# FIRST THOUGHTS ON A DATA LAKE ARCHITECTURE FOR AN OPEN SEARCH INFRASTRUCTURE

L. Martin†, A. Henrich*, University of Bamberg, 96049 Bamberg, Germany

*Abstract*

The goal of the Open Search Foundation is to establish a decentralized open search infrastructure for navigating, searching, and analyzing the internet. Due to the variety of requirements, the need for a robust and extensible architecture arises. This paper gives first thoughts on an architecture for such an infrastructure. As a starting point, a basic architecture for a Data-Lake-based document-centric search engine is proposed. Afterwards, components are added and extended to meet further requirements of the envisaged search infrastructure.

## INTRODUCTION

The goal of the Open Search Foundation (OSF) [1] is to establish a decentralized open search infrastructure for navigating, searching, and analyzing the internet that combines the computational resources of different European data centers and institutes to sustain its operation. Open source principles and public moderation are central points of the pan-European vision. Users will be able to use the search infrastructure by accessing, adding, analyzing, annotating, and enriching data with respect to their specific needs and means.

Establishing such a search infrastructure imposes various challenges. Technological and computational aspects for setting up, maintaining, and operating the infrastructure include topics like distributed crawling, indexing and search, distributed storage of Big Data, and security. At the same time, OSF values societal aspects of search. Societal aspects include topics like the right to be forgotten, transparency, access management, and fake news detection. Thus, data governance processes, logging, and mechanisms that keep track of the data provenance are part of the scope. These heterogeneous aspects motivate the need for a robust and extensible architecture that comprises a suitable data storage system, and well-defined interfaces between the components. Note that at this scale and complexity it is not recommended to use the Big-Design-Up-Front approach [2]. Instead, we propose a basic extensible architecture that covers key aspects of the envisaged open search infrastructure.

The remainder of this paper is structured as follows: in the section *Foundations*, popular frameworks and technologies for search and data storage systems are discussed with respect to the requirements of the search infrastructure. As a starting point, in the section *A Data-Lake-Based Search Engine*, an architecture for a Data-Lake-based search engine is outlined. The architecture is then refined to meet the further requirements of the search infrastructure, in the section *Extending the Architecture*. Finally, the section *Conclusion* gives a conclusive summary. Note that the points discussed in this paper are by no means complete or settled. Instead, our goal is to fuel further discussions on the important architectural aspects of an open search infrastructure.

## FOUNDATIONS

At its core, a search infrastructure provides means of performing search tasks. Among the multiple options for implementing search systems, two popular examples are Elasticsearch [3] and Apache Solr [4]. Although Elasticsearch is more modern, the two options are similar in many regards: both are based on the well-known Apache Lucene [5], both provide indexing, querying and ranking functionality, and both support sharding. Shards are logical partitions that contain a subset of the document collection [6]. Distributing the shards across a set of storage nodes, allows the implementation of distributed indexing and search, thus improving the performance of the search system. Accordingly, the requirements of the search infrastructure that are related to these topics can be met using both Elasticsearch and Solr. But their document-focused data storage capabilities do not suffice for Big Data, i.e., very large datasets of high variability [7] comprising mostly semi-structured or unstructured data [8].

Data Warehouses (DWs) [9, 10] are a popular means of storing large amounts of data. [9] defines a DW as a physical information system that provides an integrated view on arbitrary data for data analysis. Because of the focus on data analysis, DWs are designed as analytical rather than transactional systems [9]. This has several implications. One of them is that data is only added or read, but never updated or deleted. For updating data, new versions of the data are added instead of overwriting old data. Furthermore, DWs employ rigid data models that are tailored to a specific data mining purpose [9, 10] which makes it possible to employ predefined data models and schemata that fit the intended purpose. Predefined data models and schemata facilitate processing complex queries for which analytical systems are designed. While DWs perform well for the intended purpose, they are also limited to them. Additionally, it is hardly possible to squeeze Big Data into predefined data models. These two limitations impede the applicability of DWs for the search infrastructure.

