

MINISTRY OF SCIENCE AND HIGHER EDUCATION OF THE RUSSIAN FEDERATION
FEDERAL STATE AUTONOMOUS EDUCATIONAL
INSTITUTION OF HIGHER EDUCATION
NORTH CAUCASIAN FEDERAL UNIVERSITY

GUIDELINES
for the completion of practical work
By discipline "Big Data Analysis and Mining / Анализ и обработка больших данных"
for students in the field of study
02.04.01 Mathematics and Computer Science
Directionality (profile)
"Training Program for Electronic Information"

Content

Introduction

Practical Work #1: Deploying a Training Environment for Big Data

Practical Work #2 Working with HDFS: Big Data File System

Practical Work #3: Analysis of the Data Storage Structure in HDFS

Practical Work #4 Classic MapReduce Model

Practical Work #5: Processing Structured Data with Hadoop

Hands-on Work #6: Introduction to Apache Spark and PySpark

RDD Workshop #7: Basic Transformations and Actions

Hands-on Exercise #8: Spark DataFrame API

Practical Work #9: Spark SQL and Analytical Queries

Practical Work #10: Data Storage Formats: CSV, JSON, Parquet

Hands-on Exercise #11: Joins, Groupings, and Window Functions in Spark

Practical Work #12 Optimizing Spark Applications

Practical Work #13: Building a Batch Data Processing Pipeline

Practical work #14 Structured Streaming in Apache Spark

Practical Work #15: Data Mining on Spark: Classification and Model Quality Assessment

Practical Work #16: Data Mining on Spark: Clustering and Association Rule Search

Conclusion

Educational, methodological and informational support of the discipline

INTRODUCTION

1. The purpose and objectives of mastering the discipline (module)

The purpose of the discipline is to develop, in accordance with the educational program, the competencies of OPK-2, PC-6 of the student.

The goal of mastering the discipline "Big Data Analysis and Mining / Анализ и обработка больших данных" is to develop students' systematic theoretical knowledge and practical skills in the field of storage, distributed processing, analysis and intelligent extraction of patterns from big data using modern technologies Hadoop, Apache Spark, PySpark, Spark SQL and Spark MLlib.

To achieve this goal, the following tasks are solved in the process of studying the discipline: familiarization with the basic concepts of big data, their characteristics, sources, formats and areas of application; study of architectural approaches to the construction of big data processing systems, including data lake, lakehouse, batch, and streaming architectures; mastering the principles of distributed data storage in HDFS, replication mechanisms, block storage structure, and features of working with large files; studying the classical model of distributed processing MapReduce, the map, shuffle, and reduce stages, as well as the limitations of this approach; forming an understanding of the Apache Spark architecture, the role of drivers, executors, jobs, stages, tasks, and the DAG model for performing calculations; mastering basic Spark abstractions, including RDD, DataFrame, Spark SQL, and Dataset; studying methods for loading, cleaning, filtering, aggregating, joining, and transforming large data sets; mastering techniques for working with columnar data storage formats, including Parquet and ORC; studying methods for optimizing distributed data processing, including partitioning, caching, broadcast join, shuffle analysis, and data skew detection; mastering the principles of building reproducible batch pipelines with raw, cleaned, and curated layers; familiarization with the basics of stream data processing, window analytics, and event time, processing time, watermark and checkpointing in Spark Structured Streaming; study of methods of intelligent analysis of big data, including classification, regression, clustering, search for association rules, anomaly detection and construction of recommender models; mastering Spark MLlib tools for preparing features, building ML pipelines, training models and assessing their quality; developing skills in choosing mathematical models and algorithms for data analysis taking into account the volume, structure, quality and dynamics of the source data; development of practical skills in interpreting data mining results, assessing model limitations, identifying overfitting, class imbalance, data leakage and data drift; developing the ability to justify the choice of tools, algorithms and architectural solutions for big data analysis tasks taking into account the requirements of scalability, reproducibility, fault tolerance and computational efficiency.

2. The place of the discipline (module) in the structure of the educational program

The discipline is a compulsory part.

3. List of planned learning outcomes for the discipline (module), correlated with the planned learning outcomes for the educational program

Code, formulation of competence	Code, indicator formulation	Planned learning outcomes for the discipline (module), characterizing the stages of formation of competencies, indicators
---------------------------------	-----------------------------	---

OPK- 2 Able to create and explore new mathematical models in the natural sciences, improve and develop concepts, theories and methods	ID-1 OPK-2 Conducts research into new mathematical models in the natural sciences	Conducting research into new mathematical models applied to big data analysis tasks; formulating a research problem based on the structure, volume, quality, and characteristics of the source data; selecting mathematical models for solving problems of classification, regression, clustering, searching for association rules, detecting anomalies, and analyzing streaming data; studying the properties of data mining models taking into account the requirements of scalability, stability, interpretability, and reproducibility; assessing the quality of mathematical models based on relevant metrics; comparing alternative models and methods for big data analysis; identifying the limitations of the models used when working with large, heterogeneous, and dynamically changing datasets; formulating substantiated conclusions on the applicability of the studied mathematical models to big data analysis and mining tasks.
	ID-2 OPK-2 Develops new and improves existing concepts, theories and methods	Developing new and improving existing concepts, theories, and methods applied to big data analysis; analyzing modern approaches to distributed storage, processing, and intellectual analysis of big data; identifying the limitations of existing methods of classification, regression, clustering, association rule mining, anomaly detection, and stream data analysis;

		improving methods for preparing, transforming, and selecting features, taking into account the structure, volume, quality, and dynamics of the source data; developing and justifying options for modifying algorithms and models to improve the quality, scalability, stability, and reproducibility of analysis results; comparing alternative concepts and methods of big data analysis and mining based on objective criteria of quality and computational efficiency; formulating proposals for adapting existing methods to large, heterogeneous, and dynamically changing datasets; substantiating the applicability of developed or improved approaches to solving research and applied problems of big data analysis.
PC-6. Able to use modern methods for developing and implementing specific algorithms for mathematical models based on programming languages and application packages for modeling.	ID-1 PC-6 Using modern packages of applied modeling programs, it explores newly created	Study of newly created mathematical models using modern application packages of modeling software; selection of software tools for implementation and testing of models in big data analysis tasks; application of Hadoop , Apache Spark, PySpark , Spark SQL, Spark MLlib and other distributed data processing tools for conducting computational experiments; preparation of large data sets for modeling, including cleaning, transformation, aggregation and formation of features; implementation of models of classification, regression, clustering, search for association rules, anomaly

		detection and analysis of stream data using modern software packages; tuning of model parameters and conducting an experimental study of their behavior on various data sets; assessment of the quality, stability, scalability and computational efficiency of the studied models; analysis of modeling results and identification of limitations of the applied software tools; formulation of substantiated conclusions on the applicability of the studied mathematical models and application packages to the tasks of Big Data Analysis and Mining.
ID-2 _{PC-6}	Uses modern methods for developing algorithms for mathematical models	Using modern methods for developing algorithms for mathematical models in big data analysis problems; selecting an algorithmic approach taking into account the structure, volume, quality and dynamics of the source data; developing algorithms for pre-processing, filtering, aggregation, joining, transformation and feature selection for large data sets; applying distributed and parallel processing methods in the implementation of algorithms for mathematical models; developing algorithms for classification, regression, clustering, searching for association rules, anomaly detection and analysis of streaming data; adapting existing algorithms to the conditions of scalable data processing in the Hadoop and Apache Spark environments; assessing the computational complexity, performance, stability and

	reproducibility of the developed algorithms; comparing alternative algorithmic solutions based on the quality of the results and the efficiency of processing; justifying the choice of methods for developing algorithms for mathematical models to solve Big Data Analysis and Mining problems.
ID-3_{PC-6} Uses modern programming languages to implement specific algorithms and mathematical models	Using modern programming languages to implement specific algorithms and mathematical models in Big Data analysis tasks; selecting a programming language, programming interface, and library taking into account the specifics of the problem being solved, the volume of data, and the requirements for distributed processing; using Python, PySpark, Spark SQL, Spark MLlib, and other modern tools for software implementation of Big Data Analysis and Mining algorithms; developing software solutions for loading, cleaning, filtering, aggregating, joining, and transforming large data sets; implementing algorithms for classification, regression, clustering, searching for association rules, anomaly detection, and analyzing streaming data; building software pipelines for preparing features, training models, and evaluating the quality of the results; debugging, testing, and verifying the correctness of the implemented algorithms; analyzing the performance, scalability, and reproducibility of the software implementation;

		interpreting the results of the implemented mathematical models and justifying the applicability of the selected programming languages and software tools to Big Data Analysis and Mining tasks.
--	--	--

4. Volume of the academic discipline (module) and forms of control *

Amount of classes: total: 3 credits , 108 academic hours .	OFO, In academic hours
Contact work:	32.00/0
Lectures/including practical training	0/0
Laboratory work /including practical training	
Practical classes /including practical training	32.00/0
Independent work	76.00
Forms of control	
Credit	Yes

5. Importance practical work students

Practical work is a key element of mastering the "Big Data Analysis and Mining / Анализ и обработка больших данных" course. In the field of big data analysis, it's not enough to simply know the basic concepts, architectures, and algorithms. It's important to understand how these approaches are implemented in real-world software environments, how computation is distributed, how data storage is organized, and what limitations arise when processing large, heterogeneous, and dynamically changing datasets.

The main goal of practical classes is not only to reinforce theoretical material but also to develop students' solid skills in working with modern technologies for distributed storage, processing, and data mining. The practical training allows students to:

- understand how big data processing systems are structured and how theoretical principles are implemented in Hadoop , HDFS and Apache Spark;
- master basic operations of distributed data storage, including loading, reading, organizing directories, working with files and analyzing the storage structure in HDFS;
- learn to perform distributed data processing using MapReduce , PySpark , Spark DataFrameAPI and Spark SQL;
- work through the full cycle of preparing big data: loading, cleaning, filtering, aggregation, joining tables, transforming features and saving results;
- study the features of working with various data formats, including CSV, JSON, Parquet and other formats used in Big Data systems;

- understand the causes of distributed computing performance degradation, including shuffle , data skew, poor partitioning , inefficient joins, and redundant data reads;
- master methods for optimizing Spark applications, including caching, partition configuration , broadcast join , execution plan analysis and using Spark UI;

- gain practical experience in building batch data processing pipelines with the separation of raw , cleaned and curated layers ;
- become familiar with the principles of stream data processing, window analytics, checkpointing , watermarking and processing of delayed events;
- apply data mining techniques, including classification, regression, clustering, association rule mining and anomaly detection;
- use Spark MLlib to build ML pipelines , prepare features, train models, select parameters and evaluate the quality of results;
- develop skills in interpreting analysis results, comparing models, identifying limitations of algorithms and justifying the choice of a specific method or tool.

Practical work also develops professionally significant competencies related to the independent development of big data processing algorithms, the selection of distributed processing tools, the implementation of mathematical models in modern programming languages, and the evaluation of the quality of the obtained results. Regularly completing practical assignments allows students not only to memorize terminology but also to understand the real-world connections between the architecture of big data systems, data analysis algorithms, computational efficiency, and the quality of analytical conclusions.

Practical classes are especially important for preparing students to solve research and applied problems. Working with Hadoop , Apache Spark, PySpark , Spark SQL, and Spark MLlib helps them understand the full lifecycle of a big data analysis project: from acquiring and storing source data to building models, interpreting results, and assessing the solution's suitability for scalable application. This experience enhances students' readiness for participation in real-world projects, scientific research, and professional work in the field of Big Data Analysis and Mining.

PRACTICAL WORK #1

Topic: Deploying a Training Environment for Big Data

Objective:

To become familiar with the educational software environment for practical work in the "Big Data Analysis and Mining / Анализ и обработка больших данных" course. To master the basic steps of installing, running, and testing Hadoop , HDFS, Apache Spark, PySpark , and JupyterLab tools . To gain basic skills in working with a containerized environment for big data processing.

Theoretical rationale:

Modern big data analysis tasks require specialized software tools that enable storing, processing, and analyzing data in a distributed environment. Unlike local processing of small files, the Big Data approach involves working with large volumes of data, parallel computation, distributed storage, and the use of software platforms designed for scalability.

Hadoop is one of the foundational technologies in big data. Its ecosystem includes the HDFS distributed file system, designed for storing large files across a cluster of computing nodes, as well as tools for organizing distributed data processing. HDFS allows files to be divided into blocks, stored on different nodes, and ensures fault tolerance through data replication.

Apache Spark is a modern distributed data processing platform widely used for analytics, machine learning, stream processing, and building data processing pipelines . Spark supports various programming interfaces, including PySpark , which allows you to work with Spark using Py-

thon. This makes the platform suitable for students already familiar with the basics of programming and data analysis.

JupyterLab is used as an interactive environment for executing code, documenting experiments, and analyzing results. Within the discipline, JupyterLab can be used to run PySpark code, explore datasets, execute SQL queries, build simple analytics pipelines, and prepare reports on practical work.

For educational purposes, it is convenient to use a containerized environment based on Docker and Docker Compose. This approach allows for the rapid deployment of a consistent set of services without complex manual configuration of the operating system. Containers can include Hadoop, Spark, JupyterLab, and supporting components. This reduces the likelihood of installation errors and allows all students to work in the same software environment.

Deploying a training environment is a necessary first step in mastering the course. Without a properly configured environment, it is impossible to complete further practical work related to HDFS, MapReduce, Spark SQL, PySpark, Spark MLlib, batch pipelines, and stream data processing. Therefore, students must be able to launch the environment, check the availability of key components, execute basic Hadoop and Spark commands, and understand the purpose of each element of the software stack.

Equipment and materials:

Computer or laptop with internet access; Windows, Linux, or macOS operating system;

Docker Desktop or Docker Engine

installed; Docker installed Compose

Browser Google Chrome, Mozilla Firefox, or Microsoft Edge

Operating system terminal or command line Tutorial Docker Compose file for starting Hadoop and Spark environment

Sample text or CSV file for testing data loading Software: Hadoop, HDFS, Apache Spark, PySpark, JupyterLab, Python 3.x

Typical tasks:

1. Install Docker and Docker Compose on your working computer. Verify the installation using the Docker version and Docker commands. Compose. Record the verification results in a report.
2. Get or create a tutorial project for deploying a Big Data environment. Learn about the project structure and the purpose of Docker. Compose file and a list of services included in the environment.
3. Run a Big Data training environment using Docker Compose. Ensure that the Hadoop, HDFS, Spark, and JupyterLab containers are running without errors.
4. Check the availability of the web interfaces for the main components of the environment, if they are included in the configuration. Open the HDFS NameNode interface, the Spark interface, or the JupyterLab interface in a browser.
5. Hadoop verification commands. Print the Hadoop version, check HDFS availability, and create a test directory in HDFS for future practice.
6. Upload a small test file from your local file system to HDFS. Verify the file's existence in the distributed file system and display its contents using HDFS commands.
7. Launch PySpark or Spark-shell. Create a simple Spark session and perform a minimal data processing operation, such as counting the number of rows in a test file.

8. Open JupyterLab and create a new notebook. In the notebook, check that Python and PySpark are available , create a SparkSession , and perform a simple operation on the test dataset.
9. Prepare a directory structure for storing practical work files: source data, program code, intermediate results and reports.
10. Provide a brief description of the deployed environment: what components are included in the training stack, what role each component plays, and how they will be used in subsequent practical work.

