Software Redocumentation Using Distributed Data Processing Technique to Support Program Understanding for Legacy System: A Proposed Approach

Sugumaran Nallusamy[1(✉)], Hoo Meei Hao[1(✉)], and Farizuwana Akma Zulkifle[2(✉)]

[1] Universiti Tunku Abdul Rahman, Sungai Long Campus, Jalan Sungai Long, Bandar Sungai Long, 43000 Kajang, Selangor, Malaysia
{sugumaran,hoomh}@utar.edu.my
[2] Universiti Teknologi MARA, Kuala Pilah Campus, 72000 Shah Alam, Negeri Sembilan, Malaysia
farizuwana@uitm.edu.my

Abstract. Source code is the most updated source among all the available software artifacts. The majority of existing software redocumentation approaches relied on source code to extract the necessary information for program comprehension in order to support software maintenance tasks. However, performing Extract, Transform and Load (ETL) using a parser from the source code becoming a challenging task. The traditional approach is no longer able to handle the ETL efficiently due to the effect of the analysis efficiency, especially for large source code. This paper proposed to use distributed data processing technique to extract legacy source code components to generate detailed designed or technical software documentation at source code level to support program understanding. The objective of this paper is to apply the distributed data processing technique to the parser by using Hadoop Distributed File System and Apache Spark. Legacy java source code used as a case study to apply our proposed approach to extract the source code components and generate the technical software documentation.

Keywords: Software redocumentation · HDFS · Spark · Legacy system · Program understanding · Software maintenance

1 Introduction

Industry practitioners perceived legacy systems as business critical and reliable system operated for more than ten to twenty years, but inflexible to adapt to new changes [1]. In respect of legacy system modernization, lacking of knowledge [2] and high maintenance cost [3] are the main drivers [1]. Similarly, it is a challenge for software developers to maintain and support the legacy systems due to lacking the latest documentation and increasing complexity of source codes as times are passing. Thus, software redocumentation is essential to rebuild the documentation of existing resources in order to

H. Badioze Zaman et al. (Eds.): IVIC 2021, LNCS 13051, pp. 239–252, 2021.
https://doi.org/10.1007/978-3-030-90235-3_21

give a better understanding of the system for the purpose to generate documentation for the modified programs, update the system changes, and creating alternative views. The redocumentation process can be carried out at several stages of the software design process, such as source code, design, or requirement. However, according to a survey conducted by Souza et al. [4], source code level documentation is more relevant documentation or technical that may be classified as technical documentation or functional documentation to aid in program comprehension and maintenance tasks. As specified by Geet et al. [5], the technical document contains features such as a summary of the source code, source code metrics, forms, and method dependencies that are derived from current redocumentation tools [6].

The redocumentation process is comprised of five major components: source code, parser, system knowledge base, view composer, and software documentation [7]. We concentrated on the parser in this study because it is critical for extracting the necessary information to build the documentation. The present limitation of the existing parser is its inability to extract the pieces required for technical documentation from huge amounts of old source code. Existing parsers are embedded into software tools that place a premium on traditional extraction methods, which reduces the effectiveness of extracting pertinent information due to the enormous source code size. Additionally, retrieval should be performed in a timely manner in order to comprehend the program and assist with software maintenance tasks. The limitations of the parser in the redocumentation tool in terms of handling huge source code, the explosion of big data, and the evolution of data processing technologies all encourage the investigation of the proposed approach in the process of software redocumentation.

Analysing software systems and extracting source code components requires processing of source code and rebuild the structure of information. There are many existing studies related to different techniques used in extracting the data [8, 9]. Nevertheless, those approaches used Extract, Transform, Load (ETL) based on relational query approach unable to handle streaming or near real-time data and stimulating environment which demands high availability, low latency and scalability features [10]. Although the traditional ETL may prove to be effective in managing structured, batch-oriented data which, up to date and within scope for corporate insights and decision making [10, 11], it is not suitable for source code that consists of semi structured or unstructured data [12].

Thus, we proposed an approach to use Hadoop Distributed File System (HDFS) and Spark which provide a cluster computing model for distributed computing platforms intended to run the process of redocumentation. The proposed approach would assist software developers maintain the source code efficiently and effectively solving the problems of processing, analyzing and redocument the massive source codes.