In 2010, Data Lakes (DLs) were proposed as a novel architecture for handling Big Data [7, 11]. There are several characteristics, even beyond data storage, that set DLs apart from preceding systems like DWs and make them suitable for the search infrastructure. In the following, the

† leon.martin@uni-bamberg.de
* andreas.henrich@uni-bamberg.de

aspects data storage, data governance, data interaction, and data maturation are explored.

### *Data Storage*

The utilization of the data that is added to DLs is neither known nor defined a priori, i.e., they provide a use case agnostic data storage. Instead of applying data models and schemata, the raw data is stored in their native format in the data repositories of the DL [12], thus providing the capability to digest heterogeneous Big Data. A central catalog continuously takes inventory of incoming data for later discovery [7]. DLs can receive data through regular data imports as well as through streams of real-time data [12].

Often, Apache Hadoop [13, 14], a framework for the distributed storing and processing of large datasets, is used to implement the data repositories in DLs [10, 15]. Because of Hadoop's popularity and capabilities, it is the main data storage technology addressed in this paper. Storing raw data in native format is only possible due to the significant decrease of storage costs over the past years. In Hadoop, the scalable Hadoop Distributed File System (HDFS) [16] is responsible for storing the data across multiple servers and making the data accessible.

Through user interaction, the raw data in DLs is categorized, tagged, linked, analyzed, and in other ways edited, i.e., the data matures over time. Similar to DWs, instead of overwriting the past versions of the data, copies of the data are made, then manipulated, and finally stored in the repositories of the DLs [12]. Throughout the data lifecycle, metadata is used to keep track of all changes to the data [7]. The availability of raw data and all their versions has several benefits:

1. By directly storing the raw data, the expensive activities data modelling and schema definition are deferred until a specific data representation is requested through a query [10]. The problem of squeezing semi-structured and unstructured data into predefined formats is thereby postponed. Whenever a query is executed, the requested data is bound to a dynamic schema and delivered to the users. This concept is called schema-on-read or late binding [10, 7].
2. The provenance of the data is comprehensible at all times [10]. As the data matures, DLs keep track of the changes throughout the data life cycle. Every manipulation is therefore known to the system and the users can easily comprehend where their data comes from. This is an essential capability of a transparent search infrastructure.
3. DLs yield a new level of data accessibility [10]. They are designed to encourage data sharing between the users by storing all data and their versions within the same repositories. Hence, both the raw data and the matured versions can be directly accessed by the users [12] given that the required security levels are met and thus internal political and technical barriers are avoided [10].

Despite the advantages of data storage in DLs, they are no replacement for DWs. Instead, they can be used as a staging area for DWs where structured data that is produced in the DLs is processed [8].

### *Data Governance*

In general, data governance addresses the decisions that have to be made to ensure effective management and use of IT and the agents making the decisions [17]. Making reasonable and informed decisions in complex software systems requires a clear-cut componentization, well-designed communication between the components, and a sufficient amount of decisive data. Since DLs do not enforce predefined schemata and data models, the proper use of expressive metadata is mandatory for an effective utilization of the data. Otherwise, DLs are at risk of degenerating into invisible, inaccessible, and unreliable data swamps [18]. As it is hardly possible to retransform data swamps back into DLs due to the sheer amount of data, the degeneration process must be prevented by enforcing the disciplined use of metadata that adheres to predefined standards.

According to IBM's reference architecture [12], metadata and other descriptive data is also stored in the catalog. [18] mentions key-value stores, XML documents, relational databases and ontologies as means of implementing metadata management systems for DLs. Depending on the metadata that data owners ascribe to their data when adding it, different management routines that are stored in the catalog are applied and executed by the responsible components. For example, data provided by physical sensors could be automatically preprocessed by a designated preprocessing component to eliminate noise. These processes are an important aspect of the workflow within DLs as they allow the effective utilization of the data.

