process (Sect. 3.2), and we distinguish various levels of functionality (Sect. 3.3). Next, in Sect. 4 we describe the parallelization of this process using the MapReduce framework.

3.1 Notations and entity ranking

Let D be the set of all documents and C the set of all supported categories, e.g. $C = \{Locations, Persons, Organizations, Events\}$. Considering a query q, let A be the set of returned hits (or the top-HK hits of the answer), and let E_c be the set of entities that have been identified and fall in a category $c \in C$. For ranking the elements of E_c, we follow the ranking method proposed in [19]: we count the elements of A in which the entity appears (its *frequency*) but we also take into account the *rank* of the documents that contain that entity in order to promote those entities that are identified in more highly ranked documents (otherwise an entity occurring in the first two hits will receive the same score as one occurring in the last two). For an $a \in A$, let $rank(a)$ be its position in the answer (the first hit has rank equal to 1, the second 2, and so on). We use the formula: $Score(e) = \sum_{a \in docs(e)} ((|A| + 1) - rank(a))$. We can see that an occurrence of e in the first hit counts $|A|$, while an occurrence of the answer in the last document counts for 1.

3.2 The centralized algorithm

Using the notation introduced above, a centralized (non-parallel) algorithm for the general exploratory search process (steps 1–5) described in Sect. 3 is provided below (Algorithm 1). For brevity, the initialization of variables (0 for integer-valued and $\emptyset$ for set-valued attributes respectively) has been omitted. The algorithm takes as input

Algorithm 1 Centralized algorithm

```
1:  function DoTask(Query q, Int HK, EK)
2:      A = Ans(q, HK)                          ▷ Get the top HK hits of the answer
3:      for all i = 1 to HK do
4:          d = download(A[i])                  ▷ Download the contents of hit i
5:          outcome = nem(d)                    ▷ Apply NEM on d
6:          for all (e, c) ∈ outcome do
7:              AC = AC ∪ {c}                   ▷ Update active categories
8:              e.score(c)+ = i                 ▷ Update the score of e wrt c
9:              e.count(c)+ = 1                 ▷ Update the count of e wrt c
10:             e.doclist(c)∪ = {A[i]}          ▷ Update the (e, c) doclist
11:     for all c ∈ AC do                       ▷ For each (active) category
12:         c.elist = top-EK entities after sorting wrt ∗.score(c)
13:         for all e ∈ c.elist do              ▷ LOD-based sem. enrichment
14:             if e.semd=empty then
15:                 e.semd=queryLOD(e)
16:     return {(c, e, e.count(c), e.score(c), e.doclist(c), e.semd) | c ∈ AC, e ∈ c.elist}
```

the query string q, the number HK of top hits to analyze, and the number EK of top entities to show for each category. Its results is a set of tuples, each tuple describing one entity and consisting of 6 values: *category, entity name, entity count, entity score, entity occurrences (doc list), entity's semantic description*.

For clarity, we have used a simplified scoring formula in Algorithm 1. To use the exact entity ranking method described in Sect. 3.1, line 8 should be replaced with $e.score(c)+ = (HK + 1) - i$.

3.3 Levels of functionality

Algorithm 1 can be seen as providing different *levels of functionality*, from *minimal* to *full*, each with different computational requirements and progressively richer post-processing tasks. The *minimal* level functionality (or $L0$) identifies only categories and their entities. The next level ($L1$) contains the results of $L0$ plus *count* information of the identified entities (what is usually called *histogram*). The next level ($L2$) extends the results of $L1$ with the ranking of entities using the method described earlier. Level $L3$ additionally includes the computation of the document list for each entity. The results of $L3$ allow the gradual restriction process by the user. Level $L4$ or *full functionality* further enriches the identified entities with their semantic description. Algorithm 1 corresponds to $L4$.

Each level produces a different kind of summary, capturing different features of the underlying contents and enabling different tasks to be applied over it. The parameters HK and EK can be used to control the size of the corpus covered by the summary, and the desired number of entities to identify.

Note that instead of applying this process over the set A (the top-HK hits returned by the underlying search system(s)) one could apply it over the set of all documents D, if that set is available. This scenario corresponds to the case where one wants to construct (usually offline) an entity-based index of a collection. Consequently, the

parallelization that we will propose in the next sections, could also be used for speeding up the construction of such an index. The extra time required in this case is only the time needed for storing the results of the process in files. Moreover, in that scenario the collection is usually available, thus there is no cost for downloading it. However, our original motivation and focus is to provide these services at meta-level and at query time, which is more challenging.

4 Parallelization

In this section we describe a parallel version of Algorithm 1 and then adapt it to the MapReduce framework (Sect. 4.1). Note that our exposition here focuses on algorithmic issues. We will describe our MapReduce implementations in Sect. 5.

The main idea is to partition the computation performed by Algorithm 1 by documents. Let $AP = \{A_1, \ldots, A_z\}$ be a partition of A, i.e. $A_1 \cup \cdots \cup A_z = A$, and if $i \neq j$ then $A_i \cap A_j = \emptyset$. The downloading of A can be parallelized by assigning to each node n_i the responsibility to download a slice A_i of the partition. The same partitioning can be applied to the *NEM* analysis, namely n_i will apply *NEM* over the contents of the docs in A_i. Other tasks in Algorithm 1 however are not independent as they operate on global (aggregated) information. This is true for the collection of the active categories (AC), the collection of entities falling in each category of AC, the count information for each entity of a category, the doc list of each entity of category. While the task of getting the semantic description of an entity is independent, the same entity may be identified by the docs assigned to several nodes (so redundant computation can take place).

In more detail, instead of having one node responsible for all $1, \ldots, HK$ documents, we can have z nodes responsible for parts of the documents: the first node for $1, \ldots, HK_1$, the second for $HK_1, \ldots, HK_2$, and so on, and the last for $HK_{z-1}, \ldots, HK$. Algorithm 2 (`DoSubTask`) is the part of the computation that each such node should execute, a straightforward part that is essentially similar to the previous algorithm. In line (3) the algorithm assumes access to a table $A[]$ holding the locators (e.g. URLs) of the documents. Alternatively, the values in the cells $A[LowK] - A[HighK]$ can be passed as a parameter to the algorithm.

Algorithm 2 Algorithm for a set of document

```
1:  function DoSubTask(Int LowK, HighK)
2:      for all i = LowK to HighK do
3:          d = download(A[i])                    ▷ Download the contents of hit i
4:          outcome = nem(d)                      ▷ Applies NEM on d
5:          for all (e, c) ∈ outcome do
6:              AC = AC ∪ {c}                     ▷ Update active categories
7:              e.score(c) += i                   ▷ Update the score of e wrt c
8:              e.count(c) += 1                   ▷ Update the count of e wrt c
9:              e.doclist(c) ∪= {A[i]}            ▷ Update the (e, c) doclist
10:     return {(c, e, e.count(c), e.score(c), e.doclist(c)) | c ∈ AC, e ∈ c.elist}
```

Having seen how to create z parallel subtasks, we will now discuss how the results of those subtasks can be aggregated. Note that entity ranking requires aggregated information while the semantic enrichment of the identified entities can be done after ranking. This will allow us to pay this cost for the top-ranked entities only (recall the parameter EK of Algorithm 1), that is those entities that have to be shown at the UI. Semantic enrichment for the rest can be performed on demand, only if the user decides to expand the entity list of a category.

The aggregation required for entity ranking can be performed by Algorithm 3 (`AggregateSubTask`). This algorithm assumes that a single node receives the results from all nodes and performs the final aggregation.

The aggregation task can be parallelized straightforwardly by dividing the work by categories, i.e. use $|AC|$ nodes to each aggregate the results of one category (essentially each will contribute one "rectangle" of information at the final GUI like the one shown in Fig. 2). This is sketched in Algorithm 4 (`AggregateByCategory`). Notice that the count information produced by the reduction phase is correct (i.e. equal to the count produced by the centralized algorithm), because a document is the

Algorithm 3 Aggregation function for all categories

function AggregateSubTask(...)
  Concatenate the results of all SubTasks in a table TABLE
  $AC = \{ c \mid (c, *, *, *, *) \in TABLE\}$
  for all $c \in AC$ **do** ▷ For each (active) category
    $c.entities = \{ e \mid (c, e, *, *, *) \in TABLE\}$
    for all $e \in c.entities$ **do**
      $e.count(c) = \sum\{ cnt \mid (c, e, cnt, *, *) \in TABLE\}$
      $e.score(c) = \sum\{ s \mid (c, e, *, s, *) \in TABLE\}$
      $e.doclist(c) = \cup\{ dl \mid (c, e, *, *, dl) \in TABLE\}$
    $c.elist =$ top-EK entities after sorting wrt $*.score(c)$
    for all $e \in c.elist$ **do** ▷ LOD-based sem. enrichment
      if $e.semd =$empty **then**
        $e.semd =$queryLOD(e)
  return $\{(c, e, e.count(c), e.score(c), e.doclist(c), e.semd) \mid c \in AC, e \in c.elist\}$

Algorithm 4 Aggregation function for one category

function AggregateByCategory(Category c)
  Merge the results of all SubTasks that concern c in a table $TABLE$
  $c.entities = \{ e \mid (c, e, *, *, *) \in TABLE\}$
  for all $e \in c.entities$ **do**
    $e.count(c) = \sum\{ cnt \mid (c, e, cnt, *, *) \in TABLE\}$
    $e.score(c) = \sum\{ s \mid (c, e, *, s, *) \in TABLE\}$
    $e.doclist(c) = \cup\{ dl \mid (c, e, *, *, dl) \in TABLE\}$
  $c.elist =$ top-EK entities after sorting wrt $*.score(c)$
  for all $e \in c.elist$ **do** ▷ LOD-based semantic enrichment
    if $e.semd =$empty **then**
      $e.semd =$queryLOD(e)
  return $\{(c, e, e.count(c), e.score(c), e.doclist(c), e.semd) \mid c \in AC, e \in c.elist\}$

responsibility of only one mapper. The ranking of the entities of each category is correct because Algorithm 2 takes as parameters the *LowK* and *HighK* and uses them in the for loop and line (7).

4.1 Adaptation for MapReduce

In this section we cast the above algorithms in the MapReduce programming style, whose key concepts were introduced in Sect. 2.2. From a logical point of view, if we ignore entity ranking, count information and doclists, the core task of Algorithm 1 becomes the computation of the function $nem : A \rightarrow E \times C$. Using MapReduce, the mapping phase partitions the set A to z blocks and assigns to each node i the responsibility of computing a function $nem_i : A_i \rightarrow E \times C$. The original function nem can be derived by taking the union of the partial functions, i.e. $nem = nem_1 \cup \cdots \cup nem_z$. Mapper tasks therefore carry out the downloading and mining tasks for their assigned set A_i. Note that the partitioning $A \rightarrow \{A_1, \ldots, A_z\}$ should be done in a way that balances the work load. Methods to achieve this will be described in Sect. 5. One or more reducer tasks will aggregate the results by category, as described earlier, and a final reducer will combine the results of the $|C|$ reducers. The correspondence with MapReduce terminology is depicted in the following table:

	Algorithms	MapReduce functions
Algorithm 2	DoSubTask	Map
Algorithm 2	return	emit (with key c, value the rest parts of the tuples)
Algorithm 4	AggregateByCategory	Reduce (params: key c, and value the rest parts of tuples)
Algorithm 4	return	emit (with key c, value the rest parts of the tuples)

Mapper tasks executing Algorithm 2 are emitting (key, value) pairs grouped by category. MapReduce ensures that all pairs with the same key are handled by the same reducer. This means that line (2) of Algorithm 4 is implemented automatically by the MapReduce platform. The platform actually provides an *iterator* over the results of all subTasks that concern c in a Table TABLE.