Test questions:

1. What is the Big Data learning environment used for in the Big Data Analysis and Mining course?
2. What role does Docker play in deploying a big data analytics software environment?
3. Than Docker Is Compose useful for running multiple related services?
4. Hadoop and Spark training environment ?
5. What is the purpose of the HDFS distributed file system?
6. What is the role of NameNode in HDFS architecture?
7. What is the role of DataNode in HDFS architecture?
8. Why is Apache Spark used in big data analytics tasks?
9. What is PySpark and why is it useful for educational tasks?
10. What role does JupyterLab play in learning big data analytics?
11. What commands can I use to check that Docker is installed and working correctly?
12. Docker containers ?
13. How to create a directory in HDFS using the command line?
14. How to upload a file from a local file system to HDFS?
15. How to display the contents of a file located in HDFS?
16. What is SparkSession and what is it used for?
17. Why is it necessary to test Hadoop , HDFS, and Spark before performing practical exercises?
18. What are the benefits of using a containerized environment over manually installing Hadoop and Spark?
19. What errors might occur when launching a learning environment and how can they be diagnosed?
20. Why is a unified software environment important for the reproducibility of Big Data practical assignments?

PRACTICAL WORK #2

Topic: Working with HDFS: A Big Data File System

Objective:

To become familiar with the purpose and basic operating principles of the HDFS distributed file system. To master basic commands for creating directories, loading, viewing, copying, moving, and deleting files in HDFS. To gain practical skills in organizing a data storage structure for subsequent processing using Hadoop and Apache Spark.

Theoretical background:

HDFS, Hadoop Distributed File System (HDFS) is a distributed file system designed for storing large volumes of data across a cluster of computing nodes. Unlike a traditional local file system,

HDFS is designed for working with large files, which are split into blocks and placed on different cluster nodes. This approach enables parallel processing of large data and reduces the system's dependence on a single physical storage medium.

The HDFS architecture is built on the interaction of the NameNode, a control node, and DataNodes, worker nodes. The NameNode stores file system metadata: the directory structure, file information, block information, and their locations. DataNodes store the data blocks themselves and perform read and write operations upon client requests. The user works with HDFS as a single file system, without manually managing the placement of individual blocks.

One of the key mechanisms of HDFS is replication. Each file block can be stored in multiple copies on different nodes. This increases system fault tolerance: if one node fails, the data remains accessible thanks to copies on other nodes. The replication factor can be configured in the HDFS settings or modified for individual files.

HDFS is well suited for storing large files, event logs, data dumps from information systems, and datasets for subsequent analytical processing and machine learning. However, HDFS is inefficient for storing large numbers of small files, as each file places a load on the NameNode, which stores metadata in memory. Therefore, when designing Big Data solutions, it is important to consider file size, directory structure, and data storage format.

Working with HDFS is usually done through `hdfs` commands. `dfs`, which allow you to create directories, load data, view file contents, and copy and delete objects. These commands are largely similar to standard Linux commands, but they are executed not on the local file system, but on the Hadoop distributed storage.

Mastering HDFS is a necessary step in studying the Big Data Analysis and Mining discipline, since most of the subsequent practical work will be related to loading source data, storing intermediate results, organizing data layers, and launching distributed processing using Hadoop and Apache Spark.

Equipment and materials:

Computer or laptop with internet access; Deployed Big Data training environment from practical work #1; Installed and running Hadoop and HDFS components
; Access to the terminal or command line of the Hadoop container; Browser for viewing the HDFS

NameNode

web interface ; Test files in TXT, CSV, or JSON formats; Software: Hadoop , HDFS, Docker , Docker Compose , an operating system terminal

Typical tasks:

1. Launch the Big Data training environment prepared in Lab 1. Verify that the Hadoop and HDFS containers are running. List the running containers and record the results in a report.
2. Log into a container or workstation where Hadoop commands are available . Verify HDFS availability by running the list root directory command.
3. / user / student directory in HDFS if it doesn't already exist. Within it, create the following subdirectories: input , output , raw , processed , and archive . Explain the purpose of each directory.

4. On your local computer or inside a container, create a small text file called `sample.txt` containing a few lines of arbitrary text. Upload this file to the `/ user / student / input` directory in HDFS.
5. Verify the presence of the `sample.txt` file in HDFS. List the files in the `/ user / student / input` directory , then view the contents of the downloaded file using HDFS.
6. Create a CSV file, `data.csv` , containing at least 10 rows of conditional data, such as user ID, event category, and numeric value. Upload the file to the `/ user / student / raw` directory .
7. Display the first lines of the `data.csv` file from HDFS. Check the file size and access rights. Compare the output of the file listing commands on the local file system and on HDFS.
8. Copy file `data.csv` from `catalog /user/student/raw` V `catalog /user/student/processed` . Then rename the copy of the file to `data_processed.csv` .
9. Download the `data_processed.csv` file from HDFS back to your local file system. Verify that the file's contents remain unchanged.
10. Change the replication factor for one of the uploaded files, if allowed by your learning environment. Review the change and explain the role of replication in HDFS.
11. Delete the temporary `sample.txt` file from the `/ user / student / input` directory . Verify that the file no longer appears in the HDFS object list.
12. Create several small files and upload them to a separate directory , `/ user / student / small_files` . Consider why a large number of small files might be undesirable for HDFS.
13. NameNode web interface . Find information about the file system, uploaded files, available space, and DataNode status .
14. Provide a brief diagram of the directory structure you have created in HDFS and indicate which data should be stored in which directories when performing subsequent practical tasks.
15. Prepare a report that includes the HDFS commands used, screenshots or text fragments of their execution results, and brief conclusions about the purpose of HDFS in big data analysis tasks.

Test questions:

1. What is HDFS and what is it used for in the Hadoop ecosystem ?
2. How is HDFS different from a regular local file system?
3. Why is HDFS designed for storing large files?
4. What is the role of NameNode in HDFS architecture?
5. What is the role of DataNode in HDFS architecture?
6. What is a data block in HDFS?
7. What is block replication used for in HDFS?
8. What happens to the data if one of the DataNode fails if replication is enabled?
9. Why is storing large numbers of small files considered a problem for HDFS?
10. What command is used to list the contents of a directory in HDFS?
11. What command is used to create a directory in HDFS?
12. How to upload a file from a local file system to HDFS?
13. How to download a file from HDFS to a local file system?
14. How to view the contents of a file located in HDFS?
15. the `put` and `get` commands when working with HDFS?
16. What are the `raw` , `processed` , and `archive` directories used for when organizing data storage?

17. What information about files can be obtained using HDFS commands?
18. NameNode web interface help monitor HDFS status?
19. Why is directory structure important when organizing data for further processing in Spark?
20. What errors can occur when working with HDFS and how can they be diagnosed?

PRACTICAL WORK #3

Topic: Analysis of the data storage structure in HDFS

Objective:

To study the physical and logical organization of data storage in HDFS. To master the analysis of file structure, blocks, replication factors, data size, and file placement in a distributed file system. To develop an understanding of the impact of file size, number of files, and storage parameters on the efficiency of subsequent big data processing.

Theoretical rationale:

HDFS is designed for storing large volumes of data in a distributed environment. Unlike a local file system, where a file is typically stored on a single device, HDFS breaks a file into fixed-size blocks and distributes these blocks across cluster nodes. This approach enables parallel data access and ensures storage fault tolerance.

A data block is one of the fundamental storage units in HDFS. The block size is determined by file system settings and is typically significantly larger than the block size in traditional file systems. This is because HDFS is designed to store and process large files. A larger block size reduces the load on the metadata system and allows for more efficient sequential reading of large data arrays.

The NameNode is responsible for storing metadata in HDFS. It stores information about the directory structure, files, sizes, access rights, blocks, and block locations on the DataNode. The data itself is stored on the DataNode. When accessing a file, the client obtains block location information from the NameNode and then reads the data directly from the DataNode.

Block replication is an important fault-tolerance mechanism in HDFS. Each block can be stored in multiple copies on different nodes. If one node becomes unavailable, the system can continue working with another copy of the block. The replication factor determines how many copies of each block should be allocated in the cluster. The higher the replication factor, the greater the storage reliability, but the more disk space is required.

When analyzing the data storage structure, it's important to consider not only the total data volume but also the number of files. HDFS handles large files efficiently, but is poorly suited for storing large numbers of small files. Each file, even the smallest, creates a metadata entry in the NameNode. If there are many such files, the memory load on the NameNode increases, and data processing efficiency decreases.

The data storage structure in HDFS directly impacts the performance of subsequent processing in Hadoop and Apache Spark. Incorrect file organization, an excessive number of small objects, poor partitioning, or excessive data fragmentation can lead to a large number of tasks, unnecessary read overhead, and reduced computational performance. Therefore, a Big Data specialist must be able to analyze not only the data content but also the way it is stored.

In this lab, students explore file storage parameters in HDFS, compare the placement of a single large file with a set of small files, analyze the replication factor, block sizes, and information available through HDFS commands and the NameNode web interface .

Equipment and materials:

Computer or laptop with internet access; Expanded Big Data training environment from practical work #1; Running Hadoop and HDFS components

; Access to the terminal or command line of the Hadoop container; Browser for working with the HDFS

NameNode

web interface; Test files of different sizes in TXT or CSV formats; A set of small test files for analyzing the problem Software

: Hadoop , HDFS, Docker , Docker Compose , an operating system terminal

Typical tasks:

1. Launch the Big Data training environment and check the status of the Hadoop and HDFS containers. Ensure that HDFS is accessible for executing commands.
2. Create in HDFS directory `/user/student/ storage_analysis` . Inside him create subdirectories `large_file` , `small_files` , `replication_test` And `reports` .
3. Prepare one relatively large test file. If you don't have one, create one by repeating lines of text or CSV data multiple times. Upload file V catalog `/user/student/ stor-age_analysis / large_file` .
4. Prepare a set of several small files. Each file should contain a small amount of text or tabular data. Upload these files V catalog `/user/student/ storage_analysis / small_files` .
5. Display information about the uploaded files using HDFS commands. Record the file name, size, permissions, owner, group, and file system path.
6. Compare the total size of a single large file and a set of small files. Explain why the same total data size can have different impacts on HDFS performance depending on the number of files.
7. Use the HDFS file details command. Determine the number of blocks, block size, and replication factor for a large file.
8. Perform a similar analysis for several small files. Compare how many metadata records are created when storing one large file versus a set of small files.
9. Review the file's block details and their locations on the DataNode , if available in your training environment. Record your findings in a report.
10. Change the replication factor for the file in the `/ user / student / stor-age_analysis / replication_test` directory if your environment settings allow it. Check how the file information changes after changing the replication factor.
11. Compare the actual file size with the disk space required after replication. Explain why replication increases the required storage space.
12. NameNode web interface . Locate the uploaded files and view available information about the directory structure, file size, blocks, and DataNode status .
13. Analyze how HDFS displays directories and files through the web interface. Compare the information obtained through the web interface and through terminal commands.
14. Create a brief comparison table: one large file, several small files, and a file with a modified replication factor. For each option, specify the number of files, total size, number of blocks, and storage features.
15. Formulate a conclusion about which data storage options are more suitable for subsequent processing by Hadoop and Apache Spark.

16. Remove any test data that won't be needed in future tutorials. Verify that the HDFS directory structure is in order.
17. Prepare a report on the practical work. The report should include the commands used, file analysis results, block and replication information, and conclusions about the impact of the storage structure on big data processing.

Test questions:

1. What is a data block in HDFS?
2. Why is the block size in HDFS typically larger than in traditional file systems?
3. How does HDFS store files larger than a single block size?
4. What role does NameNode play when storing data in HDFS?
5. What information about files and blocks is stored in NameNode metadata ?
6. What is the role of DataNodes in HDFS?
7. What is replication factor in HDFS?
8. How does replication improve data storage fault tolerance?
9. Why does increasing the replication factor require more disk space?
10. Why is HDFS considered efficient for storing large files?
11. What is the problem with large number of small files in HDFS?
12. Why does a large number of small files increase the load on the NameNode ?
13. How does data storage structure affect downstream processing performance in Spark?
14. What HDFS commands allow you to get information about files and directories?
15. How can I view the contents of a directory in HDFS?
16. How can I get file size information in HDFS?
17. How can I change the replication factor of a file in HDFS?
18. NameNode web interface ?
19. Why is it important to think through the directory structure in advance when designing a Big Data warehouse?
20. What recommendations can be formulated for storing data that will later be processed using Hadoop and Apache Spark?

PRACTICAL WORK #4

MapReduce Model

Objective:

To become familiar with the classic MapReduce model of distributed batch data processing . To master the basic stages of a MapReduce task: reading input data, executing the map function , intermediate grouping and redistribution of data during the shuffle stage , executing the reduce function , and storing the result in HDFS. To gain practical skills in running a simple MapReduce task in the Hadoop environment .

Theoretical background:

MapReduce is one of the classic models of distributed big data processing. It was developed to perform computations on large data sets that are impossible or inefficient to process on a single computer. The basic idea of MapReduce is to divide a task into multiple independent operations that can be executed in parallel on different nodes of a cluster.

MapReduce model consists of two main stages: Map and Reduce . In the Map stage , the input data is broken into individual chunks, after which a mapping function is applied to each chunk.

This function transforms the input records into key- value pairs. For example, when counting words in a text, the Map function might transform each found word into a pair of the form: word, 1.

Map execution, an intermediate Shuffle step occurs. and Sort . At this stage, the system groups all pairs with identical keys and passes them to the appropriate Reduce tasks . Shuffling is often the most expensive part of distributed processing, as it can require transferring large amounts of data between cluster nodes.

Reduce step aggregates values related to a single key. In a word count task, the Reduce function receives a word and a list of units, then sums them to produce a final result: the word and the number of times it appears in the input dataset.

A classic example of a MapReduce task is Word Count , which calculates the frequency of words in a text file. Despite its simplicity, this task effectively demonstrates the basic principles of distributed processing: partitioning the input data, parallel execution of map functions , grouping by keys, and final aggregation.

Hadoop MapReduce is closely tied to HDFS. Input data is typically stored in HDFS, and processing results are also written to HDFS. This allows computations to be performed close to the data storage location and reduces network transfer costs. This approach is especially important when working with large files.

In modern Big Data systems, classic MapReduce has given way to Apache Spark for many tasks, offering a more flexible and faster computing model. However, learning MapReduce is essential for understanding the basic principles of distributed data processing, the role of keys and values, the shuffle stage , and the general logic of batch processing large datasets.

Equipment and materials:

Computer or laptop with internet access; Deployed Big Data training environment from previous practical work; Running Hadoop and HDFS components;
Access to the terminal or command line of the Hadoop container; Browser for viewing the HDFS NameNode web interface and, if available, the Hadoop task execution interface
; Test text file for running the Word Count task;
Software: Hadoop , HDFS, MapReduce , Docker , Docker Compose , an operating system terminal

Typical tasks:

1. Launch the Big Data training environment and verify that Hadoop and HDFS components are working correctly. Ensure that HDFS commands are accessible from the terminal.
2. Create in HDFS directory `/user/student/ mapreduce` . Inside it, create the input and output subdirectories . If the output directory already exists from previous runs, delete it before rerunning the task.
3. Prepare a text file called `text.txt` containing several paragraphs of any text in Russian or English. For ease of analysis, add any repeated words.
4. Upload the `text.txt` file to the `/ user / student / mapreduce / input` directory in HDFS. Verify that the file has been successfully placed in the distributed file system.
5. Display the contents of a file from HDFS using the command line. Ensure the text displays correctly and is ready for processing.