### *Data Interaction & Data Maturation*

[12] describe two ways of accessing the data within a DL's repositories. The first one, raw data interaction, allows accessing the data in their actual format. This interaction style allows analysis and maintenance in case of a problem as well as masking sensitive personal information for all other components [12]. However, only admins and otherwise eligible persons should be allowed to use this interaction style due to the high degree of freedom that this style provides. For most applications, this interaction style is not suited anyway because it is hard to perform effective analyses on the vast amounts of files of raw Big Data without appropriate interfaces and tools efficiently. In Hadoop, raw data interaction can be easily realized by directly accessing the HDFS.

For regular use cases, the second interaction style, view-based interaction, is intended. This interaction style supports ad-hoc queries, search, simple analytics and data exploration [12]. Depending on the use case, for which the data is requested the data is flattened, simplified, and labeled using schema-on-read. In the Hadoop ecosystem, there are several technologies that can be used to implement an interface for view-based interaction. Hadoop itself

provides Hadoop MapReduce [14], which is an implementation of the MapReduce programming model [19]. A MapReduce job works as follows [19]: first, the input files are partitioned into so called splits across multiple machines. Then, workers apply mapping functions to the individual splits that transform the partitioned data and result in intermediate files. Next, workers apply reduce functions to the intermediate files that combine the transformed partitioned data and thus produce output files. Because of the divide-and-conquer-like processing that can be distributed across multiple machines, MapReduce jobs are a powerful and fast means of processing and generating large datasets [19]. The Hadoop MapReduce jobs can be conveniently implemented in Java.

Another option for implementing view-based interaction is Apache Hive [20]. Hive is a framework for data warehousing that builds on top of Hadoop. The framework takes queries written in HiveQL, a dialect of SQL, and transforms them into MapReduce jobs that are processed by Hadoop [14]. While Hive is a technology that is intended for deep analyses of the data [14], there are other applications that require fast, random, and real-time read/write access to the data. For such applications, Apache HBase [21], which follows the concepts of Google Bigtable [22], is an option. Hence there are several possibilities for implementing view-based interaction.

The new views on the data and the extracted datasets that are created using view-based interaction contribute to the maturation of the data [10]. Other activities that contribute to the data maturation are the consolidation and categorization of the raw data, attribute-level metadata tagging and linking, and business-specific tagging and synonym identification which eventually result in the convergence of meaning within context [10]. When this level is reached, a deep understanding of the data has emerged which leads to high applicability for various use cases.

## A DATA-LAKE-BASED SEARCH ENGINE

DLs are highly flexible and use case agnostic data management systems. Their generic architecture provides capabilities that go beyond pure data storage. The IBM reference architecture for DLs [12] is designed to be applicable in many different domains. Consequently, some included components might not be necessary for the open search infrastructure. To cope with this, we provide an architecture for a DL-based search engine that only comprises the bare minimum of components as a starting point. The components that are left out in the following can still be added to the architecture later if the need for their capabilities arises. Afterwards, we extend the architecture to meet further requirements of the search infrastructure.

### *Storage Components*

In terms of storage, document-centric search engines commonly comprise the following components [23]: a document data store, an index, and a repository for log data. The following paragraphs show how those components can be realized using DLs.

**Document Data Store** To be able to search documents, the document data store must be populated with crawled documents first. The crawled documents can both be ingested through a crawler or stored in a separate database and then imported into the DL repository. In traditional search engines, the document data store is a (relational) database that stores the original documents for fast access as well as associated structured data like metadata, links, and anchor text [23]. Using a DL as the basis for a search engine, the document data store is located inside the repository of the DL. Note that a document data store is just a matured view on raw documents and that the store-everything-approach of DLs allows embedding entire databases within the DL repository. Consequently, the document data store, e.g., a relational database, can be derived from the raw documents, stored in the DL repository, and finally accessed. This way, the same functionality compared to a non-DL-based document data store is provided since no changes to the document data store in terms of its workings are made.

