Big Data Deduplication in Data Lake

Jakub Hlavačka, Martin Bobák* and Ladislav Hluchý

Institute of Informatics, Slovak Academy of Sciences
Dúbravská cesta 9, 845 07 Bratislava, Slovakia
e-mails: xhlavackaj@is.stuba.sk martin.bobak@savba.sk,
ladislav.hluchy@savba.sk

*Corresponding author: M. Bobák, https://orcid.org/0000-0002-2905-4999

Abstract: Data lakes are the next generation of technology to process and store big data. As usual, new challenges and problems arise inevitably with new technologies. One of these problems is the occurrence of duplicate data in the storage. Our paper aims to address this challenge during the data ingestion phase that is currently overlooked or addressed insufficiently. The first part discusses the design of a suitable architecture for the data lake and deduplication workflow for processing structured and unstructured data. The proposed solution is evaluated through experiments that deal with the flexible deduplication window, the scalability of the proposed solution, the suitable hash function, and the advantages of an in-memory pointer repository.

Keywords: data lake; deduplication; big data

1 Introduction

The volume and variety of data are constantly increasing [1]. Most technologies that work with big data cannot adapt to this trend. Therefore, to solve these problems, the technology of data lakes [2] was created. Its main advantage is flexible work with data stored in low-budget storage [3]. Another benefit is minimal or no data processing before storing, which prevents the loss of data that may show potential value in the future.

Despite all the advantages, this technology has not yet perfected the work with big data. There are still open issues [4]. One of them is the occurrence of duplicate records in data lake storage [5]. Our work is focused on this problem. The proposed solution is based on the application of a deduplication process in the data ingestion phase when the data lake receives uploaded data.

2 Background and Related Work

2.1 Data Lakes

Every large organization uses a certain way of storing and processing data. In recent years, data warehouses have been used for these purposes, but data lakes [6] are being used more often. Data warehouses have become overwhelmed with data processing and storage due to the increasing volume and variety of data that need to be processed [7]. Since data warehouses process, clean, and then store the received data, it can lead to potential data loss. The raw data appeared to be useful over time because it could be used for other purposes [8]. It was necessary to react to emerging shortcomings of data warehouses, which is why the technology of data lakes was created. Modern **data lake** is a decentralized system (often exploiting cloud computing) that allows to store big data in its natural format.

Another important aspect is scalability. Cloud computing paved the way for scalable [9] and effective [10] infrastructures and platforms that can handle even big data stored in data warehouses or data lakes [11]. To obtain results from the data warehouses, the user must first understand their schema, structure, and quality. For this reason and to be able to store raw data, data lake works on the principle of schema-on-read.

There are several architecture types for data lakes. When the structure of the data stored in a data lake is known their analysis is easier. The most suitable architectural type is based on the principle of **data ponds** which divides the data in the data lake, according to their structure, into four basic ponds [8]. When the granularity of this approach is too low, **zone architecture** is more appropriate. Its main idea is the division of the data into several zones according to the stage of processing [12]. A special type of zone architecture is **lambda architecture**. It is divided into two zones: a batch processing zone for bulk data and a real-time processing zone for fast data (e.g. edge devices, IoT [13]) [14].

2.2 State of the Art

Data lake driven by data pipelines [15]: The data lake collects heterogeneous data from various sources. Its architecture is based on the principle of a data pipeline [16]. The data lake is designed to receive, process, store, analyze and visualize data. The whole process is divided into several phases. The first phase is custom data collection that takes advantage of multiple data extraction tools to collect data from sources such as web pages, application programming interfaces, or files in various formats. The data ingestion phase mainly gathers data from the data collection phase [17] carried out mainly by Apache Flume. It solves data loss problems that can occur during the collection of data. Thus, the data lake becomes reliable and fault

tolerant. The proposed approach supports manual data ingestion (the user uploads data directly into the data lake), as well as automatic data ingestion. The data are stored directly in the Apache Hadoop file system, which supports storing data in different formats. Afterward, the user can analyze the stored data by Apache Solr and Apache Spark. The last phase is data visualization using the Hue web interface together with Matplotlib.

Data lake lambda architecture [18]: Its architecture is divided into four layers: data collecting, data storing and processing, data querying and analytics. Since the data come from various sources and their format is not uniform, the first layer collects the data using Apache Flume. It is capable of handling data source diversity and data heterogeneity. Its main advantage is ensuring that data are saved in storage, even when disconnections or outages occur. Protection against data loss is solved by virtual memory channels. Each of these channels contains data that have been removed only after they are fully migrated to storage. Data storage and processing are performed by the batch layer and the speed layer. The batch sublayer uses Hadoop as a distributed file system and MapReduce to create previews of the stored data. The speed sublayer processes data in real-time. It fulfills the functionality of supplementing previews from the batch layer. The speed layer uses Apache Spark which can process data fast due to memory clustering. Previews from both layers are connected to the service layer. The data querying layer supports data extraction, loading, and aggregation. According to it, this layer is made up of several tools. The last layer dedicated to data analysis is carried out through various methods of artificial intelligence.

Data lake for archeological data [19]: The core of the data lake is divided into six layers. The whole data lake has 11 layers. Since the other layers are dedicated to resource and workflow management, data governance, or security, the analysis is focused on the core layers, which handle the whole data lifecycle within the data lake. The first layer selects a metadata model that best suits the received data. After collecting data properties and choosing the right metadata model, data ingestion takes place. Two different tools are used for data ingestion. Apache Sqoop is used to ingest structured data, and semi-structured and unstructured data are ingested by Apache Flume. The data quality after the data ingestion process is not always sufficient. The data often contains duplicate records along with other unwanted values. That is why data are polished by the data distillation layer which cleans data from duplicate records and missing or inappropriate values. The distillation layer can clean data that come directly from the data ingestion layer or are stored in the data lake. The data lake storage itself is located in the data storage layer. It is based on Hadoop, which can be easily replaced with Amazon S3. The data storage layer is directly connected to the data ingestion, distillation, and insights layers. The functionality of the last-mentioned layer is related to exploratory data analysis or data transformation. Data transformation provides data preparation for use in other data applications, which are part of the data application layer. Through the tools in this layer, the user can discover useful information from the data itself.