6. Find the standard Hadoop example for the Word Count task in your installed environment . This is typically included in the Hadoop examples JAR file .
7. Run the Word Count MapReduce task on the file located in the `/ user / student / mapreduce / input directory` . Specify the output directory as `/ user / student / mapreduce / output` .
8. Check the task's execution status in the terminal. Note the messages about the map and reduce tasks running, the number of records processed, and the task's completion.
9. Browse the contents of the output directory in HDFS. Find the file containing the task results, such as `part-r-000000` .
10. Print the word count results to the terminal. Analyze which words appeared in the input text and how many times each word was found.
11. Re-run the task after changing the source code. Before re-running, delete the previous output directory, as Hadoop MapReduce does not write the result to an existing directory.
12. Compare the results of the two runs. Explain how changing the input file affected the resulting word-count pairs.
13. Create a flowchart for the Word Count task : input text, Map step , intermediate key-value pairs, Shuffle step and Sort , Reduce stage , final result.
14. Count problem as an example , explain which data are keys and which are values in the Map and Reduce steps .
15. NameNode web interface and locate the MapReduce task's input and output directories. Verify the presence of the source and output files.
16. If a web interface for monitoring Hadoop tasks is available in your training environment, open it and examine the details of the completed task: execution time, number of map and reduce tasks, and the amount of data processed.
17. the MapReduce task output from HDFS to the local file system. Open the output file and verify that the data matches the terminal output.
18. Formulate a conclusion about what actions Hadoop performs automatically when starting a MapReduce task: reading data from HDFS, splitting data, starting tasks, grouping keys, writing the result.
19. Compare manually counting words in a small text file with performing the same task using MapReduce . Explain why MapReduce makes sense for large datasets rather than small files.
20. Prepare a report on the practical exercise. The report should include the commands used, the HDFS directory structure, screenshots or terminal output fragments, the Word Count result , and a brief explanation of the Map and Shuffle steps. and Sort and Reduce .

Test questions:

1. What is MapReduce ?
2. What tasks is the MapReduce model used for ?
3. What are the two main steps involved in the MapReduce model ?
4. What is the role of the Map function ?
5. What is the role of the Reduce function ?
6. What is a key-value pair in MapReduce ?
7. What is the key and value in a Word Count problem ?
8. What happens during the Shuffle phase? and Sort ?
9. Why can the shuffle step be expensive when processing big data?
10. How is HDFS used when executing a MapReduce task?
11. MapReduce input and output data usually stored in HDFS?
12. Why doesn't Hadoop allow writing output to an existing output directory?
13. What is Word Count and why is this problem often used to study MapReduce ?
14. How does MapReduce achieve parallel data processing?

15. What role does input partitioning play when running a MapReduce task?
16. How is MapReduce different from regular local file processing?
17. What are the limitations of the classical MapReduce model ?
18. Why is Apache Spark more efficient than classic MapReduce in many tasks ?
19. What information about the execution of a MapReduce task can be obtained from the terminal or monitoring web interface?
20. Why is learning MapReduce important for understanding modern big data processing systems?

PRACTICAL WORK #5

Topic: Processing Structured Data with Hadoop

Objective:

To become familiar with the specifics of processing structured data in the Hadoop ecosystem .
To master the application of Hadoop. Streaming for distributed CSV data processing using custom mapper and reducer programs . Gain practical skills in loading tabular data into HDFS, performing key-based aggregation, and storing processing results in a distributed file system.

Theoretical background:

In big data analysis tasks, a significant portion of the information is presented in a structured or semi-structured form. This data includes tables, event logs, transactions, user information, measurement results, monitoring data, and other data sets containing records with a fixed set of fields. One of the most common storage formats for such data is CSV, where each row corresponds to a separate record, and field values are separated by a special character, most often a comma or semicolon.

Hadoop can be used not only for processing text files but also for analyzing structured data. Each line of the file can be treated as a separate record, from which the required fields are extracted. The data can then be grouped, aggregated, filtered, and transformed using the MapReduce model .

Hadoop is convenient for processing structured data . Streaming . Hadoop Streaming allows you to run MapReduce tasks in which the mapper and reducer functions are implemented not only in Java, but also in other programming languages, such as Python. This is what Hadoop does. Streaming is a convenient tool for educational tasks, since the student can focus on the logic of data processing without delving into the development of Java applications for Hadoop .

The Mapper receives input strings, parses them into fields, and generates intermediate key-value pairs. For example, if the input CSV file contains user event information, the Mapper might extract the event category and numeric value, then output the category-value pair. The Reducer receives all values related to a single key and performs the final aggregation: counting the number of records, summing, calculating the average, and finding the maximum or minimum.

When processing structured data, proper format parsing is especially important. It's important to consider the table header, field separator, data types, possible omissions, and invalid values. Errors during the CSV file parsing stage can lead to incorrect processing results or MapReduce task failure.

In this practical, students process a CSV file using Hadoop. Streaming . The main task is to group records by a selected field and calculate aggregated metrics. This work allows us to under-

stand how the classic MapReduce model applies to tabular data and why structured processing requires careful attention to the schema, types, and quality of the data.

Equipment and materials:

Computer or laptop with internet access; Deployed Big Data training environment from previous practical work; Running Hadoop and HDFS components;

Access to the terminal or command line of the Hadoop container;

Text editor or development environment for creating mapper and reducer files

; Test CSV file with tabular data; Software: Hadoop , HDFS, Hadoop Streaming , Python 3.x, Docker , Docker Compose , an operating system terminal

Typical tasks:

1. Launch the Big Data training environment and verify that the Hadoop and HDFS components are working correctly. Ensure that `hdfs` commands are available from the terminal. `dfs` and Hadoop startup command Streaming .
2. Create in HDFS directory `/user/student/ structured_data` . Inside it, create the subdirectories `input` , `output` , `scripts` , and `results` .
3. Prepare a CSV file `events.csv` containing at least 20 records. file should be fields : `user_id` , `event_type` , `category` , `value` , `timestamp` . For example, one row might describe a user action, an event category, and a numeric value.
4. Verify the validity of the CSV file in your local file system. Ensure the first row contains a header and all other rows have the same number of fields.
5. Download file `events.csv` V catalog `/user/student/ structured_data /input` in HDFS. Check for the file's existence and print the first lines using HDFS commands.
6. Develop a mapper program in Python. The mapper should read the rows of the input CSV file, skip the header row, extract the `category` field and the `value` field , and then output the intermediate pairs of the form `category value` .
7. Test the mapper locally on a small portion of the CSV file. Ensure that the output generates correct key-value pairs.
8. Develop a reducer program in Python. The reducer should receive sorted category pairs. `value` , group data by category and calculate the number of records and the sum of values for each category.
9. the mapper and reducer connection locally by passing the mapper's output to the reducer's input . Ensure that the resulting aggregation is performed correctly.
10. Run a MapReduce task using Hadoop Streaming , specifying a mapper file , a reducer file , an HDFS input directory, and an HDFS output directory. Make sure the output directory doesn't exist before running.
11. Check the task's execution status in the terminal. Note the messages about the number of rows processed, the map and reduce steps running, and any warnings or errors.
12. Browse the contents of the output directory `/ user / student / structured_data / output` . Find the file containing the processing results and output its contents to the terminal.
13. Copy the task's output from HDFS to your local file system. Save the resulting file as part of your practical report.
14. Modify the mapper so that grouping is performed by the `event_type` field instead of the `category` field . Rerun the MapReduce task and compare the results.
15. Add a calculation of the average value for each group to the reducer . Check the correctness of the result and explain how the average is calculated in the Reduce step .
16. Add handling for invalid rows: rows with insufficient fields, empty values, or a non-numeric value in the `value` field . Explain why this check is important when processing real data.

17. Compare the results of manual calculations using a small CSV file with the results of the MapReduce task. Ensure that distributed processing produces the correct results.
18. Create a structured data processing flow chart: CSV file, mapper row reading , field extraction, key-value pair generation, shuffle grouping, reducer aggregation , and writing the result to HDFS.
19. Formulate a conclusion about the advantages and limitations of processing structured data using Hadoop Streaming .
20. Prepare a report on the practical work. The report should include the CSV file structure, the mapper and reducer code, the Hadoop commands used , the results of the task, and a brief explanation of the data processing steps.

Test questions:

1. What is meant by structured data?
2. Why are CSV files often used to store and exchange tabular data?
3. What issues might arise when processing CSV files in a distributed environment?
4. Hadoop used for ? Streaming ?
5. What is Hadoop? How is Streaming different from the classic MapReduce implementation in Java?
6. What role does a mapper play in processing structured data?
7. What role does a reducer play in processing structured data?
8. What is the key and value when grouping CSV data by category?
9. Why should the header row of a CSV file usually be excluded from processing?
10. What happens during the shuffle step when aggregating structured data?
11. Why is it necessary to delete the output directory before rerunning a Hadoop task?
12. How to check the result of a MapReduce task in HDFS?
13. What aggregate metrics can be calculated using reducer ?
14. Why is it important to check data types when parsing a CSV file?
15. How does handling missing or invalid values affect the quality of the result?
16. In what cases is the MapReduce approach suitable for processing tabular data?
17. What are the limitations of processing CSV files using Hadoop? Streaming ?
18. Hive often used for complex analytical processing of structured data ?
19. How are structured data processing and data mining tasks related?
20. MapReduce task input and output data in HDFS?

PRACTICAL WORK #6

Topic: Introduction to Apache Spark and PySpark

Objective:

To become familiar with the purpose of Apache Spark as a platform for distributed big data processing. To learn how to launch Spark applications using PySpark , create a SparkSession , load data from a local file system and HDFS, perform simple data processing operations, and save results. To develop a basic understanding of Spark architecture and how it differs from the classic Hadoop model. MapReduce .

Theoretical background:

Apache Spark is one of the most widely used platforms for distributed big data processing. It is used for batch processing, interactive analytics, stream processing, machine learning, and building complex data processing pipelines . Unlike classic Hadoop, MapReduce , Spark supports a

more flexible computing model and makes extensive use of in-memory data processing, allowing it to speed up many analytical tasks.

Spark organizes computations as a set of operations on distributed data. The user defines a sequence of transformations, and Spark automatically creates an execution plan, breaks the work into tasks, and distributes them among computing nodes. This allows processing data volumes larger than the capacity of a single computer.

One of the key concepts in Spark is the `SparkSession`. The `SparkSession` is the primary entry point into a Spark application. It is used to create `DataFrames`, execute SQL queries, configure interactions with data sources, and initiate processing operations. In modern Spark applications, the `SparkSession` replaces the lower-level `SparkContext` when working with `DataFrames` and Spark SQL.

PySpark is an Apache Spark interface for Python. It allows Spark capabilities to be leveraged within the Python environment familiar to analysts and researchers. Using PySpark, you can load data, filter, aggregate, and transform it, build machine learning models, and save the results in a variety of formats.

The Spark architecture consists of several core components. A Driver is responsible for managing application execution, building a computation plan, and coordinating tasks. Executors perform data processing tasks on worker nodes and store intermediate data. A Job is a computational task launched when an action is executed. A Stage groups a set of tasks that can be executed without additional data redistribution. A Task is the smallest unit of execution for a single data partition.

A key feature of Spark is lazy execution of computations. Data transformation operations, such as filter or map, are not executed immediately after being called. Spark merely remembers the sequence of transformations and builds a logical plan. Actual execution begins only when an action is called, such as count, collect, show, or save. This approach allows Spark to optimize the execution of operations.

When starting to learn Spark, it's important to master basic data loading and processing operations. Students should learn how to create a `SparkSession`, read text and CSV files, perform simple filtering, row counting, column selection, aggregation, and storing results. These skills will form the foundation for subsequent practical work involving RDDs, the `DataFrame` API, Spark SQL, Spark application optimization, and using Spark MLlib.

Equipment and materials:

Computer or laptop with internet access; Deployed Big Data training environment from previous practical work; Running Hadoop, HDFS, and Apache Spark components;
; Access to the terminal or command line of the Spark container; Access to JupyterLab or another PySpark code runtime; Test files in TXT and CSV formats; Software: Apache Spark, PySpark, Hadoop, HDFS, Python 3.x, JupyterLab, Docker Compose, an operating system terminal

Typical tasks:

1. Launch the Big Data training environment and verify that the Hadoop, HDFS, Spark, and JupyterLab containers are running correctly. List the running containers and record the results in a report.

2. Check Apache Spark availability from the terminal. Print the Spark version and ensure that the PySpark or spark-submit launch command is available in your environment.
3. Open JupyterLab and create a new notebook for the practical. In the first text box, enter the notebook number, topic, objective, and a brief description of the environment used.
4. Create a SparkSession in PySpark . Specify an application name, such as `BigDataPractice06` . Verify that the SparkSession was created without errors.
5. Print basic information about the current Spark application: Spark version, application name, run mode, and available SparkContext parameters .
6. Prepare a small text file, `words.txt` , containing a few lines of text. Upload it to the container's local file system or to the JupyterLab working directory .
7. Read a text file using Spark. Count the number of lines in the file and print the first few lines to the screen.
8. Perform simple processing on a text file: split lines into words, convert words to lower-case, remove empty elements, and count the total number of words.
9. Implement a basic word frequency count using PySpark . Store the result as a set of word-count pairs.
10. Prepare a CSV file `events.csv` , containing fields `user_id` , `event_type` , `category` , `value` , `timestamp` . Use the file created in the previous lab or prepare a new test dataset.
11. Upload a CSV file in Spark DataFrame . Specify the presence of a header and enable automatic data schema detection, or specify the schema manually.
12. Derive the schema of the DataFrame . Analyze the column data types and explain why correctly inferring the types is important for further processing.
13. Display the first rows of the DataFrame . Verify that the data loaded correctly and that the column values display as expected.
14. Perform simple data filtering. For example, select only rows in which the `value` field is greater than a specified threshold.
15. Group the data by the `category` or `event_type` field . Calculate the number of records in each group and the sum of the values in the `value` field .
16. Save the aggregation results to the local file system or HDFS in CSV or Parquet format . Verify that the output files were created successfully.
17. Upload the source CSV file to HDFS if it hasn't already been loaded. Read the file into Spark again from HDFS and verify that processing is performed correctly.
18. Compare data processing approaches in Hadoop MapReduce and Apache Spark, using the example of counting words or aggregating CSV data. Indicate which operations are simpler and more compact in Spark.
19. Terminate the SparkSession after all operations are complete. Explain why graceful application termination is important when working with computing resources.
20. Prepare a report on the practical work. The report should include the commands used, PySpark code fragments, the results of the operations, the DataFrame schema , the aggregation results, and conclusions about the use of Apache Spark for big data analysis tasks.

Test questions:

1. Why is Apache Spark used in big data analytics tasks?
2. How Apache Spark differs from the classic Hadoop model MapReduce ?
3. What is PySpark ?
4. Why is PySpark useful for educational and analytical tasks?
5. What is SparkSession ?
6. What is the role of the driver in the Spark architecture?
7. What role do executors play in the Spark architecture?
8. What is a job in Apache Spark?

9. What is a stage in the Spark execution model?
10. What is a task in Apache Spark?
11. What does lazy execution mean in Spark?
12. How is transformation different from action ?
13. Why don't data transformation operations in Spark run immediately?
14. What actions can trigger actual execution of computations in Spark?
15. What is DataFrame used for in Spark?
16. Why is it important to analyze the DataFrame schema after loading data?
17. What data formats can Apache Spark read?
18. How can Spark work with files stored in HDFS?
19. Why Spark is considered a more flexible data processing tool than Hadoop MapReduce ?
20. PySpark skills inform subsequent topics in the Big Data Analysis and Mining discipline?

PRACTICAL WORK #7

Topic: RDD: Basic Transformations and Actions

Objective:

To become familiar with RDDs as a basic low-level data abstraction in Apache Spark. To learn how to create RDDs from local collections and files, perform basic transformations and actions, and understand the difference between transformations and actions . To gain practical skills in using the opera-

tions `map` , `flatMap` , `filter` , `reduceByKey` , `groupByKey` , `count` , `collect` , `take` , and storing processing results.

Theoretical background:

RDD, Resilient Distributed A dataset is one of Apache Spark's core abstractions. An RDD is a distributed collection of elements that can be processed in parallel across multiple cluster nodes. Each RDD is divided into partitions , and operations on these partitions are performed independently, ensuring scalability of data processing.

