Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інфокомунікацій
(повна назва)

Кафедра Інформаційно-мережної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження розподіленої системи обробки даних у реальному часі

(тема)

Виконав:
здобувач 2 року навчання,
групи ІМІм-23-2
Петрачков М.О.
(прізвище, ініціали)

Спеціальність 172 Електронні комунікації та радіотехніка
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Інформаційно-мережна інженерія
(повна назва освітньої програми)

Керівник: ст.викл. Калюжний М.М.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ________ Безрук В.М.
(підпис) (прізвище, ініціали)

2025 р.

Не містить відомостей, заборонених до відкритого публікування

Студент ______________ / Петрачков.М.О.
(підпис) (прізвище та ініціали)

Керівник ______________ / Калюжний М.М. /
(підпис) (прізвище та ініціали)

Харківський національний університет радіоелектроніки

Факультет Інфокомунікацій

Кафедра Інформаційно-мережної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 172 Електронні комунікації та радіотехніка

(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформаційно-мережна інженерія

(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри ________________

(підпис)

« 28 » жовтня 2024 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Петрачкову Максиму Олеговичу

(прізвище, ім'я, по батькові)

1. Тема роботи Побудова розподіленої системи обробки даних у реальному часі

затверджена наказом по університету від« 28» жовтня 2024 р. № 1148 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 23 січня 2025 р.

3. Вхідні дані до роботи Дослідити процес побудови, архітектуру та принципи роботи розподіленої обчислювальної системи реального часу на базі платформи Apache Spark . Виконати дослідження процесу розгортання Apache Spark. На базі цього розглянути можливість виконання завдань MapReduce з Python та Scala до датасету з Apache Spark. Дослідити стратегії оптимізації Apache Spark.

4. Перелік питань, що потрібно опрацювати у роботі Вступ

1 Розподілені обчислення та Big Data

2. Розгортання та налаштування розподіленої обчислювальної системи на основі Apache Spark

3. Виконання завдань MapReduce у середовищі Apache Spark

4. Тестування та моніторинг продуктивності розподіленої обчислювальної системи на основі Apache Spark

Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп’ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) слайди презентації в форматі Power Point (Назва та мета роботи, обробка Big Data, Map Reduce, Apache Spark як засіб обробки Великих даних, dиконання завдань MapReduce у середовищі Apache Spark, тестування продуктивності розподіленої обчислювальної системи на основі Apache Spark, висновки)

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Вступ		виконано
2	Розподілені обчислення та Big Data		виконано
3	Розгортання та налаштування розподіленої обчислювальної системи на основі Apache		виконано
4	Виконання завдань MapReduce у середовищі Apache Spark		виконано
5	Тестування та моніторинг продуктивності		виконано
6	Висновки		виконано
7	Оформлення пояснювальної записки		виконано

Дата видачі завдання 28 жовтня 2024 р.

Здобувач ______________________________
(підпис)

Керівник роботи ________________ ст.викл. Калюжний М.М.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 69 с., 12рис., 14 джерел, 2 додатка.

Об'єкт дослідження – платформа Apache Spark у середовищі Hadoop.

Мета роботи – дослідження процесу розгортання розподіленої обчислювальної системи реального часу на базі Apache Spark.

Розглядається архітектура та принципи роботи інструментарію. Досліджуються механізми розгортання та конфігурування інструментарію на стаціонарному комп’ютері. Вивчаються закономірності та специфіка роботи з Big Data. Виконується огляд інструментів моніторингу процесів, які мають перебіг у кластері та стратегії оптимізації Apache Spark у середовищі Hadoop.

APACHE SPARK, HADOOP, BIG DATA, MAPREDUCE ОБЧИСЛЕННЯ, YARN, PROMETHEUS, GRAFANA, ПАРТИЦІОНУВАННЯ ДАНИХ, ОБЧИСЛЮВАЛЬНИЙ КЛАСТЕР.

THE ABSTRACT

Explanatory note: 69 p., 12 fig., 14 sources, 2 app.

The object of research is the Apache Spark platform in the Hadoop environment.

The aim of this paper is to study the process of deploying a real – time distributed computing system based on Apache Spark.

The architecture and operating principles of the toolkit are considered. Mechanisms for deploying and configuring tools on a desktop computer are investigated. The regularities and specifics of working with Big Data are studied. An overview of process monitoring tools running in the cluster and Apache Spark optimization strategies in the Hadoop environment is performed.

APACHE SPARK, HADOOP, BIG DATA, MAPREDUCE COMPUTING, YARN, PROMETHEUS, GRAFANA, DATA PARTITIONING, COMPUTING CLUSTER.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ.......... 9
ВСТУП.......... 10
1 РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ ТА BIG DATA 11
1.1 Сутність розподілених обчислень 11
1.2 BigData.......... 12
1.3 Принципи роботи з великими даними 13
1.4 MapReduce.......... 13
1.5 Приклади завдань, що ефективно вирішуються за допомогою MapReduce.......... 15
1.5.1 Word Count 15
1.5.2 Обробка логів рекламної системи 16
2 РОЗГОРТАННЯ ТА НАЛАШТУВАННЯ РОЗПОДІЛЕНОЇ ОБЧИСЛЮВАЛЬНОЇ СИСТЕМИ НА ОСНОВІ APACHE SPARK 17
2.1 Apache Spark як засіб обробки Великих даних 17
2.2 Архітектура Apache Spark 18
2.3 Підготовка оточення та режими роботи кластера Spark 19
2.3.1 Встановлення Java.......... 19
2.3.2 Установка Scala (необов'язково).......... 19
2.3.3 Завантаження та розпакування Apache Spark 19
2.3.4 Режими роботи кластера Spark 19
2.4 Розгортання кластера Spark (Standalone та Hadoop YARN) 20
2.4.1 Розгортання в режимі Standalone.......... 20
2.4.2 Розгортання Spark на кластері Hadoop (YARN) 21
2.5 Виконання завдань та налаштування параметрів 22
2.5.1 Виконання простих завдань 22
2.5.2 Spark-submit 23
2.6 Налаштування параметрів та оптимізація Spark.......... 23
2.7 Моніторинг, налагодження та безпека платформи.......... 24
3 ВИКОНАННЯ ЗАВДАНЬ MAPREDUCE У СЕРЕДОВИЩІ APACHE SPARK 26
3.1 MapReduce у Spark 26
3.2 Виконання завдань MapReduce у Spark 27

3.3 Типові операції MapReduce в Apache Spark 28
3.3.1 Трансформації в Apache Spark 28
3.3.2 Дії (Actions) в Apache Spark 31
3.4 Порівняння трансформацій та дій 34
3.5 Розширені приклади використання дій 34
3.6 Спільне використання трансформацій та дій 35
4 ТЕСТУВАННЯ ТА МОНІТОРИНГ ПРОДУКТИВНОСТІ РОЗПОДІЛЕНОЇ ОБЧИСЛЮВАЛЬНОЇ СИСТЕМИ НА ОСНОВІ APACHE SPARK 38
4.1 Інструменти моніторингу в Apache Spark 39
4.1.1 Spark Web UI 39
4.1.2 Ganglia 42
4.1.3 Prometheus та Grafana 45
4.2 Тестування продуктивності системи 46
4.2.1 тапи тестування продуктивності 47
4.2.2 Роль моніторингу у тестуванні продуктивності 47
4.3 Оптимізація та аналіз результатів моніторингу 49
4.4 Автоматизація тестування та моніторингу 52
ВИСНОВКИ 53
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 54
ДОДАТОК А ТЕЗИ КОНФЕРЕНЦІЇ 55
ДОДАТОК Б СЛАЙДИ ПРЕЗЕНТАЦІЇ 61

ПЕРЕЛІК СКОРОЧЕНЬ

CUDA – (Compute Unified Device Architecture) – Паралельна обчислювальна платформа і модель програмування;

HDFS – (Hadoop Distributed File System) – розподілена файлова система, призначена для зберігання та обробки великих об'ємів даних на кластерах обчислювальних вузлів;

API – (Application Programming Interface) інтерфейс прикладного програмування.

ВСТУП

Впровадження обчислювальних пристроїв у різні галузі діяльності людини, яке було розпочато у другій половині XX століття, кардинальним чином вплинуло на подальший розвиток технологій та суспільства взагалі. Це стосується, зокрема:

- підвищення продуктивності виробництв;
- розвитку наукових знань;
- появи принципово нових сфер діяльності та удосконалення вже існуючих;
- спрощення доступу до інформації;
- появи нових форматів комунікацій і т.д.

