Minersoft: A Keyword-based Search Engine for Software Resources in Large-scale Grid Infrastructures[*]

Marios D. Dikaiakos [1], Asterios Katsifodimos [2], George Pallis [3]

Computer Science Department, University of Cyprus, Nicosia, CY 1678, Cyprus

[1]mdd@cs.ucy.ac.cy, [2]asteriosk@cs.ucy.ac.cy, [3]gpallis@cs.ucy.ac.cy

ABSTRACT

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 computing infrastructures. We address a number of challenges that arise from the unstructured nature of software and the unavailability of software-related metadata on Grid sites. We present Minersoft, a Grid harvester that visits Grid sites, 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 a production-quality Grid infrastructure. Experimental results show that Minersoft is a powerful tool achieving high search efficiency.

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. One of the main goals of these large-scale distributed computing environments is to make their software resources and services easily accessible and attractive for end-users [7]. 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 [9, 25, 29].

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 [20]. To motivate the importance of such a tool, let us consider a researcher who is searching for graph mining software deployed on a Grid infrastructure. Unfortunately, the manual discovery of such software is a daunting, nearly impossible task: taking the case of EGEE [2], one of the largest production Grids currently in operation, the researcher 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 1,700 Amazon Machine Images (AMIs), with each AMI hosting at least 90,000 files, including installed software. Envisioning the existence of a software search engine, the researcher would submit a query to the search engine using some keywords (e.g. "graph tool," or "communities discovery"). In response to this query, the engine would return a list of software matching the query's keywords, along with computing sites where this software is located. Thus, the researcher would be able to identify the sites hosting an application suitable to her needs, and would accordingly prepare and submit jobs to these sites.

However, 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. Furthermore, the registries of distributed computing infrastructures rarely publish little, if any information about installed software [13]. 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 new techniques 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 [3]. 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.

[*]This work is an extended version of the paper that has been presented to IEEE/ACM WI 2009 Conference.

Contributions. 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 infrastructures. Furthermore, Minersoft can be easily extended to support search on Cloud infrastructures like Amazon's EC. Minersoft visits a Grid site, 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 Grid sites 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 implementation and the performance evaluation of the Minersoft crawler are presented in [17]. In this paper, we introduce the core information retrieval component of Minersoft - *the Software Graph*, and its related algorithms. The main contributions of this work can be summarized as follows:

- 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 present the design, the architecture, and implementation of the Minersoft harvester.
- We 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. To this end, we use Minersoft to harvest sites of the EGEE Grid. Results show that Minersoft achieves high search efficiency.

Roadmap. Section 2 presents an overview of related work. In Section 3, we introduce the concepts of software resources, software package and Software Graph. Section 4 describes the proposed algorithm to create a Software Graph annotated with keyword-based metadata. Section 5 describes the architecture of Minersoft. In Section 6 we present an experimental assessment of our work. We conclude in Section 7.

2. RELATED WORK

A number of research efforts 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 [21]).

In [22], 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. Similar approaches have been proposed also in [6, 23]. 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. The use of folksonomy concepts has been investigated in the context of the Maracatu system [28]. 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 [18]; 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 [14], the Linking File System (LiFS) [5], or from file systems that provide extensions to support search through facets [19], contextualization [26], desktop search (e.g., Confluence [15], Wumpus [30]), etc. Although Minersoft could easily take advantage of 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.

In the Grid context, a recent work has proposed a software search service, called GRIDLE [25]; 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.

Although we are not aware of any work that proposes a keyword-based paradigm for locating software resources on large-scale Grid infrastructures, our work overlaps with prior work on software resources retrieval [6, 22, 23, 28]. 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 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 federated infrastructures: i) Software management is a decentralized activity; different sites may follow different policies about software installation, directory naming etc. Also, software entities on a Grid site 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. ii) Harvesting the sites of a Grid infrastructure 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. iii) The users of a Grid infrastructure do not have direct access to local Grid sites. 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 infrastructures.

3. BACKGROUND

In this section we provide some background and define software resource, software package and Software Graph, which are the main focus of this paper.

