DELIVERABLE

Project Acronym: E-ARK

Grant Agreement Number: 620998

Project Title: European Archival Records and Knowledge Preservation

DELIVERABLE DETAILS

DELIVERABLE REFERENCE NO.	D6.1
DELIVERABLE TITLE	Faceted Query Interface and API
REVISION	1.0

AUTHOR(S)	
Name(s)	Organisation(s)
Rainer Schmidt, Roman Karl	AIT Austrian Institute of Technology GmbH
REVIEWER(S)	
Name(s)	Organisation(s)
Richard Healey Kuldar Aas	University of Portsmouth The National Archives of Estonia

Project co-funded by the European Commission within the ICT Policy Support Programme

Dissemination Level		
P	Public	P
C	Confidential, only for members of the Consortium and the Commission Services	

REVISION HISTORY AND STATEMENT OF ORIGINALITY

Submitted Revisions History

Revision No.	Date	Authors(s)	Organisation	Description
01	4/5/15	R Schmidt	AIT	Initial document and structure
02	6/5/15	R Karl	AIT	Initial technical documentation
0.3	19/5/15	R Karl, R Schmidt	AIT	Extended technical documentation
0.4	26/5/15	R Schmidt, R Karl	AIT	First complete draft
0.5	1/6/15	R Schmidt	AIT	Draft ready for review
0.6	16/7/15	R Schmidt, R Karl	AIT	Response to reviewer comments
0.7	27/7/15	R Karl	AIT	Response to further reviewer comments
0.8	28/7/15	J Delve	UPHEC	Final proof read
1.0	30/7/15	R Karl	AIT	Submitted to EC

Statement of originality:
This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

EXECUTIVE SUMMARY

The E-ARK Work package 6 (WP6) - Archival Storage, Services, and Integration, is developing a scalable open-source reference implementation for ingesting, searching, and accessing E-ARK information packages. A major task in this context is the development of a faceted query interface for searching archived content which can be utilized by end-users as well as external software components. The reference implementation aims at providing an archiving and search prototype that is flexible in regard to the type and volume of the ingested payloads. The reference implementation is designed to scale from a single host out to a cluster deployment by employing technologies like Apache Hadoop, Solr, and the Lily repository, supporting different types of input data ranging from text-based files and structured records to office documents and binary content.

Deliverable D6.1 provides a technical documentation of the infrastructure supporting the E-ARK Faceted Query Interface and Application Programming Interface (API). It provides a description of the underlying software components utilized for the development of the search functionality of the E-ARK reference implementation and discusses the required interactions to work as an integrated solution. Furthermore, technical documentation of the developed software and system configuration is provided. The document describes also methods to customize the faceted query interface and provides examples for its utilization.

Table of Contents

1. Introduction

E-ARK has conducted a GAP analysis that identifies user requirements for access services which are described in deliverable D5.1 “GAP report between requirements for access and current access solutions”[1]. The study investigates the current landscape of archival solutions regarding the available access components and identifies gaps and requirements from the perspective of national archives and 3rd party users, as well as content providers. D5.1 has identified a major gap in the identification process, where users browse and search collections to identify material of potential interest. D5.1 states that a lack of comprehensive metadata that is available and indexed in finding aids, compromises the performance and efficiency of finding aids, which directly impacts the user experience and the user’s access to the archival holdings in their entirety.

To fill this gap, work on the E-ARK Faceted Query Interface and Application Programming Interface (API) has the goal to establish a scalable search infrastructure for archived content. It is important to note that Work package 6 (WP6) is not working with the whole content ecosystem of an archive, but is concentrating only on indexing and searching of the born-digital E-ARK Information Packages (IPs). The goal is not to replace existing systems but to augment these components (like available catalogues) with a “content repository” that can be searched based on a full text index. As described later in this document, the content repository introduced by WP6 concentrates on search and access based on the content (ie. data/files) contained within an Archival IP (AIP) rather than selected metadata elements. The reference implementation developed in the context of WP6/D6.1 employs scalable (cluster) technology, as scalability issues must be taken into account when operating a detailed content-based search facility.

The E-ARK WP6 - Archival Storage, Services, and Integration is developing a scalable open-source reference implementation for ingesting, searching, and accessing E-ARK information packages. A major task in this context is the development of a faceted query interface for searching archived content which can be utilized directly by end-users or integrated with other software components like archival catalogues. This document provides a description of the techniques and underlying software components utilized for the development of the search functionality of the E-ARK reference implementation.

Work on *Query and Indexing* concerns the configuration and generation of a repository index which holds detailed information on the archives’ digital holdings. For developing a reference implementation, it is important to provide a solution that is flexible and configurable with respect to a range of requirements. The exact configuration of the faceted search interface will be driven by requirements of the access components (like Dissemination IP (DIP) creation)[2] as well as individual institutional requirements and content specific aspects. As a consequence, the reference implementation must provide a configurable query interface which should be accessible via a service API. This API can be used through a web interface and/or an access component (WP5) for searching the repository based on a full text index.

[1] http://www.eark-project.com/resources/project-deliverables/3-d51-e-ark-gap-report

[2] The specific access component requirements are being currently defined in E-ARK WP5 and are already partially described in deliverable D5.2 “E-ARK DIP draft specification”

The reference implementation integrates this query API with a repository implementation, which in turn provides access to an application layer via its access API. The application layer implemented in E-ARK WP5 develops end user components for search, access, and display of archived records. Figure 1 provides a conceptual overview of the architecture and workflow supported by the reference implementation.

