Entity-based Summarization of Web Search Results using MapReduce

Ioannis Kitsos

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

Thesis Advisor: Assistant Prof. *Yannis Tzitzikas*

This work was partially supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

Entity-based Summarization of Web Search Results using MapReduce

Thesis submitted by
Ioannis Kitsos
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: Ioannis Kitsos

Committee approvals:

Yannis Tzitzikas
Assistant Professor, Thesis Supervisor

Kostas Magoutis
Research Scientist, ICS-FORTH, Thesis Co-Supervisor

Dimitris Plexousakis
Professor, Committee Member

Departmental approval:

Angelos Bilas
Professor, Director of Graduate Studies

Heraklion, April 2013

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, while exploiting also the structured (and semantic) data that are now available, is to enrich the search process with *entity mining* over the *full contents* of the search results where the entities of interest can be specified by semantic sources. 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.

In this thesis 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 and we introduce two different evaluation procedures the *Single-Job* (SJ) and the *Chain-Job* (CJ). Moreover, we specify criteria that determine the selection and ranking of the (often numerous) discovered entities.

In the sequel, we report experimental results about the achieved speedup in various settings. We show that with the SJ procedure the achieved speedup is close to the theoretically optimal speedup ($2,5\% - 19,7\%$ lower than the optimal for a $300MB$ dataset and from 2 up to 8 Amazon EC2 VMs respectively) and justify this difference. Indicatively, we achieve a speedup of up to $x6.4$ on 8 EC2 VMs when analyzing $4,365$ hits (corresponding to $300MB$) with a total runtime of less than 7 minutes (an infeasible task when using a single machine due to high computational and memory requirements). CJ exhibits somewhat lower scalability compared to SJ ($x5.66$ on 8 EC2 VMs) with a longer total runtime (about 30 secs more for a $300MB$ dataset) due to the overhead of using two rather than one MapReduce job. On the other hand, CJ offers the qualitative benefit of providing a quick preview of the results of the analysis.

Another important contribution of this thesis is a thorough evaluation of platform configuration and tuning, an aspect that is often disregarded and inadequately addressed in prior work, but crucial for the efficient utilization of resources. We show that the proposed evaluation methods utilize well the resources (fully utilized CPU, efficient memory allocation), and the tasks do not have an unreasonably high overhead (e.g. garbage collection, unnecessarily startup/teardown of JVMs during task initialization and termination, imbalance in last-task execution times).

Περίληψη

Παρόλο που οι Μηχανές Αναζήτησης ευρετηριάζουν τεράστιους όγκους ιστοσελίδων (εγγράφων γενικότερα), για τις επερωτήσεις των χρηστών επιστρέφουν μόνο μια γραμμική λίστα «επιτυχιών» (hits). Αν και αυτό να είναι ικανοποιητικό για τις ανάγκες τις επικεντρωμένης αναζήτησης (focalized search), αυτού του τύπου οι αποκρίσεις δεν παρέχουν στο χρήστη ούτε εποπτεία των επιτυχιών, ούτε τη δυνατότητα ευέλικτης εξερεύνησης τους, ούτε κάποια βαθύτερη ανάλυση των περιεχομένων τους. Ένας τρόπος για παροχή προηγμένης πλοήγησης, και συνάμα αξιοποίησης των (σημασιολογικά) δομημένων δεδομένων που είναι πλέον διαθέσιμα, είναι ο εμπλουτισμός της διαδικασίας αναζήτησης με εξόρυξη οντοτήτων επί του περιεχομένου των επιτυχιών, όπου οι οντότητες που μας ενδιαφέρουν μπορούν να προσδιοριστούν από σημασιολογικές πηγές. Αυτός ο εμπλουτισμός δίνει στο χρήστη μια εποπτεία του πληροφοριακού χώρου των επιτυχιών, η οποία επίσης του επιτρέπει τη σταδιακή μείωση τους ώστε να μπορεί εκείνος να εντοπίσει τις επιθυμητές επιτυχίες, ακόμα και αν αυτές είναι πολύ πίσω στην κατάταξη.

Σε αυτήν τη διατριβή θεωρούμε το γενικό σενάριο όπου αυτές οι υπηρεσίες προσφέρονται ως *μέτα-υπηρεσίες* (ήτοι επί συστημάτων που προσφέρουν αναζήτηση μέσω λέξεων κλειδιών), χωρίς να απαιτείται ο εκ των προτέρων ευρετηριασμός των υποκείμενων συλλογών εγγράφων. Για να κάνουμε εφικτή την παροχή τέτοιων υπηρεσιών για μεγάλους όγκους αποτελεσμάτων, χρησιμοποιούμε το μοντέλο κατανεμημένου υπολογισμού MapReduce επί μιας υποδομής Υπολογιστικού Νέφους (Amazon EC2). Συγκεκριμένα δείχνουμε πως ο απαιτούμενος υπολογισμός μπορεί να παραγοντοποιηθεί σε συναρτήσεις MapReduce και παρουσιάζουμε δυο διαφορετικές διαδικασίες υπολογισμού, την «μονοκόμματη» (στο εξής SJ από το *Single-Job*) και την αλυσιδωτή (CJ, από το *Chain-job*). Επιπλέον, προσδιορίζουμε κριτήρια που καθορίζουν την επιλογή και κατάταξη των ευρεθέντων οντοτήτων (συχνά πολυπληθείς).

Στη συνέχεια, αναφέρουμε εκτενή πειραματικά αποτελέσματα σχετικά με την επιτευχθείσα επιτάχυνση (speedup) σε διαφορετικές ρυθμίσεις. Δείχνουμε ότι με τη διαδικασία SJ επιτυγχάνουμε επιτάχυνση η οποία είναι κοντά στην θεωρητικά βέλτιστη επιτάχυνση ($2,5-19,7\%$ χαμηλότερη από την θεωρητικά βέλτιστη για ένα σύνολο δεδομένων $300MB$ και από 2 έως 8 Amazon EC2 VMs αντίστοιχα) και αναλύουμε αυτή την απόκλιση. Ενδεικτικά, μπορούμε να επιτύχουμε επιτάχυνση έως και $x6.4$ με 8 EC2 VMs κατά την ανάλυση 4.365 επιτυχιών (που αντιστοιχούν σε $300MB$) με συνολικό χρόνο εκτέλεσης λιγότερο από 7 λεπτά (μια ανέφικτη διαδικασία από ένα μόνο μηχάνημα λόγω των υψηλών απαιτήσεων, υπολογιστικών και μνήμης). Η διαδικασία CJ παρουσιάζει κάπως χαμηλότερη κλιμακωσιμότητα σε σύγκριση με την SJ ($x5.66$ στις 8 EC2 VMs) με μεγαλύτερο συνολικό χρόνο εκτέλεσης (περίπου 30 δευτερόλεπτα περισσότερο για ένα σύνολο δεδομένων $300MB$), ο οποίος οφείλεται στην επιβάρυνση από τη χρήση δύο αντί της μιας MapReduce διεργασίας. Από την άλλη, ένα ποιοτικό πλεονέκτημα της διαδικασίας CJ (σε σύγκριση με την SJ) είναι ότι προσφέρει μια γρήγορη προεπισκόπηση των αποτελεσμάτων της ανάλυσης.

Μια ακόμη σημαντική συνεισφορά αυτής της διατριβής είναι η εκτενής αξιολόγηση

των θεμάτων διαμόρφωσης (configuration and tuning), μια διάσταση η οποία συχνά παραβλέπεται ή δεν μελετάται επαρκώς, η οποία όμως είναι κρίσιμη για την επίδοση και την καλή χρησιμοποίηση των πόρων γενικότερα. Δείξαμε ότι οι προτεινόμενες διαδικασίες υπολογισμού χρησιμοποιούν βέλτιστα τους υπολογιστικούς πόρους (πλήρης χρησιμοποίηση των διαθέσιμων CPU, αποτελεσματική κατανομή μνήμης), και ότι δεν υπάρχει κάποια αδικαιολόγητη επιβάρυνση (π.χ. στη συλλογή απορριμμάτων (GC), άσκοπα ξεκινήματα/τερματισμοί των JVMs, ποσοστό «ανισορροπίας» μεταξύ του χρόνου ολοκλήρωσης των τελευταίων διαδικασιών, κ.α.).

Ευχαριστίες

Πρωτίστως θα ήθελα να ευχαριστήσω θερμά τους γονείς μου, Γιώργο και Δήμητρα και την πολυαγαπημένη μου αδερφή Δάφνη, για την αμέριστη στήριξή που μου δείχνουν σε κάθε μου προσπάθεια και καταβάλλουν ό,τι είναι δυνατό για να επιτύχω τους στόχους μου.

Κατόπιν ευχαριστώ τον επόπτη καθηγητή μου κ. Γιάννη Τζίτζικα για την αγαστή συνεργασία, ορθή καθοδήγηση και ουσιαστική συμβολή του στην ολοκλήρωση της παρούσας διατριβής. Επίσης, θα ήθελα να ευχαριστήσω τον συνεπιβλέπων καθηγητή κ. Κώστα Μαγκούτη για την εξαίρετη συνεργασία και τη σημαντική συμβολή του στην καθοδήγηση και ολοκλήρωση της διατριβής. Επιπλέον, να παραθέσω τις ευχαριστίες μου στον καθηγητή κ. Δημήτρη Πλεξουσάκη για την προθυμία του να συμμετέχει στην τριμελή επιτροπή.

Ακόμα να ευχαριστήσω το Ινστιτούτο Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας για την υποτροφία που μου προσέφερε καθ΄ όλη τη διάρκεια της μεταπτυχιακής μου εργασίας, όπως και για την πολύτιμη υποστήριξη σε υλικοτεχνική υποδομή και τεχνογνωσία.

Τέλος, ένα μεγάλο ευχαριστώ στους φίλους μου, καθώς επίσης και στους συνεργάτες μου στο Εργαστήριο Πληροφοριακών Συστημάτων, για την υποστήριξη και για τις όμορφες στιγμές που μοιραστήκαμε.

στους γονείς μου

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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 a deeper analysis of the search results is to perform name entity mining (for short *NEM*).

Entity search engines aim at providing the user with entities and relationships between these entities, instead of providing the user with links to web pages. Although this is an interesting line of research and there are already various entity search engines and approaches [1, 2, 3], according to our opinion, these approaches/tools are still in their infancy in the sense that they are not really useful for the common information needs of the users.

Several user studies [4, 5, 6] 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 [7].

For those reasons, instead of radically changing the way users search for information, we propose a method [8] for enriching the classical interaction scheme of search systems (keyword search over Web pages), with *(Named) Entity Mining* (or *NEM*). Furthermore, we specify criteria that determine the selection and ranking of the (often numerous) identified entities and we propose linking the identified entities with structured information about them that can reside in different sources (e.g. in the Linked-Open Data (LOD) cloud, accessible through SPARQL endpoints). Apart from identifying various ways for enriching the results of keyword search systems with entity mining, we focus on providing this service over the *textual snippets* of the search results at *query time*, i.e., without any pre-processing.

From an information integration point of view we could say that entity names are used as the "glue" for automatically connecting documents with data (and knowledge). This approach does not require deciding or designing an integrated

schema/view (e.g. [9]), nor mappings between concepts as in knowledge bases (e.g. [10, 11]), or mappings in the form of queries as in the case of databases (e.g. [12]). The key point is that entities can be identified in documents, data, database cells, metadata attributes and knowledge bases.

In this thesis we consider a general scenario where these services are provided as *meta-services*, i.e. on top of systems that support keywords search (in our implementation any system that supports OpenSearch [13] can be straightforwardly be used). Figure 1.1 illustrates the process that we consider. The initial query is forwarded to the underlying search system(s) 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 ranked and enriched with semantic descriptions derived by querying external SPARQL endpoints. The user can gradually restrict the derived answer by clicking on the entities.

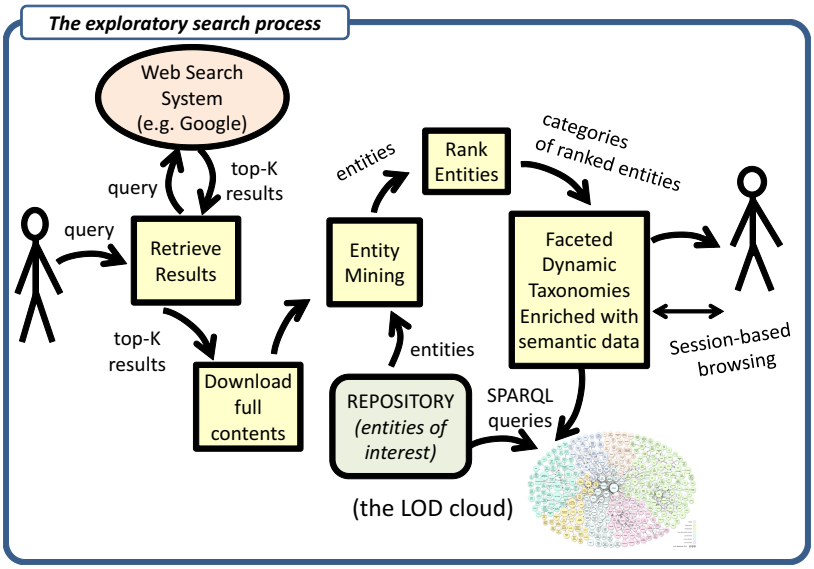


Figure 1.1: The exploratory search process

The process described above is dynamic in nature: the user can set up the desired underlying search system, the desired kind of entity types 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 1.2 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

Figure 1.2: An indicative screendump from a NEM process

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 LOD cloud. Furthermore, the system, after user's request, can apply mining over a desired hit and discover all entities of that hit.

In a nutshell, in this thesis:

(a) we detail a novel combination of *NEM* technologies for enriching the classical web (meta) searching process with entity mining performed at query time, where the mined entities are exploited for offering faceted exploration, (which is a type of knowledge base , or *browsable summary*),

(b) we outline the centralized algorithm for the *NEM* process,

(c) we elaborate on the ranking of entities and we report the results of a comparative evaluation with users,

(d) we compare the results of *NEM* over document snippets versus *NEM* over the full document contents according to various perspectives (mined entities, computational cost),

(e) we show that applying *NEM* over the *textual snippets* of the top hits, where the snippet of a hit consists of 10 to 20 words, produces results in real time,

(f) we show that extending *NEM* (without pre-processing) to the full contents of the top-hits however, is either infeasible (due to the high computational and

memory requirements) or can take hours to complete even for hundreds of hits and thus we aim for a scalable methodology by exploiting MapReduce distributed computation model to parallelize the *NEM* process,

(g) we outline two ways for distributing the *NEM* process onto MapReduce tasks and examine ways to efficiently execute those tasks on the Apache Hadoop MapReduce platform on EC2,

(h) we provide the required algorithms and discuss their requirements (e.g. in terms of exchanged data),

(i) we execute the resulting tasks in a large public Cloud environment (the Amazon Elastic Compute Cloud (EC2)),

(j) we establish analogies with other tasks (e.g. inverted index construction) and report the difficulties that we encountered,

(k) we report the factors that affect performance and how to tune them for optimal resource utilization, and

(l) finally we report extensive and comparative experimental results.

It is important to note when the application performs *NEM* over the full contents, 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. In fact, performing entity mining on several thousand hits (the scale of queries considered in this thesis) using the sequential *NEM* procedure exceeds the capabilities of any single Cloud VM, eventually leading to a crash. Finally, its parallelization requires methods that can balance the load (key to scalability) despite the fact that the sizes of the downloaded contents are not known a-priori. 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. One of our key contributions in this thesis is to thoroughly evaluate the parameter space of the underlying platform and to explain how to best tune it for optimal execution. We believe that the methods and results of this thesis are applicable to the parallelization and efficient execution of other related applications.

The rest of this thesis is organized as follows. In Chapter 2 we discuss the motivation, context, and related work. In Chapter 3 we describe the centralized task (architecture, entity-ranking, LOD-based enrichment), and in Chapter 4 we show how it can be logically decomposed and expressed as MapReduce functions. In Chapter 5 we detail the implementation of the MapReduce tasks and in Chapter 6 we report experimental results and in Chapter 7 we present the applications/prototypes that we have designed and developed. Finally, in Chapter 8 we provide our conclusions and discuss future work.