This paper is organized as follows: The first section is a background of software redocumentation and related works that gives an overview of some studies and research carried out involved big data processing. In the methodology section, we describe the proposed approach of distributed data processing in software redocumentation. The Case Study section presents the initial work following the proposed approach. We end with a conclusion and give some future works to complete the process of redocumentation.

2 Background and Related Work

As defined in the IEEE Standard for Software Maintenance (IEEE 14764-2006-ISO/IEC), after development and delivery, software maintenance undergoes a similar process as in software development to modify the software product to correct faults, improve performance, or to adapt the product to the modified environment. Software maintenance aims to preserve the software product over time. Christa et al. [13] indicated that the legacy system is a contributor to maintenance cost, effort, and productivity. A significant challenge for software developers is taking over development work if the source code is the only source of understanding the written codes and system documentation is out-of-date, lacking, or incomplete. The software developer spent more time with existing code than creating new software. These are emphasized by [13].

Nallusamy et al. [7, 14] mentioned that software redocumentation is a software documentation update created within the same abstraction and in line with the latest developments of the code. Additionally, it includes analyzing a static representation of a software system to give a different perspective. Earlier studies examined software redocumentation to support software evolution and software maintenance. Essentially, re-documentation is intended to help developers comprehend programs [15, 16]. The results of a four-year long case study [17] demonstrate a significant decrease in maintenance costs and effort due to redocumentation effort. Methods and approaches to solve program understanding have attracted the software engineering community. Such development in this area can be seen from the studies on the category of redocumentation approaches comprise of XML based approach, incremental redocumentation, model-oriented redocumentation, island grammar, doclike modularized graph, ontology-based approach [7], and reverse engineering to transform the source code to UML diagrams such as [18–20].

Most evaluation studies of format, granularity, and efficiency showed that all approaches were of low quality when it came to granularity and efficiency [14]. Comparison of two approaches of incremental and model-oriented had shown different usage and purposes [21]. This approach works if outdated or missing documentation is an issue. Rebuild the documentation incrementally, while using a model-oriented approach will produce models from existing systems and generate documentation based on the models. As a result, model-oriented approach is best for redocumenting a legacy system from source code. In other words, effective and efficient software maintenance is necessary. Moreover, code analysis keeps evolving in terms of technique and application development.

As software becomes more complex over time, the number of lines of source code increases, particularly for huge legacy systems. The team has spent several years writing and maintaining these program codes. When new programmer takes over the maintenance job including change request, new programmer required an efficient tool to extract the software components from the source code which handle by the parser in the software redocumentation process. Current parsers in re-documentation tools may be incapable of handling this massive volume of data, as they were not designed for high-volume data processing [22].

As a result, the endeavor to leverage contemporary technology in the management of big legacy systems continues to evolve, as indicated by Wolfart et al. [23]. On the

other hand, the work of Ruchir Puri et al. [24] emphasized the necessity of artificial intelligence in acquiring information from huge amounts of source code in order to assist software maintainers in performing maintenance activities. Verena Geist et al. [22] used machine learning to analyze source code comments. These studies focus the strategies used to circumvent the current difficulties connected with digesting large volumes of source code in order to comprehend and conduct software maintenance activities on time. Additionally, several of these investigations used cutting-edge data processing methods to redocument the source code. As a result of these investigations, we have begun to investigate data processing using distributed computing frameworks based on commodity cluster designs, such as Hadoop and Apache Spark. This approach is widely utilized in a variety of fields for the processing of large amounts of data and is constantly evolving [25]. Recent examples include processing and analyzing YouTube big data to determine the success of films and items in comparison to competitors [26] and analyzing airline delays using Spark [27].