Figure 3 sketches the performed computations and the basic flow of control for a query.

4.1.1 Amount of exchanged information

In this section we estimate the amount of information that must be exchanged in our MapReduce procedure over network communication. *NEM* could be used for mining all possible entities, or just the named entities in a predefined list. In the extreme case, the entities that occur in a document are in the order of the number of its words. Another option is to mine only the statistically important entities of a document. If d_{asz} denotes the average size in words of a document in D, then the average size of the mined entities per document is in $\mathcal{O}(d_{asz})$. A node assigned a subcollection D_i ($D_i \subseteq D$) will communicate to the corresponding reducers data with size in $\mathcal{O}(|D_i|d_{asz})$. Therefore, the total amount of information communicated over the network for performing mining over the contents of an answer A is in $\mathcal{O}(|A|d_{asz})$.

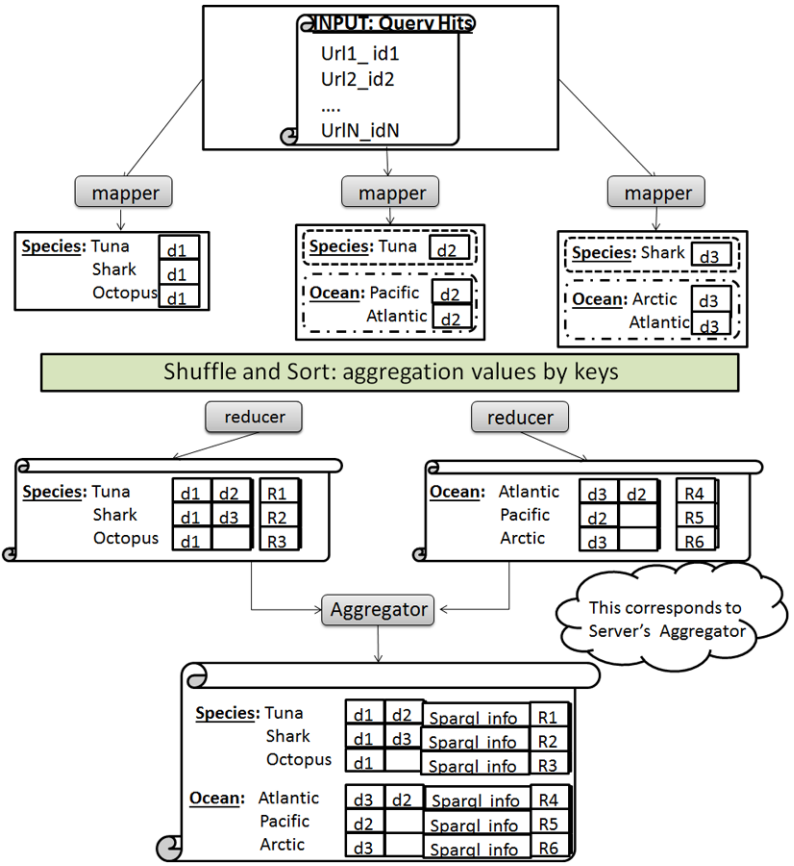

Fig. 3 Example of distributed *NEM* processing using MapReduce

If the set of entities of interest E is predefined and a-priori known, then the above quantity can be expressed as $|A||E|$, so in general, the amount of communicated data is in $\mathcal{O}(|A|\min(d_{asz}, |E|))$. Note that if the entities have only count information and no document lists (functionality *L2* described in Sect. 3.3), then the exchanged information is significantly lower, specifically it is in $\mathcal{O}(z\min(d_{asz}, |E|))$ where z is the number of partitions. This is because each of the z nodes has to send at most d_{asz} (or $|E|$) entities.

4.1.2 An analogy to inverted files

Suppose that the answer A is not ranked, and thus the entities are ranked by their count. In this case the results of our task resemble the construction of an *Inverted File* (IF), otherwise called *Inverted Index*, for the documents in A where the vocabulary of that index is the list of entities of interest (i.e. the set E). The fact that we have $|C|$ categories is analogous to having $|C|$ vocabularies, i.e. as if we have to create $|C|$ inverted files. The *count* information of an entity for a category c ($e.count(c)$)

corresponds to the document frequency (*df*) of that IF, while the doclist of each entity ($e.doclist(c)$) corresponds to the posting list (consisting of document identifiers only, not $tf * idf$ weights) of an IF. This analogy reveals the fact that the size of the output can be very large. An important difference with MapReduce-based IF construction [41] is that our task is more CPU and memory intensive. Besides the cost of initializing the *NEM* component (described in more detail in Sect. 5.2.1), entity mining requires performing lookups, checking rules, running finite state algorithms etc.

5 Implementation

This section describes the MapReduce platform in more detail and outlines two MapReduce procedures to perform scalable entity mining at query time over the full search contents. It also highlights the key factors that affect performance. An important objective guiding our implementation is to achieve effective load balancing of work across the available resources in order to ensure scalable behavior.

5.1 MapReduce platform: Apache Hadoop

Our implementation uses Apache Hadoop [3] version 1.0.3, an open-source Java implementation of the MapReduce [17] framework. MapReduce supports a specific model of concurrent programming expressed as *map* and *reduce* functions, executed by *mapper* and *reducer* tasks respectively. A mapper receives a set of tuples in the form (*key*, *value*), and produces another set of tuples. A reducer receives all tuples (outputs of a mapper) within a given subset of the key space.

Hadoop provides runtime support for the execution of MapReduce *jobs* handling issues such as task scheduling and placement, data transfer, and error management on clusters of commodity (possibly virtual) machines. Hadoop deploys a JobTracker to manage the execution of all tasks within a job, and a TaskTracker in each cluster node to manage the execution of tasks on that node. It uses the HDFS distributed file system to store input and output files. HDFS stores replicas of file blocks in DataNodes and uses a NameNode to store metadata. HDFS DataNodes are typically collocated with TaskTracker nodes, providing a direct (local) path between mapper and reducer tasks and input and output file replicas.

5.2 MapReduce procedures

We have identified two important challenges that must be addressed in our MapReduce implementation: (i) distributing documents to be processed by *NEM* tasks is complicated by the fact that important information, such as content size of the hits, is not known a-priori; and (ii) even with excellent scalability, the end-user may not want to wait for the entire job to complete, preferring a quick *preview* followed by a complete analysis. We have thus experimented with two different MapReduce procedures: a straightforward implementation (oblivious to (i), (ii)) focusing on scalability, and a more sophisticated implementation taking (i) and (ii) into account. The former

is the *single-job* procedure, in which partitioning of work to tasks is done without taking document sizes into account (since the documents are downloaded after the work has been assigned to tasks). The latter is the *chain-job* procedure, in which a first job downloads the documents (and thus determines their sizes) and performs some preliminary entity-mining (producing the preview), while a second job (chained with the first) continues the mining over size-aware partitions of the contents to produce the complete *NEM* analysis.

5.2.1 Single-job procedure

The single-job procedure comprises a first stage that queries a search engine to receive the hits to be processed and prepares the distribution to tasks, followed by a second stage of the MapReduce job itself, both shown in Fig. 4. A Master node (where the JobTracker executes) performs preliminary processing. First it queries a Web Search Engine (WSE), which returns a set of titles, URLs, and snippets. Next, the Master tries to determine the URL content length in order to better balance the downloading and processing of URL contents in the MapReduce job. One way to achieve this is to perform an HTTP HEAD request for each URL prior to downloading it. Unfortunately, our experiments showed that HEAD requests often do not report correct information about content length (they are correct in only 9 %–30 % of the time). In cases of missing/incorrect information, we resort to assigning URL content length to the median size of web pages reported by Google developers [49]. Therefore we consider that the single-job procedure practically does not have a-priori knowledge of URL content sizes.

Our methodology to split work to tasks proceeds as follows: First, we sort documents in desceding order (based on approximate content length) in a stack data structure and compute the aggregated content length of all search results. Then we compute the number of work units (or *splits*) to be created as *(aggregated content length)/(target split size)*, where *target split size* is an upper bound for the amount of document data (MB) to be assigned to each task. We investigate the impact of split size on performance in Sect. 6.5. When not stated otherwise we use a target split size of 1.5 MB. Our process repeatedly pops the top of the stack and inserts it to the split

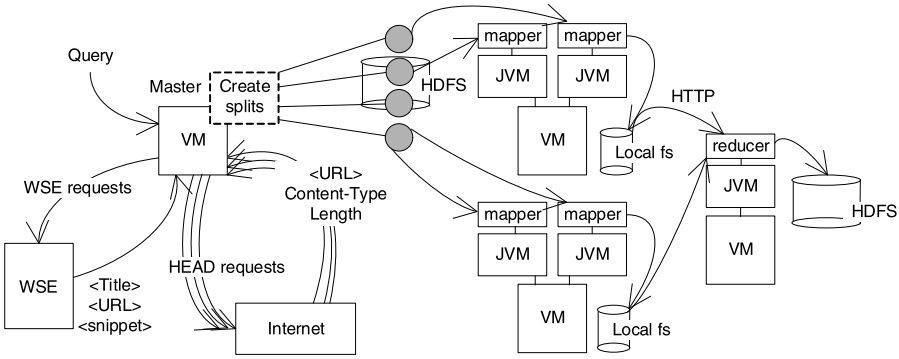

Fig. 4 Single-job design

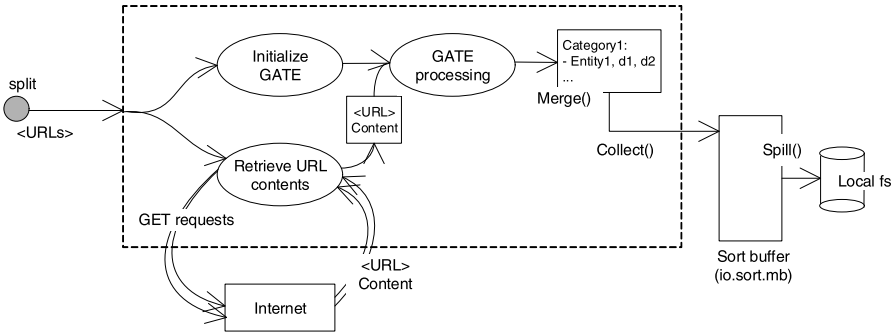

Fig. 5 Single-job mapper

with the minimum total size, until the stack is empty. When the assignment of URLs is complete, the produced splits are stored in HDFS.

At the second stage of the single-job procedure (Fig. 4) a number of *mapper* tasks are created on a number of JVMs hosted by Cloud VMs. JVM configuration and number of JVMs per VM are key parameters that are further discussed in Sects. 5.3 and 6. The operation of each mapper is depicted in Fig. 5. Besides the creation of appropriate-size splits, we are taking care to determine the order of task execution for optimal resource utilization. Taking into account the fact that our datasets typically include a few large documents (Sect. 6.2) we schedule the corresponding long tasks early in the job to increase the degree of overlap with other tasks.

We use the GATE [8, 15] component for performing *NEM* processing. GATE relies on finite state algorithms and the JAPE (regular expressions over annotations) language [51]. Our installation of GATE consists of various components, such as the Unicode Tokeniser (for splitting text into simple tokens such as numbers, punctuation and words), the Gazetteer (predefined lists of entity names), and the Sentence Splitter (for segmenting text into sentences). GATE typically spends about 12 seconds initializing before it is ready for processing documents. This time is spent among other things in loading various artifacts such as lists of entities, expression rules, configuration files, etc. Ideally, the cost of initializing GATE should be paid once and amortized over multiple mapper task executions. To reduce the impact of GATE initialization we decided to exploit the use of reusable JVMs (Sect. 5.3) as well as to overlap that time with the fetching of URL content from the Internet. As soon as HTML content is retrieved it is fed to GATE which processes it and outputs categories and entities.

