ABSTRACT

Software retrieval is concerned with locating and identifying appropriate software resources to satisfy users requirements. It is considered to be one of the key technical issues in software reuse since "You must find it before you can reuse it". In this thesis, we investigate the problem of supporting keyword-based searching for the discovery of software resources that are installed on the nodes of large-scale, federated Grid and Cloud computing infrastructures. We address a number of challenges that arise from the unstructured nature of software and the unavailability of software-related metadata on large-scale networked environments. We present Minersoft, a harvester that visits Grid/Cloud infrastructures, crawls their file-systems, identifies and classifies software resources, and discovers implicit associations between them. The results of Minersoft harvesting are encoded in a weighted, typed graph, named the Software Graph. A number of IR algorithms are used to enrich this graph with structural and content associations, to annotate software resources with keywords, and build inverted indexes to support keyword-based searching for software. Using a real testbed, we present an evaluation study of our approach, using data extracted from production-quality Grid and Cloud computing infrastructures. Experimental results show that Minersoft is a powerful tool for software retrieval.

Asterios Katsifodimos, University of Cyprus, September 2009

MINERSOFT: SEARCHING SOFTWARE RESOURCES IN LARGE-SCALE GRID AND CLOUD INFRASTRUCTURES

Asterios Katsifodimos

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at the

University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

September, 2009

APPROVAL PAGE

Master of Science Thesis

MINERSOFT: SEARCHING SOFTWARE RESOURCES IN LARGE-SCALE GRID AND CLOUD INFRASTRUCTURES

Presented by

Asterios Katsifodimos

Research Supervisor _____
Marios D. Dikaiakos

Committee Member _____
George Angelos Papadpoulos

Committee Member _____
Demetrios Zeinalipour-Yazti

University of Cyprus

September, 2009

ACKNOWLEDGEMENTS

I offer my sincerest gratitude to my supervisor, Prof. Marios D. Dikaiakos, who has helped me throughout my thesis with his patience, trust, experience and knowledge. During the last 3 years Prof. Dikaiakos has been more than a supervisor to me. He guided me in my early steps in research, he gave me the chance to work on international projects, allowed me to travel and gain experience in international collaborations, pushed me to respect deadlines and be responsible for a number of things. I would like to thank him for his advice and the fruitful chats during all these years and for his invaluable support in the period that I was preparing for the next step in my career. I am sure that many things would be the different for me without his guidance.

I am also very grateful to my colleague Dr. George Pallis. George taught me how to promote and strengthen the outlook of my work, he showed me how to set milestones and deadlines and pushed me to concentrate on my targets. He was the one to show me all the detailed steps of a research work - from implementing a system to publishing its results. This work would not be the same without his help.

In my daily work I have been surrounded with a very cheerful group of colleagues. Andoena Balla, Nikolas Loulloudes, Nikolas Stylianides, Mustafa Jarrar, Maria Poveda and Jesus Luna have been a reason to wake up early! We had a great time - in and out of the lab.

I am also indebted to the many countless contributors to the "Open Source" programming community for providing the numerous tools and systems I have used to produce

both my results and this thesis. The entirety of my thesis has been completed using such technologies and I consider it to have been an enormous benefit.

Many many thanks go to my friends of my student years in Cyprus. They have been the reason to sleep late! Also to Giannis, Asterios, Paulos and Miltos (in random order) that possess the greatest qualities as friends.

I would also like to mention that this work was financially supported (in part) by the European Commission under the Seventh Framework Programme Enabling Grids for E-sciencE project (contract number INFSO-RI-222667).

Finally, I thank my beloved parents for supporting me throughout all my studies at University, and being the ones to teach me early enough that "nothing comes for free".

CREDITS

- **"Minersoft: Searching Software Resources in Grid and Cloud Computing Infrastructures"**, G. Pallis, **A. Katsifodimos**, M.D. Dikaiakos, submitted to the "ACM Transactions on Software Engineering and Methodology Journal".

- **"Minersoft: Searching Software Resources in EGEE infrastructure"**, G. Pallis, **A. Katsifodimos**, M.D. Dikaiakos, submitted to the "Grid Computing Journal", Springer.

- **"Effective Keyword search for Software Resources installed in Large-scale Grid Environments"**, G. Pallis, **A. Katsifodimos**, M.D. Dikaiakos: The 2009 IEEE/WIC/ACM International Conference on Web Intelligence (WI2009, acceptance rate 16%), 15-18 September 2009, Milan Italy.

- **"Harvesting Large-Scale Grids for Software Resources"**, **A. Katsifodimos**, G. Pallis, M.D. Dikaiakos, 9th IEEE International Symposium on Cluster Computing and the Grid, (CCGrid09, acceptance rate 21%), May 18-21, 2009. Shanghai, China.

- **"Minersoft: A Keyword-based Search Engine for Software Resources in Large-scale Grid Infrastructures"**, M.D. Dikaiakos, **A. Katsifodimos**, G. Pallis: the 8th Hellenic Data Management Symposium, September 2009, Athens, Greece.

- **"Searching Software Resources in the Grid"**, **A. Katsifodimos**, G. Pallis, M.D. Dikaiakos, Poster in the 4th EGEE User Forum/OGF 25, March 2-6, 2009, Catania, Italy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

A growing number of large-scale Grid and Cloud infrastructures are in operation around the world, providing production-quality computing and storage services to many thousands of users from a wide range of scientific and business fields. The emergence of Grid and Cloud computing infrastructures has definitely opened new perspectives in software engineering community; software developers should exploit the hardware capabilities and services that offer these large-scale distributed computing environments. One of the main goals of these infrastructures is to make their software resources and services easily accessible and attractive for end-users [14]. To achieve this goal, it is important to establish advanced, user-friendly tools for software search and discovery, in order to help end-users locate application software suitable to their needs and encourage software reuse [16, 46], software investigation [49], clone detection [26] and computational resources selection.

1.1 Motivation

Adopting a keyword-based search paradigm for locating software seems like an obvious choice, given that keyword search is currently the dominant paradigm for information discovery [37]. Keyword-based search traditionally relies on Information Retrieval (IR) algorithms that explore the occurrence of words in documents. To motivate the importance of such a tool, let us consider a software developer who is searching for graph mining software deployed on a Grid/Cloud infrastructure. Although informative tags about installed software can be published through Grid information systems, Grid system administrators seldom follow such a practice [23]. Unfortunately, the manual discovery of such software is a daunting, nearly impossible task: taking the case of EGEE [3], one of the largest production Grids currently in operation, the software developer would have to search among 300 sites with several sites hosting well over 1 million software-related files. The situation is not better in emerging Cloud infrastructures: a user of the Amazon Elastic Cloud service can choose among 2.100 Amazon Machine Images (AMIs), with each AMI hosting at least 14.000 files, including installed software. Existing Grid/Cloud providers do not support any tool that would help developers to select computational resources so as to run their applications. The only service which is supported by some Cloud providers is an interface where the user can search only the names of their computational resources (i.e. AMIs).

Envisioning the existence of a software search engine, the software developer would submit a query to the search engine using some keywords (e.g. "statistical analysis software," or "ffmpeg video processing"). In response to this query, the engine would return a list of software matching the query's keywords, along with computational resources where this software is located. Thus, the software developer would be able to identify the

providers hosting an application suitable to his/her needs, and would accordingly prepare and submit jobs to these ones. Figure 1 shows a mockup of the first page of results for the query "protein docking software". Consequently, this would save considerable time since the searching of offered services is time-consuming task. In case of Clouds, where the user pays with respect to the time he/she uses the underlying Cloud services, such a tool would also provide economic benefits. The cost of running an application on a Cloud provider depends on the compute, storage and communication resources it will provision and consume. The less time a user spends for searching Cloud services and instantiates the machines, the less money a user spends in Cloud providers.

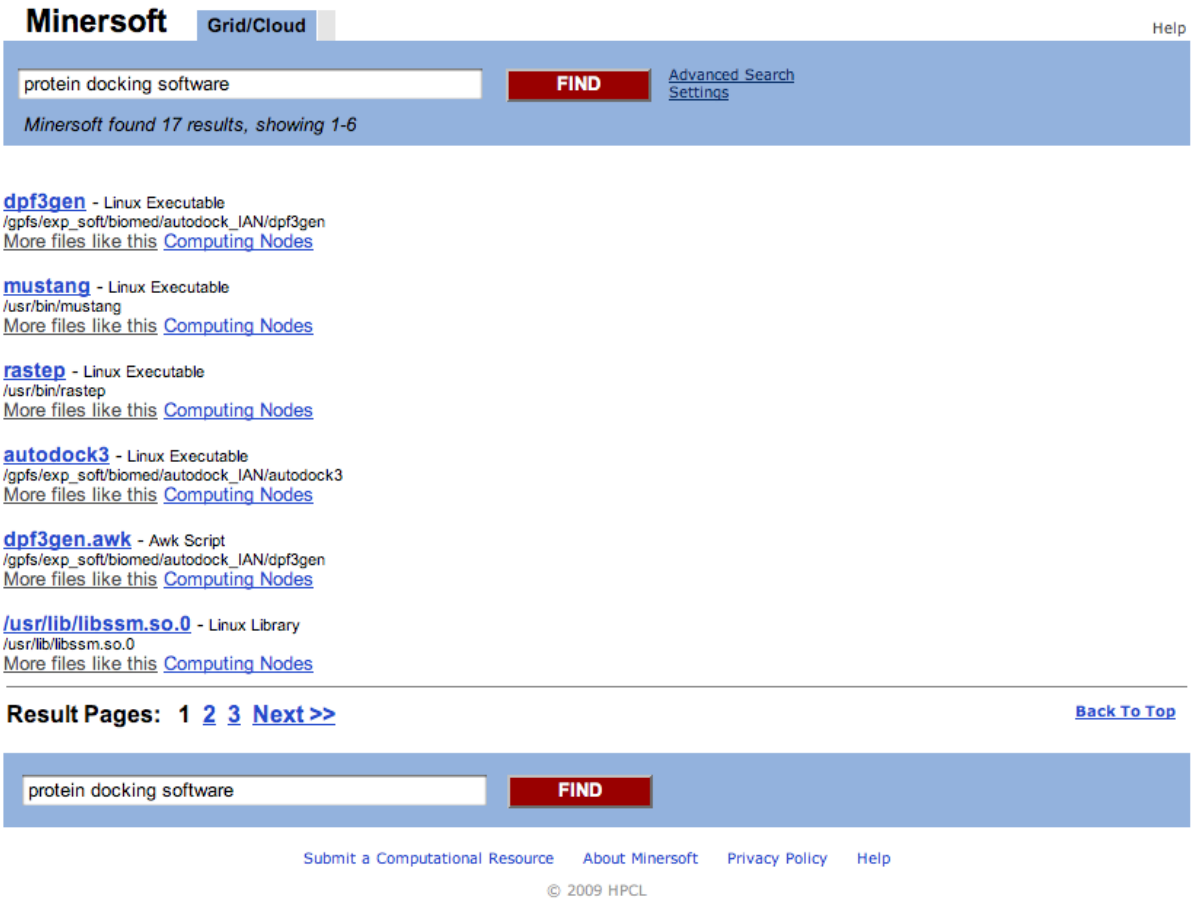

Figure 1: Minersoft's result page showing the results for the query "protein docking software" (mockup).

Following this motivation, we developed the *Minersoft* software search engine. To the best of our knowledge, Minersoft provides the first full-text search facility for locating software resources installed in large-scale Grid and Cloud infrastructures. Minersoft visits a computational resource, crawls its file-system, identifies software resources of interest (binaries, libraries, documentations etc.), assigns type information to these resources, and discovers implicit associations between them. Also, Minersoft extracts a number of terms by exploiting the path within file-system and the filename of software resources.

To achieve these tasks, Minersoft invokes file-system utilities and object-code analyzers, implements heuristics for file-type identification and filename normalization, and performs document analysis algorithms on software documentation files and source-code comments. The results of Minersoft harvesting are encoded in the Software Graph, which is used to represent the context of discovered software resources. We process the Software Graph to annotate software resources with metadata and keywords, and use these to build an inverted index of software. Indexes from different computational resource providers in the Grid and Cloud are retrieved and merged into a central inverted index, which is used to support full-text searching for software installed on the nodes of a Grid infrastructure. The present work continues and improves upon the authors preliminary efforts in [47, 33]. Specifically, we extend Minersoft so as to support efficient search on Grid and Cloud computing infrastructures.

1.2 Challenges

Existing Web search engines cannot be used for software retrieval. In the context of Web, harvesting is a well-studied research problem [24]; crawlers traverse the directed graph of the World Wide Web following edges of the graph, which correspond to hyperlinks that connect together its nodes, i.e., the Web pages. However, the situation is fundamentally different on the context of Grids and Clouds. In terms of harvesting and indexing in a Grid and Cloud setting, there are several limitations (e.g., time, pricing etc.) that should be addressed and do not exist in the context of Web. Thus, we should take advantage of various parallelization techniques. In terms of information retrieval, the software usually resides in file systems, together with numerous other files of different kinds. Traditional file systems do not maintain metadata representing file semantics

and distinguishing between different file types. Also, there are not hyperlinks between software files. Furthermore, the registries of distributed computing infrastructures rarely publish little, if any information about installed software [23]. Also, many software packages in such infrastructures cannot be installed using package management systems (e.g. RPMs). Finally, software files usually come with few or no free-text descriptors. Consequently, the software-search problem cannot be addressed by traditional IR approaches. Instead, we need a new methodology that will: i) discover automatically software-related resources installed in file systems that host a great number of files and a large variety of file types; ii) extract structure and meaning from those resources, capturing their context, and iii) discover implicit relationships among them. Also, we need to develop methods for effective querying and for deriving insight from query results. The provision of full-text search over large, distributed collections of unstructured data has been identified among the main open research challenges in data management that are expected to bring a high impact in the future [10]. Searching for software falls under this general problem since file-systems treat software resources as unstructured data and maintain very little if any metadata about installed software.

1.3 Contributions

The main contributions of this work can be summarized as follows:

- We present the design, the architecture, and implementation of the Minersoft harvester.
- We introduce the *Software Graph*, a typed, weighted graph that captures the types and properties of software resources found in a file system, along with structural and

content associations between them (e.g. directory containment, library dependencies, documentation of software).

- We present the Software Graph construction algorithm. This algorithm comprises techniques for discovering structural and content associations between software resources that are installed on the file systems of large-scale distributed computing environments.

- We provide a study about the installed software resources in i) Grid resource provider (EGEE [3], one of the largest Grid production services currently in operation) and ii) in two commercial Cloud providers (Amazon EC2 Cloud [1] and Rackspace Cloud [9]).

- We provide a test dataset for evaluating software retrieval systems. Specifically, our dataset collection contains software files, installed in Grid/Cloud infrastructures, and their relevance judgments for a set of keyword queries. The dataset is available to the software engineering community.

- We conduct an experimental evaluation of Minersoft, on real, large-scale Grid and Cloud testbeds, exploring performance issues of the proposed scheme. We also demonstrate the effectiveness of the Software Graph as a structure for annotating software resources with descriptive keywords, and for supporting full-text search for software. Results show that Minersoft achieves high search efficiency.

The remainder of this thesis is organized as follows. Chapter 2 presents an overview of related work. In Chapter 3, we introduce the concepts of Grid and Cloud model and we provide the definitions for software resources, software package and Software Graph.

Chapter 4 describes the architecture of Minersoft. Chapter 5 describes the proposed algorithm to create a Software Graph annotated with keyword-based metadata. Chapter 6 presents various implementation details of Minersoft. In Chapter 7 we present an experimental assessment of our work. We conclude in Chapter 8.

Chapter 2

Related Work

A number of research efforts [38, 56] have investigated the problem of software-component retrieval in the context of language-specific software repositories and CASE tools (a survey of recent work can be found in [41]). One of the key distinguishing traits of these approaches is the corpus upon which the search is conducted:

2.1 Searching in a software repository

In [42], Maarek et. al. presented GURU, possibly the first effort to establish a keyword-based paradigm for the retrieval of source code residing in software repositories. The cosine similarity metric is used between queries and documented software resources. GURU uses probabilistic models (quantity of information) to map documents to terms providing results that include both full and partial matches. Similar approaches have also been proposed in [13, 44, 40]. All these works exploit source-code comments and documentation files, representing them as term-vectors and using similarity metrics from Information Retrieval (IR) to identify the associations between software resources. Results showed that such schemes work well in practice and are able to discover links between documentation

files and source codes. Except of using IR techniques, authors presented Suade [49], a tool which identifies the associations between software resources by exploiting the topology of a graph of structural dependencies for a software system.

The use of folksonomy concepts has been investigated in the context of the Maracatu system [54]. Folksonomy is a cooperative classification scheme where the users assign keywords (called tags) to software resources. A drawback of this approach is that it requires user intervention to manually tag software resources. Finally, the use of ontologies is proposed in [34]; however, this work provides little evidence on the applicability and effectiveness of its solution.

The search for software can also benefit from extended file systems that capture file-related metadata and/or semantics, such as the Semantic File System [27], the Linking File System (LiFS) [12], or from file systems that provide extensions to support search through facets [35], contextualization [52], desktop search (e.g., Confluence [28], Wumpus [55]), etc. Although Minersoft could easily take advantage of the above file systems offering this kind of support, in our current design we assume that the file system provides the metadata found in traditional Unix and Linux systems, which are common in most Grid and Cloud infrastructures.

2.2 Searching over the Internet

In [29], authors described an approach for harvesting software components from the Web. The basic idea is to use the Web as the underlying repository, and to utilize standard search engines, such as Google, as the means of discovering appropriate software assets. Other researchers have crawled through Internet publicly available CVS repositories to build their own source code search engines (e.g., SPARS-J) [45]. In [38], authors developed

a keyword-based paradigm, called Sourcerer, for searching repositories of source code which are available over the Internet. Finally, Google Code [5] and Koders [7] are two well-known source code search engines which are also available through the Internet. Google Code Search is for developers interested in Google-related/open-source development. The user can search for open source-code and a list of Google services which support public APIs. On the other hand, Koders is a search engine for open source code. It enables software developers to easily search and browse source code in thousands of projects posted at hundreds of open source repositories.

2.3 Searching in the Grid

In the Grid context, a recent work has proposed a software search service, called GRIDLE [50]; this scheme allows users to specify a high-level workflow plan including the requirements of each software file. Then, GRIDLE presents a ranked list of files that match partially or totally user requirements. However, GRIDLE cannot be used as a keyword-based paradigm for locating software resources in the Grid since neither crawls the Grid sites, nor searches installed software files.

2.4 Minersoft vs. existing approaches

Although we are not aware of any work that proposes a keyword-based paradigm for locating software resources on large-scale Grid/Cloud infrastructures, our work overlaps with prior work on software resources retrieval [13, 42, 44, 54]. These works mostly focus on developing schemes that facilitate the retrieval of software source files using the keyword-based paradigm. Minersoft is different from all the above works in a number of key aspects:

- Minersoft supports searching in Grid, Cloud, cluster infrastructures as well as repositories;
- Minersoft supports searching not only for source codes but also for executables and libraries stored in binary format;
- Minersoft does not presume that file-systems maintain metadata (tags etc.) to support software search; instead, the Minersoft harvester generates such metadata automatically by invoking standard file-system utilities and tools and by exploiting the hierarchical organization of file-systems;
- Minersoft introduces the concept of the Software Graph, a weighted, typed graph. The Software Graph is used to represent software resources and associations under a single data structure, amenable to further processing.
- Minersoft addresses a number of additional implementation challenges that are specific to Grid and Cloud infrastructures:
  - Software management is a decentralized activity; different machines in Grids/-Clouds may follow different policies about software installation, directory naming etc. Also, software entities on such infrastructures often come in a wide variety of packaging configurations and formats. Therefore, solutions that are language-specific or tailored to some specific software-component architecture are not applicable.
  - Harvesting the software resources in Grid and Cloud infrastructures is a demanding task for computational, storage, and communication resources. Also,

most Grid systems do not support interactive computation. Therefore, software harvesting needs to be performed in a distributed, non-interactive manner.