One of the key features of RDDs is fault tolerance. Spark stores not the intermediate data itself at each stage, but rather information about how it was obtained. This information is called lineage . If part of the data is lost due to a failure, Spark can recreate the required partition by re-running the transformation chain. This approach allows for reliable computations without constantly writing intermediate results to disk.

RDD operations are divided into two main groups: transformations and actions. Transformations create a new RDD based on an existing one. These operations include `map` , `flatMap` , `filter` , `distinct` , `union` , `reduceByKey` , `groupByKey` , and others. Transformations in Spark are executed lazily: when they are called, the calculations are not executed immediately, but are only added to the execution plan.

Actions initiate the actual execution of computations and return the result to the user or save it to an external system. Actions include `count` , `collect` , `take` , `first` , `reduce` , `saveAsTextFile` , and others. When an action is called, Spark builds a physical computation plan, breaks the task into stages and tasks , and initiates data processing.

The `map` operation is used to transform each element of an RDD into a new element. The `flatMap` operation is similar to `map` but allows you to obtain multiple output elements from a single input element. It is often used to split lines of text into words. The `filter` operation selects only those elements that satisfy a given condition.

Special operations are used to process key-value pairs. For example, `reduceByKey` groups values with the same key and performs aggregation. Unlike `groupByKey`, `reduceByKey` is generally more efficient because it can perform partial aggregation before transferring data between nodes. This reduces shuffle overhead and improves processing performance.

Although modern Spark applications more often use the DataFrame API and Spark SQL, understanding RDDs remains essential for understanding Spark's internals. RDDs provide insight into distributed collections, partitions, lazy execution, dependencies between operations, shuffles, and the underlying Spark computation model.

In this practical, students learn basic operations on RDDs using examples of processing text and structured data. Particular attention is paid to the difference between transformations and actions, as well as the comparison of `reduceByKey` and `groupByKey` operations when performing aggregation by keys.

Equipment and materials:

Computer or laptop with internet access; Deployed Big Data training environment from previous practical work; Running Hadoop, HDFS, and Apache Spark components; Access to JupyterLab or a terminal with PySpark; Test text file for processing strings and words; Test CSV file with simple structured data; Software: Apache Spark, PySpark, Hadoop, HDFS, Python 3.x, JupyterLab, Docker Compose, an operating system terminal

Typical tasks:

1. Launch the Big Data training environment and verify that Apache Spark is available to run PySpark code. Open JupyterLab or the PySpark terminal.
2. Create a new notebook for this practical assignment. In the first text box, enter the assignment number, topic, objective, and a brief description of the environment used.
3. Create a `SparkSession` with the application name `BigDataPractice07`. Get the `SparkContext` from the created `SparkSession` and display basic information about it.
4. Create an RDD from a local collection of numbers, such as 1 through 20. Print the first elements of the RDD and determine the number of elements using the `count` action.
5. Apply the `map` operation to a numeric RDD to obtain a new RDD containing the squares of the original numbers. Print the first few elements of the result.
6. Use the `filter` operation to select only even numbers from the original RDD. Count the number of selected elements and print them.
7. Use the `reduce` operation to calculate the sum of the elements of a numeric RDD. Compare the result with a regular sum calculation in Python.
8. Prepare a text file, `text_rdd.txt`, containing several lines of text with repeated words. Upload the file to your working directory or to HDFS.
9. Create an RDD based on a text file. Print the file's line count and the first lines using the `count` and `take` actions.
10. Use the `flatMap` operation to split lines of text into individual words. Convert words to lowercase and remove empty elements.
11. Implement word frequency counting using `map` and `reduceByKey` operations. Form pairs of the form `word, 1`, then perform word aggregation.

12. Sort the word count results by descending frequency. Display the ten most frequently occurring words.
13. Repeat word aggregation using `groupByKey` . Compare the execution logic of `groupByKey` and `reduceByKey` . Explain why `reduceByKey` is generally preferred for large datasets.
14. Create an RDD from the CSV file `events.csv` containing the `user_id` , `event_type` , `category` , `value` , and `timestamp` fields . Skip the header row and parse the rows into individual fields.
15. Create an RDD of category-value pairs. Aggregate the sum of the values by category using `reduceByKey` .
16. Count events by event type. To do this, create `event_type` , `1` pairs and apply `reduceByKey` .
17. Determine the number of partitions for multiple created RDDs. Change the number of partitions using `repartition` or when creating the RDD, if supported by the selected data loading method.
18. Save the result of one of the aggregations to HDFS or the local file system using `saveAsTextFile` . Check the output files that were created.
19. Formulate a conclusion about the difference between transformations and actions in Spark. Indicate which operations in the work performed are transformations and which are actions .
20. Prepare a report on the practical work. The report should include PySpark code fragments, the results of the operations, examples of RDD transformations, and conclusions about `map` , `flatMap` , `filter` , `reduceByKey` , `groupByKey` , `count` , `collect` , `take` , and `saveAsTextFile` .

Test questions:

1. What is RDD in Apache Spark?
2. Why is RDD called distributed dataset?
3. What does RDD fault tolerance mean?
4. What is lineage in Spark?
5. What is an RDD partition ?
6. How does the number of partitions affect parallel data processing?
7. How is transformation different from action ?
8. Why are transformations lazily executed in Spark?
9. What operations are considered transformations ?
10. What operations are related to actions ?
11. `map` operation used for ?
12. How is `flatMap` different from `map` ?
13. `filter` operation used for ?
14. What does the `count` action do ?
15. What are the dangers of using `collect` when working with big data?
16. `take` operation used for ?
17. What is a key-value RDD?
18. `reduceByKey` used for ?
19. How is `reduceByKey` different from `groupByKey` ?
20. Why might `groupByKey` be ineffective when handling large data?
21. What is shuffle in Spark?
22. Why might key aggregation operations cause shuffle ?
23. How to save the result of RDD processing to a file?
24. Why do modern Spark applications often use the DataFrame API instead of RDDs?

25. Why learn RDDs when DataFrames and Spark SQL are more commonly used in practical tasks?

PRACTICAL WORK #8

Practical Work #8

Topic: Spark DataFrame API

Objective:

To become familiar with the DataFrame API as the primary tool for processing structured and semi-structured data in Apache Spark. Learn how to load data from CSV, JSON, and Parquet files, view the data schema, select columns, filter rows, create new columns, group, aggregate, and save processing results. Develop practical skills in working with distributed tabular data in PySpark .

Theoretical rationale:

The Spark DataFrame API is one of Apache Spark's primary programming interfaces for processing structured data. A DataFrame is a distributed dataset organized as a table with rows and named columns. Unlike an RDD, a DataFrame has a schema that defines column names and data types. Having a schema allows Spark to analyze the data structure, build an optimized execution plan, and process large datasets more efficiently.

The DataFrame API occupies a middle ground between the low-level RDD model and the declarative approach of Spark SQL. Users work with data as if it were a table, but utilize API methods such as `select` , `filter` , `where` , `withColumn` , `groupBy` , `agg` , `orderBy` , `join` , and others. This approach is convenient for developing analytical pipelines , processing event logs, transactional data, user activity, and other typical Big Data collections.

One of the advantages of DataFrame is its ability to work with various data sources. Spark can read data from the local file system, HDFS, object storage, databases, and streaming sources. CSV, JSON, and Parquet files are most often used in training tasks . CSV is convenient for easily exchanging tabular data, JSON is often used for semi-structured records and events, and Parquet is a columnar format effective for big data analytics.

When loading data into a DataFrame, the schema plays an important role. The schema can be automatically detected or specified manually. Automatic schema detection is convenient for initial experiments, but in real-world projects, it's preferable to explicitly specify the schema to avoid data type interpretation errors. For example, a numeric field might be mistakenly read as a string, and a date field might require additional conversion.

The DataFrame API supports common data preparation and analysis operations. `select` selects desired columns, `filter` and `where` allow you to select rows based on a condition, `withColumn` is used to create new columns or modify existing ones, and `groupBy` and `agg` are used to aggregate data into groups. These operations form the basis of most big data preprocessing and analytics tasks.

A key feature of Spark is its lazy execution of computations. As with RDDs, most DataFrame operations are transformations and do not immediately trigger computations. Actual processing begins when an action is called, such as `show` , `count` , or `collect` , or when the result is saved to a file. This allows Spark to analyze the chain of operations and optimize the execution plan.

The DataFrame API is tightly coupled with the Catalyst optimizer, which converts a logical query plan into a physical execution plan. This makes the DataFrame API a more convenient and performant tool than manual RDD processing for many common analytical tasks. Therefore, in modern Big Data projects, the DataFrame API and Spark SQL are used significantly more often than low-level RDDs.

Mastering the Spark DataFrame API is an important step in the Big Data Analysis and Mining discipline, as this interface is used to prepare data, build data marts, execute analytical queries, transform features, and transfer data to Spark MLlib for further intelligent analysis.

Equipment and materials:

Computer or laptop with internet access; Deployed Big Data training environment from previous practical work; Running Hadoop , HDFS, and Apache Spark components

; Access to JupyterLab or a terminal with PySpark;

Test files in CSV, JSON, and Parquet formats

; File with data on user events or transactions; Software: Apache Spark, PySpark , Hadoop , HDFS, Python 3.x, JupyterLab , Docker , Docker Compose , an operating system terminal

Typical tasks:

1. Launch the Big Data training environment and verify that Apache Spark, HDFS, and JupyterLab are working correctly. Open JupyterLab or another PySpark runtime .
2. Create a new notebook for this practical assignment. In the first text box, enter the assignment number, topic, objective, and a brief description of the environment used.
3. Create a SparkSession with the application name `BigDataPractice08` . Check the Spark version and ensure the session was created without errors.
4. Prepare a CSV file `events.csv` , containing fields `user_id` , `event_type` , `category` , `value` , `timestamp` , `device` . The file must contain at least 30 records with repeating categories and event types.
5. Upload a CSV file in a Spark DataFrame . Specify the presence of a header and enable automatic schema detection. Print the first rows of the DataFrame using the `show` method .
6. Print the DataFrame schema using the `printSchema` method . Analyze the data types Spark has determined for each column.
7. Repeat the CSV file upload with an explicit data schema definition. Compare the results of automatic schema detection and the explicitly defined schema.
8. Perform a selection of individual columns using `select` . Create a DataFrame containing only `user_id` , `event_type` , `category` , and `value` .
9. Filter rows using `filter` or `where` . Select only those records in which the `value` field is greater than the specified threshold.
10. Create a new column using `withColumn` . For example, add a `value_category` column that takes the value `high` if `value` is greater than a specified threshold, and `normal` otherwise.
11. Group the data by the `category` field . Calculate the number of records, sum, average, minimum, and maximum values of the `value` field for each category.
12. Group the data by two fields: `category` and `event_type` . Calculate the number of events for each pair of values.
13. Sort the aggregation result by descending number of records or sum of values. Display the first rows of the sorted result.
14. Find unique values in the `category` , `event_type` , and `device` columns . Determine the number of unique users in the dataset.

15. Add missing value handling. Create multiple rows with empty values or use a prepared file with missing values. Delete rows with missing values or fill in values according to a specified rule.
16. Save the aggregation result in CSV format to your local file system or HDFS. Check which files were created in the output directory.
17. Save the same result in Parquet format . Compare the structure of the CSV and Parquet output files .
18. Load the saved Parquet file back into the Spark DataFrame . Verify that the data schema and values are preserved correctly.
19. Use the `explain` method to view the execution plan for one of the filtering or aggregation operations. Capture the output and briefly explain the processing steps Spark performs.
20. Compare processing structured data using RDDs and the DataFrame API. Explain why the DataFrame API is more convenient for tabular data and analytical operations.
21. Prepare a report on the practical work. The report should include fragments of PySpark code, the DataFrame schema , the results of filtering, grouping, aggregation, and saving data, and conclusions about the purpose of the Spark DataFrame API.

Test questions:

1. What such DataFrame in Apache Spark?
2. How is DataFrame different from RDD?
3. Why is having a data schema important when working with a DataFrame ?
4. What data sources can the Spark DataFrame API read?
5. What is the use of `read` method in Spark?
6. How does automatic schema detection differ from explicit schema definition?
7. Why is it common to define the data schema manually in real-world projects?
8. `show` method used for ?
9. `printSchema` method used for ?
10. `select` operation used for ?
11. How is `filter` different from `where` in the Spark DataFrame API?
12. `withColumn` method used for ?
13. How to group data in Spark DataFrame API?
14. What aggregate functions can be applied to a DataFrame ?
15. `orderBy` method used for ?
16. How to determine the number of rows in a DataFrame ?
17. How to find unique values in a DataFrame column ?
18. What missing value handling options are available in the Spark DataFrame API?
19. Why is the Parquet format effective for big data analytics?
20. How does writing data to CSV differ from writing to Parquet ?
21. What does lazy execution mean when working with a DataFrame ?
22. DataFrame operations typically trigger the actual execution of computations?
23. `explain` method used for ?
24. How does the DataFrame API relate to the Catalyst optimizer?
25. Why is the DataFrame API the primary tool for processing structured data in modern Spark applications?

PRACTICAL WORK #9

Topic: Spark SQL and Analytical Query Execution

Objective:

To become familiar with Spark SQL's capabilities for performing analytical queries on large structured data. To master the creation of temporal views based on DataFrames , execution of SQL queries, filtering, grouping, aggregation, table joins, sorting results, and query plan analysis. To develop practical skills in applying the SQL approach in a distributed Apache Spark environment.

Theoretical background:

Spark SQL is a core component of Apache Spark, designed for processing structured and semi-structured data using SQL queries. It allows you to work with distributed data as tables, run analytical queries, combine datasets, calculate aggregated metrics, and use the familiar declarative SQL language in a big data environment.

Unlike traditional relational DBMSs, Spark SQL executes queries over distributed datasets, which can be stored in HDFS, a local file system, object storage, or other sources. The user formulates a SQL query, and Spark automatically builds a logical and physical execution plan, distributes computations among executors , and processes the data in parallel.

To execute SQL queries in Spark, data is typically first loaded into a DataFrame . The DataFrame is then registered as a temporary view using the `createOrReplaceTempView` method . It can then be accessed as a table in SQL queries. This approach combines the benefits of the DataFrame API with the expressiveness of SQL.

Spark SQL supports the core SQL language constructs: `SELECT` , `WHERE` , `GROUP BY` , `HAVING` , `ORDER BY` , `JOIN` , aggregate functions, conditional expressions, and date and string manipulation. This makes it a convenient tool for analysts, data engineers, and data mining specialists who need to quickly obtain slices, aggregates, and analytical metrics from large data sets.

The Catalyst optimizer plays a key role in Spark SQL. It analyzes the SQL query, builds a logical plan, performs optimizations, and generates a physical execution plan. This allows Spark to choose more efficient execution methods, such as applying filtering optimizations, reordering operations, and using broadcast. join for small tables and reduce the amount of data processed.

When executing analytical queries, table join operations are especially important. Joins are often one of the most expensive operations in Big Data systems, as they can cause shuffles , or redistribution of data between cluster nodes. Therefore, when working with Spark SQL, it's important not only to be able to write correct queries but also to understand how they are physically executed.

`explain` method displays the execution plan for an SQL query or DataFrame operation . Analyzing the execution plan helps understand the operations Spark is performing, including where filtering, aggregation, sorting, joining , and shuffling occur . This is essential for further exploration of Spark application optimization.

Spark SQL is widely used for building analytics marts, preparing data for machine learning, exploring event logs, analyzing transactions, processing user activity, and generating reports. Within the discipline of "Big Data Analysis and Mining / Анализ и обработка больших данных," Spark SQL serves as the connecting tool between data storage, distributed processing, and subsequent intelligent analysis.