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 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 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[1] of the software resource represented by v.

- $type(v)$ denotes the type of v; a vertex can be classified into one of a finite number of types (more details on this are given in the following section).

- $site(v)$ denotes the computing site where file v is located.

- $path(v)$ is a set of terms derived from the path-name of software resource v in the file system of $site(v)$.

- $zone_l(v), l = 1, \ldots, z_v$ 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, $0 <$

[1]Normalization techniques for filenames are presented in [24].

$w_i \leq 1$ 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 how the Software Graph can be constructed (section 4), the Minersoft architecture section 5) and we evaluate its contribution (section 6).

4. SOFTWARE GRAPH CONSTRUCTION AND INDEXING

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. To this end, we propose an algorithm comprising a number of steps described below :

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 (e.g. Java, C++), libraries, software-description documents 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* and *html* 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 inserted in the graph; each of these edges is assigned a weight of one, 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.

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 stemmed and 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 [6] and [23], 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 4.1), keyword extraction and keyword flow (section 4.2), and content association mining (section 4.3).

4.1 Context Enrichment

During the structural dependency mining phase, Minersoft seeks to discover associations between documentation and software leaves of the file-system tree. These associations are represented as edges in the SG and contribute to the enrichment of the context of software resources. 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. 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.

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.

4.2 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 *content zone* of a vertex v is more important for the description of v than its *documentation zones*. 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.

4.3 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.

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 [22]. 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.

4.4 Parallelization

For the efficient implementation of the SG construction algorithm in a Grid setting, we should take advantage of various parallelization techniques in order to:

- Distribute parts of the Minersoft computation to Grid sites, in order to take advantage of the Grid computing and storage power, to reduce the communication exchange between the Minersoft system and local Grid sites, and to sustain the scalability of Minersoft with respect to the total number of Grid sites. Minersoft tasks are wrapped as Grid jobs that are submitted to Grid sites via the Grid workload-management system.
- Avoid overloading Grid sites by applying load-balancing techniques when deploying Minersoft jobs to the Grid.
- Improve the performance of Minersoft jobs by employing multi-threading to overlap local computation with I/O.
- Adapt to the policies put in place by different Grid sites regarding 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 site, etc.

More details on the parallelization of the Minersoft algorithm and its deployment on the EGEE Grid are given in the following section.

5. MINERSOFT ARCHITECTURE

Creating a search engine for software that can cope with the scale of emerging Grid infrastructures presents several challenges. Fast crawling technology is required to gather the Grid software resources and keep them up to date. Storage space must be used efficiently to store indices and the files themselves. The indexing system must process hundreds of gigabytes of data efficiently.

In this section, we will provide a description of how the whole system works as depicted in Figure 1. Minersoft architecture is a Map-Reduce-like system; the crawling and indexing is done by several distributed multi-threaded crawler and indexer Grid jobs, which run in parallel on different Grid sites for improved performance and efficiency. The crawler and indexer jobs process a specific number of files, called *splits*. A key component of the Minersoft architecture is the *Grid job manager*, which has the overall supervision for crawler and indexer jobs. The *graph constructor* module is responsible for constructing the SG exporting data from the full-text inverted indexes which are distributed in the Grid sites. For each site, this module performs structural dependency mining, keyword scrapping, keyword flow and content association mining tasks (described in previous section). These tasks result in enriching the *full-text inverted indexes* of Grid sites. The *query processor* module is responsible for providing quality search results efficiently.

5.1 Crawling phase

Crawling the Grid is a challenging task that needs to address various performance, reliability and site-policy issues since it involves interaction with hundreds of Grid sites, which are beyond the control of the system. Minersoft undertakes the crawling of Grid sites in a distributed manner. The *Grid job manager* sends a number of multi-threaded crawler jobs to each Grid site. A challenge for crawler jobs is to harvest all the software resources residing within Grid sites, without exceeding the time constraints imposed by site policies: jobs which run longer than the allowed time are terminated by the sites' batch systems. The maximum wall clock time for a Grid site usually ranges between 2 and 72 hours.