– The users of a Grid/Cloud infrastructure do not have direct access to servers. Therefore, a harvester has to be either part of middleware services (something that would require the intervention to the middleware) or to be submitted for execution as a normal job, through the middleware. In the Minersoft architecture and implementation we adopt the latter approach, which facilitates the deployment of the system on different Grid and Cloud infrastructures.

Approach	Searching paradigm	Corpus
GURU	keyword-based	Repository
Suade	software elements (fields and methods)	Repository
Marakatu	keyword-based	Repository
SEC	keyword-based	Repository
Wumpus	keyword-based	Repository
Extreme Harvesting	keyword-based	Web
SPARS-J	keyword-based	Internet repositories
Sourcerer	keyword-based	Internet repositories
Koders	keyword-based	Internet open-source repositories
Google Code Search	keyword-based	Web
Minersoft	keyword-based	Grid, Cloud, Cluster, Repository

Table 1: Minersoft VS other approaches (search paradigm & corpus)

Approach	binaries/scripts	source code libraries	Software-description documents	Binary libraries
GURU		✓	✓	
Suade		✓		
Marakatu		✓		
SEC		✓		
Wumpus		✓		
Extreme Harvesting		✓		
SPARS-J		✓		
Sourcerer		✓		
Koders		✓	✓	
Google Code Search		✓	✓	
Minersoft	✓	✓	✓	✓

Table 2: Minersoft VS other approaches(search objects)

Chapter 3

Overview

In this chapter we provide some background for the system model of Grid and Cloud. Then, we define software resource, software package and Software Graph, which are the main focus of this thesis.

3.1 Grid System Model

A Grid is a large-scale network computing system that scales to Internet size environments with thousands of machines distributed across multiple organizations and administrative domains. Machines in a Grid, called *workernodes* are typically grouped into autonomous administrative domains, called *Grid sites*, which communicate via high-speed communication links.

The system elements that comprise a Grid are: processing, network and storage elements. Processing elements are the CPUs which can be classified into uniprocessor, multiprocessor, cluster, and parallel processing elements. Network elements are the routers, switches, gateways, virtual private network devices and firewalls. Storage elements are the network attached storage devices, such as RAID devices, database machines etc. The

jobs which run in a Grid site are terminated by site's batch system if they exceed a time threshold. The jobs are submitted to Grid site via a Grid middleware.

The Grid architecture consists of four layers: fabric, core middleware, user-level middleware and Grid applications. The Grid fabric layers consists of the system elements. The core Grid middleware provides services which abstract the complexity and heterogeneity of the fabric layer (i.e., remote process management, storage access, information registration and discovery etc.). The user-level Grid middleware utilizes the interfaces provided by the low level middleware so as to provide higher abstractions and services, such as programming tools, resource brokers for managing resources and scheduling application tasks for execution on global resources. Finally, the Grid applications layer utilizes the services provided by user-level middleware so as to offer engineering and scientific applications and software toolkits to Grid users.

3.1.1 The EGEE Grid

The Enabling Grids for E-scienceE (EGEE) grid infrastructure consists of a set of middleware services deployed on a worldwide collection of computational and storage resources, plus the services and support structures put in place to operate them. EGEE is a large, multi-science Grid infrastructure, federating some 270 resource centres worldwide, providing around 140.000 CPUs and 20 Petabytes of storage. This infrastructure is used on a daily basis by several thousands of scientists federated in over 200 Virtual Organizations on a daily basis. EGEE is a stable, well-supported infrastructure, running the latest released versions of the gLite middleware [4].

gLite comprises a variety of job and data management services that are described below:

- *Computing Element*: A Computing Element (CE), in Grid terminology, is a set of computing resources localized at a site (i.e. a cluster, a computing farm). A CE includes a Grid Gate (GG), which acts as a generic interface to the cluster; a Local Resource Management System (LRMS) (sometimes called batch system), and the cluster itself, a collection of Worker Nodes (WNs), the nodes where the jobs actually run.

- *Workload Management System*: The purpose of the Workload Management System (WMS) is to accept user jobs, to assign them to the most appropriate Computing Element, to record their status and retrieve their output. The Resource Broker (RB) is the machine where the WMS services run. Jobs to be submitted are described using the Job Description Language (JDL), which specifies, for example, which executable to run and its parameters, files to be moved to and from the Workernode on which the job is run, input Grid files needed, and any requirements on the CE and the Worker Node.

- *Storage Element*: The Storage Element (SE) provides uniform access to data storage resources. The Storage Element may control simple disk servers, large disk arrays or tape-based Mass Storage Systems (MSS). Most EGEE sites provide at least one SE. Storage Elements can support different data access protocols and interfaces. The , GSIFTP (a GSI-secure FTP) is the protocol for whole-file transfers, while local and remote file access is performed using RFIO or gsidcap. Most storage resources are managed by a Storage Resource Manager (SRM), a middleware service providing capabilities like transparent file migration from disk to tape, file pinning, space

reservation, etc. However, different SEs may support different versions of the SRM protocol and the capabilities can vary.

Other storage services include: the Logical File Catalog, which holds information about the location of files and replicas held at different Storage Elements, the File Transfer Service, which is responsible for replicating files across different Storage Elements, the Metadata Grid Application (AMGA), which manages metadata and the Encryption Data Service, which manages the encryption of data stored in Storage Elements.

An *EGEE site* is a group of a number of the aforementioned services. Most Grid sites are consisted from a Computing Element, a Storage Element and a set of Workernodes. Each EGEE site is maintained by a *Resource Provider*. Resource Providers are legal entities like research/academic institutions or companies.

Users that want to access the EGEE Grid infrastructure and make use of its services need to join a Virtual Organization (VO) supported by the infrastructure and its Resource Providers. A Virtual Organization is a dynamic collection of individuals and/or institutions that share resources in a controlled and mutually agreed fashion [25]. More specifically, EGEE users registered within a particular Virtual Organization obtain security credentials for single Grid sign-on that enable them to obtain controlled access to resources belonging to that particular VO, despite the fact that such resources span different EGEE sites across different countries.

3.2 Cloud System Model

Cloud Computing describes a recent trend in Information Technology (IT) that moves computing and data away from desktop and portable PCs into large data centers that provide on-demand services through the Internet on a “pay as you go” basis. The computing

nodes of a Cloud, called *Cloud Virtual Servers*, are managed by a single administrative domain. Typically, the service offerings of Cloud service providers (*Cloud providers*) comprise access to computing and storage capacity, to software platforms for developing and deploying applications, and to actual applications. User access to Cloud services is achieved via SSH calls, Web services or batch systems. The Cloud architecture consists of three abstract layers: infrastructure, platform and application. Infrastructure is the lowest layer and is a means of delivering basic storage and computing capabilities as standardized services over the network. Servers, storage systems, switches, routers, and other systems handle specific types of workloads, batch processing to server/storage augmentation during peak loads. The middle layer provides higher abstractions and services to develop, test, deploy, host and maintain applications in the same integrated development environment. Application layer is the highest layer and features a complete application offered as a service.

The main technical underpinnings of Cloud Computing infrastructures and services include virtualization, service-oriented software, Grid Computing technologies, management of large facilities, power efficiency etc. Cloud service consumers purchase Cloud services in the form of Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) or Software-as-a-Service (SaaS) and sell value-added services (e.g. utility services) to end-users. Within the Cloud, the laws of probability give the service provider great leverage through statistical multiplexing of varying workloads and easier management - since a single software installation can cover many users' needs.

Minersoft focuses on Cloud infrastructures that provide IaaS. Minersoft can be very useful in that context since Virtualized Server instances are provided without any facilities

that support searching for software installed in them. However, investigation of providing searching in PaaS or SaaS can be done in a future version of Minersoft.

Below, we provide the architectural principles of the Amazon Compute Cloud and the Rackspace Cloud. We chose those two Cloud providers because they are the leading companies on the Cloud market (as of the date of writing this thesis) and they both provide IaaS services.

	Computational Resource Provider	**Computational Resource**	**Architecture**	**Jobs Submission**	**Time CPU Constraints**
Grid Computing	Grid Resource Provider	Grid Site	Distributed	Grid Middleware	Yes
Cloud Computing	Cloud Provider	Cloud Virtual Server	Centralized	SSH calls, Web services, batch systems	No

Table 3: Features: Grid computing vs. Cloud Computing

3.2.1 The Amazon Elastic Compute Cloud

The Amazon EC2 (Amazon Elastic Compute Cloud) [1], is a collection of web services that provide resizable compute capacity in the cloud. It provides users with complete control of their computing resources and let them run on Amazons computing environment. Amazon EC2 allows users, to scale capacity, both up and down, as their computing requirements change. EC2 allows users to pay only for capacity that they actually use. Use of Amazon servers is paid by the hour in a pay-as-you-go manner.

Amazon provides server instances in the form of machine images, named *Amazon Machine Images(AMIs)*. Each AMI is a disk image that contains an operating system

and a set of softwares installed. Instantiated AMIs are called *server instances*. Amazon does not provide any means of job management services to their clients. Each client is responsible of installing the software required in order to satisfy his needs. Users can store their own AMIs and share them with other users.

Amazon S3 (Amazon Simple Storage Service): Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any user access to the same highly scalable, reliable, and fast data storage infrastructure that Amazon uses to run its own global network of web sites. Amazon S3 provides web service based protocols in order to perform operations on data stored in the Amazon infrastructure.

Amazon server instances have the ability to *mount* the Amazon S3 data buckets using the *Elastic Block Store* service. Using the combination of the Elastic Compute Cloud and the S3 storage services, users can process data stored on S3 as if they were local data.

3.2.2 The Rackspace Cloud

Rackspace Cloud is a Rackspace-owned company who specializes in cloud hosting services. It provides very similar services to that os Amazon, but in a much simpler way. Rackspace provides three main cloud services.

- *Cloud Sites*: Cloud Sites concentrate on hosting web applications with the potential to scale. Cloud Sites are capable of scaling automatically whenever needed without users' asking for it. In cloud sites, servers are preconfigured with a range of software options and are fully managed similar to a shared hosting environment. Keeping their hosting platform standardized is what allows Rackspace Cloud to easily monitor and scale the service as needed. Users simply install web applications (in a language

like Ruby, PHP etc) and Cloud Sites scale as needed whenever more bandwidth and/or server capacity needed.

- *Cloud Servers*: Just like Amazon Elastic Compute Cloud, Rackspace provides its users the ability to choose from a list of preconfigured server images that can be launched. Users launch instances on demand and pay in a pay-as-you-go manner. More specifically users pay by the instance hours and the bandwidth used.

- *Cloud Files*: Cloud Files is a reliable, scalable and affordable web-based storage for backing up and archiving static content. Similar to Amazon S3, volumes can be mounted in cloud servers and provide more storage capacity. The difference from Amazon S3 is that with cloud files, users can use the LimeLight CDN [8] automatically, in order to provide high bandwidth and low latency file download to web users.

3.3 Minersoft's Abstract System Model

Working on top of both Grid and Cloud infrastructures, requires that Minersoft treats them in a uniform way. In order to do that, access to different types of storage and computational resources has to be abstracted in a way that accessing them is easy and infrastructure-agnostic. For Minersoft to achieve that abstraction, we had to identify the common things that infrastructures provide. It is very easy to see that all of the cloud and Grid infrastructures that Minersoft supports, have some common characteristics (see Figure 2):

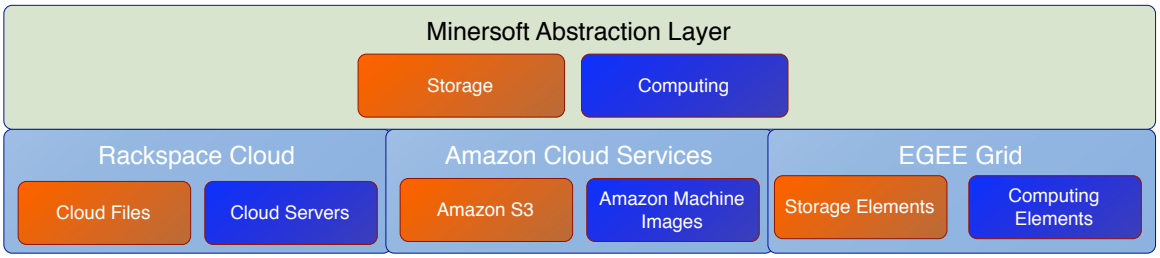

Figure 2: Minersoft Abstraction Layer over different underlying Infrastructures.

1. **Storage Services:** All the infrastructures provide storage services; Amazon provides *Amazon S3*, Rackspace provides *Cloud files* and the EGEE Grid provides the *Storage Element*.

2. **Computational Services:** Amazon provides the *Amazon Machine Images*, Rackspace provides the *Cloud Servers* and EGEE provides *Workernodes* (grouped in *Grid Sites*).

3. **Access protocols:** For the storage/computational resources to be accessible by their users each infrastructure provides (different) access protocols.

   Computational Resources access protcols:

   - In the case of Cloud (Amazon and Rackspace), access to computational resources (Virtual Servers) is being done through SSH.
   - In the case of the EGEE Grid, the protocol to access computational resources is the API of the gLite Workload Management System.

   Storage Resources access protcols:

- In the case of Cloud (Amazon and Rackspace), access to storage resources is being done through the Amazon S3 API and the Cloud Files API respectively.
- In the case of the EGEE Grid, the protocol to access storage resources is the SRM (Storage Resource Management) Protocol.

3.4 Definitions

Definition 1 Software Resource. A software resource is a file that is installed on a machine and belongs to one of the following categories: i) *executables* (binary or script), ii) software *libraries*, iii) *source codes* written in some programming language, iv) *configuration files* required for the compilation and/or installation of code (e.g. makefiles), v) unstructured or semi-structured *software-description documents*, which provide human-readable information about the software, its installation, operation, and maintenance (manuals, readme files, etc).

The identification of a software resource and its classification into one of these categories can be done by heuristic approaches which have been invented by human experts (system administrators, software engineers, advanced users).

Definition 2 Software Package. A software package consists of one or more content or/and structurally associated software resources that function as a single entity to accomplish a task, or group of related tasks.

Human experts can recognize the associations that establish the grouping of software resources into a software package. Normally, these associations are not represented through some common, explicit metadata format maintained in the file-system. Instead, they are expressed implicitly by location and naming conventions or hidden inside configuration files

(e.g., makefiles, software libraries). Therefore, the automation of software-file classification and grouping is a non-trivial task. To represent the software resources found in a file-system and the associations between them we introduce the concept of the *Software Graph*.

Definition 3 Software Graph. Software Graph is a weighted, metadata-rich, typed graph $G(V, E)$. The vertex-set V of the graph comprises: i) vertices representing software resources found on the file-system of a computing node (*file-vertices*), and ii) vertices representing directories of the file-system (*directory-vertices*). The edges E of the graph represent structural and content associations between vertices.

Structural associations correspond to relationships between software resources and file-system directories. These relationships are derived from library dependencies, file-system structure according to various conventions (e.g., about the location and naming of documentation files) or from configuration files that describe the structuring of software packages (RPMs, tar files, etc). *Content associations* correspond to relationships between software resources derived by text similarity.

The Software Graph is "typed" because its vertices and edges are assigned to different types (classes). Each vertex v of the Software Graph $G(V, E)$ is annotated with a number of associated metadata attributes, describing its content and context:

- $name(v)$ is the normalized name of the software resource represented by v. Normalization is being done with a set of heuristics that come from the linux filesystem hierarchy standard [39] and the domain knowledge about naming conventions of various file types. More specifically, the heuristics used detect the type of file and according to that they remove either the suffix or the prefix of the filename:

  - **Man pages:** The linux manual pages are named according to the naming conventions of the linux filesystem standard. Manual page filenames end with a number (1-9) that denote the manual page section (1-Commands available to users, 2-Unix and C system calls, 3-C library routines for C programs etc). In order to normalize the filename of a manual page, the suffix must be removed. If the manual page is compressed the filename ends with an extra ".gz". If not, the filename's suffix is just a number (1-9), thus only the number is removed. for example the manual for gcc, "gcc.1.gz" or "gcc.1" is normalized to "gcc".
  - **Linux library files:** Linux library filenames are prefixed with "lib" and suffixed with ".so.number" where the *number* denotes the library release number (e.g."libgcc.so.1"). In order to normalize the filename of a library both the prefix (*lib*) and the suffix (*.so.∗*) are removed.
  - **Other files:** Except from the aforementioned heuristics for normalization, almost all kinds of files use a suffix to denote their type. In order to normalize those files, their filename suffix (also known as *extension*) is removed.

- $tags(v)$ is a set of tags of the software resource v; tags of a software resource are words that describe its content. Tags can be zero or more of the following software resource "characteristics": *man page*, *binary executable*, *text*, *source code*, *script*, *readme*, *library*. Heuristics based on the output of the linux *file* command and the filename's extension of the file (software resource), are used to extract tags from files. See Chapter 6 for details on how the tagging is done.

- $type(v)$ denotes the type of v; a vertex can be classified into one of a finite number of types (more details on how this is done, are given in the next sections). In order

to decide the type of the file and store it in the vertex, the tags above are used. The type of a software resource v can be one of the following: *Binary Executable* (linux executables with the tag "Binary Executable"), *Binary Library* (linux libraries with the tag "Binary Library"), *Documentation* (readme files, man pages and all other text files that are not source code), *Source* (source code files and scripts), *Garbage* (anything that does not belong to any of the aforementioned types).

- $site(v)$ denotes the computing site where file v is located.
- $path(v)$ is a set of terms that are derived from the tokenization of the path-name of software resource v in the file system of $site(v)$.
- $original\_path(v)$ is the original path (untokenized) in which the software resource v is located in the file system of $site(v)$.
- $file\_size(v)$ is the file-size (in bytes) of the software resource v.
- $zone_l(v), l = 1, \ldots, z_v$ is a set of zones assigned to vertex v. Each zone contains terms extracted from a software resource that is associated to v and which contains textual content. In particular, $zone_1(v)$ stores the terms extracted from v's own contents, whereas $zone_2(v), \ldots, zone_{z_v}(v)$ store terms extracted from software documentation files associated to v. The number $(z_v - 1)$ of these files depends on the file-system organization of $site(v)$ and on the algorithm that discovers such associations (see subsequent section). Each term of a zone is assigned an associated weight w_i, $0 < w_i \leq 1$ equal to the term's TF/IDF value in the corpus. Furthermore, each $zone_l(v)$ is assigned a weight g_l so that $\sum_{l=1}^{z_v} g_l = 1$. Zone weights are introduced to support weighted zone scoring in the resolution of end-user queries.

Each edge e of the graph has two attributes: $e = (type, w)$, where *type* denotes the association represented by e and w is a real-valued weight ($0 < w \leq 1$) expressing the degree of correlation between the edge's vertices.