GATE outputs are continuously merged so that entities under the same category are grouped together, avoiding redundancies. The merged output of a mapper task is kept in a memory buffer until the task finishes, at which point it is collected and emitted into a buffer (of size `io.sort.mb` MB) where it is sorted by category. If the produced output exceeds a threshold (set by `io.sort.spill.percent`) it starts to spill outputs to the local file system (to be merged at a later time). We have sized the sort buffer appropriately to ensure a single spill to disk per mapper. We use a *combiner* [52] to merge the results from multiple mappers operating on the same node.

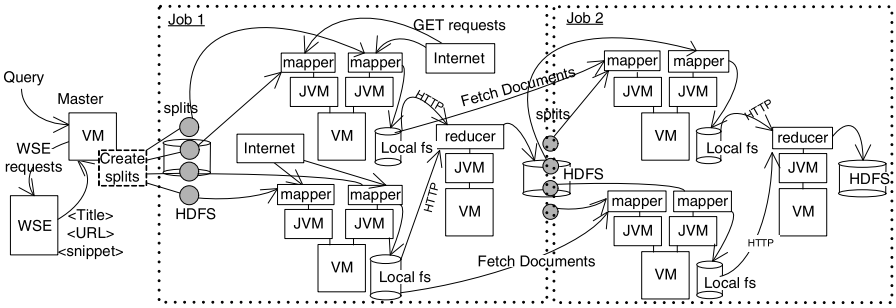

Fig. 6 Chain-job design

The reduce phase performs the merging of mapper outputs per category and computes the scores of the different entities (Sect. 3.1). The latter is possible since the document identifiers reflect the positions of the documents in the list (e.g. $d18$ means that this document was the 18^{th} in the answer). Since this is fairly lightweight functionality we anticipate that there is little benefit from parallelizing this phase and thus use a single reducer task. This choice has the additional benefit of avoiding the need to merge outputs from multiple reducers.

5.2.2 Chain-job procedure

We have developed an alternative MapReduce procedure that consists of two chained [60] jobs (Jobs #1 and #2, where the output of Job #1 is input to Job #2) as shown in Fig. 6. The rationale behind this design is the following: Job #1 downloads the entire document set and thus gains exact information about content sizes. Therefore Job #2 (full analysis) is now able to perform a size-aware assignment of the remaining documents to tasks. At the same time, we believe that most users appreciate a quick *NEM* preview on a sample of the hits before getting the full-scale analysis. Job #1 is designed to perform such a preview.

In Fig. 6, the Master node queries the search engine getting the initial set of titles, URLs, and snippets. Then, it creates the initial split of the URLs without using any information about their sizes. Since Job #1 tasks are primarily downloading documents while performing only limited-scale *NEM* analysis, there is no need to create more tasks than the number of JVM slots available. Job #1 mappers (Fig. 7) will read their split and begin downloading URL contents while starting the initialization of GATE. Downloaded content is stored as local files. As soon as GATE is ready, it starts consuming some of these files. As soon as downloading of all URLs in its split is complete, each map task continues with a certain amount of entity mining and then terminates. Once all mappers are done, a reducer uses the sizes of all yet-unprocessed files to create the splits for Job #2. Having accurate knowledge of file sizes, we can ensure that the splits are as balanced (in terms of size) as possible using the methodology described in Sect. 5.2.1. Additionally, having already performed some amount of entity mining, the system provides a preview of the *NEM* analysis to the user. The entire Job #1 is currently scheduled to take about a minute (though this is configurable), including the overhead of starting up and terminating it. A key point is that

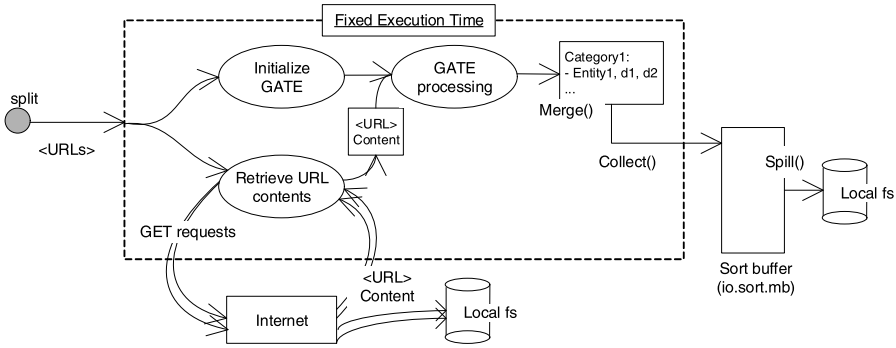

Fig. 7 Chain-job mapper #1 (preview analysis)

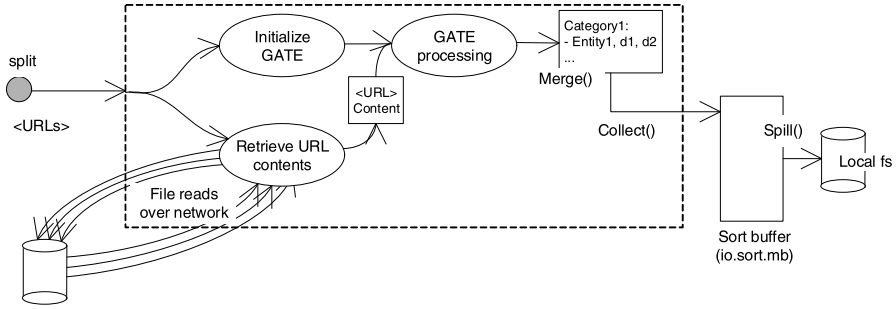

Fig. 8 Chain-job mapper #2 (full analysis)

within the fixed amount of time for Job #1, one can choose to perform a deeper preview (process more documents) by allocating more resources (VMs) to that job. This point is further investigated in Sect. 6.4.1.

Job #2 features mappers (Fig. 8) that initially read files downloaded by the previous job and process them through GATE. The files are originally stored in the local file systems of the nodes that downloaded them, so reading them typically involves high-speed network communication [32]. Having created a balanced split via Job #1, we have ensured a more efficient utilization of resources compared to what is possible with the single-job procedure. Just as in the single-job procedure, the scores of the entities are computed at the single reducer of Job #2.

5.3 Platform parameters impacting performance

While MapReduce is a straightforward model of concurrent programming, tuning the underlying platform appropriately is a major undertaking that is often not well understood by application programmers. A variety of configuration parameters set at their default values usually result in bad performance, and arbitrary experimentation with them often leads to crashing applications. A major objective of this paper is to highlight the key characteristics of the underlying platform (Hadoop and the Ama-

zon EC2 Cloud), tune them appropriately for our workloads, and to investigate their impact on performance.

5.3.1 Mapper parameters

A key parameter is the *split*, the input data given to each task, which can be either a static or a dynamic parameter (e.g., either fixed part of an input file or dynamically composed from arbitrary input sources). Dividing the overall workload size by the average size of the split determines how many map tasks will be scheduled and executed within the MapReduce job. For example, if the total size of hits that we want to analyse is 12 MB and the split set to 2 MB we will need a total of six tasks. One needs to carefully size the split assigned to each mapper. Generally, Hadoop implementers are advised to avoid fine-grain tasks due to the large overhead of starting up and terminating them. On the other hand, larger splits increase their memory requirements eventually increasing garbage collection (GC) activities and their overhead. In our evaluation we examine the precise impact of task granularity in performance (Sect. 6.5).

Another key parameter is the number of Java Virtual Machines (JVMs) per TaskTracker node (VM) available to concurrently execute tasks, which is controlled via `mapred.tasktracker.map.tasks.maximum`; we will refer to this parameter as *JpV* (or JVMs per VM). Generally, this parameter should be set taking the parallelism (number of cores) and memory capacity of the underlying TaskTracker into account. Fewer JVMs per TaskTracker means that there is more heap space available to allocate to them. On the other hand, a higher number of JVMs will better match parallelism in the underlying VM. Another potential optimization is the *reusability* of JVMs across task executions. MapReduce can be configured to reuse (rather than start fresh) a JVM [52, p. 170] across task invocations, thus amortizing its startup/termination costs. The degree of reusability of JVMs is configured via the `mapred.job.reuse.jvm.num.tasks` parameter, which defines the maximum number of tasks to assign per JVM instance (the default is one).

At the output of a mapper, one needs to allocate sufficient memory to the Sort buffer (Fig. 5) to avoid repeated spills and subsequent merge-sort operations. The size of the buffer (controlled by the `io.sort.mb` parameter) defaults to 100 MB. Overdrawing on available memory for this buffer means that there will be less memory left for GATE processing. In our case, the summarization performed by *NEM* reduces the size of the input by an order of magnitude. Even at the default setting of `io.sort.mb`, the rate of output expected from our mappers is not expected to produce spills to disk. Thus `io.sort.mb` is a non-critical parameter for our MapReduce jobs.

Since the map phase takes up the bulk of our MapReduce jobs we have paid particular attention on how to optimally tune MapReduce parameters for it. Our tuning methodology explores the tradeoffs and interdependencies between these parameters and outputs the heap size per JVM, JVMs per VM (*JpV* parameter), degree of reusability, and split size (MB). The methodology relies on a systematic exploration of the parameter space using targeted experiments in two phases: The first (or *intra-JVM*) phase explores single-JVM performance whereas the second (or *inter-JVM*)

Table 1 Execution time varying split size and heap size with fixed reusability R (✗ means the job failed)

Split size (MB)	Reusability: R				
	Heap size (MB)				
	heap_1	heap_2	heap_3	heap_4	heap_5
split_1	✗	$time_1$	$time_2$	$time_3$	$time_4$
split_2	✗	✗	$time_5$	$time_6$	$time_7$
split_3	✗	✗	✗	$time_8$	$time_9$

phase explores performance of concurrently executing JVMs. The intra-JVM phase explores values of split size, heap size, and reusability for a JVM and produces tables such as Table 1.

These tables are produced by first creating splits of different sizes typical of the input workload. For each size, a group of splits are given as input to a JVM configured for a specific heap size and reusability level. The size of the group is chosen to ensure that the job reaches steady state but remains reasonably short to keep the overall process manageable. The final outcome (success/failure, execution time) is recorded in the corresponding cell of the table. The tradeoffs in the parameter space are: Higher split sizes improve efficiency but require larger amounts of heap to ensure successful and efficient execution. Higher reusability improves amortization of the JVM startup/termination and GATE initialization costs but requires increasing amounts of heap size to avoid failures and to improve performance. The heap size parameter takes specific values computed as follows:

$$\text{JVM heap memory} = \frac{\text{total VM memory available}}{JpV}. \tag{1}$$

JpV ranges from a minimum level of parallelism (equal to the number of cores in the VM) to a maximum level that corresponds to the minimum JVM heap deemed essential for operation of the JVM.

Our methodology selects configurations from the parameter space of Table 1 with the following two requirements: (i) they terminate successfully; (ii) their execution time is close to a minimum. In our experience, a set of feasible and efficient solutions can be rapidly determined by direct observation of the tables by a human expert as exhibited in Sect. 6.7.