The HDFS is based on the write-once, read-many-times pattern, i.e., it performs best when the source is often copied and then analyzed over time [14]. This pattern suits search engines because the stored documents are expected to be red more often than updated after being crawled.

**Index** In search engines, the index is used to gather statistics and, following that, to calculate weights that are used for ranking documents, e.g., using TF-IDF [23]. From the point of view of a DL-based search engine, the index is essentially a dataset that is derived from the stored documents. Like the document data store, the index is thus stored in the data repository of the DL-based search engine.

To construct the index, the documents must be retrieved from the document data store in the data repository first. However, the HDFS is just a means of storing data, not of performing operations on it. As explained in [24], Solr [4] supports a direct integration with Hadoop: after replacing the local file system with the HDFS in the configuration files, Solr writes and reads the index and transaction log files to and from Hadoop. This eliminates the need for a manual implementation of an interface between the search platform and the storage system. Search platforms that do not support an integration with Hadoop require such an implementation either using the view-based interaction or the raw data interaction.

In the competitive domain of ad-hoc retrieval, fast query processing is mandatory. This can only be achieved through fast interaction with the index. For this purpose, the index is usually distributed and replicated across multiple locations [23]. This requirement is unproblematic using Hadoop since replication is one of Hadoop's core concepts.

**Transaction Log Data** As [25] explains, transactional log data that are gathered as users interact with the search engine, are an invaluable resource. By analyzing this data, insights regarding the information-searching process of users and the general user behavior are gained [23, 25]. This knowledge can then be applied to the information system design, the interface development, and the

development of information architectures for content collections [25]. Furthermore, log data facilitates tuning the ranking functionality, thus improving the central functionality of a search engine [23].

In a DL-based search engine, the transactional logs are stored in the data repository as well. Unlike the previously discussed components, most log data are not derivations of already available data, but new data continuously generated through user interaction. In this regard, Solr's direct integration with Hadoop is again convenient as it provides a pre-implemented means of writing log data to the DL. Accordingly, other solutions might require manually implementing an interface for this purpose [24].

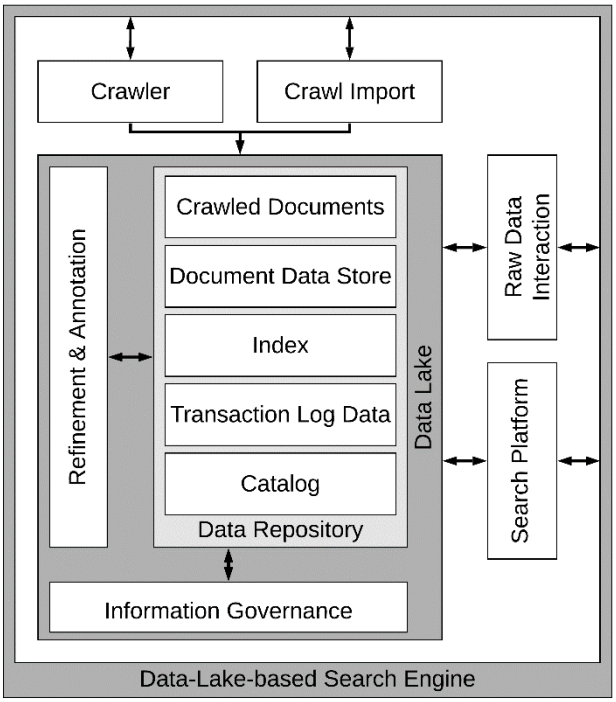


Figure 1: The architecture of a DL-based search engine.

## *Architecture*

Following the visualization style of the IBM Redguide [12], Fig. 1 shows the architecture of our DL-based search engine. As explained, the document data store, the index, and the transaction log data are all located within the data repository of the underlying DL. The metadata of these resources describe their origin and the changes made to them. Additionally, the catalog takes inventory of data in the data repository such that this information is available, e.g., for statistics and for the accessing interfaces. Since the catalog is a data storing component as well, it is also located within the data repository. Crawled documents can either be fed in directly via a crawler, e.g., Apache Nutch [26], Scrapy [27], and Heritrix [28], or imported as archives, e.g., the releases of commoncrawl [29], through the same interface. From a strategic point of view, both options should be considered here to gain the maximum number of crawled documents. Utilizing existing crawl data also allows working on and testing the system before an appropriate crawler has been determined and set up.