Apache Spark is distributed computing system designed to run in a cluster, it is also fast and general purpose. Spark extends the MapReduce model of Hadoop to efficiently support more types of computations, including interactive queries and stream processing [28]. One of the main features Spark offers for speed is the ability to run computations in memory, but the system is also more efficient than MapReduce for complex applications running on disk. The core data units in Spark are called Resilient Distributed Datasets (RDDs). They are only read only collections which partitioned to multiple machines and can be rebuilt if the partitions are lost. RDDs are collections of elements that are distributed, immutable, and fault-tolerant. They can be produced by performing a series of actions on either stable storage data or other RDDs. RDDs can be stored in memory, on disk, or in a combination of the two storage media types. Furthermore, RDD is adopting the Lazy Evaluation approach in order to complete the action task. This is done in order to ensure that compute and memory are used as less as possible. As RDDs are not cached in RAM by default, a persist method is required when data is reused to avoid re-computation [27]. One of the advantages of the Spark environment is provide Application Programming Interface in Scala, Phyton and Java. Furthermore, Spark provides Spark Context as a master node and distributed to worker node through cluster manager. Spark allows to configure properties such as the number of executors, the number of cores per executor, and the amount of memory per executor for each application [29].

Hadoop Distributed File System (HDFS) a reliable and has scalable storage and processing system for a large volume of distributed unstructured data. On the other hand, Apache spark used to speed the processing power which is 100 times faster in memory and 10 times faster by running on disk. Thus, HDFS is an ideal technique to develop a highly scalable application that able to process massive data as compared to a traditional method such as database management systems [25]. As far as our knowledge goes until this paper is written, none of the study that using distributed data processing techniques in the field of software maintenance. This paper shows the approach of using HDFS and Spark environment to generate documentation to assist in software maintenance.

3 Proposed Approach

Our main contribution to the suggested solution is the development of a parser that is utilized to extract the source code component using HDFS and Apache Spark via a distributed data processing technique. As a result, the parser processes raw source code using a distribution strategy to accelerate the process of extracting source code components within the constraints of limited run times.

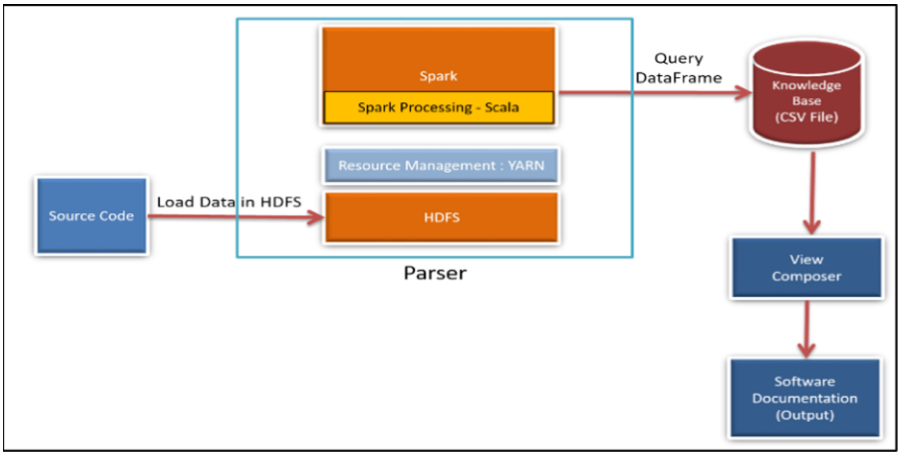

Fig. 1. Software redocumentation using distribution data technique

Figure 1 illustrates the system architecture for locating and generating source code components via the HDFS and Spark environments. Each component is thoroughly explained as follows:

3.1 Legacy Source Code (LSC)

A software work product or artefact consists of source code, configuration files, built scripts, and auxiliary artifacts [7, 30]. However, this study looked at only LSC during the redocumentation process [4]. SWPs were excluded for two main reasons. First, the most up-to-date or reliable source is the source code. SWPs contain greater precision when compared to other data sets. Second, SWPs are poorly maintained when compared to the source code [31]. Legacy systems undergo numerous changes. Additionally, these changes include numerous software maintainers, who must spend almost half of their time understanding the program's functionality versus their total time spent on maintenance. This problem affects the software maintenance efficiency.