При цьому, синхронно до розвитку обчислювальної техніки та алгоритмів обчислень, розширювалося коло задач, які може бути вирішено на їх базі.

Врешті-решт, склалися умови, за яких класична модель обчислень у рамках архітектури фон Неймана може бути неефективною для рішення завдань різних класів, а саме [1]:

- коли існує необхідність обробки великих масивів даних за деякий визначений час;
- коли існує необхідність виконання складних обчислень у реальному масштабі часу.

Таким чином, у зазначених умовах було необхідно збільшити продуктивність обчислювальних систем, знаходячись у межах існуючого технологічного базису. Свого часу, дане завдання було вирішено за двома напрямками, а саме:

- впровадженням технологій паралельних обчислень, які передбачають виконання обчислювальних операцій на рівні одного вузла;
- застосуванням технологій розподілених обчислень, у рамках яких те чи інше обчислювальне завдання вирішується на багатьох розподілених вузлах.

У свою чергу, якщо за рахунок паралельних обчислень вдалося збільшити продуктивність окремого вузла, то за рахунок розподілених обчислень розробники та дослідники отримали можливість вирішувати складні комплексні завдання, які при класичному підході могли не мати рішення. Сьогодні розподілені обчислення використовуються для рішення широкого діапазону завдань у різних галузях. Отже, тематика кваліфікаційної роботи є актуальною.

1 РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ ТА BIG DATA

1.1 Сутність розподілених обчислень

Розподілені обчислення – це спосіб одночасного паралельного рішення одного і того ж обчислювального завдання кількома одночасними процесами, що дозволяє певної мірою зменшити час обчислень [1, 2].

Розподілені обчислення зазвичай застосовуються для вирішення складного або комплексного завдання, для якого використання класичних обчислювальних моделей неефективне.

У цілому, методи таких обчислення схожі на методи паралельного програмування, для яких властиво виконання обробки даних багатьма окремими потоками (зокрема, у рамках технології CUDA). Водночас, на відміну від паралельних обчислень, систему розподілених обчислення може бути створено на базі об'єднання кількох комп'ютерів. У цьому випадку комп'ютерна мережа перетворюється на потужний єдиний комп'ютер, який надає широкомасштабні ресурси на вирішення складних завдань.

На сьогодні систему розподілених обчислень може бути побудовано на базі:

− поєднання локально розміщених вузлів з утворенням обчислювального кластеру;

− поєднання вузлів, комунікації між якими забезпечують канали Всесвітньої мережі (grid-система);

− хмарних ресурсів.

Одним з основних напрямків застосування розподілених обчислень є обробка т. зв. Великих даних (BigData) [1].

Це, зокрема, стосується таких завдань, як:

− шифрування великих обсягів даних;

− виконування аналізу великих об’ємів даних у реальному часі;

− вирішення фізичних чи хімічних рівнянь з багатьма змінними;

− створення високоякісних тривимірних відеоанімацій;

− моделювання та прогноз перебігу складних процесів (природні явища, ядерні реакції тощо) і т.д.

− Розподілені системи, розподілене програмування та розподілені алгоритми – ще кілька термінів, що належать до розподілених обчислень.

1.2 BigData

Сам термін Big Data з'явився порівняно недавно. Google Trends показує початок активного використання словосполучення, починаючи з 2011 року [1].

У цілому, часто можна зустріти такі трактування даного терміну:

− Big Data - це коли даних більше 100 Гб (500 Гб, 1 ТБ, і т.д.);

− Big Data – це дані, які неможливо обробляти у Excel;

− Big Data – це дані, які неможливо обробити на одному комп'ютері.

− І навіть такі:

− Big Data – це взагалі будь-які дані.

− Big Data – штучно вигаданий термін, який є не більш, ніж маркетингових ходом.

У той же час, умовно-офіційне визначення Великих даних наступне - це серія підходів, інструментів та методів обробки структурованих та неструктурованих даних величезних обсягів та значного розмаїття для отримання результатів, сприйманих людиною, ефективних в умовах безперервного приросту, розподілу по численних вузлах обчислювальної мережі, що сформувалися наприкінці 2000-х років, системам управління базами даних та рішенням класу Business Intelligence.

Таким чином, під Big Data мається на увазі не якийсь конкретний обсяг даних і навіть самі дані, а методи їх обробки, що дозволяють розподілено обробляти інформацію. Ці методи можна застосувати як до великих масивів даних (таких як зміст усіх сторінок в інтернеті), так і до маленьких (наприклад, розміром з кілька КБ).

Наведемо кілька прикладів того, що може бути джерелом даних, для яких потрібні методи роботи з великими даними:

− логи поведінки користувачів в Інтернеті;

− GPS-сигнали від автомобілів для транспортної компанії;

− дані, що знімаються з датчиків у великому адронному колайдері;

− оцифровані книги у Державній Бібліотеці;

− інформація про транзакції всіх клієнтів банку;

− інформація про всі покупки у великій рітейл-мережі тощо.

Число джерел даних стрімко зростає, отже, на технології їх обробки зростає попит.

1.3 Принципи роботи з великими даними

Виходячи з визначення Big Data, можна сформулювати основні засади роботи з такими даними [2, 3]:

1. Горизонтальна масштабованість.

Оскільки даних може бути скільки завгодно – будь-яка система, що передбачає обробку великих даних, має бути розширюваною. У 2 рази збільшився обсяг даних – у 2 рази слід збільшити кількість апаратних ресурсів у кластері і т.д.

2. Відмовостійкість.

Принцип горизонтальної масштабованості передбачає, що машин у кластері може бути багато. Наприклад, Hadoop-кластер Yahoo має понад 42 тисячі обчислювальних вузлів. Це означає, що частина даних машин гарантовано виходитиме з ладу. Методи роботи з великими даними повинні враховувати можливість таких збоїв і забезпечувати невпинність процесу обробки, або часткову зупинку без значних наслідків.

3. Локальність даних.

Очевидно, що у великих розподілених системах дані розподілені за великою кількістю машин. Якщо дані фізично перебувають у одному сервері, а обробляються у іншому – витрати на передачу даних можуть перевищити витрати на обробку. Тому одним із найважливіших принципів проектування BigData-рішень є принцип локальності даних – тобто, обробку даних доцільно виконувати на тій же машині, де їх зберігається.

Усі сучасні засоби роботи з великими даними так чи інакше дотримуються цих трьох принципів. Для того, щоб їх дотримуватися, необхідно вигадувати якісь методи, способи та парадигми розробки засобів обробки даних. Один із класичних методів розглянемо далі.

1.4 MapReduce

MapReduce – це модель розподіленої обробки даних, запропонована Google для обробки великих обсягів даних на комп'ютерних кластерах. Ідеологія MapReduce вичерпно ілюструється рисунком 1.1.