Serverless data lake [20]: The serverless paradigm [21] enables maximum use of shared resources with the lowest possible costs. All this can be achieved by turning off shared resources during idle time [22]. The data lake processes received data in batches, within pre-planned deadlines to exploit the serverless benefits as much as possible. Based on these assumptions, the proposed data processing consists of data extraction, task scheduling, data distribution, and data deduplication. The first step of the data lifecycle is to create tasks that are initialized by providing a configuration file to the tasks creator. It loads the credentials and sends requests to add the tasks to the tasks queue. In the next step, the tasks executor is emptying the tasks queue by fulfilling task requests and storing the received data in the landing zone. The tasks executor and tasks creator estimate the completion time of the tasks on limited system resources, so they are regulated by throttlers. The last step is to combine and deduplicate the data located in the landing zone. Tasks data partitioner and deduplicationer perform them according to tasks metadata. Unique data are stored in the persistent storage layer.

Encrypted data lake [23]: Since the classic methods of deduplication do not work, a new deduplication approach suitable for encrypted data in large storages (e.g., data lake) is proposed. The first step is keyword extraction from the uploaded data which are further analyzed by the multi-label unsupervised algorithm and used for data management. Once the data are uploaded to the server, the deduplication process is triggered. If the uniqueness check is negative, additional information is added to the existing data in the storage. It fulfills the functionality of a pointer on the data, and they are not uploaded to the data lake. In case of the opposite result, the data are encrypted and together with the deduplication token are stored in the data lake.

Data lake for the financial sector [24]: Part of the data lake platform is a data warehouse. Before uploaded data are stored in the data warehouse, they are processed through extraction, transformation, and loading, followed by a deduplication process. The deduplication pipeline consists of several steps. In the first step, the data received by the data channel are divided into smaller blocks that may represent duplicate records. The goal of the division is to reduce the data. The blocks are compared to each other and clustered according to their degree of similarity. After clustering, groups of similar blocks are merged. The core component is the Cloud Data Repository, which follows the data lake paradigm. Its main task is to receive data from heterogeneous internal and external sources. The extraction, transformation, and loading processes are applied later to the data in the data lake, after which they are directed to the data warehouse storage, which is an internal component of the data lake.

3 Design of Deduplication Process for Data Lake

3.1 Data Lake Architecture

The proposed data lake architecture is depicted in Figure 1. The data lake consists of data ingestion and data storage. The other parts of the data lake are not considered, while they are not a part of our research.

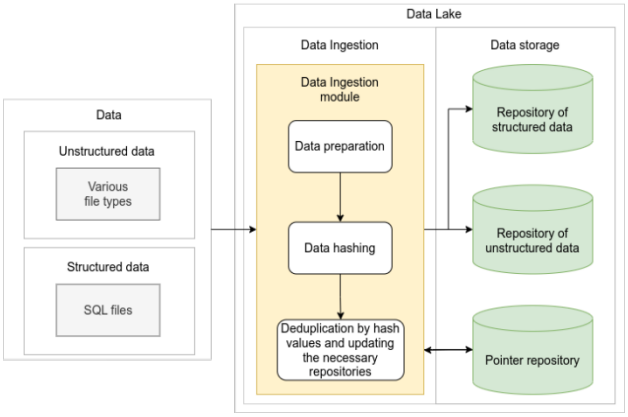

Figure 1
Data Lake Architecture (the relevant parts)

The proposed data lake can receive both unstructured and structured data. Unstructured data are not restricted. On the other hand, structured data are restricted to the SQL format. However, it is not a significant limitation of the proposed approach since receiving data in other formats can be supported through an extension using the API within the data lake. The uploaded data are handled by a dedicated data ingestion module, which is the main and only component of the data ingestion phase. The module not only serves to receive diverse types of data and ensures that the data lake receives complete data, but our approach also uses it to deduplicate uploaded data before saving them to the data lake. The proposed data **ingestion module** consists of the following parts:

- The first part of the deduplication process is **data preparation**. It depends on the data format. In the case of unstructured data, it divides the input data into smaller blocks (chunks). On the other hand, in the case of structured data, individual insertion queries are identified.
- After data preparation, **hash** values are calculated from the outputs of the previous step. In the case of structured data, hash values are calculated from data input queries, and in the case of unstructured data, hash values

are calculated from data chunks that were created from uploaded unstructured files in the first part of the deduplication process.

- The last part is **deduplication according to the hash values and updating the necessary repositories**. The data are handled iteratively, while the last chunk does not have to process its hash value. Initially, data chunks are marked as unique or duplicates based on their hash values. This requires access to a pointer repository, which stores data as key pairs. The key is the hash value of a data chunk already stored in the data lake, and the value is a pointer to that chunk (see Sections 3.3 and 3.4).

Data storage is divided into three repositories that store data in the data lake. One particularly important repository is the **pointer repository**. According to the state of the art, there is no data lake similarly designed. Thus, its impact on the data lake is evaluated in the experiment section. Other data storages are the **unstructured data repository** and the **structured data repository**. As the names suggest, the unstructured data repository contains data chunks together with its metadata, and the structured data repository contains structured data in tabular form.