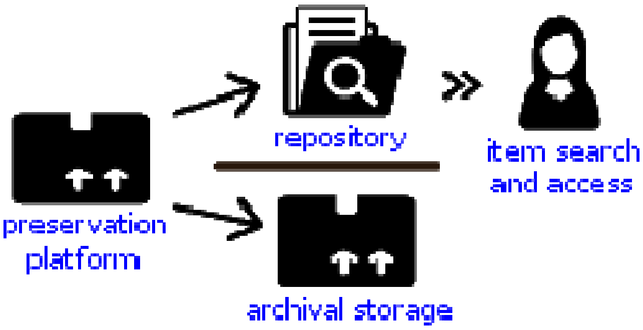


*Figure 1**: Information packages are received and processed by an ingest system like the ESSArch Preservation Platform[3]. As part of the ingest workflow, the information packages are written to an archival storage medium. In addition[4], these packages are also ingested into a content repository which provides basic data management capabilities and search. The created repository records are indexed and can be queried by an end-user application through a service interface. At the repository level, random access is provided on an item/file based level.*

* Icons made by Freepik

The goal of the E-ARK Faceted Query Interface and API is the establishment of a reference service that enables application components to search through the entire archived data and to link the applications with the data management layer[5] (provided through the content repository). The reference implementation will concentrate on data management functionality that supports search, access, and data mining (like providing a CRUD API and support for versioning). The implementation of a fully-fledged archival data management system, however, is out of focus for the reference implementation.

The search functionality is provided by an indexing infrastructure which generates a full-text index for data being ingested into the data management component (ie. the content repository). The goal is to enable end users to efficiently search archival records based on different aspects (or facets) extracted from the archived data and metadata. The search index includes enclosed archival descriptions (metadata) but most crucially the archived data itself (e.g. based on extracted text portions) and generated technical metadata (like file format information).

[3] http://www.essarch.org/

[4] The full-text search and access component developed in WP6 does not replace an existing archival system (like catalogues) but can be utilized to augment these systems.

[5] Here, data management refers to functionality provided by the content repository introduced by the WP6 infrastructure, which is intended to augment the existing archival ecosystem.

The employed indexing techniques are not intended to provide a finding aid based on archival metadata, as for example provided by archival cataloguing systems. The intention of the E-ARK Faceted Query Interface and API is to provide a complementary service that takes advantage of information retrieval techniques like full text indexing, faceted search, and ranking to improve the search through archived data. The indexing workflow is however configurable and able to extract specific information from the archival metadata. This flexibility can for example be utilized to develop specific search facets and/or to handle information related to data confidentiality.

The Faceted Query Interface and API are being developed as part of the E-ARK reference implementation which builds upon a scalable technology stack. The intention is to provide an archiving and search solution that works for different payloads. The reference implementation can therefore be scaled from a single host out to a cluster deployment that is capable of maintaining large volumes of data, e.g. in the magnitude of hundreds of terabytes of archived data organized in hundreds of millions of repository records. The indexing infrastructure is however intended to be deployed next to established archiving systems in order to extend the functionality of the available finding aids. The intention is not to replace the existing systems but rather to extend these infrastructures.

2. System Components

2.1. Overview

As described in the following sections, the Faceted Query Interface and API provided by the E-ARK reference implementation must be backed by software components and generated data products in order to provide the desired functionality. Here, we give an overview of the major components required for implementing the search facility, and describe their interactions.

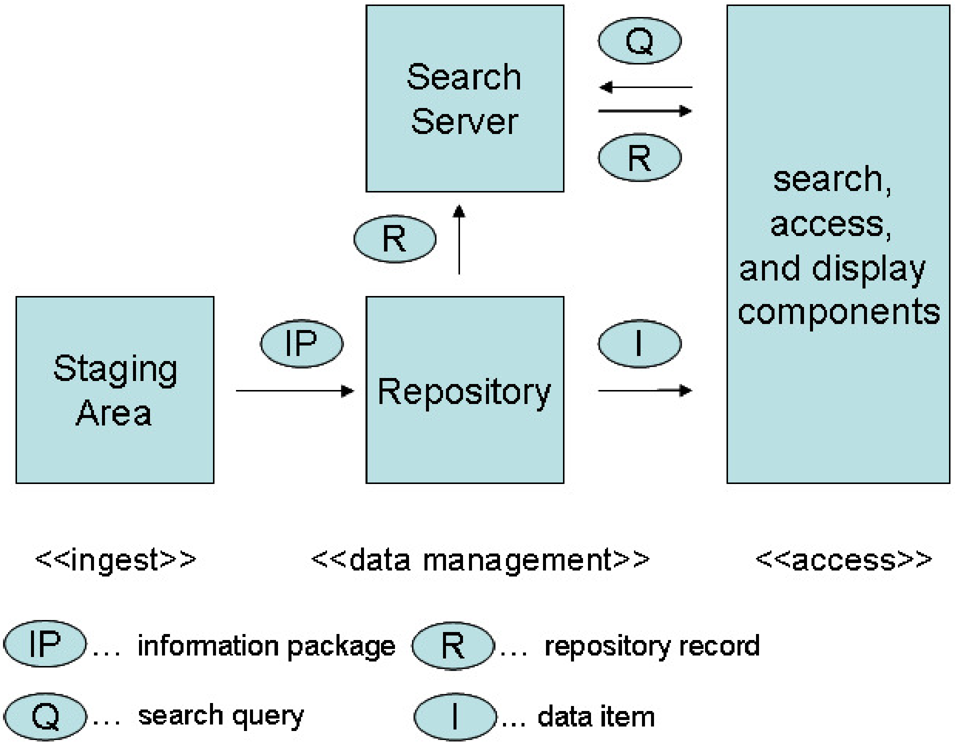


***Figure 2**: System Components and their interactions used by the E-ARK reference implementation for implementing the Faceted Query Interface and API.*

Staging Area: The staging area is a file-system based storage location provided in combination with the data management component of the reference implementation. The staging area is accessible to other components based on an API allowing these components to deposit information packages for ingestion into the content repository (as shown in Figure 1). The staging area is in the first place used to access the information packages during the repository ingest workflow, but can also be employed to serve other purposes like archival storage and (package-based) access.

Repository: The information packages residing on the staging area are ingested into the repository where they are stored in the form of structured repository records, as described in section 3. The repository interacts with the search server, which reads and indexes the repository records, as well as with client components which access data items on a file or record level.

Search Server: The generation and/or update of the index provided by the search server can be triggered by the repository component in case records are added, deleted, or modified. The index provides the necessary data structure to evaluate search queries and to return results which point to records stored in the repository. The index and search functionality is provided by the search server through the Faceted Query Interface and API.