The *Software Packages* are coherent clusters of "correlated" software resources in *Software Graph*. Next, we focus on presenting the architecture of Minersoft (Chapter 4), how the Software Graph can be constructed (Chapter 5), some implementation details (Chapter 6) and we evaluate its contribution (Chapter 7).

Chapter 4

Minersoft Architecture

Creating an information retrieval system for software resources that can cope with the scale of emerging distributed computing infrastructures (Grids and Clouds) presents several challenges. Fast crawling technology is required to gather the software resources and keep them up to date. Storage space must be used efficiently to store indices and metadata. The indexing system must process hundreds of gigabytes of data efficiently. In this section, we provide a description of how the Minersoft architecture, depicted in Figure 3.

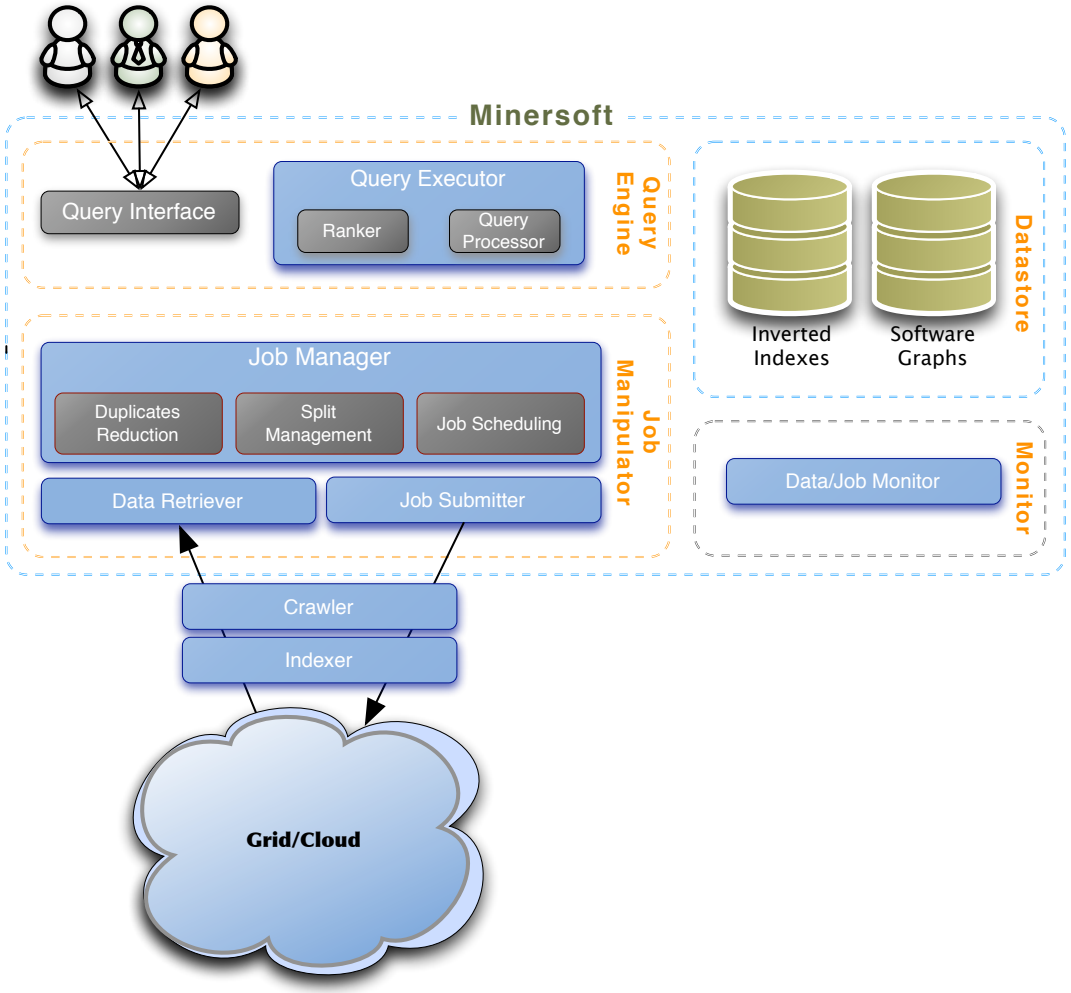

Figure 3: Minersoft Architecture.

For the efficient implementation of Minersoft in a Grid and Cloud setting, we take advantage of various parallelization techniques in order to:

- Distribute parts of the Minersoft computation to Grid and Cloud resource providers. Thus, we take advantage of their computation and storage power, to speedup the file retrieval and indexing processes, to reduce the communication exchange between the Minersoft system and resource-provider sites, and to achieve Minersoft's scalability

in the context of an increasing number of resource-provider sites. Minersoft tasks are wrapped as jobs that are submitted for execution to Grid and Cloud systems.

- Avoid overloading resource-provider sites by applying load-balancing techniques when deploying Minersoft jobs.

- Improve the performance of Minersoft jobs by employing multi-threading to overlap local computation with Input/Output (I/O).

- Adapt to the policies put in place by different Grid and Cloud computing resource providers regarding their limitations, such as the number of jobs that can be accepted by their queuing systems, the total time that each of these jobs is allowed to run on a given Grid site, etc.

4.1 Overview

Minersoft has a MapReduce-like architecture [22]; the crawling and indexing is done by several distributed multi-threaded crawler and indexer jobs, which run in parallel for improved performance and efficiency. Each crawler and indexer is assigned for processing a number of *splits*, with each split comprising a different set of files. The key components of the Minersoft architecture are (see Figure 3):

1. The *Job manipulator*, which manages crawler and indexer jobs and their outputs. The manipulator comprises three modules: the Job manager, the Job submitter, and the Data retriever. The *Job manager* undertakes the allocation of files into splits and the scheduling of crawler and indexer jobs. The *Job submitter* handles the job submission and the *Data retriever* retrieves the output of crawler and indexer jobs from the infrastructure.

2. The *Monitor* module, which maintains the overall supervision of Minersoft jobs. To this end, the *monitor* communicates with the job manager, the data-store and the underlying infrastructure.

3. The *Datastore* module stores the resulting Software Graphs and the full-text inverted indexes.

4. The *Query engine* module, which is responsible for providing quality search results in response to user searches. The query engine module comprises the *Query processor* and the *Ranker*. The former receives search queries and matches them against the inverted indexes of Minersoft. The latter ranks query results in order to improve user-perceived accuracy and relevance of replies. Ranking is especially important when queries result to large numbers of "relevant" software resources. *Ranker* uses the Lucene relevance ranking algorithms [2]. The default scoring algorithms of Lucene take into account factors such as the frequencies of a particular query-term's appearance in the zones of a software resource and in the zones of all the software resources inside the Software Graph.

4.2 Minersoft Crawler

The crawler is a multi-threaded program that performs FST construction, classification and pruning, and structural dependency mining. To this end, the crawler scans the file-system of a computing site and constructs the FST, identifies software-related files and classifies them into the categories described earlier (binaries, libraries, documentation,

etc), drops irrelevant files, and applies the structural dependency mining rules and algorithms described in Chapter 5. The crawler terminates after finishing with the processing of all splits assigned to it by the job manager.

Crawling of a computational resource is being done in two phases. In the first phase, a single crawler per Grid Site/Cloud Virtual Server, is sent to construct the FST. In the second phase, the files of the FST have to be classified and structural dependencies between them must be found. The process of classifying the FST files and detecting structural dependencies is computationally intensive. To this end, the FST files are separated into splits. Each split contains a set of files of the FST. Multiple crawlers per Grid Site/Cloud Virtual Server are submitted and the splits are assigned to them.

The output of the two-phase crawler is the first version of the SG that corresponds to the site assigned to the crawler. This output is saved as a *metadata store file* comprising the file-id, name, type, path, size, and structural dependencies for all identified software resources. The metadata store files are saved at the storage services associated with the computing site visited by the crawler, that is, at the local Storage Element of a Grid site or at the Storage Service of a Cloud service.

4.3 Duplicates reduction Policy

Typically, popular software applications and packages are installed on multiple sites of distributed computing infrastructures. If we identify duplicates, we will be able to avoid indexing them multiple times. Consequently, the performance of indexing is improved.

After the files of all the Grid Sites/Cloud Virtual Servers have been classified into categories, Minersoft has obtained enough information to be able to detect duplicates

(e.g. path, filename, size). Files with the same name, path and size that belong to different Grid Sites/Cloud Server are considered to be duplicates.

To address the duplication issue, the *job manager* uses a *duplicate reduction policy* to identify the exact duplicate files. According to our policy, a duplicate file is assigned to the Grid site/Cloud Virtual Server which has the minimum number of assigned files that should be indexed. The key idea behind this policy is to avoid multiple indexing of duplicate software resources in Grid sites/Cloud Virtual Servers so as to prevent their overloading. In this context, for each Grid site/Cloud Virtual Server, the following steps take place:

1. The *file index* is sorted in ascending order with respect to the count of Grid sites/-Cloud Virtual Servers that a file exists.

2. The files which do not have duplicates are directly assigned to the corresponding Grid site/Cloud Virtual Server.

3. If a file belongs to more than one Grid sites or Cloud Virtual Servers, the file is assigned to the site with the minimum number of assigned files.

By reducing the duplicates, splits that are going to be assigned to indexers are reduced, thus the indexing phase takes less time than it would be needed if the duplicates were not reduced. Note that the splits assigned to indexers, are not the same with the splits assigned to crawlers. The number of files that have to be indexed is much smaller than the number of files that have to be crawled. This happens because after the classification process (crawling) software-irrelevant files are dropped and also because the files are distributed according to the aforementioned file-assignment policy.

4.4 Minersoft Indexer

The Minersoft indexer is a multi-threaded program that reads the files captured in the *metadata store files* and creates full-text inverted indexes (see Chapter 5). To this end, the indexer performs first keyword scraping, keyword flow and content association mining, in order to enrich the vertices of its assigned SG with keywords mined from associated documentation-related vertices. This results in enriching the terms and posting lists of inverted indexes with extra keywords. At the end of indexing process, for each computing site (Grid site or Cloud Virtual Server) there is an inverted index containing a set of terms, with each term associated to a posting list of pointers to the software files containing the term. The terms are extracted from the zones of SG vertices.

4.5 Distributed Crawling and Indexing Process

The crawling and indexing of distributed computing infrastructures requires the retrieval and processing of large parts of the file systems of numerous sites. Therefore, this task needs to address various performance, reliability and policy issues.

In the Grid context, a challenge for crawler and indexer jobs is to process all the software resources residing within a Grid site without exceeding the time constraints imposed by site policies. Jobs which exceed the maximum allowed execution time are terminated by the site's batch system. The maximum wall-clock time usually ranges between 2 and 72 hours. Similarly, in the Cloud context, the challenge is to accomplish the crawling and indexing computation with the least possible cost spend on computing, storing, and transferring data.

To allow for the more flexible management of its tasks, Minersoft decomposes the file system of each site into a number of *splits*, with the size of each split chosen so that the crawling can be distributed evenly and efficiently within the constraints of the underlying computing infrastructure. The splits are assigned to crawler/indexer jobs on a continuous basis: When a Grid site or Cloud Virtual Server finishes with its assigned splits, the *monitor* informs the *job manager* in order to send more splits for processing. If a site becomes laggard, the *monitor* sends an alert message to the *job manager*, which cancels the running jobs and reschedules them to run when the workload of the site is reduced. Furthermore, if the batch system queue of a Grid site is full and does not accept new jobs, the *monitor* sends an alert signal and the *job submitter* suspends the submission of new crawler/indexer jobs to that site until the batch system becomes ready to accept more.

When the crawling completes, Minersoft's *data retriever* module fetches the *metadata store files* from all machines, and merges them into a *file index*. The *file index* comprises information about each software resource and is temporally stored in the *Datastore*. The *file index* will be used in order to identify the duplicate files during the indexing process; the duplication reduction policy is described in the following subsection. When the indexing has been completed, the *file index* is deleted. Then, the *data retriever* fetches the resulted inverted indexes the individual SGs from all sites. Both the full-text inverted indexes and the SGs are stored in the *Datastore*.

4.6 Minersoft Implementation and Deployment

The implementation of the *job manager* and *monitor* rely upon the Ganga system [18], which is used to create and submit jobs as well as to resubmit them in case of failure. We adopted Ganga in order to have full control of the jobs and their respective arguments

and input files. We have also implemented a Ganga plugin so as to support SSH job submissions for Cloud computing infrastructures. In this context, the *monitor* (through Ganga scripts) monitors the status of jobs after their submission and keeps a list of Grid sites/Cloud Virtual Servers and their failure rate. If there are Grid sites/Cloud Virtual Servers with a very high failure rate, the *monitor* eventually puts them in a black list and notifies *job manager* so as to stop submitting jobs to them.

The crawler is written in Python. The Python code scripts are put in a tar file and copied on a storage element before job submission starts. The tar file is being downloaded and untarred to the target Grid site/Cloud Virtual Server before the crawler execution starts. By doing that, the size of the jobs input sandbox is reduced, thus job submission is accelerated because the Workload Management System has to deal with much less files per job.

The indexer is written in Java and Bash and uses an open-source high performance, full-text index and search library (Apache Lucene [2]). In order to execute the indexer jobs, we follow the same code-deployment scenario as with crawlers.

The *job manager* has to distribute the crawling and indexing workload before the job submission starts. This is done by creating splits for each Grid site/Cloud Virtual Server that Minersoft has to crawl. The input file for each split is uploaded on the storage element and registered to a file catalog. The split input is then downloaded from a storage element and used to start the processing of files. The split input is a text file containing the list of files that have to be crawled or indexed. After execution, the jobs upload their outputs on storage elements and register the output files to a file catalog. The logical file names and the directories containing them in the file catalog are properly named so that they implicitly state the split number and the site that they came from or going to.

4.6.1 Minersoft Crawler/Indexer Job's lifecycle

Minersoft indexer and crawler jobs follow the same file-staging and execution principles. Their lifecycle is depicted in Figure 4. The lifecycle of a job starts by copying the job's input files to a Storage Element and end by downloading it's outputs to the centralized Minersoft infrastructure. The details of each of the individual steps are described below:

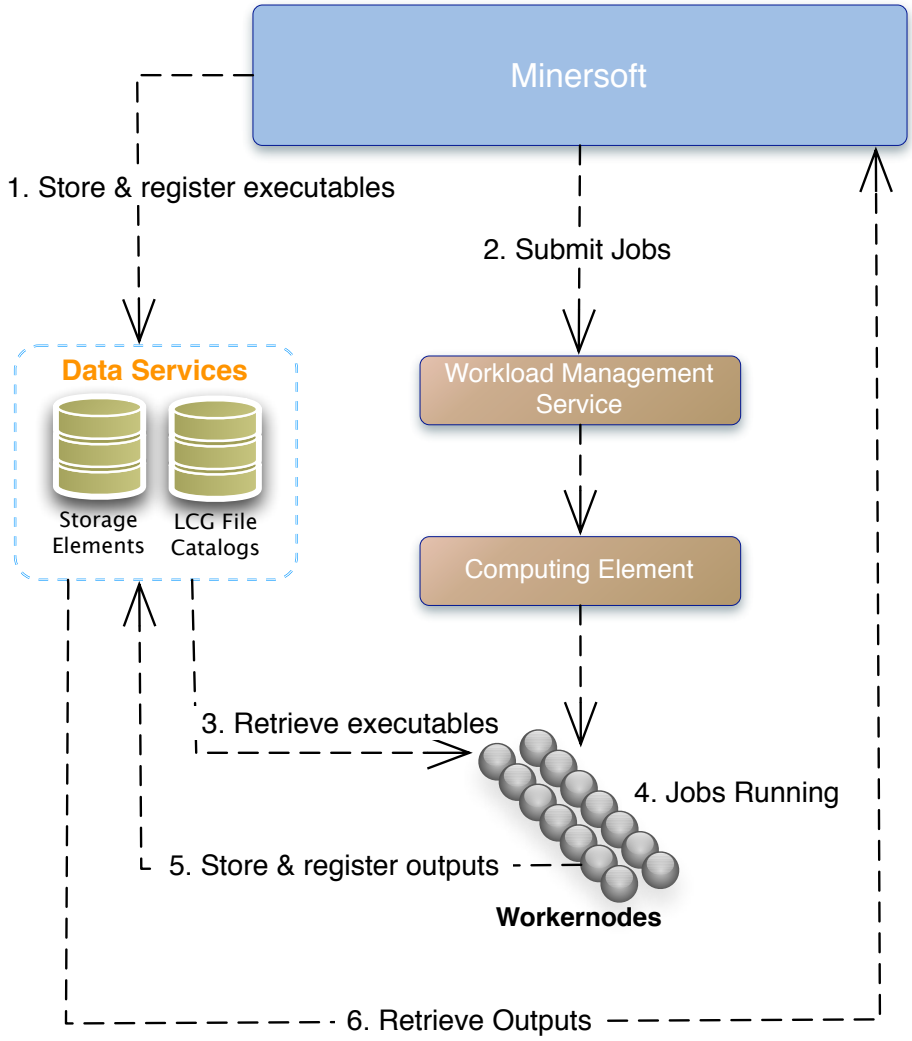

Figure 4: Minersoft crawler/indexer job's lifecycle.

1. *Store & register executables*: In order to submit and run crawler and indexer jobs efficiently, input and output files are copied into Storage Elements(SEs) and registered to LCG File Catalogs(LFCs). By utilizing the Storage Elements, the Workload Management System (WMS) and Minersoft itself don't have to cope with many(sometimes big) files, as they are downloaded directly from the Storage Elements before a job starts it's execution into a Workernode.

2. *Submit Jobs*: After the executables are copied to the storage elements, Minersoft submits the Indexer/Crawler jobs using the Workload Management Systems. The WMS moves the jobs to the target Computing Elements for the execution to start.

3. *Retrieve Job executables*: After a job is moved from the Computing Element into the target Workernode, the crawler/indexer jobs start to run. The executable files are downloaded from the Storage Elements just before the actual execution.

4. *Jobs running*: After the executables are downloaded the jobs start its' execution.

5. *Store & register outputs*: After the jobs have successful finished their execution, outputs are stored in the SE's and registered to the LFCs making them available to Minersoft for further processing.

6. *Retrieve outputs*: Minersoft retrieves the outputs from the SEs and keeps them to it's centralized infrastructure.

Chapter 5

Software Graph Construction and Indexing

Web search engines rely on Web crawlers for the retrieval of resources from the World Wide Web. Collected resources are stored in repositories and processed to extract indices used for answering user queries. Typically, crawlers start from a carefully selected set of Web pages (a seed list) and try to "visit" the largest possible subset of the Web in a given time-frame crossing administrative domains, retrieving and indexing interesting/useful resources. To this end, they traverse the directed graph of the World Wide Web following edges of the graph, which correspond to hyperlinks that connect together its nodes, i.e., the Web pages. During such a traversal (crawl), a crawler employs the HTTP protocol to discover and retrieve Web resources and rudimentary metadata from Web server hosts. Additionally, crawlers use the Domain Name Service (DNS) for domain name resolution.

Due to the absence of hyperlinks in software resources, the situation is fundamentally different for software searching on the context of Grids and Clouds. A key responsibility of the Minersoft harvester is to construct a Software Graph (SG) for each computing site, starting from the contents of its file system. Figure 5 depicts an example of a filesystem that contains binary executables, libraries, text files, source codes, readme files and manual

pages. As you see in the figure, binary executables are connected with readme files and manual pages, libraries are connected with binary executables and readme files etc.