The proposed methodology for deduplication is based on hashing, which is a very suitable approach for this purpose. However, several critical aspects have to be considered when applying this technique. The most significant is collision handling. While this paper concentrates on data deduplication in data lakes rather than the hash functions as the concept, it operates under the assumption of an ideal hash function that ensures no collisions occur.

3.2 Deduplication Workflow

The whole deduplication workflow is depicted in Figure 2. It starts with uploading data to the data lake. They are received by a specialized data ingestion module (see Section 3.1), which starts processing the uploaded data. The module distinguishes whether the user has uploaded structured or unstructured data because it affects how the data are treated in the next parts of the workflow.

For structured data, the ingestion module identifies all insertion queries of the uploaded SQL file. Subsequently, it calculates hashes for each data from the insertion queries. In the case of unstructured data, the ingestion module divides them into smaller chunks, which hashes are calculated. After the end of this cycle or, in the case of structured data, after the calculation of the hash value of the last identified data insertion query, data deduplication begins with the processing of the hash values, and the relevant repositories are updated according to the deduplication results.

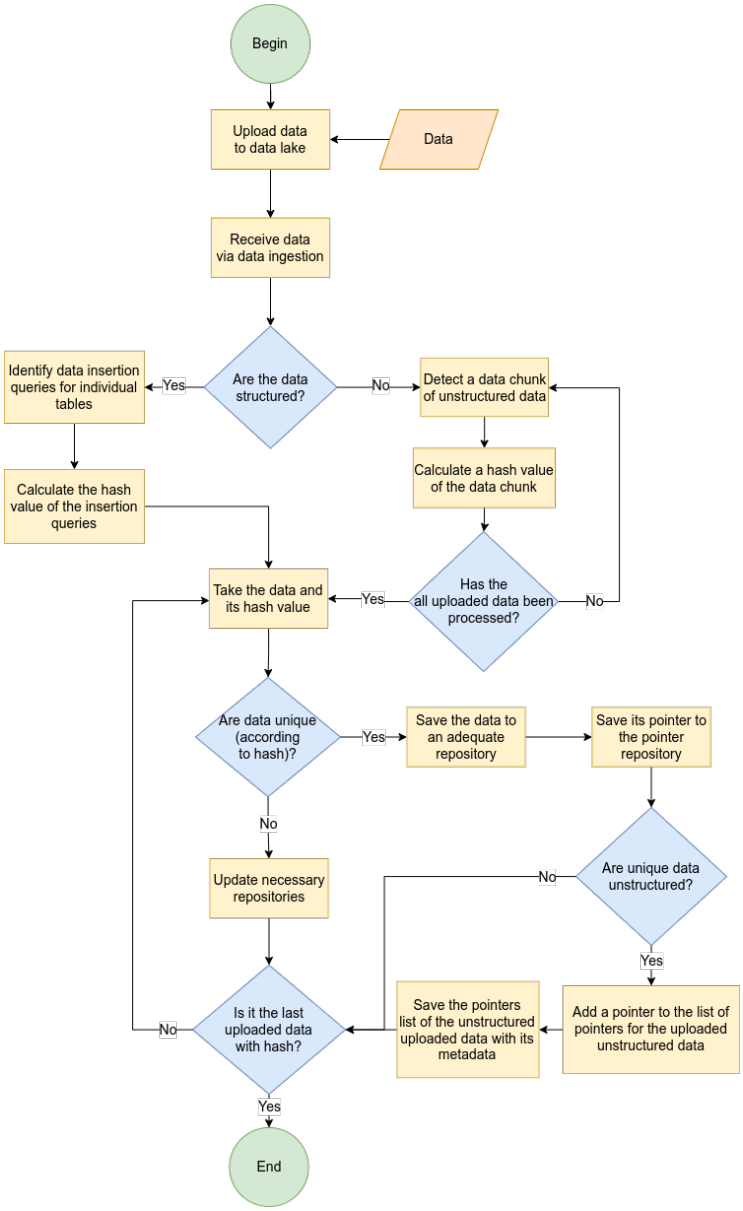

Figure 2
General workflow for big data deduplication within data lake

After successfully storing the unique data, it is necessary to save its pointer in the pointer repository. In the case of unstructured data, the pointer on a unique data chunk is added to the list of pointers for the entire unstructured data (i.e., the file)

that the user initially uploaded. This list of the uploaded unstructured data is then stored, together with its metadata, in the unstructured data repository.

If the ingestion module marked the uploaded data as duplicate, the necessary repositories are updated. This update occurs only under the following conditions:

1) A duplicate data chunk of uploaded unstructured data has the same hash as a data chunk of other unstructured data. In this case, the metadata of the duplicate data chunk is updated, and the updated list of pointers for the uploaded unstructured data is saved together with its metadata.
2) The unstructured data uploaded by the user is composed of several data chunks which are used more than once in the uploaded unstructured data. Firstly, the data ingestion module processed it as unique, but since it is in the data lake, it becomes a duplicate. However, because it is used in a different place in the unstructured data, it is necessary to add this information to the metadata of the data chunk in the unstructured repository. The list of pointers is also updated for the uploaded unstructured data, as well as its metadata.

Regardless of whether the data have been identified as unique or duplicated and subsequently processed adequately, the deduplication process and updating of the repositories continue until all uploaded data (i.e., its insertion queries or data chunks) have been processed.

3.3 Structured Data Processing

Structured data can be handled as tabular data. According to it, they can be stored in databases with a predefined schema [25]. The advantage of this approach is quick and simple analysis. Unfortunately, data lakes cannot define the data schema for uploaded data in advance [26], which complicates the deduplication process in general.