3.2 Parser

The parser is used to extract necessary information from the SWP and store it in the repository. The proposed approach utilized HDFS for storing, processing, and analyzing

the LSC across multiple nodes of commodity hardware. There will be a master node (Name Node) and a slave node (Data Node) [27]. A Name node distributes the works to data node at load time and blocks from the same file are all on different machines. When a data node is failed, it is replicated across multiple data nodes. Yarn acts as a distributed container for the master node's resources. Figure 2 shows how the source code is distributed across the network through the data nodes. Distributed computing has multiple advantages. It's scalable and makes it easier to share resources. It also speeds up computation tasks.

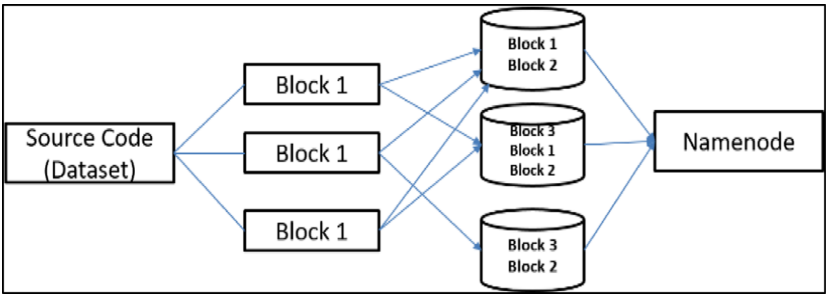

Fig. 2. Block replication of the source code dataset in Hadoop cluster

In our proposed context, Spark plays an important role as a parser [29], performing ETL from source code and providing source code components. Yarn is a distributed container manager, like Mesos for example, whereas Spark is a data processing tool. Spark can run on Yarn. We used Scala as the programming language and the Databricks Community Cloud (DCC) platform [32] to execute the Scala code, which includes a notebook (workspace) and a spark session. Figure 3 illustrates the Spark data processing flow in DCC:

Fig. 3. Spark data processing flow

3.3 Knowledge Based

A repository component is used in the software redocumentation knowledge based on the data process and produce documentation as defined by Nallusamy et al. [7]. It provides appropriate semi-structured data on source code for building documentation content as well as allowing browsing and searching for relevant content in the documentation [33]. In the redocumentation, some current repositories used conventional models like flat archives, databases, or knowledge bases [34]. These repositories must be used to locate and create different views or documents that software maintainers have requested. These

capabilities save software maintainers time and effort in learning about the application domain. As a result, in our proposal, we used a Command Separated Values (CSV) file to store the flat file in the repository and convert it to a data frame during document generation. Data frames enable the query for specific data to be used in the documentation.

3.4 View Composer

View Composer is a user interface that is used to interact with a knowledge-based system in order to retrieve specific components [7]. In addition to a list of modules, the interface needed to be able to see the dependencies between them. Understanding the dependencies between the components is a crucial problem during software maintenance tasks. Software maintainers must be aware of the change impact of making improvements to a specific piece of source code. Therefore, it's critical to use the search and browsing functions to locate the relevant item.

3.5 Technical Documentation

According to a survey conducted by Souza et al. [4], the most important documents for software maintenance are the source code and comments. However, the problem with outdated comments in the source code can lead to the wrong interpretation of the meanings of the code. Moreover, experts are not available and new software maintainers may find it difficult to understand the current system for carrying out the maintenance tasks. In this perspective, Van Geet et al. [5] emphasized a redocumentation technique to generate a detailed design document. This document is related to the technical documents containing the structure with all the functions, database tables, screens, batch jobs, dependencies among the components and the slices of the program [35].

Table 1. Software technical documentation schema

Technical documentation section	Components
Source code metrics	• Finding out the number of method in a class • Total lines of code • Total number of imported libraries • Total number of class in a class • Total word counts • Total number of public function • Total number of the private method • Total number of a protected method
Object and descriptions	• Find out what are the available packages in the application • List the classes, interfaces, abstract classes in the packages • List the functions that are in the classes

(*continued*)

Table 1. (*continued*)

Technical documentation section	Components
Dependency diagram	• Dependency analysis – Package level analysis – Class level analysis – Function level analysis

In the proposed approach, as shown in Table 1, the documentation generated is the technical document that consists of certain elements, such as the source code summary, source code metrics, classes, packages and functional dependencies. The functionalities and elements in this technical documentation are defined to retrieve only relevant parts of the source code using the HTML based documentation, as suggested by Van Geet et al. [5].