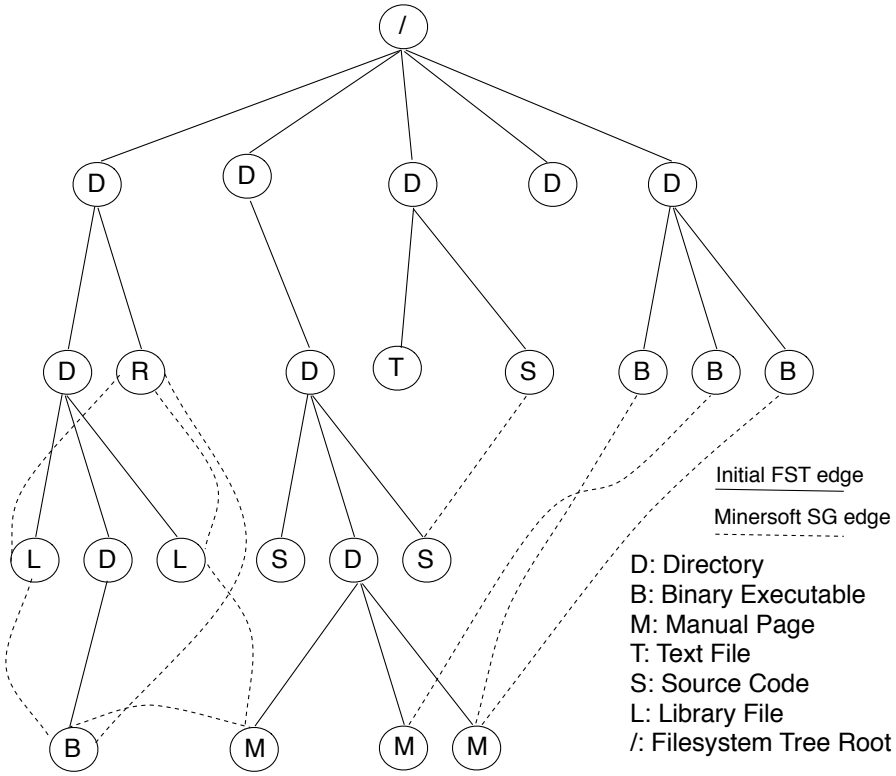

Figure 5: An example of a filesystem tree converted to a Software Graph

Below we describe the steps that we propose in order to build the filesystem tree of a computational node and then convert it into a Software Graph:

FST construction: Initially, Minersoft scans the file system of a site and creates a *file-system tree* (FST) data structure. The internal vertices of the tree correspond to directories of the file system; its leaves correspond to files. Edges represent containment relationships between directories and sub-directories or files. All FST edges are assigned a weight equal to one. During the scan, Minersoft ignores a *stop list* of files and directories that do not contain information of interest to software search (e.g., `/tmp, /proc`).

Classification and pruning: Names and pathnames play an important role in file classification and in the discovery of associations between files. Accordingly, Minersoft normalizes filenames and pathnames of FST vertices, by identifying and removing suffixes and prefixes. The normalized names are stored as metadata annotations in the FST vertices. Subsequently, Minersoft applies a combination of system utilities and heuristics to classify each FST file-vertex into one of the following categories: binary executables, source code[1] (e.g. Java, C++), libraries, software-description documents (e.g. man-pages, readme files, html files) and irrelevant files.

Minersoft prunes all FST leaves found to be irrelevant to software search, dropping also all internal FST vertices that are left with no descendants. This step results to a pruned version of the FST that contains only software-related file-vertices and the corresponding directory-vertices.

Structural dependency mining: Subsequently, Minersoft searches for "structural" relationships between software-related files (leaves of the file-system tree). Discovered relationships are inserted as edges that connect leaves of the FST, transforming the tree into a graph. Structural relationships can be identified by: i) Rules that represent expert knowledge about file-system organization, such as naming and location conventions. For instance, a set of rules link files that contain *man-pages* to the corresponding executables. *Readme* files are linked to related software files. ii) Dynamic dependencies that exist between libraries and binary executables. Binary executables and libraries usually depend on other libraries that need to be dynamically linked during runtime. These dependencies are mined from the headers of libraries and executables and the corresponding edges are

[1]Executable scripts (e.g. python, perl, bash) are also source code.

inserted in the graph; each of these edges is assigned a weight of 1, as there exists a direct association of files.

The structural dependency mining step produces the first version of the SG, which captures software resources and their structural relationships. Subsequently, Minersoft seeks to enrich file-vertex annotation with additional metadata and to add more edges into the SG, in order to better express content associations between software resources.

Keyword scraping: In this step, Minersoft performs deep content analysis for each file-vertex of the SG, in order to extract its descriptive keywords. This is a resource-demanding computation that requires the transfer of all file contents from disk to memory, to perform content parsing, stop-word elimination, stemming and keyword extraction. Different keyword-scraping techniques are used for different types of files: for instance, in the case of source code, we extract keywords only from the comments inside the source, since the actual code lines would create unnecessary noise without producing descriptive features. Details on how the scraping and content analysis was implemented, can be found in Chapter 6.

Binary executable files and libraries contain strings that are used for printing out messages to the users, debugging information, logging etc. All this information can be used in order to get useful features from these resources. Minersoft parses the binary files byte by byte and captures the printable character sequences that are at least four characters long and are followed by an unprintable character.

The extracted keywords are saved in the zones of the file-vertices of the SG.

Keyword flow: Software files (executables, libraries, source code) usually contain little or no free-text descriptions. Therefore, content analysis typically discovers very few keywords inside such files. To enrich the keyword sets of software-related file-vertices,

Minersoft identifies edges that connect software-documentation file-vertices with software file-vertices, and copies selected keywords from the former into the zones of the latter.

Content association mining: Similar to [13] and [44], we further improve the density of SG by calculating the cosine similarity between the SG vertices of source files. To implement this calculation, we represent each source-file vertex as a weighted term-vector derived from its source-code comments. To improve the performance of content association mining, we apply a feature extraction technique to estimate the quantity of information of individual terms and to disregard keywords of low value. Source codes that exhibit a high cosine-similarity value are joined through an edge that denotes the existence of a content relationship between them.

Inverted index construction: To support full-text search for software resources, Minersoft creates an inverted index of software-related file-vertices of the SG. The inverted index has a set of terms, with each term being associated to a "posting" list of pointers to the software files containing the term. The terms are extracted from the zones of SG vertices.

In the subsequent sections, we provide more details on the algorithms for finding relationships between documentation and software-related files (Section 5.1), keyword extraction and keyword flow (Section 5.2), and content association mining (Section 5.3).

5.1 Context Enrichment

During the structural dependency mining phase, Minersoft seeks to discover associations between i) documentation (text files) and software leaves (binary executables,

libraries and scripts) ii) library dependencies between binary executables and binary libraries iii) interlibrary dependencies between binary libraries. These associations are represented as edges in the SG and contribute to the enrichment of the context of software resources. As we have mentioned before, the edges are "typed". In this context, the *type* of the edges denotes the semantic relationship between two connected files.

Man pages and other text files: The discovery of such associations is relatively straightforward in the case of Unix/Javadoc online manuals since, by convention, the normalized name of a file storing a manual is identical to the normalized file name of the corresponding executable. In the same manner, other text files (like html and .txt) can be associated with binaries, libraries and scripts. Minersoft can easily detect such a connection and insert an edge joining the associated leaves of the file-system tree. The association represented by this edge is considered strong and the edge is assigned a weight equal to 1. The type of the edge for this kind of associations is called "describes" because, manual pages and text files describe binary executables, libraries and scripts.

Readme files: In the case of *readme* files, however, the association between documentation and software is not obvious: software engineers do not follow a common, unambiguous convention when creating and placing readme files inside the directory of some software package. Therefore, we introduce a heuristic to identify the software-files that are potentially described by a readme, and to calculate their degree of association. The key idea behind this heuristic is that a readme file describes its siblings in the file-system tree; if a sibling is a directory, then the readme-file's "influence" flows to the directory's descendants so that equidistant vertices receive the same amount of influence and vertices that are farther away receive a diminishing influence. If, for example, a readme-file leaf v^r has a vertex-set V^r of siblings in the file-system tree, then:

- Each *leaf* $v_i^r \in V^r$ receives from v^r an "influence" of 1.
- Each leaf f that is a descendant of an internal node $v_k^r \in V^r$, receives from v^r an "influence" of $1/(d-1)$, where d is the length of the FST path from v^r to f.

The association between software-file and readme-file vertices can be computed easily with a simple linear-time *breadth-first search* traversal of the FST, which maintains a stack to keep track of discovered readme files during the FST traversal. For each discovered association we insert a corresponding edge in the SG; the weight of the edge is equal to the association degree. The type for this kind of edges is also, "describes".

Library dependency mining: Binary executables require libraries in order to run (binary to library dependencies). Libraries require other libraries in order to provide their functionality (inter-library dependencies). Minersoft detects those dependencies using the operating system's dynamic linker (`ldd`). For every dependency that is found, an edge is added to the graph. Dependencies between libraries and executables are considered very strong so the edges representing them, have a weight of 1. The type of these edges is "depends on" because binaries and libraries depend on other libraries.

5.2 Content Enrichment

Minersoft performs the "keyword-flow" step, which enriches software-related vertices of the SG with keywords mined from associated documentation-related vertices. The keyword-flow algorithm is simple: for all software-related vertices v, we find all adjacent edges $e_d = (v, y)$ in the SG, where y is a documentation vertex. For each such edge e_d, we attach a documentation *zone* to v.

As we referred in the previous section, each software file is described by a number of zones. A zone includes a set of keywords. If there is an edge in G between a software-description document (i.e., readme, manual) and a software file (i.e., executable file, library, source code), then we enrich the content of the software file by adding a new zone. Such an action improves keyword-based searching since software files contain little or no free-text descriptions. So, the software files are represented by a number of zones. However, each zone has a different degree of importance in terms of describing the content of a software file. For instance, the *zone* of a vertex v is more important for the description of v than the other *zones* which are derived from software documentation files associated to v. Thus, each $zone_l(v)$ is assigned a weight g_l so that $Z = \sum_{l=1}^{z_v} g_l = 1$, where z_v is the total number of zones for a software file v. The weight of each zone is computed as follows: the weight of *zone* which includes the textual content of v takes the value α. The weights of the other zones of each file are determined by the edge weights of the SG G that has been occurred by exploiting the file-system tree, multiplied by α. The value of α is a normalization constant calculated so that the sum of the weights of the zones attached to each vertex equals 1. Recall that a software file is enriched by a zone if there already exists an edge between this file and a software-description document. Each zone includes the selected terms of the underlying software-description document.

5.3 Content Association

Minersoft enriches the SG with edges that capture content association between source-code files in order to support, later on, the automatic identification of software packages in the SG. This kind of edges is called “isSimillarTo” edges, because they denote the similarity between two documents.

To this end, we represent each source file s as a weighted term-vector $\overrightarrow{V}(s)$ in the Vector Space Model (VSM). We estimate the similarity between any two source-code files s_i and s_j as the cosine similarity of their respective term-vectors: $\overrightarrow{V}(s_i) \cdot \overrightarrow{V}(s_j)$. If the similarity score is larger than a specific threshold (for our experiments we have set the $threshold \geq 0,05$), we add a new typed, weighted edge to the SG, connecting s_i to s_j. The weight w of the new edge equals the calculated similarity score.

The components of the term-vectors correspond to terms of our dictionary. These terms are derived from comments found inside source-code files and their weights are calculated using a TF-IDF weighing scheme. To reduce the dimensionality of the vectors and noise, we apply a feature selection technique in order to choose the most important terms among the keywords assigned to the content zones source files. Feature selection is based on the *quantity of information* $Q(t)$ metric that a term t has within a corpus, and is defined by the following equation: $Q(t) = -log_2(P(t))$, where $P(t)$ is the observed probability of occurrence of term t inside a corpus [42]. In our case, the corpus is the union of all content zones of SG vertices of source files. To estimate the probability $P(t)$, we measure the percentage of content zones of SG vertices of source files wherein t appears; we do not count the frequency of appearance of t in a content zone, as this would create noise.

Subsequently, we drop terms which their quantity of information values from the content zones of SG vertices of source files are lower than a specific threshold (for our experiments we remove the terms where $Q(t) < 3,5$). The reason is that low-Q terms would be useful for identifying different classes of vertices. In our case, however, we already know the class where each vertex belongs to (this corresponds to the type of the respective file).

Therefore, by dropping terms that are frequent inside the source-code class, we maintain terms that can be useful for discriminating between files inside a source-code class.

Chapter 6

Implementation Details

In this Chapter we are presenting details about Minersoft's most critical components. Open Source tools are presented with details on how they were used while building Minersoft. More specifically, we are going to present the Job Management, Graph processing & storage and indexing & text processing.

Minersoft is implemented using many Open Source tools, each for a specific reason. Grid/Cloud job management is being done with Ganga [18], full text indexing and searching is being done with Apache Lucene [2]. In-memory Graph processing is being done with JUNG [6]. Python and Bash are used for general-purpose scripting and "gluing" different subsystems together.

6.1 Job Management with Ganga

Job Management is one of the most critical elements of any system that is working over a distributed Grid/Cloud infrastructure. Job creation and submission can be very tricky in the presence of failures. In the case of Minersoft, thousands of jobs have to be sent and their outputs to be retrieved, thus an automated system that will take over these

actions is needed. Ganga is an easy-to-use framework for job definition and management, implemented in Python. The Ganga API can be used to write scripts in Python to support customized job creation and management.

A job in Ganga is constructed from a set of building blocks. All jobs must specify the software to be run (application) and the processing system (backend) to be used. The backends used by Minersoft is the gLite middleware backend (for Grid infrastructures) and the SSH shell backend (for the cloud infrastructures). Jobs can specify an input dataset to be read and/or an output dataset to be produced. Ganga provides a framework for handling different types of application, backend, dataset, splitter and merger, implemented as plugin classes.

Ganga's `Job` object is used to create a job and submit it. The following example (Listing 1) is used to create crawler jobs with a number of environmental variables (`job.applcation.env`). The code is very simple and uses only the basic functionality of Ganga.

```
1  def prepareCrawlers():
2      config = ConfigObj("../config/minersoft.ini")
3
4      #Get the sites list
5      sites = open(config['crawler']['sites_to_visit_file'], "r")
6      if not sites:
7          print "Could not read the sites list file ("
8          + config['crawler']['sites_to_visit_file'] + ")..."
9
10         return False;
11
12     site = sites.readline()[:-1]
13     while site:
14         print              'Preparing a Job for: ' + site
15         job               = Job()
16         job.application   =
```

```
17              Executable(exe=File(config['crawler']['code_dir']
18                      +'Crawler.sh'), args=[''])
19          job.name            = site.split(':')[0]
20                                + '_crawler'
21
22          job.application.env = {
23                      'MINERSOFT_SITE_NAME' : site.split(':')[0],
24                      'MINERSOFT_STORAGE_URL' :
25                      config['crawler']['storage_dir']
26                              + '/' + site.split(':')[0] + '/crawls',
27                      'LFC_HOST'              : 'lfc.phy.bg.ac.yu'
28           }
29
30          job.backend.CE      = site
31
32          print               'done'
33          site                = sites.readline()[:-1]
34      return True
```

Listing 1: Crawler jobs preparation for a list of Grid sites

Failure management: Failures are very common in a Grid infrastructure like EGEE. There are many reasons for a job to fail. Network errors, expired certificates, hardware failures and site misconfigurations are the most common reasons. Ganga can detect job failures and set a job's status to `failed`. In order to decide which jobs have to be re-submitted, a script has to be created to check the jobs' status and resubmit the failed ones. It is very simple to do that with Ganga. A single line of code can take over this task `jobs.select(status='failed').resubmit()`. Jobs that keep failing are dropped and the site that is responsible to run those jobs is put into a blacklist in order to avoid sending future jobs that are very likely to fail.

6.2 Software Resource Tagging & Categorization

As we have mentioned in Chapter 3, each Software Graph vertex v is characterized by a set of tags that describe its content. Heuristics based on the output of the linux *file* command and the filename of the software resource, are used to extract tags. Before we continue on how this is done, we present the Linux `file` command and the way that we use it to extract information about files.

The Linux `file` command: Linux `file` command tests a file in an attempt to classify it. There are three sets of tests performed in this order: filesystem tests, magic number tests, and language tests. The first test that succeeds causes the file description to be printed. The description printed will usually contain one of the words: *text* (the file contains only printing characters and a few common control characters and is probably safe to read on an ASCII terminal), *executable* (the file contains the result of compiling a program in a form understandable to some UNIX kernel or another), or *data* meaning anything else (data is usually 'binary' or non-printable). Exceptions are well-known file formats (tar archives, shell scripts, binary linux executables, libraries etc) that are known to contain specific data. By parsing the output of the `file` command, one can extract useful tags of a file and use them to categorize it. Note that the `file` command is not 100% accurate. It works in a best effort manner as it uses heuristics.

In order to tag a file, we use both the file's filename and the `file` command. However, if the filename's extension denotes the file's type then checking with the `file` command is not necessary. Avoiding the use of the `file` command, the processing effort (needed in order to classify a file) is reduced. Note that checking a filename with a regular expression is much "cheaper" (in terms of processing) than executing a linux command.

Below we present the details of each of the heuristics used to extract tags.

- **Binary Executable:** Binary executables are described by the `file` as ‘‘`ELF, LSB executable`’’. Binary executables cannot be recognized by their filename because there is no naming convention (like in Windows, where binary executables are called .exe files).

- **Binary Library:** Binary libraries are recognized by their filename suffix (.so, .la, .a, etc). More information about the man page filename conventions were covered in Chapter 3, in Definitions section.

- **Man page:** Man pages are files contained in directories named man1 to man9. Files inside those directories are considered man pages only if the `file` describes them as “troff input text”. More information about the man page filename conventions were covered in Chapter 3, in Definitions section.

- **Text:** Text files are files that contain text in any language. `file` describes them as “text”. Source code files, readme files and man pages are also considered text.

- **Source Code:** Source code files are files written in any of the Minersoft-recognized programming languages (see Chapter 6 for details). The extension of the filename itself denotes the language that the source is written in.

- **Readme:** Readme files are named “readme” followed by a filename extension. For example “readme.txt” or “readme.en.txt”.

- **Script:** Scripts written in languages recognized by Minersoft have a file extension that denotes the language in which the script is written in. Also, Scripts are described by `file` as “text, script”. There are cases where the script does not denote

its source language by the filename's extension. In that case, the file command is used.