Structured data are uploaded to the data lake as an SQL file that is processed by the ingestion module. The module divides individual queries into several categories according to their functionality during query extraction. From a data lake perspective, the most interesting are insertion queries. However, insertion queries can also contain duplicate data. To identify duplicates, the ingestion module calculates a hash for each insertion query.

In the case of unique data, its insertion query is executed, which stores the new data in the structured data repository. Subsequently, a new pointer is stored in the pointer repository (in this case, the table name changed by the query) together with the hash of the query. This type of data is stored in the *key:value* pair, where *key* is the hash value of the query, and the *value* is the table name over which the insertion query is executed. In the case of a duplicate request, none of these operations are performed.

3.4 Unstructured Data Processing

To solve the deduplication problem of unstructured data, it is useful to detect duplicate data at the level of smaller regions because the approach is more effective. The first step is to divide the uploaded data (i.e., files) into data chunks and calculate their hashes, based on which the data ingestion module determines whether it is a unique or duplicate data chunk.

In the case of a unique data chunk, the ingestion module stores it in the unstructured data repository. Subsequently, the *key:value* pair is stored in the pointer repository. *Key* is the hash value of the data chunk, and *value* is a pointer to this chunk in the unstructured data repository. The last step is to update the information about its usage, which plays an important role in the reuse of data chunks in multiple unstructured data. In the case of a duplicate data chunk, the metadata in the unstructured data repository is updated, because it has to add another reference to a different data on its pointer.

4 Platform for Data Lake Deduplication

Since the paper deals with big data in data lakes, it is very important to choose a suitable tool for the deduplication process. The deduplication is performed during the data ingestion phase, which reduces a set of all existing tools to a set of tools that are designed for data ingestion. Majority of existing data lakes [19] [15] [18] use Apache Flume which was originally designed to ingest heterogeneous data in Hadoop Distributed File System (HDFS) [27] [28] [29], according to which it is necessary to find a different tool since not all data lake repositories use HDFS. Apache Flume alternatives are Apache NiFi and Apache Kafka [18], according to which the module is built on Apache Kafka [28] because Apache NiFi was designed to stream data from one system to another [30]. Given that the data ingestion module receives files via API, the choice of Apache Kafka over Apache NiFi is clear. Apache Kafka is a distributed and scalable tool that allows sending messages with low latency and high throughput. It was created to process log messages [28], but is currently also used in systems that require real-time data processing [31]. The module uses this tool for data ingestion, in which data deduplication is implemented.

However, Apache Kafka does not support data deduplication, and thus the module had to use external libraries through which it becomes a part of data ingestion. As part of the deduplication process, it is necessary to solve the division of unstructured data into data chunks and the calculation of hash values which are computed as SHA256 hashes.

Dividing unstructured data into data chunks is implemented by the FastCDC library, which uses the method of dividing data into smaller blocks of different lengths (content-defined-chunking). The advantage of its algorithm is that it divides the data several times faster than the tools that use the Asymmetric Extremumu algorithm or the Rabin algorithm [32].

Apache Kafka manages messages by publishing and subscribing. The component that publishes messages is called a producer, and the component that subscribes these messages is called a consumer. The consumer does not need to immediately subscribe to the messages that the producer has published, so these messages are stored in a component named topic. Individual topics are filled with messages of a certain type. The module uses two different topics. One of them is filled with messages containing structured data and the other with messages containing unstructured data. These topics are part of the broker component, which behaves as a server and is coordinated by Zookeeper.

Messages in the topics are stored in a byte array in the Avro format. Operations over Avro schemes are handled by the schema registry component. The producer and the consumer communicate with the schema register.

4.1 Data Repositories

The data ingestion module has three different data repositories which use the following technologies. The **structured data repository** is based on PostgreSQL because the structured data are deduplicated at the query level. Thus, it is necessary to select a technology supporting query-driven data management, i.e. storing structured data in tabular form, allowing the creation of relationships between stored data and inserted data via insertion queries.

The **unstructured data repository** is based on an open-source technology MinIO, which is an object storage designed for cloud usage. It is chosen as an alternative to HDFS because other technologies are outdated, have limited performance, and scale poorly [33].

- In MinIO, unstructured data is stored in buckets. The data ingestion module uses a *files-bytes* bucket and a *files-pointers* bucket. The *files-bytes* bucket contains records in JSON format. Their content consists of attributes *used_in_files* and *data*. The *data* attribute contains a data chunk (as a string) to which this record in the *files-bytes* bucket belongs. The names of the records in the *files-bytes* bucket are the hash values of the data chunks stored in the *data* attribute.
- The *used_in_files* attribute is more complex than the *data* attribute. It contains information about the occurrence of the data chunks in different files that have been uploaded to the data lake.

- The information consists of the pair *key:value*. *Key* in this case is the name of the file that contains the data chunk. *Value* consists of two attributes. The attribute named *occurrences* contains a number that represents information about the number of data chunk occurrences in the file with the name stored in the *key*. The attribute *at_indexes* is an array of numbers through which it is possible to correctly reconstruct a file whose content consists of multiple data chunks. The array contains the order of the data chunk within the content of the file. Since a data chunk can be used more than once in a file, the *at_indexes* attribute is implemented as an array of digits.

The **pointer repository** consists of records that contain lists of pointers to files that were previously uploaded to the data lake. The records are in JSON format, and their name contains the original file name. The record has two attributes: *pointers* and *original_file_name*. The *pointer* attribute contains a list of pointers to the data chunks of the original file. Each pointer has a *bucket* attribute and a *chunk_hash* attribute. The *bucket* attribute carries information about the bucket in which the record of the data chunk is stored. The *chunk_hash* attribute contains the hash value of the data chunk so that the data lake can find the record of that data chunk in the bucket where it is stored.