For those configurations $C_i = (\text{split}_i, \text{heap}_i, \text{reusability}_i)$ that are feasible and have minimal execution time, we continue to the inter-JVM phase that examines them on concurrently executing JVMs. For each configuration C_i, we deploy a number of splits (a multiple of that used in the previous phase, to account for concurrently executing JVMs) on as many JVMs as C_i's heap allows (JpV, Eq. (1)). Our goal in this phase is to examine the impact of different degrees of concurrency on efficiency. Configurations with higher concurrency than can be efficiently supported by the VM platform will be excluded in this phase. Between configurations that perform best, we select that with the largest heap size for its improved ability to handle larger-than-average splits.

5.3.2 Reducer parameters

Our MapReduce jobs require that a reducer collects a fixed set of categories. Deciding on the number of reducers to use in a particular MapReduce job has to take into account the overall amount of work that needs to be performed. More reducer tasks will help better parallelize the work (assuming units of work are not too small) while fewer reducer tasks reduce the need for aggregating their outputs (performed through an explicit HDFS command). The summarization performed by *NEM* processing as well as the tuple merging in our mappers significantly reduce the amount of information flowing between the map and reduce stages, making a single reducer task the best option in our targeted input datasets. The execution time of the reducer is proportional to the size of the mappers' output as quantified in Sect. 4.1.1.

The reduce process starts by fetching map outputs via HTTP. The time spent on communication during this phase depends on the amount of exchanged information and the network bandwidth. Terminated mappers communicate their results to the reducer in parallel to the execution of subsequent instances of mappers, thus there is a significant degree of overlap. Therefore, communication time is in the critical path only after all mappers have completed (and this time is expected to be minimal for most practical purposes).

While receiving tuples from mappers, the reduce task performs a merge-sort of the incoming tuples [52] spilling buffers to disk if needed. The default behavior of Hadoop MapReduce is to always spill in-memory segments to disk even if all fetched files fit in the reducer's memory, aiming to free up memory for use in executing the reducer function. When memory is not an issue, the default behavior can be overridden, to avoid the I/O overhead of unnecessarily spilling tuples to disk. This can be done by specifying a threshold (percentage over total heap size, default 0 %) over which data collected at the reducer should be spilled to disk. Setting the spill threshold higher (for example, to 60 % of 256 MB of heap allocated to the reducer) is sufficient to fully avoid spills in our experiments without creating memory pressure for the mapper. For example, a 300 MB input dataset produces about 24 MB of total mapper output, which is well below the set threshold.

As reduce tasks store their output on HDFS, having a local DataNode collocated with each TaskTracker helps, since writes from reduce tasks always go to a local replica at local-disk speeds. HDFS supports data replication with a default value of 3. Since we are not interested in long-term persistent storage for the files written to HDFS, we set the replication factor to one. This has the added benefit of avoiding the overhead of maintaining consistency across replicas. Finally, we decided to install/locate all the needed resources for tasks on all machines instead of using the DistributedCache facility [57] to fetch them on demand over the network. While this requires extra effort on the part of the administrator, it results in faster job startup times.

5.4 A Measure of imbalance in task execution times

To capture the degree of imbalance in task execution times in MapReduce jobs (which may be a cause for inefficiency as shown in our evaluation, Sect. 6.5), we have defined

a measure that we term the *imbalance percentage (IP)*. IP refers to the variation in last-task completion times (we focus on mappers since this is the dominant phase in our jobs) across the available JVMs of a given node i and is defined as follows. Assume that there are N_i JVMs available to execute tasks on node i and that all JVMs start executing tasks at the same time ($T_{i,0}$). The first JVM to run out of tasks does so at time $T_{i,min}$ and the last JVM to run out of tasks does so at time $T_{i,max}$. The ideal execution time on node i would therefore be $T_{i,min} + D_i$ where $D_i = (T_{i,max} - T_{i,min})/N_i$. The imbalance percentage on node i is thus defined as

$$IP_i = \frac{D_i}{T_{i,max}} \times 100\ \%$$

The imbalance percentage for the entire job, denoted as IP, is calculated as the average of the above quantities across all nodes.

6 Evaluation

In this section we evaluate performance of our MapReduce procedures during entity mining of different datasets and measure the achieved speedup with different numbers of nodes as well as the impact on performance of a number of platform parameters. In Sect. 6.1 we outline sources of non-determinism in our system and ways to address them. In Sect. 6.2 we describe the procedure with which we create realistic synthetic datasets and in Sect. 6.3 we describe our experimental platform. From Sect. 6.4 we focus on scalability and on the impact of different platform parameters to efficiency and high performance.

6.1 Sources of non-determinism

A number of external factors that exhibit varying and time-dependent behavior are sources of non-determinism that had to be carefully considered when setting up our experiments. In more detail, these external factors fall into two categories:

Search engine Results returned by the Bing[3] search RSS service over multiple invocations of the same query (top-K hits) vary in both number and contents over time. The Bing search RSS service associates about 650 results per query and each request can bring back at most 50 results. These results can differ at each request invocation. Finally, Bing results are accessible via XML pages, which in several occasions are not well formed (missing XML tags) returning different results for the same query.

Internet access Web page download times can vary significantly depending on the Internet connectivity of the Cloud provider as well as dynamic Internet conditions (e.g. network congestion) at the time of the experiment. Another highly-variable factor concerns the availability of web pages. Even when the Bing search engine returns

[3]We chose Bing because it does not limit the number of queries submitted, in contrast to Google, which blocks the account for one hour if more than 600 queries are submitted.

identical results for the same query, trying to download the full content of the search results from the Internet may fail as some pages may be inaccessible at times (connection refused, connection/read time-out) leading to variations in our input collections. Furthermore, the fact that in 70 %–91 % of HTTP HEAD requests Web servers either do not provide content-length information or have refused our requests (connection/read time-out) adds further variability. Finally, the efficiency of the external SPARQL endpoints is highly variable.

To reduce the effect of the above factors on the evaluation of our systems and to facilitate reproducibility of our results we decided to perform our experiments with controlled datasets (Sect. 6.2), which we plan to make available to the research community. Additionally, since semantic enrichment depends strongly on the efficiency of the external SPARQL endpoints and is orthogonal to the scalability of the core MapReduce procedures we decided to omit it from this evaluation. While these assumptions lead to better insight into the operation of our core system on the MapReduce platform, it is important to note that our system is fully functional and available for experimentation upon request.

6.2 Creating datasets

To evaluate our system under workloads of progressively larger size we create several different datasets. Our dataset creation process starts by performing multiple queries to the Bing RSS engine. The queries are chosen from the top 2011 searches reported by Bing. These queries are based on the aggregation of billions of search queries and are grouped by People, News Stories, Sports Stars, Musicians, Consumer Electronics, TV Shows, Movies, Celebrity Events, Destinations, and Other Interesting Search. For each one of these groups the top-10 queries are reported. After retrieving the results of queries from Bing we merge them into a single set. We then download the contents of all Web pages onto a single VM. We also create the cluster of URLs and store the IDs of documents that belong to each cluster.

An example on how to create a dataset from these queries is the following: Choose the first query from each group (if the query is already added to the collection then we omit it) and submit it to the search engine. For each submitted query, download the contents of the top-K hits. It is hard to estimate an appropriate K such as to achieve a given dataset size (e.g. 100 MB). Our way to achieve this is to keep downloading results until the aggregated content length exceeds the target. From this collection we randomly remove documents until we achieve the desired size. We randomly remove documents, instead of removing only low ranked documents, in order to simulate a realistic situation. In a real situation the system has to analyze every document (in the set of top-k results), even those which are low ranked. Consequently, a random removal yields a more realistic dataset, in the sense that the latter will also contain low-ranked documents. However, we should note that even if we were removing only low-ranked documents, the only difference would be on the quality of the identified entities. The process and the measurements would not be affected.

Note that since documents are of arbitrary size it is still hard to achieve the identical size targeted, so we settle for removing the documents that best approximate the aggregated dataset size. We repeat this procedure with queries in the second position of each group, and so on, to create more datasets.

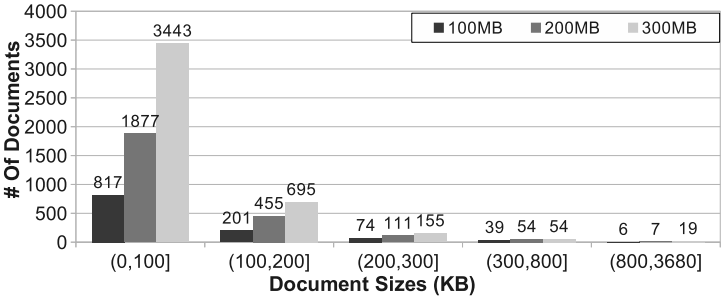

Fig. 9 Distributions of sizes for xMB-SET1, $x \in \{100, 200, 300\}$

We use the naming scheme xMB-SETy for our created datasets, where x is the dataset size ($\in \{100, 200, 300\}$) and y is the dataset sample identifier ($\in \{1, 2, 3\}$). Figure 9 presents the distribution of sizes for xMB-SET1. The created datasets represent a range from 1226 (100 MB) to 4365 (300 MB) documents (hits) on average. Most documents are small: 89.5 %, 93.1 %, and 94.7 % of the documents in the 100 MB, 200 MB, and 300 MB datasets respectively are less than 200 KB in size. The largest dataset (300 MB) corresponds (approximately) to the first 87 pages of a search result (where by default each page has 50 hits). We thus believe that the created datasets fully cover our targeted application domain.

6.3 Experimental platform: Amazon EC2

Our experiments were performed on the Amazon Elastic Compute Cloud (EC2) using up to 9 virtual machine (VM) nodes. A VM of type `m1.medium` (1 virtual core, 3.4 GB of memory) was assigned the role of JobTracker (collocated with an HDFS NameNode and a secondary NameNode). We used up to 8 VMs of type `m1.large` (2 virtual cores, 7.5 GB of main memory each) as TaskTrackers (collocated with an HDFS DataNode), a sufficient cluster size for the targeted problem domain. We provision 4 JVMs to execute concurrently on each VM, 3 used for mappers and one for a reduce task. This setup was experimentally determined to be optimal, allowing sufficient memory to be used as JVM heap (2.2 GB for a mapper, 256 MB for a reducer) while also taking advantage of the parallelism available in the VM. We use a single reducer task for all jobs in our experiments, for reasons explained in Sect. 5.3. We have verified that there is no benefit from increasing the number of reducers in our experiments. We configure JVMs to be reused by 20 tasks before terminated. The VM images used were based on Linux Ubuntu 12.04 64-bit.

To monitor the execution of our MapReduce jobs we employed *CloudWatch*, an Amazon monitoring service, and our own deployment of Ganglia [38, 39], a scalable cluster monitoring tool that provides visual information on the state of individual machines in a cluster, along with the `sFlow` [46] plug-in to get the metrics for each JVM (e.g. mapTask, reduceTask). To monitor the performance of the Java garbage collector we used the IBM Pattern Monitoring and Diagnostic Tool [25] to analyse JVM GC logs and tune the system appropriately.