In Listing 2 we present the regular expressions that were used to assign tags to files.

```
1 man_filename = ".*man[0-9]*\/.*(\.[0-9]?.*)?(\.gz)?"
2 lib_filename = ".*\.so$|.*\.so\..*$|.*\.la$|.*\.la\..*$|.*\.a$"
3 script_filename = ".*\.(py|sh|pm|pl|rb|tcsh|csh|bash|zsh|ksh|js|php)$"
4 source_filename = ".*\.(java|c|cpp|h|hpp)$"
5 readme_filename = ".*readme.*"
6 text_filename    = ".*\.txt$"
7 script_description = ".*script.*"
8 text_description   = ".*text.*"
9 bin_description    = ".*ELF.*|.*LSB.*|.*80[2-6]86.*|.*x86_64.*"
```

Listing 2: Regular Expressions used to assign tags to files

In Listing 3, we show the function used to test each file's filename and description (the output of the `file` command) against the aforementioned regular expressions. If a regular expression matches, then the tag is assigned. In the end, the function returns the whole set of tags that is later used for file categorization.

```
1  def assign_tags(self, filename, description=None):
2      text,man,bin,source,script,readme,lib = 0,0,0,0,0,0,0
3      if self.lib_filename.match(filename):
4          lib = 1
5          self.lib_files +=1
6      if self.man_filename.match(filename):
7          man = 1
8          self.man_files +=1
9      if self.bin_description.match(description) and not lib==1:
10         bin = 1
11         self.bin_files +=1
12     if self.readme_filename.match(filename):
13         readme = 1
14         self.readme_files +=1
```

```
15      if self.script_filename.match(filename) or self.script_description.match(
            description):
16          script = 1
17          self.script_files +=1
18      if self.source_filename.match(filename) or script==1:
19          source = 1
20          self.source_files +=1
21      if self.text_filename.match(filename) or self.text_description.match(
            description) or readme==1 or source==1 or man==1:
22          text = 1
23          self.text_files +=1
24      return text,man,bin,source,script,readme,lib
```

Listing 3: Use of regular expressions in order to assign tags

6.3 Software Graph Storage & Processing

During the process of implementing Minersoft, it has been proven very difficult to build a single, uniform data structure to store all of the SG's vertices, attributes, edges and inverted indexes. A very important problem is the fact that the Software Graph itself cannot fit in main memory. The SG (including full text indexes, vertices and edges) for 20 computational nodes (Chapter 7) reaches 400GBs in size. External memory has to be used to store a graph of that size. Instead of keeping the graph in a single data structure, we separate the SG of each site in many files, each containing a specific part of it. Inverted indexes are kept in Lucene indexes. Edges and vertices are kept in separate compressed text files.

6.3.1 Storage in External Memory

Graph Edges: The edges are kept in tab-separated text files named according to the type of the edges contained. Minersoft keeps track of 4 types of edges: isSimilarTo, dependsOn, describes and contains. Each file contains data in the following format:

File 1	File 2	edge weight
/path/to/file1	/path/to/file2	0.34
/path/to/file2	/path/to/file4	1
...	...	...

Graph Vertices: The vertices are kept in a similar way. Each site has a tab-separated compressed text file that contains the following columns:

full path	isText	isMan	isBin	isSource	isScript	isReadme	isLibrary	file size
/bin/bash	0	0	1	0	0	0	0	325.321

Tags (text, manual page, binary executable etc) are represented as boolean values and the file size is represented in bytes.

6.3.2 In-memory Processing

Before any kind of processing, the graph has to be loaded in main memory in order to be easily and fast accessible. Algorithms such as Breadth or Depth First Traversal (used in calculating the relationships between Readme files and binaries), use the metadata of vertices as well as the edges between them. In order to be able to annotate vertices with metadata and represent typed edges, the in-memory graph implementation is based on the graph library named JUNG [6]. JUNG supports a variety of representations of entities

and their relations, such as directed and undirected graphs, multi-modal graphs, graphs with parallel edges, and hypergraphs. It provides a mechanism for annotating graphs, entities, and relations with metadata. Only basic functionality of JUNG was used.

6.4 Inverted Indexes

6.4.1 Lucene

Minersoft uses inverted indexes to search in the Software Graph. The inverted indexes are built using the Lucene [2] full-text search library. Lucene is high-performance, scalable, full-featured, open-source text indexing/searching library. Lucene makes it easy to create inverted indexes of massive text repositories and search them quickly, with little programming effort. In Lucene, every text file is represented as a `Document` and the zone inside a Document is called `Field`. Every field in Lucene can have a different weight, resulting to changes in scoring. The weight of each field in Lucene is called a `boost factor`.

Field (zone) boosting in Lucene: The default `boost factor` in Lucene is 1. A field can be "boosted" in order to change the default scoring of Lucene. Numbers less than 1 result to lower score and more than 1 result to higher score during ranking. The weights of the edges in the SG, have been used to set the boost factor of the *documentation* fields.

6.4.2 Content Parsers & Analyzers

During the indexing process, the text of the various data types that are being indexed (like source code and manual pages) have to be parsed in order to extract their text. In order to do that, a number of *parsers* had to be implemented. The parsers are used before text is passed to Lucene for indexing. The first step towards parsing a file is to extract

it's meaningful text (source code comments, html text without tags etc.). Afterwards, the text has to be analyzed, according to some heuristics. Analysis of text includes:

- **Stemming**: Terms are stemmed using the porter stemmer algorithm, in order to achieve higher recall.
- **Stop words elimination**: Stop words are very common english words that exist in almost every english text. Searching for so many common words affects the precision of an IR system. For example searching for the word "the", too many results would be returned that the user is not interested in. Therefore, during the analysis of text, stop words are eliminated, thus reducing the risk of returning irrelative results on user queries.
- **Lower-case Word Conversion**: It is very common to capitalize words in text. Sometimes, words are capitalized because of grammar rules and other times in order to highlight some of the letters of words. In Minersoft, terms are converted to lower-case in order to avoid misspellings in the case of users submitting queries with the wrong capitalization. For example, if a dataset contains the words "LONDON" and "London" and a user is searching for "LonDon" no results will be returned.

HTML Parser

A very simple example of a Minersoft parser is the html parser. The HTML parser, filters out all html tags keeping the actual HTML text. HTML tags are considered noise in Software search and that is the reason that they are filtered out. Below is the code used to do the text extraction:

```
1 package ucy.cs.hpcl.minerSoft.util;
```

```
2  import java.util.regex.*;
3
4  public class HTML2Text {
5    public static Pattern pattern =
6            Pattern.compile("<[^>]*>|\\&[^;]*;");
7
8    public static String htmlToText(String html){
9      return pattern.matcher(html).replaceAll("␣");
10   }
11 }
```

Listing 4: HTML2Text.java

Comments Extractor Parser

For each source code file, Minersoft, indexes only its comments. Every programming language has its own grammar, thus, in order to extract comments from many different languages, a parser for each of the languages has to be implemented. Most of the times, a simple regular expression is enough to extract comments from source code. However there are some cases that regular expressions are not enough and parsing has to be done with language-grammars. One example of an imperfect comment extraction is the following (Listing 5):

```
1 printf("␣/*␣");
2 for (i = 0; i < 100; i++) {
3     a += i;
4 }
5 printf("␣*/␣");
```

Listing 5: Imperfect comment matching

In Minersoft, we implemented the parsing with regular expressions as writing grammars for many programming languages would take a lot of effort.

The languages that Minersoft was designed to get comments from, are listed in Table 4. Every language has its own syntax for inserting comments into source code. Every source code file states the language in which it is written (by the suffix of its filename). For example in *C* the suffix of the source filename is ".c" or ".h". In *Python* the suffix is ".py". However, in many cases, scripts do not state their language by their suffix. In *Bash* and many other shell scripting languages like *Tcsh*, the first line of the file states the interpreter that will be needed to run the file. For example a Bash script starts like this:

```
1 #/bin/env bash
2 ...
```

Listing 6: Bash file header

Programming Language	**Types of comments**
Java, C, C++, Javascript	//, /**/
PHP	// , /**/ , #
Python	#, ""
Ruby	# , **begin ... end**
Awk, Perl, Tcsh, Bash	#

Table 4: Programming languages and their comment types

Below (Listing 7) is part of the code that contains the regular expressions used to identify and extract source code comments.

```
1 class CommentsExtractor:
2
3     def __init__(self):
4         self.re_lang_mapping = {'':''}
5
```

```
6           slash_comments       = "//.*"
7           hash_comments        = "#.*"
8           slash_star_comments = "/\*(?:.|[\r\n])*?\*/"
9           begin_end_comments  = "=begin(?:.|[\r\n])*=end"
10          quote_comments       = "'''(?:.|[\r\n])*?'''"
11  ...
12  ...
13      def extract(self, filename):
14          ext = os.path.splitext(filename)[1]
15          file = open(filename, 'r')
16          text = file.read()
17
18          if self.re_lang_mapping[ext] :
19              regexp = self.re_lang_mapping[ext]
20          else:
21              regexp = self.script_comments
22
23          for match in regexp.findall(text):
24              print match
25
26  if __name__ == '__main__':
27              extractor = CommentsExtractor()
28              extractor.extract(sys.argv[1])
```

Listing 7: CommentsExtractor.py

Chapter 7

Evaluation

7.1 Testbed

The usefulness of the findings of any study depends on the realism of the data upon which the study operates. For this purpose, the experiments are conducted on a) 10 Grid sites of EGEE, one of the largest Grid production services currently in operation and b) 10 Virtual Servers in 2 commercial Cloud providers (6 Virtual Servers of Amazon Elastic Computing Cloud provider, and 4 Virtual Servers of Rackspace Cloud provider). Tables 5, 6 present the Grid and Cloud sites that have been crawled and indexed by Minersoft respectively.

7.2 Crawling and Indexing Evaluation

In this section, we elaborate on the performance evaluation of the crawling and indexing tasks of Minersoft. Our objective is to show that Minersoft works sufficiently on the examined real, large-scale Grid and Cloud testbed.

Grid Site	# of Files	Size (GB)
ce01.kallisto.hellasgrid.gr	3.541.403	247,9107344
ce301.intercol.edu	97.906	3,53996803
grid-ce.ii.edu.mk	194.556	3,731716045
paugrid1.pamukkale.edu.tr	132.645	10,34736117
ce01.grid.info.uvt.ro	270.445	2,693567113
grid-lab-ce.ii.edu.mk	109.286	18,63437903
ce01.mosigrid.utcluj.ro	70.419	61,88081632
ce101.grid.ucy.ac.cy	1.278.851	6,375885554
ce64.phy.bg.ac.yu	150.661	6,787445784
testbed001.grid.ici.ro	125.028	7.117.152,75
Total	5.971.200	366,6529735

Table 5: Grid Testbed.

7.2.1 Examined metrics

To assess the crawling and indexing in Minersoft, we investigate the performance of crawler and indexer jobs; recall that each job is responsible for a number of files (called splits) that exist on a Grid site/Cloud Virtual Server. In this context, we use the following metrics:

- Run time: the average time that a crawler/indexer job spends on a Grid site/-Cloud Virtual Server, including processing and I/O; this metric measures the average elapsed time that Minersoft needs to process (crawl or index) a split.

- CPU time: the average CPU time in seconds spent by a crawler/indexer job while processing a split on a Grid site/Cloud Virtual Server.

- File rate: the number of files that Minersoft crawls/indexes per second on a Grid site/Cloud Virtual Server.

Cloud Virtual Server	# of Files	Size (GB)
Amazon1	35.362	0,644329774
Amazon2	29.980	0,656549389
Amazon3	25.906	1,020230504
Amazon4	31.049	1,327836617
Amazon5	83.256	2,102089881
Amazon6	33.522	0,789997322
Rackspace1	24.035	1,109502204
Rackspace2	17.266	0,661989137
Rackspace3	13.589	0,537499689
Rackspace4	47.156	2,306268435
Total	341.121	11,15629295

Table 6: Cloud Testbed.

- Size rate: the size of files in bytes that Minersoft crawls/indexes per second on a Grid site/Cloud Virtual Server.

In our experiments, each crawler and indexer job was configured to run with five threads. We also ran experiments with different numbers of threads (from 1, 5, 9 to 13) and concluded that 5 threads per crawler/indexer job provide a good trade-off between crawling/indexing performance and server workload. Smaller or larger numbers of threads per crawler/indexer job usually result to significantly higher run times, due to poor CPU utilization or I/O contention, respectively. Recall, that the crawler and indexer jobs process a specific number of files, called *splits*. In our experiments, each split processes from 1 to 100.000 files.

7.2.2 Crawling Evaluation

Figures 6, 7 depict the *per-job average run-time* and *per-job average CPU-time* for crawling the Grid sites/Cloud Virtual Servers. The per-job CPU time takes into account the total time that all the job's threads spend in the CPU. The run-time values are

significantly larger than the CPU times due to the system calls and Input/Output that each crawler performs while processing its file split. I/O is much more expensive in the case of Grid sites with shared file systems. Another observation is that the run-time and CPU-time of crawler jobs vary significantly across different Grid sites. This imbalance is due to several factors, including the hardware heterogeneity of the infrastructure, the dynamic workload conditions of shared sites, and the dependence of the crawler processing on site-dependent aspects. For example, the crawler performs expensive "deep" processing of binary and library files to deduce their type and extract dependencies. This is not required for text files. Consequently, the percentage of binaries/libraries found in each site determines to some extent the corresponding crawling computation. In the case of Cloud providers, we observe that the performance of Cloud Virtual Servers of the same Cloud providers vary significantly. At a first sight, this is quite surprising since Cloud providers do not have hardware heterogeneity among their Cloud Virtual Servers. These changes in run-time and CPU-time of crawler jobs of Cloud Virtual Servers are mainly due to the dependence of the crawler processing on sever-dependent aspects (i.e., file types).

Tables 9, 10 depict the throughput achieved by the Minersoft crawler on different Grid sites and Cloud Virtual Servers respectively, expressed in terms of the number of files and the number of bytes processed. Both Grid sites and Cloud Virtual Servers achieve high performance. However, the crawling of a Grid site, in general, takes longer time than the crawling of a Cloud Virtual Server. This is explained by the fact that the number of files in a Cloud Virtual Server is much smaller than a Grid site.

The files found by the crawlers to be irrelevant to software search are pruned from subsequent processing. Figures 8, 9 present the percentage of files that have been dropped in Grid sites and Cloud Virtual Servers respectively. We observe that a large percentage of

content in most Grid sites/Cloud Virtual Servers includes software resources. Specifically, on average 75% (75% of total files' size) and 70% (77% of total files' size) of total files that exist in Grid sites and Cloud Virtual Servers have been categorized as software resources. These findings confirm the need to establish advanced software discovery services in Grid and Cloud infrastructures. The software-related files are categorized with respect to their type. From Tables 7, 8, we can see that most software-related files in the Grid/Cloud infrastructure are documentation files (man-pages, readme files, html files) and sources. Sources are files written in any programming language. Executable scripts (e.g. python, perl, bash) are also considered as sources (e.g. Java, C++). Finally, each crawler stores within the storage element of its site a *metadata store file* capturing the file-id, name, type, path, size and structural dependencies of the identified software resources. Tables 16, 17 present the number of splits that have been created in order to crawl the files.

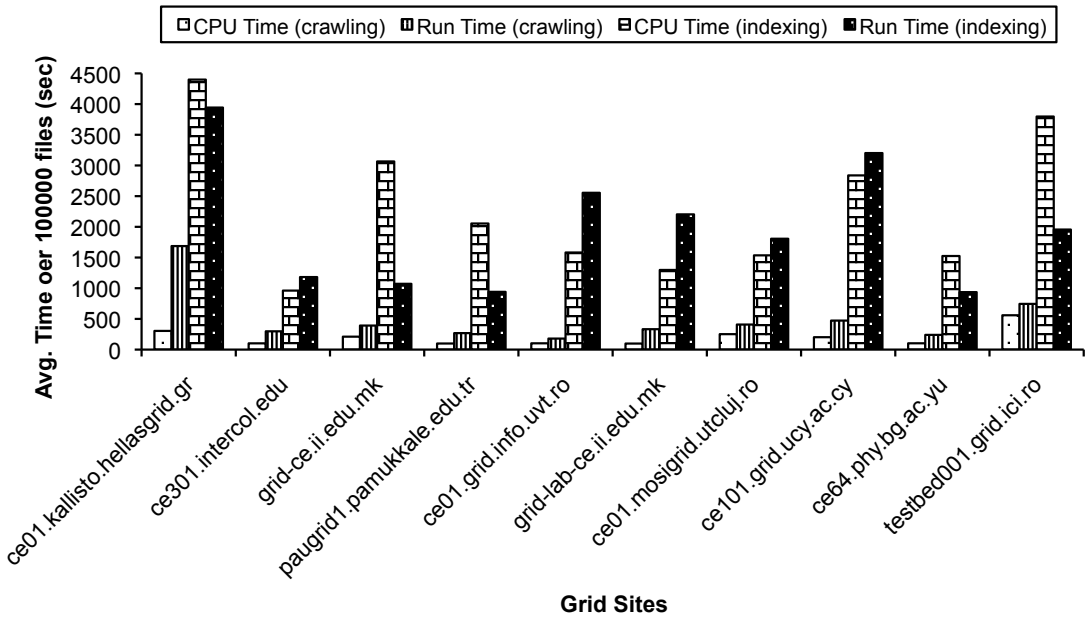

Figure 6: Average times for jobs in Grid Infrastructure.

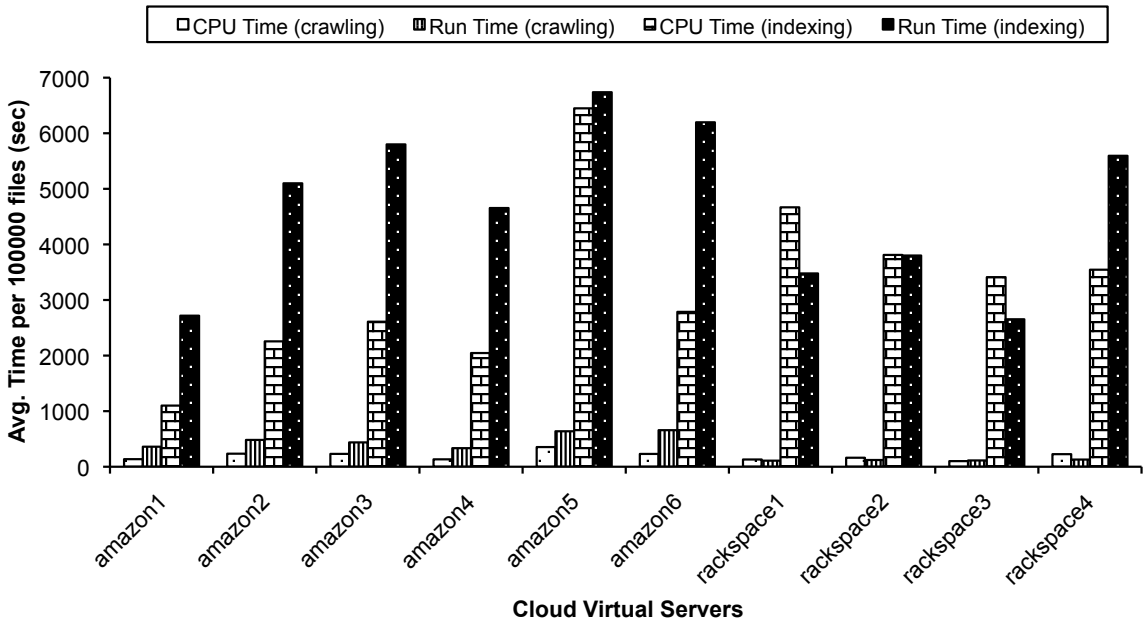

Figure 7: Average times for jobs in Cloud Infrastructure.

7.2.3 Indexing Evaluation

Figures 6, 7 depict the *per-job average run-time* and the *per-job CPU time* for indexing Grid sites/Cloud Virtual Servers. As expected, we observe that indexing is more computationally-intensive than crawling, since we need to conduct "deep" parsing inside the content of all files.