- To identify whether a user uploads unique or duplicate data, it is necessary to find out if the hash of the data already exists in the pointer repository. This means that for every deduplication check, the module has to query this repository. According to it, the repository needs to respond as fast as possible to queries providing information about the occurrence of the searched data. Thus, the pointer repository is based on Redis. Searching for data in this technology is faster than in classic databases operating over the disk because it works as an in-memory database.
- Redis database stores data in the format *key:value*. *Key* contains the hash value of the unique data, and the content of the *value* is a pointer to it. The hash value has the same form whether it belongs to structured or unstructured data. On the other hand, the pointer that is stored in a *value* always has a JSON format, but its content and structure depend on the data type. In the case of structured data, it is very simple but sufficient. It only contains the *table_name* attribute with the name of the table in which the structured data is stored. A pointer to unstructured data contains the *bucket* and *chunk_hash*. The *bucket* attribute carries information about the bucket in which the unstructured data is stored within the MinIO repository. To be able to access the given unstructured data, it is necessary to know the name of the record in which it is stored within the bucket, which is the content of the *chunk_hash* attribute.

4.2 Data Ingestion

Figure 3 shows the architecture of the data ingestion module for data lake deduplication. The whole solution is dockerized, which enables scaling individual components as needed, and at the same time, dockerization simplifies module launch in different environments.

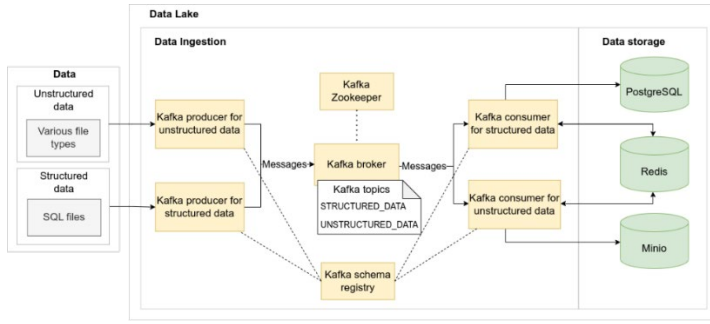

Figure 3
The architecture of the data ingestion module

The whole data ingestion process is based on Apache Kafka, which prevents data loss in the data lake. The data are uploaded to the data lake through application programming interfaces which are available to the structured data producer and the unstructured data producer. The uploaded data are subsequently processed and published by the relevant producer. The published data are further consumed by an adequate consumer, which performs the appropriate operations, according to the results of the deduplication process. If the uploaded data are unique, then they are forwarded to the PostgreSQL storage repository (structured data) or the MINIO storage repository (unstructured data). Together with these data, their pointers are also stored in the Redis database as a key:value pair, where the key is the hash of the stored data.

5 Experiments and Evaluation

This section presents the experiments through which the proposed solution is evaluated. The following experiment aspects are considered:

- **Ideal size of a flexible window** - the experiment deals with the hypothesis of whether there is a bottleneck when a small flexible window is used.
- **Scalability of uploaded files** - the experiment investigates if the total time of data ingestion grows linearly with the increasing size of uploaded files.

- **Hash functions** – the experiment is focused on trade-off simple hash functions.
- **Advantages of in-memory pointer repository** - the experiment examines if the total time of published messages processing is faster when the pointer repository is based on an in-memory database instead of a disk database.

5.1 Experiment Environment

The experiments are evaluated on a virtual machine with the Linux operating system (Ubuntu 20.04.5 LTS). This virtual machine had 8 CPU cores Intel(R) Xeon(R) model X5570 @ 2.93GHz with x86_64 architecture. The size of the SSD disk of the virtual machine was 40GB, and its RAM has 16 GB.

The virtual machine had installed Docker and docker-compose along with all required dependencies. The docker version is 23.0.1 and the docker-compose version is 1.25.0. The experiments are dockerized and run within the IISAS scientific cloud. Figure 4 shows a deployment diagram of the experiment environment. The only externally accessible point is MinIO GUI, through which the individual experiments are monitored. This interface is available on port 80 using NGINX.

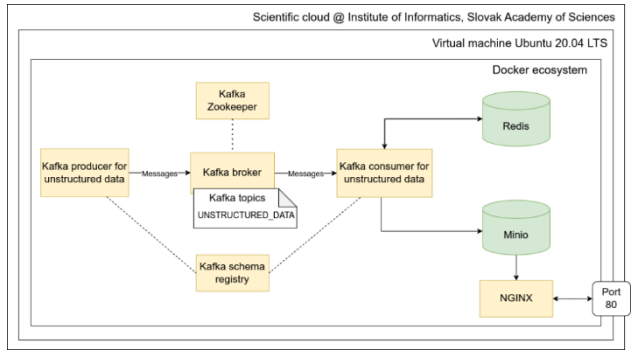

Figure 4
Deployment diagram

5.2 Ideal Size of Flexible Window

The experiment aims to find out whether there exists a bottleneck related to the size of a flexible window determining the size of data chunks.

The dataset of this experiment consists of 100 000 files with a size of 4000 bytes. The file content is generated randomly by our proprietary generator. The experiment evaluates several flexible windows, and its results are shown in Figure 5.

The experiment also shows that Apache Kafka is the main bottleneck in terms of message processing and consumption. This bottleneck grows in direct proportion to the size of the flexible window.