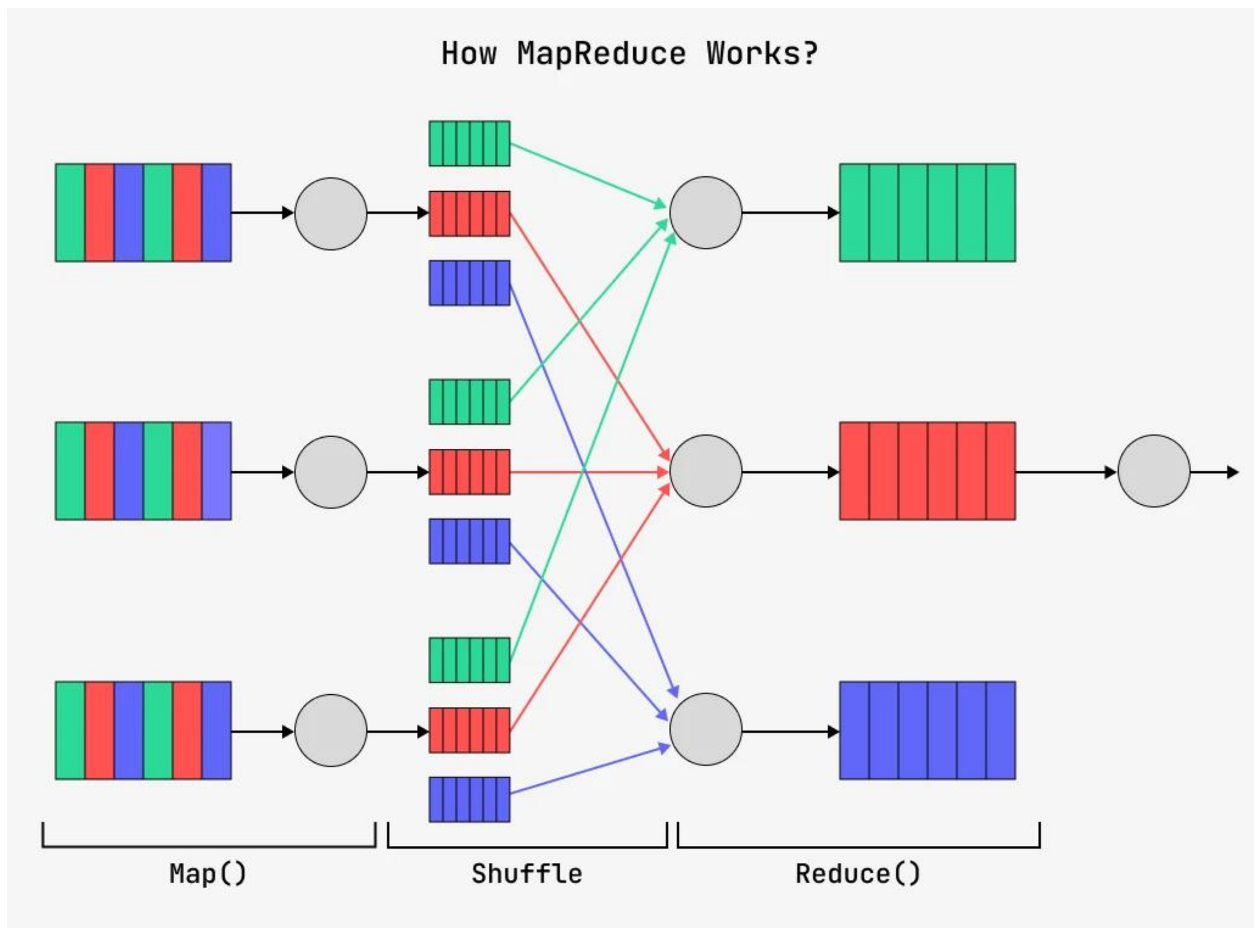

Рисунок 1.1 – Принцип роботи MapReduce

MapReduce передбачає, що дані організовані у вигляді певних записів. Обробка даних відбувається у 3 стадії [3, 4]:

1. Стадія Map. На цій стадії дані передбачаються за допомогою функції map(), яка визначається користувачем. Робота цієї стадії полягає у переробці та фільтрації даних. Робота дуже схожа на операцію map у функціональних мовах програмування - функція користувача застосовується до кожного вхідного запису.

Функція map(), застосована до одного вхідного запису і видає множину пар ключ-значення. При цьому, функція може видати лише один запис, може не видати нічого, а може видати кілька пар ключ-значення. Що буде в ключі та в значенні – вирішувати користувачеві, але ключ – дуже важлива річ, оскільки дані з одним ключем у майбутньому потраплять до одного екземпляра функції reduce.

2. Стадія Shuffle. Виконується непомітно для користувача. На цій стадії результати виведення функції map розбираються на окремі частини (фрагменти даних), при цьому кожна частина відповідає одному ключу виведення стадії map. Надалі ці частини будуть слугувати входом до reduce.

3. Стадія Reduce. Кожен фрагмент даних із значеннями, сформований на стадії shuffle, потрапляє на вхід функції reduce().

Функція reduce задається користувачем та обчислює фінальний результат для окремого фрагменту. Множина всіх значень, що повертаються функцією reduce(), є фінальним результатом MapReduce-завдання.

Для MapReduce справедливими є наступні властивості:

1. Усі запуски функції map працюють незалежно і можуть працювати паралельно, у тому числі різних кластерних машинах.

2. Усі запуски функції reduce працюють незалежно і можуть працювати паралельно, у тому числі різних кластерних машинах.

3. Shuffle являє собою паралельне сортування, тому також може працювати на різних машинах кластера. Пункти 1-3 дозволяють виконати принцип горизонтального масштабування.

4. Функція map, зазвичай, застосовується у межах тієї ж машини, де зберігаються дані – це дозволяє знизити передачу даних у мережі (принцип локальності даних).

5. MapReduce – це повне сканування даних, ніяких індексів немає. Це означає, що MapReduce має недостатню ефективність у випадках, коли відповідь потрібна дуже швидко.

1.5 Приклади завдань, що ефективно вирішуються за допомогою MapReduce

1.5.1 Word Count

Почнемо з класичного завдання – Word Count. Завдання формулюється так: є великий корпус документів. Завдання – для кожного слова, яке хоча б одноразово зустрічається в корпусі, порахувати сумарну кількість разів, яку ці слова фігурували [4].

Рішення завдання.

Якщо маємо великий корпус документів, нехай один документ буде одним вхідним записом для MapRreduce–завдання. У MapReduce ми можемо тільки вказувати функції користувача, у цьому випадку скористаємося python-like псевдокодом:

```
def map(doc):
for word in doc:
yield word, 1
```

```
def reduce(word, values):
yield word, sum(values)
```

Функція map перетворює вхідний документ на набір пар (слово, 1), shuffle прозоро для нас перетворює це на пари (слово, [1,1,1,1 ,1,1]), reduce підсумовує ці одиниці, повертаючи фінальну відповідь для слова.

1.5.2 Обробка логів рекламної системи

Припустимо, що існує csv-лог рекламної системи вигляду [4]:

```
<user_id>,<country>,<city>,<campaign_id>,<creative_id>,<payment></p>

11111, CN,Shanghai,2,4,0.3
22222, UA, Kyiv, 2,3,0.2
13413,CN,Shenzhen,4,11,0.7
```

Необхідно розрахувати середню вартість показу реклами містами Китаю.

Рішення:

```
def map(record):
user_id, country, city, campaign_id, creative_id, payment = record.split(",")
payment=float(payment)
if country == "PL":
yield city, payment
def reduce(city, payments):
yield city, sum(payments)/len(payments)
```

Функція map перевіряє, чи потрібний нам цей запис, і, якщо цей запас потрібен, залишає лише потрібну інформацію (місто та розмір платежу). Функція reduce розраховує фінальну відповідь містом, маючи список всіх платежів у цьому місті.

2 РОЗГОРТАННЯ ТА НАЛАШТУВАННЯ РОЗПОДІЛЕНОЇ ОБЧИСЛЮВАЛЬНОЇ СИСТЕМИ НА ОСНОВІ APACHE SPARK

2.1 Apache Spark як засіб обробки Великих даних

Сучасні великі дані (Big Data) стали невід'ємною частиною інформаційних систем багатьох компаній, наукових лабораторій та державних організацій. Обробка таких обсягів даних потребує особливого підходу, і тут застосовуються технології розподілених обчислень [5].

Однією з найпопулярніших платформ для їх реалізації є Apache Spark (Рис.2.1).

Apache Spark - це потужний фреймворк, що забезпечує високу швидкість обробки даних завдяки механізму in-memory обчислень та можливості масштабування на кластерах, що складаються з десятків, сотень і навіть тисяч вузлів.