Considering that a Grid site contains a large volume of files, we decompose the file system of each Grid site into a number of *splits*, where the size of each split is chosen so that the crawling can be distributed evenly and efficiently within the time constraints of the underlying site. The splits are assigned by the *Grid job manager* to crawler jobs on a continuous basis: As a site finishes with its assigned splits, it is receives more splits for processing. If a site becomes laggard, the crawler job is canceled and rescheduled to run when the site's workload is reduced. Furthermore, if the batch system queue of a Grid site is full and does not accept new jobs, the *Grid job manager* stops submitting crawler jobs to that site until the batch system becomes ready to accept more. Minersoft crawlers undertake the task of classifying software files into categories, as described earlier (e.g., binaries, libraries, documentations). The files found to be irrelevant are dropped from the FST data structure. The results of a crawler are stored at the Storage Element of each Grid site. In particular, we keep in a *metadata store file* the file-id, name, type, path, size and structural dependencies of the identified software resources. Then, the *Grid job manager* fetches the resulted *metadata store files* from all Grid sites and merges them into a *file index*. The *file index* comprises information about each software resource and is stored in Minersoft's dedicated infrastructure. From this index we can easily construct the first version of the SG, which captures software-related files and their structural relationships.

5.2 Indexing phase

During the indexing phase, the *file index* is used by the *Grid job manager* in order to create multi-threaded indexer jobs, and to dispatch them for execution to Grid sites. The task of the indexers is to read and parse local files of interest and create a full-text inverted index. Since most sites do not allow jobs running more than 48 hours, several indexer jobs should be submitted and executed simultaneously on each site, to improve the file-processing throughput and to reduce the overall indexing time per site. Each indexer job is responsible for a specific number of files in a Grid site.

Similarly to the crawling process, the list of files of each Grid site is decomposed into a number of splits where the

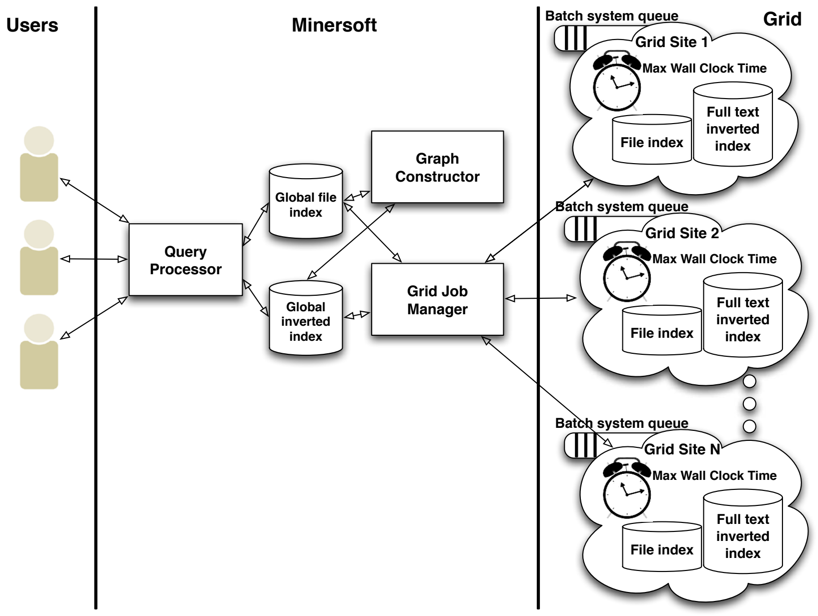

Figure 1: Minersoft architecture.

size of the split is chosen to ensure that indexing can be distributed evenly and efficiently within the time constraints of the system. In this context, the *Grid job manager* assigns the splits to a number of indexer jobs, taking into account the current status of the Grid site. As a site finishes indexing the assigned splits, it receives the next ones. When the indexing has been completed, each Grid site has a full-text inverted index.