4 Case Study

The implementation of this approach is still in the initial stage. In this stage, we have created a simple prototype and have implemented each process specified in Fig. 1. The detailed process of implementation is described in the following sections.

4.1 Legacy Source Code

We used Restaurant Management System legacy source code for this proposed model. This software, which was built ten years ago in Java, provides restaurant management for customers. This end-to-end restaurant management system manages orders, inventory, and employee management. All orders and employee data will be stored in a database. The application has 14905 lines of code. These java files include the backend database code until the front-end GUI interfaces.

4.2 Parser

Once the LSC identified, the first step is to load LSC into HDFS environment. As specified earlier, we used Scala as a programming language and DCC as a cloud platform. We have created the cluster and notebook space to execute the Scala commands. During data preparation, we have identified and grouped java files on Restaurant Management system as specified in the previous section loaded in the data directory of the cloud platform. Next step, we load the source code into RDD which is the fundamental storage unit of Spark in order to extract some information from the source code to do analysis. Spark automatically and transparently divides the data in RDDs into partitions which are distributed across worker nodes in the cluster and parallelize the data performed on these partitions as specified in Fig. 2. Figure 4 shows the command to load the source code into RDD for each 14 java file. After loading all the files into their respective RDD,

```
val file1 = sc.textFile("/FileStore/tables/src/Controller.java")
val file2 = sc.textFile("/FileStore/tables/src/Controller_GUI.java")
val file3 = sc.textFile("/FileStore/tables/src/Database.java")
val file4 = sc.textFile("/FileStore/tables/src/DatabaseException.java")
val file5 = sc.textFile("/FileStore/tables/src/Employee.java")
val file6 = sc.textFile("/FileStore/tables/src/Manager.java")
val file7 = sc.textFile("/FileStore/tables/src/MenuItem.java")
val file8 = sc.textFile("/FileStore/tables/src/Order.java")
val file9 = sc.textFile("/FileStore/tables/src/OrderDetail.java")
val file10 = sc.textFile("/FileStore/tables/src/RMS.java")
val file11 = sc.textFile("/FileStore/tables/src/RMS_GUI.java")
val file12 = sc.textFile("/FileStore/tables/src/Staff.java")
val file13 = sc.textFile("/FileStore/tables/src/UserInterface.java")
val file14 = sc.textFile("/FileStore/tables/src/UserInterface_GUI.java")
```

Fig. 4. Load source code into RDD

we created a list to store all the RDD so that created RDD it is easy to use later by using loop structure instead of typing a command that works for each RDD repeatedly.

In data transformation, the process of classification done through the process of filtration in the data loaded in RDD using Action and Transformation operation. The main classification that needs to be done in this source code is based on the documentation section specified in Table 1. RDD transformation commands such as filter, map, flatMap used in our proposed approach to filter the source code by extracting java packages, classes, interfaces and abstract class in java packages. On the other hand, RDD

```
//read class files
val readClassFile = sc.textFile("/FileStore/tables/src/Classes/*")

//get all libraries name used in class
val findLibrariesInClasses = readClassFile.filter(lines => lines.contains("import") && lines != "").map(x =>
x.replace(";","")).map(x => x.replace("import", "")).map(x => x.trim).distinct
//transform into dataframe
import spark.implicits._
// for implicit conversions from Spark RDD to Dataframe
val librariesInClassDf = findLibrariesInClasses.toDF("Libraries")

librariesInClassDf.show
//save the dataframe into csv
// librariesInClassDf.repartition(1).write.format("com.databricks.spark.csv").option("header",
"true").save("dbfs:/FileStore/tables/librariesInClass.csv")
```

Fig. 5. Partial Scala code to extract component

Actions such as count and aggregate used to perform aggregation functions to provide the source code metrics such as finding the total line of codes, imported libraries, classes, packages and other relevant components. Figure 5 shows the partial Scala command use RDD Transformation and Action used to extract source code components in Spark environment.