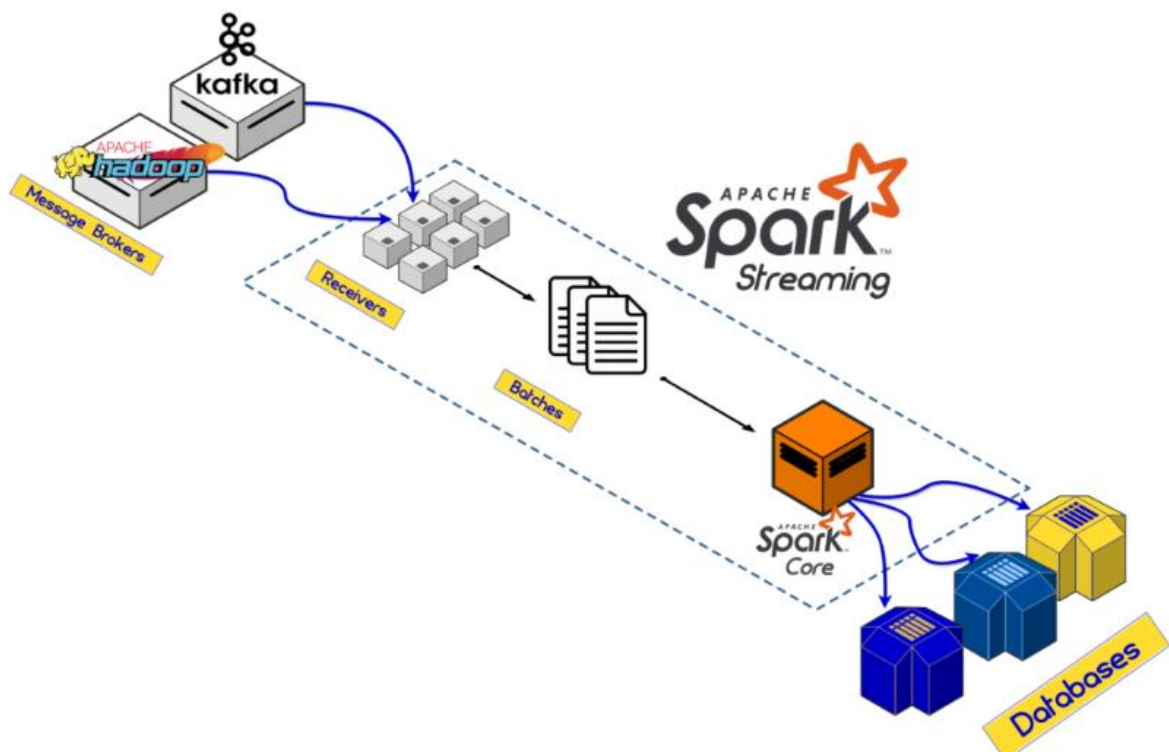

Рисунок 2.1 – Місце Apache Spark в екосистемі Big Data

2.2 Архітектура Apache Spark

Перш ніж розпочати розгортання платформи, важливо зрозуміти базові архітектурні компоненти та принципи роботи Spark (Рис.2.2) [5]:

– Driver. Головний процес програми, де відбувається управління виконанням завдань, координація обчислень та агрегування результатів;

– Executor. Виконавець обчислювальних завдань; кожен вузол кластера може мати один або кілька процесорів Executor;

– Cluster Manager. Компонент, який розподіляє ресурси між програмами. Apache Spark підтримує кілька типів менеджерів: Standalone, YARN, Mesos та Kubernetes;

– RDD (Resilient Distributed Dataset). Базова абстракція даних у Spark; є розподіленою колекцією об'єктів, яка стійка до збоїв. У більш сучасних версіях Spark активно використовуються DataFrame і Dataset, але RDD залишаються «фундаментальною цеглою» екосистеми.

Завдяки такій архітектурі Spark може працювати як у локальному режимі на одній машині, так і в розподіленому режимі кластерів різних розмірів.

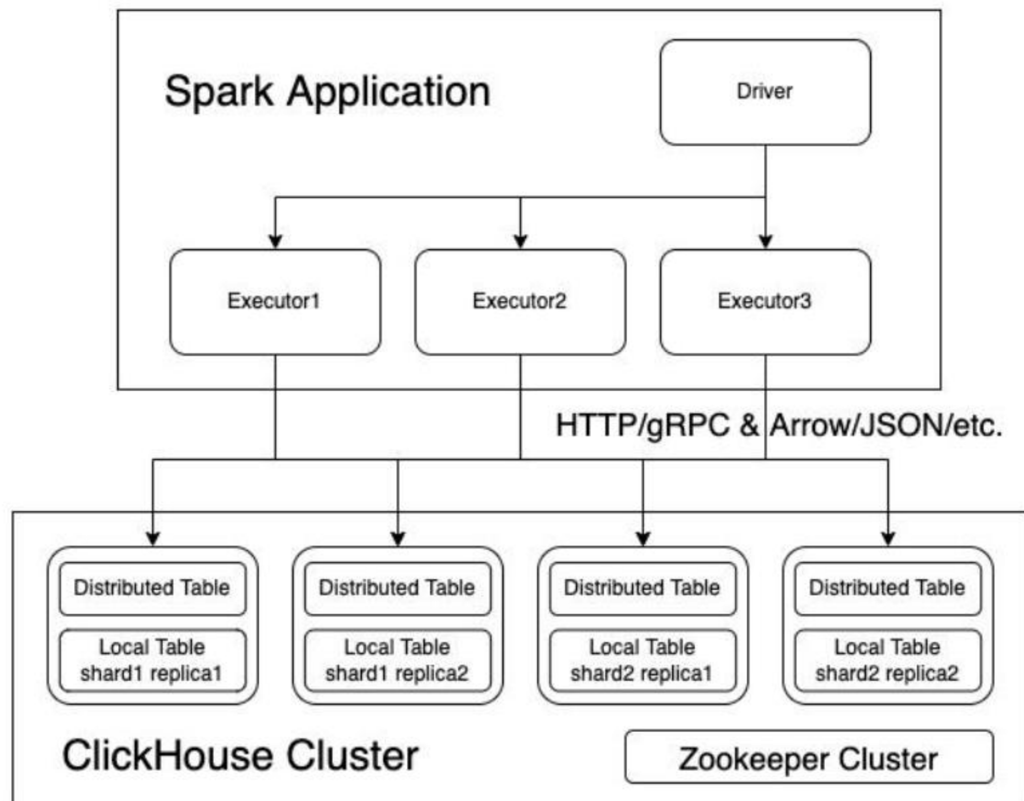

Рисунок 2.2 – Базова архітектура Apache Spark

2.3 Підготовка оточення та режими роботи кластера Spark

2.3.1 Встановлення Java

Для роботи Apache Spark потрібна встановлена Java (версія 8 або 11). Необхідно перевірити встановлену версію Java командою [5, 6]:

```
java -version
```

Якщо Java не встановлено, можна завантажити дистрибутив із офіційного сайту Oracle, або використовувати відкриті реалізації (наприклад, OpenJDK). Java встановлюється згідно з інструкціями для потрібної операційної системи (Windows, Linux або MacOS).

2.3.2 Установка Scala (необов'язково)

Apache Spark написано на Scala, проте, для запуску простих завдань на Python (PySpark) або SQL Scala як окремий пакет не потребується. Якщо планується розробка на Scala, потрібно встановити відповідний компілятор (Scala Compiler) і середовище (sbt або Maven).

2.3.3 Завантаження та розпакування Apache Spark

Щоб завантажити архів з Apache Spark, потрібно зайти на офіційний сайт spark.apache.org. Далі зазвичай вибирають стабільну версію (наприклад, 3.x), сумісну з поточною версією Hadoop/YARN (якщо планується інтеграція).

2.3.4 Режими роботи кластера Spark

Apache Spark може працювати в кількох режимах, що залежать від вибору менеджеру кластеру (Cluster Manager) [5, 6]:

1. Standalone - базовий менеджер ресурсів, що входить до складу Spark за замовчанням. Він простий у встановленні та достатній для невеликого кластера або тестового оточення.
2. YARN - менеджер ресурсів у Hadoop-кластерах. Дозволяє ефективно ділити ресурси між різними Hadoop-додатками (MapReduce, Spark та ін.).
3. Apache Mesos - Загальний планувальник ресурсів для великих кластерів.
4. Kubernetes - засіб контейнеризації та оркестрації додатків, у тому числі Spark.

2.4 Розгортання кластера Spark (Standalone та Hadoop YARN)

2.4.1 Розгортання в режимі Standalone

Схему кластера Spark у режимі Standalone зображено на рис. 2.3.