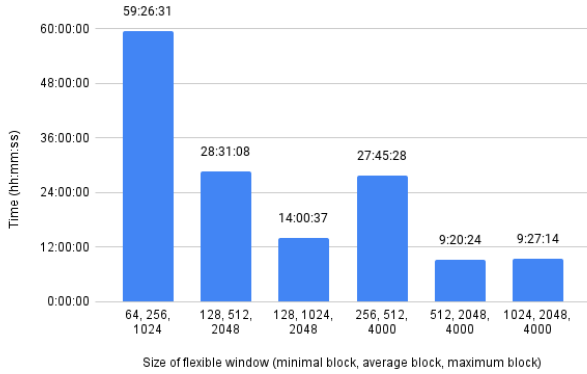

Figure 5
Consumption time per individual size of the flexible window

Part of the experiment was the evaluation of deduplication process sensitivity, chunking time, production time, and repository sizes. It is focused on the size of data chunks. The whole chunking process works with a flexible window, which is defined by its minimum, average, and maximum size. The ratio between these values is 4. The exception are configurations reaching value 4000, which is the size of the whole file. The paper presents them for demonstration purposes. The configuration 512, 2048, and 4000 has the best time, but it is affected by the experiment configuration (the file size is 4000 bytes). Within these aspects, the configuration 128, 1024, and 2048[1] is dominating, according to that it is considered the ideal size of the flexible window.

5.3 Scalability of Uploaded Files

This experiment scenario explores how uploading an increasing number of unique files affects the deduplication process. The experiment starts with 100 000 unique files and ends with 300 000 unique files. The size of a file is 4000 bytes. Figure 6 shows that the consumption time increases linearly with the number of uploaded files.

[1] These values determine the minimum, average, and maximum size of the data chunks.

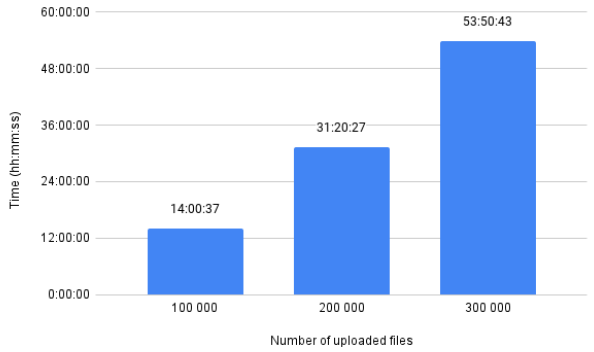

Figure 6
Consumption time per different number of unique files

The part of the experiment was also how the deduplication process depends on the size of the uploaded file. There were three datasets composed of 4000 bytes, 8000 bytes, and 12000 bytes. The results are similar. According to it, the achieved results can be generalized to that the deduplication process has linear space complexity.

5.4 Hash Functions

The proposed deduplication approach for the data lake is based on the SHA256 hash function. The design considered SHA-family hash functions, which is the most popular hashing algorithm nowadays [34]. There are also several other reasons for this decision, however, the most important aspect is its tradeoff between breakability and time complexity. Since SHA256 is not a simple hash function [35], this experiment tries to find out how much the use of computationally simpler hashing functions can speed up the overall data ingestion process in our data lake.

During this experiment, 100 000 unique randomly generated files of size 4000 bytes were uploaded to the data lake. These files were uploaded in three different runs, each run using a different hash function in the deduplication process.

As Figure 7 shows, there is a speed-up when a simpler hash function is used. However, the improvement is not significant enough to outweigh the risk of a hashing collision. The stronger hash functions make the deduplication process more time-consuming. On the other hand, MD5 weaker hash function does not represent a significant improvement in the context of the deduplication process. According to it, weaker hash functions are not considered in the experiment.

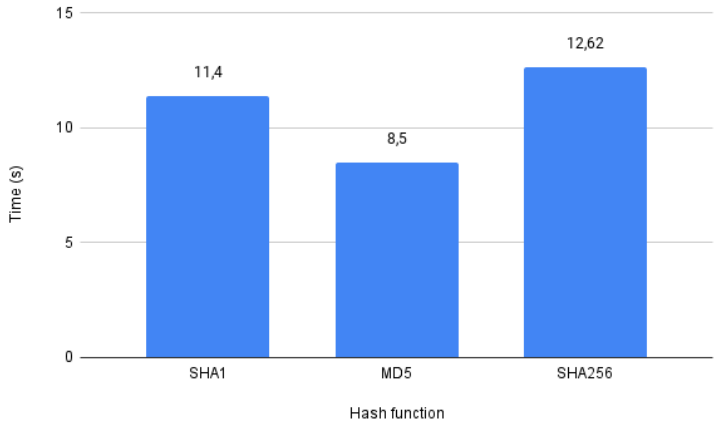

Figure 7
Total chunking time for the individual hash functions

5.5 Advantages of In-Memory Pointer Repository

The proposed approach uses the in-memory pointer repository. Thus, this experiment compares an in-memory database (Redis) with a disk database (HBase). The experiment examines whether the in-memory approach is more suitable because disk databases have to read each data chunk from the disk, which is a time-consuming operation.

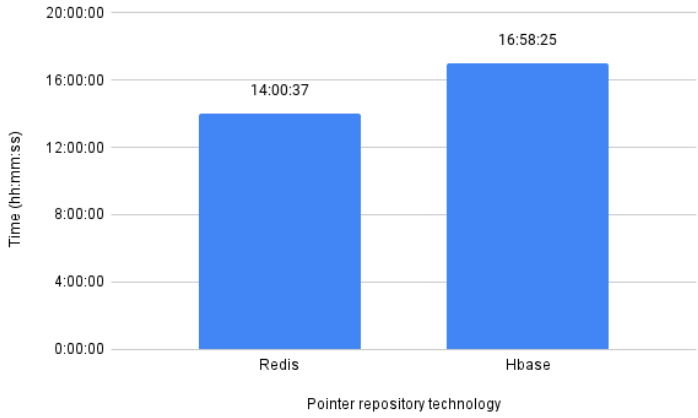

Figure 8
Total time consumed by different pointer repository implementations

The experiment uses a dataset consisting of 100 000 files with a size of 4000 bytes. Figure 8 shows that the total consumption time is shorter in the case of the in-memory database.