On the other hand, one more important component that needs to be extracted is the dependency which in our implementation emphasize Package, class and function dependency analysis. Figure 6 shows partial Scala code to extract the functional dependency.

```
//Using Databricks Community Edition

//remove file if exist
dbutils.fs.rm("/FileStore/tables/ClassDependency",true)

//remove ; and comments
def trimming(item:String) :(String) = {
  val pos = item.indexOf(";")
  item.substring(0,pos)
}

//find all class name to act as filepath later
val allClass = sc.textFile("/FileStore/tables/Classes/*")
val findClass = allClass.filter(x => (x.contains("public class") || x.contains("public enum"))).map(line => line.split(" "))
val extractClass = findClass.map(x => x(2))

```

Fig. 6. Extract the dependency code

The next process is to load dataset to dataframe. Source code components are extracted and dependencies loaded into a DataFrame. We performed column transformation, and query the DataFrame to get useful information such as code metrics, source code component list and component dependencies to save into CSV file.

4.3 Knowledge Based

The extracted source code components stored in few CSV files based on documentation elements namely source code metrics, and list the components and dependencies among the components.

4.4 View Composer

View Composer or in our context called web-based user interface provides related function to extract the components and present them as HTML documentation. The user interface will use the data from CSV and loaded it into dataframe.

4.5 Technical Documentation

As specified in Sect. 4.2, once data loaded into dataframe, SPARK SQL used to query and retrieve relevant source code components and classified according to documentation

Class	Total number of methods	lines of code	number of	number of	number of class	Total of words counts	Total Number of public method	Total Number of private method
Controller.java	35	1887	318	3	1	35004	4	31
Controller_GUI.java	37	633	104	2	1	13025	36	1
Database.java	42	835	148	4	3	17565	38	4
DatabaseException.java	2	10	0	0	1	154	2	0
Employee.java	4	26	0	0	1	465	4	0
Manager.java	4	29	0	0	1	488	4	0
MenuItem.java	12	85	0	0	1	1209	12	0
Order.java	12	112	8	1	1	1664	12	0
OrderDetail.java	7	47	3	0	1	841	7	0
RMS.java	1	11	1	0	1	161	1	0
RMS_GUI.java	1	11	0	0	1	175	1	0
Staff.java	21	176	4	2	1	3588	16	0
UserInterface.java	33	500	42	2	1	12773	30	3
UserInterface_GUI.java	84	2219	518	8	13	50921	50	34
Total	**295**	**6581**	**1146**	**22**	**28**	**138033**	**217**	**73**

dependency	function name	dependent function	class
Iterator<OrderDetail> it = orderDetailList.iterator();	addItem	iterator	Order
while(it.hasNext() && !found)	addItem	hasNext	Order
re = it.next();	addItem	next	Order
if(rNewMenuItem.getID() == re.getItemID())	addItem	getID	Order
re.addQuantity(quantity);	addItem	addQuantity	Order
orderDetailList.add(detail);	addItem	add	Order
orderDetailList.remove(index);	deleteItem	remove	Order
//System.out.println(e.toString() + ":" + e.getMessage());	deleteItem	println	Order
Iterator<OrderDetail> it = orderDetailList.iterator();	calculateTotal	iterator	Order
while (it.hasNext()) {	calculateTotal	hasNext	Order
re = it.next();	calculateTotal	next	Order
total += re.getTotalPrice();	calculateTotal	getTotalPrice	Order

Fig. 7. Generate software technical documentation

section as shown in Table 1. The sample technical documentation generated can be referred to Fig. 7.

On other hand, the functional dependencies also shown in Fig. 8 below generated from the Scala code in Fig. 6.