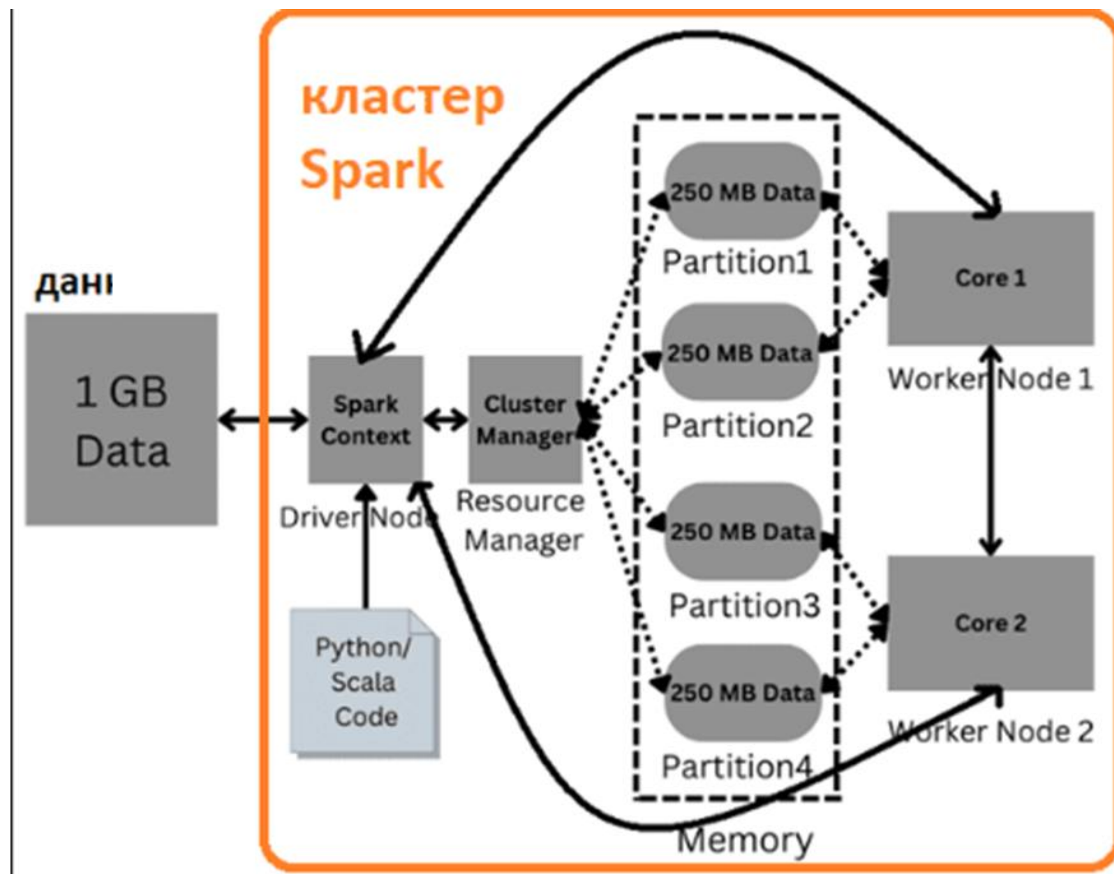

Рисунок 2.3 – Схема кластера Spark у режимі Standalone

Спочатку необхідно виконати налаштування ряду конфігураційних файлів усередині директорії conf, а саме:

– spark-env.sh (Linux/Mac) або spark-env.cmd (Windows);
– spark-defaults.conf;
– slaves (називається workers у нових версіях Spark).

Розглянемо приклад мінімальної конфігурації для Linux/Mac [6]:

```
# Налаштування змінних оточення
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk
export SPARK_MASTER_HOST=master-node
export SPARK_WORKER_MEMORY=2g
export SPARK_WORKER_CORES=2
```

У файлі slaves (або workers) потрібно перерахувати всі вузли кластера, на яких працюватимуть Spark Worker-процеси, наприклад:

```
worker01
worker02
worker03
```

Далі потрібно перейти до директорії, де встановлено Spark, потім виконати такі команди [5, 6]:

```
# Запуск Spark Master
sbin/start-master.sh
# Запуск Spark Worker на кожному вузлі, вказавши адресу Master:
sbin/start-worker.sh spark://master-node:7077
```

Якщо є декілька worker-вузлів, зазвичай запускають start-worker.sh на кожному вузлі. Для Windows аналогічними командами будуть файли .cmd у директорії sbin.

Після успішного запуску в браузері буде доступний інтерфейс Master за замовчуванням за адресою http://master-node:8080 (замість master-node слід використовувати реальне ім'я хоста або IP-адресу).

2.4.2 Розгортання Spark на кластері Hadoop (YARN)

Якщо в інфраструктурі вже є кластер Hadoop, то логічно інтегрувати Spark з YARN [5, 6]:

1. Для початку потрібно переконатися, що Hadoop встановлений і коректно налаштований (NameNode, ResourceManager та ін.).

2. Spark слід налаштувати на роботу в режимі YARN, вказавши шляхи до Hadoop у spark-env.sh та spark-defaults.conf.

3. Далі завдання запускають за допомогою команд [6]:

```
spark-submit \
  --master yarn \
  --deploy-mode cluster \
  --class org.apache.spark.examples.SparkPi \
  /path/to/examples.jar
```

Таким чином, Spark буде використовувати планувальник YARN для розподілу ресурсів. На цей випадок обробка Spark-завдань буде виконуватися згідно логіці, поданій рис. 2.4.

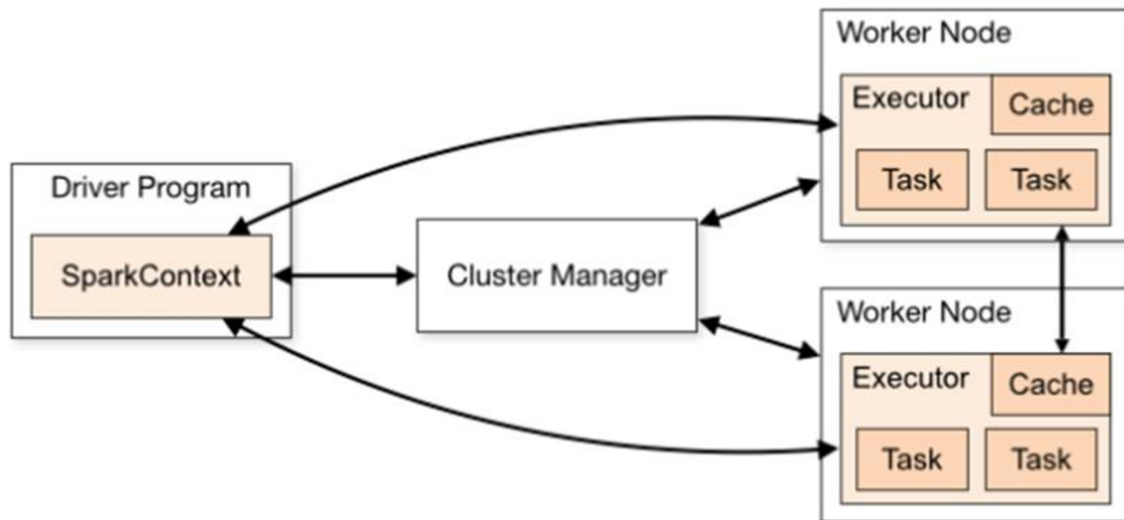

Рисунок 2.4 – Spark-завдання в інтерфейсі YARN ResourceManager

2.5 Виконання завдань та налаштування параметрів

2.5.1 Виконання простих завдань

Spark надає кілька інтерфейсів для роботи [5, 6]:

− spark-shell для Scala - ./bin/spark-shell;
− pyspark для Python - ./bin/pyspark;
− spark-sql для SQL-запитів - ./bin/spark-sql.

В інтерактивному режимі можна швидко створювати RDD, DataFrame, тестувати логіку та налагоджувати завдання.

Розглянемо приклад створення простого RDD у Scala [7]:

```
val data = sc.parallelize(Seq(1, 2, 3, 4, 5))
val squared = data.map(x => x * x)
println(squared.collect().mkString(", "))
```

Результат виконання буде виведений у консоль.

2.5.2 Spark-submit

Для важких/масштабних завдань використовується spark-submit. Розглянемо приклад скрипта для Scala/Java-програми зі spark-submit [6]:

```
spark-submit \
  --class com.example.SparkApp \
  --master spark://master-node:7077 \
  --deploy-mode cluster \
  --executor-memory 2G \
  --total-executor-cores 4 \
  /path/to/yourApp.jar
```

Аналогічний приклад для випадку PySpark:

```
spark-submit \
  --master spark://master-node:7077 \
  --deploy-mode cluster \
  --executor-memory 2G \
  --total-executor-cores 4 \
  /path/to/script.py
```

2.6 Налаштування параметрів та оптимізація Spark

У Spark існує ряд ключових параметрів, маніпулювання величинами яких може змінювати продуктивність виконання обчислювальних завдань. Це, зокрема [5]:

– spark.driver.memory – обсяг пам'яті, доступний драйверу;

– spark.executor.memory - обсяг пам'яті, доступний кожному executor-процесу;

– spark.executor.cores – кількість ядер, виділених кожному executor-процесу.