Chapter 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 [14] show that categorizing the search results improves the search speed and increases the accuracy of the selected results. The user study [15] 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 [6] and [5], *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 [16]. The user study [4] 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 the simple relevance ranking. As another example, in professional *patent search*, in many cases one has to look beyond keywords to find and analyse patents based on a more sophisticated understanding of the patent's content and meaning [17]. 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 [18]. Analogous results have been reported for search over collections of *structured* artifacts, e.g. ontologies. For instance, [19] 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. [20] comparatively evaluates *clustering* versus *diversification* services.

2.1.2 Past Work on Entity Mining over Search Results

There are several approaches that could be used in order to enrich the classical web searching with *NEM*. Some of them are described below.

$\mathcal{RS}$**: Real-time *NEM*** over the **Snippets** of the **top hits** of the answer.
Here entity mining is performed only over the *snippets* of the top hits of the returned answer.

$\mathcal{QC}$**: Query-time *NEM*** over the **Contents** of the **top hits** of the answer.
Here the full contents of the top hits of the returned answer are downloaded and then entity mining is performed. Clearly, this process can take much more time than $\mathcal{RS}$.

$\mathcal{OC}$**: Off-line *NEM*** over the **entire Corpus**.
Here we mine all entities of the corpus *offline* (assuming that the corpus is available), and we build an appropriate index (or database) for using it at run time. For each incoming query, the entities of the top-K (e.g. $K = 100$) hits of the answer are fetched from the index, and are given to the user. An important observation is that the size of the entity index in the worst case could be in the scale of the corpus. Also note that this approach cannot be applied at meta (uncooperative) search level.

$\mathcal{OFQ}$**: Offline *NEM*** over the **top hits** of the answers of the **Frequent Queries**. Here, also *offline*, for each frequent query of the log file (e.g. for those which are used for query suggestion), we compute its answer, we fetch the top-K hits, we apply *NEM* and save its results as they should be shown (i.e. what the left bar should show) using the approach and indexes described at [21, 22]. The benefit of this approach in comparison to $\mathcal{OC}$ is that here we do not have to apply *NEM* at the entire collection but only at the top hits of the most frequent queries. This significantly reduces the required computational effort and storage space. The downside of this approach is that if a user submits a query which does not belong to the frequent queries, and thus it has not been processed, then the system cannot offer entities. In that case the system could offer the choice to the user to apply *NEM* at query time, i.e. approach $\mathcal{RS}$ or $\mathcal{QC}$ as described earlier. Finally, we should note that this approach is applicable also at a meta search level, but periodically the index has to be refreshed (mainly incrementally).

There is a plethora of related works and systems that offer a kind of *entity search*. We should clarify that the various *Entity Search Engines* are not directly related to this thesis, 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. Below we describe in brief a few of them.

The *Entity Search Engine* [1, 23, 24] supports only two categories (phone and email) and users have to type formatted queries (using # to denote entities). *NEM* is applied over the entire corpus and the extracted entities are stored in the form of ordered lists based on document ID (much like storing inverted indices for keywords), in order to provide their results instantly.

EntityCube [25] is an entity search engine by Mircosoft which extracts entities and relationships from semi-structured as well as natural-language Web sources. The goal is to automatically construct and maintain a knowledge base of facts about named entities, their semantic classes, and their mutual relations as well as temporal contexts.

MediaFaces [2, 26] provides faceted exploration of media collections and offers a *machine learned* ranking of entity facets based on user click feedback and features extracted from three different ranking sources. For a given entity of interest, they have collected (from knowledge bases like Wikipedia and GeoPlanet) a large pool of related candidate facets (actually related entities).

Google's Knowledge Graph [27] 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.

In addition, works on *query expansion* using lexical resources (thesauri, ontologies, etc), or other methods that exploit named entities for improving search (e.g. [3, 28]) are out the scope of this thesis, since we focus on (meta-)services that can be applied on top of search results.

With respect to the approaches $\mathcal{RS}$, $\mathcal{QC}$, $\mathcal{OC}$ and $\mathcal{OFQ}$ described earlier, most systems follow approach $\mathcal{OC}$, while the only system that offers $\mathcal{OFQ}$ is [22]. To the best of our knowledge the current work is the first that investigates the $\mathcal{RS}$ approach; and the $\mathcal{QC}$ by exploiting the scalability of the MapReduce framework over distributed Cloud computing resources.

2.2 MapReduce and Summarization of Big Data

MapReduce [29, 30, 31, 32] 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 [33, 34, 35, 36, 37], impacting a wide spectrum of areas like information retrieval [33, 38, 39, 40], scientific simulation [33], image processing [32], distributed co-clustering [41], latent dirichlet allocation [42], nearest neighbours queries [43], and the Semantic Web [44] (e.g. from storing/retrieving the increasing amount of RDF data[1][34] to distributed querying [37] and reasoning [35, 36]).

Previous work by Li et al. [45] 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 [46]) over which other data analytics tasks can be executed in a more scalable and efficient manner (e.g. see [47]).

This thesis 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

[1] 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.

(NLP) community for nearly the last half century (see [48, 49] 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 Section 3.4, 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 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 [48].

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* [50, 51]. 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.

Chapter 3

The Centralized Process

Recalling the exploratory search process described at a high level and depicted in Figures 1.1 and 1.2, we will now describe a centralized (non-parallel) version of the process in more detail. The process consists of the following general steps:

- Get the top *HK* (e.g. *HK*=200) results of a (keyword) search query
- Download the contents of the hits and mine their entities
- Group the entities according to their categories and rank them
- 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
- 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 [7] 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 [52]). Therefore we do not further consider them in this thesis.

In what follows we first introduce notations and elaborate on the ranking of entities (Section 3.1), we show how to exploit the Linked Open Data (Section 3.2), and then describe the steps of the above process in more detail (Section 3.3), and distinguish various levels of functionality (Section 3.4). Next, in Chapter 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 $c \in C$.

Entity Ranking

Entity selection and ranking is important since usually the UI has limited space therefore only a few can be shown at the beginning. We propose tackling this problem by (a) ranking all identified entities for deciding the top-10 entities to be shown for each category, and (b) offer to user the ability to show more entities (all) on demand. Below we focus on ranking methods that do not rely on any log analysis, so they are aligned with the dynamic nature of our approach.

We need a method to rank the elements of E_c. One approach is to count the elements of A in which the entity appears, i.e. its *frequency*. Furthermore, we can 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 can capture this requirement by a formula of the form:

$$Score_{rank}(e) = \sum_{a \in docs(e)} ((|A| + 1) - rank(a)) \tag{3.1}$$

We can see that an occurrence of e in the first hit, counts $|A|$, while an occurrence in the last document of the answer counts for 1.

Another approach is to take into account the *words* of the entity name and the query string. If for an entity $e \in E$, we denote by $w(e)$ the words of its name, and by $w(q)$ the words of the query, then we can define $Score_{name}(q, e) = \frac{|w(q) \cap w(e)|}{|w(e)|}$. To tolerate small differences (due to typos or lexical variations), we can define an alternative scoring function that is based on the *Edit Distance*:

$$Score_{nameDist}(q, e) = \frac{|\{\ a \in w(q) \mid \exists b \in w(e), EDist(a, b) \leq 2\}|}{|w(q)|} \tag{3.2}$$

which returns the percentage of the words of q which can be "matched" to one word of the entity e either exactly or up to an Edit distance equal to 2.

The above scores can be combined to reach a final score that considers both perspectives. We can adopt the *harmonic mean* for promoting those entities which have high scores in both perspectives. However notice that if an entity has not any query word (or a word that is close to a query word), that entity would take zero at $Score_{name}$ and that would zero also the harmonic mean. One approach to tackle this problem is to compute the plain (instead of the harmonic) mean, or in place of $Score_{nameDist}(q, e)$ to have the sum $Score_{nameDist}(q, e) + b$ for a very small positive constant b (e.g. $b = 0.01$).

$$Score(q, e) = \frac{2\ Score_{rank}(q, e)\ Score_{nameDist}(q, e)}{Score_{rank}(q, e) + Score_{nameDist}(q, e)} \tag{3.3}$$

3.2 On Exploiting Linked Open Data

In this Section we provide some information on how to exploit the *Linked Open Data (LOD)* in order to enrich the identified entities with more information; a procedure that was proposed at [8].

There are already vast amounts of structured information published according to the principles of (LOD). The availability of such datasets enables the enrichment of the identified entities with more information about them. In this way the user not only can get useful information about one entity without having to submit a new query, but he can also start browsing the entities that are linked to that entity.

Another important point is that exploiting LOD is more dynamic, affordable and feasible, than an approach that requires each search system to keep stored and maintain its own knowledge base of entities and facts. Returning to our setting, the first rising question is which LOD dataset(s) to use. One approach is to identify and specify one or more appropriate datasets for each category of entities. For example, for entities in category "Location", the *GeoNames* [53] dataset is ideal since it offers access to millions of placenames. Furthermore, *DBpedia* [54] is appropriate for multiple categories such as "Organizations", "People" and "Locations". Other sources that could be used include: *FreeBase* [55] (for persons, places and things), *YAGO [56]* (for Wikipedia, WordNet and GeoNames). In addition *FactForge*[57] includes 8 LOD datasets (including DBpedia, Freebase, Geonames, UMBEL, Wordnet). DBpedia and FactForge offer access through SPARQL endpoints [58, 59].

Running one (SPARQL) query for each entity would be a very expensive task, especially if the system has discovered a lot of entities. Some possible ways to tackle this problem are: (a) offer this service on demand, (b) for the frequent queries pay this cost at pre-processing time and exploit the results as described in [21, 22], (c) periodically retrieve and store locally all entities of each category, so at real time only a matching process is required (however here we have increased space and maintenance requirements). Note however that approach (c) is essentially the approach of our prototype (even though no Linked Data are used), since the Gazetteers of *GateAnnie* that we use include names of persons (11,974), organizations (8,544), and locations (29,984); in total about 50,502 names are used in our setting. Furthermore, lists of prefixes and postfixes are contained that aid the identification of entities (e.g. from a phrase "Web *inventor* Tim Berners-Lee", it recognizes "Tim Berners-Lee" as a *person* due to the prefix "inventor"). So the essential difference could be the following: instead of having a *NEM* component that contains predefined named lists/rules, it is beneficial (certainly from an architectural point of view) to offer the ability to the system to download the required lists (from the constantly evolving LOD) that are appropriate for the application at hand.

3.2.1 Case Study: Fisheries and Aquaculture publications

Apart from the case of general purpose Web searching, we have started investigating this approach in vertical search scenarios. One of this is the domain of *FAO* (Food and Agriculture Organization) publications about *fisheries and aquaculture*. The underlying keyword search system is the *FIGIS search component* [60] which can receive queries through an HTTP API. The search result apart from formatted HTML can be returned in XML format which uses Dublin Core schema to encapsulate bibliographic information. Each returned hit has various textual elements, including publication title and abstract. The first is around 9 words, the second cannot go beyond 3,000 characters. As concern *NEM*, we identified the following relevant categories: *Countries, Water Areas, Regional Fisheries Bodies*, and *Marine Species*. For each one there is a list of entities: 240 countries, 28 water areas, 47 regional fisheries bodies and 8,277 marine species, in total 8,592 names. Each such entity is also described and mutually networked in the *Fisheries Linked Open Data (FLOD)* RDF dataset. FLOD extended network of entities is exposed via a public SPARQL endpoint and web based services.

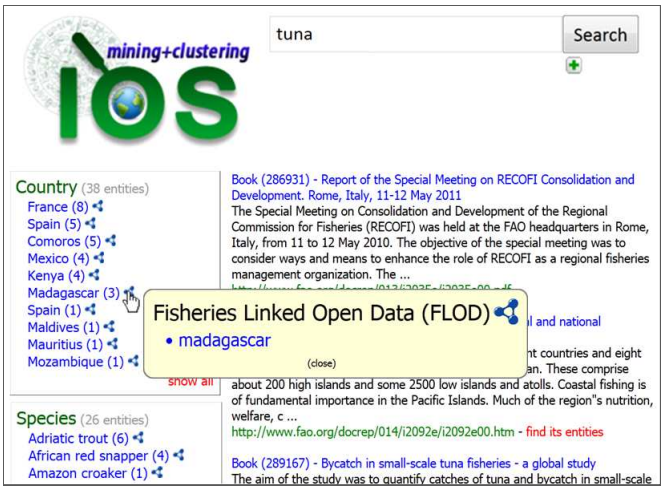


Figure 3.1: A prototype over FAO publications with links to FLOD

The objective is to investigate how to enrich keyword search with entity mining where the identified entities are linked to entities in FLOD endpoint, and from which semantic description can be created and served. A screendump of this prototype is shown in Figure 3.1.

3.3 The Centralized Algorithm

Using the notation introduced above, a centralized (non-parallel) algorithm for the general exploratory search process (steps 1-5) described in Chapter 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 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.*