Removing the duplicate files via the *duplicate reduction policy* leads to reducing either the number of splits or the number of files within splits since we found that in our data set about 11% of files belong to more than one Grid sites. In case of Clouds, we observe that replication is larger than Grids. Our findings showed that 32% of files belong to more than one Cloud Virtual Servers. Note that the percentage of duplicate files depends on the examined testbed. For instance, in a previous study [33] we observed that 33% of files belongs to more than one Grid sites. So, the main observation of these findings is that there is a large number of duplicate files in EGEE. So, the main observation of these

Grid Site	Binaries	Sources	Libraries	Docs	Irrelevant
ce01.kallisto.hellasgrid.gr	41.990	1.407.701	142.873	1.672.246	276.593
ce301.intercol.edu	34.134	8.972	3.724	23.536	27.540
grid-ce.ii.edu.mk	16.869	69.915	8.080	61.469	38.223
paugrid1.pamukkale.edu.tr	7.383	47.388	7.935	43.861	26.078
ce01.grid.info.uvt.ro	8.999	40.442	3.778	42.652	174.574
grid-lab-ce.ii.edu.mk	7.703	46.116	2.983	37.333	15.151
ce01.mosigrid.utcluj.ro	17.828	12.475	2.310	18.091	19.715
ce101.grid.ucy.ac.cy	26.377	433.115	37.463	672.211	109.685
ce64.phy.bg.ac.yu	6.047	31.889	7.672	67.388	37.665
testbed001.grid.ici.ro	29.261	22.961	6.120	28.239	38.447
Total	196.591	2.120.974	222.938	2.667.026	763.671

Table 7: Files Categories in Grid sites of EGEE.

findings is that there is a large number of duplicate files in Grid and Cloud infrastructures. Tables 16, 17 present the number of splits and the size of inverted file indexes in each Grid site and Cloud Virtual Server respectively. In order to study the benefits of duplicate reduction policy, we also present the number of splits without performing duplication. From this table we observe that ce01.kallisto.hellasgrid.gr has 31 splits instead of 33 splits (including duplicate files). Moreover, even if the number of splits is not reduced, the number of files that have been assigned in a split is reduced . Consequently, the total indexing time is significantly reduced. Comparing with the crawling, we observe that less number of splits are required for indexing than crawling since the irrelevant files have been deleted. Regarding the size of inverted indexes, we present the size of inverted indexes with and without performing stemming. Details about the tradeoffs of stemming are presented in the next section, where we evaluate the software retrieval of Minersoft.

Finally, Tables 13, 14 depict the throughput of the indexer expressed in terms of the number of files and the number of bytes processed per second in each Grid site/Cloud Virtual server. In case of Grids, the performance of indexing is affected by the hardware

Cloud Virtual Server	Binaries	Sources	Libraries	Docs	Irrelevant
Amazon1	2.442	6.358	1.000	19.546	6.016
Amazon2	2.328	7.351	1.075	12.474	6.752
Amazon3	2.344	5.708	1.259	8.066	8.529
Amazon4	3.229	3.534	1.480	11.925	10.881
Amazon5	5.098	25.076	1.458	27.777	23.847
Amazon6	1.562	11.348	1.087	13.216	6.309
Rackspace1	2.900	2.859	963	4.488	12.825
Rackspace2	2.498	1.880	798	4.134	7.956
Rackspace3	2.347	2.047	684	3.491	5.020
Rackspace4	2.963	10.425	1.950	19.997	11.821
Total	27.711	76.586	11.754	125.114	99.956

Table 8: Files Categories in Amazon and Rackspace Cloud Providers.

Grid Site	File rate (files/sec)	Size rate(MB/sec)
ce01.kallisto.hellasgrid.gr	59,287	4,067
ce301.intercol.edu	335,773	6,085
grid-ce.ii.edu.mk	255,395	4,141
paugrid1.pamukkale.edu.tr	372,467	4,744
ce01.grid.info.uvt.ro	557,289	14,762
grid-lab-ce.ii.edu.mk	300,905	2,766
ce01.mosigrid.utcluj.ro	245,074	23,382
ce101.grid.ucy.ac.cy	211,604	9,577
ce64.phy.bg.ac.yu	417,031	9,075
testbed001.grid.ici.ro	134,300	3,111

Table 9: Crawling rates in Grid sites.

(disk seek, CPU/memory performance), file types, and the workload of each site. In case of Clouds, the indexing is mainly affected by file types.

To sum up, our experimentations concluded to the following empirical observations:

- Minersoft successfully crawled about 6.3 million valid files (378 GB size) and sustained high crawling rates.
- A large percentage of duplicate files exists in Grid sites/Cloud Virtual Servers. Identifying these files, the performance of indexing is significantly improved.

Cloud Virtual Server	File rate (files/sec)	Size rate(MB/sec)
Amazon1	277,436	1,830
Amazon2	207,258	1,393
Amazon3	228,307	2,385
Amazon4	299,209	4,068
Amazon5	156,699	3,373
Amazon6	152,020	1,229
Rackspace1	915,270	10,398
Rackspace2	824,152	5,586
Rackspace3	882,402	4,856
Rackspace4	774,318	18,286

Table 10: Crawling rates in Cloud Virtual Servers.

- The crawling and indexing in Grid infrastructures is significantly affected by the hardware (local disk, shared file system), file types and the current workload of Grid sites. In case of Clouds, the crawling and indexing is mainly affected by file types.

- It is important to establish advanced software discovery services in the Grid/Cloud since, in most cases, more than 70% of files that exist in the Workernodes file systems of Grid sites/Cloud Virtual Servers are software files.

7.2.4 Discussion

The experimental results showed that Minersoft crawls and indexes Grid sites/Cloud Virtual Servers in an efficient way. Despite this fact, the crawling and indexing rates of Minersoft can further be improved by investigating novel efficient policies in terms of:

- Determining the size of splits: As we referred above, the files of each Grid site/Cloud Virtual Server are split into a number of splits where the size of the split is chosen to ensure that the crawling and indexing can be distributed evenly and efficiently. Considering that in such an infrastructure the execution time and workload cannot

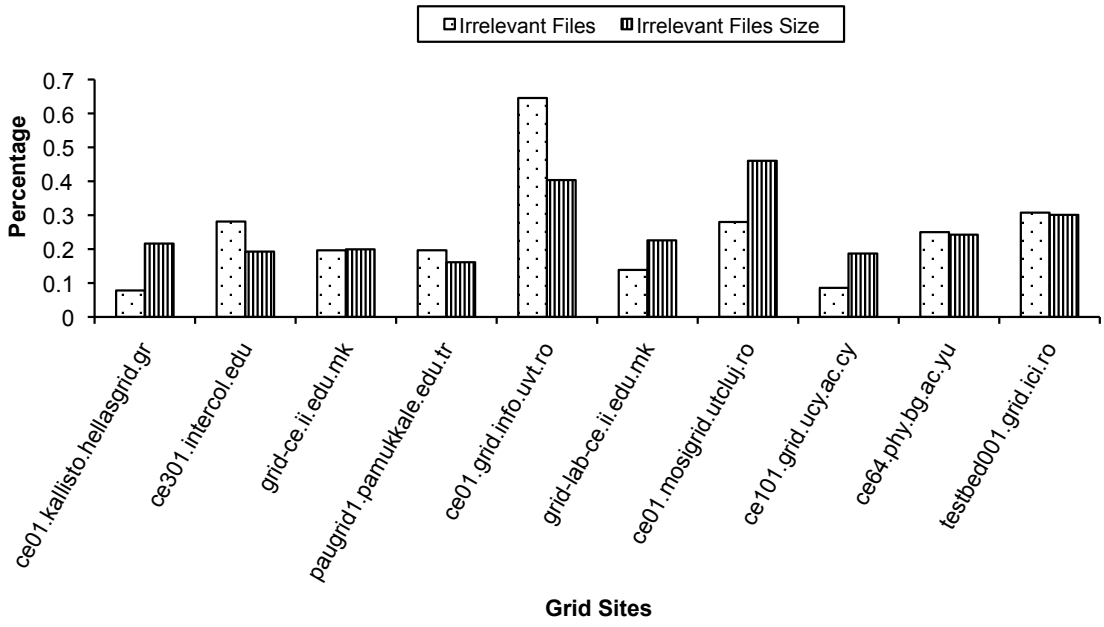

Figure 8: Percentage of irrelevant files in Grid Sites.

be determined in advance using historical data, the size of splits should be adapted to the working environment. To meet this challenge, the *monitor* should have a global view of the system's workload so as to dynamically rearrange the size of splits as well as schedule them to Grid sites/Cloud Virtual Servers.

- Determining the number of threads per crawler/indexer jobs: To improve the efficiency of crawling and indexing, Minersoft should enhance self-adaptive mechanisms in order to assign a sufficient number of threads to Grid/Cloud resources. Grid/-Cloud monitoring systems provide information about resource utilization (CPU utilization, memory utilization, disk utilization, etc.) and network connectivity in Grid sites/Cloud Virtual Servers. Thus, if the current status of a Grid site/Cloud Virtual Server has changed, the *job manipulator* should modify the number of threads per crawler/indexer jobs in this site.

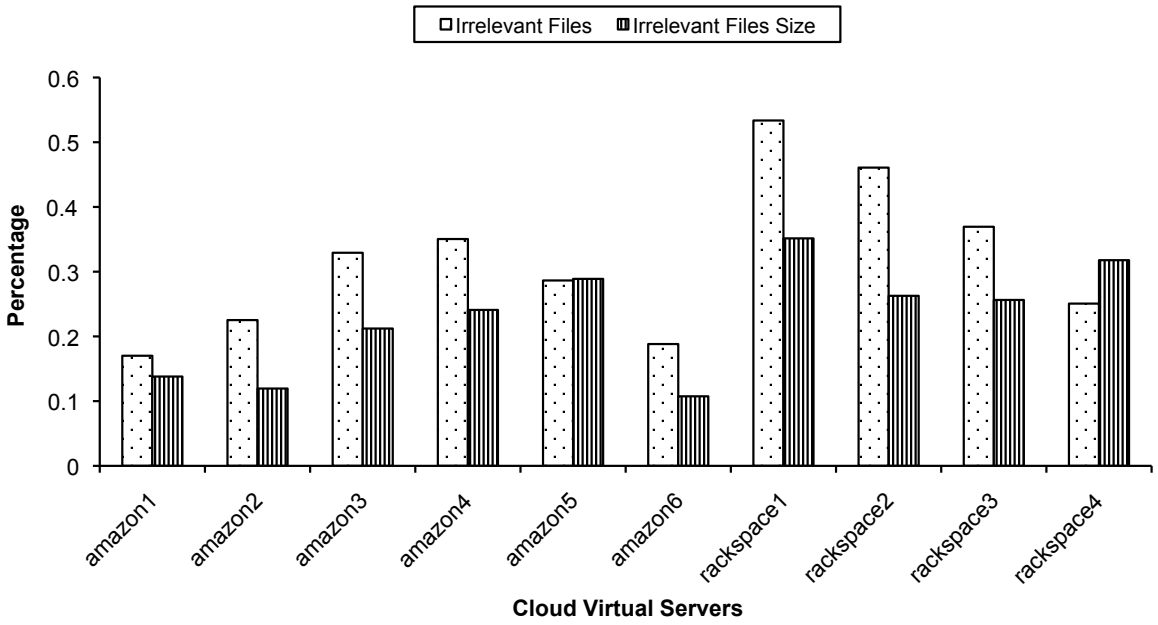

Figure 9: Percentage of irrelevant files in Cloud Virtual Servers.

- Detecting duplicate software files: Files that are exact duplicates of each other can be identified by either heuristic techniques or checksumming techniques. In order to prevent duplicate files, crawler jobs need to periodically communicate to coordinate with each other. However, this communication may result in overhead. Can we minimize this communication overhead while maintaining the effectiveness of the crawler job? Authors in [19] dealt with this problem in the context of the Web. Another issue is the identification of near-duplicate files. If we could successfully identify these files, we could improve the performance of indexing, since a percentage of files will be deleted. Manber [43] has developed algorithms for near-duplicate detection to reduce storage in large-scale file systems.

- Determining politeness: Minersoft jobs should not obstruct the normal operation of Grid sites. Minersoft should adhere to strict rate-limiting policies when accessing poorly provisioned (in terms of workload) Grid sites. To address this issue, Minersoft

Grid Sites	Crawling Statistics	Indexing Statistics			
	# of splits	# of splits	# of splits including duplicates	Inverted Index size with stemming (MB)	Inverted Index size w/o stemming (MB)
ce01.kallisto.hellasgrid.gr	37	31	33	58.129,359	60.988,246
ce301.intercol.edu	2	1	1	358,0312	395,089
grid-ce.ii.edu.mk	1	1	2	702,796	778,972
paugrid1.pamukkale.edu.tr	3	1	2	632,035	702,296
ce01.grid.info.uvt.ro	4	1	1	1152,386	1258,464
grid-lab-ce.ii.edu.mk	3	1	1	57,414	62,863
ce01.mosigrid.utcluj.ro	2	1	1	257,984	288,214
ce101.grid.ucy.ac.cy	14	10	12	13588,468	14073,417
ce64.phy.bg.ac.yu	3	1	2	871,449	964,621
testbed001.grid.ici.ro	3	1	1	646,671	715,148

Table 11: Crawling & Indexing statistics in Grid sites.

should implement a flexible policy that would avoid running multiple crawler jobs to overloaded Grid sites.

7.3 Software Graph Evaluation

In this section, we evaluate the effectiveness of the Minersoft search engine for locating software on the Grid/Cloud infrastructures. A difficulty in the evaluation of such a system is that there are not widely accepted any benchmark data collections dedicated to software (e.g., TREC, OHSUMED etc). In this context, we use the following methodology in order to evaluate the performance of Minersoft:

- *Data collection*: Our dataset consists of the software installed in 10 Grid sites of EGEE infrastructure (Table 5) and 10 Cloud Virtual Servers from Amazon Elastic Computing and Rackspace Cloud providers (Table 6). From the data collection, we exclude a *stop list* of files and directories that do not contain information of interest

Cloud Virtual Servers	Crawling Statistics	Indexing Statistics			
	# of splits	# of splits	# of splits including duplicates	Inverted Index size with stemming (MB)	Inverted Index size w/o stemming (MB)
Amazon1	1	1	1	133,113	148,062
Amazon2	1	1	1	162,558	177,042
Amazon3	1	1	1	134,777	149,710
Amazon4	1	1	1	180,570	202,433
Amazon5	1	1	1	299,585	338,593
Amazon6	1	1	1	162,160	180,417
Rackspace1	1	1	1	78,136	87,085
Rackspace2	1	1	1	84,335	94,851
Rackspace3	1	1	1	60,574	66,332
Rackspace4	1	1	1	221,902	245,011

Table 12: Crawling & Indexing statistics in Cloud Virtual Servers.

to software search (e.g., `/tmp, /proc`). Tables 7, 8 present the software resources that have been identified by Minersoft on those Grid sites/Cloud Virtual Servers.

- *Queries*: We use a collection of 28 keyword queries, which were extracted by EGEE users and by the recent submissions to the Sourcerer system [38]. These queries comprise either single- or multiple-keywords. Each query has an average of 2 keywords; this is comparable to values reported in the literature for Web search engines [53]. To further investigate the sensitivity of Minersoft, we have classified the queries into two categories: general-content and software-specific (see Table 15).

- *Relevance judgment*: A software resource is considered relevant if it addresses the stated information need and not because it just happens to contain all the keywords in the query. A software resource returned by Minersoft in response to some query is given a binary classification as either relevant or non-relevant with respect to the user information need behind the query. In addition, the result of each query has been rated at three levels of user satisfaction: “not satisfied”, “satisfied”, “very

Grid Site	File rate (files/sec)	Size rate(MB/sec)
ce01.kallisto.hellasgrid.gr	25,359	1,363
ce301.intercol.edu	84,620	1,238
grid-ce.ii.edu.mk	93,297	1,211
paugrid1.pamukkale.edu.tr	106,283	1,135
ce01.grid.info.uvt.ro	39,143	0,618
grid-lab-ce.ii.edu.mk	45,387	0,323
ce01.mosigrid.utcluj.ro	55,363	2,850
ce101.grid.ucy.ac.cy	31,231	1,149
ce64.phy.bg.ac.yu	106,818	1,760
testbed001.grid.ici.ro	51,127	0,827

Table 13: Indexing rates in Grid sites.

Cloud Virtual Server	File rate (files/sec)	Size rate(MB/sec)
Amazon1	36,793	0,209
Amazon2	19,609	0,116
Amazon3	17,242	0,141
Amazon4	21,483	0,221
Amazon5	14,840	0,227
Amazon6	16,136	0,116
Rackspace1	28,753	0,211
Rackspace2	26,314	0,131
Rackspace3	37,689	0,154
Rackspace4	17,874	0,287

Table 14: Indexing rates in Cloud Virtual Servers.

satisfied". These classifications have been done manually by EGEE administrators and experienced users and are referred to as the *gold standard* for our experiments.

7.3.1 Performance Measures

The effectiveness of Minersoft should be evaluated on the basis of how much it helps users achieve their software searches efficiently and effectively. In this context, we used the following performance measures:

Software-specific queries	General-content queries
imagemagick; octave numerical computations; lapack library; gsl library; boost c++ library; glite data management; xerces xml; subversion client; gcc fortran; thrudb; lucene; jboss; rails ruby; mpich; autodock docking; atlas software	linear algebra package; fftw library; earthquake analysis; java virtual machine; statistical analysis software; ftp client; regular expression; sigmoid function; histogram plot; binary tree; zip deflater; pdf reader

Table 15: Queries.

- *Precision*@10 reports the fraction of software resources ranked in the top 10 results that are labeled as relevant. The relevance of the retrieved results is determined by the *gold standard*. The results are ranked with respect to the ranking function of Lucene [2], which is based on the TF-IDF metric and has been used extensively in the literature for the ranking of Web-search results [15, 21]. In our case, the TF-IDF calculation is based on the zones associated by Minersoft to software files. The maximum Precision@10 value that can be achieved is 1.

- NDCG (Normalized Discounted Cumulative Gain) is a retrieval measure devised specifically for evaluating user satisfaction [30]. For a given query q, the $top - K$ ranked results are examined in decreasing order of rank, and the NDCG value is computed as: $NDCG_q = M_q \cdot \sum_{j=1}^{K=10} \frac{2^{r(j)}-1}{\log_2(1+j)}$, where each r(j) is an integer relevance label (0="not satisfied", 1="satisfied", 2="very satisfied") of the result returned at position j and M_q is a normalization constant calculated so that a perfect ordering would obtain NDCG of 1.

- NCG (Normalized Cumulative Gain) is the predecessor of NDCG and its main difference is that it does not take into account the position of the results. For a given

query q, the NCG is computed as: $NCG_q = M_q \cdot \sum_{j=1}^{K=10} r(j)$. A perfect ordering would obtain NCG of 1.

Cumulative gain measures (NDCG, NCG) and precision complement each other when evaluating the effectiveness of IR systems [11, 20]. In our evaluation metrics we do not consider the recall metric (the percentage of the number of relevant results). Such a metric requires to have full knowledge about all the relevant software resources with respect to a query. However, such a knowledge is not feasible in a large-scale networked environment.