The internal component *Refinement & Annotation* is responsible for creating and updating the document data store with respect to incoming crawled documents. Potential duplicates are handled by this component, too. After the document data store is created, the search platform accesses it to construct the index while additionally providing a search interface for the users. As users interact with the search platform, transaction log data is collected and deposited. Depending on the search platform the interaction with the data lake must be implemented manually or is already provided, e.g., using Solr and Hadoop. In any case, an additional interface for raw data interaction is still required for accessing the raw data for maintenance and troubleshooting in case of a problem.

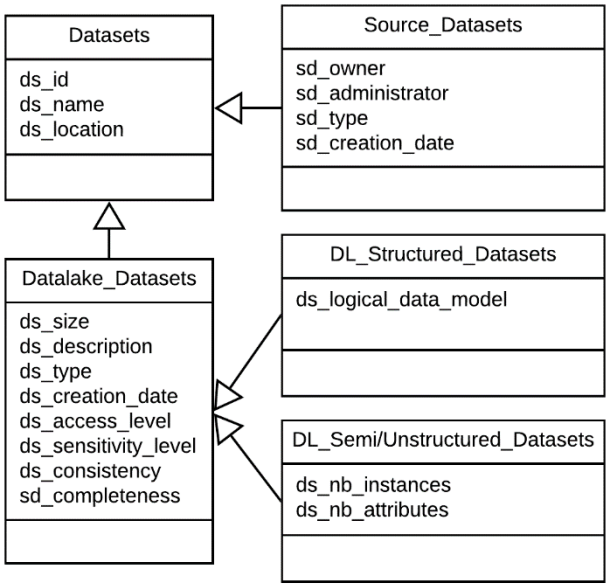


Figure 2: The inheritance tree of the *Datasets* class adopted from the conceptual metadata model in [18].

Storing data in their original format comes with the benefit that the interaction between the DL and the supplementary components of the Data-Lake-based search engine does not dictate specific formats for the data payload. However, the data payload must be described, e.g., using metadata, though. Otherwise, the DL has no information about the proper handling of incoming data and is prone to degenerating into a data swamp. For example, incoming crawled documents must be described such that the refinement and annotation processes to construct or update the document data store are triggered. As pointed out, the conditions and the associated management routines are managed within the catalog. Initiating the routines when triggering conditions are met is the responsibility of the internal component *Information Governance* that continuously monitors the events in the DL and ensures correct operation. We recommend enforcing the use of descriptions for all data at each interface such that no ambiguous data can enter the DL. In our case, this means that the four components interacting with the DL must support describing the data payload and apply a fallback description if none is provided by the data owners. The descriptions themselves

must be standardized to a certain degree, e.g., by employing a set of predefined tags or keywords, such that triggering conditions can be easily defined. Furthermore, data owners, dataset IDs, creation timestamps, and dataset sizes are to be set as explicit metadata by the system by default for later identification and provenance analyses. When data that does not fulfill the standards regarding metadata are about to be added, the *Information Governance* component intervenes and cancels the operation.

Due to the underlying use case agnostic DL, the architecture is easily extensible even if new components require special data structures for storage. To be effective at a pan-European scale the technology implementing the DL must feature seamless distribution and replication of data for fast data delivery.