Search, Access, and Display Components: These components interact with the search server and the repository as clients. Specific archival access components (e.g. required for DIP creation) are being implemented in the context of the E-ARK Work Package 5 (Access). The search facility described in this deliverable comes with a generic HTML based search interface for experimenting with different search configurations. The protocol for interacting with the Faceted Query Interface and API is however independent of the employed client component, and ultimately allows for the integration with an external user interface. Client components typically provide a graphical representation of the query language and facets provided by the search server. When a query is submitted to the search server, it is evaluated against the index. The search server subsequently returns a ranked list of record references (and optionally content fragments) to the client. Besides interfaces required for searching, the repository also provides an access service providing clients with random access to data on a file-level, based on references, which may subsequently be invoked by the client application.

2.2. Technical Perspective

Apache Hadoop[6] is a framework for distributed processing. It can be seen as an umbrella term for its associated components. One of the most fundamental is the Hadoop distributed file system (HDFS), which the reference implementation uses for the staging area. While in principle any file system could be employed as staging area, HDFS is chosen for performance, scalability and reliability reasons. Another component is the non-relational database HBase, which is used as back end of the repository.

In the reference implementation Apache Solr[7] is used as the search server. The query interface is provided by a defined REST API through Apache Solr, which is customized based on the individual structure of the repository records. For supporting multiple and heterogeneous collections, it is possible to generate different indexes for different datasets maintained by the repository.

The reference implementation makes use of NGDATA's Lily project[8] as content repository. The purpose of Lily is to build a link between HBase and Solr. It does that by placing itself on top of the database and sending messages to the index to keep it synchronised.

3. Conceptual Workflow

In the following, we describe an ingest workflow for the generation of a faceted full-text index enabling one to search information packages within an archive. In the reference implementation, data is made searchable after it has been ingested into the repository component. Figure 3 shows the conceptual workflow for ingesting data items residing on the staging area (for example based on the Hadoop File system) into the content repository.

[6] https://hadoop.apache.org/
[7] http://lucene.apache.org/solr/
[8] http://www.lilyproject.org/lily/index.html

Practically, this means that after the repository has been populated and/or updated, a full text index is generated and/or updated respectively.

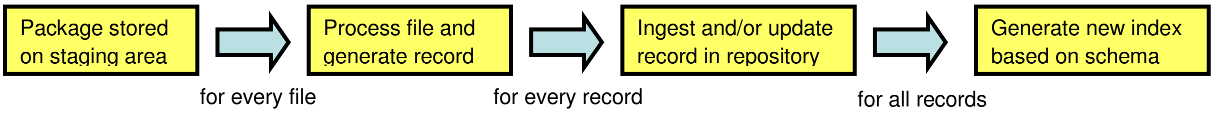

Package stored on staging area
for every file
Process file and generate record
for every record
Ingest and/or update record in repository
for all records
Generate new index based on schema

***Figure 3**: Conceptual workflow for ingestion and indexing of information packages to the content repository provided by the reference implementation.*

The reference implementation implements the ingest of IPs to the content repository on a file-based level which is in contrast to ingesting on a package level. The ingest workflow is implemented in a way that every item (or file) contained in an information package is considered a record. In the repository, packages are represented as a set of records sharing a common identifier.

Once the ingest process is started, the workflow iterates over all files contained in the individual information packages. Each file extracted from the information package is processed separately. The exact implementation of the processing step is highly dependent on the data set and institutional requirements. Examples that have been implemented as part of the reference implementation include the extraction of text portions, structure, and context information from web, office, or XML documents; file size calculation; mime-type identification; and checksums.

The data extracted from an individual file is subsequently stored into a data structure, called a record[9], which can be ingested into the repository. The individual structure of a record can be customized depending on data and institutional needs[10]. A record for a newspaper article, for example, could contain fields like author, title, body, publisher, and publishing date. Fields of a record are “typed” which means they can be restricted to certain data types like for example numbers, string, or date. A record identifier that encodes the identifier of the package as well as the location of the original file within the package, is generated automatically.

Once a record is created, it is ingested into the content repository. As records organize information in a structured way, they can be interpreted by the repository and consequently stored in a (structured) database.

The E-ARK reference implementation, however, aims at providing a faceted query interface based on full text indexing in addition to rather limited search mechanisms provided through database indexing. Such a full text search functionality can be provided through a search servicer (like Apache Solr), which relies on a previously generated full-text index (using Apache Lucene). The reference implementation makes use of a configuration file (called a schema) that provides a detailed specification of the indexing process. This controls, for example, which parts of a record should be

[9] In this document, the term “record” refers to a technical term for data sets stored within a repository system as for example provided by Lily. It should not be confused with the meaning of the term “record” in the context of e-records management systems.

[10] The reader is referred to chapter 6 for more details on customising Solr indexes and Lily records.

indexed, available fields, and the information that should be stored with the index (e.g. only document references and/or also content portions).

After new content has been ingested and/or updated, the repository index should be generated or updated at periodic intervals. The reference implementation provides specific commands for triggering the creation of the index from the records available within the repository. Depending on the volume of content, indexing as well as ingestion can become very resource and time consuming processes. Both processes have therefore been implemented as parallel applications that can take advantage of a computer cluster to scale out for large data rates. Within the E-ARK reference implementation, indexing and ingestion have been deployed on a cluster at AIT, providing a total of 48 CPU-cores.

The generated index is made available by the search server as a Faceted Query Interface and API. The API enables a client to formulate and execute queries against the index, compose complex queries based on facets, and rank them based on different characteristics. It is, however, important to note that although a defined query API is exposed by the reference implementation, the API is highly configurable and customizable with respect to the parameters it accepts, and the nature of the results it returns.

The workflow shown in Figure 3 has been implemented based on the software components described in section 2 (and shown in Figure 2). It has been configured for different test data and deployed in a single-node environment as well as in a cluster environment available at AIT.