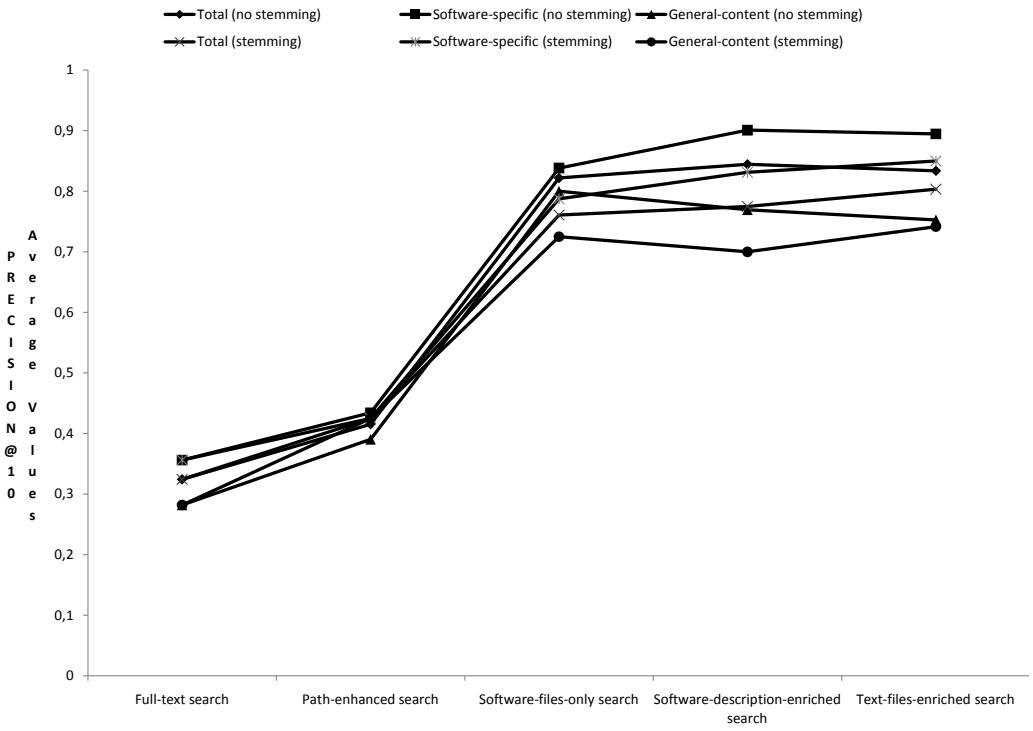

Figure 10: Precision@10 results (average values).

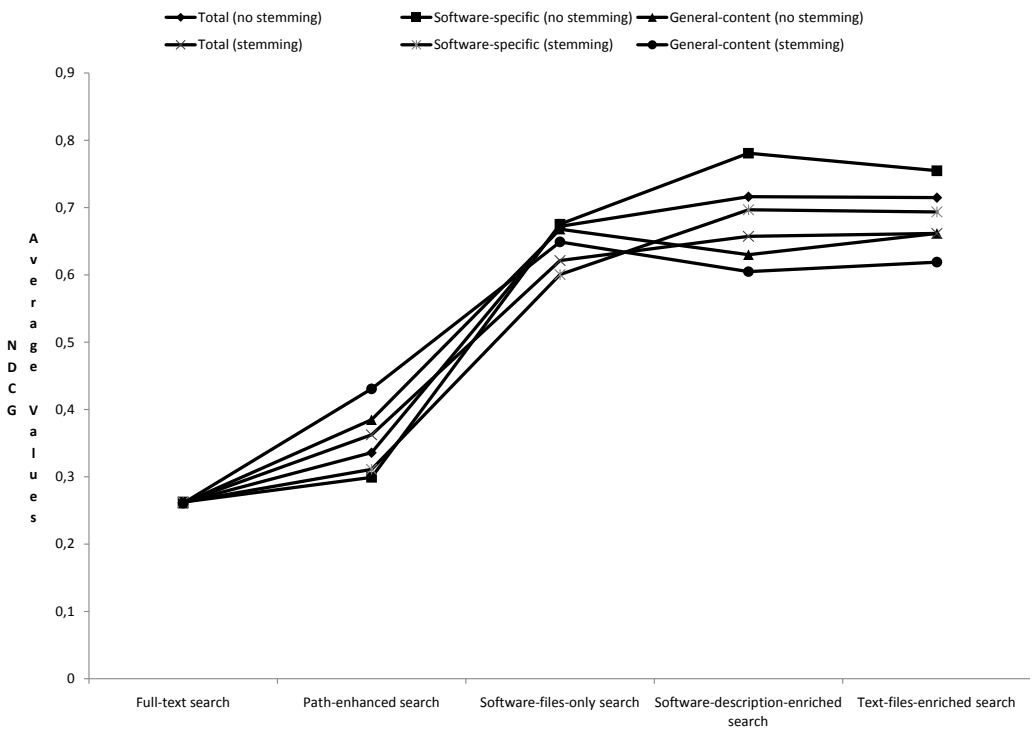

Figure 11: NDCG results (average values).

7.3.2 Evaluation Scenarios

We calculate the metrics mentioned above for the queries of Table 15, when the searches are performed against a set of different indexes representing alternative optimizations implemented inside Minersoft. In particular, we estimate the effectiveness of Minersoft in the following scenarios:

- *Full-text search*: Inverted index terms are extracted only from the full-text content of discovered files in the examined testbed infrastructure without any preprocessing. This approach is relevant to the desktop search systems (e.g., Confluence [28], Wumpus [55]). *Full-text search* is used as a baseline for our experiments.

- *Path-enhanced search*: The terms of the inverted index are extracted from the content and path zones of SG vertices. This scenario is used to evaluate the contribution of paths in software retrieval process.

- *Software-files-only search*: The files have been categorized into file categories. Software-description documents and irrelevant files (files which do not belong to any category) are discarded from the posting lists. The terms of the inverted index are extracted from the content and path of SG vertices.

- *Software-description-enriched search*: The terms of inverted index are extracted from the content of SG vertices as well as from the zones of documentation files (i.e., man-pages and readme files) and the path of SG vertices.

- *Text-files-enriched search*: The terms of the inverted index are extracted from the content, the path and the zones from the other text files of SG vertices with the same normalized filename. Recall that Minersoft normalizes filenames and pathnames of SG vertices, by identifying and removing suffixes and prefixes.

7.3.3 Evaluation

Figures 10, 11, and 12 present the results of the examined approaches with respect to the query types for *Precision*@10, NDCG and NCG. Each approach is a step towards the construction of the inverted index that is implemented in Minersoft (section 5). We conduct two set of experiments (stemming/no stemming) in order to study the effects of stemming in software retrieval. For completeness of presentation, we also present the median values of the examined metrics (Figures 13, 14, and 15). The general observation

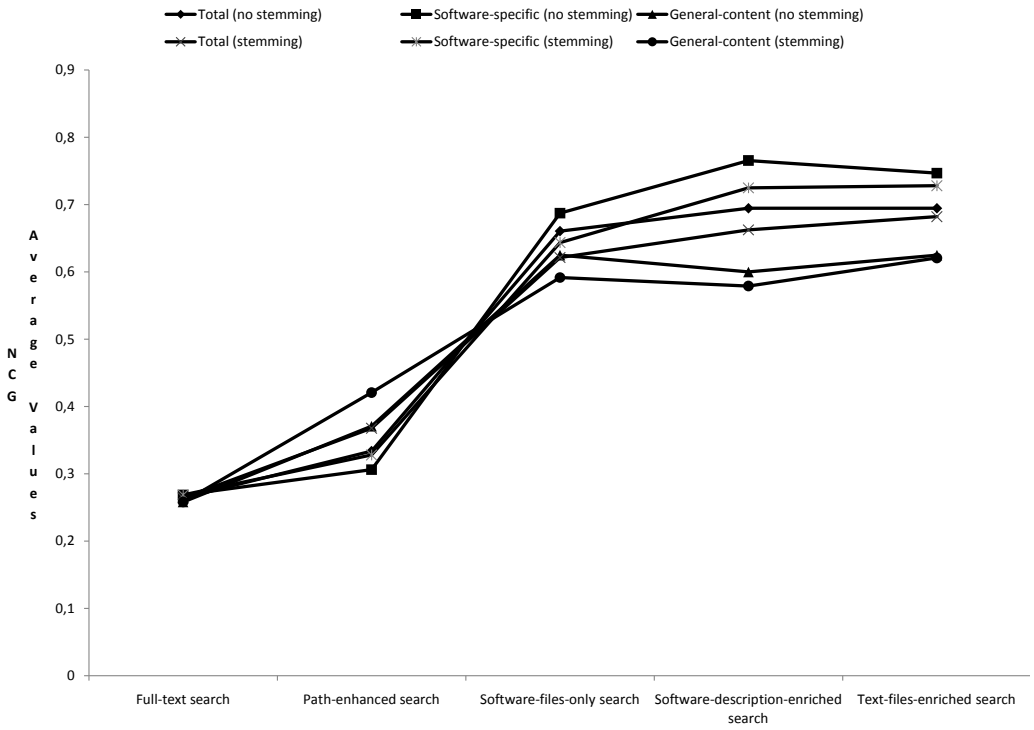

Figure 12: NCG results (average values).

is that Minersoft improves significantly both the *Precision*@10 and the examined cumulative gain measures compared with the baseline approach - *full-text search* - for both types of queries. Specifically, Minersoft improves the *Precision*@10 about 160% and the cumulative gain measures (NDCG, NCG) about 173% for NDCG and 163% for NCG with respect to the baseline approach.

Regarding the intermediate steps for the construction of SG, the highest improvement is observed at *Software-files-only search*. This is explained by the fact that documentation and irrelevant files have been removed and the searching is done only in the remaining software files. Our findings show also that the addition of metadata attribute of path in

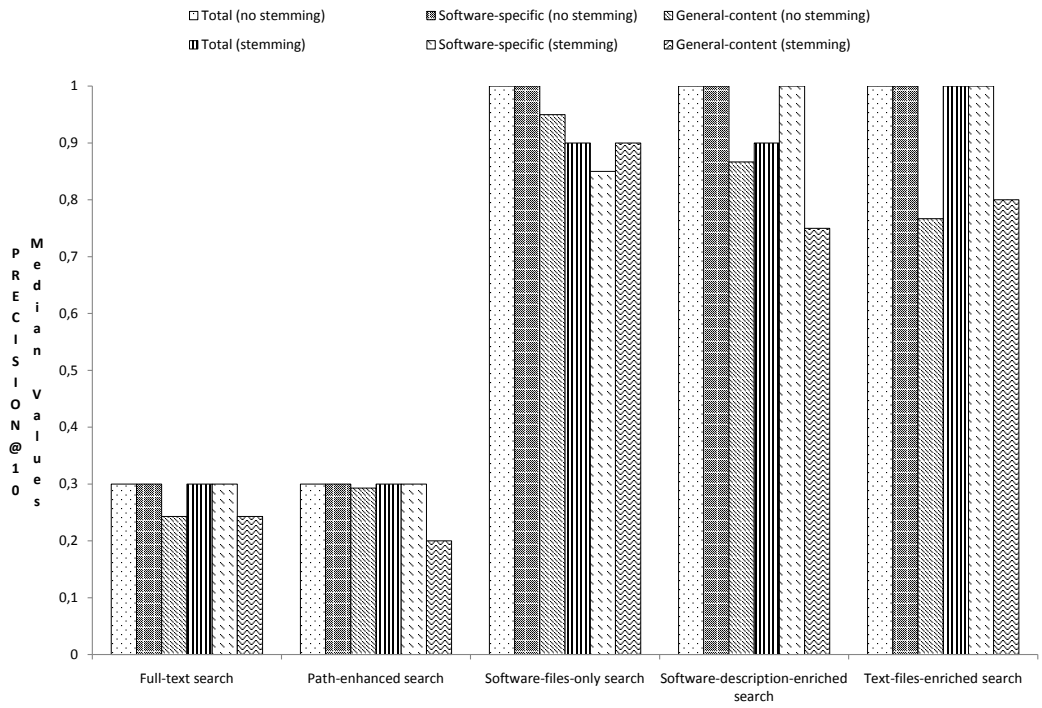

Figure 13: Precision@10 results (median values).

software resources makes Minersoft more effective. In particular, *Path-enhanced search* improves both the *Precision*@10 and the cumulative gain measures (NDCG, NCG) about 28% with respect to the baseline approach. This is an indication that the paths of software files include descriptive keywords for software resources (e.g. /usr/include/magick-/shear.h, /usr/bin/svn).

The enrichment of software-description documents increases the precision as well as user satisfaction. Specifically, *Software-description-enriched search* achieves higher *Precision*@10 (about 3%) and higher cumulative gain measures (on average about 7% for NDCG and

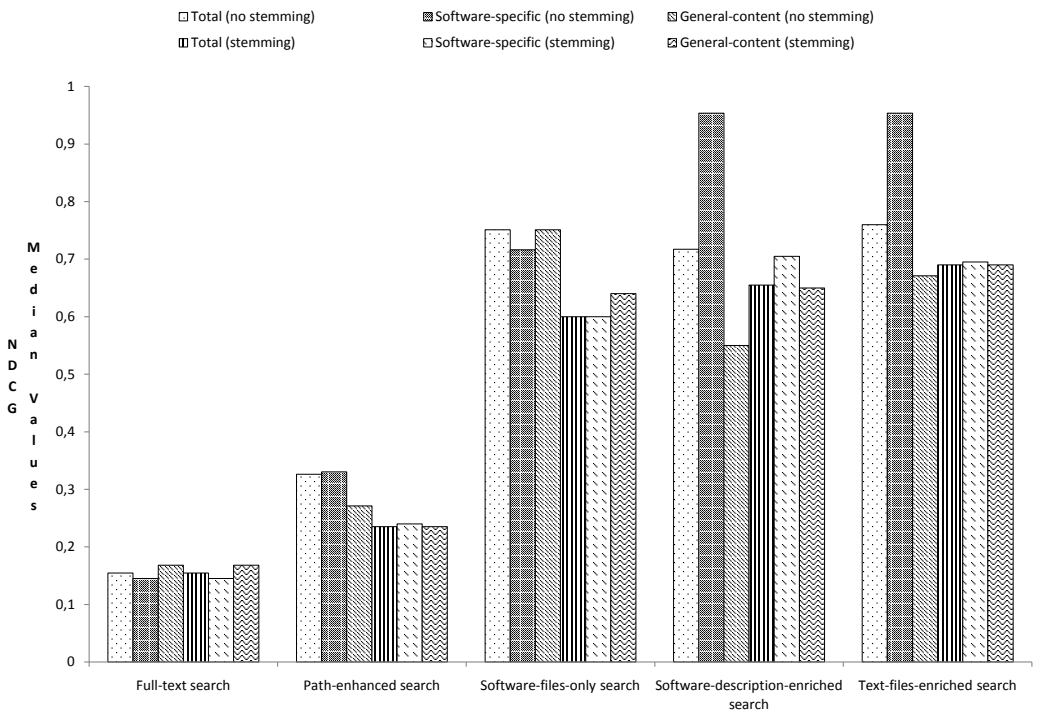

Figure 14: NDCG results (median values).

5% for NCG) than the *Software-files-only search*. The improvements during the *Software-description-enriched search* step are affected by the number of the executables and software libraries that exist in the data set. Recall that this step enriches the executables and software libraries with extra keywords. Thus, the larger the number of these types of files, the better results are obtained. We also observe that the *Software-description-enriched search* step slightly decreases the median values of total queries. In general, *Software-description-enriched search* increases the recall since the terms of inverted index are also extracted from the documentation files. Consequently, this results in lower median values

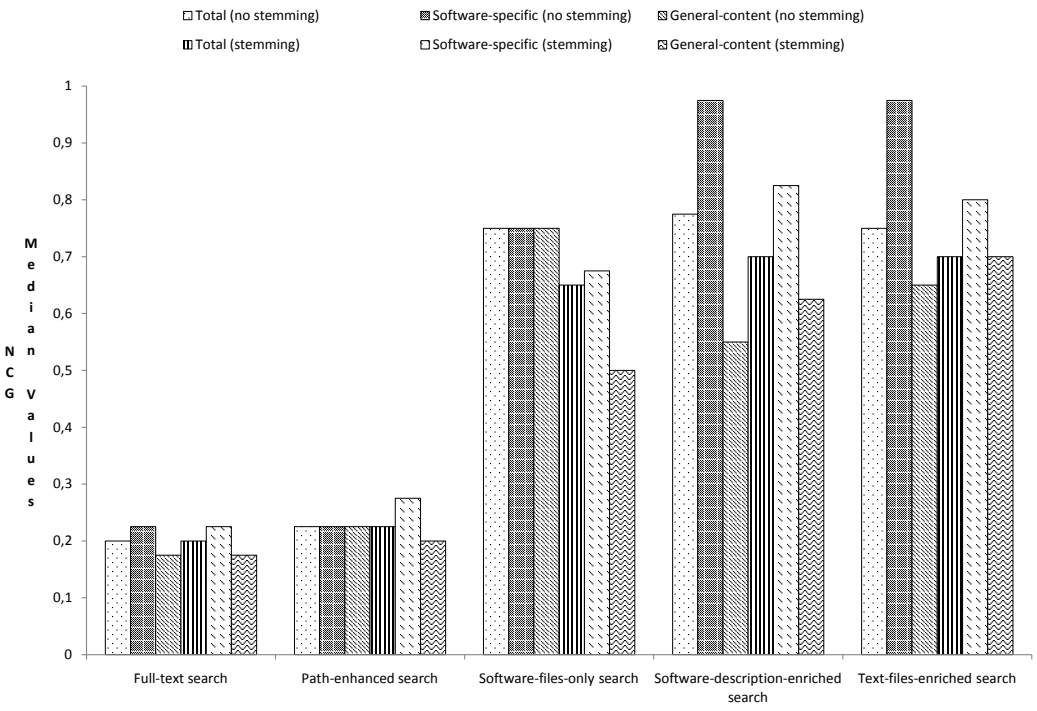

Figure 15: NCG results (median values).

for general-content queries. Another interesting observation is that most of the software-specific queries indicate *Precision*@10 and cumulative gain measures (NDCG, NCG) close to 1 (see median values), whereas the average *Precision*@10, NDCG and NCG values for all the software-specific queries are about 0.84, 0.78, 0.76 respectively.

Regarding the *text-files-enriched search*, we observe that this approach does not improve the general system's performance. This is explained by the fact the software developers use similar filenames in their software packages. However, taking a deeper look at the results, we observe *text-files-enriched search* improves user satisfaction about 5%

Grid Sites	Size with stemming (MB)	Size w/o stemming (MB)
ce01.kallisto.hellasgrid.gr	58.129,359	60.988,246
ce301.intercol.edu	358,0312	395,089
grid-ce.ii.edu.mk	702,796	778,972
paugrid1.pamukkale.edu.tr	632,035	702,296
ce01.grid.info.uvt.ro	1.152,386	1.258,464
grid-lab-ce.ii.edu.mk	57,414	62,863
ce01.mosigrid.utcluj.ro	257,984	288,214
ce101.grid.ucy.ac.cy	13.588,468	14.073,417
ce64.phy.bg.ac.yu	871,449	964,621
testbed001.grid.ici.ro	646,671	715,148

Table 16: Inverted Indexes Size in Grid sites.

for general-content queries since more results are returned to users than the previous examined approach.