## EXTENDING THE ARCHITECTURE

To enable the architecture meeting further requirements of the envisaged open search infrastructure some of the previously discussed components and processes must be extended while some new must be added. The previous section already mentioned a few types of necessary metadata. However, they do not suffice as metadata for the full search infrastructure. [18] proposes a metadata conceptual model for DLs containing twelve classes each featuring different metadata. Fig. 2 shows the inheritance tree of the model's *Datasets* class as an example. As one can see, the model includes several types of metadata that must be respected by all components. One prominent example is the metadata addressing the sensitivity level of the DL's datasets. Previously, all documents in the DL-based search engine were accessible through the search platform as soon as they have been indexed unless they are removed or masked manually via the *Raw Data Interaction* component. This naïve approach clearly violates data privacy aspirations because it does not enable data privacy by design. A better approach is to extend the responsibilities of the *Information Governance* component such that data accesses are constantly monitored, and users' access rights are evaluated against the specified sensitivity level of the accessed data.

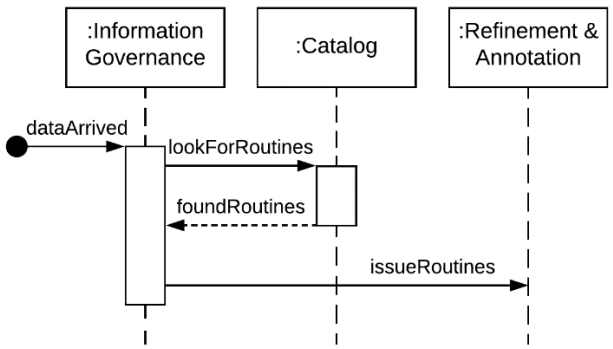


Figure 3: A possible approach for the workflow of routine application.

Generally, the responsibilities of the *Refinement & Annotation* component include the application of automated predefined routines to the data in the repository for maximizing their utility. Currently, the component takes care of creating and updating the document data store for making use of the data in the search engine use case. Aside from this, the information within the document data store may also by useful for statistical purposes. However, there are other use cases for which the data could be prepared automatically. For instance, natural language processing operations could be applied to the data or related documents could be explicitly linked using metadata, thus improving the utility of the data. Automatically applying such routines comes with the benefit that the resulting data derivations are already available for interested users. For example, scientists in natural language processing could access already computed dependency parse trees without having to issue the required routines themselves. Obviously, not all future usages of the data can be anticipated. Hence, we recommend continuously inspecting the log data to identify data that are requested frequently and adding routines to the *Refinement & Annotation* accordingly. To enable the automatic application of new routines, appropriate triggering conditions must be specified in the catalog. Fig. 3 shows one possible approach as a sequence diagram for the routine application workflow.

So far, interaction with the system is only supported via raw data interaction and the search platform which limits the potential use cases. To open the system for other purposes, an additional component for view-based interaction that interacts with the DL is required to provide additional user interfaces. In the beginning, the interfaces provided by this component are rather generic because the future usages are not known a priori. Appropriate initial interfaces allow users to perform operations that contribute to the data maturation like attribute-level metadata tagging, as said before. Such functionalities require the ability to browse and edit data which can be implemented using technologies like HBase and Hive. Over time new interfaces for frequent use cases can be added. Again, using the example of natural language processing, an interface could provide means of following coreferences, i.e., phrases that refer to the same named entity, across documents for explorative search. The possibilities in this regard are numerous. Note that multiple aspects of the proposed architecture depend on log data. Hence, aside from the architectural aspects of the open search infrastructure, user engagement is a mission critical aspect. To attract and convince users, designing fast, flat, and powerful interfaces is mandatory.

## CONCLUSION

In this paper, we summarized the goal of the Open Search Foundation to establish an open search infrastructure and discussed the core aspects of Data Lakes including data storage, data governance, data interaction, and data maturation to assess their applicability for such an ambitious infrastructure. Subsequently, we proposed the architecture for a basic Data-Lake-based search engine as a starting point comprising the minimum number of necessary components. To cover further requirements of the infrastructure we added new and extended existing components afterwards, while also providing recommendations

regarding metadata and user interface design. The next steps are to further discuss the architectural aspects of the open search infrastructure with the goal of determining and subsequently building up a first basic but extensible architecture for the open search infrastructure soon. To this end, we will further develop and expand the architectural considerations based on selected application scenarios. Examples besides search engines are web analytics or alert services.