4. System Documentation

The Faceted Query Interface and API rely on the search facility provided by content repository of the reference implementation. The content repository has been set up based on the Lily project which is built on top of Cloudera's CDH4[11] distribution of Apache Hadoop and the Apache Solr search server. Although there are detailed instructions for installing Lily[12], Cloudera and Solr available on the official websites, it is still a complex task to deploy them as an integrated system. To set up the environment, it is required to resolve the individual dependencies between the different components and align specific software versions and configuration files. This section gives an overview of the Lily deployment at AIT and points out possible pitfalls. There is also a collection of scripts[13] for cluster installation and management purposes available on the E-ARK Github site. The current deployment makes use of Lily version 2.4[14], Apache Solr v. 4.0.0, and CDH4 v. 4.2.2.

4.1. System Requirements

One prerequisite of Lily is a Hadoop installation running on a cluster or in a single node environment. Both setups are available on the E-ARK infrastructure. Hadoop is typically installed in a Linux environment. It is however also possible to use other Unix-like operating systems like Mac OS,

[11] http://www.cloudera.com/content/cloudera/en/products-and-services/cdh.html
[12] http://www.lilyproject.org/lily/downloads/releases.html
[13] https://github.com/eark-project/dm-etl/tree/master/cluster_scripts
[14] http://lilyproject.org/release/2.4/lily-2.4.tar.gz

although not officially supported by CDH4[15]. Installing Hadoop in a Windows environment is possible using emulators like Cygwin or specific Hadoop distributions[16], but these have not been tested in the context of this deliverable. The following description is based on different deployments done on Linux environments. It was tested for the Linux distributions Ubuntu 12.04/14.04 and Debian 7. For development purposes it is not required to set up a cluster deployment as it is done for production systems. There is also the option to set up a pseudo-distributed environment on a single computer. The two installation forms vary only slightly. Maintaining a private cluster environment can, however, introduce considerable administrative overhead, depending on the size and complexity of the setup.

4.2. Java

The first step is installing Java 7 and verifying that only one version of Java is used. Mixing between Java 6 and Java 7, or between OpenJDK and Oracle Java often causes exceptions. Such a mix between versions can easily occur, because applications use different methods to determine the Java path in the file system. Three methods are common for Hadoop and Hadoop-related applications: The environment variable JAVA_HOME should be set on system start and additionally added to the Apache Bigtop Utils configuration file (/etc/default/bigtop-utils). Bigtop is used by various big-data frameworks and can be used by Hadoop to determine the Java version. Furthermore Java should be accessible via the environment variable PATH.

4.3. Network

Another crucial step is the network connectivity between all hosts of a cluster. Every host has to be able to contact every other host by its host name. This means that an address resolution system has to be in place. This can be DNS, but it can also be done by static entries for all hosts. For nodes with multiple interfaces, the routing becomes an important point, too. Using the wrong route for communication between master and slave nodes can result in unknown source IP addresses and communication failures.

4.4. Hadoop

Lily 2.4 was developed and tested using the Cloudera CDH4 Hadoop distribution. This makes CDH4 also the first choice for using Lily in a production system. Lily 2.4 is not compatible with YARN, Hadoop's successor of MapReduce. Several Hadoop nodes are needed in order to run Lily.

	Master	Slave
HDFS	NameNode	DataNode
MapReduce	JobTracker	TaskTracker
HBase	HBase Master	Region Server

***Table 1**: Daemons running on Hadoop cluster installation*

[15] http://www.cloudera.com/content/cloudera/en/documentation/cdh4/v4-2-2/CDH4-Requirements-and-Supported-Versions/cdhrsv_topic_1.html

[16] http://hortonworks.com/partner/microsoft/

Not every node has to run on its own physical machine. In pseudo-distributed mode, all nodes are running on the same computer. For experimental clusters, all master nodes may run on the same machine. Usually one would use at least two slaves to get a performance advantage from the distributed environment. Each slave machine runs all services, which means that it runs a DataNode, a TaskTracker and a Region Server. For production clusters, it is recommended to deploy the NameNode on its own physical machine and furthermore use a SecondaryNameNode as a backup service. Although Lily is deployed on multiple nodes, it does follow the concept of master and slave nodes. There is only one type of Lily node which is intended to run co-located with Region Servers on the cluster.

4.5. Lily

Lily provides a repository that is build on top of the HBase, a NoSQL database that has been built on top of Hadoop. Lily defines some data types where most of them are based on existing Java data types. Lily records are defined using these data types as compared to using plain HBase tables, which makes them better suited for indexing due to a richer data model. The Lily Indexer is the component which sends the data to the Solr server and keeps the index synchronized with the Lily repository. Solr neither reads data from HDFS nor writes data to HDFS. The index is stored on the local file system and optionally distributed over multiple cluster nodes if index sharding or replication is used. Solr can be run as a standalone Web-based search server which uses the Apache Lucene search library for full-text indexing and search. The reference implementation utilizes the Lily Java API as part of a Hadoop MapReduce job in order to ingest large volumes of files in parallel.

4.6. Solr

There are several options to run Solr. The first option is to run Solr only on one machine. In this case the index is not split and only one shard is used. The second option is to use multiple shards and configure Lily to distribute the input over all shards. As Solr 4 introduced SolrCloud this became the third option, and it is also the preferred option for a production system. SolrCloud does not only take care of the sharding, it also provides a mechanism for replication. Using Lily in combination with SolrCloud requires some additional configuration work being done, as Lily was developed against Solr 4.0, where SolrCloud was not yet entirely mature. For an example, it is required to create an empty directory in ZooKeeper manually where SolrCloud can store its information[17].

4.7. ZooKeeper

HBase, but also Lily and SolrCloud, depend on a running ZooKeeper cluster. ZooKeeper is a framework that supports distributed applications in maintaining configuration information, naming, providing distributed synchronization, and providing group services. ZooKeeper stores small amounts of information, typically configuration data, which can be accessed by all nodes. For experimental clusters that do not need to provide high fault tolerance, it is sufficient to run one ZooKeeper node, which is also called Quorum Peer. A higher fault tolerance can be achieved by

[17] http://archive.apache.org/dist/lucene/solr/ref-guide/apache-solr-ref-guide-4.4.pdf

running three, five or more Quorum Peers. If more than half of the nodes keep running without failures, ZooKeeper stays reliable.

4.8. Configuration Files

Even when using ZooKeeper, the configuration files for HBase, HDFS, and MapReduce have to be copied separately to all nodes of a cluster. An important part of the configuration of MapReduce is to set the maximal number of mappers[18] and reducers in the configuration file mapred-site.xml.