Since a large percentage of duplicate software resources exists in Grid sites, Minersoft uses a *duplicate reduction policy* to preprocess the *file index* and identify the exact duplicate files. Its ultimate goal is to further improve the performance of indexing. Specifically, a file may belong to more than one Grid sites. Files with the same name, path and size are considered to be duplicates. According to our policy, a duplicate file is assigned to the Grid site which has the minimum number of assigned files that should be indexed. The key idea behind this policy is to distribute the duplicate software resources in Grid sites so as to prevent their overloading. In this context, for each Grid site, the following steps take place:

1. The *file index* is sorted in ascending order with respect to the count of sites that a file exists.
2. The files which do not have duplicates are directly assigned to the corresponding Grid site.
3. If a file belongs to more than one Grid sites, the file is assigned to the site with the minimum number of assigned files.

Finally, the *Grid job manager* fetches all the resulted local inverted-indexes and merges them into a global full-text inverted index which is stored in the Minersoft repository.

5.3 Harvester Implementation and Deployment

The implementation of the *Grid job manager* relies upon the Ganga system [10], 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. The *Grid job manager* (through Ganga scripts) monitors the status of jobs after their submission and keeps a list of sites and their failure rate. If there are sites with a very high failure rate, the *Grid job manager* eventually puts them in a black list and stops 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 site 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 [1]). In order to execute the indexer jobs, we follow the same code-deployment scenario as with crawlers.

Before the job submission starts, the *Grid job manager* has to distribute the crawling/indexing workload. This is done by creating splits for each site that Minersoft has to crawl. The input file for each split is uploaded on a storage element and registered to an LCG File Catalog (LFC). Every Job has its own ID (given as an argument during submission). A job's ID is the split number that the job will have to process. 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 an LFC.

The logical file names and the directories containing them in the LFC are properly named so that they implicitly state the split number and the site that they came from or going to.

6. EVALUATION

The software design of Minersoft enables the distribution of its crawling and indexing tasks to the computing nodes of EGEE [2]. In the current implementation we used Java, Python, and an open-source high performance, full-text index and search library (Apache Lucene). In [17], we evaluated the performance of the overall system on 9 Grid sites of EGEE infrastructure and we concluded to the following empirical observations:

- A large percentage of duplicate files exists in Grid sites. Specifically, 33% of files belongs to more than one Grid sites.
- The crawling and indexing is significantly affected by the hardware (local disk, shared file system), file types and the current workload of Grid sites.
- It is important to establish advanced software discovery services in the Grid since, in most cases, more than 50% of files that exist in the workernodes file systems of Grid sites are software files.

In this work, we evaluate the effectiveness of the Minersoft search engine for locating software on the EGEE. 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). On the other hand, 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 EGEE. 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 6 Grid sites of EGEE infrastructure. Table 1 presents the software resources that have been identified by Minersoft on those sites.
- *Queries*: We use a collection of 27 queries, which were provided to us by EGEE users, and which comprise either single- or multiple-keywords. Each query has an average of 2.3 keywords; this is comparable to values reported in the literature for Web search engines [27]. To further investigate the sensitivity of Minersoft, we have classified the queries into two categories: general-content and software-specific (see Table 2).
- *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 satisfied." These classifications are referred to as the *gold standard* and have been done manually by EGEE administrators and/or experienced users.

General-content queries	**Software-specific queries**
linear algebra package; fast fourier transformations; symbolic algebra computation library; mathematics statistics analysis; earthquake analysis; scientific data processing; statistical analysis software; atlas software	ImageMagick; lapack library; GSL library; crab; k3b cd burning; xerces xml; gcc fortran; octave numerical computations; matlab; hpc netlib; scalapack; mpich; autodock docking; boost c++ library; subversion client; java virtual machine; ffmpeg video processing; FFTW library

Table 2: Queries.

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:

- *Precision*@20: reports the fraction of software resources ranked in the top 20 results that are labeled as relevant. The relevance of the retrieved results is determined by the *gold standard*. By default, we consider that the results are ranked with respect to the ranking function of Lucene, which is based on TF-IDF of documents and has extensively been used in the literature [8, 12]. The maximum Precision@20 value that can be achieved is 1.
- NDCG (Normalized Discounted Cumulative Gain) [16]: is a retrieval measure devised specifically for evaluating user satisfaction. For a given query q, the K ranked results are examined in decreasing order of rank, and the NDCG computed as: $NDCG_q = M_q \cdot \sum_{j=1}^{K=20} \frac{2^{r(j)}-1}{\log_2(1+j)}$, 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 Mq is a normalization constant calculated so that a perfect ordering would obtain NDCG of 1.
- NCG: This 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: $NCG_q = M_q \cdot \sum_{j=1}^{K=20} r(j)$. 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 [4, 11].