## REFERENCES


[1] Open Search Foundation, https://opensearchfoundation.org/.

[2] The Architecture of a Large-Scale Web Search Engine, https://0x65.dev/blog/2019-12-14/the-architecture-of-a-large-scale-web-search-engine-circa-2019.html.

[3] Elasticsearch, https://www.elastic.co/products/elasticsearch.

[4] Apache Solr, https://lucene.apache.org/solr/.

[5] Apache Lucene, https://lucene.apache.org/.

[6] Shards and Indexing Data in SolrCloud, https://lucene.apache.org/solr/guide/6_6/shards-and-indexing-data-in-solrcloud.html.

[7] N. Miloslavskaya and A. Tolstoy, “Big Data, Fast Data and Data Lake Concepts”, in *Procedia Computer Science*, pp. 300-305, 88, 2016.
doi:10.1016/j.procs.2016.07.439

[8] C. Madera and A. Laurent, “The next information architecture evolution”, in *Proceedings of the 8th International Conference on Management of Digital EcoSystems - MEDES*, New York, New York, USA, pp. 174-180, 2016.
doi:10.1145/3012071

[9] Bauer, Andreas; Günzel, Holger, Data-Warehouse-Systeme. Heidelberg: dpunkt-Verl., 2013.

[10] B. Stein and A. Morrison, “The enterprise data lake: Better integration and deeper analytics”, in *PwC Technology Forecast: Rethinking integration*, pp. 1-9, 2014.

[11] Pentaho, Hadoop, and Data Lakes, https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/.

[12] M. Chessell, F. Scheepers, N. Nguyen, R. van Kessel, and R. van der Starre, “Governing and managing big data for analytics and decision makers”, in *IBM Redguides*, 2014.

[13] Apache Hadoop, https://hadoop.apache.org/.

[14] T. White, Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (3. ed., revised and updated): O’Reilly, 2012.

[15] A. Farrugia, R. Claxton, and S. Thompson, “Towards social network analytics for understanding and managing enterprise data lakes”, in *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pp. 1213-1220, 2016.

[16] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop Distributed File System”, in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1-10, 2010.

[17] V. Khatri and C. V. Brown, “Designing data governance”, in *Communications of the ACM*, 1, pp. 148-152, 53, 2010.
doi:10.1145/1629175.1629210

[18] F. Ravat and Y. Zhao, “Metadata Management for Data Lakes”, in *New Trends in Databases and Information Systems*, Cham, pp. 37-44, 2019.
doi:10.1007/978-3-030-30278-8

[19] J. Dean and S. Ghemawat, “MapReduce”, in *Communications of the ACM*, 1, pp. 107-113, 51, 2008.
doi:10.1145/1327452.1327492

[20] Apache Hive, https://hive.apache.org/.

[21] Apache HBase, https://hbase.apache.org/.

[22] F. Chang *et al.*, “Bigtable”, in *ACM Transactions on Computer Systems*, 2, pp. 1-26, 26, 2008.
doi:10.1145/1365815.1365816

[23] W. B. Croft, D. Metzler, and T. Strohman, Search engines. Boston, Columbus, Indianapolis, New York: Addison Wesley, 2010.

[24] Running Solr on HDFS | Apache Solr Reference Guide 6.6, https://lucene.apache.org/solr/guide/6_6/running-solr-on-hdfs.html.

[25] B. J. Jansen, “Search log analysis: What it is, what's been done, how to do it”, in *Library & Information Science Research*, 3, pp. 407-432, 28, 2006.
doi:10.1016/j.lisr.2006.06.005

[26] Apache Nutch, http://nutch.apache.org/.

[27] Scrapy, https://scrapy.org/.

[28] Heritrix, https://github.com/internetarchive/heritrix3.

[29] Common Crawl, http://commoncrawl.org/.