An oversubscription of mappers can improve performance, in case sufficient memory is available on the cluster nodes and given the MapReduce jobs are I/O-bound. In addition certain MapReduce jobs – like the E-ARK ingest workflow – are implemented as map only jobs (i.e. no reducers are employed). In these cases, one might instantiate mapper tasks on all available nodes on the cluster. In the present configuration, we utilize 14 mappers per machine which is a slight oversubscription of the available 12 threads on 6 cores per node. Furthermore speculative execution (mapred.map.tasks.speculative.execution) should be turned off as it is not suitable for tasks which write their output to HBase.

The Hadoop/Lily installation also requires the creation of a certain folder structure on HDFS. Consequently, it is required to start-up HDFS on the cluster and to create all the directories needed for MapReduce, HBase, Lily and users which are allowed to run MapReduce jobs.

4.9. Physical Cluster Setup

The Lily/Hadoop deployment on the development cluster at AIT is shown in Figure 4. The cluster comprises eight physical machines which are structured into a master and seven physical slave nodes. Each node on the cluster provides 6 CPU cores (12 threads using Intel HT). Each cluster node is equipped with two network interfaces allowing us to attach a node to two network infrastructures. The cluster is connected to the internal network allowing us to directly access each node from desktop/working environments. The second private network is used for managing the cluster. For example, new cluster nodes can be automatically booted and configured using PXE together with a private FAI server[19]. Attaching a cluster to two different networks can provide a number of benefits for administrating a cluster. This is however not a requirement for deploying a Lily/Hadoop cluster.

[18] Using the property: mapred.tasktracker.map.tasks.maximum

[19] http://fai-project.org/

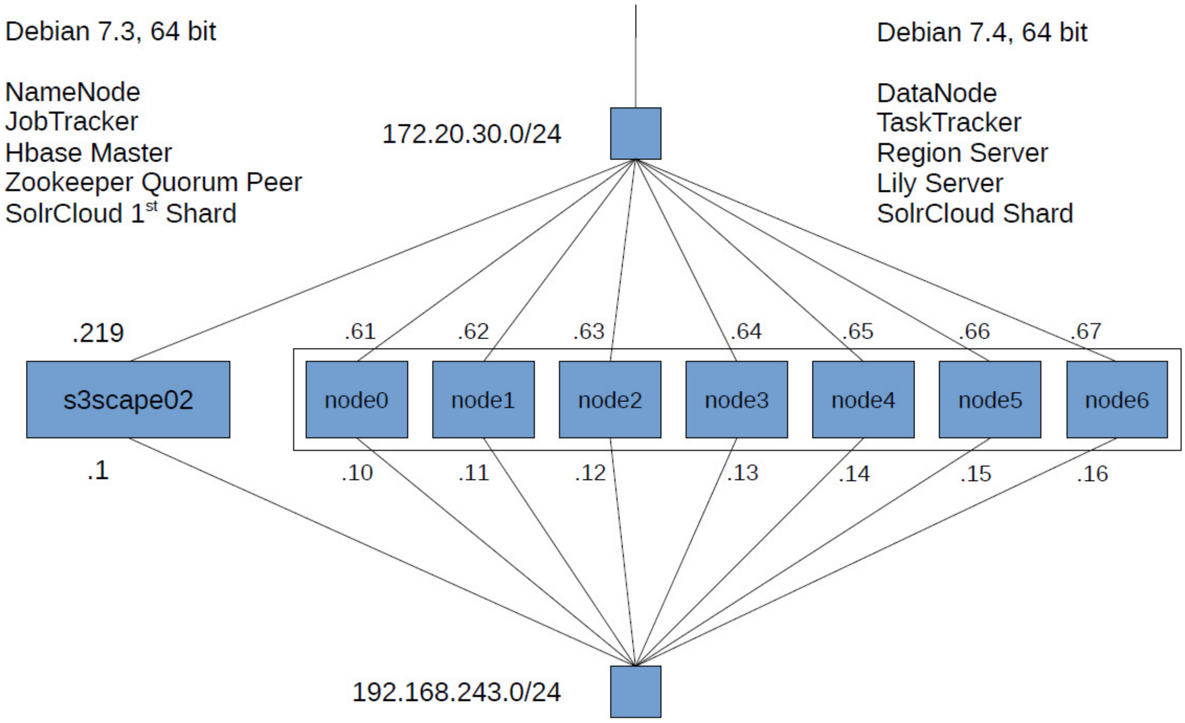


***Figure 4**: Hardware cluster at AIT used to host a Lily repository on top of Hadoop, HDFS and HBase.*

5. Software Documentation

In the following we describe the technical workflow that has been implemented for ingesting and indexing information packages based on a practical example. The implementation of the workflow is customizable and greatly affects the specific functionality that is provided through the query interface (described in section 7).

The file ingest[20] workflow is written as a Hadoop MapReduce job, which takes information packages (containing files and directories) as input. The concrete specification of E-ARK information packages and the processing of the contained metadata is however not part of this deliverable. The current situation is that the implemented ingest workflow is configurable and provides baseline support for E-ARK information packages. The E-ARK AIP specification is however not yet fully supported as this is still work in progress. Specific E-ARK AIP support will be added and described in detail in later deliverables.

The content repository is presently available as a standalone system, and workflows like content ingestion need to be started manually. The development of an integrated archival prototype system that integrates AIP creation, search, and access tools will be presented in deliverable D6.2, “Integrated Platform Reference Implementation.

Using the reference implementation, the information packages must be stored on the Hadoop Distributed File System (HDFS) prior to ingestion into the content repository. During execution, the

[20] https://github.com/eark-project/dm-file-ingest

job reads the individual files contained within the information packages, and uses the Lily Java API to store them in the repository. The files can subsequently be searched and ranked based on contained words or basic metadata information, like content type or size, using the Solr search server. Additionally, an access interface is provided to retrieve individual files from Lily, either via HTTP or based on its Java API.

Let us assume we have an information package which is a tar file ip_001.tar.gz stored on the local file system of one cluster node, and we are working with a Linux user who is allowed to execute MapReduce jobs.