Equipment and materials:

Computer or laptop with internet access; Deployed Big Data training environment from previous practical work; Running Hadoop , HDFS, and Apache Spark components ; Access to JupyterLab or a terminal with PySpark; Test CSV or Parquet files with data on events, users, products, or transactions; Software: Apache Spark, PySpark , Spark SQL, Hadoop , HDFS, Python 3.x, JupyterLab , Docker , Docker Compose , an operating system terminal

Typical tasks:

1. Launch the Big Data training environment and verify that Apache Spark, HDFS, and JupyterLab are working correctly. Open JupyterLab or another PySpark runtime .
2. Create a new notebook for this practical assignment. In the first text box, enter the assignment number, topic, objective, and a brief description of the environment used.
3. Create a SparkSession with the application name `BigDataPractice09` . Check the Spark version and ensure the session was created without errors.
4. Prepare a CSV file `events.csv` , containing fields `event_id` , `user_id` , `event_type` , `category` , `value` , `event_time` , `device` . The file must contain at least 30 records with different categories and event types.
5. Prepare a second CSV file, `users.csv` , containing the `user_id` , `user_name` , `region` , and `age_group` fields . The file should contain users whose IDs appear in the `events.csv` file .
6. Load both CSV files into a Spark DataFrame . Specify the presence of a header and enable automatic schema detection, or specify the schema manually.
7. Print the schemas of both DataFrames using the `printSchema` method . Verify that numeric fields, IDs, and time data are interpreted correctly.
8. Register the DataFrame with events as the temporary view `events` , and the DataFrame with users as the temporary view `users` .
9. Run a simple SQL query against the `events` table , selecting all records. Limit the output to the first ten rows.
10. Run an SQL query selecting individual columns: `user_id` , `event_type` , `category` , `value` . Sort the results in descending order by `value` .
11. Filter the data using an SQL query. Select only events with a value greater than the specified threshold.
12. Group events by the `category` field . Calculate the number of events, sum, average, minimum, and maximum values of the `value` field for each category.
13. Group by two fields: `category` and `event_type` . Determine the number of events for each combination of category and event type.
14. Use a `CASE WHEN` conditional expression to separate events into groups, such as `high_value` and `normal_value` , based on the value of the `value` field .
15. Run a SQL query with the `HAVING` clause to return only those categories in which the number of events exceeds the specified threshold.
16. Join the `events` and `users` tables on the `user_id` field . Generate a result containing information about the user, region, event type, category, and event value.
17. Calculate aggregated indicators by user regions: number of events, total value , average value .
18. Run a query to identify the most active users by number of events. Display the top 5 users.
19. Run a query to determine the most valuable categories based on the total value . Sort the results by descending sum.
20. Save one of the SQL query results in CSV or Parquet format to the local file system or HDFS.

21. Read the saved result back into the DataFrame and register it as a new temporary view. Run the SQL query again against the saved result.
22. Use the `explain` method for a single, simple filtered query. Capture the execution plan and explain what operations Spark performs.
23. Use the `explain` method for a query with grouping. Note the presence of aggregation operations and possible data redistribution.
24. Use the `explain` method for queries that join tables. Identify which execution steps may be the most expensive.
25. Compare the same analytical operation using the DataFrame API and Spark SQL. For example, group by category using both methods and compare the results.
26. Formulate a conclusion about the advantages of Spark SQL when performing analytical queries on big data.
27. Prepare a report on the practical work. The report should include PySpark code fragments, SQL queries, query results, table schemas, example execution plans, and conclusions about the application of Spark SQL to Big Data Analysis and Mining tasks.

Test questions:

1. What is Spark SQL used for?
2. How is Spark SQL different from regular SQL in a relational database?
3. What is a temporal view in Spark SQL?
4. `createOrReplaceTempView` method used for ?
5. How to execute SQL query in PySpark ?
6. What common SQL constructs are supported in Spark SQL?
7. `SELECT` statement used for ?
8. `WHERE` operator used for ?
9. `GROUP BY` used for ?
10. How is the `WHERE` condition different from the `HAVING` condition ?
11. What are aggregate functions used for in SQL?
12. What are the most commonly used aggregate functions in data analysis?
13. the `ORDER BY` operator used for ?
14. What is join in SQL?
15. What types of table joins can be used in Spark SQL?
16. join operations be expensive when processing large data?
17. What is shuffle and why might it occur when running SQL queries?
18. What is the Catalyst optimizer used for?
19. What does the `explain` method show ?
20. How does a logical execution plan differ from a physical plan?
21. Why is it important to analyze the execution plan of an analytical query?
22. When is Spark SQL more convenient than RDD API?
23. When is Spark SQL more convenient than DataFrame API?
24. How can Spark SQL be used in building analytical data marts?
25. How can SQL query results be used for subsequent Data Mining?

PRACTICAL WORK #10

Topic: Data Storage Formats: CSV, JSON, Parquet

Objective:

To become familiar with the main data storage formats used in Big Data Analysis and Mining.

To master reading and writing data in CSV, JSON, and Parquet formats using Apache Spark. To learn how to compare storage formats based on structure, file size, schema usability, read speed, and applicability to analytical tasks.

Theoretical rationale:

In big data analytics systems, the choice of storage format plays a crucial role. It determines disk space consumption, read and write speed, schema processing ease, support for nested structures, the ability to effectively filter, and the cost of executing analytical queries. An incorrectly chosen format can lead to excessive memory consumption, slow data reading, and reduced distributed processing performance.

CSV is one of the most common text formats for storing tabular data. Each row in a CSV file corresponds to a single record, and field values are separated by a special character, most often a comma or semicolon. CSV is simple, human-readable, and supported by virtually all data processing tools. However, this format does not preserve the data schema, does not support complex nested structures, is inefficient when selecting individual columns, and can lead to data type interpretation errors.

JSON is a text format convenient for storing semi-structured data. It is widely used for transferring data through APIs, storing events, logs, user actions, and documents with variable structures. JSON supports nested objects and arrays, making it more flexible than CSV. However, JSON typically takes up more space than CSV or Parquet and may be less efficient for mass analytical processing of large datasets.

Parquet is a columnar binary data storage format widely used in the Hadoop, Spark, and Lakehouse ecosystems. Unlike row-based formats, Parquet stores data columnarly, which is particularly convenient for analytical queries that often only use a subset of a table's fields. Columnar storage reduces read overhead, efficiently compresses data, and accelerates queries with filtering and aggregation. Furthermore, Parquet stores information about the data schema, reducing the risk of errors during subsequent processing.

and Mining tasks, the data format should be chosen based on the nature of the processing. CSV is convenient for exchanging small table sets and initial data loading. JSON is suitable for events, logs, and semi-structured records. Parquet is preferred for long-term storage of large analytical datasets, preparing data marts, batch pipelines, and subsequent use in Spark SQL and Spark MLlib.

Apache Spark supports reading and writing all of these formats. Using the Spark DataFrame API, you can load the same dataset from CSV, convert it, and save it in JSON and Parquet. You can then compare the resulting file size, read speed, data schema, and ease of analytical operations. This comparison provides a practical understanding of why modern Big Data systems often use text formats for exchange and loading, while columnar formats are used for analytical storage and processing.

In this practical, students convert a single data set between CSV, JSON, and Parquet formats, analyze the features of each format, compare file sizes, validate data schemas, and run simple analytical queries to evaluate the usability of further processing.

Equipment and materials:

Computer or laptop with internet access; Deployed Big Data training environment from previous practicals;

Running Hadoop, HDFS, and Apache Spark components

; Access to JupyterLab or a terminal with PySpark;
Test CSV file with tabular data; Test JSON file with semi-structured data
; HDFS directory or local directory for saving results; Software: Apache Spark, PySpark , Spark SQL, Hadoop , HDFS, Python 3.x , JupyterLab , Docker Compose , an operating system terminal

Typical tasks:

1. Launch the Big Data training environment and check the functionality of Apache Spark, HDFS, and JupyterLab . Ensure that HDFS commands are available and that PySpark code can be executed.
2. Create a new notebook for this practical assignment. In the first text box, enter the assignment number, topic, objective, and a brief description of the environment used.
3. Create a SparkSession with the application name BigDataPractice10 . Check the Spark version and ensure the session was created without errors.
4. Prepare a source CSV file `events.csv` containing at least 50 records. file should be fields `event_id` , `user_id` , `event_type` , `category` , `value` , `event_time` , `device` , `region` .
5. Upload a CSV file in Spark DataFrame . Specify the presence of a header and enable automatic schema detection, or specify the schema manually.
6. Display the first rows of the DataFrame and the data schema. Check that the column types are defined correctly, especially for numeric and time fields.
7. Save the downloaded DataFrame in CSV format to a separate directory. Specify the overwrite option to prevent rerunning the data from causing an error with an existing directory.
8. Save the same DataFrame as JSON. Check the output directory structure and open one of the resulting files to view the contents.
9. Save the same DataFrame in Parquet format . Verify that Spark has created a directory with the Parquet binary files and result support files.
10. Compare the directory sizes obtained after saving data in CSV, JSON, and Parquet formats . Record the results in a report.
11. Load the data back into Spark, saved in CSV format. Print the data schema and compare it with the schema of the original DataFrame .
12. Load back the data saved in JSON format. Analyze the schema and row structure. Pay attention to the specifics of data representation in JSON.
13. Load back the data saved in Parquet format . Print the schema and verify that the data types were saved correctly.
14. Perform the same analytical operation for the three storage options: for example, grouping by category with a count of the number of events and the sum of the value field .
15. Measure the approximate reading and operation times for each format, if allowed by the learning environment. Compare the results and draw a conclusion.
16. Perform a selection of only a few columns, such as category , value , and region , from CSV, JSON, and Parquet data . Explain why Parquet might be more efficient when reading a limited set of columns.
17. Filter data by one of the fields, such as region or category . Compare the ease of filtering for different formats.
18. Create a small JSON file with a nested structure, such as a user object containing the fields id , region , and device . Load it into Spark and examine the resulting schema.
19. Extract nested fields from the JSON structure and convert the result into a flat DataFrame suitable for analytical processing.
20. Save the cleaned and transformed dataset in Parquet format as a final analysis dataset for further practical work.

21. Open the HDFS or local directory containing the results and compare the files Spark creates when writing CSV, JSON, and Parquet . Note the presence of part- * files and service files.
22. Use the explain method for a Parquet read operation with a limited number of columns selected. Analyze how Spark builds an execution plan.
23. Create a comparison table of CSV, JSON, and Parquet formats based on the following criteria: human readability, schema availability, support for nested structures, storage efficiency, ease of analytical queries, and applicability to Big Data.
24. Formulate a conclusion about which format is appropriate to use for initial data loading, event storage, analytics marts, and data preparation for machine learning.
25. Prepare a report on the practical work. The report should include PySpark code fragments, data schemas, file reading and writing results, directory size comparisons, examples of analytical operations, and conclusions about the applicability of the CSV, JSON, and Parquet formats .

Test questions:

1. What are data storage formats used for in Big Data systems?
2. What is CSV format?
3. What are the benefits of CSV?
4. What are the limitations of CSV when processing big data?
5. Why is CSV not always suitable for long-term analytical storage?
6. What is JSON format?
7. For what tasks is JSON useful?
8. What are the advantages of JSON over CSV?
9. What are the disadvantages of JSON when processing large amounts of data?
10. What is the Parquet format ?
11. Why is Parquet called a columnar storage format?
12. How is columnar storage different from row-based storage?
13. Why is Parquet effective for analytical queries?
14. How does the presence of a data schema affect the reliability of processing?
15. Parquet faster when selecting individual columns?
16. What formats are best for initial data loading?
17. What formats are best for storing analytics displays?
18. Why can the same dataset take up different amounts of space in CSV, JSON, and Parquet ?
19. What factors influence file sizes in different storage formats?
20. How does Spark write data to a directory when using DataFrameWriter ?
21. part - * files when writing ?
22. How to read CSV file into Spark DataFrame ?
23. How to read JSON file into Spark DataFrame ?
24. How to read Parquet file into Spark DataFrame ?
25. Why is the Parquet format often used with Spark SQL and Spark MLlib ?

PRACTICAL WORK #11

Topic: Joins, Groupings, and Window Functions in Spark

Objective:

Learn how to use the Spark DataFrame API and Spark SQL to perform analytical operations on

multiple related datasets. Learn how to join tables, group, aggregate, sort, build ratings, and use window functions. Gain practical skills in analyzing user events, transactions, or other structured data in a distributed Apache Spark environment.

Theoretical rationale:

Big data analysis tasks rarely use just one isolated dataset. More often, an analytical task requires combining multiple tables: user events, user references, product information, transactions, regions, devices, or categories. To obtain useful analytical insights, it is necessary to be able to relate this data, perform groupings, calculate aggregated metrics, and analyze the behavior of objects within groups.

One of the basic operations when working with multiple datasets is joining . Joining allows you to combine rows from different DataFrames based on a common key, such as `user_id` , `product_id` , `event_id` , or another identifier. Spark supports different types Joins : inner join, left join, right join, full outer join, semi join, and anti join. The choice of join type depends on the problem statement and the requirements for storing records from one or both tables.

Join operations in a distributed environment can be quite expensive. If data with the same keys resides on different nodes or in different partitions , Spark is forced to redistribute the data between executors . This redistribution is called shuffle . The larger the volume of data being joined, the higher the potential cost of the operation. Therefore, when performing a join, it is important to understand the data structure, table sizes, the presence of duplicate keys, and the possibility of applying optimizations such as broadcast . join for a small lookup table.

Grouping and aggregation are used to obtain summary metrics. Using `groupBy` and aggregate functions, you can calculate the number of events, the sum of values, the average, the minimum, the maximum, the number of unique users, and other metrics. These operations form the basis for building analytics displays, reports, and preliminary data analysis before applying data mining methods.

Window functions allow you to perform calculations on groups of rows without collapsing the result to a single row per group. Unlike standard aggregation, a window function preserves the detail of the source records and adds calculated values to them. For example, you can calculate the ordinal number of a user event, the rank of a product within a category, a running sum, the average value for a group, or the difference between the current and previous event.

In Spark, window functions are defined using the `Window` object , which specifies rules for dividing data into groups, sorting order, and window boundaries. The most common window functions include `row_number` , `rank` , `dense_rank` , `lag` , `lead` , `sum` , `avg` , `count` , and others. They are especially useful for analyzing time series, user events, transactions, rankings, sessions, and metric dynamics.

Join, group, and window operations are basic tools for preparing data for data mining. Before training classification, regression, clustering, or recommender models, it is often necessary to generate features based on historical events: number of user actions, average purchase value, last event type, activity frequency, object position in a ranking, or change in a metric over time.

In this practical, students analyze multiple related DataFrames , apply various join types , calculate aggregate metrics, and use window functions to construct rankings and analyze events within groups. Particular attention is paid to understanding which operations can cause shuffles and why execution plan analysis is important when working with big data.