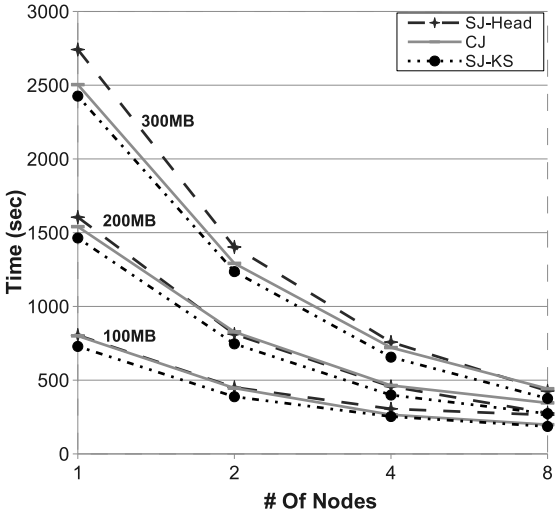

Fig. 10 Query execution time for growing dataset size and increasing number of nodes (VMs)

6.4 Scalability

In this section we evaluate the performance improvement as the number of nodes (VMs) used to perform *NEM* processing increases. We used datasets of sizes 100 MB, 200 MB, and 300 MB (Sect. 6.2) and evaluate the following three system configurations: (1) Single-job procedure using HTTP HEAD info, referred to as *SJ-HEAD*; (2) Single-job with a-priori exact knowledge of document sizes, referred to as *SJ-KS* (this is an artificial configuration that we created solely for comparison purposes); and (3) Chain job, referred to as *CJ*.

We define the *speedup* achieved using N nodes (VMs) as $S_N = T_1/T_N$ where T_1 and T_N are the execution times of our MapReduce procedures on a single node and on N nodes respectively. Note that we do not use as T_1 the time the sequential *NEM* algorithm takes on a single node since such an execution is infeasible for our problem sizes (the JVM where GATE executes crashes). Note that the optimal speedup possible for a computation is limited by its sequential components, as stated by Amdahl's law [2]. Namely, if f is the fraction of the computational task that cannot be parallelized then the theoretically maximum possible speedup is $S_N = 1/(f + (1-f)/N)$.

Figure 10 depicts the execution time for the three different system configurations with increasing number of nodes (VMs). Tables 2, 3, 4 depict the speedups achieved in all cases. Our observations are:

– All systems exhibit good scalability, which improves with increasing dataset size. For the 300 MB dataset using 8VMs, we observe a SJ-KS speedup of $S_8 = 6.45$ and a SJ-HEAD speedup of $S_8 = 6.42$ compared to the single-node case. This is the best speedup we achieved in our experiments. We believe that the overall runtime of about 6.3′ is within tolerable limits and justifies real-world deployment of our service. We have not attempted larger system sizes because—as will be described next—scalability at this point is practically limited by the tasks that analyze the largest documents in our sets.

Table 2 Speedup for SJ-HEAD

# VMs	100 MB	200 MB	300 MB
1	1	1	1
2	1.79	1.97	1.95
4	3.01	3.51	3.61
8	4.04	5.79	6.42

Table 3 Speedup for SJ-KS

# VMs	100 MB	200 MB	300 MB
1	1	1	1
2	1.87	1.96	1.96
4	2.87	3.66	3.69
8	3.91	5.36	6.45

Table 4 Speedup for CJ

# VMs	100 MB	200 MB	300 MB
1	1	1	1
2	1.78	1.86	1.93
4	2.63	3.32	3.47
8	3.05	4.45	5.66

– To compare the observed scalability to the theoretically optimal (taking Amdahl's law into account) we need to consider the sequential components of the MapReduce job as well as scheduling issues (imbalances in last-task completion times, examined in Sect. 6.5) that reduce the degree of parallelism in the map phase of the job. We have analyzed these components for a specific case, the 200 MB-SET1 dataset in the SJ-HEAD configuration (Table 5). In this case, we measured the total run time of a job, the sequential components in each case, the execution time of the longest task (analyzing a 3.68 MB document, a size that far exceeds the vast majority of other documents in the set (Sect. 6.2)) and the total runtime of the map phase. The ideal speedup is computed based on Amdahl's law, assuming perfect parallelization of the map phase. The observed speedup is close to (within 9 % of) the ideal for 2VMs and 4VMs but diverges from it for 8VMs. The reason for the lower efficiency in this case is the fact that the map phase becomes bounded by the longest task (3.68 MB, executing for $4'09''$ out of the $4'36''$ the entire job takes), which cannot be subdivided. It is important to note that despite our size-aware task scheduling algorithm (where long tasks are scheduled early in the job, Sect. 5.2.1), tasks of that size in some cases create scheduling imbalances resulting in suboptimal scalability.

Table 5 Detailed analysis: SJ-HEAD, 200 MB-SET1

#VMs	Job time (s)	Sequential component (s)	Longest task (s)	Map phase (s)	Observed speedup	Ideal speedup
1	1585	29	249	1556	1	1
2	818	26	249	792	1.93	1.97
4	458	27	249	431	3.46	3.81
8	276	27	249	249	5.74	7.15

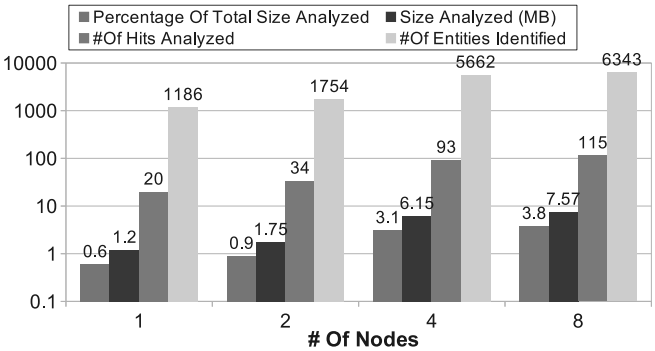

Fig. 11 Analysis of chain-job #1 on 200 MB dataset, 1-min job execution time

– The CJ speedup observed for the 300 MB dataset using 8VMs is $S_8 = 5.66$. The disadvantage of CJ compared to the single-job configurations can be attributed to the additional non-parallelizable overhead of its two jobs.
– SJ-KS outperforms SJ-HEAD and CJ by a small margin (0.5–3 %, decreasing with higher dataset sizes). This is expected since it leverages a-priori knowledge about document sizes with reduced overhead from using one rather than two jobs.

6.4.1 Scalability of job #1 in chain-job procedure

In this section we focus on the quality of the summarization work (documents processed and entities identified) performed by Job #1 in the chain-job procedure with increasing system size. Our measure of scalability in this evaluation is the amount of work performed within a fixed amount of time as the number of processing nodes increases. We define the amount of work done as the size of documents analyzed (as a percentage over the entire document list and in absolute numbers (MB)) and the number of entities identified.

Figure 11 depicts the amount of work performed by Job #1 with an increasing system size for a fixed execution time (one minute). We observe that as the number of processing nodes increases, the percentage of documents analyzed grows from 0.6 % to 3.8 % (the number of entities identified grows from 1186 to 6343) yielding a progressively better preview (summary) of the entire document list. Based on these results we conclude that Job #1 exhibits good scalability, and offers a powerful trade-off to an administrator when aiming to improve the quality of preview: either allocate

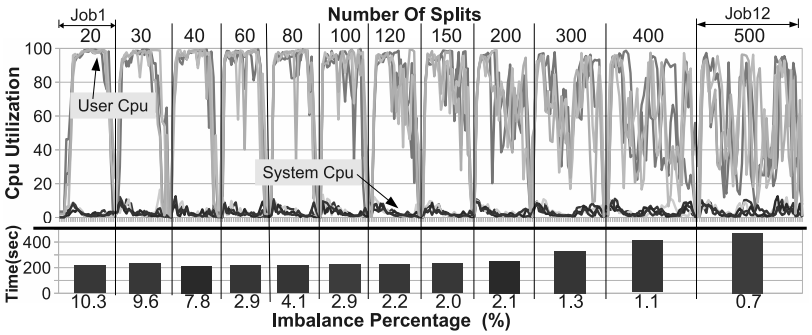

Fig. 12 CPU utilization, execution time and imbalance percentage for a number of jobs whose only difference is the number of splits

more VMs to Job #1 (costly but faster option) or allow more execution time on fewer VMs (cheaper but slower option).

6.5 Impact of number of splits

In this section we study the effectiveness of a job as we vary the number of input splits. Using 4 nodes (VMs) and the dataset 100 MB-SET1 (created from about 1226 query hits), we execute a sequence of jobs over it with progressively larger number of splits (and consequently, decreasing split size). For each job we measure CPU utilization reported by each VM (in `m1.large` VMs the reported CPU utilization is the average of the VM's two virtual cores), job execution times, and the imbalance percentage within each job.

Figure 12 (top portion) depicts per-node CPU utilization for each job. Figure 12 (bottom) presents the job execution times (bars) and the imbalance percentage within each job. We observe that CPU utilization is better for fewer splits (20–100), where the workload assigned to each mapper task takes on average from 96.5 s to 18.5 s as shown in Fig. 13. As we increase the number of splits (120–500), CPU utilization decreases due to the higher scheduling overhead associated with many small (granularity of a few (tens) of seconds) tasks. For example, for 500 splits the job execution time is nearly 2.2 times the execution time of a the job with 20 splits.

A key observation is that job-execution times in the range of 20 to 120 splits are nearly constant. Within this range, the workload balance (evidenced by the imbalance percentage) improves as the number of splits grows. Combining with our previous observation (that split sizes in the range 150–500 suffer from excessive scheduling overhead) we arrive at the conclusion that a number of splits between 100 and 120 is a reasonable choice taking all things into account. Choosing a smaller number of splits would increase the probability that a split may include several big documents, increasing the garbage collection (GC) overhead (more details in Sect. 6.6) and imbalance percentage. However, big documents make their presence felt even in the case of small split sizes (they are responsible for the large ratio between maximum and average execution times in Fig. 13).

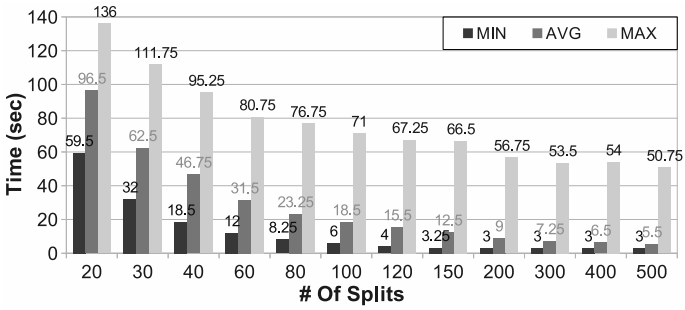

Fig. 13 Map task min/average/max execution time for different number of splits

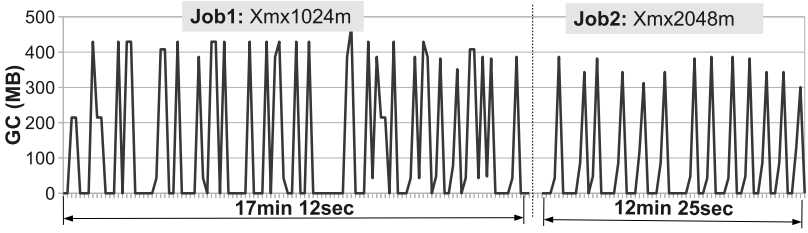

Fig. 14 Garbage collection activity for two jobs that differ in the amount of memory allocated to JVMs (1 GB (*left*) vs. 2 GB (*right*))

6.6 Impact of heap size

In this section we evaluate the impact of JVM heap allocations to job performance. First we compare query execution time of two jobs, Job 1 and Job 2, each executing on a single node (VM), with the node hosting three JVMs (assigned to mapper tasks), using the 100 MB-SET1 dataset instance. One of the jobs assigns 1 GB of heap space to its JVMs whereas the other job assigns twice that amount (2 GB). We used the `mapred.map.child.java.opts` parameter (set to `Xmx1024m` and `Xmx2048m` respectively) to control JVM heap memory allocations. The JVMs used in this experiment where configured to be reusable (20 times).