As a first step, the information package is copied to the input directory files in the home directory of the current user on HDFS.

```
tar -xvzf ip_001.tar.gz
hadoop fs -put ip_001 files/
```

The source code20 for the ingestion workflow is available on the E-ARK Github repository. In order to start the ingest workflow on the cluster as a MapReduce job, the following command has to be executed:

```
hadoop jar dm-file-ingest-mapreduce-job.jar --zookeeper <connection-string>
```

The <connection string> points to a node on the cluster that is running ZooKeeper provided via its hostname (s3scape02 on the AIT infrastructure). ZooKeeper enables the ingest application (which uses the Lily client API) to find the Lily nodes in the cluster[21].

The Hadoop job scheduler will create multiple map tasks on the cluster for processing in parallel the payload data (i.e. the files contained in one/multiple information packages). In the following, we explain the ingest workflow based on a single file. Let's assume our information package contains the file dir/file.pdf.

The absolute path for this file on the HDFS is:

```
/user/<user>/files/ip_001/dir/file.pdf.
```

The ingest workflow which is executed by a map task for each file is configurable. Below, we provide an overview of processing steps carried out by the present reference ingest implementation:

1. A new Lily record is created using the relative path (ip_001/dir/file.pdf) as its identifier.
2. The size of the payload file is calculated and added as a field to the record.
3. Apache Tika is used to identify the content type of the file, which is application/pdf in this example, and added as a field to the record.

[21] It is neither required for the node submitting the job nor for a node executing a map task on the cluster to be a Lily node.

4. The entire content of the file is copied and added as a binary field (or BLOB) to the record.

5. The record is stored within the Lily repository.

6. The update of the full-text index is triggered.

The extraction of text portions and document structure can be part of the ingest workflow, and, (depending on the payload content), partially be delegated to the search server. The technical details and internal behaviour triggered when creating Lily records and the Solr full-text index are defined by configuration files which need to be in place prior to file ingest, as described in section 6.

6. Schema Configuration

Lily can be thought of as a link between the NoSQL database HBase and the Lucene index managed by Solr. In HBase a record is just stored as a key-value pair in a table, both key and value being strings or data with no specified type. There is the possibility to define columns and groups of columns, called column families, for a table. Analogous to records in a database, the Lucene index utilizes documents which are structured based on fields. In contrast to HBase, every field has a specific data type, and users can define their own types. Lily introduces its own data representation which is similar to the one used by Solr. The connection between HBase records and Solr documents is handled based on defined keys. The following section describes how schemas in Lily and Solr are created and matched with each other.

6.1. Solr Configuration

For every collection[22] maintaining its own index, Solr provides a configuration directory which is read at start up. The schema for the index is defined in the file `schema.xml` (Appendix A).

For example, in this configuration file the content field for the file ingest is defined as

```
<field name="content"
       type="text_general"
       indexed="true"
       stored="true"
       required="false"/>
```

where `text_general` is a custom data type which is processed using rules which apply to written language, for example spaces and slashes are interpreted as delimiters and not as part of words. Such custom data types are also defined in the file schema.xml.

When the language of the data is known, it is often useful to work with linguistic techniques like word stemming. As an example this means that searching for the singular word “tree” in English texts also finds the plural word “trees” because both words refer to the same token “tree”.

[22] Here, the term “collection” refers to a logical group of information packages for which a separate index should be maintained (e.g. publications, e-records, statistical data). In particular, it makes sense to create different collections for repository records that require a different data model (ie. schema definition). It is however not required to maintain multiple indexes, in particular if there are only small variations in the data and metadata formats contained in the IPs.

The index itself has two different ways of storing data, called `indexed` and `stored`: Using the attribute `indexed`, for each token (representing the occurrence of a term), a reference to all the documents containing this token is stored in the index. Although this is sufficient for fast searching, only very limited information on the original data can be provided through the index. It is therefore possible to store also parts of the original data with the index using the attribute `stored`, allowing for example to show the original text containing the search term with the search result.

Another configuration file is `solrconfig.xml`[23]. In this file the behaviour of the indexer can be configured. For example it can be specified whether changes made to the index are automatically available or must be manually confirmed.

6.2. Lily Schema

The Lily schema (Appendix B) for a repository record is typically defined in a JSON file. Analogous to the Solr schema the field for the content can be defined as:

```
{
  name: "eark$content",
  valueType: "BLOB",
  scope: "non_versioned"
}
```

Here `eark` is a short name of the namespace `org.eu.eark`. The type BLOB indicates to Lily to find an appropriate storage strategy based on the size. Lily also has its own versioning concept which is not enabled for this field.

Lily provides a set of command-line tools allowing users to carry out common actions. The following command is used to import a schema file.

```
bin/lily-import -s schema.json
```

Furthermore it is required to make a connection between the Lily and Solr schema. Responsible for this is the Lily indexer, which sends new record data to the Solr search server. The Lily indexer is configured via an XML file containing the mapping information between the Lily and Solr schemas:

```
<field name="content"
       value="eark:content"
       extractContent="true"/>
```

For most fields the configuration is straight forward. For the content field there is also the attribute `extractContent` which means that not the original data is sent to Solr, but the text content extracted by Apache Tika.

The following command is used to add the indexer configuration (Appendix C):

```
bin/lily-add-index --name eark1 \
  --indexer-config indexerconf.xml \
  --solr-mode cloud \
```

[23] https://github.com/eark-project/dm-file-ingest/blob/master/src/main/config/solr/solrconfig.xml

```
  --collection eark1
```

In the above example, the parameters are provided for SolrCloud. In order to utilize Solr shards without using the cloud mode, the parameters need to be adopted accordingly.

7. Description of the Query Interface

Solr is a standard Java Web application that utilizes the open source Java Servlet engine Jetty when run as a standalone application. Solr provides an interface for querying the index over HTTP and does not provide specific support for generating user interface components. This section provides a brief overview and examples for using the Solr API to query an index generated using the previously described E-ARK reference implementation.

A deployment of the reference implementation together with different example indexes is available on the publicly accessible E-ARK demonstration server, here called EARKDEV.