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}
```

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

3.4 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.

Chapter 4

Parallelization

In this section we describe a parallel version of Algorithm 1 and then adapt it to the MapReduce framework (Section 4.1). Note that our exposition here focuses on algorithmic issues. We will describe our MapReduce implementations in Chapter 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. $A_1 \cup \ldots \cup A_z = A$, 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 this 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.

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.

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** $(e, c) \in outcome$ **do**
6: $AC = AC \cup \{c\}$ ▷ 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: $e.doclist(c) \cup = \{A[i]\}$ ▷ Update the (e, c) doclist
10: **return** $\{(c, e, e.count(c), e.score(c), e.doclist(c)) \mid c \in AC, e \in c.elist\}$

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.

Algorithm 3 Aggregation Function for all categories

1: **function AggregateSubTask**(...)
2: Concatenate the results of all SubTasks in a table TABLE
3: $AC = \{\ c \mid (c, *, *, *, *) \in TABLE\}$
4: **for all** $c \in AC$ **do** ▷ For each (active) category
5: $c.entities = \{\ e \mid (c, e, *, *, *) \in TABLE\}$
6: **for all** $e \in c.entities$ **do**
7: $e.count(c) = \sum\{\ cnt \mid (c, e, cnt, *, *) \in TABLE\}$
8: $e.score(c) = \sum\{\ s \mid (c, e, *, s, *) \in TABLE\}$
9: $e.doclist(c) = \cup\{\ dl \mid (c, e, *, *, dl) \in TABLE\}$
10: $c.elist$ = top-EK entities after sorting wrt $*.score(c)$
11: **for all** $e \in c.elist$ **do** ▷ LOD-based sem. enrichment
12: **if** $e.semd$ =empty **then**
13: $e.semd$=queryLOD(e)
14: **return** $\{(c, e, e.count(c), e.score(c), e.doclist(c), e.semd) \mid c \in AC, e \in c.elist\}$

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 **Figure 1.2**). 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 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).

A concise version of Algorithm 3 (`AggregateSubTask`) that exploits Algorithm 4 (`AggregateByCategory`) is given in Algorithm 5 (`AggregateSubTaskConcise`).

Algorithm 4 Aggregation Function for one category

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

Algorithm 5 Aggregation Function for all categories (Concise version of Alg. 3 that exploits Alg. 4)

1: **function** AggregateSubTaskConcise(...) ▷ ..
2: Merge the results of all SubTasks in a table TABLE
3: $AC = \{\ c \mid (c, *, *, *, *) \in TABLE\}$
4: **for all** $c \in AC$ **do** ▷ For each (active) category
5: AggregateByCategory(c)
6: **return** $\{(c, e, e.count(c), e.score(c), e.doclist(c), e.semd) \mid c \in AC, e \in c.elist\}$

4.1 Adaptation for MapReduce

In this section we cast the above algorithms in the MapReduce programming style, whose key concepts were introduced in Section 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 $nem : A \to E \times C$. Using MapReduce, the mapping phase partitions the set A to z blocks and assigns to each node i the responsibility of computing a function $nem_i : A_i \to E \times C$. The original function nem can be derived by taking the union of the partial functions, i.e. $nem = nm_1 \cup \ldots \cup nm_z$. Mapper tasks therefore carry out the downloading and mining tasks for their assigned set A_i. Note that the partitioning $A \to \{A_1, \ldots, A_z\}$ should be done in a way that balances the work load. Methods to achieve this will be described in Chapter 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:

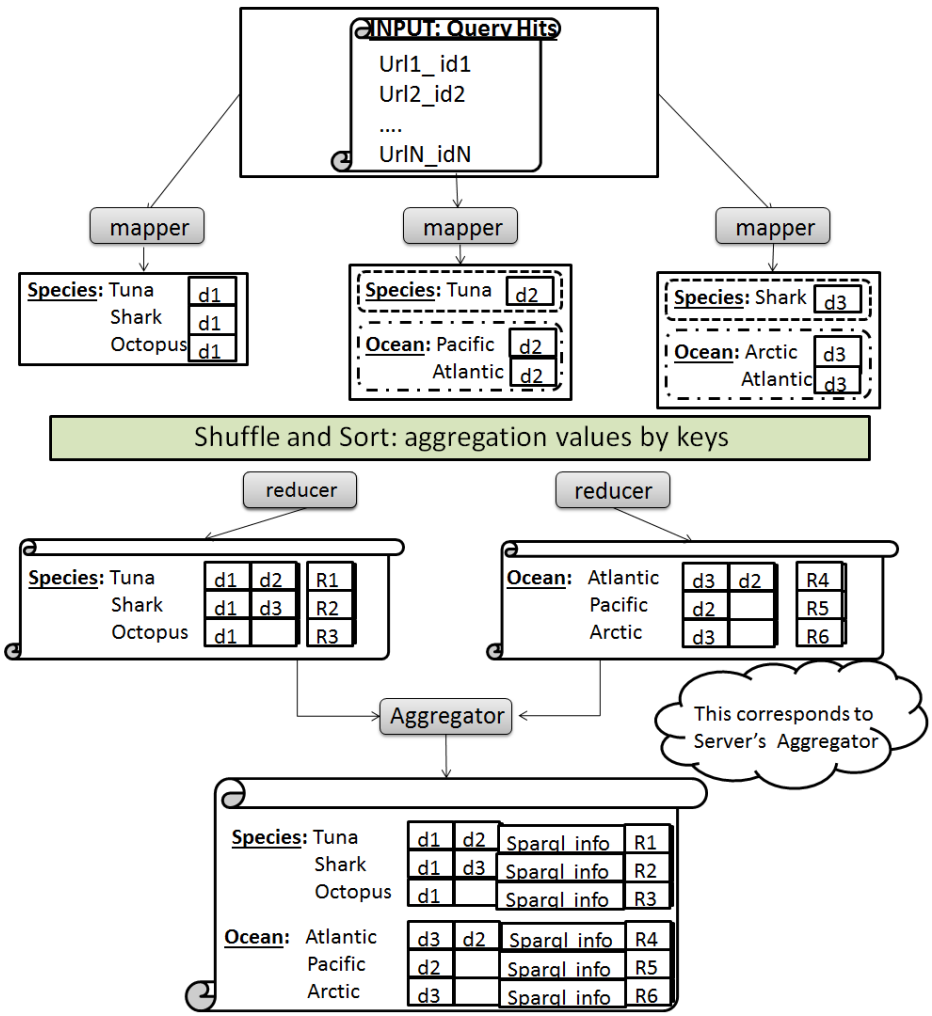


Figure 4.1: Example of distributed *NEM* processing using MapReduce

Algorithms		MapReduce functions
Alg. 2	DoSubTask	Map
Alg. 2	return	emit (with key c, value the rest parts of the tuples)
Alg. 4	AggregateByCategory	Reduce(params: key c, and value the rest parts of tuples)
Alg. 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 4.1 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 it 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 $\mathcal{O}(|A|d_{asz})$. 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 Section 3.4) 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 [33] is that our task is more CPU and memory intensive. Besides the cost of initializing the *NEM* component (described in more detail in Section 5.2.1), entity mining requires performing lookups, checking rules, running finite state algorithms etc.

Chapter 5

Implementation of Parallelized Process

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 [61] version 1.0.3, an open-source Java implementation of the MapReduce [30] 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 Figure 5.1. 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 [62]. Therefore we consider that the single-job procedure is practically oblivious to a-priori knowledge of URL content sizes.

Whether we have approximate (single-job procedure) or accurate (chain-job procedure described in Section 5.2.2) content sizes, we split work to tasks as follows: First, we sort documents in descending order (based on 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 Section 6.2.5. When not stated otherwise we use a target split size of 1.5MB. Our process repeatedly pops the top of the stack and inserts it to the split 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 (Figure 5.1) a number of *mapper* tasks are created on a number of JVMs hosted by Cloud VMs. JVM configuration

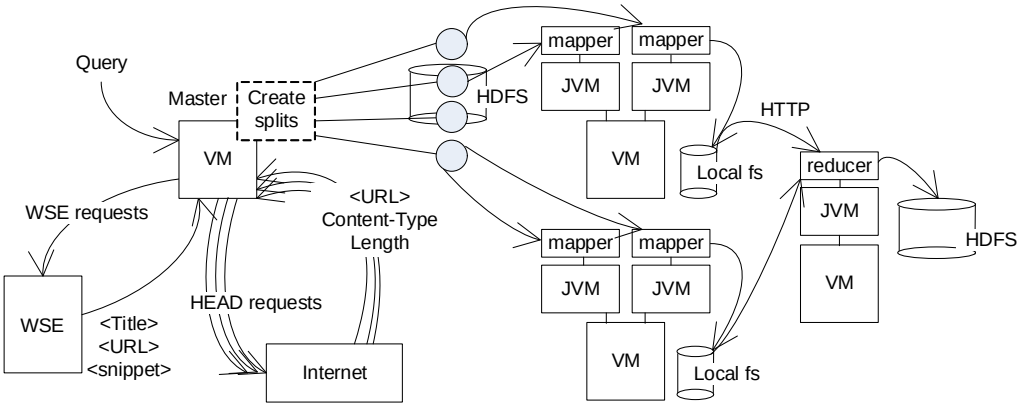


Figure 5.1: Single-job design.

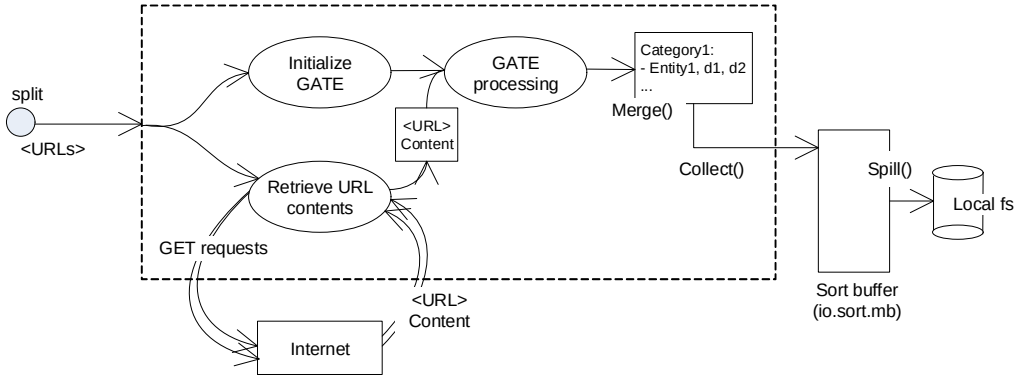


Figure 5.2: Single-job mapper.

and number of JVMs per VM are key parameters that are further discussed in Sections 5.3 and 6.2. The operation of each mapper is depicted in Figure 5.2. Besides the creation of appropriate-size splits, we are taking care to determine the order of task execution for optimal resource utililization. Taking into account the fact that our datasets typically include a few large documents (Section 6.2.2) we schedule the corresponding long tasks early in the job to increase the degree of overlap with other tasks.

We use the GATE [63, 64] component for performing *NEM* processing. GATE relies on finite state algorithms and the JAPE (regular expressions over annotations) language [65]. 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 (Section 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* [66] to merge the results from multiple mappers operating on the same node.

The reduce phase performs the merging of mapper outputs per category and computes the scores of the different entities (Section 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 [67] jobs (Jobs #1 and #2) as shown in Figure 5.3. 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.

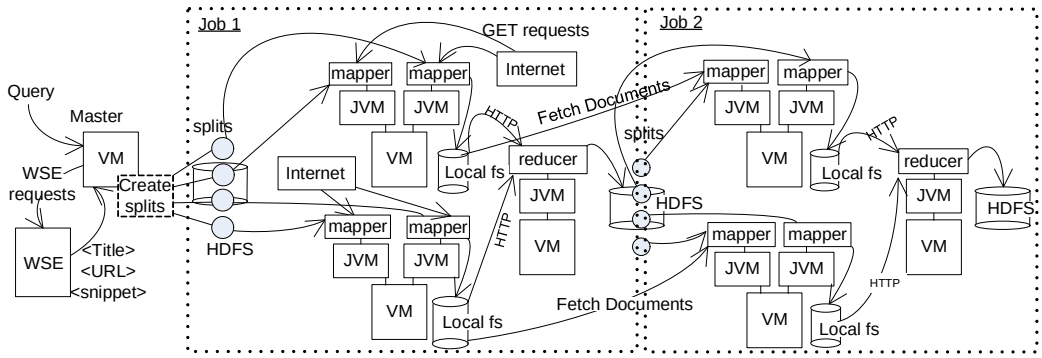


Figure 5.3: Chain-job design.

In Figure 5.3, 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 (Figure 5.4) 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. 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 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 Section 6.2.4.

Job #2 features mappers (Figure 5.5) 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 [51]. 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 Amazon 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

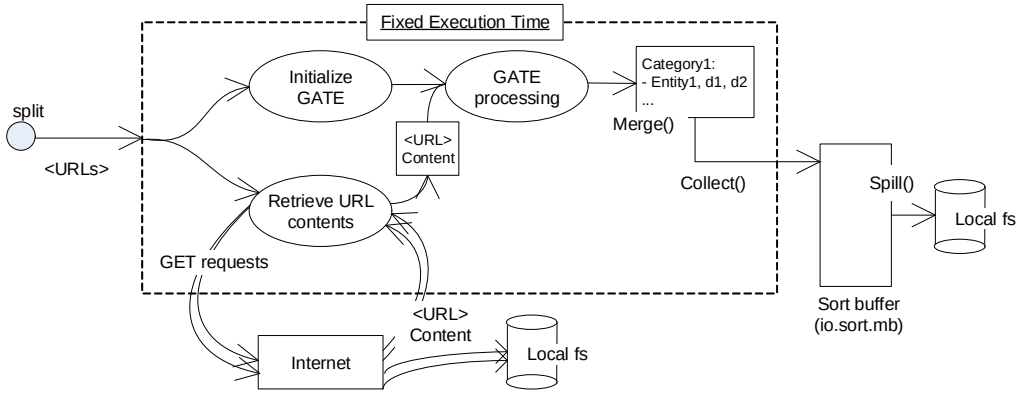


Figure 5.4: Chain-job mapper #1 (preview analysis).

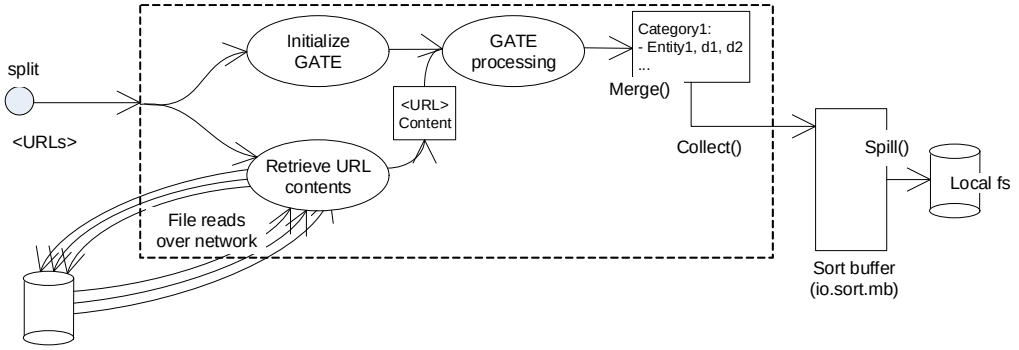


Figure 5.5: Chain-job mapper #2 (full analysis).

executed within the MapReduce job. For example, if the total size of hits that we want to analyse is 12MB and the split set to 2MB 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 (Section 6.2.5).

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 [66, 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 (Figure 5.2) to avoid repeated spills and subsequent merge-sort operations. The size of the buffer (controlled by the `io.sort.mb` parameter) defaults to 100MB. 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*) 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 this:

	Reusability: R				
Split Size (MB) \ 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	✗	✗	✗	$time_8$	$time_9$

Table 5.1: Execution time varying split size and heap size with fixed reusability R (✗ means the job failed).

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:

$$\text{JVM heap memory} = \frac{\text{Total VM memory available}}{JpV}. \tag{5.1}$$

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 5.1 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 Section 6.2.7.

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, Equation 5.1). 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 Section 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 [66] 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 256MB 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 300MB input dataset produces about 24MB 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 [68] 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, Section 6.2.5), 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

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

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

Chapter 6

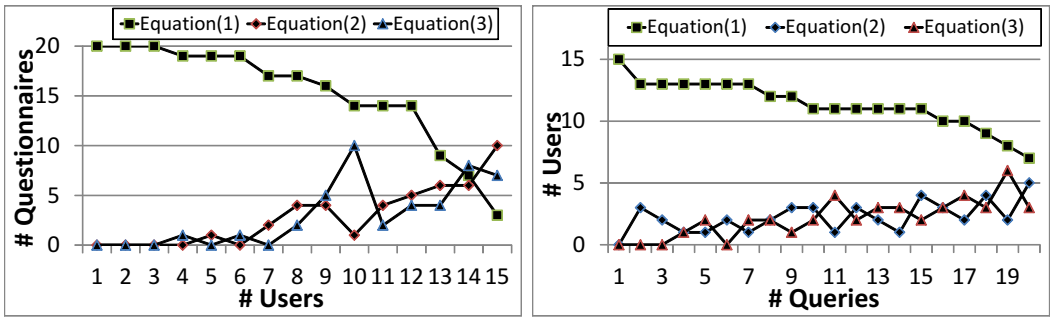
Evaluation and Experimental Results

6.1 Centralized Process

At section 6.1.1 we report the results of a comparative evaluation of the three *entity ranking methods* by users, while at section 6.1.2 we compare *snippet-mining* versus *contents-mining* from various perspectives.

6.1.1 Comparative Evaluation of Entity Ranking Methods by Users

We comparatively evaluated with users the *three* methods for entity ranking (equations (3.1), (3.2) and (3.3) of Section 3.1). Fifteen users participated in the evaluation with ages ranging from 20 to 28, 73.3% males and 26.6% females. We selected a set of 20 queries (given at Appendix A.1) and for each one we printed one page consisting of three columns, one for each ranking method. Each column was showing the top-10 entities for the categories *Person, Location, Organization*. *NEM* over full contents were used. We gave these pages to each participant and asked him/her to mark the most preferred ranking. If (s)he could not identify the most preferred, (s)he could mark two or even all three as equally preferred. We aggregated the results based on plurality ranking (by considering only the most preferred options). The results showed that the most preferred ranking is that of equation (3.1), which was the most preferred in 228 of the 15x20=300 questions. The equations (3.2) and (3.3) got almost the same preference, specifically they were selected as most preferred options in 43 and 44 of the 15x20=300 questions.
In more detail, Figure 6.1a shows that for 13 of the 15 participants, equation (3.1) is the most preferred, while Figure 6.1b shows that for each of the 20 queries, equation (3.1) is the most preferred.
From these we conclude that the string similarity between the query and the entity name did not improve entity ranking in our setting.

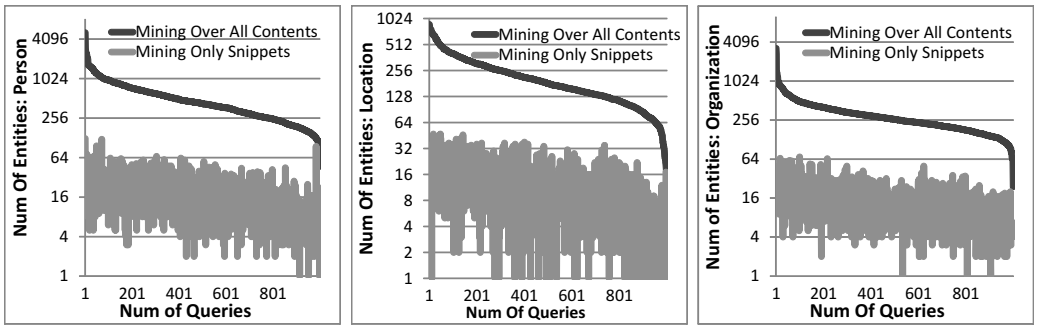


(a) (b)

Figure 6.1: Left: Aggregated preferences for each user. Right: Aggregated preferences for each query.

6.1.2 Contents Mining vs Snippet Mining