Equipment and materials:

Computer or laptop with internet access; Deployed Big Data training environment from previous practical work; Running Hadoop , HDFS, and Apache Spark components
; Access to JupyterLab or a terminal with PySpark;
Test CSV or Parquet files with data on users, events, products, or transactions
; HDFS directory or a local directory for storing source and result data; Software: Apache Spark, PySpark , Spark SQL, Hadoop , HDFS, Python 3.x, JupyterLab , Docker , Docker Compose , an operating system terminal

Typical tasks:

1. Launch the Big Data training environment and check the functionality of Apache Spark, HDFS, and JupyterLab . Ensure that the previously created data directories are accessible or prepare new working directories.
2. Create a new notebook for this practical assignment. In the first text box, enter the assignment number, topic, objective, and a brief description of the environment used.
3. Create a SparkSession with the application name `BigDataPractice11` . Check the Spark version and ensure the session was created without errors.
4. Prepare kit data `users.csv` , containing fields `user_id` , `user_name` , `region` , `age_group` , `registration_date` . The data must contain at least 15 users.
5. Prepare kit data `events.csv` , containing fields `event_id` , `user_id` , `event_type` , `category` , `value` , `event_time` , `device` . The data must contain at least 50 events associated with users from the `users` table .
6. Prepare an additional `categories.csv` directory containing the `category` , `category_group` , and `description` fields . The directory should allow you to supplement events with category group information.
7. Load all three files into a Spark DataFrame . For each DataFrame , output the first rows and the data schema. Check that the column types are correct.
8. Perform inner Join the `events` and `users` tables by the `user_id` field . Create a DataFrame containing information about the event, user, region, and age group.
9. Execute left Join the `events` and `users` tables . Compare the number of rows in the inner result, join and left join . Explain why the results may differ.
10. Join the result with the `categories` table using the `category` field . Add category group information to the events.
11. Group events by user region. Calculate the number of events, the total `value` , the average `value` , and the number of unique users in each region.
12. Group by two criteria: `region` and `event_type` . Determine which event types are most common in different regions.
13. Perform aggregation by category. Calculate the number of events, the sum of values, the average value, the minimum value, and the maximum value for each category.
14. Sort the aggregation results by descending number of events and descending sum of values. Compare whether the most frequent and most valuable categories match.
15. `row_number` window function to assign an event sequence number to each user in event time order.
16. the `rank` or `dense_rank` window function to rank the categories within each region by the sum of their `value` .
17. `lag` window function to get the value of the previous event for each user. Calculate the difference between the current and previous `value` .
18. Use windowed aggregation to calculate the cumulative sum of `value` for each user in event time order.

19. Create a table of user attributes: number of events, sum of values, average event value, number of unique categories, date of first and last event.
20. Save the resulting feature table in Parquet format to your local file system or HDFS. Verify that the data was written successfully.
21. Register the underlying DataFrames as temporary views and rerun some of the analytics queries using Spark SQL.
22. Compare the implementation of a single grouping operation using the DataFrame API and Spark SQL. Ensure the results are consistent.
23. Use the `explain` method for the join operation. Determine which execution steps Spark uses when joining tables.
24. Use the `explain` method for grouping or windowing operations. Note the sorting, aggregation, and possible shuffle operations.
25. Formulate a conclusion about which operations in this work are the most costly and why.
26. Prepare a report on the practical work. The report should include PySpark code fragments, source DataFrame schemas, results of joins, groupings, and window functions, example execution plans, and conclusions about the application of analytical operations in Spark.

Test questions:

1. Why are table join operations used in big data analysis tasks?
2. What is join in Spark DataFrame API?
3. How is inner join different from left join?
4. When is right join used?
5. When is full join used?
6. Why are join operations expensive in a distributed environment?
7. What is shuffle and why does it often occur when joining?
8. In what cases can broadcast join be used?
9. What is data grouping used for?
10. What are the most commonly used aggregate functions in big data analysis?
11. How is aggregation different from a window function?
12. What is a window function in Spark?
13. What is a window object used for in PySpark?
14. What does `partitionBy` specify in a window specification?
15. What does `orderBy` specify in a window specification?
16. What is the `row_number` function used for?
17. How is `rank` different from `dense_rank`?
18. What are `lag` and `lead` functions used for?
19. How can you calculate the cumulative sum using a window function?
20. Why are window functions useful when analyzing user events?
21. How are joins and groupings used in feature preparation for Data Mining?
22. Why is it convenient to save the user characteristics table in Parquet format?
23. How can Spark SQL be used with the DataFrame API?
24. What is the purpose of the `explain` method when analyzing join and window functions?
25. What factors should be considered when designing big data analytics queries?

PRACTICAL WORK #12

Topic: Optimizing Spark Applications

Objective:

To study the main approaches to optimizing Spark applications when processing big data. To master execution plan analysis, identifying costly operations, working with partitioning, caching, and broadcasting. Join, shuffle, and data skew. Gain practical skills in comparing different implementation options for a single analytical task based on execution time, volume of redistributed data, and computational resource efficiency.

Theoretical rationale:

Apache Spark provides convenient tools for distributed big data processing, but high performance of a Spark application is not achieved automatically. Even well-written code can run slowly if it uses inefficient joins, redundant shuffle operations, a suboptimal number of partitions, repeated recalculations of the same data, or operations that overload the driver.

Optimizing Spark applications begins with understanding the execution model. The user describes a sequence of data transformations, after which Spark builds a logical and physical execution plan. Based on this plan, the task is broken down into jobs, stages, and tasks. With narrow dependencies, data can be processed without redistribution between nodes. With broad dependencies, shuffling occurs, that is, the transfer and redistribution of data between executors. Shuffling is often one of the most expensive operations in Spark, as it requires network communication, writing intermediate data, and additional sorting or grouping.

One of the key performance factors is data partitioning. Partitions determine how a dataset is divided and how these parts are distributed among tasks. Too few partitions can lead to insufficient parallelism, while too many partitions increase the overhead of task scheduling and execution. The `repartition` and `coalesce` operations allow you to change the number of partitions, but they should be used wisely, as `repartition` causes a complete redistribution of the data, while `coalesce` is typically used to reduce the number of partitions with lower overhead.

Caching is an important tool for speeding up Spark applications when the same `DataFrame` or `RDD` is used multiple times. The `cache` method stores intermediate data in memory, while `persist` allows you to choose the storage layer. However, caching doesn't always speed up an application: if the data is used only once or doesn't fit in available memory, caching can create additional overhead.

Join operations often become a bottleneck in Spark applications. When joining large tables, Spark may perform shuffle operations. `join`, which redistributes data by keys between executors. If one of the tables is small, broadcast can be used `join`: A small table is passed to all executors, avoiding costly shuffles. This approach is especially effective when joining a large fact table with a small reference table.

Another typical problem is data skew, or data Skew occurs when some keys are encountered significantly more frequently than others, causing some partitions to become much larger than others. As a result, some tasks complete quickly, while others take too long and slow down the entire application. Skew can be addressed by using key distribution pre-analysis, join key rotation, salting, and broadcast. `join`, redistribute data, or change aggregation logic.

To analyze the performance of Spark applications, we use the `explain` method, Spark UI, execution logs, and comparison of execution times for different code variants. The `explain` method allows you to see the logical and physical execution plan, while the Spark UI displays jobs, stages, tasks, execution time, and shuffle volume. `read / write`, memory usage and possible errors.

In this practical, students perform the same analytical task in different ways, analyze execution plans, measure execution time, apply caching, change partitioning, use broadcast join and examine the impact of data skew. This allows us to understand that in Big Data, it's not just the correctness of an algorithm that matters, but also the efficiency of its implementation in a distributed environment.

Equipment and materials:

Computer or laptop with internet access; Deployed Big Data learning environment from previous practical work; Running Hadoop, HDFS, and Apache Spark components;
; Access to JupyterLab or a terminal with PySpark;
Access to the Spark UI, if provided by the learning environment configuration; Test CSV or Parquet files with data on events, users, products, or transactions;
A dataset with uneven key distribution for data analysis skew
HDFS directory or local directory for storing source and destination data
Software: Apache Spark, PySpark, Spark SQL, Hadoop, HDFS, Python 3.x, JupyterLab, Docker, Docker Compose, an operating system terminal

Typical tasks:

1. Launch the Big Data training environment and check the functionality of Apache Spark, HDFS, and JupyterLab. Ensure that Spark applications run successfully and can read data from the local file system or HDFS.
2. Create a new notebook for this practical assignment. In the first text box, enter the assignment number, topic, objective, and a brief description of the environment used.
3. Create a SparkSession with the application name `BigDataPractice12`. Check the Spark version and basic session parameters.
4. Prepare or download `kit data events.csv` With fields `event_id`, `user_id`, `event_type`, `category`, `value`, `event_time`, `device`, `region`. The set must contain enough rows to demonstrate differences in the execution times of operations.
5. Prepare a directory `users.csv` with the fields `user_id`, `user_name`, `region`, `age_group` and a directory `categories.csv` with the fields `category`, `category_group`, `description`.
6. Load the source files into a Spark DataFrame. Print the data schemas and the first rows of each DataFrame. Check that the data types are correct.
7. Determine the number of partitions for a DataFrame containing events. Report the results and explain why the number of partitions impacts Spark job execution.
8. Perform a simple analytical task: group events by category and calculate the number of events, the sum, and the average value of the `value` field. Measure the operation execution time.
9. Use the `explain` method for the grouping operation. Find the aggregation operations and possible shuffle in the execution plan. Record the plan fragment in the report.
10. Change the number of partitions in the original DataFrame using `repartition`. Repeat the aggregation and compare the execution time with the original.
11. Reduce the number of partitions using `coalesce`. Repeat the aggregation and compare the result with the previous versions. Explain the difference between `repartition` and `coalesce`.
12. Join a large `events` table with a `categories` reference table using the `category` field. Measure execution time and analyze the execution plan using `explain`.
13. Apply broadcast A join to connect `events` and `categories` if the category reference table is a small table. Compare the execution plan and running time with a regular join.

14. Join `events` and `users` by the `user_id` field . Analyze whether shuffle occurs when executing the join and explain the reason.
15. Create an intermediate DataFrame that is used in several subsequent operations, such as a cleaned event set with specified data types and filtered rows. Perform two or three aggregations on it without caching and measure the execution time.
16. Apply `cache` or `persist` to the intermediate DataFrame . Repeat the same operations and compare execution times. Explain in which cases caching is beneficial.
17. Review the actions that trigger actual materialized data caching. Explain why a single `cache` call is not sufficient to immediately load data into memory.
18. Create or use a dataset with skewed keys, such as one category that occurs significantly more frequently than others. Group or join on this key and analyze any potential performance impact.
19. Determine the distribution of the number of records by key. Find the most frequently occurring keys and explain why they might lead to data skew .
20. Suggest a way to reduce the impact of data skew: key rekeying, pre-aggregation, salting , broadcast join or change the processing logic. Implement one of these options if possible in the training environment.
21. Use the Spark UI to analyze one of your running applications, if the interface is available. Find information about jobs , stages , tasks , execution time, and shuffle read / write .
22. Compare several options for performing the same task: the original option, the option with modified partitioning , the option with caching, and the option with broadcast join . Create a comparison table.
23. Determine which changes actually improved task performance, and which had no noticeable effect or worsened the outcome. Explain the possible reasons.
24. Save the optimized processing result in Parquet format . Verify that the data was successfully written and can be read again.
25. Provide practical recommendations for optimizing Spark applications for big data processing.
26. Prepare a report on the practical work. The report should include PySpark code snippets, execution time measurements, explain execution plans, partition information , and examples of `cache` / `persist` and broadcast usage. join , as well as conclusions about the impact of optimizations on performance.

Test questions:

1. Why might a Spark application run slowly even though the code is correct?
2. What is a logical execution plan in Spark?
3. What is a physical execution plan in Spark?
4. explain method used for ?
5. What is a job in Spark?
6. What is a stage in Spark?
7. What is a task in Spark?
8. What is shuffle in Spark?
9. Why is shuffle an expensive operation?
10. What operations most often cause shuffle ?
11. What is data partitioning in Spark?
12. How does the number of partitions affect the performance of a Spark application?
13. How is repartition different from coalesce ?
14. In what cases is it appropriate to use repartition ?
15. When is it appropriate to use coalesce ?
16. What is cache used for ?
17. What is the difference between `cache` and `persist` ?

18. Why doesn't caching always speed up a Spark application?
19. What is a broadcast? join ?
20. In what case is broadcast can join be more efficient than regular join ?
21. Why can joining large tables be expensive?
22. What is data? skew ?
23. How does data skew affect Spark job execution?
24. What methods can be used to reduce the impact of data skew ?
25. What is Spark UI used for?
26. What metrics in Spark UI help identify performance bottlenecks?
27. Why is it important to compare several options for implementing the same task?
28. How can the Parquet format contribute to increased processing efficiency?
29. Why should Spark application optimization consider both code and data structure?
30. What are some key guidelines for developing effective Spark applications?

PRACTICAL WORK #13

Topic: Building a batch data processing pipeline

Objective:

To become familiar with the principles of building a batch data processing pipeline for Big Data Analysis and Mining tasks. To master the organization of raw , cleaned , and curated data layers , performing sequential processing of source data, cleaning and transforming records, creating an analytical dataset, and saving the results in a format suitable for subsequent analysis and modeling.

Theoretical rationale:

A batch pipeline is a sequence of data processing operations performed on a pre-accumulated data set. Unlike stream processing, where events are processed as they arrive, batch processing operates on data over a specific period: a day, a week, a month, or another interval. This approach is widely used in building analytics marts, preparing data for machine learning, generating reports, and regularly recalculating metrics.

In big data tasks, batch pipelines are typically built with a multi-tiered storage organization in mind. One common approach involves distinguishing raw , cleaned, and curated layers . The raw layer contains the original data in a form as close to the original source as possible. In this layer, the data is typically unmodified to preserve the ability to reprocess and validate the original records.

cleaned layer is designed to store cleaned and normalized data. This stage includes removing or processing invalid records, converting data types, filling or eliminating gaps, standardizing values, checking ranges, and eliminating obvious errors. Data in the cleaned layer is already suitable for further analysis, but is not yet necessarily presented in the final analytics display.

curated layer contains prepared analytical datasets, data marts, and aggregated tables. This data is focused on specific tasks: analyzing user behavior, calculating activity metrics, preparing features for machine learning models, building reports, or performing data mining. At this level, data typically has a clear structure, consistent types, and pre-calculated metrics.

An important property of a batch pipeline is reproducibility. When run again on the same input data, the pipeline should produce the same results. This requires careful catalog organization, da-

ta schema control, documented transformation logic, and the use of predictable processing rules. Reproducibility is especially important when exploring mathematical models and comparing data mining results.

Another important property is idempotency. An idempotent pipeline can be safely rerun without introducing duplicates, incorrect rewrites, or the accumulation of erroneous results. This is typically achieved through output rewrite modes, partition control, period-specific data processing, and clear separation of input, intermediate, and final data sets.

Apache Spark is well suited for building batch pipelines because it allows reading data from HDFS and other storage, performing distributed cleansing, filtering, aggregation, joining tables, generating features, and storing results in efficient formats such as Parquet. The Spark Data-Frame API and Spark SQL allow you to describe processing logic as a sequence of understandable operations.

In this practical, students build a simple batch data processing pipeline: they load source events into a raw layer, perform cleansing and normalization in a cleaned layer, and then create a curated dataset with aggregated metrics. This work allows them to understand the full data preparation cycle: from the source file to an analytical display suitable for subsequent data mining.

Hardware and Materials:

Computer or laptop with internet access; Deployed Big Data learning environment from previous hands-on work; Running Hadoop, HDFS, and Apache Spark components; Access to JupyterLab or a terminal with PySpark; Test CSV or JSON files with event, user, or transaction data; HDFS directory or local directory for organizing raw, cleaned, and curated layers; Software: Apache Spark, PySpark, Spark SQL, Hadoop, HDFS, Python 3.x, JupyterLab, Docker, Docker Compose, an operating system terminal

Typical tasks:

1. Launch the Big Data training environment and check the functionality of Apache Spark, HDFS, and JupyterLab. Ensure that HDFS directories are accessible and that PySpark code can be executed.
2. Create a new notebook for this practical assignment. In the first text box, enter the assignment number, topic, objective, and a brief description of the environment used.
3. Create a SparkSession with the application name `BigDataPractice13`. Check the Spark version and ensure the session was created without errors.
4. the batch pipeline in HDFS or a local file system: `raw`, `cleaned`, `curated`, `logs`, and `reports`. Explain the purpose of each directory.
5. Prepare original CSV file `events_raw.csv`, containing fields `event_id`, `user_id`, `event_type`, `category`, `value`, `event_time`, `device`, `region`. The file must include at least 50 records, including several lines with missing, incorrect, or duplicate values.
6. Load the `events_raw.csv` file into the raw layer. Check for the file in the raw directory and output the first few rows of the raw data.
7. Read the raw layer data into a Spark DataFrame. Print the data schema and the first rows. Determine which fields require type conversion or cleanup.
8. Perform a quality check of the source data: determine the number of rows, the number of unique event identifiers, the presence of gaps, duplicates, and incorrect values in the numeric value field.