Examined Approaches. In order to evaluate the SG efficiency, we conducted experiments during the construction of inverted index. Specifically, we examine the following:

Grid Site	Binaries	Sources	Libraries	Docs	Irrelevant
AEGIS01-PHY-SCL	6.064	31.734	7.669	66.810	38.559
CY-03-INTERCOLLEGE	26.971	8.925	3.644	23.064	27.296
CY-01-KIMON	28.691	166.294	22.571	295.074	45.666
RO-08-UVT	8.134	56.793	4.199	68.335	146.940
HG-05-FORTH	28.351	495.995	65.507	759.571	114.138
BG04-ACAD	46.330	960.824	93.663	1.305.390	298.039
Total	**144.541**	**1.720.565**	**197.253**	**2.518.244**	**670.638**

Table 1: Files Categories.

Grid Sites	V	E (total edges)	E_{SD}	E_{CA}	Index Size(GB)
AEGIS01-PHY-SCL	120.369	1.007.508	207.080	800.428	0.73
CY-03-INTERCOLLEGE	72.424	209.243	154.998	54.245	0.34
CY-01-KIMON	565.799	20.670.759	1.050.076	19.620.683	2
RO-08-UVT	157.591	862.005	228.299	633.706	0.66
HG-05-FORTH	1.508.986	164.657.942	3.632.165	161.025.777	15
BG04-ACAD	2.632.193	617.084.993	6.359.610	610.725.383	16
Total	**5.057.362**	**804.492.450**	**11.632.228**	**792.860.222**	**34.73**

Table 3: Software Graphs Statistics.

- *File-search*: Inverted index terms are only extracted from the full-text content of discovered files in EGEE infrastructure without any preprocessing. This approach searches files matching given query terms and it is relevant to the desktop search systems (e.g., Confluence [15], Wumpus [30]). $File - search$ is used as a baseline for our experiments.

- *Context-enhanced search*: The files have been classified into file categories. The terms of inverted index are extracted from the content zone and path of SG vertices. The irrelevant files are discarded. We also exclude the software-description documents from the posting lists.

- *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-file-enriched search*: The terms of 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.

Results and Analysis. Figures 2, 3 and 4 present the results of the examined approaches with respect to the query types. For completeness of presentation, we present the average and median values of the examined metrics. The general observation is that *context-enhanced search* improves significantly both the $Precision$@20 and the examined cumulative gain measures compared with *file-search* for both types of queries. Specifically, *context-enhanced search* improves the $Precision$@20 about 97% and NDCG about 87% with respect to the baseline approach. Another interesting observation is that most of software-specific queries indicate average $Precision$@20 close to 1 (see median values), whereas the average $Precision$@20 for all the queries is about 0,8. Regarding the *software-description-enriched search*, we make the following observations: Although the enrichment of software-description documents decreases the precision (about 5%) with respect to *context-enhanced search*, it does increase user satisfaction achieving higher cumulative gain measures (on average about 7%). The decrease of precision is due to the side-effects of stemming. On the other hand, the *text-file-enriched search* deteriorates the general system's performance. This is explained by the fact the software developers use similar filenames in their software packages. On the other hand, *text-file-enriched search* improves user satisfaction for general-content queries since more results are returned to users than the previous examined approaches. To sum up, the results show that Minersoft is a powerful tool since it achieves high effectiveness for both types of queries.

Table 3 presents the statistics of the resulted SGs. Recall that Minersoft harvester constructs a SG in each Grid site. In this context, Table 3 presents 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. One observation is that the SGs are not sparse. Specifically, we found that they follow the relation $E = V^{\alpha}$, where $1.1 < \alpha < 1.37$; 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 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.