Since a snippet is a part of a document, the entities discovered in a snippet are *subset* of the entities discoverable at the document contents. From this perspective we could say that the results of snippet mining are "sound" w.r.t. the results of documents mining.



(a) (b) (c)

Figure 6.2: Comparing the number of mined entities (for categories Person, Location, Organization) over all contents and over only snippets for 1000 queries (for each query its top-50 hits are mined).

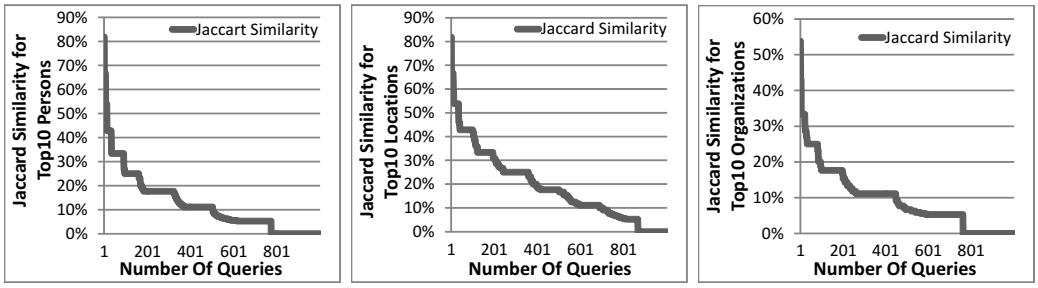
To check how different they are we performed various measurements. For a set of 1000 queries we compared the results of *snippet-mining* and *content-mining* over the top-50 hits of the query answers. In our experiments we considered only the categories *Person, Location* and *Organization*, and the results are shown in Figure 6.2a, 6.2b and 6.2c respectively (the y-axis is in log scale and the queries are ordered in descending order with respect to the number of mined entities over their full contents).

Figure 6.2a shows that the average number of identified *persons* over full contents is about 527, while the average number of identified persons over snippets

is about 18, meaning that content mining yields around 29 times more persons. We should also note that 50% of the queries return less than 500 entities, 43% of queries retrieve from 500 to 1000 entities and only 7% return more than 1000 entities.

In Figure 6.2b we observe the same pattern for *locations*: contents-mining in average returns about 219 entities per query while snippet-mining about 12 entities. Finally, according to Figure 6.2c, content mining identifies on average 309 organizations while snippet-mining 14 organizations (22 times less).

To sum up, we could say that content mining yields around 20 times more entities than snippet mining.



(a) (b) (c)

Figure 6.3: Jaccard Similarity between top-10 mined entities (for the categories `Person, Location, Organization` over snippet and over full contents for 1000 queries (for each query answer the top-50 hits are considered).

In addition, we compared only the top-10 mined entities as produced by equation (3.1). We compared these lists as sets using the Jaccard similarity. The results are shown in the three diagrams of Figure 6.3. We observe that the entities in the category "Person" have Jaccard similarity 0% for the 65% of queries and more than 20% for less than the 10% of queries. For the categories "Location" and "Organization", about half of the queries have 0% similarity. Furthermore, for entities in "Organization" there are not queries with Jaccard similarity more than 50%, while for entities in "Location" no queries have more than 70% similarity. In general, we can conclude that the top-10 entities of snippet mining and contents mining for the same queries (i.e. the same collection of results) have many differences and only few entities are the same. This is a predictable result since (as we saw) mining of contents yields about 20 times more entities than snippets mining.

Computational and Memory Costs

All experiments were carried out using a laptop with processor Intel Core 2 Duo T8300 @ 2.40Ghz CPU, 4GB RAM and running Ubuntu 10.04 (32 bit), and Google was used as the underlying engine. The implementation of the system is in Java 1.6 (J2EE platform), using Apache Tomcat 7 (2GB of RAM).

Snippet-mining *Time:* The whole procedure over the top-50 snippets (each having the size of 193 bytes in average) for one query takes in average 1.5 seconds. The whole procedure comprises the following steps. At first we retrieve the results pages from the underlying WSE which costs about 547 ms (36,2% of the total time). Second, we apply *NEM* over the snippets of the retrieved query's answers which takes about 901 ms (60,4% of the total time). Third, we apply *NEM* over the query string which costs about 5,6 ms (0,37% of the total time). Finally, we create a string representation of the first page of results with cost about 36 ms (2,4% of the total time) and also a string representation of all entities in about 9 ms (0,6% of total time). The time for ranking entities and categories is negligible (less than 1 ms).In more detail, and only for the *NEM* task, some indicative times follow: 0.2 secs for 10 snippets of total size 0.1 MB, 1.2 secs for 100 snippets of total size 1.94 MB.
The average main memory requirements for one query (for the whole process) is about 37MB.

Content-mining *Time:* The whole procedure over the top-50 full documents (of total size about 6.8 MB) for one query takes in average 78 seconds. The retrieval of results from the underlying WSE costs less than one second (1% of the total time). The downloading of the contents of each result costs about 28 seconds (36% of the total time). The application of *NEM* over the contents of the fetched documents takes about 45 seconds (57% of the total time). The creation of the string representation of the first page of results costs about 33 ms (0.04% of the total time) and also a string representation of all entities in about 4,5 seconds (6% of total time). The sorting of the categories and entities costs only a few ms. Some indicative times for *NEM* only: 5.2 seconds for 10 documents of total size 1.5 MB, 107 seconds for 100 documents of total size 16.3 MB.
The average main memory requirements for one query (the whole process) is about 300 MB.

Synopsis. The comparison between snippet and contents mining (over the top-50 hits) can be summarized as:

	$\mathcal{RS}$	$\mathcal{QC}$
entities per hit:	1.2	10.1
overall time:	1.5 secs	78 secs
main memory footprint for one query:	37 MB	300 MB.

6.2 Parallelized Process

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 Section 6.2.1 we outline sources of non-determinism in our system and ways to address them. In Section 6.2.2 we describe the procedure with which we create realistic synthetic datasets and in Section 6.2.3 we describe our experimental platform. From Section 6.2.4 on we focus on scalability and on the impact of different platform parameters to efficiency and high performance.

6.2.1 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 [1] 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 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 (Section 6.2.2), 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.

[1] 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.

6.2.2 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. 100MB). 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.

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 6.4 presents the distribution of sizes for xMB-SET1. The created datasets represent a range from 1226 (100MB) to 4365 (300MB) documents (hits) on average. Most documents are small: 89.5%, 93.1%, and 94.7% of the documents in the 100MB, 200MB, and 300MB datasets respectively are less than 200KB in size. The largest dataset (300MB) 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.

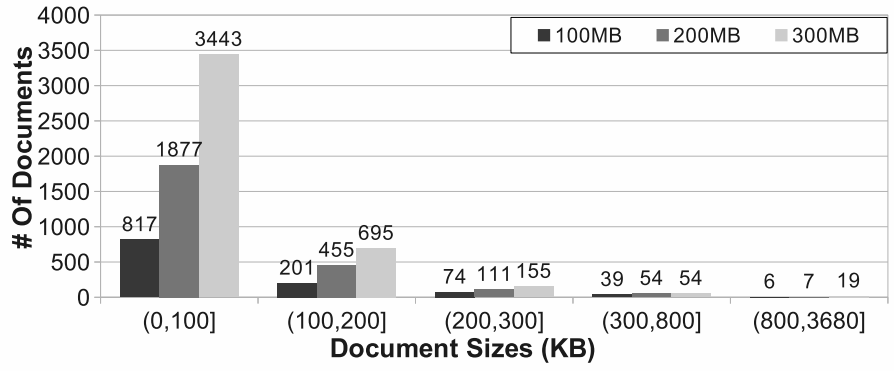


Figure 6.4: Distributions of sizes for xMB-SET1, $x \in \{100, 200, 300\}$

6.2.3 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 (2.2GB for a mapper, 256MB 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 Section 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 [69, 70], a scalable cluster monitoring tool that provides visual information on the state of individual machines in a cluster, along with the `sFlow` [71] plug-in to get the metrics for each JVM (e.g. mapTask, reduceTask). Ganglia is composed of two servers: the `gmetad` server, which provides historical data and collects current data and the `gmond` server which collects and servers current statistics. Ganglia also provides a web interface offering a graphical view of the cluster information. In our implementation we have set one `gmetad` (version 3.4.0) server with one Web Interface (version 3.5.4) in the same node and one `gmond` (version 3.4.0) to each node. Finally, to monitor the performance of the Java *garbage collector* we used the IBM Monitoring and Diagnostic Tool [72] to analyse JVM logs and tune the system appropriately.

6.2.4 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 100MB, 200MB, and 300MB (Section 6.2.2) 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 [73]. 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 6.5 depicts the execution time for the three different system configurations with increasing number of nodes (VMs). Tables 6.1-6.3 depict the speedups achieved in all cases. Our observations are:

- All systems exhibit good scalability, which improves with increasing dataset size. For the 300MB 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.

- 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 Section 6.2.5) that reduce the degree of parallelism in the map phase of the job. We have analyzed these components for a specific case, the 200MB-SET1 dataset in the SJ-HEAD configuration (Table 6.4). 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 3.68MB document, a size that far exceeds the vast majority of other documents in the set (Section 6.2.2)) 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 (3.68MB, 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, Section 5.2.1), tasks of that size in some cases create scheduling imbalances resulting in suboptimal scalability.

- The CJ speedup observed for the 300MB 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.

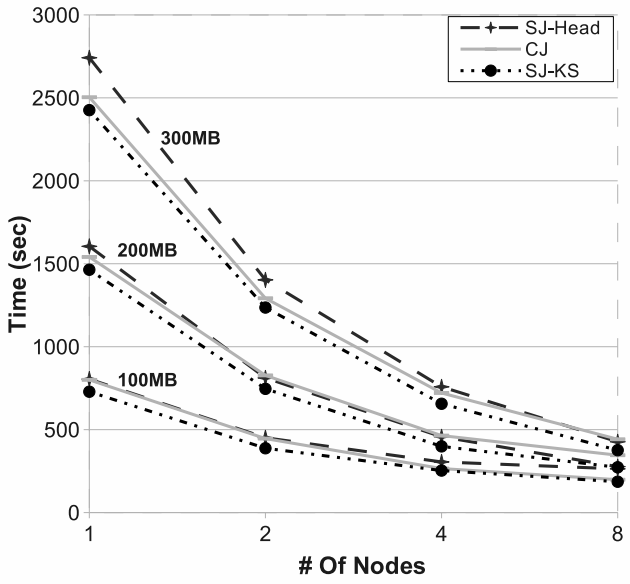


Figure 6.5: Query execution time for growing dataset size and increasing number of nodes (VMs).

# VMs	100MB	200MB	300MB
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 6.1: Speedup for SJ-HEAD

# VMs	100MB	200MB	300MB
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 6.2: Speedup for SJ-KS

# VMs	100MB	200MB	300MB
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

Table 6.3: Speedup for CJ

#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

Table 6.4: Detailed analysis: SJ-HEAD, 200MB-SET1

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 6.6 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 overview (summary) of the entire document list. Based on these results we conclude that Job #1 exhibits good scalability, and offers a powerful tradeoff to an administrator when aiming to improve the quality of overview: either allocate more VMs to Job #1 (costly but faster option) or allow more execution time on fewer VMs (cheaper but slower option).

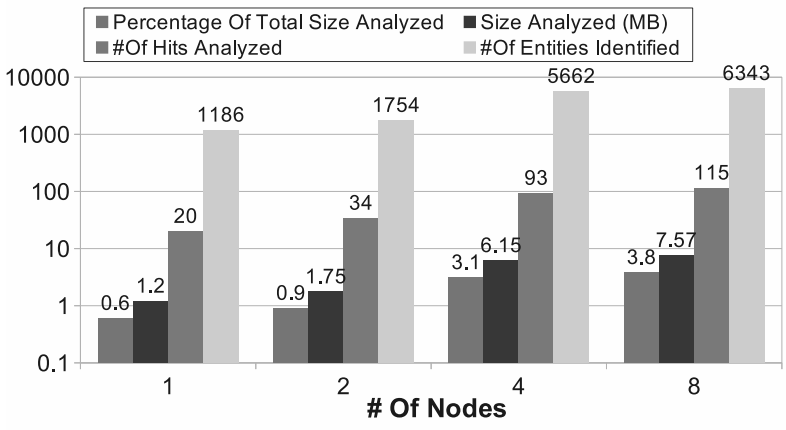


Figure 6.6: Analysis of chain-job #1 on *200*MB dataset, 1-min job execution time.

6.2.5 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 100MB-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.

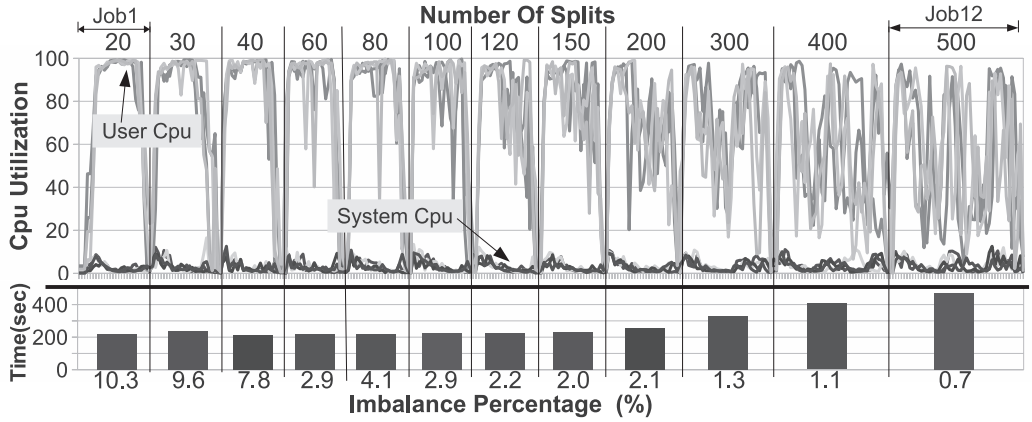


Figure 6.7: CPU utilization, execution time and imbalance percentage for a number of jobs whose only difference is the number of splits.

Figure 6.7 (top portion) depicts per-node CPU utilization for each job. Figure 6.7 (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 96.5s to 18.5s as shown in Figure 6.8. 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.

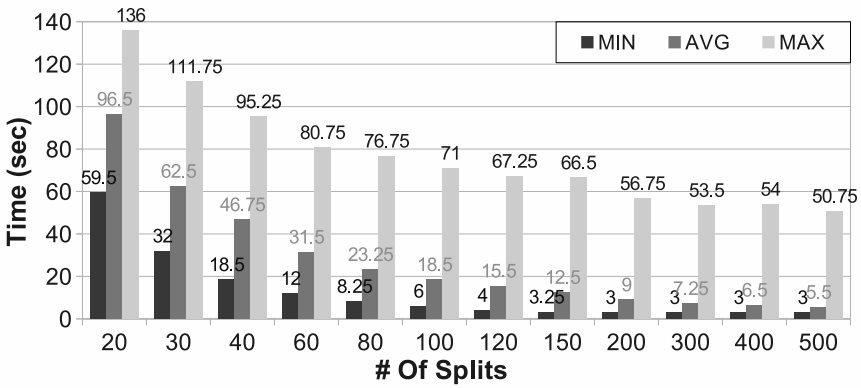


Figure 6.8: Map task min/average/max execution time for different number of splits.

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 Section 6.2.6) 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 Figure 6.8).