9. Perform data cleanup: remove obvious duplicates, exclude rows without required identifiers, handle missing values, and filter out invalid values of the `value` field .
10. Convert data types. Convert the `value` field to a numeric type, and convert the `event_time` field to a date or time type, if supported by the prepared dataset.
11. Add technical columns that are useful for further processing: processing date, event date, record validity indicator, or data source.
12. Save the cleaned dataset to a cleaned layer in Parquet format . Verify that the data is written correctly and can be read again.
13. Read the data from the cleaned layer. Re-render the schema and ensure that the data types are preserved correctly.
14. Perform basic data aggregation: calculate the number of events, the sum of values, the average value, the minimum and maximum value of the `value` field by category.
15. Build an aggregated table by region and event type. Calculate the number of events and the number of unique users for each region and `event_type` pair .
16. Create a custom data mart. For each `user_id` , calculate the number of events, the sum of values, the average event value, the number of unique categories, and the first and last times of events.
17. Add derived features to your custom showcase, such as average event value, activity frequency, high activity feature, or number of days between first and last event.
18. Save the final user storefront as a curated layer in Parquet format . Verify that the data is written and readable.
19. Register the curated dataset as a temporary Spark SQL view. Run several analytical SQL queries against the resulting display, for example, find users with the most events or categories with the highest total value.
20. pipeline reruns with output overwrite mode. Ensure that reruns do not result in duplicate results.
21. Test the idempotency of the pipeline : re-run all processing steps on the same source file and compare the row counts and aggregated metrics in the resulting curated set.
22. Create a simple pipeline diagram : raw → cleaned → curated . For each layer, indicate what operations are performed and what data is stored.
23. Add a data quality control table to the report: number of input rows, number of removed duplicates, number of rows with gaps, number of rows after cleaning, number of rows in the final display.
24. Use the `explain` method for a single aggregation or data storage operation. Analyze the steps Spark takes to build the resulting dataset.
25. Compare storing cleaned and curated data in CSV and Parquet . Explain why Parquet is preferable for final analytical datasets .
26. Formulate a conclusion about the role of batch pipelines in preparing data for subsequent intelligent analysis and machine learning.
27. Prepare a report on the practical work. The report should include the directory structure, PySpark code snippets, data quality check results, a description of the cleaning operations, final aggregated tables, a pipeline diagram, and conclusions on the reproducibility of the processing.

Test questions:

1. What is a batch data processing pipeline?
2. How is batch processing different from stream processing?
3. What tasks is batch used for? processing in Big Data systems?
4. What is the raw layer ?
5. Why is data in a raw layer usually saved in its original form?
6. What is the cleaned layer ?

7. What operations are usually performed when going from raw to cleaned ?
8. What is a curated layer ?
9. Why are analytical data marts created?
10. Why is it important to separate raw, cleaned, and final data?
11. What is data processing reproducibility?
12. What does it mean for a batch pipeline to be idempotent ?
13. shouldn't rerunning a pipeline result in data duplication?
14. What data quality errors should be identified during the data cleaning phase?
15. Why is it important to control for missing values?
16. Why is it necessary to check for duplicates in source data?
17. How does data type conversion affect the accuracy of analytics?
18. Why is the Parquet format suitable for cleaned and curated layers?
19. How is the Spark DataFrame API used when building a batch pipeline ?
20. How can Spark SQL be used to validate the resulting display?
21. What metrics can be used to control the quality of a batch pipeline ?
22. Why is it important to record the number of lines at each stage of processing?
23. What features can be formed for subsequent Data Mining based on user events?
24. How does the batch pipeline relate to data preparation for Spark MLlib ?
25. Why is a reproducible pipeline important for Big Data Analysis and Mining research tasks ?

PRACTICAL WORK #14

Topic : Structured Streaming in Apache Spark

Objective:

To become familiar with the principles of stream data processing in Apache Spark. To master the creation of streaming DataFrame , reading streaming data from a file source or socket, performing simple transformations, aggregating events across time windows, using checkpointing , and storing stream processing results. Develop practical skills in developing a simple streaming pipeline in Spark Structured. Streaming .

Theoretical justification:

Structured Streaming is a component of Apache Spark designed for processing streaming data. It allows you to work with continuously incoming events using familiar DataFrame API and Spark SQL. Unlike batch processing, where analysis is performed on a pre-accumulated dataset, stream processing is focused on data that arrives gradually and must be processed as it appears.

Streaming data is widely used in modern information systems. This includes user events on a website, transactions, application logs, sensor data, network events, equipment telemetry, messages from event brokers, and other continuous sources of information. In Big Data Analysis and Mining, stream processing is used for near-term data analysis. Real-time analytics, monitoring, anomaly detection, building operational data displays, and preparing data for subsequent analysis.

The main idea of Structured Streaming treats the data stream as an unbounded table, with new rows continually added. The user describes transformations the same way they would with a regular DataFrame : selecting columns, filtering rows, performing grouping, aggregation, and writing the result. Spark automatically organizes the periodic execution of microbatches , processes the new data, and updates the result.

In Structured Streaming The concept of a data source plays an important role in streaming . A source can be a file directory, a socket, Kafka , or rate. source or another streaming source. In training tasks, a file source or socket is often used, as they allow one to simulate data inflow without complex external infrastructure. With a file source, Spark monitors new files in the directory and processes them as new data chunks.

Time is often used in streaming event processing. A distinction is made between event time and processing time. Event time refers to the time the event actually occurred, recorded in the data itself. Processing time refers to the time the event was processed by the system. The distinction between these concepts is important because data may arrive with a delay, out of order, or duplicated.

Window processing allows you to group streaming events by time intervals. For example, you can count the number of events per minute, calculate the sum of transactions per 10 minutes, or determine the number of active users per hour. Spark uses the window function for this purpose , which creates time windows based on the event field.

When working with streaming data, late events may arrive—that is, events that belong to a past time interval but arrive in the system later. A watermark is used to manage such situations . A watermark defines an acceptable event latency and allows Spark to limit the amount of state that must be stored to correctly process windowed aggregations.

Checkpointing is a necessary element of reliable stream processing. A checkpoint stores the service state of a request, information about processing progress, and intermediate data necessary for failure recovery. Without checkpointing A streaming application cannot reliably continue execution after stopping or experiencing an error.

The stream processing result is written to a sink , or data receiver. The sink can be the console, a file, memory, Kafka , or another storage system. For academic projects, it's convenient to use the console sink to monitor the results during execution and a file sink to save the results.

In this practical work, students create a simple streaming pipeline : they simulate the arrival of events, read them using Structured Streaming , perform filtering and aggregation, calculate metrics by time windows, configure checkpointing , and analyze the behavior of a streaming request.

Equipment and materials:

Computer or laptop with internet access; Deployed Big Data training environment from previous practical work; Running Apache Spark components and, if necessary, HDFS; Access to JupyterLab or a terminal with PySpark;

Directory for simulating the incoming streaming files; Test JSON or CSV files with events;

Terminal for running the data source via socket , if socket is used source

Directory for checkpoint and stream processing results

Software: Apache Spark, PySpark , Spark Structured Streaming , Hadoop , HDFS, Python 3.x, JupyterLab , Docker , Docker Compose , an operating system terminal

Typical tasks:

1. Launch the Big Data training environment and check Apache Spark and JupyterLab . Ensure that PySpark code runs successfully in the selected environment.
2. Create a new notebook for this practical assignment. In the first text box, enter the assignment number, topic, objective, and a brief description of the environment used.

3. Create a SparkSession with the application name `BigDataPractice14` . Check the Spark version and ensure the session was created without errors.
4. Create working directories for stream processing: the `stream_input` directory , the `stream_output` directory, and the `stream_checkpoint` directory .
5. Prepare several small JSON files with events. Each recording should contain fields `event_id` , `user_id` , `event_type` , `category` , `value` , `event_time` , `device` , `region` .
6. Define an explicit schema for the input data. Specify types for identifiers, categories, numeric values, and event times.
7. Create a streaming `DataFrame` that reads new JSON files from the `stream_input` directory . Specify the data schema and source format.
8. Derive the streaming `DataFrame` . Verify that Spark correctly recognizes the stream structure.
9. Perform simple filtering of streaming data: keep only events whose `value` is greater than a specified threshold.
10. Add a calculated column, such as `value_level` , that evaluates to `high` if `value` exceeds a specified threshold, and `normal` otherwise.
11. Configure logging of filter results to console sink . Start streaming query and gradually add new JSON files to the `stream_input` directory .
12. Observe how Spark processes new files as they appear. Capture several snippets of the output in a report.
13. Perform a streaming aggregation on the `category` field : count the number of events and the sum of the `value` field for each category.
14. complete or update output mode for aggregated data. Explain why append mode is not always appropriate for aggregations .
15. Convert the `event_time` field to a timestamp type if it was loaded as a string. Check that the conversion is correct.
16. Implement windowed aggregation: calculate the number of events and the sum of the values over time windows, such as every 1 or 5 minutes.
17. Add grouping by window and category to determine category activity over time.
18. Configure a watermark for the `event_time` field , for example, with an acceptable delay of several minutes. Explain the problem the watermark solves .
19. Set up checkpoint location For a streaming query, make sure that service files appear in the checkpoint directory after running the query.
20. Stop streaming query and rerun it with the same checkpoint directory. Explain why checkpointing is important for restoring the processing state.
21. Save the output of one of the stream processings to a file sink in Parquet format , if supported by the selected configuration.
22. Read the saved result as a normal batch `DataFrame` and run a simple analytical query on it.
23. Compare the batch pipeline from the previous lab and the streaming pipeline in the current lab. Describe the differences in data sources, processing mode, state storage, and result logging.
24. Diagram the streaming pipeline : Event source → streaming `DataFrame` → filtering → windowed aggregation → checkpoint → sink .
25. Formulate a conclusion about the applicability of Structured Streaming for monitoring tasks, near Real-time analytics and data preparation for intelligent analysis.
26. Prepare a report on the practical work. The report should include fragments of PySpark code, an input data schema, examples of input events, stream processing results, a description of checkpointing , an example of window aggregation, and conclusions about the application of Structured Streaming .

Test questions:

1. What is Spark Structured Streaming?
2. How is stream processing different from batch processing?
3. What types of data can be processed as streaming events?
4. What does it mean to represent a stream as an unconstrained table?
5. What is streaming? DataFrame ?
6. What data sources can Structured use? Streaming ?
7. Why is a file source useful for educational streaming data simulation?
8. What is a sink in stream processing?
9. What data sinks can be used in Structured? Streaming ?
10. What is streaming? query ?
11. What output modes are supported in Structured? Streaming ?
12. append mode different from update and complete ?
13. update or complete modes often used for aggregated streaming data ?
14. What is an event? time ?
15. What is processing? time ?
16. Why event time and processing time may differ?
17. What is late? events ?
18. What is windowing used for in streaming analytics?
19. Which one? role performs window function in Spark Structured Streaming?
20. What is a watermark ?
21. How does watermark help limit the state of stream processing?
22. What is checkpointing in Structured Streaming?
23. Why is checkpointing important for disaster recovery?
24. What data is stored in the checkpoint directory?
25. How does stream aggregation differ from regular batch aggregation?
26. Why can stream processing be useful for anomaly detection?
27. How is Structured Streaming related? With DataFrame API?
28. file-based streaming simulation ?
29. When is it appropriate to use Kafka for stream processing ?
30. How can stream processing results be used in Big Data Analysis and Mining tasks?

PRACTICAL WORK #15

Topic: Data Mining on Spark: Classification and Model Quality Assessment

Objective:

To become familiar with the use of Apache Spark and Spark MLlib to solve classification problems on big data. To master the full cycle of building a classification model: loading and preparing data, processing categorical and numerical features, generating a feature vector, splitting the data into training and test sets, training the model, obtaining predictions, and assessing the classification quality using accuracy , precision , recall , and F1 metrics.

Theoretical rationale:

Classification is one of the main tasks of data mining. Its goal is to assign objects to one of several predetermined classes based on a set of characteristics. In Big Data, classification is used to analyze user behavior, detect fraudulent transactions, predict customer churn, filter events, recognize activity patterns, diagnose technical conditions, and solve other applied problems.

Unlike local machine learning, classification on big data requires consideration of sample size, data structure, computational costs, distributed storage, and parallel processing. Apache Spark provides the Spark MLlib library, designed for building scalable machine learning pipelines. It enables feature preparation, model training, and quality assessment in a distributed environment.

In Spark MLlib, training data is typically represented as a `DataFrame`. One column contains the target variable, or label, and the other column contains a feature vector, or features. Before training a model, the source data must be prepared: gaps must be removed or processed, data types must be converted, categorical features must be encoded, numerical features must be scaled if necessary, and individual features must be combined into a single vector.

The `StringIndexer` and `OneHotEncoder` components are often used to process categorical features in Spark MLlib. `StringIndexer` converts string categories to numeric indices, and `OneHotEncoder` converts category indices to binary vectors. `VectorAssembler` is used to combine multiple features into a single vector. These steps are conveniently combined into a `Pipeline`, which defines the sequence of transformations and model training.

Spark MLlib pipeline allows you to formalize the entire data processing process: from the original `DataFrame` to the trained model. It can include feature transformers, a classification algorithm, and subsequent processing steps. This approach improves the reproducibility of experiments and reduces the risk of discrepancies between the processing of training and test data.

Various algorithms can be used to solve classification problems in Spark MLlib: `LogisticRegression`, `DecisionTreeClassifier`, `RandomForestClassifier`, `GBTCClassifier`, and others. In academic work, it is advisable to use one or two algorithms, such as logistic regression and decision tree, to compare the quality of the models and understand the specifics of their application.

After training, the model is used to obtain predictions on the test set. Classification quality is assessed using metrics. Accuracy measures the proportion of correctly classified objects. Precision reflects the proportion of correct positive predictions among all objects classified by the model as positive. Recall measures the proportion of positive objects found among all true positive objects. The F1 score is the harmonic mean of precision and recall and is especially useful for imbalanced classes.

When evaluating a model, it's important to consider not only the numerical values of the metrics but also the meaning of the errors. In some tasks, false positives and false negatives have different costs. For example, in fraud detection, missing a fraudulent transaction may be more critical than erroneously checking a normal transaction. Therefore, interpreting a model's quality should be based on the subject area and the purpose of the classifier.

In this practical, students build a classification model using Spark MLlib, prepare features, train the model, obtain predictions, and calculate quality metrics. This course provides an understanding of how data mining methods are implemented in the distributed Apache Spark environment and how classification results can be used in big data analysis tasks.