Figure 14 presents overall execution time and GC activity (in MB) for the two jobs. We observe that Job 1 (1 GB JVM heap) takes an additional 4.8 minutes (about 30 %) to completion compared to Job 2 (2 GB JVM heap). The reason for this delay is the additional GC overhead that impacts overall query execution time. The figure shows that GC activity is less frequent for Job 2 (2 GB).

The impact of heap size can in fact be far more severe than presented above. In fact, standard heap allocations (default of 200 MB in several JVMs) always lead to job failure. Table 6 depicts measurements of overall execution time of jobs consisting of a single split, where the split size varies from 1 MB to 35 MB and JVM heap size varies from its default value of 200 MB to 2.2 GB. These ranges of split sizes and heap sizes represent practically relevant values (we have not seen additional performance benefits from higher heap sizes for this range of split sizes). Each job runs on a single

Table 6 Execution time (in seconds) of single-split jobs with varying split and heap sizes

Split size (in MB)	Heap size (in MB)								
	200	512	756	1024	1256	1512	1756	2048	2256
1	✗	58	56	54	56	56	56	57	55
5	✗	129	106	105	105	105	105	107	105
10	✗	✗	161	163	161	163	162	165	166
15	✗	✗	500	208	210	204	208	210	213
20	✗	✗	✗	420	259	258	259	262	260
25	✗	✗	✗	✗	787	404	346	350	347
35	✗	✗	✗	✗	✗	1100	550	441	448

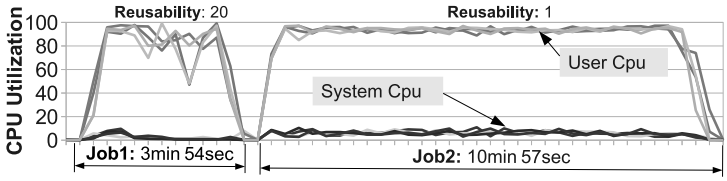

Fig. 15 CPU utilization and job execution time for two jobs whose only difference is reusability of JVMs

non-reusable JVM. The reported numbers are average execution times from three different splits created randomly for each size. A ✗-value in a cell indicates that the job either crashed or terminated for being unresponsive (executing within GATE) for more than 10 minutes (default value of `mapred.task.timeout` parameter).

Table 6 shows that with increasing split size one needs to use increasingly higher JVM heap sizes to avoid job failures. Furthermore, within those heap sizes that lead to successfully completed jobs, increasing heap size allocations lead to better performance (as also evidenced by Fig. 14), up to a point where additional heap does not help: excessively high heap can hurt due to the JVM startup and teardown overheads (part of which are proportional to heap size).

In the following section we consider the impact of an additional parameter, JVM reusability, and then exhibit the tuning methodology outlined in Sect. 5.3 for selecting key parameters of our MapReduce jobs.

6.7 Impact of JVM reusability

In this section we first exhibit the performance advantage of JVM reusability by setting up two *NEM* jobs on the 100 MB-SET1 dataset (100 splits) using 4 nodes (VMs) and 3 JVMs per VM. Job 1 uses reusable JVMs (the value of the `mapred.job.reuse.jvm.num.tasks` parameter set to 20), whereas Job 2 does not reuse JVMs (the value of the parameter is set to 1, which is the default). Figure 15 presents the CPU utilization for two jobs whose only difference is JVM reusability. Each of the curves corresponds to one of the four VMs used in this experiment.

Table 7 Execution time (in seconds) varying split size and heap size with reusability 1

Split size (MB)	Reusability: 1						
	Heap size (GB)						
	0.2	0.5	1.1	1.3	1.6	2.2	3.3
1	✗	2833	2821	2811	2798	2911	2939
2.5	✗	1998	1958	1962	1971	2004	2019
5	✗	✗	1619	1612	1634	1653	1664

Table 8 Execution time (in seconds) varying split size and heap size with reusability 10

Split size (MB)	Reusability: 10						
	Heap size (GB)						
	0.2	0.5	1.1	1.3	1.6	2.2	3.3
1	✗	✗	1700	1693	1697	1703	1712
2.5	✗	✗	✗	3488	1679	1523	1468
5	✗	✗	✗	✗	✗	3445	1468

Table 9 Execution time (in seconds) varying split size and heap size with reusability 20

Split size (MB)	Reusability: 20						
	Heap size (GB)						
	0.2	0.5	1.1	1.3	1.6	2.2	3.3
1	✗	✗	4258	1773	1634	1647	1658
2.5	✗	✗	✗	✗	✗	3645	1561
5	✗	✗	✗	✗	✗	✗	✗

We observe that JVM reusability improves job execution time by about 2.8 times in this case. The advantage of reusability can be higher with increasing number of splits (e.g. for a 300 MB dataset or 300 splits, job performance improves by about 3.3 times). We note that with reusability the cost of initializing GATE (about 12 s) is paid once during startup of each JVM and amortized over for the rest of tasks that are executed in the same JVM. Finally, we observe that JVM reusability affects CPU utilization: the job featuring non-reusable JVMs consumes more CPU that is spent in startup/teardown of JVMs during task initialization and termination.

Having seen the individual impact of the number/size of splits, heap size, and reusability parameters, we now demonstrate our tuning methodology described in Sect. 5.3 to select appropriate settings for these parameters. Tables 7, 8, 9 depict execution time (in seconds) of jobs analyzing a 100 MB dataset[4] varying split sizes (1 MB, 2.5 MB, 5 MB) corresponding to (100, 40, 20) splits; heap sizes (200 MB–

[4]This size is chosen to ensure full utilization in all cases, as max(Reusability) × max(Split size) = 20 × 5 MB = 100 MB.

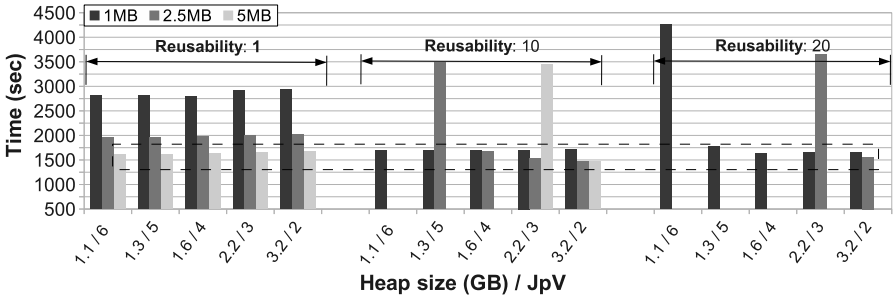

Fig. 16 Feasible configurations from Tables 7, 8, 9. The dashed box highlights the best choices

3.3 GB) computed from Eq. (1); and reusability in the range (1, 10, 20). The reported times are averages over three executions using different split sets created randomly. During this *intra-JVM* phase we use one VM with a single JVM executing on it at any time. The ✗-value has the same meaning as in Table 6.

Our first observation is that Table 7 features the largest fraction of successful runs (17 out of 21 possible cases), albeit at the cost of excessive execution times for 1 MB and 2.5 MB splits sizes. Execution times of 5 MB-split jobs outperform the others because it has the smallest number of splits (20) minimizing the costs of JVM startups/terminations and GATE initializations. A related observation (counter-intuitive at first) is that job execution times worsens with higher heap allocations, especially for smaller splits. This is explained by the fact that higher heap sizes increase the cost of JVM startup and teardown overhead, which is paid all too frequently at the default reusability level of one.

For increasing reusability we observe that execution times improve at the cost of fewer successful job configurations. For example, in Table 9 (reusability 20) we observe that performance improves for particular configurations, e.g., by up to 44 % for (1 MB split size, 2256 MB heap size) compared to the same configuration with reusability one. Figure 16 depicts execution times of those configurations from Tables 7, 8, 9 that are feasible (do not fail). Within the dashed box we distinguish execution times that are within a small range of the minimum, thus reducing the list of possible configurations to 30 % of the initial set (19 out of 63).

Having completed the *intra-JVM* phase, we move to the *inter-JVM* phase to explore the performance of the selected configurations with concurrently executing JVMs, as described in Sect. 5.3. In this phase we must use larger datasets to ensure full utilization in all cases. To keep the size of the experiments manageable we study each split size separately with a dataset sized max(Reusability) × max(JpV) × Split size MB. We have excluded split size 5 MB mainly due to the high imbalance percentage it leads to as demonstrated in Fig. 12. Figure 17 depicts execution times for jobs executing on JpV JVMs within a single VM.

Choosing the best configurations from Figure 17 for each split size leads to the following cases:

- For split size 1 MB: Best choices are
  - C_1 = (1.6 GB heap, 4 JVMs, reusability 20)
  - C_2 = (2.2 GB heap, 3 JVMs, reusability 20)

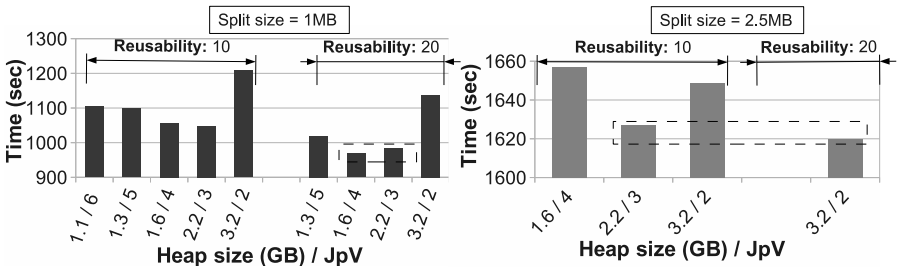

Fig. 17 Execution time (s) under concurrently executing JVMs. Dashed boxes highlight the best choices for split sizes 1 MB (*left*) and 2.5 MB (*right*)

Between the two we prefer C_2 for its larger heap size (and thus its ability to handle larger than average objects in a collection).

- For split size 2.5 MB: Best choices are
  - C_3 = (2.2 GB heap, 3 JVMs, reusability 10)
  - C_4 = (3.2 GB heap, 2 JVMs, reusability 20)

  Between them we prefer C_4 for its larger heap size.

Finally, to compare C_2 to C_4 we run a last set of experiments on both configurations with a dataset sized max(Split size) $\times$ max(JpV) $\times$ max(Reusability) $= 2.5 \times 3 \times 20 = 150$ MB (150 splits for C_2 or 60 splits for C_4) for full utilization. C_2 is found to result in (marginally) lower execution time by about 4 %, pointing to it as the best among the 63 choices considered.

6.8 Comparative results for different functionalities and number of categories

In Sect. 3.3 we described the different levels of functionality that can be supported by our parallel *NEM* algorithms, ranging from *minimal* to *full* functionality. In this section we evaluate the impact of the levels of functionality on performance. Moreover we discuss the impact of increasing the number of supported categories on performance and output size.

At first, we note that the time required by the mining component is *independent* of the level of functionality, since the mining tool always has to scan the documents and apply the mining rules. Of course, an increased number of categories will increase the number of lookups, but the extra cost is relatively low. The main difference between the various levels of functionality is the size of the mappers' output and the size of the reducers' output.