7. CONCLUSION AND FUTURE WORK

In this paper, we present the design and implementation of the core information retrieval component of Minersoft - the *Software Graph*. Experimental results showed that SG represents in an efficient way the software resources, improving the searching of software packages in large-scale network environments. Except of Grids, Minersoft can also be used as keyword-based paradigm for any large-scale distributed

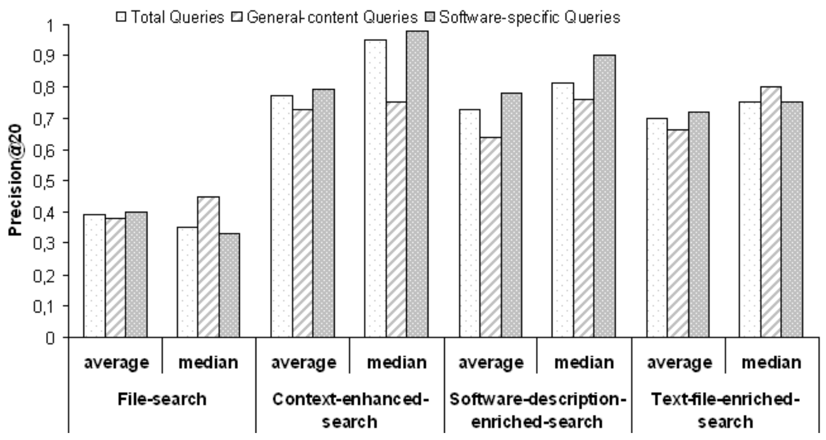

Figure 2: Precision Results.

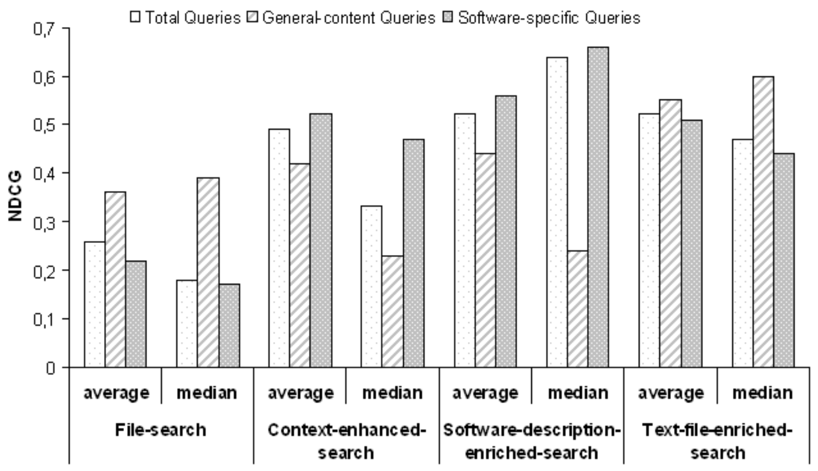

Figure 3: NDCG Results.

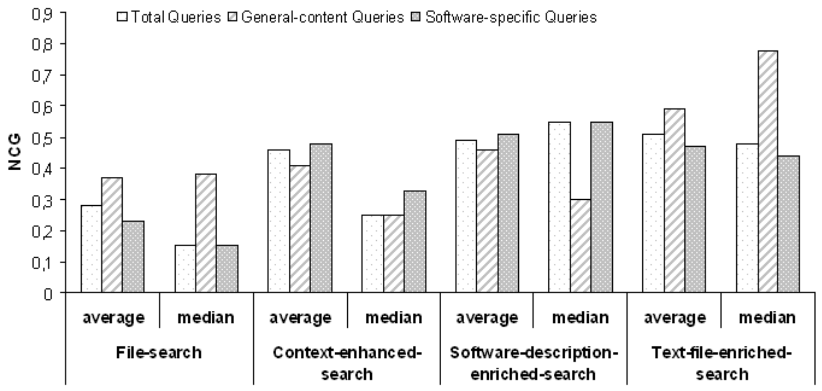

Figure 4: NCG Results.

computing platform, such as Clouds, as well as, for standalone computers. We are currently extending Minersoft for harvesting and indexing software resources in Cloud computing infrastructures. In future work we also intend to exploit the linkage structure of SG so as to identify coherent clusters of "correlated" software resources and improve the ranking of results.