6.2.6 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 100MB-SET1 dataset instance. One of the jobs assigns 1GB of heap space to its JVMs whereas the other job assigns twice that amount (2GB). 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 6.9 presents overall execution time and GC activity (in MB) for the two jobs. We observe that Job 1 (1GB JVM heap) takes an additional 4.8 minutes (about 30%) to completion compared to Job 2 (2GB 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 (2GB).

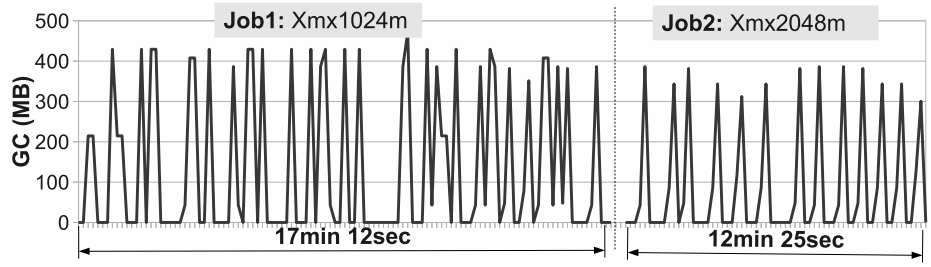


Figure 6.9: Garbage collection activity for two jobs that differ in the amount of memory allocated to JVMs (1GB (left) vs. 2GB (right)).

The impact of heap size can in fact be far more severe than presented above. In fact, standard heap allocations (default of 200MB in several JVMs) always result to job failure. Table 6.5 depicts measurements of overall execution time of jobs consisting of a single split, where the split size varies from 1MB to 35MB and JVM heap size varies from its default value of 200MB to 2.2GB. 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 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).

	Heap Size (in MB)								
Split 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

Table 6.5: Execution time (sec) of single-split jobs with varying split and heap sizes.

Table 6.5 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 Figure 6.9), up to a point where additional heap does not help: excessivelly 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 Section 5.3 for selecting key parameters of our MapReduce jobs.

6.2.7 Impact of JVM Reusability

In this section we first exhibit the performance advantage of JVM reusability by setting up two *NEM* jobs on the 100MB-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 6.10 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.

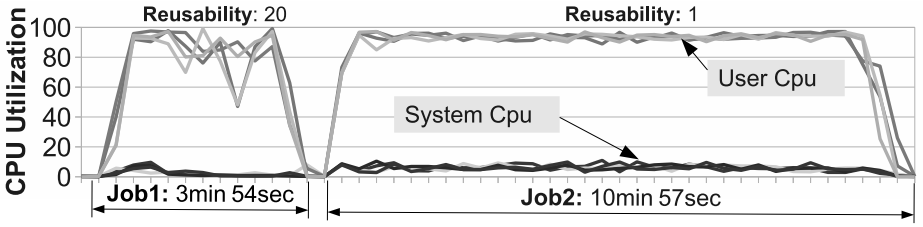


Figure 6.10: CPU utilization and job execution time for two jobs whose only difference is reusability of JVMs.

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 300MB dataset or 300 splits, job performance improves by about 3.3 times). We note that with reusability the cost of initializing GATE (about 12sec) 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 Section 5.3 to select appropriate settings for these parameters. Tables 6.6-6.8 depict execution time (sec) of jobs analyzing a 100MB dataset[2] varying split sizes (1MB, 2.5MB, 5MB) corresponding to (100, 40, 20) splits; heap sizes (200MB-3.3GB) computed from Equation 5.1; 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.

[2]This size is chosen to ensure full utilization in all cases, as $max(\text{Reusability}) \times max(\text{Split size})$ $= 20 \times 5\text{MB} = 100\text{MB}$.

The ✗-value has the same meaning as in Table 6.5.

	Reusability: 1						
Heap Size (GB) / Split Size (MB)	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 6.6: Execution time (sec) varying split size and heap size with reusability 1.

	Reusability: 10						
Heap Size (GB) / Split Size (MB)	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 6.7: Execution time (sec) varying split size and heap size with reusability 10.

	Reusability: 20						
Heap Size (GB) / Split Size (MB)	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	✗	✗	✗	✗	✗	✗	✗

Table 6.8: Execution time (sec) varying split size and heap size with reusability 20.

Our first observation is that Table 6.6 features the largest fraction of successful runs (17 out of 21 possible cases), albeit at the cost of excessive execution times for 1MB and 2.5MB splits sizes. Execution times of 5MB-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 6.8 (reusability 20) we observe that performance improves for particular configurations, e.g., by up to 44% for (1MB split size, 2256MB heap size) compared to the same configuration with reusability one. Figure 6.11 depicts configurations from Tables 6.6-6.8 that are feasible (do not fail). Within the dashed box we distinguish those configurations whose execution times 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).

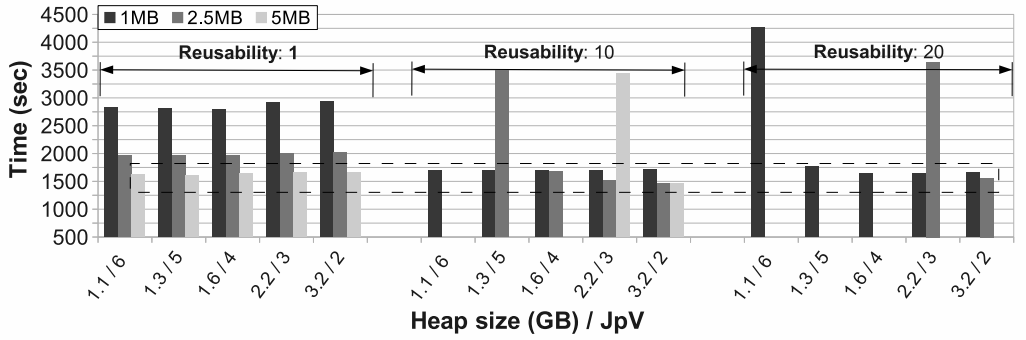


Figure 6.11: Feasible configurations from Tables 6.6-6.8. The dashed box highlights the best choices.

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 Section 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) $\times$ $max(JpV)$ $\times$ Split size MB. We have excluded split size 5MB mainly due to the high imbalance percentage it leads to as demonstrated in Figure 6.7. Figure 6.12 depicts execution times for jobs executing on JpV JVMs within a single VM.

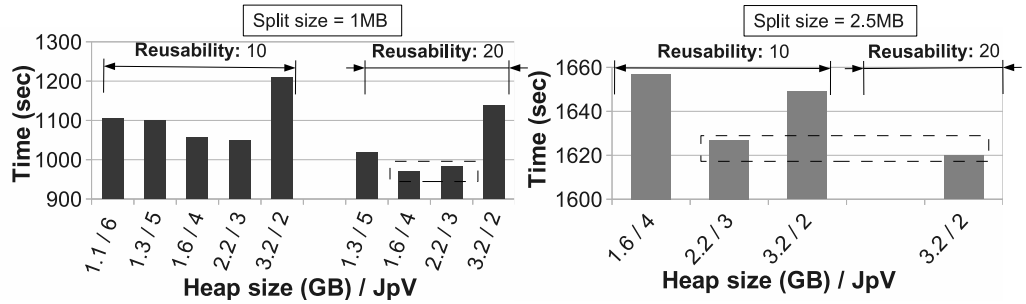


Figure 6.12: Execution time (sec) under concurrently executing JVMs. Dashed boxes highlight the best choices for split sizes 1MB (left) and 2.5MB (right).

Choosing the best configurations from Figure 6.12 for each split size leads to the following cases:

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

  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 2.5MB: Best choices are
  - C_3 = (2.2GB heap, 3 JVMs, reusability 10)
  - C_4 = (3.2GB 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.

6.2.8 Comparative Results for Different Functionalities and Number of Categories

In Section 3.4 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 200MB-SET1, for $L0$ (minimal functionality) the map output was 6MB, 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 Figure 6.13 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 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.

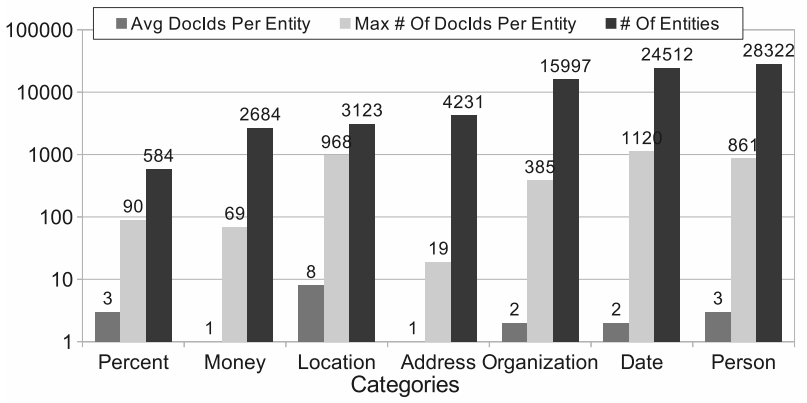


Figure 6.13: Content analysis of Reducer's output

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[3].

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 Section 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 Section 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 $\mathcal{O}(|A|d_{asz} - zd_{asz}) = \mathcal{O}(d_{asz}(|A| - z))$.

To grasp the consequences, let's put some values in the above formulae. For $|A| = 100$, $d_{asz} = 400KB$, 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 68MB/s ([51]). It follows that 39.2 MB require less than one second.

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

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 $d_{asz} = 400KB$ and $z = 11$, the extra amount of exchanged data will be $400*10^9/10$ 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 100MB-SETs was from 16,300 (with two categories, Person and Location, enabled) to 45,227 with all ten categories enabled[4].

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.

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

6.2.9 Improving Scalability through Fragmentation of Documents

In Section 6.2.4 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 800KB in fragments whose size does not exceed the split size. Our results show that this variation of CJ achieves a speedup of $x6.2$ for a 300MB dataset when using 8 Amazon EC2 VMs, an improvement over the speedup of $x5.66$ with standard CJ (without document fragments). This is attributed to the fact that the longest task execution time is now 80 sec in contrast to standard CJ where this was 249 sec. 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.

6.2.10 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 $x6.4$ 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 [73]) scalability due to two primary reasons: The existence of a few very large documents in the input dataset (Section 6.2.2) 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*, Section 6.2.5) 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 alloting 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 200MB dataset. CJ exhibits somewhat lower scalability compared to SJ ($x5.66$ vs. $x6.45$ for a 300MB 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.

Chapter 7

Applications

In this chapter we describe a number of applications/prototypes that we have developed in the Information System Laboratory [74] to which this thesis has contributed.

7.1 IOS Prototypes

Instant Overview Searching (IOS) is a powerful search-as-you-type functionality introduced in [21] which apart from showing on-the-fly only the first page of results of the guessed query, it can show several other kinds of supplementary information that provide the user with a better overview of the search space. In more detail, an off-line analysis (such as *NEM* or clustering) over the search results is performed for a set of queries (e.g. the most frequent) and the results are stored in a partitioned trie-based index [21] by exploiting the available main memory, disk and dedicated caching techniques [22] in order to provide autocompletion with complementary information (identified entities, clusters) which can be further exploited in a faceted search-like [7] interaction scheme.

Figure 7.1 shows the implemented applications that (most are web accessible [75]) include: 1) a meta-search engine (MSE) over Bing offering instant clustering of the results (see Fig. 7.1c). 2) a standalone web search engine offering instant metadata-based groupings [76] of the results (see Fig. 7.1b), and 3) a MSE offering instant entity mining over the top hits (see Fig. 7.1a). This system retrieves the top-50 hits from Bing, mines the content of each result and presents to the user a categorized list with the discovered entities. When the user clicks on an entity, the results of the specific entity are loaded instantly. Moreover, the system ranks the categories and the entities which are identified (based on scoring formulas presented at Section 3.1). Additionally, by clicking at "find its entities" the categories and the entities that lies to that specific hit are presented. Note that, without IOS functionality, this computation costs a lot both in time (about one minute) and in main memory (500 MB in average) per query. In all these applications, the user with a few keystrokes gets quite informative overviews of the search space.

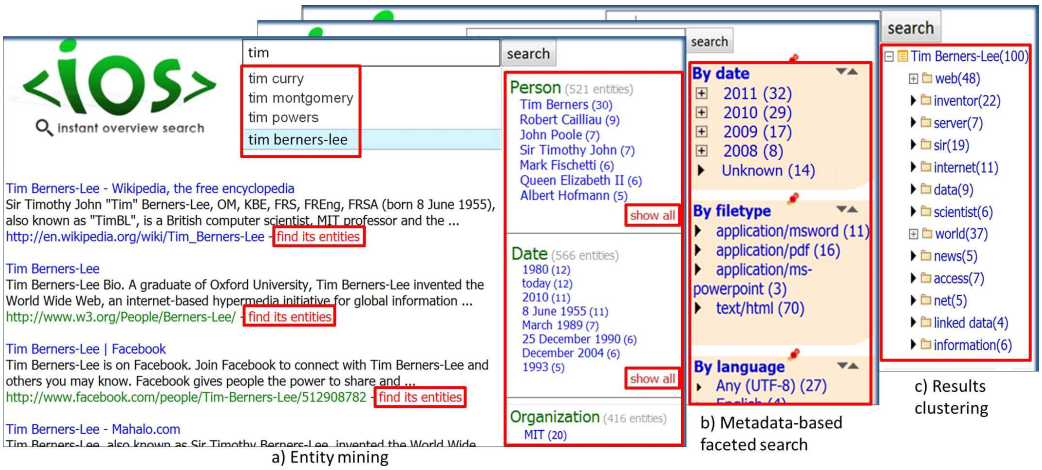


Figure 7.1: IOS Applications

All these applications also provide their functionality at query time i.e. for queries whose analysis is not stored in the trie-based indexes. Moreover, the user can configure the number of results and the hit's part (snippet or full content) that would be analyzed (see at the top left portion of Fig. 7.2).

We have also investigated how to exploit the Semantic Web technologies for enriching the entities [8] that have been identified in the web search results. Specifically, we have studied how the LOD can be exploited for providing further information about the entities that lie in the search results of both general purpose web search engines and vertical search scenarios.

Figure 7.2 presents a prototype where for each entity the user can ask the system to fetch more information from the LOD cloud. Currently our prototype adopts the (a) approach for the general web search scenario, and the (c) approach for vertical search scenarios. Specifically, when the user clicks on the small icon at the right of an entity, the system at that time checks if that entity lies in the LOD cloud (by performing a SPARQL query) and if yes it collects more information about that entity which are visualized in a popup window as shown in Figure 7.2.