Equipment and materials:

Computer or laptop with internet access; Deployed Big Data training environment from previous practical work; Running Hadoop, HDFS, and Apache Spark components

; Access to JupyterLab or a terminal with PySpark;

Test CSV or Parquet file with data for classification; Dataset with numerical and categorical features and a target variable; HDFS directory or local directory for storing source data, intermedi-

ate results, and

the model; Software: Apache Spark, PySpark , Spark SQL, Spark MLlib , Hadoop , HDFS, Python 3.x , JupyterLab , Docker Compose , an operating system terminal

Typical tasks:

1. Launch the Big Data training environment and check Apache Spark, HDFS, and JupyterLab . Ensure that the Spark MLlib libraries are available .
2. Create a new notebook for this practical assignment. In the first text box, enter the assignment number, topic, objective, and a brief description of the environment used.
3. Create a SparkSession with the application name BigDataPractice15 . Check the Spark version and ensure the session was created without errors.
4. Prepare or load a dataset for classification. The dataset should contain several features and a target variable, such as a label feature indicating whether an object belongs to one of the classes.
5. As a training case, one can use user event data, where the target variable indicates whether the user is active, whether he or she is prone to the target action, or whether the event belongs to a certain class.
6. Load data into a Spark DataFrame from a CSV or Parquet file. Display the first rows and the data schema. Check that the numeric and categorical column types are correct.
7. Perform a primary data analysis: determine the number of rows, the number of columns, the number of unique classes in the target variable, and the distribution of objects across classes.
8. Check for missing values in the features and target variable. Remove or fill in missing values based on the meaning of the data.
9. Select the features that will be used to train the model. Divide them into numerical and categorical.
10. Transform categorical features using StringIndexer . For each categorical column, create an indexed numeric column.
11. Apply OneHotEncoder to indexed categorical features if required by the chosen model.
12. Generate the final feature vector using VectorAssembler . Include the numerical features and the transformed categorical features.
13. If the target variable is represented by string values, convert it to a numeric label column using StringIndexer .
14. Split the data into training and test sets, for example, 70/30 or 80/20. Record the random value used. seed for reproducibility of the result.
15. Create a classification model, such as LogisticRegression , DecisionTreeClassifier , or RandomForestClassifier . Specify the basic model parameters.
16. Combine the feature processing steps and the model in a Pipeline . Train the Pipeline on the training set.
17. Obtain model predictions on the test set. Output columns with the actual label, predicted class, probability, and key features.
18. Calculate the quality of the model using MulticlassClassificationEvaluator . Evaluate accuracy , F1 , weightedPrecision , and weightedRecall .
19. For binary classification, additionally use BinaryClassificationEvaluator and calculate areaUnderROC or areaUnderPR .
20. Build a confusion matrix using the Spark DataFrame API or Spark SQL: group the results by actual label and predicted class.
21. Analyze which classes the model recognizes better and which worse. Explain the possible causes of classification errors.
22. Train a second model of a different type, such as DecisionTreeClassifier or RandomForestClassifier . Compare its performance with the first model using the selected metrics.

23. Change one or more model parameters and repeat training. Analyze how changing the parameters affects classification performance.
24. Check for class imbalance. Explain why accuracy may be an insufficient quality metric for imbalanced data.
25. Save the trained model or PipelineModel to the local file system or HDFS, if supported by the training environment. Check if the model can be reloaded.
26. Save the model predictions in Parquet format for later analysis. Verify that the results were successfully written and can be read again.
27. Formulate a conclusion about the applicability of the selected model to the classification problem under consideration. Indicate its strengths, limitations, and potential areas for improvement.
28. Prepare a report on the practical work. The report should include PySpark code fragments, a description of the dataset, the DataFrame schema, the feature preparation steps, model parameters, quality metrics, a confusion matrix, and conclusions from the classification results.

Test questions:

1. What is classification in data mining problems?
2. How is classification different from regression?
3. What is the target variable in a classification problem?
4. What is a feature in a machine learning problem?
5. Why is it necessary to prepare data before training a model?
6. StringIndexer used for in Spark MLlib ?
7. What is OneHotEncoder used for in Spark MLlib ?
8. VectorAssembler used for ?
9. Why do Spark MLlib models typically expect a features column ?
10. Why should the target variable often be represented as a label column ?
11. What is Pipeline in Spark MLlib ?
12. What are the benefits of using Pipeline when building a model?
13. Why is data divided into training and test samples?
14. Why is it important to capture randomness? seed when splitting data?
15. What such LogisticRegression in Spark MLlib ?
16. What is DecisionTreeClassifier ?
17. What is RandomForestClassifier ?
18. How does the model make predictions for test data?
19. What does the accuracy metric show ?
20. What does the precision metric show ?
21. What does the recall metric show ?
22. What does the F1 measure show?
23. Why might the F1 score be more useful than accuracy when classes are imbalanced?
24. What does the confusion matrix show?
25. MulticlassClassificationEvaluator used for ?
26. BinaryClassificationEvaluator used for ?
27. What is areaUnderROC ?
28. Why is it important to analyze not only metrics but also the nature of model errors?
29. How does Spark MLlib help scale model training on big data?
30. How can classification results be used in Big Data Analysis and Mining tasks?

Practical Work #16

Topic: Data Mining on Spark: Clustering and Association Rule Search

Objective:

To become familiar with the use of Apache Spark and Spark MLlib for solving unsupervised learning problems and finding patterns in big data. To master the preparation of features for clustering, the application of the k- means algorithm , the evaluation and interpretation of the resulting clusters, and the use of the FP- growth algorithm for finding frequent itemsets and association rules. To develop practical skills in intelligent big data analysis in a distributed Apache Spark environment.

Theoretical rationale:

Data mining encompasses a set of data mining methods aimed at identifying hidden patterns, structures, relationships, and useful insights in large data sets. Unlike simple analytical queries, data mining methods allow not only to calculate aggregated indicators but also to discover groups of similar objects, typical behavior patterns, associative links between events, and potentially useful dependencies.

One of the most common tasks in data mining is clustering. Clustering is an unsupervised learning method and is used when objects lack predefined class labels. The goal of clustering is to divide objects into groups such that objects within a cluster are similar to each other, while objects from different clusters differ in their characteristics.

In big data analysis, clustering can be used for user segmentation, product grouping, customer behavior analysis, identifying typical activity profiles, event exploration, detecting atypical groups of objects, and pre-processing data for subsequent modeling. In Spark MLlib, one of the basic clustering algorithms is k- means .

The k- means algorithm partitions objects into a predetermined number of clusters. Each cluster is described by a center, or centroid . The algorithm iteratively assigns objects to the closest centroid , then recalculates cluster centers and repeats the process until the result stabilizes or the specified number of iterations is reached. The quality of clustering depends on the chosen number of clusters, the scale of the features, the initialization of the centers, and the data structure.

Before using k- means, numerical features must be prepared. If the dataset contains categorical features, they must be converted to numerical form. Numerical features often need to be scaled, as features with large values can have a disproportionate effect on inter-feature distances. Spark MLlib uses StringIndexer , OneHotEncoder , VectorAssembler , StandardScaler , and other components for feature preparation.

Assessing clustering quality differs from evaluating supervised models, as proper class labels are usually missing. The intra-cluster sum of squared distances (Silhouette) can be used to analyze the results. score , comparison of results with different numbers of clusters, and meaningful interpretation of clusters. In educational tasks, it is important not only to obtain cluster numbers but also to explain how clusters differ from each other.

Another important task in data mining is the discovery of association rules. Association rules allow us to identify relationships between objects or events that frequently occur together. A classic example of this problem is shopping cart analysis: if a user purchases one item, the probability of purchasing another can be estimated. However, association rules are applicable not only to retail but also to the analysis of events, user actions, website section visits, feature sets, and category co-occurrence.

To find frequent itemsets in Spark MLlib, the FP- growth algorithm is used . It allows finding frequent Itemsets , which are sets of elements that occur in transactions at least within a specified support threshold. Frequent itemsets are used to construct association rules of the form "if a transaction contains item A, then with some probability it also contains item B."

The main metrics for association rules are support , confidence , and lift . Support indicates how frequently a set of elements occurs in the data. Confidence indicates how often a rule's consequence occurs given its condition. Lift assesses the extent to which the relationship between the condition and consequence is stronger than chance alone. If lift is greater than 1, elements co-occur more often than would be expected given independence.

In this practical, students complete two related data mining tasks. The first involves clustering objects based on pre-defined features. The second involves finding frequent sets of elements and association rules in transactional data. This structure allows for the exploration of two important areas of data mining: identifying groups of similar objects and discovering patterns of co-occurrence.

Equipment and materials:

Computer or laptop with internet access; Deployed Big Data training environment from previous practical work; Running Hadoop , HDFS, and Apache Spark components ; Access to JupyterLab or a terminal with PySpark; Test CSV or Parquet file with data for clustering ; Test set of transactional data for finding association rules; HDFS directory or local directory for storing source data, intermediate results, and final models; Software : Apache Spark, PySpark , Spark SQL, Spark MLlib , Hadoop , HDFS, Python 3.x, JupyterLab , Docker Compose , an operating system terminal

Typical tasks:

1. Launch the Big Data training environment and check the functionality of Apache Spark, HDFS, and JupyterLab . Ensure that the Spark MLlib components required for clustering and association rule mining are available.
2. Create a new notebook for this practical assignment. In the first text box, enter the assignment number, topic, objective, and a brief description of the environment used.
3. Create a SparkSession with the application name BigDataPractice16 . Check the Spark version and ensure the session was created without errors.
4. Prepare or load a dataset for clustering. The dataset should contain an object identifier and several numeric features, such as the number of user events, the total value of actions, the average event value, the number of unique categories, and the duration of activity.
5. If necessary, use the custom data mart created in Practice #13 or #15. Load it from Parquet or CSV format into a Spark DataFrame .
6. Print the first rows and the schema of the dataset. Check that the clustering features are represented as numeric data types.
7. Perform a primary feature analysis: determine the number of objects, the presence of missing values, the minimum, maximum, and average values for numeric columns.
8. Process missing values. If necessary, remove rows with critical gaps or fill them with valid values.
9. Select the numerical features to be used for clustering. Justify your choice of features in terms of the purpose of the problem being studied.
10. Generate a feature vector using VectorAssembler . The resulting column should contain a single features vector used by the clustering algorithm.

11. Scale features using StandardScaler if features have significantly different value ranges. Compare the original and scaled features.
12. Train a k- means model for a given number of clusters, say $k = 3$. Obtain the assigned cluster number for each feature.
13. Find the cluster centers. Analyze the feature values that are characteristic of each cluster.
14. Calculate the number of objects in each cluster. Determine whether any clusters are too small or too large.
15. Evaluate the quality of clustering using ClusteringEvaluator And Silhouette score metrics . Explain the meaning of the obtained value.
16. Repeat training k- means for several values of k , such as 2, 3, 4, and 5. Compare the silhouette values score and distribution of objects across clusters.
17. Select the most appropriate number of clusters based on the quality metric and meaningful interpretation of the result.
18. Generate the final DataFrame with the object ID, original features, and cluster number. Save the clustering result in Parquet format .
19. Provide a meaningful interpretation for each cluster. For example, identify groups of users with low, medium, and high activity, or groups of objects with different behavioral profiles.
20. Prepare a transactional dataset for association rule mining. Each row should contain a transaction ID and a list of elements. For example, a list of user event categories, a list of items in an order, or a list of actions in a session.
21. Load the transactional data into a Spark DataFrame . Verify that one of the columns contains an array of elements suitable for the FP- growth algorithm .
22. If the source data is presented as separate rows `transaction_id` , `item` , group them by `transaction_id` and form a list of items using an aggregate function.
23. Apply the FPGrowth algorithm from Spark MLlib . Specify the minimum support and confidence values . Explain how these parameters affect the number of rules found.
24. Get frequent item sets. Sort them by descending frequency and output the most frequent items.
25. Get association rules. For each rule, output the antecedent , consequent , confidence , and lift .
26. Analyze the rules you've found. Determine which connections between elements are the strongest and most meaningfully interpretable.
27. the minSupport and minConfidence parameters . Repeat the rule search and compare the number of frequent sets and association rules found.
28. Save the found frequent sets and association rules in CSV or Parquet format for later analysis.
29. Formulate a conclusion about the difference between clustering and association rule mining as Data Mining tasks.
30. Prepare a report on the practical work. The report should include PySpark code fragments, a description of the source data, feature preparation steps, clustering results, metric values, cluster interpretation, found frequent sets and association rules, and conclusions on the applicability of data mining methods in Spark.

Test questions:

1. What is Data Mining?
2. What is clustering?
3. Why is clustering considered unsupervised learning?
4. For what big data analysis tasks can clustering be applied?
5. What is the basic idea of k- means algorithm ?
6. What is a cluster centroid ?

7. Why does k- means require pre-specifying the number of clusters?
8. Why is feature scaling important when using distance-based algorithms?
9. What is VectorAssembler used for in Spark MLlib ?
10. StandardScaler used for ?
11. What does the silhouette show? score ?
12. Why is it important to consider not only the metric but also the meaningful interpretation of clusters when assessing clustering?
13. What features can be used to segment users?
14. What does it mean when a cluster is too small in the clustering result?
15. What does it mean when a cluster is too large in clustering results?
16. How can clustering results be used in applied problems?
17. What is transactional data?
18. What is a frequent set of elements?
19. What is an association rule?
20. What is the basic idea behind association rule mining?
21. growth algorithm used for ?
22. What does support show ?
23. What does confidence show ?
24. What does lift show ?
25. How to interpret a rule with lift greater than 1?
26. How does changing minSupport affect the number of frequent sets found?
27. How does changing minConfidence affect the number of rules found?
28. How is clustering different from association rule mining?
29. How does Spark MLlib help scale Data Mining tasks?
30. What are the limitations of clustering and association rule mining methods when working with big data?

CONCLUSION

The practical work presented within the "Big Data Analysis and Mining" course is an important part of the educational process and aims to develop students' practical skills in storing, distributed processing, analyzing, and intelligently extracting patterns from big data. Completing the assignments allows students to sequentially progress through the main stages of working with Big Data: from deploying a learning environment and mastering HDFS to building batch and streaming pipelines , optimizing Spark applications, and applying data mining methods in a distributed environment.

During practical work, students master the key technologies and tools of the modern big data ecosystem: Hadoop , HDFS, MapReduce , Apache Spark, PySpark , Spark SQL, Spark DataFrame API, Spark Structured Streaming and Spark MLlib . Particular attention is paid not only to running ready-made tools but also to understanding their operating principles: distributed file storage, the HDFS block structure, the MapReduce model , lazy execution of calculations in Spark, shuffle operations , partitioning , caching, table joins, window functions, and optimization of computational processes.

A set of practical exercises develops skills in working with various data formats, including CSV, JSON, and Parquet , cleaning and transforming data, building analytical data marts, preparing features, training classification models, performing clustering, and finding association rules. This approach allows students to experience the full lifecycle of a big data analysis project: from obtaining initial data to interpreting modeling results and assessing the quality of the resulting solutions.

Practical assignments promote the development of systems and algorithmic thinking, the ability to work with distributed computing environments, analyze the performance of software

solutions, identify model limitations, and justify the choice of data processing methods. The focus of the assignments on Hadoop and Apache Spark enhances the applied focus of the course and brings the learning process closer to the real-world challenges of modern data engineering and intelligent analysis of large data sets.

Completing practical work in the "Big Data Analysis and Mining" course ensures a practice-oriented assimilation of the course's theoretical principles, promotes the development of OPK-2 and PC-6 competencies, and prepares students to study mathematical models, develop and implement big data analysis algorithms, and apply modern software tools in research and professional activities.

EDUCATIONAL, METHODOLOGICAL AND INFORMATIONAL SUPPORT OF THE DISCIPLINE

1. Reis J., Housley M. Fundamentals of Data Engineering: Plan and Build Robust Data Systems. Sebastopol: O'Reilly Media, 2022. 446 p. ISBN 9781098108304.
2. Dehghani Z. Data Mesh: Delivering Data-Driven Value at Scale. Sebastopol: O'Reilly Media, 2022. 270 p. ISBN 9781492092391.
3. Avila JK Snowflake: The Definitive Guide: Architecting, Designing, and Deploying on the Snowflake Data Cloud. Sebastopol: O'Reilly Media, 2022. 400 p. ISBN 9781098103828.
4. Menon P. Data Lakehouse in Action: Architecting a Modern and Scalable Data Analytics Platform. Birmingham: Packt Publishing, 2022. 206 p. ISBN 9781801815932.
5. Huyen C. Designing Machine Learning Systems: An Iterative Process for Production-Ready Applications. Sebastopol: O'Reilly Media, 2022. 386 p. ISBN 9781098107963.