Request all documents:

The request

http:// EARKDEV:8983/solr/govdocs/select?q=%3A*&wt=json*

returns the number of all documents in the index of the collection "govdocs" together with the first ten documents of the result list including all their field values.

Please note that specific characters in the above query string need to be URL encoded in the HTTP request. The decoded query of the request above reads:

q=:*&wt=json*

In this context, the term query should not be confused with the query part of an URL. The query part of a Solr request is the value of the parameter "q". In this example *:* meaning "select all documents". The format of the response has been set to JSON, the default would be XML.

Search for text within a specific field

q=content:begin

The query specifies a single search term (or token) "begin" and a field "content". The field "content" is defined by the Solr index and has a corresponding field in the structure of the Lily record. Query results are restricted to records containing the search term in the specified field.

Search for text using a wildcard

*q=content:begi**

It is also possible to search for sub-strings contained in search terms using the asterisk as a wildcard.

Search within packages

A sequence of words in a search string is specified using quotation marks. Using this mechanism, it is for example possible to limit search results to packages and/or parts of packages, (based on the folder hierarchy), if applied to the identifier field of a document. The reference implementation encodes the package name and folder structure into the identifiers field for each file. Using the previously described example, the string "/`ip_001/dir1/`" can be used to search a particular subdirectory "dir1" of the information package `ip_001`.

q=path:"package/directory"

Solr recognizes that the above query represents a search term that represents a particular sequence of two words (or tokens). This is enabled through the configuration of the field "path" in the Solr index. Here, the slash character is configured as delimiter for this particular field.

Querying multiple fields

q=content:word AND (contentType:"text/html" contentType:"application/pdf")

It is also possible to compose more complex queries by combining simple queries with logical operators. The default operator in Solr is OR. The above query returns only HTML and PDF files containing the search term "word". Solr can however be configured to use a different logical operator by default.

Sorting and Ranking

q=contentType:"application/pdf"&sort=size asc

In addition to filters which are defined by the query, there is also the option to sort the query results. The above query returns results sorted by the field "size" in ascending order.

In case a query does not specify a sort criterion, Solr employs a metric (scoring or ranking algorithm) to order search results by relevance. The goal is to determine how relevant a given document is to a user's query, and order search results based on relevance. The scoring and ranking of query results is a key feature of Apache Lucene, and can be customized if required. It is also important to note that the design of the index fields, (which are corresponding to Lily records in the E-ARK reference implementation), can significantly influence the calculated score for a particular query. It is therefore important to carefully design the repository and index schemas in order to create an index that conforms to the user’s search requirements.

Faceted Search

Solr’s support for faceted search provides support for clustering search results into different categories. This supports the development of advanced user interfaces, allowing users to get statistics on the search results and to drill down into useful subcategories. Faceted search also relies heavily on the schema design, allowing one to structure information based on facet fields (as previously explained).

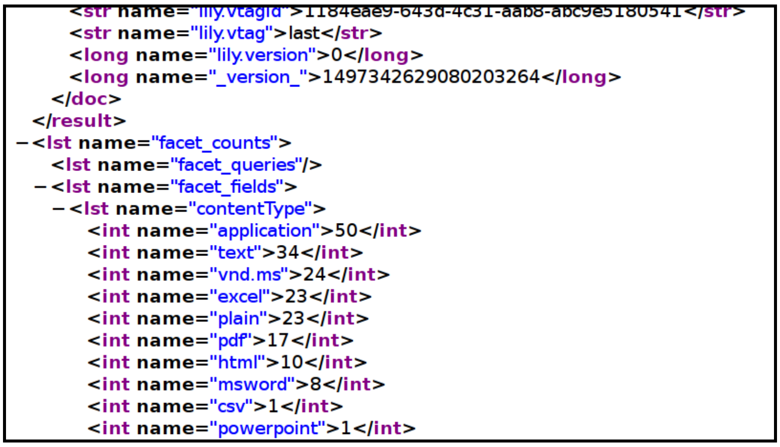
Faceting commands are added to a Solr search query, which is sent to the server as a HTTP request. Solr returns the faceting information like statistics on the returned documents in the HTTP response.

In the following, we show a few example queries that have been executed on the E-ARK demonstration server.

The HTTP request below provides an example for field faceting. The query searches for records containing the term "roger" and asks faceting counts based on the field "contentType".

http://EARKDEV/solr/govdocs/select?q=roger&facet=true&facet.field=contentType

The figure below shows the response returned by Solr for the field faceting query. The faceting counts for the returned document are added at the end of the HTTP response containing search results.

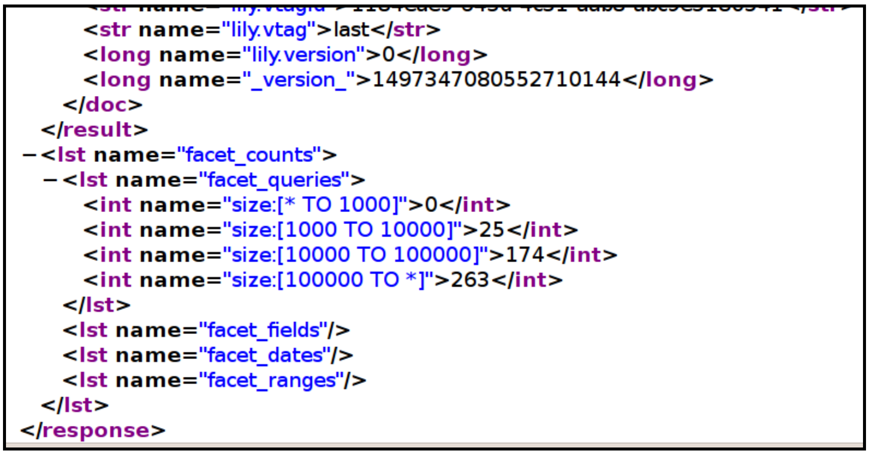


```
      <str name="lily.vtagid">1184eae9-643d-4c31-aab8-abc9e5180541</str>
      <str name="lily.vtag">last</str>
      <long name="lily.version">0</long>
      <long name="_version_">1497342629080203264</long>
    </doc>
  </result>
- <lst name="facet_counts">
    <lst name="facet_queries"/>
  - <lst name="facet_fields">
    - <lst name="contentType">
        <int name="application">50</int>
        <int name="text">34</int>
        <int name="vnd.ms">24</int>
        <int name="excel">23</int>
        <int name="plain">23</int>
        <int name="pdf">17</int>
        <int name="html">10</int>
        <int name="msword">8</int>
        <int name="csv">1</int>
        <int name="powerpoint">1</int>
```