7.2 XSearch in the EU iMarine Project

iMarine [77] is an ongoing FP7 Research Infrastructure Project that will establish a data infrastructure to support the Ecosystem Approach to fisheries management and conservation of marine living resources. *iMarine* empowers practitioners and policy makers from multiple scientific fields such as fisheries, biodiversity and ocean observation. The *iMarine* infrastructure will ensure that otherwise dispersed and heterogeneous data is available to all stakeholder communities through a shared virtual environment that brings together multidisciplinary data sources, supports cross-cutting scientific analysis, and assists communication. The final aim of *iMa-*

Figure 7.2: Indicative screendump from IOS Entity mining prototype

rine is to contribute to sustainable environmental management with invaluable direct or indirect benefits to the future of our planet, from climate change mitigation and marine biodiversity loss containment to poverty alleviation and disaster risk reduction.

Part of the iMarine is to leverage and improve existing technologies and tools, first and foremost, the gCube software system [78] which is a large software framework designed (and being improved from 2006) to abstract over a variety of technologies belonging data, process and resource management on top of Grid/Cloud enabled middleware. By exposing them through a comprehensive and homogeneous set of APIs, portlets and services.

One of the gCube infrastructure's family of components offers Information Retrieval (IR) facilities, i.e. it allows searching over data and information by a wide range of techniques. The IR framework is decomposed in three major categories: i) Search Framework, which includes all services focused on the search-specific aspects of the gCube platform, ii) Index Management Framework, which includes all services that are involved in the creation and management of gCube indices and iii) Distributed Information Retrieval Support Framework, which includes all services which enhance and support the IR system.

In the context of the participation of the ISL group of FORTH-ICS to the *iMarine* Project we implemented the XSearch [79] functionality/component. XSearch is a meta-search engine that reads the description of an underlying search source, and is able to query that source and analyze in various ways the returned results and also exploit the availability of semantic repositories. The offered functionalities are: clustering of the results, provision of extracted textual entities, provision of gradual faceted search, ability to fetch semantic information about extracted entities and exploitation of the offered services in any web page.

The functionality of XSearch is offered as a portlet [80] (named as XSearch-Portlet) or as a web application (named as XSearch-Service) either with the use of both.

At first we provide some general information for the portlets and portal. Portlets [80] are pluggable user interface software components that are managed and displayed in a portal which is a framework for integrating information, people and processes across organizational boundaries. One of several portal's implementations is the Liferay Portal [81], a free and open source enterprise portal written in Java that allows users to set up features common to websites. The portat is fundamentally constructed of portlets.

XSearch has three implementations:

- Web application (in which are implemented both Front-end and Back-end) implemented using Java Servlets at server side and Javascript at client (browser) side. Figure 7.3 presents an indicative screendump of the implemented application.

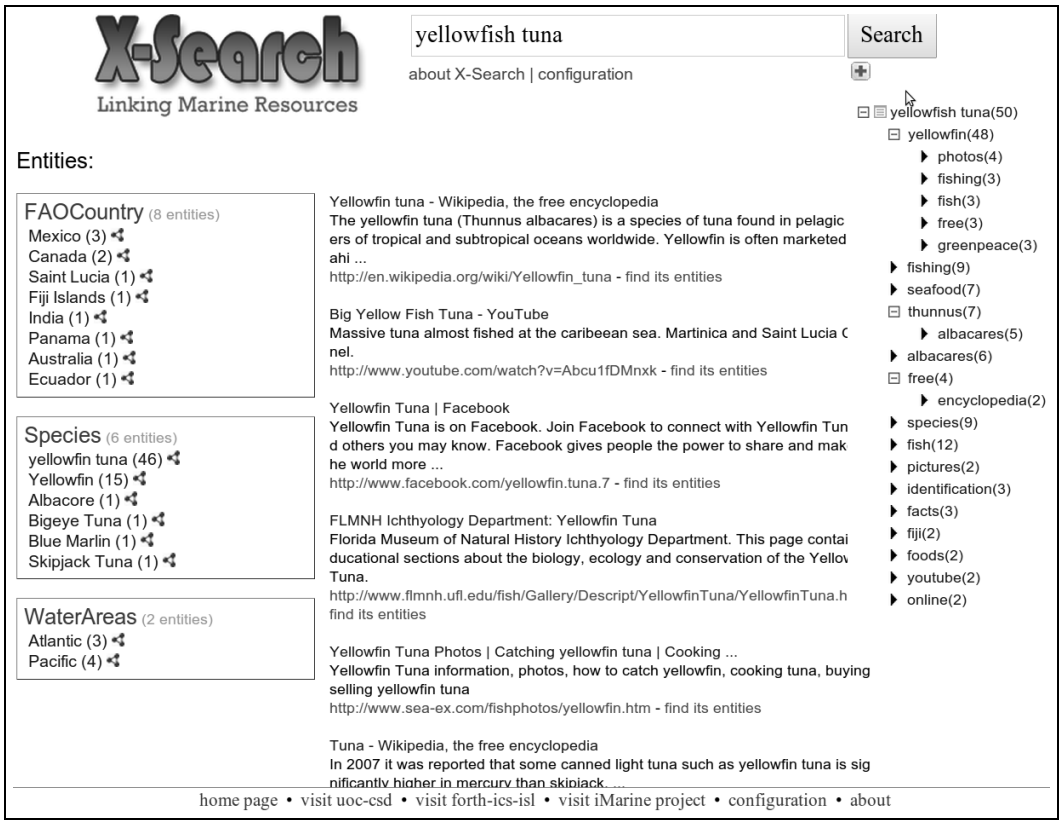


Figure 7.3: Indicative screendump from XSearch as a web application

In more detail, a user submit the query "yellowfish tuna" and are presented

the search results (with Bing as underline search system and FLOD [82] as SPARQL end point) enriched with the clustering and mining results.

- As a portlet and service, responsible for the front-end and the back-end respectively.

  At first we provide some implementation details for the XSearch-Portlet. XSearch-Portlet is a mavenized [83] portlet that is hosted by gCube (a framework dedicated to scientists which adopts Liferay Portal [81] as enabling portal technology for its Infrastructure Gateway). For a better and more efficient Ajax implementation it was developed with the Google Web Toolkit [84] (GWT) which is an open source Java software development framework that emphasizes reusable, efficient solutions to recurring Ajax challenges, namely asynchronous remote procedure calls, history management, bookmarking, and cross-browser portability.

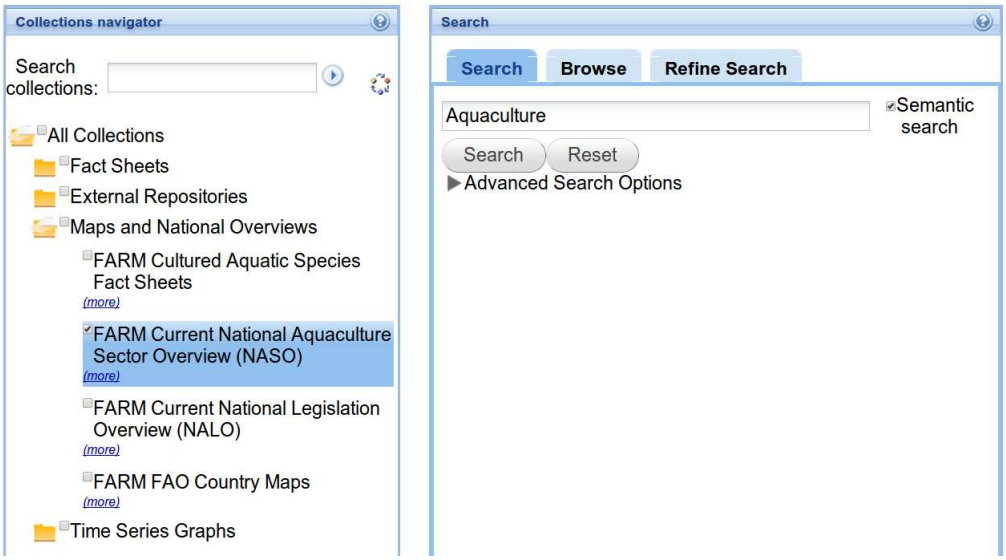


Figure 7.4: Indicative screendump of gCube search

In this approach, the underline search system is the gCube Search system [85]. We are using the gCube Search-Portlet in order to submit the query and enable the semantic analysis. Figure 7.4 shows an indicative screenshot of gCube Search-Portlet. In particular, a user selects the collections (left portion of fig. 7.4) that he wants to search through (e.g. Naso) and types the query (e.g. "Aquaculture"). By selecting the check button "Semantic Search" is enabled the XSearch functionality and finally the query is submitted by clicking on the "Search" button. After the query is submitted (with semantic search enabled), the gCube Search-Portlet formulates the CQL query through the Application Support Layer (ASL)[1] , submits it to the Search-

[1]A middleware between gCube Lower level Services and the gCube Presentation Layer.

System and receives references to the results. This occurs for keeping the required memory in the portlet in a low state and retrieving the results as they are required. After receiving the references to the results, the gCube Search-Portlet stores them in the ASL-Session, i.e. a unique session (an instance of the SessionManager object that is based on the Singleton resource pattern) that can be retrieved for each user and used to store information. Also, in the ASL-Session are stored the query terms and the selected collections that were specified by the user. Then the gCube Search-Portlet triggers XSearch-Portlet and the web page is redirected to XSearch-Portlet.

XSearch-Portlet starts fetching the top-k (the default value for K is 50) results to the ASL-Session by using the result's references. After receiving all these results it starts two parallel actions; a) it shows the top-K results to the user and b) send the top-K results to the XSearch-Service for semantic data analysis. The results (the metadata that are required for analysis, which are the title and the snippet for each hit) are sent using a TCPLocator; a stream which allows to expose the local results over the network (through a TCP connection). XSearch-Service starts consuming the results and when it reaches the desired number of results (the default is 50) it starts analyzing them; performing named entity mining and results clustering. When the analysis is done the semantic data analysis results are sent back to the XSearch-Portlet which is presented at figure 7.5.

Figure 7.5: Indicative screendump of XSearch-Portlet

As we can observe at figure 7.5, the XSearch-Portlet presents the results enriched with the clustering and mining results of semantic data analysis. The first page of the results contains only the top-10 results. If the user wants to see more results, (s)he should click on a specific page number. After clicking on the page number, XSearch-Portlet locates the corresponding 10 results and shows them directly to the user. Furthermore, if the user clicks on a specific entity/cluster the result-space is limited to contain only the results that are related with the specific entity/cluster. A more detailed analysis of the process is presented at the sequence diagram of figure 7.6.

- A portlet responsible for both Front-end and Back-end. Note that the portlet was implemented with exactly the same way as reported in previous approach.

  In this approach, the underline search system is either Bing or Google. Figure 7.7 shows an indicative example on which the user types and submits the query "Barack Obama". As we can observe, for the submitted query the search results enriched with the clustering and mining results of semantic analysis are presented.

  This approach offers some more functionalities. Specifically, by clicking at option "Advanced Criteria" it appears a drop-down list (see fig. 7.7) from which the user can select the number of top results to be analyzed and whether the analysis should be done over the contents or over the snippet.

  By clicking the "Settings" link (top-left portion of figure 7.7) it appears a pop-up presented at figure 7.8. We can observe that this window consists of three different tabs: a) Mining b) Clustering c) Search, from which the user can change the settings for each different component.

  In more detail, from the Mining tab (see fig. 7.8): we can select the number of top results that would be analyzed, whether we want to perform the analysis over content or snippet, whether we want to add domain restrictions, choose the desired categories, add a new category either by uploading a file with the entities or by giving a link to download the list of entities. Additionally, we can select the preferred ranking formula.

  From the clustering tab (see fig. 7.9a): the user can select the number of top results to be analyzed, whether the analysis should be done over the contents or over the snippet, the number of clusters that wants to be created and the preferred clustering algorithm.

  Finally, from the Search tab (see fig. 7.9b) we can choose the underline search system (with Bing and Google currently supported) and add a domain restriction to the search results.

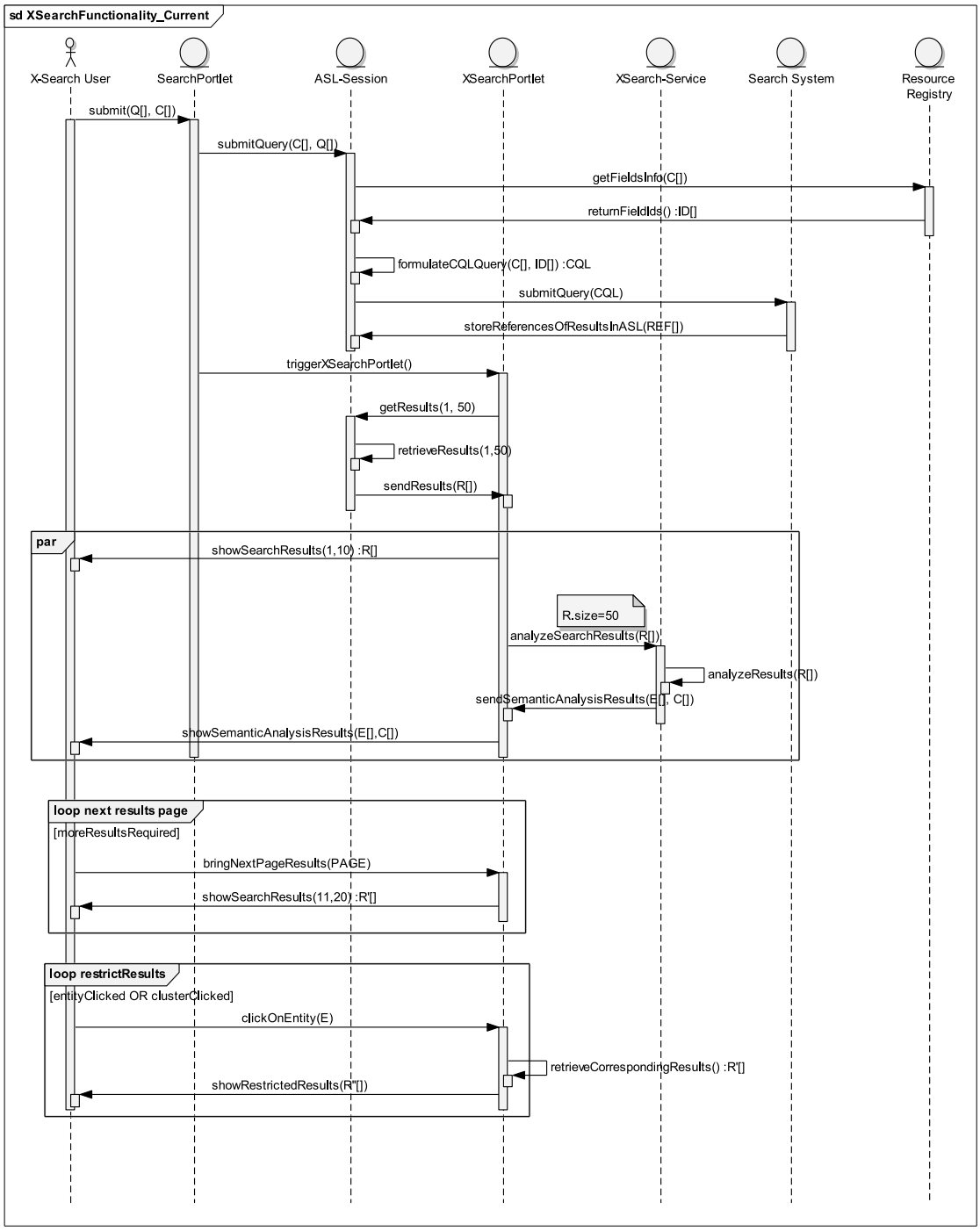


Figure 7.6: Sequence diagram of the approach that uses both portlet and service.

Figure 7.7: Indicative screendump of standalone XSearch-Portlet

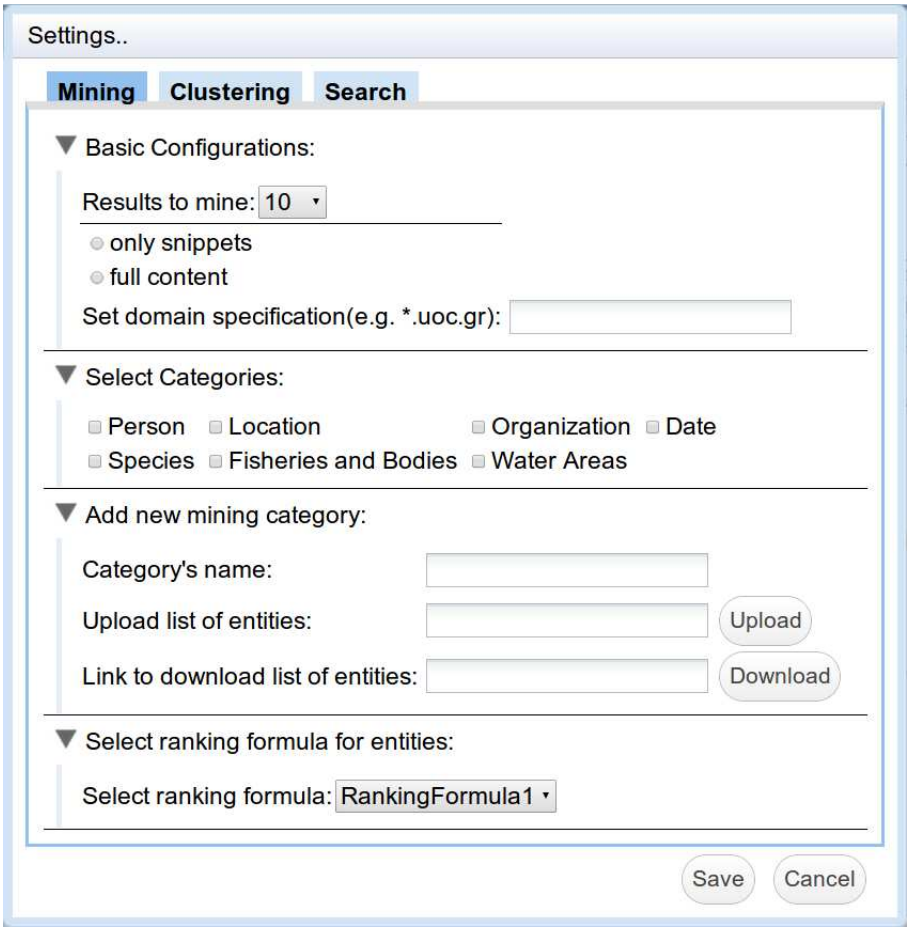
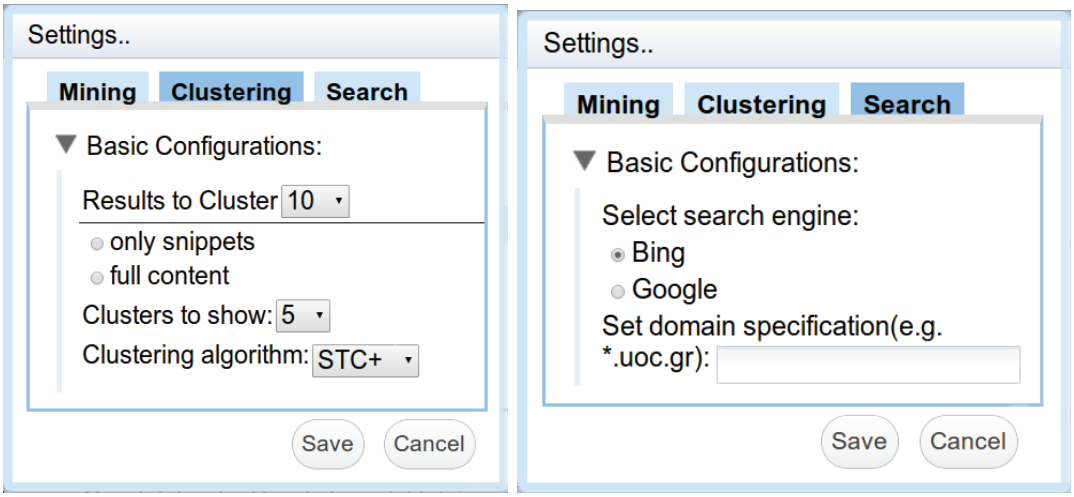


Figure 7.8: Mining settings for the standalone XSearch-Portlet



(a) (b)

Figure 7.9: Clustering (a) and Search (b) settings for the standalone XSearch-Portlet

7.3 Cloud Prototypes

We evaluated the distributed approach over two commercial Cloud providers; the Amazon [86] and the Flexiant [87]. As we mentioned before all the experiments that we have reported in Section 6.2 are on Amazon. The reason why we do not report any experimental result over Flexiant, is because we faced the common problem of having diverted system clocks. With unsynchronized clocks (between cluster's VMs) we could not trust the reported times of MapReduce framework in which the task times are relative to the Job launch time. Also, Ganglia is affected because it requires synchronized clocks between the monitoring VMs (as reported at [70]) in order to report accurate monitoring values.

In order to run a new Single Job (described at Section 5.2.1) we created a script in Bash programming language. Figure 7.10 presents the parameters that are needed to run the script which are presented each time that a user tries to run the script without any parameters. If some of the parameters malformed, informative error message appears.