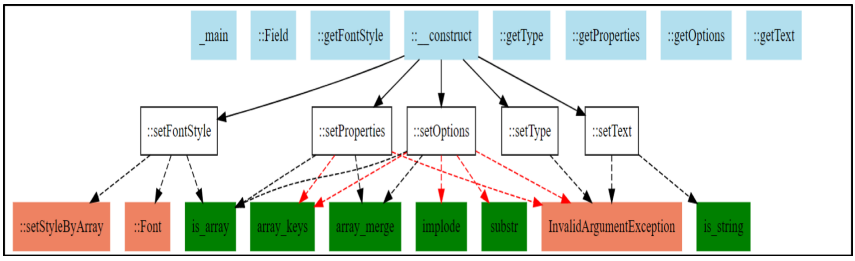

Fig. 8. A function call graph

5 Conclusions and Future Work

In this paper, we have presented our proposed approach for software redocumentation that employs the distribution data technique. As an initial effort, we present the system architecture for locating and generating the source code component through HDFS and Spark environments. Our experiments focus on the parser used to extract the source

code components from the SWP and store it in the repository. As a result, Sparks plays an important role as a parser to perform ETL on legacy java source code. The significance of the experiment shows the process of a raw source code by using a distribution technique. This technique helps to speed up the extraction process of the source code component within limited run times. For future work, we plan to use the same approach in different languages and other large legacy systems with precise evaluation to improve the efficiency of our proposed approach.

References

1. Khadka, R., Batlajery, B.V., Saeidi, A.M., Jansen, S., Hage, J.: How do professionals perceive legacy systems and software modernization? In: Proc. Int. Conf. Softw. Eng., pp. 36–47 (2014). https://doi.org/10.1145/2568225.2568318
2. Matthiesen, S., Bjørn, P.: Why replacing legacy systems is so hard in global software development: an information infrastructure perspective. In: Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing, pp. 876–890 (2015)
3. Crotty, J., Horrocks, I.: Managing legacy system costs: a case study of a meta-assessment model to identify solutions in a large financial services company. Appl. Comput. Inform. **13**, 175–183 (2017)
4. de Souza, S.C.B., Anquetil, N., de Oliveira, K.M.: Which documentation for software maintenance? J. Braz. Comput. Soc. **12**(3), 31–44 (2007). https://doi.org/10.1007/BF03194494
5. Van Geet, J., Ebraert, P., Demeyer, S.: Redocumentation of a legacy banking system: an experience report. In: Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), pp. 33–41 (2010)
6. Tadonki, C.: Universal Report: a generic reverse engineering tool. In: 12th IEEE International Workshop on Program Comprehension (IWPC 2004), pp. 266–267 (2004)
7. Nallusamy, S., Ibrahim, S., Mahrin, M.N.: A software redocumentation process using ontology based approach in software maintenance. Int. J. Inf. Electron. Eng. **1**, 133 (2011)
8. Dorninger, B., Moser, M., Pichler, J.: Multi-language re-documentation to support a COBOL to Java migration project. In: SANER 2017 – 24th IEEE Int. Conf. Softw. Anal. Evol. Reengineering, pp. 536–540 (2017). https://doi.org/10.1109/SANER.2017.7884669
9. Kienle, H.M., Müller, H.A.: Rigi – an environment for software reverse engineering, exploration, visualization, and redocumentation. Sci. Comput. Program. **75**, 247–263 (2010). https://doi.org/10.1016/j.scico.2009.10.007
10. Sabtu, A., et al.: The challenges of Extract, Transform and Loading (ETL) system implementation for near real-time environment. In: Int. Conf. Res. Innov. Inf. Syst. ICRIIS, pp. 3–7 (2017). https://doi.org/10.1109/ICRIIS.2017.8002467
11. García, S., Ramírez-Gallego, S., Luengo, J., Benítez, J.M., Herrera, F.: Big data preprocessing: methods and prospects. Big Data Anal. **1**, 1–23 (2016). https://doi.org/10.1186/s41044-016-0014-0
12. Ragab, M., Tommasini, R., Awaysheh, F.M., Ramos, J.C.: An In-depth Investigation of Large-Scale RDF Relational Schema Optimizations Using Spark-SQL (2021)
13. Christa, S., Madhusudhan, V., Suma, V., Rao, J.J.: Software maintenance: from the perspective of effort and cost requirement. In: Proceedings of the International Conference on Data Engineering and Communication Technology, pp. 759–768. Springer (2017)