***Figure 5**: Response of a field faceting search*

The HTTP request below provides an example for query faceting. The query searches for records containing the term "roger" and asks faceting counts based on different file size ranges. The file size has been configured as a numeric value (of type long) in the field "size" of the index schema.

http://EARKDEV/solr/govdocs/select?q=roger&facet=true&facet.query=size:[*%20TO%201000]&facet.query=size:[1000%20TO%2010000]&facet.query=size:[10000%20TO%20100000]&facet.query=size:[100000%20TO%20*]



```
      <str name="lily.vtagid">1184eae9-643d-4c31-aab8-abc9e5180541</str>
      <str name="lily.vtag">last</str>
      <long name="lily.version">0</long>
      <long name="_version_">1497347080552710144</long>
    </doc>
  </result>
- <lst name="facet_counts">
  - <lst name="facet_queries">
      <int name="size:[* TO 1000]">0</int>
      <int name="size:[1000 TO 10000]">25</int>
      <int name="size:[10000 TO 100000]">174</int>
      <int name="size:[100000 TO *]">263</int>
    </lst>
    <lst name="facet_fields"/>
    <lst name="facet_dates"/>
    <lst name="facet_ranges"/>
  </lst>
</response>
```

***Figure 6**: Response of a query faceting search*

The figure below shows the response returned by Solr for the query faceting request. The faceting counts (based on file size ranges) for the returned document are added at the end of the HTTP response containing search results.

APPENDIX

A. Schema for the Solr Index (index.xml)

```
<?xml version="1.0" encoding="UTF-8" ?>


<schema name="example" version="1.5">


  <types>
    <fieldType name="string" class="solr.StrField" sortMissingLast="true"
omitNorms="true"/>


    <!-- A general text field that has reasonable, generic
         cross-language defaults: it tokenizes with StandardTokenizer,
         removes stop words from case-insensitive "stopwords.txt"
         (empty by default), and down cases.  At query time only, it
         also applies synonyms. -->
    <fieldType name="text_general" class="solr.TextField"
positionIncrementGap="100">
      <analyzer type="index">
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.StopFilterFactory" ignoreCase="true"
words="stopwords.txt" />
        <!-- in this example, we will only use synonyms at query time
        <filter class="solr.SynonymFilterFactory"
synonyms="index_synonyms.txt" ignoreCase="true" expand="false"/>
        -->
        <filter class="solr.LowerCaseFilterFactory"/>
      </analyzer>
      <analyzer type="query">
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.StopFilterFactory" ignoreCase="true"
words="stopwords.txt" />
        <filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt"
ignoreCase="true" expand="true"/>
        <filter class="solr.LowerCaseFilterFactory"/>
      </analyzer>
    </fieldType>


    <fieldType name="long" class="solr.TrieLongField" precisionStep="0"
positionIncrementGap="0"/>
    <fieldType name="boolean" class="solr.BoolField"/>
 </types>
 <fields>
   <!-- The _version_ field is required when using the Solr update log or
SolrCloud (cfr. SOLR-3432) -->
   <field name="_version_" type="long" indexed="true" stored="true"/>


   <field name="lily.key" type="string" indexed="true" stored="true"
required="true"/>
   <field name="lily.id" type="string" indexed="true" stored="true"
required="true"/>
   <field name="lily.table" type="string" indexed="true" stored="true"
required="true"/>
   <field name="lily.vtagId" type="string" indexed="true" stored="true"/>
   <field name="lily.vtag" type="string" indexed="true" stored="true"/>
   <field name="lily.version" type="long" indexed="true" stored="true"/>


   <field name="path" type="text_general" indexed="true" stored="true"
required="true"/>
   <field name="contentType" type="text_general" indexed="true"
stored="true" required="false"/>
   <field name="size" type="long" indexed="true" stored="true"
required="true"/>
   <field name="confidential" type="boolean" indexed="true" stored="true"
required="true"/>
   <field name="content" type="text_general" indexed="true" stored="true"
required="false"/>
 </fields>


 <uniqueKey>lily.key</uniqueKey>


</schema>
```

B. Schema for Lily record (index.json)

```
{
  namespaces: {
    "org.eu.eark": "eark"
  },
  fieldTypes: [
    {
      name: "eark$path",
      valueType: "STRING",
      scope: "non_versioned"
    },
    {
      name: "eark$contentType",
      valueType: "STRING",
      scope: "non_versioned"
    },
    {
      name: "eark$size",
      valueType: "LONG",
      scope: "non_versioned"
    },
    {
      name: "eark$confidential",
      valueType: "BOOLEAN",
      scope: "non_versioned"
    },
    {
      name: "eark$content",
      valueType: "BLOB",
      scope: "non_versioned"
    }
  ],
  recordTypes: [
    {
      name: "eark$File",
      fields: [
        {name: "eark$path", mandatory: true},
        {name: "eark$contentType", mandatory: false},
        {name: "eark$size", mandatory: true},
        {name: "eark$confidential", mandatory: true},
        {name: "eark$content", mandatory: true}
      ]
    }
  ]
}
```

C. Lily Indexer Configuration (indexerconf.xml)

```
<?xml version="1.0"?>

<indexer xmlns:eark="org.eu.eark">
  <recordFilter>
    <includes>
      <include recordType="eark:File" vtags="last"/>
    </includes>
  </recordFilter>

  <fields>
    <field name="path" value="eark:path"/>
    <field name="contentType" value="eark:contentType"/>
    <field name="size" value="eark:size"/>
    <field name="confidential" value="eark:confidential"/>
    <field name="content" value="eark:content" extractContent="true"/>
  </fields>

</indexer>
```