Finally, we study the effects of stemming in software resources. As far as the stemming is concerned, Minersoft processes the terms of posting lists using the Porter stemming algorithm[1] . In general stemming decreases the size of inverted indexes since the number of terms is reduced. However, a side effect of stemming is that it results in decreasing precision and user satisfaction. Which is the trade-off between stemming and no stemming in software retrieval? In general, we observe that stemming sacrifices precision in order to decrease the inverted index sizes and improve the recall. Specifically, our findings showed that stemming deteriorates Minersoft's performance about 4%. On the other hand, in terms of storage, we observe that stemming decreases the size of inverted indexes about 10%. Tables 16 and 17 present the size of inverted indexes in Grid sites and Cloud Virtual Servers respectively.

[1]Porter stemming algorithm. http://tartarus.org/martin/PorterStemmer/

Cloud Virtual Servers	Size with stemming (MB)	Size w/o stemming (MB)
Amazon1	133,113	148,062
Amazon2	162,558	177,042
Amazon3	134,777	149,710
Amazon4	180,570	202,433
Amazon5	299,585	338,593
Amazon6	162,160	180,417
Rackspace1	78,136	87,085
Rackspace2	84,335	94,851
Rackspace3	60,574	66,332
Rackspace4	221,902	245,011

Table 17: Inverted Indexes Size in Cloud Virtual Servers.

To sum up, the results show that Minersoft is a powerful tool since it achieves high effectiveness for both types of queries. Focusing on the query types, we observe that Minersoft presents high efficiency for both types of queries, achieving very high performance for software-specific queries. Specifically, our experimentations concluded to the following empirical observations:

- Minersoft improves the *Precision*@10 about 160% and Cumulative gain measures (NDCG, NCG) over 163% with respect to the baseline approach.
- The paths of software files in file-systems include descriptive keywords for software resources.
- Stemming deteriorates about 4% the system's performance. On the other hand, it decreases the size of inverted indexes about 10%.
- Software developers use similar filenames for their software packages.

7.3.4 Software Graph Statistics

Tables 18, 19 presents the statistics of the resulted SGs. Recall that Minersoft harvester constructs a SG in each Grid site/Virtual Cloud Server. We do not present further analysis of the SGs since this is out of the scope of this work. Of course, a thorough study of the structure and evolution of SGs would lead to insightful conclusions in software engineering community. In the literature, a large number of dynamic large-scale networks have been extensively studied [36] in order to identify their latent characteristics.

Here, we briefly present the main characteristics of these graphs. Tables 18, 19 present the edges that have been added due to structure dependency (E_{SD}) and content associations (E_{CA}). For completeness of presentation, the index size of each graph is presented. Based on these statistics, a general observation is that the SGs are not sparse. Specifically, we found that in case of Grids most of them follow the relation $E = V^{\alpha}$, where $1.1 < \alpha < 1.36$, whereas, in case of Clouds most of Cloud Virtual Servers follow the relation $E = V^{\alpha}$, where $1.1 < \alpha < 1.31$; note that $\alpha = 2$ corresponds to an extremely dense graph where each node has, on average, edges to a constant fraction of all nodes. Another interesting observation is that most of the edges are due to content associations. However, most of these edges have lower weights $(0,05 \leq w < 0,2)$ than the edges which are due to structure dependency associations.

Grid Sites	**V**	**E (total edges)**	E_{SD}	E_{CA}
ce01.kallisto.hellasgrid.gr	3.264.810	1.291.884.123	9.540.597	1.282.343.526
ce301.intercol.edu	70.366	150.033	96.922	53.111
grid-ce.ii.edu.mk	156.333	1.659.309	322.495	1.336.814
paugrid1.pamukkale.edu.tr	106.567	1.195.702	223.529	972.173
ce01.grid.info.uvt.ro	95.871	1.465.779	199.537	1.266.242
grid-lab-ce.ii.edu.mk	94.135	179.127	158.733	20.394
ce01.mosigrid.utcluj.ro	50.704	158.451	86.249	72.202
ce101.grid.ucy.ac.cy	1.169.166	97.967.442	2.117.300	95.850.142
ce64.phy.bg.ac.yu	112.996	987.759	201.950	785.809
testbed001.grid.ici.ro	86.581	772.005	225.591	546.414
Total	5.207.529	1.396.419.730	13.172.903	1.383.246.827

Table 18: Software Graphs Statistics in Grid Sites.

Cloud Virtual Servers	**V**	**E (total edges)**	E_{SD}	E_{CA}
Amazon1	29.346	79.636	49.368	30.268
Amazon2	23.228	74.180	41.107	33.073
Amazon3	17.377	62.032	34.919	27.113
Amazon4	20.168	61.688	45.811	15.877
Amazon5	59.409	583.198	102.534	480.664
Amazon6	27.213	96.513	47.484	49.029
Rackspace1	11.210	30.236	24.221	6.015
Rackspace2	9.310	23.959	20.714	3.245
Rackspace3	8.569	21.181	17.968	3.213
Rackspace4	35.335	157.574	68.185	89.389
Total	241.165	1.190.197	452.311	737.886

Table 19: Software Graphs Statistics in Cloud Virtual Servers.

Chapter 8

Conclusions & future work

8.1 Minersoft in Practice

Minersoft can be used for keyword-based searching software resources in large-scale network infrastructures. Its environment allows researchers and software practitioners to locate software resources suitable to their needs and encourage software investigation, software reuse, clone detection and computational resources selection.

- **Software investigation:** Software projects include a number of resources which often go through several modifications. As part of most modification tasks, a developer must investigate the software resources associated with the project prior to modifying it. The investigation of software resources in an efficient way is a challenging problem for software engineering because it is an inherently human activity, whose success depends on several unpredictable factors, such as developer intuition and luck [49]. Minersoft can help software developers quickly identify the software resources that are likely to be associated with the underlying project. Through the

content and structural associations of SG, Minersoft indicates software resources worthy of investigation, independently of the semantics of the source codes.

- **Software reuse:** Software reuse is the systematic use of existing software resources to construct new resources or products. Software reuse has been recognized as one of most realistic and promising ways to improve software productivity, quality and reliability, shorten the time required to release software for client use and reduce maintenance costs [46]. To achieve these goals, software reuse needs effective retrieval techniques to make development with the reusable components more convenient than development from scratch. Although software reuse has been promoted as an effective means to develop software products, in practice it has proved to be hard to achieve and of limited use [46]. Minersoft encourages software reuse by providing keyword-based searching techniques for software packages which are installed on large-scale network infrastructures.

- **Clone detection:** Clone detection techniques aim at finding duplicated code, which may have been adapted slightly from the original. These techniques arise because duplication does exist for several reasons such as time pressure, inexperienced developers or even the use of similar mathematic formulas [26]. Detecting clones would also be used for studying the evolution of a software system. Various research on finding software similarities has been performed [26], where most studies focused on detecting program plagiarism. Through the exploitation of SG, Minersoft can be used for detecting such clones.

- **Computational resources selection:** A challenge for software developers in Grid and Cloud computing infrastructures is to select the appropriate computational resources in order to build new applications. Up to now, existing Grid/Cloud providers do not support any tool that would help developers to select computational resources so as to run their applications. Minersoft covers this gap through the keyword-based searching on installed software in Grid sites/Cloud Virtual Servers.

8.2 Open Issues

We are currently exploring a number of open issues that require further analysis:

- Determining the size of splits: As we referred above, the files of each Grid site are split into a number of splits where the size of the split is chosen to ensure that the crawling and indexing can be distributed evenly and efficiently within the time constraints of the underlying site. Considering that in a Grid infrastructure the execution time and workload cannot be determined in advance using historical data, the size of splits should be adapted to the working environment. To meet this challenge, the *Grid job manager* should have a global view of the system's workload so as to dynamically rearrange the size of splits as well as schedule them to Grid sites.

- Determining the number of threads per crawler/indexer jobs: From our experiments it is obvious that the number of threads per crawler/indexer job affects significantly the performance of Minersoft. To improve the efficiency of crawling and indexing, Minersoft should enhance self-adaptive mechanisms in order to assign a sufficient number of threads to Grid resources (CPUs in Grid sites). Grid monitoring systems

provide information about resource utilization (CPU utilization, memory utilization, disk utilization, etc.) and network connectivity in Grid sites. Thus, if the current status of a Grid site has changed, the *Grid job manager* should modify the number of threads per crawler/indexer jobs in this site.

- Detecting duplicate software files: Files that are exact duplicates of each other can be identified by either heuristic techniques or checksumming techniques. In order to prevent duplicate files, crawler jobs need to periodically communicate to coordinate with each other. However, this communication may result in overhead. Can we minimize this communication overhead while maintaining the effectiveness of the crawler job? Authors in [19] dealt with this problem in the context of the Web. Another issue is the identification of near-duplicate files. If we could successfully identify these files, we could improve the performance of indexing, since a percentage of files will be deleted. Manber [43] has developed algorithms for near-duplicate detection to reduce storage in large-scale file systems.

- Determining politeness: Minersoft jobs should not obstruct the normal operation of Grid sites. Minersoft should adhere to strict rate-limiting policies when accessing poorly provisioned (in terms of workload) Grid sites. To address this issue, Minersoft should implement a flexible policy that would avoid running multiple crawler jobs to overloaded Grid sites.

- Exploiting the Software Graph, we could identify coherent clusters. In the literature, a wide range of algorithms have been proposed towards to this goal [32]. From the IR perspective, it has been observed that the clusters identification increases the diversity in search results [51]. Diversity is the extent to which the results

returned by a search engine pertain to different information needs. If a user query is broad (i.e., a query that covers a variety of topics), a search engine can assume that software resources from different clusters represent different topics. Thus, a search engine which returns results from many clusters is likely to have greater diversity. Identifying of coherent clusters of software resources is also beneficial in terms of locating relevant individual softwares, classifying and labeling them with a set of tags [48]. Last but not least, the SG may contribute in improving the ranking of query results [31]. Ranking is an integral component of any information retrieval system. In the case of software search in large-scale network environments the role of ranking the results becomes critical. To this end, the SG may offer a rich context of information which is expressed through its edges. A ranking function can be built by analyzing these edges. Kleinberg, and Brin and Page have built upon this idea by introducing the area of link analysis ranking on the Web, where hyperlink structures are used to rank Web pages [17].

Bibliography

[1] Amazon Elastic Compute (EC2) Cloud. http://aws.amazon.com/ec2 (accessed June 2009). 7, 19

[2] Apache Lucene. http://lucene.apache.org/java/docs/(last accessed December 2008). 31, 36, 49, 57, 76

[3] Enabling Grids for E-SciencE project. http://www.eu-egee.org/ (last accessed July 2009). 2, 7

[4] gLite Middleware. http://glite.web.cern.ch/glite/ (last accessed July 2009). 15

[5] Google Code search engine. http://www.google.com/codesearch (accessed June 2009). 11

[6] JUNG the Java Universal Network/Graph Framework. http://grid.ucy.ac.cy/GridBench (accessed April 2005). 49, 56

[7] Koders search engine. http://www.koders.com (accessed June 2009). 11

[8] The Limelight Content Distribution Platform. http://uk.limelightnetworks.com/ (accessed July 2009). 21

[9] The Rackspace Cloud. http://www.mosso.com/rackspace.jsp (accessed June 2009). 7

[10] Rakesh Agrawal and et al. The claremont report on database research. *SIGMOD Rec.*, 37(3):9–19, 2008. 6

[11] Azzah Al-Maskari, Mark Sanderson, and Paul Clough. The relationship between ir effectiveness measures and user satisfaction. In *SIGIR '07*, pages 773–774, New York, NY, USA, 2007. ACM. 77

[12] Alexander Ames, Carlos Maltzahn, Nikhil Bobb, Ethan L. Miller, Scott A. Brandt, Alisa Neeman, Adam Hiatt, and Deepa Tuteja. Richer file system metadata using links and attributes. In *MSST '05*, pages 49–60, Washington, DC, USA, 2005. IEEE Computer Society. 10

[13] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002. 9, 11, 43

[14] Michael Armbrust and et al. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. 1

[15] Shenghua Bao, Guirong Xue, Xiaoyuan Wu, Yong Yu, Ben Fei, and Zhong Su. Optimizing web search using social annotations. In *WWW '07*, pages 501–510, New York, NY, USA, 2007. ACM. 76

[16] Len Bass, Paul Clements, Rick Kazman, and Mark Klein. Evaluating the software architecture competence of organizations. In *WICSA '08*, pages 249–252, 2008. 1

[17] Allan Borodin, Gareth O. Roberts, Jeffrey S. Rosenthal, and Panayiotis Tsaparas. Link analysis ranking: algorithms, theory, and experiments. *ACM Trans. Interet Technol.*, 5(1):231–297, 2005. 92

[18] F. Brochu, U. Egede, J. Elmsheuser, and K. Harrison et al. Ganga: a tool for computational-task management and easy access to Grid resources. *Computer Physics Communications (submitted)*, 2009. http://ganga.web.cern.ch/ganga/documents/index.php. 35, 49

[19] Junghoo Cho and Hector Garcia-Molina. Parallel crawlers. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 124–135, New York, NY, USA, 2002. ACM. 72, 91

[20] Charles L.A. Clarke and et al. Novelty and diversity in information retrieval evaluation. In *SIGIR '08*, pages 659–666, New York, NY, USA, 2008. ACM. 77

[21] Sara Cohen, Carmel Domshlak, and Naama Zwerdling. On ranking techniques for desktop search. *ACM Trans. Inf. Syst.*, 26(2):1–24, 2008. 76

[22] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI '04: 6th Symposium on Operating System Design and Implemention*, pages 137–150. Usenix Association, December 2004. 30

[23] M. D. Dikaiakos, R. Sakellariou, and Y. Ioannidis. *Information Services for Large-scale Grids: A Case for a Grid Search Engine*, chapter Engineering the Grid: status and perspectives, pages 571–585. American Scientific Publishers, 2006. 2, 6

[24] Marios D. Dikaiakos, Athena Stassopoulou, and Loizos Papageorgiou. An investigation of web crawler behavior: Characterization and metrics. *Computer Communications*, 28(8):880–897, May 2005. 5

[25] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International J. Supercomputer Applications*, 15(3):200–222, 2001. 17

[26] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 321–330, New York, NY, USA, 2008. ACM. 1, 89

[27] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and Jr. James W. O'Toole. Semantic file systems. In *SOSP '91*, pages 16–25, New York, NY, USA, 1991. ACM. 10

[28] Karl Anders Gyllstrom, Craig Soules, and Alistair Veitch. Confluence: enhancing contextual desktop search. In *SIGIR '07*, pages 717–718, New York, NY, USA, 2007. ACM. 10, 78

[29] Oliver Hummel and Colin Atkinson. Extreme harvesting: Test driven discovery and reuse of software components. In *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, IRI - 2004, Las Vegas Hilton, Las Vegas, NV, USA*, pages 66–72, 2004. 10

[30] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002. 76

[31] Hung-Yu Kao and Seng-Feng Lin. A fast pagerank convergence method based on the cluster prediction. In *WI '07*, pages 593–599, Washington, DC, USA, 2007. IEEE Computer Society. 92

[32] Dimitrios Katsaros, George Pallis, Konstantinos Stamos, Athena Vakali, Antonis Sidiropoulos, and Yannis Manolopoulos. Cdns content outsourcing via generalized communities. *IEEE TKDE*, 2009. 91

[33] A. Katsifodimos, G. Pallis, and D. M. Dikaiakos. Harvesting large-scale grids for software resources. In *CCGRID '09*, Shanghai, China, 2009. IEEE Computer Society. 5, 67

[34] Sofien Khemakhem, Khalil Drira, and Mohamed Jmaiel. Sec+: an enhanced search engine for component-based software development. *SIGSOFT Softw. Eng. Notes*, 32(4):4, 2007. 10

[35] Jonathan Koren, Andrew Leung, Yi Zhang, Carlos Maltzahn, Sasha Ames, and Ethan Miller. Searching and navigating petabyte-scale file systems based on facets. In *PDSW '07*, pages 21–25, 2007. 10

[36] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM TKDD*, 1(1), 2007. 86

[37] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD 2008*, pages 903–914, New York, NY, USA, 2008. ACM. 2

[38] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.*, 18(2):300–336, 2009. 9, 10, 74

[39] The Linux Filesystem Hierarchy Standard. Available online at: http://www.pathname.com/fhs/, last accessed on September 2009. 24

[40] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16(4):13, 2007. 9

[41] Daniel Lucrédio, Antônio Francisco do Prado, and Eduardo Santana de Almeida. A survey on software components search and retrieval. In *Proceedings of the 30th Euromicro Conference*, pages 152–159, 2004. 9

[42] Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Softw. Eng.*, 17(8):800–813, 1991. 9, 11, 47

[43] Udi Manber. Finding similar files in a large file system. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 2–2, Berkeley, CA, USA, 1994. USENIX Association. 72, 91

[44] Andrian Marcus and Jonathan Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE 2003*, pages 125–135, May 2003. 9, 11, 43

[45] Makoto Matsushita. Ranking significance of software components based on use relations. *IEEE Trans. Softw. Eng.*, 31(3):213–225, 2005. 10

[46] Parastoo Mohagheghi and Reidar Conradi. An empirical investigation of software reuse benefits in a large telecom product. *ACM Trans. Softw. Eng. Methodol.*, 17(3):1–31, 2008. 1, 89

[47] G. Pallis, A. Katsifodimos, and D. M. Dikaiakos. Effective keyword search for software resources installed in large-scale grid infrastructures. In *2009 IEEE/WIC/ACM International Conference on Web Intelligence*, Milano, Italy, 2009. 5

[48] Daniel Ramage, Paul Heymann, Christopher D. Manning, and Hector Garcia-Molina. Clustering the tagged web. In *WSDM '09*, pages 54–63, New York, NY, USA, 2009. ACM. 92

[49] Martin P. Robillard. Topology analysis of software dependencies. *ACM Trans. Softw. Eng. Methodol.*, 17(4):1–36, 2008. 1, 10, 88

[50] Fabrizio Silvestri, Diego Puppin, Domenico Laforenza, and Salvatore Orlando. A search architecture for grid software components. In *WI '04*, pages 495–498, Washington, DC, USA, 2004. IEEE Computer Society. 11

[51] Kai Song, Yonghong Tian, Wen Gao, and Tiejun Huang. Diversifying the image retrieval results. In *MULTIMEDIA '06*, pages 707–710, New York, NY, USA, 2006. ACM. 91

[52] Craig A. N. Soules and Gregory R. Ganger. Connections: using context to enhance file search. *SIGOPS Oper. Syst. Rev.*, 39(5):119–132, 2005. 10

[53] Jaime Teevan, Eytan Adar, Rosie Jones, and Michael A. S. Potts. Information re-retrieval: repeat queries in yahoo's logs. In *SIGIR '07*, pages 151–158, New York, NY, USA, 2007. ACM. 74

[54] Taciana Vanderlei and et. al. A cooperative classification mechanism for search and retrieval software components. In *SAC '07*, pages 866–871, New York, NY, USA, 2007. ACM. 10, 11

[55] Peter C.K. Yeung, Luanne Freund, and Charles L.A. Clarke. X-site: a workplace search tool for software engineers. In *SIGIR '07*, New York, NY, USA, 2007. ACM. 10, 78

[56] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.*, 6(4):333–369, 1997. 9