For instance, in our experiments over 200 MB-SET1, for $L0$ (minimal functionality) the map output was 6 MB, which is 2.8 times less than the output for $L3$ (full functionality). The difference is not big. This is because the average size of the doc lists of the entities is small. This is evident by Fig. 18 which analyzes the contents of the reducers' output, specifically the figure shows the number of entities for each category and the maximum and average sizes of the entities' doc lists. We observe that the average number of documents per identified entity is around 3. This means that the sizes of the exchanged information by the different levels of functionalities

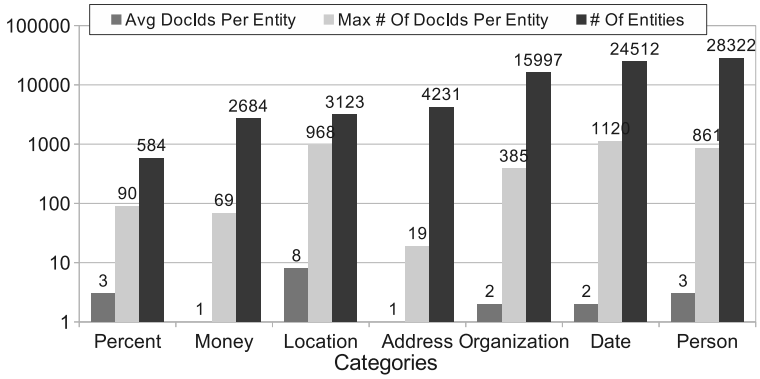

Fig. 18 Content analysis of Reducer's output

are quite close, and this is aligned with the $\times 2.8$ difference of the mappers' output.

Of course this depends on the dataset and the entities of interest. We could say that as $|E|$ increases (recall that the set E can be predefined) the average size of the doc lists of these entities tends to get smaller. Consequently, we expect significant differences in the amounts of exchanged information of these levels of functionality when $|E|$ is small, because in that case the average size of the doc lists can become high.

In our datasets, and with the selected set of categories and mining rules, the differences of the end-to-end running times were negligible. This is because most time is spent by the mining component, not for communication.[5]

Actually, one could "predict" the differences as regards the latency due to the extra amount of information to be exchanged, based on the analysis of Sect. 4.1.1, where we estimated the amount of information that has to be exchanged in various levels of functionality, and the network throughput of the cloud at hand. Recall that in Sect. 4.1.1 the amount of information that has to be exchanged in various levels of functionality is measured as a function of $|A|$ (the number of hits to be analyzed), d_{asz} (the average size in words of a document), and z (the number of partitions, i.e. number of nodes used). Specifically:

- Cases $L0$, $L1$ (i.e. $L0$ + counts) and $L2$ (i.e. $L1$ plus ranking; the latter does not increase data): $\mathcal{O}(z \min(d_{asz}, |E|))$
- Case $L3$ ($L2$ + doclists): $\mathcal{O}(|A| \min(d_{asz}, |E|))$.

It follows that the difference in the amounts of the exchanged information between $L3$ and $L0/L1/L2$ can be quantified as: for the case where E is fixed (predefined), the difference is in $\mathcal{O}(|A||E| - z|E|) = \mathcal{O}(|E|(|A| - z))$, while for the case where $|E|$ is not fixed (not predefined), the difference is in $\mathcal{O}(|A|d_{asz} - zd_{asz}) = \mathcal{O}(d_{asz}(|A| - z))$.

[5]Even with the full functionality, the reduce phase constitutes only about 2 % of the total job time when analyzing 300 MB-SET1 using 4 nodes.

To grasp the consequences, let's put some values in the above formulae. For $|A| = 100$, $d_{asz} = 400$ KB, and $z = 8$, the quantity $d_{asz}(|A| - z)$ has the value 400(100–8) KB = 39.2 MB. By considering the capacity of the cloud one could predict the maximum time required for transferring this amount of information. For instance, the throughput of Amazon Cloud is 68 MB/s ([32]). It follows that 39.2 MB require less than one second.

Moreover, the above time assumes that the information will be communicated in one shot. Since it will be done in parallel, the required time will be less. Specifically, in case we have an ideal load balancing, and thus all mappers start sending their results at the same point in time, for predicting the part of the end-to-end running time that corresponds to data transfer, it is enough to consider what one mapper will send.

For the case of $L0/L1/L2$ this amount is in $\mathcal{O}(\min(d_{asz}, |E|))$, while for the case of L3 it is in $\mathcal{O}(|D_i| \min(d_{asz}, |E|))$ where $|D_i|$ is the number of docs assigned to a node, and we can assume that $|D_i| = |A|/z$. Therefore the difference between L3 and $L0/L1/L2$, can be quantified as follows: for the case where E is fixed (predefined), the difference is in $\mathcal{O}(|A||E|/z - |E|) = \mathcal{O}(|E|(|A|/z - 1))$, while for the case where $|E|$ is not fixed (not predefined), the difference is in $\mathcal{O}(|A|/z\, d_{asz} - d_{asz}) = \mathcal{O}(d_{asz}(|A|/z - 1))$.

Obviously, for big values, the above difference can become significant. For instance, for building the index of a collection of 1 billion (10^9) of documents, with $d_{asz} = 400$ KB and $z = 11$, the extra amount of exchanged data will be $400 \times 10^9/10$ KB. With a network throughput of 100 MB/s, the extra required time will be 4×10^5 seconds, i.e. around 4.6 days.

Increasing the number of categories Next we evaluated system performance as the number of categories increases. In general, we expect that the amount of exchanged data, number of lookups, rule executions, and the output size, increase as the number of categories increases. Growing the number of categories from 2 to 10 in increments of 2 did not show a measurable impact on performance. However, the reducer output size and the number of identified entities increased: in particular, the average number of identified entities for the 100 MB-SETs was from 16,300 (with two categories, Person and Location, enabled) to 45,227 with all ten categories enabled.[6]

Compression As a final note, we originally considered the possibility to compress the doclists of the exchanged entities using *gapped* identifiers encoded with variable-length codes (e.g. Elias-Gamma), as it is done in inverted files, to reduce the amount of information exchange in $L3$. However, since our experiments showed that all different functionalities have roughly the same end-to-end running time, we decided not to use any compression as it would not affect the performance. However, compression could be beneficial in cases one wants to build an index of a big collection, as in the billion-sized scenario that we described earlier.

[6] In particular: Person, Location, Organization, Address, Date, Time, Money, Percent, Age, Drug.

6.8.1 *Improving scalability through fragmentation of documents*

In Sect. 6.4 we observed that total execution time and overall scalability are bounded by the longest tasks, which correspond to the largest files in a collection. This limit is hard to overcome in NLP tasks such as text summarization or entity mining where entities are accompanied by local scores, as these tasks require knowledge of the entire document. Thus we have considered documents as undivided elements (or "atoms") which cannot be subdivided. However, in cases where documents can be subdivided into fragments, one could adopt a finer granularity approach for better load balancing and thus improved speedup.

To validate the benefit of this approach we produced a variation of the Chain-Job (CJ) procedure in which we partition files in smaller fragments. Specifically, we divide documents larger than 800 KB in fragments whose size does not exceed the split size. Our results show that this variation of CJ achieves a speedup of $\times 6.2$ for a 300 MB dataset when using 8 Amazon EC2 VMs, an improvement over the speedup of $\times 5.66$ with standard CJ (without document fragments). This is attributed to the fact that the longest task execution time is now 80 s in contrast to standard CJ where this was 249 s. We should note that this approach is only applicable to the CJ procedure since its preview phase can be used to fragment the large documents.

6.9 Synopsis of experimental results: executive summary

The key results of our evaluation of the proposed MapReduce procedures for performing scalable entity-based summarization of Web search results are:

- Our scalable MapReduce procedures can successfully analyze input datasets of the order of 4.5K documents (search hits) at query time in less than $7'$. Such queries far exceed the capabilities of sequential *NEM* tools. The use of special computational resources (such as highly parallel multiprocessors with very large memory capacity) are a potential alternative to our use of Cloud computing resources, but we consider our solution to be more cost-effective and ubiquitous.
- We have observed speedups of $\times 6.4$ when scaling our system to 8 Amazon `m1.large` EC2 VMs (that is, 24 JVMs concurrently executing map tasks) using our single-job procedure. While we consider this to be a very good level of scalability, it deviates from perfect (theoretically possible [2]) scalability due to two primary reasons: The existence of a few very large documents in the input dataset (Sect. 6.2) means that tasks analyzing them may—even in the best possible execution schedule– become a limiting factor during the mapping phase (since documents cannot be subdivided in *NEM* analysis); additionally, variability in last-task completion times (expressed via the *imbalance percentage*, Sect. 6.5) means that even in the absence of such very large documents, tasks rarely finish simultaneously, introducing idle time in the mapping phase. The impact of these factors increases with system size (number of VMs).
- Use of our *chain-job* (CJ) MapReduce procedure performs a size-aware assignment of the remaining documents to tasks of Job #2 and offers the qualitative benefit of a *quick preview* of the *NEM* analysis, compared to the *single-job* (SJ) procedure. An

administrator can decide to perform a more accurate preview by either allocating more resources (VMs) or allotting more time to the first-job of CJ, exploiting a cost vs. wait-time tradeoff. In our experiments, going from one to 8 EC2 VMs increases the percentage of documents analyzed during the preview phase from 0.6 % to 3.8 % of a 200 MB dataset. CJ exhibits somewhat lower scalability compared to SJ ($\times$5.66 vs. $\times$6.45 for a 300 MB dataset) due to the overhead of using two rather than one MapReduce job. The issue of big documents can be tackled by an evaluation procedure that is based on document fragments. This method can be adopted if the desired text mining task can be performed on document fragments.
- For optimal tuning of the Hadoop platform on Amazon EC2, we evaluated the impact of the number and size of splits, JVM heap size, and JVM reusability parameters on performance. We also presented a tuning methodology for selecting optimal values of these parameters. Our methodology is a valuable aid to help an expert in tuning the Hadoop MapReduce platforms in order to optimize resource efficiency during execution of our MapReduce procedures.
- Our evaluation of the impact of different levels of functionality and number of categories (up to ten categories) showed no impact on end-to-end running time in the used collections, thus allowing the use of enriched analysis ($L3$) without additional cost over lower levels of functionality. However, for bigger collections the impact can be significant, and for this reason we have analyzed the expected impact analytically.

7 Concluding remarks

We have described a scalable method for entity-based summarization of Web search results at query time using the MapReduce programming framework. Performing entity mining over the full contents of results provided by keyword-search systems requires downloading and analyzing those contents at query time. Using a single machine to perform such a task for queries returning several thousands of hits becomes infeasible due to its high computational and memory cost. In this paper we have shown how to decompose a sequential Named Entity Mining algorithm into an equivalent distributed MapReduce algorithm and deploy it on the Amazon EC2 Cloud. To achieve the best possible load balancing (maximizing utilization of resources) we designed two MapReduce procedures and analyzed their scalability and overall performance under different configuration/tuning parameters in the underlying platform (Apache Hadoop). Our experimental evaluation showed that our MapReduce procedures achieve a scalability of up to $\times$6.4 on 8 Amazon EC2 VMs when analyzing 300 MB datasets, for a total runtime of less than $7'$. Our evaluation fully addresses our targeted application domain (our larger queries include on average 4365 hits or about 87 pages of a typical search result).

There are several directions for future work extending the research presented in this paper. One interesting direction is to generalize the chain-job procedure to provide progressively more results over a number of stages (rather than just two). While we anticipate an impact on the efficiency of the overall MapReduce job, a constant stream of results is expected to be a welcome feature by end users. Finally, on the issue of the type of Cloud resources allocated to a specific instance of our MapReduce

procedures, we plan to explore cost/performance tradeoffs within the large diversity of resource types available across Cloud providers.

Acknowledgements Many thanks to Carlo Allocca and to Pavlos Fafalios for their contributions. We thankfully acknowledge the support of the *iMarine* (FP7 Research Infrastructures, 2011–2014) and *PaaSage* (FP7 Integrated Project 317715, 2012–2016) EU projects and of Amazon Web Services through an Education Grant. We also acknowledge the interesting discussions we had in the context of the MUMIA COST action (IC1002, 2010–2014).