5.6 Comparison with State of the Art Solutions

The last part of the evaluation is a comparison of the proposed approach with existing solutions (see Section 2.2). The typical characteristic is to divide the data lake into several layers or zones which also adapts the proposed solution working with data ingestion zone and storage zone.

The deduplication approach of the proposed solution can handle structured and unstructured data regardless of their origin, size, or schema. Deduplication is part of the data ingestion phase, which is one of the studied innovations of the proposed solution. The evaluation primarily compares the solution parameters with related works due to the lack of qualitative assessments (see Table 1).

Table 1
Comparison overview

Deduplication approach	Deduplication position	Data storage
Deduplication is a part of data cleaning that is carried out by custom-made programs.	Unknown.	Based on Hadoop.
None.	None.	Based on Hadoop.
Deduplication is a part of the data cleaning performed by custom-made programs.	Distillation layer.	Based on Hadoop.
Based on the metadata of received data.	After the landing zone.	Located at the landing zone and persistent storage zone.
Unspecified.	After uploading the data to the server (not exactly defined).	Cloud storage without further specification
Chunking-driven approach.	In the data channel, which is part of the data warehouse.	Data warehouse and cloud storage without further specification
Based on hash values of data chunks.	Part of data ingestion.	MINIO - unstructured data PostgreSQL - structured data Redis - pointer storage.

Data ingestion	Based on Apache Flume, which receives data from a data source or a custom data collection layer.	Based on Apache Flume.	Unstructured data - Apache Flume. Structured data – Apache Sqoop.	Driven by chunking approach with unknown technology.	Unknown.	Unknown.	Based on Apache Kafka using custom data chunk workflows.
	Data lake driven by data pipelines [15]:	Data lake lambda architecture [18]:	Data lake for archaeological data [19]:	Serverless data lake [20]:	Encrypted data lake [23]:	Data lake for the financial sector [24]:	Proposed approach

Conclusions

The paper focuses on the deduplication of big data in the data ingestion phase, during which a data lake receives uploaded data. The analysis of related work confirmed that duplicate records in the data lake are one of the current open problems.

According to it, data lake architecture is specified. Subsequently, a deduplication workflow is designed. Through it, the proposed data lake ingests, deduplicates, and stores uploaded data. Given that one of the requirements of a data lake is the ability to process all data, regardless of its form, the design deals with the processing of structured and unstructured data.

The proposed solution is evaluated within several experiments that focus on various aspects of big data deduplication in the data lake. The first experiment shows how window size affects the deduplication process. The next experiment evaluates the scalability of the proposed solution. The third experiment examines whether a simple hash function is worth the collision risk. The last experiment focuses on the advantages of an in-memory pointer repository.

Acknowledgment

This publication is the result of the project implementation: Research on the application of artificial intelligence tools in the analysis and classification of hyperspectral sensing data (ITMS: NFP313011BWC9) supported by the Operational Programme Integrated Infrastructure (OPII) funded by the ERDF. This work was supported by the project AI4EOSC “Artificial Intelligence for the European Open Science Cloud” that has received funding from the European Union’s Horizon Europe Research and Innovation Programme under Grant agreement no. 101058593, by the project iMagine “Imaging data and services for

aquatic science" that has received funding from the European Union's Horizon Europe Research and Innovation Programme under Grant agreement no. 101058625, and by the project SILVANUS "Integrated Technological and Information Platform for wildfire Management" that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 101037247. It was also supported by APVV grant no. APVV-21-0448 and VEGA grant no. 2/0131/23.

References

[1] T. Hukkeri, V. Kanoria and J. Shetty, "A study of enterprise data lake solutions," *International Research Journal of Engineering and Technology (IRJET),* vol. 7, no. 5, pp. 1924-1929, 2020.

[2] R. Hai, C. Koutras, C. Quix and M. Jarke, "Data Lakes: A Survey of Functions and Systems," *IEEE Transactions on Knowledge and Data Engineering,* pp. 1-20, 2023.

[3] C. Giebler, C. Gröger, E. Hoos, H. Schwarz and B. Mitschang, "Leveraging the Data Lake: Current State and Challenges," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics),* vol. 11708 LNCS, pp. 179-188, 2019.

[4] A. Cuzzocrea, "Big data lakes: models, frameworks, and techniques," *IEEE International Conference on Big Data and Smart Computing,* pp. 1-4, 2021.

[5] M. El Ghazouani, M. El Kiram, L. Er-Rajy and Y. El Khanboubi, "Efficient method based on blockchain ensuring data integrity auditing with deduplication in cloud," *International Journal of Interactive Multimedia and Artificial Intelligence,* vol. 6, pp. 3-32, 2020.

[6] E. Zagan and M. Danubianu, "Data lake approaches: A survey," *2020 International Conference on Development and Application Systems (DAS),* pp. 189-193.

[7] J. Meizner, P. Nowakowski, J. Kapala, P. Wojtowicz, M. Bubak, V. Tran, M. Bobák and M. Höb, "Towards exascale computing architecture and its prototype: Services and infrastructure," *Computing and Informatics,* vol. 39, no. 4, pp. 860-880, 2020.

[8] P. Sawadogo and J. Darmont, "On data lake architectures and metadata management," *Journal of Intelligent Information Systems,* vol. 56, no. 1, pp. 97-120, 2 2021.

[9] M. Bobak, L. Hluchy and V. Tran, "Abstract model of k-cloud computing," *2014 11th International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2014,* pp. 710-714, 2014.