Значення цих параметрів залежить від розміру кластера, обсягів даних, і типу завдань.

У сучасних версіях Spark рекомендується використовувати API DataFrame та Dataset замість RDD, що забезпечує можливість реалізації:

– оптимізації через планувальник Catalyst;

– відкладеного обчислення;

– більш гнучкого управління схемами та типами даних.

Приклад конструкції коду Scala + DataFrame показано далі [5, 6]:

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder
  .appName("DataFrameExample")
  .master("spark://master-node:7077")
  .getOrCreate()
val df = spark.read.json("/path/to/data.json")
df.printSchema()
df.show()
val result = df.groupBy("category").count()
result.show()
```

Spark також надає кілька механізмів кешування (наприклад, persist() і cache()), що особливо корисно при повторних проходах по одним і тим самим наборам даних.

При цьому, важливо коректно розраховувати обсяг доступної оперативної пам'яті та дискового простору, щоб уникнути надмірно частого звернення до дискових пристроїв.

2.7 Моніторинг, налагодження та безпека платформи

Для контролю за завданнями та використання ресурсів в Spark передбачено кілька інструментів, зокрема:

– Spark UI, як показано рис.2.5 (за замовчуванням на порті 4040 для локального режиму та 8080 для Master у режимі Standalone);

– логи Spark, які за потреби можна переглядати на кожному вузлі кластера;

– інтеграція із системами моніторингу (наприклад, Prometheus + Grafana).

При налагодженні слід звертати увагу на стадії (stages) та завдання (tasks) у Spark UI.

Розпаралелювання та об'єднання даних (shuffle) - основні точки, де можливі вузькі місця або збої.

Spark 2.4.0-cdh6.3.3 | Jobs | Stages | Storage | Environment | Executors | SQL | spark-app-ui application UI

Details for Job 2

Status: SUCCEEDED
Completed Stages: 2

- Event Timeline
- DAG Visualization

Stage 2: WholeStageCodegen, Exchange
Stage 3: Exchange, WholeStageCodegen, mapPartitionsInternal

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	count at NativeMethodAccessorImpl.java:0 +details	2022/02/08 14:01:39	5 ms	1/1			150.0 B	
2	count at NativeMethodAccessorImpl.java:0 +details	2022/02/08 14:01:39	37 ms	2/2	103.9 KB			150.0 B

Рисунок 2.5 – Інтерфейс Spark UI з виконаними завданнями

Що стосується питань безпека та розмежування доступу, то у виробничих кластерах важливо враховувати необхідність налаштування [7]:

– аутентифікацію користувачів (використовується Kerberos при інтеграції з Hadoop або інші подібні механізми);

– шифрування трафіку між вузлами (за замовчуванням – протоколи SSL, TLS).

– розмежування прав доступу до даних (може бути використано HDFS ACL, налаштування прав на рівні DataFrame/Table).

Усі перелічені аспекти є особливо актуальними, коли йдеться про обробку конфіденційної інформації.

3 ВИКОНАННЯ ЗАВДАНЬ MAPREDUCE У СЕРЕДОВИЩІ APACHE SPARK

3.1 MapReduce у Spark

У класичній реалізації MapReduce використовується Apache Hadoop, який виконує завдання через вузли JobTracker та TaskTracker [3].

Для забезпечення високої продуктивності обчислень дана система вимагає коректного конфігурування ресурсів.

В інакшому випадку під час обробки складних завдань це може приводити до:

– високих затримок - часте звернення до диску на кожному етапі обчислень призводить до значних часових витрат;

– обмеження продуктивності, так як Hadoop MapReduce неефективно використовує оперативну пам'ять, що обмежує швидкість при роботі з великими наборами даних.

У свою чергу, Apache Spark оптимізує MapReduce за рахунок виконання обчислень у пам'яті. Основними відмінностями Apache Spark від Hadoop MapReduce є [5]:

– Наявність структури даних RDD (Resilient Distributed Dataset).

– RDD являє собою основну структуру даних в Spark, що забезпечує відмовостійкість і т.з. «ліниві» обчислення. Вона підтримує виконання процедур трансформації (наприклад, map, filter) та дії (наприклад, collect, saveAsTextFile).

– Обробка даних у пам'яті. Spark зберігає проміжні результати оперативної пам'яті, мінімізуючи тим самим звернення до диску. У свою чергу, це дозволяє прискорити виконання завдань у 10-100 разів порівняно з Hadoop MapReduce.

– Спрощений API. Високорівневі функції, такі як map, reduceByKey, filter та подібні дозволяють писати лаконічний і зрозумілий код, що спрощує розробку.

– Інструментарій DAG (Directed Acyclic Graph). Замість жорсткої послідовності Map та Reduce Spark використовує DAG для гнучкого керування завданнями. Це дозволяє оптимізувати виконання завдань рахунок об'єднання операцій на стадіях.

Незважаючи на усі переваги Spark, MapReduce не підходить у якості інструментарію для рішення деяких типів завдань, наприклад [5]:

– складних аналітичних запитів. Зокрема, виконання запитів, таких як з'єднання таблиць, може бути низькоефективним.

– ітеративні алгоритми. Так, алгоритми машинного навчання чи обробки графів, які вимагають багаторазового повторного використання даних, працюють повільніше через необхідність звернення до диску.

3.2 Виконання завдань MapReduce у Spark

Припустимо, є файл із текстом, для якого необхідно підрахувати частоту кожного слова.

Для випадку Python матимемо наступне [5]:

```
from pyspark import SparkContext
sc = SparkContext("local", "Word Count")
data = sc.textFile("input.txt")
counts = data.flatMap(lambda line: line.split(" ")) \
            . map (lambda word: (word, 1)) \
            .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("output")
```

У свою чергу, код Scala буде таким:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
val conf = новий SparkConf().setAppName("Word Count").setMaster("local")
val sc = новий SparkContext(conf)
val data = sc.textFile("input.txt")
val counts = data.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("output")
```

Дані приклади демонструють, як Spark спрощує виконання завдань MapReduce, забезпечуючи високу продуктивність та зручність розробки.

3.3 Типові операції MapReduce в Apache Spark

3.3.1 Трансформації в Apache Spark

Трансформації являють собою «ліниві» операції, які перетворюють вихідний RDD на новий. Вони виконуються лише тоді, коли викликається дія (Action), що ініціює активацію графа виконання (DAG). Це дозволяє Spark оптимізувати виконання завдань, уникаючи зайвих обчислень [6].

Концепція лінивих трансформацій

Spark не виконує обчислення відразу після виклику трансформації. Натомість створюється Directed Acyclic Graph (DAG), який оптимізується для ефективного виконання. Виконання DAG ініціюється лише під час виклику дії. Такий підхід дозволяє:

– виключати непотрібні операції;

– зменшувати обсяг даних, що передаються;

– прискорювати обчислення рахунок паралельного виконання.

У реальній практиці лінивість трансформацій означає, що можна визначити складні послідовності обчислень без негайного їх виконання. (наприклад, collect() або saveAsTextFile()).

Основні трансформації у Spark

Map. Перетворює кожен елемент RDD за допомогою заданої функції. Це одна з найпоширеніших трансформацій для зміни формату, або обчислення нових значень на основі наявних даних.

Розглянемо приклад даної трансформації у Python [6, 7]:

```
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.map(lambda x: x * 2)
print(result.collect()) # [2, 4, 6, 8]
```

Далі – такий же приклад для Scala:

```
val rdd = sc.parallelize(List(1, 2, 3, 4))
val result = rdd.map(x => x * 2)
result.collect().foreach(println) // 2, 4, 6, 8
```

Трансформація FlatMap. У цілому, працює як map, але кожен елемент може бути перетворений на кілька нових, створюючи плоску структуру даних. Ця трансформація часто використовується для обробки текстових даних, як-то аналіз слів.