```
kitsos@jk:~$ ./starNewJob.sh
Usage: ./starNewJob.sh <query> <numberOfResults> <onlySnippets>
<query>: Query
<numberOfResutls>: Number Of Results we want to mine
<onlySnippets>: Do you want to mine only snippets?(true,false)
```

Figure 7.10: Indicative screendump of running the bash script that starts a Single Job

As we can observe the parameters that are needed are: a) the query, b) the number of top-K results on which we want to perform *NEM*, c) If we want to perform the analysis over the snippets or over the full contents. Once the job is finished some Job's statistics are presented (see figure 7.11).

There are various ways to retrieve the results. Each reducer produces one output file (e.g. if there are 30 part files named part-00000 to part-00029 in the output directory). If the output is large, then it is important to have multiple parts so that more than one reducer can work in parallel.

One way of retrieving the output is by exploiting the NameNode's web interface (see at figure 7.12).

Another way of retrieving the output if it is small is to use the `-cat` option to print the output files to the console:

- `hadoop fs -cat output/*`

Another convenient command is to copy them from HDFS to our development machine. The `-getmerge` option to the hadoop fs command is useful here, as it gets all the files in the directory specified in the source pattern and merges them into a single file on the local file system:

```
11/03/13 08:15:52 INFO mapred.FileInputFormat: Total input paths to process : 101
11/03/13 08:15:53 INFO mapred.JobClient: Running job: job_200904110811_0002
11/03/13 08:15:54 INFO mapred.JobClient: map 0% reduce 0%
11/03/13 08:16:06 INFO mapred.JobClient: map 28% reduce 0%
11/03/13 08:16:07 INFO mapred.JobClient: map 30% reduce 0%
...
11/03/13 08:21:36 INFO mapred.JobClient: map 100% reduce 100%
11/03/13 08:21:38 INFO mapred.JobClient: Job complete: job_200904110811_0002
11/03/13 08:21:38 INFO mapred.JobClient: Counters: 19
11/03/13 08:21:38 INFO mapred.JobClient: Job Counters
11/03/13 08:21:38 INFO mapred.JobClient:   Launched reduce tasks=32
11/03/13 08:21:38 INFO mapred.JobClient:   Rack-local map tasks=82
11/03/13 08:21:38 INFO mapred.JobClient:   Launched map tasks=127
11/03/13 08:21:38 INFO mapred.JobClient:   Data-local map tasks=45
11/03/13 08:21:38 INFO mapred.JobClient: FileSystemCounters
11/03/13 08:21:38 INFO mapred.JobClient:   FILE_BYTES_READ=12667214
11/03/13 08:21:38 INFO mapred.JobClient:   HDFS_BYTES_READ=33485841275
11/03/13 08:21:38 INFO mapred.JobClient:   FILE_BYTES_WRITTEN=989397
11/03/13 08:21:38 INFO mapred.JobClient:   HDFS_BYTES_WRITTEN=904
11/03/13 08:21:38 INFO mapred.JobClient: Map-Reduce Framework
11/03/13 08:21:38 INFO mapred.JobClient:   Reduce input groups=100
11/03/13 08:21:38 INFO mapred.JobClient:   Combine output records=4489
11/03/13 08:21:38 INFO mapred.JobClient:   Map input records=1209901509
11/03/13 08:21:38 INFO mapred.JobClient:   Reduce shuffle bytes=19140
11/03/13 08:21:38 INFO mapred.JobClient:   Reduce output records=100
11/03/13 08:21:38 INFO mapred.JobClient:   Spilled Records=9481
11/03/13 08:21:38 INFO mapred.JobClient:   Map output bytes=10282306995
11/03/13 08:21:38 INFO mapred.JobClient:   Map input bytes=274600205558
11/03/13 08:21:38 INFO mapred.JobClient:   Combine input records=1142482941
11/03/13 08:21:38 INFO mapred.JobClient:   Map output records=1142478555
11/03/13 08:21:38 INFO mapred.JobClient:   Reduce input records=103
```

Figure 7.11: Indicative screendump of Map Reduce Job's output.

- `hadoop fs -getmerge output outputLocalCopy`

From the aforementioned ways to retrieve the Job's results it is obvious that the last approach (`-getmerge`) is the most appropriate in case that we want to present the results on a Web Application. The other two ways, are in most cases useful when we want to have a quick look over the results or to search for something specifically.

Furthermore, we have created another bash script which runs a new Chain Job procedure (described at Section 5.2.2). By default this script takes exactly the same parameters as the aforementioned script with the only difference that optionally could take another parameter that indicates the duration (in minutes) of the Job #1 .

Finally, a human readable format (see figure 7.13) was adopted for the Job's final results which was helpful for a better observation of them. In more detail, each different category is after a blank line and followed by the multitude of the belonging entities. The belonging entities are listed under the category - one entity at each line - and each one is followed by the list of document ids (the position of

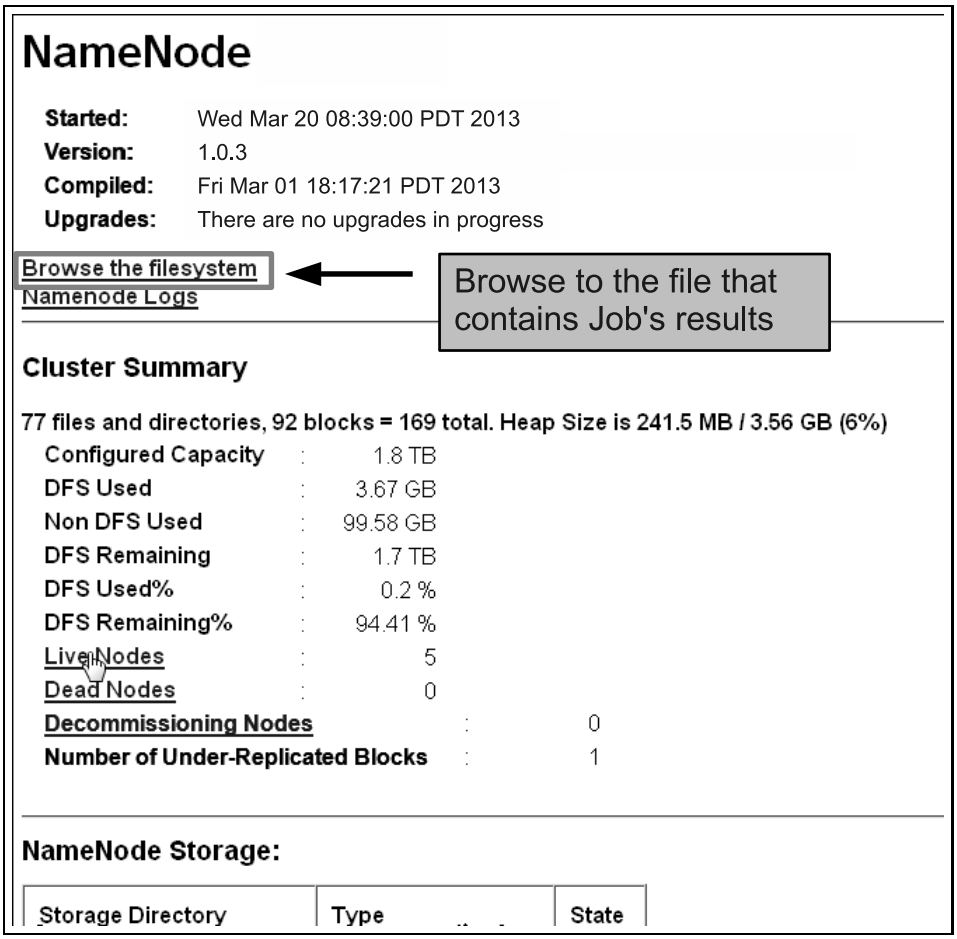


Figure 7.12: Indicative example of Namenode GUI

document in the query's returned list) in which was identified.

The aforementioned MapReduce approaches could be easily exploited through a Web Application. In particular, one of the processes that someone could follow is:

(a) create a web application that takes the parameters depicted at 7.10, and then runs one of the aforementioned bash scripts (Single or Chain Job),

(b) present the query's results (hits) on the Web Page,

(c) create a servlet which would be triggered from the MapReduce once the job has finished,

(d) configure the MapReduce parameter `mapreduce.job.end-notification.url`; in order to define the target that would be triggered once the job has finished,

(e) formulate the results in a presentable way,

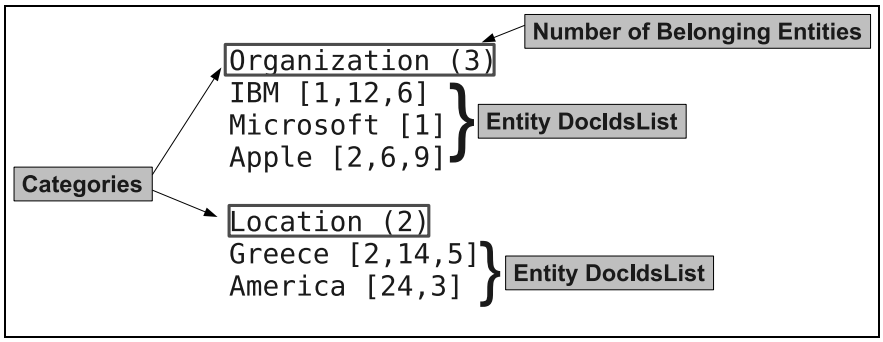


Figure 7.13: Indicative example of the Job's results

(f) load the semantic data results to the web page, and

(g) locate a tomcat on the same node on which is located the HDFS Namenode and deploy the web application.

Chapter 8

Conclusions and Future Work

In this thesis we have discussed methods to enhance web searching with entity mining. Such enhancement is useful because it gives the user an overview of the answer space, it allows the user to restrict his focus on the part of the answer where a particular entity has been mined, it is convenient for user needs (or user tasks) that require collecting entities, and it can assist the user to assess whether the submitted query is the right one (i.e. whether it fits to his information need).

We described four main approaches for supporting this functionality and we focused on two *dynamic methods*, i.e. methods that are performed at query time and do not require any pre-processing. Since such methods have not been studied in the literature (nor supported by existing systems), we compared the application of *NEM* over textual snippets versus *NEM* over the full contents (after having downloaded them at real time) of the top hits (according to various criteria). In brief, the experimental results showed that real time *NEM* over the top snippets is feasible (requires less than 2 secs for the top-50 hits) and yields about 1.2 entities per snippet. On the other hand the approach "download and mine over the full contents" is more time consuming (requires 80 secs for the top-50), but mines much more entities (in average 10.1 per hit).

However, we show that extending *NEM* (without pre-processing) to the full contents (even) for hundreds of top-hits, using a single machine, is either infeasible (due to the high computational and memory requirements) or can take hours to complete and thus we aim for a scalable methodology by exploiting MapReduce distributed computation model to parallelize the *NEM* process.

In more detail, 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 $x6.4$ on 8 Amazon EC2 VMs when analyzing 300MB 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). Our methodology and analysis however can straightforwardly extend to far larger datasets. The summarization performed by *NEM* reduces the total mapper output by an order of magnitude compared to the input dataset, thus simplifying the reducer's task in our MapReduce procedures. Larger datasets are expected to increase that output, eventually making the case for deploying multiple reducers. Considerations for provisioning and tuning reducers described in Section 5.3 will apply in this case.

As regards *entity ranking* we comparatively evaluated three methods; one based on the frequency of the entity and the rank of the hits in which it occurs, one based on similarity with the query string, and one that combines both. The user study showed that the string similarity between the query and the entity name did not improve entity ranking in our setting. Another important point is that the top-10 entities derived from snippet mining and the top-10 entities derived from contents mining for the same queries are quite different; their Jaccard similarity is less than 30% for the majority of the queries.

There are several directions for future work extending the research presented in this thesis. One issue that is worth further research is to compare the *quality* of the identified entities in snippets versus those identified in contents. Towards the same direction, it is worth investigating approaches for entity deduplication and cleaning that are appropriate for our setting.

In future, one could attempt to evaluate empirically this approach in the domains of fisheries/aquaculture and patent search.

The long term vision is to be able to mine not only correct entities but probably entire conceptual models that describe and relate the identified entities (plus other external entities) and are appropriate for the context of the user's information need. After reaching that objective the exploratory process could support the interaction paradigm of faceted search over such (crispy or fuzzy) semantic models, e.g. [88] for plain RDF/S, or [89] for the case Fuzzy RDF.

As regards to parallelized methods, 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. 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.

Bibliography

[1] T. Cheng, X. Yan, and K. Chang, "Supporting entity search: a large-scale prototype search engine," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 2007, pp. 1144–1146.

[2] R. van Zwol, L. Garcia Pueyo, M. Muralidharan, and B. Sigurbjörnsson, "Machine learned ranking of entity facets," in *Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2010, pp. 879–880.

[3] B. Ernde, M. Lebel, C. Thiele, A. Hold, F. Naumann, W. Barczyn'ski, and F. Brauer, "ECIR - a Lightweight Approach for Entity-centric Information Retrieval," in *Proceedings of the 18th Text REtrieval Conference (TREC 2010)*, 2010.

[4] W. Pratt and L. Fagan, "The usefulness of dynamically categorizing search results," *Journal of the American Medical Informatics Association*, vol. 7, no. 6, pp. 605–617, 2000.

[5] M. Wilson *et al.*, "A longitudinal study of exploratory and keyword search," in *Proceedings of the 8th ACM/IEEE-CS joint conference on Digital libraries*. ACM, 2008, pp. 52–56.

[6] B. Kules, R. Capra, M. Banta, and T. Sierra, "What do exploratory searchers look at in a faceted search interface?" in *Proceedings of the 9th ACM/IEEE-CS joint conference on Digital libraries*. ACM, 2009, pp. 313–322.

[7] G. Sacco and Y. Tzitzikas, *Dynamic taxonomies and faceted search*. Springer, 2009.

[8] P. Fafalios, I. Kitsos, Y. Marketakis, C. Baldassarre, M. Salampasis, and Y. Tzitzikas, "Web searching with entity mining at query time," *Proceedings of the 5th Information Retrieval Facility Conference, IRFC 2012, Vienna*, 2012.

[9] Y. Tzitzikas, N. Spyratos, and P. Constantopoulos, "Mediators over taxonomy-based information sources," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 14, no. 1, pp. 112–136, 2005.

[10] Y. Tzitzikas and C. Meghini, "Ostensive automatic schema mapping for taxonomy-based peer-to-peer systems," in *Cooperative Information Agents VII.* Springer, 2003, pp. 78–92.

[11] E. Jiménez-Ruiz, B. C. Grau, I. Horrocks, and R. Berlanga, "Ontology integration using mappings: Towards getting the right logical consequences," in *The Semantic Web: Research and Applications.* Springer, 2009, pp. 173–187.

[12] A. Y. Halevy, "Answering queries using views: A survey," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 10, no. 4, pp. 270–294, 2001.

[13] "Opensearch is a collection of simple formats for the sharing of search results." http://www.opensearch.org, accessed: 14/03/2013.

[14] M. Käki and A. Aula, "Findex: improving search result use through automatic filtering categories," *Interacting with Computers*, vol. 17, no. 2, pp. 187–206, 2005.

[15] M. Käki, "Findex: search result categories help users when document ranking fails," in *Proceedings of the SIGCHI conference on Human factors in computing systems.* ACM, 2005, pp. 131–140.