Appendix A: Vertical search applications

Fig. 19 Two screens from vertical search applications, one for the *marine domain* (*top*) and another for *patent search* (*bottom*)

References

1. Allocca, C., dAquin, M., Motta, E.: Impact of using relationships between ontologies to enhance the ontology search results. In: Simperl, E., Cimiano, P., Polleres, A., Corcho, O., Presutti, V. (eds.) The Semantic Web: Research and Applications. Lecture Notes in Computer Science, vol. 7295, pp. 453–468. Springer, Berlin (2012)
2. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. pages 483–485, 1967
3. Apache Software Foundation: The Apache Hadoop project develops open-source software for reliable, scalable, distributed computing. http://hadoop.apache.org/. Accessed: 03/05/2013
4. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010)
5. Assel, M., Cheptsov, A., Gallizo, G., Celino, I., Dell'Aglio, D., Bradeško, L., Witbrock, M., Della Valle, E.: Large knowledge collider—a service-oriented platform for large-scale semantic reasoning. In: Proceedings of the International Conference on Web Intelligence, Mining and Semantics (WIMS'11), pp. 41:1–41:9. ACM, New York (2011)
6. Bonino, D., Ciaramella, A., Corno, F.: Review of the state-of-the-art in patent information and forthcoming evolutions in intelligent patent informatics. World Pat. Inf. **32**(1), 30–38 (2010)
7. Broder, A.: A taxonomy of web search. SIGIR Forum **36**(2), 3–10 (2002)
8. Callaghan, G., Moffatt, L., Szasz, S.: General architecture for text engineering. http://gate.ac.uk/. Accessed: 03/04/2013
9. Callan, J.: Distributed information retrieval. *Advances in Information Retrieval*, 7, 127–150, 2002
10. Caputo, A., Basile, P., Semeraro, G.: Boosting a semantic search engine by named entities. In: Proceedings of the 18th International Symposium on Foundations of Intelligent Systems (ISMIS'09), pp. 241–250. Springer, Berlin (2009)
11. Carpineto, C., DAmico, M., Romano, G.: Evaluating subtopic retrieval methods: clustering versus diversification of search results. Inf. Process. Manag. **48**(2), 358–373 (2012)
12. Chen, S., Schlosser, S.W.: Map-reduce meets wider varieties of applications. Technical report IRP-TR-08-05, Intel Research Pittsburgh (2008)
13. Cheng, T., Yan, X., Chang, K.: Supporting entity search: a large-scale prototype search engine. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD'07), pp. 1144–1146. ACM, New York (2007)
14. Clinton, D., Tesler, J., Fagan, M., Snell, J., Suave, A., et al.: OpenSearch is a collection of simple formats for the sharing of search results. http://www.opensearch.org/. Accessed: 03/05/2013
15. Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V.: A framework and graphical development environment for robust NLP tools and applications. In: Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02) (2002)
16. Das, D., Martins, A.: A survey on automatic text summarization. Literature Survey for the Language and Statistics II course at CMU **4**, 192–195 (2007)
17. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
18. Ernde, B., Lebel, M., Thiele, C., Hold, A., Naumann, F., Barczyn'ski, W., Brauer, F.: ECIR—a lightweight approach for entity-centric information retrieval. In: Proceedings of the 18th Text REtrieval Conference (TREC 2010) (2010)
19. Fafalios, P., Kitsos, I., Marketakis, Y., Baldassarre, C., Salampasis, M., Tzitzikas, Y.: Web searching with entity mining at query time. In: Proceedings of the 5th Information Retrieval Facility Conference (IRFC 2012), Vienna (2012)
20. Fafalios, P., Salampasis, M., Tzitzikas, Y.: Exploratory patent search with faceted search and configurable entity mining. In: Proceedings of the 1st International Workshop on Integrating IR Technologies for Professional Search (ECIR 2013) (2013)
21. Grossman, R.L., Gu, Y.: Data mining using high performance data clouds: experimental studies using sector and sphere. *CoRR*, abs/0808.3019:920–927, 2008
22. Halevy, A.Y.: Answering queries using views: a survey. VLDB J. **10**(4), 270–294 (2001)
23. Herzig, D.M., Tran, T.: Heterogeneous web data search using relevance-based on the fly data integration. In: Proceedings of the 21st International Conference on World Wide Web (WWW '12), pp. 141–150. ACM, New York (2012)
24. Husain, M., Khan, L., Kantarcioglu, M., Thuraisingham, B.: Data intensive query processing for large rdf graphs using cloud computing tools. In: 2010 IEEE 3rd International Conference on Clod Computing (CLOUD), pp. 1–10. IEEE Press, New York (2010)

25. Hwang, J.: IBM pattern modeling and analysis tool for Java garbage collector. https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=22d56091-3a7b-4497-b36e-634b51838e11 Accessed: 28/01/2013
26. Jaccard, P.: The distribution of the flora in the alpine zone. New Phytol. **11**(2), 37–50 (1912)
27. Jestes, J., Yi, K., Li, F.: Building wavelet histograms on large data in mapreduce. Proc. VLDB Endow. **5**(2), 109–120 (2011)
28. Jiménez-Ruiz, E., Grau, B.C., Horrocks, I., Berlanga, R.: Ontology integration using mappings: towards getting the right logical consequences. In: The Semantic Web: Research and Applications, pp. 173–187. Springer, Berlin (2009)
29. Joho, H., Azzopardi, L., Vanderbauwhede, W.: A survey of patent users: an analysis of tasks, behavior, search functionality and system requirements. In: Proc. of the 3rd Symposium on Information Interaction in Context, pp. 13–24. ACM, New York (2010)
30. Käki, M.: Findex: search result categories help users when document ranking fails. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 131–140. ACM, New York (2005)
31. Käki, M., Aula, A.: Findex: improving search result use through automatic filtering categories. Interact. Comput. **17**(2), 187–206 (2005)
32. Kitsos, I., Papaioannou, A., Tsikoudis, N., Magoutis, K.: Adapting data-intensive workloads to generic allocation policies in cloud infrastructures. In: Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS 2012), pp. 25–33. IEEE Press, New York (2012)
33. Kohn, A., Bry, F., Manta, A., Ifenthaler, D.: Professional Search: Requirements, Prototype and Preliminary Experience Report, pp. 195–202. 2008
34. Kules, B., Capra, R., Banta, M., Sierra, T.: What do exploratory searchers look at in a faceted search interface? In: Proceedings of the 9th ACM/IEEE-CS Joint Conference on Digital Libraries, pp. 313–322. ACM, New York (2009)
35. Kulkarni, P.: Distributed SPARQL query engine using MapReduce. Master's thesis
36. Li, B., Mazur, E., Diao, Y., McGregor, A., Shenoy, P.: A platform for scalable one-pass analytics using mapreduce. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD'11), pp. 985–996. ACM, New York (2011)
37. Marketakis, Y., Tzanakis, M., Tzitzikas, Y.: Prescan: towards automating the preservation of digital objects. In: Proceedings of the International Conference on Management of Emergent Digital EcoSystems (MEDES'09), pp. 60:404–60:411. ACM, New York (2009)
38. Massie, M., Chun, B., Culler, D.: The ganglia distributed monitoring system: design, implementation, and experience. Parallel Comput. **30**(7), 817–840 (2004)
39. Massie, M., Li, B., Nicholes, B., Vuksan, V., Alexander, R., Buchbinder, J., Costa, F., Dean, A., Josephsen, D., Phaal, P., et al.: Monitoring with Ganglia. O'Reilly Media, Inc., Sebastopol (2012)
40. McCreadie, R., Macdonald, C., Ounis, I.: Comparing distributed indexing: to mapreduce or not? In: Proc. of LSDS-IR, pp. 41–48 (2009)
41. Mccreadie, R., Macdonald, C., Ounis, I.: Mapreduce indexing strategies: studying scalability and efficiency. Inf. Process. Manag. **48**(5), 873–888 (2012)
42. Mika, P., Tummarello, G.: Web semantics in the clouds. IEEE Intell. Syst. **23**(5), 82–87 (2008)
43. Nenkova, A., McKeown, K.: A survey of text summarization techniques. In: Mining Text Data, pp. 43–76 (2012)
44. Papadimitriou, S., Sun, J.: Disco: distributed co-clustering with map-reduce: a case study towards petabyte-scale end-to-end mining. In: Eighth IEEE International Conference on Data Mining (ICDM'08), pp. 512–521. IEEE Press, New York (2008)
45. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., Dewitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proceedings of the 35th SIGMOD International Conference on Management of Data (SIGMOD'09), pp. 165–178. ACM, New York (2009)
46. Phaal, P.: SFlow is an industry standard technology for monitoring high speed switched networks. http://blog.sflow.com/. Accessed: 03/05/2013
47. Poosala, V., Haas, P., Ioannidis, Y., Shekita, E.: Improved Histograms for Selectivity Estimation of Range Predicates vol. 25, pp. 294–305. ACM, New York (1996)
48. Pratt, W., Fagan, L.: The usefulness of dynamically categorizing search results. J. Am. Med. Inform. Assoc. **7**(6), 605–617 (2000)
49. Ramachandran, S.: Google developers: Web metrics. https://developers.google.com/speed/articles/web-metrics. Accessed: 03/05/2013
50. Sacco, G., Tzitzikas, Y.: Dynamic Taxonomies and Faceted Search. Springer, Berlin (2009)

51. Thakker, D., Osman, T., Lakin, P.: Java annotation patterns engine. http://en.wikipedia.org/wiki/JAPE_(linguistics). Accessed: 03/04/2013
52. Tom, W.: Hadoop: The Definitive Guide. O'Reilly, Sebastopol (2009)
53. Tzitzikas, Y., Meghini, C.: Ostensive automatic schema mapping for taxonomy-based peer-to-peer systems. In: Cooperative Information Agents VII, pp. 78–92. Springer, Berlin (2003)
54. Tzitzikas, Y., Spyratos, N., Constantopoulos, P.: Mediators over taxonomy-based information sources. VLDB J. **14**(1), 112–136 (2005)
55. Urbani, J., Kotoulas, S., Oren, E., Van Harmelen, F.: Scalable distributed reasoning using Mapreduce. pp. 634–649 (2009)
56. van Zwol, R., Garcia Pueyo, L., Muralidharan, M., Sigurbjörnsson, B.: Machine learned ranking of entity facets. In: Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'10), pp. 879–880. ACM, New York (2010)
57. Venner, J.: Pro Hadoop. Apress, Berkeley (2009)
58. White, R.W., Kules, B., Drucker, S.M., Schraefel, M.: Supporting exploratory search, introduction (special issue). Communications of the ACM. Commun. ACM **49**(4), 36–39 (2006)
59. Wilson, M., et al.: A longitudinal study of exploratory and keyword search. In: Proceedings of the 8th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL'08), pp. 52–56. ACM, New York (2008)
60. Yahoo! Inc. Chaining jobs. http://developer.yahoo.com/hadoop/tutorial/module4.html#chaining. Accessed: 09/05/2013
61. Zhai, K., Boyd-Graber, J., Asadi, N., Alkhouja, M.: Mr. LDA: a flexible large scale topic modeling package using variational inference in Mapreduce. In: Proceedings of the 21st International Conference on World Wide Web (WWW'12), pp. 879–888. ACM, New York (2012)
62. Zhang, C., Li, F., Jestes, J.: Efficient parallel knn joins for large data in Mapreduce. In: Proceedings of the 15th International Conference on Extending Database Technology, pp. 38–49. ACM, New York (2012)