Для випадку Python маємо наступний приклад даної трансформації [5]:

```
rdd = sc.parallelize(["hello world", "spark mapreduce"])
result = rdd.flatMap(lambda line: line.split(" "))
print(result.collect())    #    ['hello',    'world',    'spark', 'mapreduce']
```

Для Scala приклад використання FlatMap наступний**:**

```
val rdd = sc.parallelize(List("hello world", "spark mapreduce"))
val result = rdd.flatMap(line => line.split(" "))
result.collect().foreach(println)  //  hello,  world,  spark, mapreduce
```

Трансформація Filter. Видаляє елементи, що не задовольняють задану умову. Використовується для попередньої обробки даних, такої, як видалення невалідних записів [5, 6].

Приклад для Python [4]:

```
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.filter(lambda x: x % 2 == 0)
print(result.collect()) # [2, 4]
```

Приклад для Scala:

```
val rdd = sc.parallelize(List(1, 2, 3, 4))
val result = rdd.filter(x => x % 2 == 0)
result.collect().foreach(println) // 2, 4
```

Наступна трансформація – GroupByKey, групує дані ключів, зберігаючи всі значення для кожного ключа. Це корисно тоді, коли потрібно зберегти повний набір значень кожного ключа.

Приклад трансформації для Python [5]:

```
rdd = sc.parallelize([("a", 1), ("b", 2), ("a", 3)])
result = rdd.groupByKey().mapValues(list)
print(result.collect()) # [('a', [1, 3]), ('b', [2])]
```

У свою чергу, для Scala маємо:

```
val rdd = sc.parallelize(List(("a", 1), ("b", 2), ("a", 3))))
val result = rdd.groupByKey().mapValues(_.toList)
result.collect().foreach(println) // ('a', [1, 3]), ('b', [2])
```

Ще одна трансформація – ReduceByKey, виконує агрегацію значень з однаковими ключами. Передує groupByKey, оскільки попередня агрегація виконується локально на кожному вузлі, що зменшує обсяг даних, що передаються мережею [5, 6].

Розглянемо приклад застосування даної трансформації для Python [5]:

```
rdd = sc.parallelize([("a", 1), ("b", 2), ("a", 3)])
result = rdd.reduceByKey(lambda x, y: x + y)
print(result.collect()) # [('a', 4), ('b', 2)]
```

Далі розглянемо приклад для Scala:

```
val rdd = sc.parallelize(List(("a", 1), ("b", 2), ("a", 3))))
val result = rdd.reduceByKey(_ + _)
result.collect().foreach(println) // ('a', 4), ('b', 2)
```

Використання трансформацій надає розробнику ряд розширених можливостей, а саме [5, 6]:

1. Кешування даних. Повторне використання даних можливе завдяки cache() або persist(). Це особливо корисно для багатокрокових обчислень, де ті самі дані використовуються в різних етапах.

Приклад коду для кешування у Python [7]:

```
rdd = sc.textFile("data.txt").cache()
print(rdd.count())
print(rdd.first())
```

2. Партиціювання. Для оптимального розподілу навантаження між вузлами можна налаштувати кількість партій. Це робиться за допомогою методу partitionBy() для парних RDD.

Відповідний приклад коду наведено далі [6]:

```
rdd = sc.parallelize([("a", 1), ("b", 2), ("a", 3)])
partitioned_rdd = rdd.partitionBy(2)
print(partitioned_rdd.glom().collect())
```

Можна сформулювати такі рекомендації щодо використання трансформацій в Apache Spark:

– слід використовувати reduceByKey замість groupByKey, щоб зменшити обсяг даних, що передаються через мережу;

– необхідно оптимізувати DAG, комбінуючи операції, щоб уникнути зайвих кроків;

– Налаштуйте кількість партицій, враховуючи обсяг даних та ресурси кластера.

– для часто використовуваних RDD треба використовувати кешування.

3.3.2 Дії (Actions) в Apache Spark

Дії - це операції, які ініціюють виконання усіх трансформацій, визначених до них [5, 6, 7].

Основні типи дій:

1. Повернення даних драйверу. Тут передбачено виконання дій collect та take.

У свою чергу, collect повертає всі елементи RDD у списку. Використовується для невеликих наборів даних, які містяться у пам'яті драйвера.

Приклад для Python [5, 7]:

```
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.collect()
print(result) # [1, 2, 3, 4]
```

та для Scala:

```
val rdd = sc.parallelize(List(1, 2, 3, 4))
val result = rdd.collect()
println(result.mkString(", ")) // 1, 2, 3, 4
```

Нерідко використовується для збору даних для побудови звіту.

Однак collect слід використовувати з обережністю, оскільки для великих RDD це може призвести до переповнення драйвера пам'яті.

Наступна дія, take, повертає перші N елементів RDD, що є зручним для перевірки перших результатів трансформацій.

Використання take для Python демонструє наступний приклад [6, 7]:

```
rdd = sc.parallelize([10, 20, 30, 40])
result = rdd.take(2)
print(result) # [10, 20]
```

Для Scala маємо наступне**:**

```
val rdd = sc.parallelize(List(10, 20, 30, 40))
val result = rdd.take(2)
println(result.mkString(", ")) // 10, 20
```

Дана дія використовується для попереднього аналізу даних, наприклад для перевірки формату вхідних даних. Це також може бути корисним для ітеративного підходу при розробці аналітичних додатків, дозволяючи розробнику швидко зрозуміти структуру даних.

2. Агрегація даних

Дія reduce. Застосовується для послідовного об'єднання елементів RDD.

Приклад використання у Python [5, 7]:

```
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.reduce(lambda x, y: x + y)
print(result) # 10
```

Для випадку Scala:

```
val rdd = sc.parallelize(List(1, 2, 3, 4))
val result = rdd.reduce(_ + _)
println(result) // 10
```

На практиці у ході обробки Великих даних може бути використано, наприклад, для підрахунку загального обсягу продажів.

Важливо враховувати, що reduce повертає лише одне значення і тому підходить лише агрегування даних.

Дія count. Повертає кількість елементів у RDD.

Розглянемо приклад використання у Python [6, 7]:

```
rdd = sc.parallelize(["a", "b", "c", "d"])
result = rdd.count()
print(result) # 4
```

а також у Scala:

```
val rdd = sc.parallelize(List("a", "b", "c", "d"))
val result = rdd.count()
println(result) // 4
```

Дія може використовуватися для визначення обсягу даних, наприклад, з метою оцінки масштабу завдання.

First. Повертає перший елемент RDD.

У Python використовується наступним чином [6, 7]:

```
rdd = sc.parallelize([100, 200, 300])
result = rdd.first()
print(result) # 100
```

Для Scala, у цілому, синтаксис є схожим:

```
val rdd = sc.parallelize(List(100, 200, 300))
val result = rdd.first()
println(result) // 100
```

Дана дія може виконувати швидку перевірку структури даних у наборі.

3. Запис даних

Тут, у першу чергу, слід розглянути дію saveAsTextFile, яка зберігає RDD у текстовий файл. Кожен елемент RDD, при цьому, записується в окремий рядок файлу.

Приклад коду для Python [5, 7]:

```
rdd = sc.parallelize(["Spark", "MapReduce", "Big Data"])
rdd.saveAsTextFile("output.txt")
```

Далі розглянемо приклад коду для Scala**:**

```
val rdd = sc.parallelize(List("Spark", "MapReduce", "Big Data"))
rdd.saveAsTextFile("output.txt")
```

На практиці дія використовується для експорту оброблених даних для використання в інших системах, наприклад, системах візуалізації або сховищах даних.

3.4 Порівняння трансформацій та дій

Трансформації ліниві і не виконуються відразу, а дії ініціюють виконання DAG та повертають результат або зберігають дані.

Приклад послідовності: спочатку трансформація (map, filter), потім дія (collect).

Для Python це буде мати такий вигляд [6, 8]:

```
rdd = sc.parallelize(["error log", "info log", "error report"])
filtered_rdd = rdd.filter(lambda x: "error" in x)
result = filtered_rdd.collect()
print(result) # ["error log", "error report"]
```

3.5 Розширені приклади використання дій

Обчислення середнього значення.

Для рішення цього завдання можна використовувати комбінацію дій reduce та count.

Для Python це матиме вигляд [8]:

```
rdd = sc.parallelize([1, 2, 3, 4])
total = rdd.reduce(lambda x, y: x + y)
count = rdd.count()
= = total / count
print(average) # 2.5
```

У свою чергу, для Scala код буде таким:

```
val rdd = sc.parallelize(List(1, 2, 3, 4))
val total = rdd.reduce(_ + _)
val count = rdd.count()
Val average = total.toDouble / count
println(average) // 2.5
```

Збереження результатів обчислень.