Acknowledgement: It was supported in part by the European Commission under the Seventh Framework Programme through the SEARCHiN project (Marie Curie Action, contract number FP6-042467) and the Enabling Grids for E-sciencE project (contract number INFSO-RI-222667). The authors would like to thank EGEE users that provided characteristic queries for evaluating Minersoft.

8. REFERENCES

[1] Apache Lucene. http://lucene.apache.org/java/docs/(last accessed December 2008).

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

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

[4] A. Al-Maskari, M. Sanderson, and P. Clough. The relationship between ir effectiveness measures and user satisfaction. In *SIGIR '07*, pages 773–774, New York, NY, USA, 2007. ACM.

[5] A. Ames, C. Maltzahn, N. Bobb, E. L. Miller, S. A. Brandt, A. Neeman, A. Hiatt, and D. Tuteja. Richer file system metadata using links and attributes. In *MSST '05*, pages 49–60, Washington, DC, USA, 2005. IEEE Computer Society.

[6] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002.

[7] M. 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.

[8] S. Bao, G. Xue, X. Wu, Y. Yu, B. Fei, and Z. Su. Optimizing web search using social annotations. In *WWW '07*, pages 501–510, New York, NY, USA, 2007. ACM.

[9] L. Bass, P. Clements, R. Kazman, and M. Klein. Evaluating the software architecture competence of organizations. In *WICSA '08*, pages 249–252, 2008.

[10] F. Brochu, U. Egede, J. Elmsheuser, and K. H. 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.

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

[12] S. Cohen, C. Domshlak, and N. Zwerdling. On ranking techniques for desktop search. *ACM Trans. Inf. Syst.*, 26(2):1–24, 2008.

[13] 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.

[14] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. James W. O'Toole. Semantic file systems. In *SOSP '91*, pages 16–25, New York, NY, USA, 1991. ACM.

[15] K. A. Gyllstrom, C. Soules, and A. Veitch. Confluence: enhancing contextual desktop search. In *SIGIR '07*, pages 717–718, New York, NY, USA, 2007. ACM.

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

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

[18] S. Khemakhem, K. Drira, and M. Jmaiel. Sec+: an enhanced search engine for component-based software development. *SIGSOFT Softw. Eng. Notes*, 32(4):4, 2007.

[19] J. Koren, A. Leung, Y. Zhang, C. Maltzahn, S. Ames, and E. Miller. Searching and navigating petabyte-scale file systems based on facets. In *PDSW '07*, pages 21–25, 2007.

[20] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. 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.

[21] D. Lucrédio, A. F. do Prado, and E. S. de Almeida. A survey on software components search and retrieval. In *Proceedings of the 30th Euromicro Conference*, pages 152–159, 2004.

[22] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Softw. Eng.*, 17(8):800–813, 1991.

[23] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE 2003*, pages 125–135, May 2003.

[24] D. Robinson, I. Sung, and N. Williams. File systems, unicode, and normalization. In *Unicode '06*, 2006.

[25] F. Silvestri, D. Puppin, D. Laforenza, and S. Orlando. A search architecture for grid software components. In *WI '04*, pages 495–498, Washington, DC, USA, 2004. IEEE Computer Society.

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

[27] J. Teevan, E. Adar, R. Jones, and M. A. S. Potts. Information re-retrieval: repeat queries in yahoo's logs. In *SIGIR '07*, pages 151–158, New York, NY, USA, 2007. ACM.

[28] T. 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.

[29] P. Vitharana, F. M. Zahedi, and H. Jain. Design, retrieval, and assembly in component-based software development. *Commun. ACM*, 46(11):97–102, 2003.

[30] P. C. Yeung, L. Freund, and C. L. Clarke. X-site: a workplace search tool for software engineers. In *SIGIR '07*, New York, NY, USA, 2007. ACM.