[10] M. Bobak, L. Hluchy and V. Tran, "Methodology for intercloud multicriteria optimization," *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2015,* pp. 1786-1791, 2016.

[11] M. Bobák, L. Hluchý, O. Habala, V. Tran, R. Cushing, O. Valkering, A. Belloum, M. Graziani, H. Müller, S. Madougou and J. Maassen, "Reference exascale architecture (extended version)," *Computing and Informatics,* vol. 39, no. 4, pp. 644-677, 2020.

[12] C. Giebler, C. Groger, E. Hoos, H. Schwarz and B. Mitschang, "A zone reference model for enterprise-grade data lake management," *IEEE24th International Enterprise Distributed Object Computing Conference (EDOC),* pp. 57-66, 2020.

[13] Y. Zhao, I. Megdiche, F. Ravat and V.-n. Dang, "A Zone-Based Data Lake Architecture for IoT, Small and Big Data," *Proceedings of the 25th International Database Engineering \& Applications Symposium,* pp. 94-102, 7 2021.

[14] F. Cerezo, C. E. Cuesta, J. C. Moreno-Herranz and B. Vela, "Deconstructing the Lambda architecture: an experience report," *IEEE International Conference on Software Architecture Companion (ICSA-C),* pp. 196-201, 2019.

[15] H. Mehmood, E. Gilman, M. Cortes, P. Kostakos, A. Byrne, K. Valta, S. Tekes and J. Riekki, "Implementing big data lake for heterogeneous data sources," *Proceedings - 2019 IEEE 35th International Conference on Data Engineering Workshops, ICDEW 2019,* p. 37–44, 2019.

[16] A. R. Munappy, J. Bosch and H. H. Olsson, "Data Pipeline Management in Practice: Challenges and Opportunities," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics),* pp. 168-184, 2020.

[17] Y. Zhao, I. Megdiche and F. Ravat, "Data Lake Ingestion Management," *arXiv preprint arXiv:2107.02885,* 7 2021.

[18] A. A. Munshi and Y. A. R. I. Mohamed, "Data Lake Lambda Architecture for Smart Grids Big Data Analytics," *IEEE Access,* vol. 6, p. 40463–40471, 2018.

[19] P. Liu, S. Loudcher, J. Darmont and C. Noûs, "ArchaeoDAL: A Data Lake for Archaeological Data Management and Analytics," *ACM Proceedings of*

the 25th International Database Engineering Applications Symposium, p. 252–262, 2021.

[20] A. Bryzgalov and S. Stupnikov, "A Cloud-Native Serverless Approach for Implementation of Batch Extract-Load Processes in Data Lakes," *Communications in Computer and Information Science,* vol. 1427, pp. 27-42, 2021.

[21] G. McGrath and P. R. Brenner, "Serverless Computing: Design, Implementation, and Performance," *Proceedings - IEEE 37th International Conference on Distributed Computing Systems Workshops, ICDCSW 2017,* pp. 405-410, 2017.

[22] M. I. Malik, S. H. Wani and A. Rashid, "Cloud computing technologies," *International Journal of Advanced Research in Computer Science,* vol. 9, no. 2, pp. 379-384, 2018.

[23] A. Ashmita and R. Anitha, "Data De-Duplication on Encrypted Data Lake in Cloud Environment," *International Journal of Engineering Research & Technology (IJERT),* vol. 7, no. 3, pp. 2278-0181, 2018.

[24] M. Sienkiewicz, R. W. -. E. Workshops and u. 2021, "Managing Data in a Big Financial Institution: Conclusions from a R&D Project.," *EDBT/ICDT Workshops,* vol. 2841, 2021.

[25] X. . Yang, C. M. Procopiuc and D. . Srivastava, "Summarizing relational databases," *Proceedings of The Vldb Endowment,* vol. 2, no. 1, pp. 634-645, 2009.

[26] A. Tunjic, "The automation of the data lake ingestion process from various sources," *42nd International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2019 - Proceedings,* pp. 1276-1281, 2019.

[27] D. Borthakur, "HDFS architecture guide," *Hadoop Apache Project,* vol. 53, no. 2, pp. 1-13, 2008.

[28] J. Kreps, N. Narkhede and J. Rao, "Kafka: a Distributed Messaging System for Log Processing," *ACM SIGMOD Workshop on Networking Meets Databases,* p. 1–7, 2011.

[29] D. Vohra, "Apache Flume," in *Practical Hadoop ecosystem*, Springer, 2016, pp. 287--300.

[30] K. Racka, "Apache Nifi As A Tool For Stream Processing Of Measurement Data.," *Nauki Ekonomiczne,* pp. 115-135, 2022.

[31] R. Shree, T. Choudhury, S. C. Gupta and P. Kumar, "KAFKA: The modern platform for data management and analysis in big data domain," *2nd International Conference on Telecommunication and Networks, IEEE TEL-NET 2017,* pp. 1-5, 2017.

[32] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Y. Zhang and Q. Liu, "FastCDC: A fast and efficient content-defined chunking approach for data deduplication," *Proceedings of the 2016 USENIX Annual Technical Conference, USENIX ATC 2016,* pp. 101-114, 2016.

[33] MinIO, "High Performance Multi-Cloud Object Storage," [Online]. Available: https://min.io/resources/docs/MinIO-High-Performance-Multi-Cloud-Object-Storage.pdf. [Accessed 1 March 2023].

[34] P. Rathod, R. Sakhiya, R. Shah and S. Mehta, "Meta-Analysis of Popular Encryption and Hashing Algorithms," in *International Conference on ICT for Sustainable Development*, 2023.

[35] S. Aggarwal and N. Kumar, "Hashes," *Advances in Computers,* vol. 121, pp. 83-93, 1 2021.