14. Sugumaran, N., Ibrahim, S.: An evaluation on software redocumentation approaches and tools in software maintenance. In: Commun. IBIMA, pp. 1–10 (2011). https://doi.org/10.5171/2011.875759
15. Kaur, U., Singh, G.: A review on software maintenance issues and how to reduce maintenance efforts. Int. J. Comput. Appl. **118**, 6–11 (2015). https://doi.org/10.5120/20707-3021
16. Kaur, P.: The study of software re-engineering. WWJMRD **4**, 381–383 (2018)
17. Rostkowycz, A.J., Rajlich, V., Marcus, A.: A case study on the long-term effects of software redocumentation. In: IEEE Int. Conf. Softw. Maintenance, ICSM, pp. 92–101 (2004). https://doi.org/10.1109/ICSM.2004.1357794
18. Nanthaamornphong, A., Leatongkam, A.: Extended ForUML for automatic generation of UML sequence diagrams from object-oriented Fortran. Sci. Program. (2019). https://doi.org/10.1155/2019/2542686
19. Singh, K.: Transformation of source code into UML diagrams through visualization tool. Int. J. Adv. Sci. Technol. **29**(8), 4861–1114 (2020)
20. Sheer, A., Tahrawi, A., Jeesh, J., Al Ibrahim, Y.: A Framework for software re-documentation by using reverse engineering approach. Int. J. Comput. Appl. **118**, 1–21 (2016)
21. Pathania, Y., Bathla, G.: A review on re-documentation approaches and their comparative study. Int. J. Comput. Sci. Trends Technol. **2**, 48–51 (2014)
22. Geist, V., Moser, M., Pichler, J., Beyer, S., Pinzger, M.: Leveraging machine learning for software redocumentation. In: SANER 2020 – Proc. 2020 IEEE 27th Int. Conf. Softw. Anal. Evol. Reengineering, pp. 622–626 (2020). https://doi.org/10.1109/SANER48275.2020.9054838
23. Wolfart, D., et al.: Modernizing legacy systems with microservices: a roadmap. In: Evaluation and Assessment in Software Engineering, pp. 149–159. Association for Computing Machinery (2021)
24. Puri, R., et al.: Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. https://arxiv.org/abs/2105.12655 (2021)
25. Casado, R., Younas, M.: Emerging trends and technologies in big data processing. Concurr. Comput. **27**, 2078–2091 (2015). https://doi.org/10.1002/cpe.3398
26. Shaikh, F., Pawaskar, D., Siddiqui, A., Khan, U.: YouTube data analysis using MapReduce on Hadoop. In: 2018 3rd IEEE International Conference on Recent Trends in Electronics, Information and Communication Technology, RTEICT 2018 – Proceedings, pp. 2037–2041 (2018). https://doi.org/10.1109/RTEICT42901.2018.9012635
27. Nibareke, T., Laassiri, J.: Using Big Data-machine learning models for diabetes prediction and flight delays analytics. J. Big Data **7**(1), 1–18 (2020). https://doi.org/10.1186/s40537-020-00355-0
28. Jonnalagadda, V.S., Srikanth, P., Thumati, K., Nallamala, S.H., Dist, K.: A review study of apache spark in big data processing. Int. J. Comput. Sci. Trends Technol. **4**, 93–98 (2016)
29. Han, Z., Zhang, Y.: Spark: a big data processing platform based on memory computing. In: Proc. – Int. Symp. Parallel Archit. Algorithms Program, PAAP, pp. 172–176 (2016). https://doi.org/10.1109/PAAP.2015.41
30. Chikofsky, E.J., Cross, J.H.: Reverse engineering and design recovery: a taxonomy. IEEE Softw. **7**, 13–17 (1990)
31. Müller, H.A., Kienle, H.M.: A Small Primer on Software Reverse Engineering (2009)
32. Databricks Community Edition. https://community.cloud.databricks.com. Accessed 10 November 2020

33. Van Deursen, A., Moonen, L.: Documenting software systems using types. Sci. Comput. Program. **60**, 205–220 (2006)
34. Canfora, G., Di Penta, M., Cerulo, L.: Achievements and challenges in software reverse engineering. Commun. ACM **54**, 142–151 (2011)
35. Freeman, R.M., Munro, M.: Redocumentation for the Maintenance of Software. In: Proceedings of the 30th Annual Southeast Regional Conference, pp. 413–416 (1992)