[16] A. Kohn, F. Bry, and A. Manta, *Professional Search: Requirements, Prototype and Preliminary Experience Report*, 2008.

[17] H. Joho, L. Azzopardi, and W. Vanderbauwhede, "A survey of patent users: an analysis of tasks, behavior, search functionality and system requirements," in *Procs of the 3rd symposium on Information interaction in context.* ACM, 2010.

[18] D. Bonino, A. Ciaramella, and F. Corno, "Review of the state-of-the-art in patent information and forthcoming evolutions in intelligent patent informatics," *World Patent Information*, vol. 32, no. 1, 2010.

[19] C. Allocca, M. d'Aquin, and E. Motta, "Impact of using relationships between ontologies to enhance the ontology search results," in *Extended Semantic Web Conference '12*, 2012, pp. 453–468.

[20] C. Carpineto, M. D'Amico, and G. Romano, "Evaluating subtopic retrieval methods: Clustering versus diversification of search results," *Information Processing and Management*, vol. 48, no. 2, 2012.

[21] P. Fafalios and Y. Tzitzikas, "Exploiting available memory and disk for scalable instant overview search," *Web Information System Engineering–WISE 2011*, pp. 101–115, 2011.

[22] P. Fafalios, I. Kitsos, and Y. Tzitzikas, "Scalable, flexible and generic instant overview search," in *WWW'12 (Demo Paper), Lyon, April*, 2012.

[23] T. Cheng, X. Yan, and K. Chang, "Entityrank: searching entities directly and holistically," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 387–398.

[24] T. Cheng and K. Chang, "Entity search engine: Towards agile best-effort information integration over the web," in *Proc. of CIDR*. Citeseer, 2007, pp. 108–113.

[25] "Entitycube: a research prototype for exploring object-level search technologies." http://entitycube.research.microsoft.com/, accessed: 30/03/2013.

[26] R. van Zwol, B. Sigurbjornsson, R. Adapala, L. Garcia Pueyo, A. Katiyar, K. Kurapati, M. Muralidharan, S. Muthu, V. Murdock, P. Ng *et al.*, "Faceted exploration of image search results," in *Procs of the 19th World Wide Web*, 2010.

[27] "Google's knowledge graph." http://www.google.com/insidesearch/features/search/knowledge.html, accessed: 30/03/2013.

[28] A. Caputo, P. Basile, and G. Semeraro, "Boosting a semantic search engine by named entities," in *Proceedings of the 18th International Symposium on Foundations of Intelligent Systems*, ser. ISMIS'09. Springer-Verlag, 2009, pp. 241–250.

[29] R. L. Grossman and Y. Gu, "Data mining using high performance data clouds: Experimental studies using sector and sphere," *CoRR*, vol. abs/0808.3019, 2008.

[30] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[31] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. Dewitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *In SIGMOD 09: Proceedings of the 35th SIGMOD international conference on Management of data*. ACM, 2009, pp. 165–178.

[32] S. Chen and S. W. Schlosser, "Map-Reduce Meets Wider Varieties of Applications," Intel Research Pittsburgh, Tech. Rep. IRP-TR-08-05, 2008.

[33] R. McCreadie, C. Macdonald, and I. Ounis, "Mapreduce indexing strategies: Studying scalability and efficiency," *Information Processing and Management*, vol. 48, no. 5, 2012.

[34] M. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham, "Data intensive query processing for large RDF graphs using cloud computing tools," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE, 2010, pp. 1–10.

[35] M. Assel, A. Cheptsov, G. Gallizo, I. Celino, D. Dell'Aglio, L. Bradeško, M. Witbrock, and E. Della Valle, "Large knowledge collider-a service-oriented platform for large-scale semantic reasoning," in *Proceedings of the International Conference on Web Intelligence, Mining and Semantics*. ACM, 2011, p. 41.

[36] J. Urbani, S. Kotoulas, E. Oren, and F. Van Harmelen, "Scalable distributed reasoning using mapreduce," *The Semantic Web-ISWC 2009*, pp. 634–649, 2009.

[37] P. Kulkarni, "Distributed SPARQL query engine using MapReduce," Master's thesis. [Online]. Available: http://www.inf.ed.ac.uk/publications/thesis/online/IM100832.pdf

[38] R. McCreadie, C. Macdonald, and I. Ounis, "Mapreduce indexing strategies: Studying scalability and efficiency," *Information Processing and Management*, 2011.

[39] J. Callan, "Distributed information retrieval," *Advances in information retrieval*, pp. 127–150, 2002.

[40] R. McCreadie, C. Macdonald, and I. Ounis, "Comparing distributed indexing: To mapreduce or not?" *Proc. LSDS-IR*, pp. 41–48, 2009.

[41] S. Papadimitriou and J. Sun, "Disco: Distributed co-clustering with map-reduce. icdm," 2008.

[42] K. Zhai, J. Boyd-Graber, N. Asadi, and M. Alkhouja, "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*. ACM, 2012, pp. 879–888.

[43] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in mapreduce," in *Proceedings of the 15th International Conference on Extending Database Technology*. ACM, 2012, pp. 38–49.

[44] P. Mika and G. Tummarello, "Web semantics in the clouds," *Intelligent Systems, IEEE*, vol. 23, no. 5, pp. 82–87, 2008.

[45] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A platform for scalable one-pass analytics using mapreduce," in *Proceedings of the 2011 international conference on Management of data*. ACM, 2011, pp. 985–996.

[46] V. Poosala, P. Haas, Y. Ioannidis, and E. Shekita, "Improved histograms for selectivity estimation of range predicates," *ACM SIGMOD Record*, vol. 25, no. 2, pp. 294–305, 1996.

[47] J. Jestes, K. Yi, and F. Li, "Building wavelet histograms on large data in mapreduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 2, pp. 109–120, 2011.

[48] D. Das and A. Martins, "A survey on automatic text summarization," *Literature Survey for the Language and Statistics II course at CMU*, vol. 4, pp. 192–195, 2007.

[49] A. Nenkova and K. McKeown, "A survey of text summarization techniques," *Mining Text Data*, pp. 43–76, 2012.

[50] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672

[51] I. Kitsos, A. Papaioannou, N. Tsikoudis, and K. Magoutis, "Adapting Data-Intensive Workloads to Generic Allocation Policies in Cloud Infrastructures," in *Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS 2012)*. IEEE, 2012, pp. 25–33.

[52] Y. Marketakis, M. Tzanakis, and Y. Tzitzikas, "Prescan: towards automating the preservation of digital objects," in *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*. ACM, 2009, p. 60.

[53] "Geonames: geographical database," http://www.geonames.org/, accessed: 10/04/2013.

[54] "Dbpedia is a crowd-sourced community effort to extract structured information from wikipedia," http://dbpedia.org/, accessed: 10/04/2013.

[55] "Freebase," http://www.freebase.com/, accessed: 10/04/2013.

[56] F. Suchanek, G. Kasneci, and G. Weikum, "Yago: a core of semantic knowledge," in *Procs of the 16th World Wide Web conf.*, 2007, pp. 697–706.

[57] B. Bishop, A. Kiryakov, D. Ognyanov, I. Peikov, Z. Tashev, and R. Velkov, "Factforge: A fast track to the web of data," *Semantic Web*, vol. 2, no. 2, pp. 157–166, 2011.

[58] "Dbpedia sparql," http://dbpedia.org/sparql, accessed: 10/04/2013.

[59] "Factforge sparql," http://www.factforge.net/sparql, accessed: 10/04/2013.

[60] "Food and agriculture organization of the united nations: Fisheries and aquaculture department," http://www.fao.org/fishery/search/en, accessed: 10/04/2013.

[61] "Apache hadoop," http://hadoop.apache.org/, accessed: 11/04/2013.

[62] "Google developers: Web metrics - size and number of resources," https://developers.google.com/speed/articles/web-metrics, accessed: 11/04/2013.

[63] "General architecture for text engineering," http://gate.ac.uk/, accessed: 03/04/2013.

[64] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan, "GATE: 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.

[65] "Java annotation patterns engine," http://en.wikipedia.org/wiki/JAPE_(linguistics), accessed: 03/04/2013.

[66] T. White, *Hadoop: The Definitive Guide. 2009.* O'Reilly Media, Yahoo.

[67] "Mapreduce: Chaining jobs," http://developer.yahoo.com/hadoop/tutorial/module4.html#chaining, accessed: 11/04/2013.

[68] J. Venner, *Pro Hadoop.* Apress, 2009.

[69] M. Massie, B. Chun, and D. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[70] M. Massie, B. Li, B. Nicholes, and V. Vuksan, *Monitoring with Ganglia.* O'Reilly Media, Incorporated, 2012.

[71] "Monitoring converged networks using the sflow standard," http://blog.sflow.com/, accessed: 11/04/2013.

[72] "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.

[73] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," *AFIPS Conference Proceedings*, vol. 30, pp. 483—-485, 1967.

[74] "Information systems laboratory of the institute of computer science (ics) of the foundation for research and technology - hellas (forth)," http://www.ics.forth.gr/isl, accessed: 07/03/2013.

[75] "Ios prototypes," www.ics.forth.gr/isl/ios, accessed: 21/02/2013.

[76] P. Papadakos, N. Armenatzoglou, S. Kopidaki, and Y. Tzitzikas, "On exploiting static and dynamically mined metadata for exploratory web searching," *Knowledge and information systems*, vol. 30, no. 3, pp. 493–525, 2012.

[77] iMarine (Data e-Infrastructure Initiative for Fisheries Managment and C. of Marine Living Resources). FP7 (FP7-283644) Research Infrastructures (2011-2014), http://www.i-marine.eu/.

[78] "gcube framework," http://www.gcube-system.org/, accessed: 21/02/2013.

[79] "X-search wiki page," http://wiki.i-marine.eu/index.php/XSearch, accessed: 21/02/2013.

[80] "About portlets," http://en.wikipedia.org/wiki/Portlet, accessed: 21/02/2013.

[81] "Liferay portal," http://www.liferay.com/documentation/liferay-portal/6.0/getting-started, accessed: 21/02/2013.

[82] "Fisheries linked open data," http://aws.amazon.com/ec2/, accessed: 12/03/2013.

[83] L. F. Feick and L. L. Price, "The market maven: A diffuser of marketplace information," *The Journal of Marketing*, pp. 83–97, 1987.

[84] "Google web toolkit," https://developers.google.com/web-toolkit/, accessed: 21/02/2013.

[85] "A search functionality in the gcube system," https://gcube.wiki.gcube-system.org/gcube/index.php/Search, accessed: 07/03/2013.

[86] "Amazon web services ec2," http://aws.amazon.com/ec2/, accessed: 11/03/2013.

[87] "Flexiant flexiscale public cloud," http://www.flexiant.com, accessed: 11/03/2013.

[88] S. Ferré and A. Hermann, "Semantic search: reconciling expressive querying and exploratory search," *The Semantic Web–ISWC 2011*, pp. 177–192, 2011.

[89] N. Manolis and Y. Tzitzikas, "Interactive Exploration of Fuzzy RDF Knowledge Bases," *Proceedings of the 8th Extended Semantic Web Conference (ECSW'2011)*, pp. 1–16, 2011.

Appendix A

A.1 Queries in the User Study

Fifteen users participated in the evaluation with ages ranging from 20 to 28, 73.3% males and 26.6% females. We selected a set of 20 queries which are very familiar to the participants (Greek citizens):

> `Andreas Papandreou, Athens History, Barack Obama, cities of Italy, corporation apple, Eiffel Tower, Greek Corporations, Greek Culture, Greek History, Greek Democrasy, konstantinos karamanlis, London, Lucas Papademos, Michael Jordan, Microsoft, Olympic games, Olympic National Park, Rent a Car at Heraklion, Restaurants in Athens, Tim Berners-Lee.`

A.2 Related Works and Systems by Category

Apart from the dimension, described at Section 2.1.2 (i.e. when *NEM* takes place), existing works on entitity mining could be distinguished according to various criteria, some indicative are sketched at Figure A.1.

Table A.1 characterizes the more relevant systems according to some basic criteria.

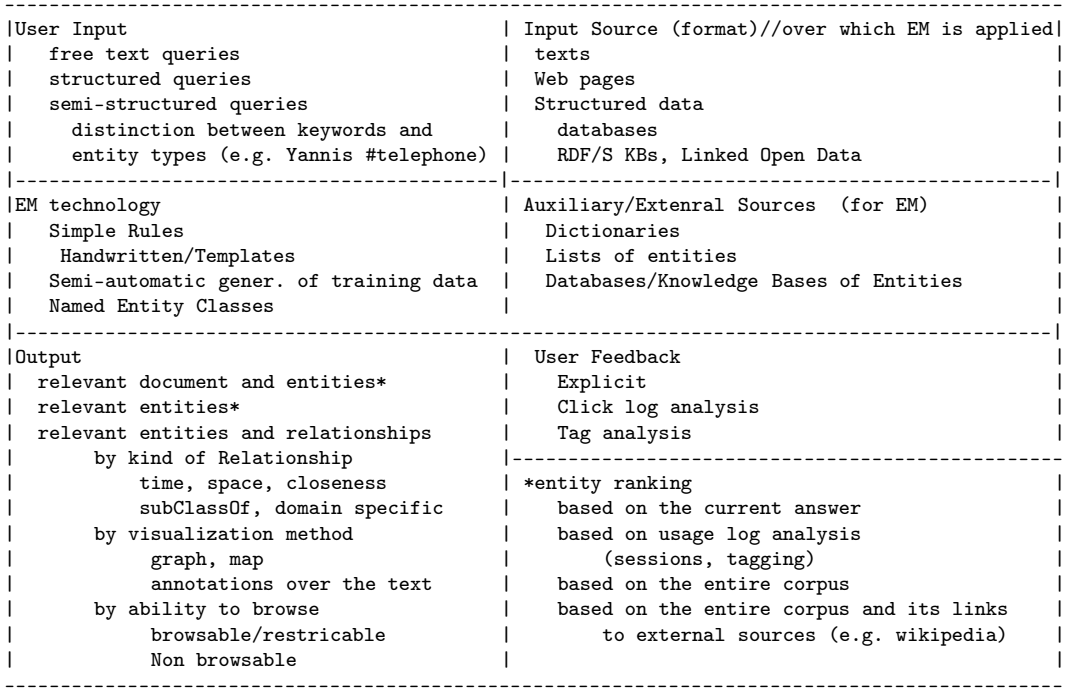


Figure A.1: A rough categorization of Entity Search engines

System	Approach	Entity Ranking	Output
IOS[21, 22]	$\mathcal{RS}$, $\mathcal{QC}$, $\mathcal{OFQ}$	Rank-aware frequency	relevant documents and entities
Entity Search Engine [1, 23, 24]	$\mathcal{OC}$	Aggregation of local and global scoring using probabilistic methods.	Ranked list of entities combined with a URL for each entity.
EntityCube [25]	$\mathcal{OC}$	According to their relevance to the query and their popularity where various different ways are proposed for computing popularity (e.g. papers citations).	Relevant documents, entities, biography, News, SNS, Publications, Name disambiguation option and Profession
MediaFaces [2, 26]	$\mathcal{OC}$	Statistical analysis of query terms/sessions and Flickr annotations	Categories combined with their entities (images)
ECIR [3]	$\mathcal{QC}$ (16 hits over Google, 64 over Bing and Yahoo)	Based on statistics from the narrowed document corpus, which was acquired via an enhanced keyword query.	One line for each found target entity with some attributes (e.g. TREC topic number)

Table A.1: Some related systems and works with the centralized process