Щоб зберегти оброблені дані у кількох форматах, таких як JSON або CSV, може бути використано дії та бібліотеки Spark SQL [8]:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Save Example").getOrCreate()
df = spark.createDataFrame([(1, "Alice"), (2, "Bob")], ["id", "name"])
df.write.csv("output.csv")
```

Щодо використання дій існують такі рекомендації [5-8]:

- дії слід використовувати для перевірки даних у процесі розробки, наприклад take або first;
- для великих наборів даних необхідно віддавати перевагу збереженню результатів у файлову систему, а не їх поверненню на драйвер;
- використання reduce доцільне лише для завдань, де потрібно повернути одне агреговане значення.
- для ітеративної розробки та налагодження слід комбінувати дії з невеликими вибірками даних.

3.6 Спільне використання трансформацій та дій

Apache Spark виконує формування Directed Acyclic Graph (DAG) для управління завданнями [6, 9].

Приклад типової роботи DAG показано на рис. 3.1.

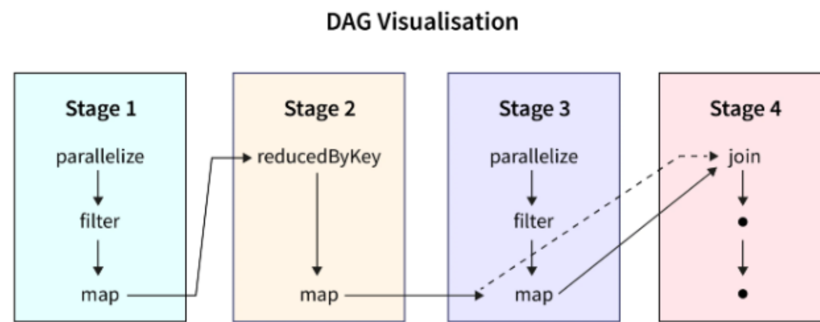

Рисунок 3.1 – Приклад роботи DAG

Припустимо, що існує текстовий файл з логами, з якого необхідно виокремити рядки з помилками, підрахувати кількість таких рядків та зберегти результати у файл. Для рішення даного завдання виконуються такі технологічні етапи [6, 9]:

1. Визначення DAG:

Для Python це буде здійснюватися наступним чином:

```
rdd = sc.textFile("logs.txt")
error_lines = rdd.filter(lambda line: "ERROR" in line)
error_count = error_lines.count()
error_lines.saveAsTextFile("errors_output")
```

Для Scala код буде подібним:

```
val rdd = sc.textFile("logs.txt")
val errorLines = rdd.filter(line => line.contains("ERROR"))
val errorCount = errorLines.count()
errorLines.saveAsTextFile("errors_output")
```

DAG включає такі етапи, як:

– читання даних (textFile);
– фільтрування рядків (filter);
– запис результатів (saveAsTextFile).

2. Оптимізація DAG. Spark оптимізує DAG, поєднуючи послідовні трансформації у стадії (stages) Наприклад, операції filter і map можуть бути виконані на одній стадії, щоб мінімізувати передачу даних.

Розглянемо ще один приклад спільного використання трансформацій та дій.

Припустимо, що необхідно виконати аналіз логів з Виявленням IP-адреси, які найчастіше зустрічаються в логах веб-сервера.

Для Python це завдання вирішується настутним чином [6, 9]:

```
rdd = sc.textFile("weblogs.txt")
ip_addresses = rdd.map(lambda line: line.split(" ")[0])
ip_counts = ip_addresses.map(lambda ip: (ip, 1)).reduceByKey(lambda a, b: a + b)
sorted_ips = ip_counts.sortBy(lambda x: x[1], ascending=False)
sorted_ips.take(5)
```

Для Scala:

```
val rdd = sc.textFile("weblogs.txt")
val ipAddresses = rdd.map(line => line.split(" ")(0))
val ipCounts = ipAddresses.map(ip => (ip, 1)).reduceByKey(_ +_)
val sortedIps = ipCounts.sortBy(_._2, ascending = false)
sortedIps.take(5).foreach(println)
```

Розглянемо ще один приклад. Виконаємо аналіз продажів з підрахунком загального доходу за категоріями товарів:

Python [6, 7, 9]:

```
rdd = sc.textFile("sales.txt")
sales_data = rdd.map(lambda line: line.split(","))
category_sales = sales_data.map(lambda fields: (fields[2], float(fields[3]))))
total_sales = category_sales.reduceByKey(lambda a, b: a + b)
total_sales.collect()
```

Код для Scala:

```
val rdd = sc.textFile("sales.txt")
val salesData = rdd.map(line => line.split(","))
val categorySales = salesData.map(fields => (fields(2), fields(3).toDouble))
val totalSales = categorySales.reduceByKey(_ + _)
totalSales.collect().foreach(println)
```

4 ТЕСТУВАННЯ ТА МОНІТОРИНГ ПРОДУКТИВНОСТІ РОЗПОДІЛЕНОЇ ОБЧИСЛЮВАЛЬНОЇ СИСТЕМИ НА ОСНОВІ APACHE SPARK

Тестування та моніторинг є важливими процесами при роботі з розподіленими системами, такими як Apache Spark. Вони необхідні для підтримки стабільності системи, підвищення її продуктивності та адаптації до різних умов експлуатації [6, 10].

У разі розподілених обчислень обробка даних виконується на множині вузлів, які працюють разом. Це створює три основні завдання:

– управління продуктивністю: необхідність оптимально використовувати обчислювальні ресурси для мінімізації витрат;

– забезпечення відмовостійкості: система повинна продовжувати роботу навіть при збої одного або кількох вузлів;

– контроль навантаження - потрібно відстежувати стан системи, щоб вчасно реагувати на проблеми.

Тестування зосереджено на перевірці того, наскільки система справляється із цими завданнями. Основні цілі включають у себе:

– оцінку продуктивності - аналіз часу виконання завдань та завантаження ресурсів;

– перевірку відмовостійкості - вивчення поведінки системи за збоїв;

– валідацію алгоритмів - перевірку ефективності роботи з більшими обсягами даних.

Моніторинг, у свою чергу, надає дані про стан системи реального часу. Це допомагає:

– виявляти вузькі місця у продуктивності, такі як перевантаження пам'яті чи мережі;

– аналізувати довгострокові тренди для покращення конфігурації системи.

Наприклад, при збільшенні часу виконання завдань моніторинг може вказати, що причина пов'язана з високим числом shuffle-операцій. Це сигналізує про необхідність змінити партиціонування даних або переглянути архітектуру завдання.

Тестування та моніторинг – нерозривна частина життєвого циклу розподілених систем. В Apache Spark вони забезпечують стабільність, ефективність та можливість масштабування під будь-які робочі навантаження.

4.1 Інструменти моніторингу в Apache Spark

Apache Spark надає ряд інструментів для моніторингу як вбудованих, так і сторонніх. Ці інструменти дозволяють у реальному часі стежити за станом системи, аналізувати продуктивність та виявляти вузькі місця.

Розглянемо три основні інструменти моніторингу, а саме:

– Spark Web UI - вбудований інтерфейс моніторингу для аналізу завдань та ресурсів кластера [10, 11];

– Ganglia - зручний інструмент для моніторингу великих кластерів [12];

– Prometheus та Grafana - зв'язка для збирання, зберігання та візуалізації метрик [13].

4.1.1 Spark Web UI

Spark Web UI являє собою вбудований інструмент моніторингу в Apache Spark, який надає інтуїтивно зрозумілий інтерфейс для аналізу продуктивності завдань, станів етапів обробки даних та використання ресурсів у реальному часі. Він доступний через веб-браузер і автоматично запускається під час виконання програми.

Spark Web UI пропонує такі ключові розділи моніторингу, як:

Jobs (Завдання) (Рис.4.1). Виконується моніторинг виконуваних, завершених та невдалі завдання. Для кожного завдання вказуються:

– час виконання;

– кількість опрацьованих даних;

– статус виконання (успішно чи помилка).

Даний розділ дозволяє оперативно виявляти завдання, які викликають затримки.

Stages, або Етапи (Рис.4.2). Завдання Spark розбиваються на етапи, які є мінімальними одиницями роботи, виконуваними паралельно. Тут забезпечується:

– відстежування чаус виконання кожного етапу;

– аналіз операцій, таких, як shuffle чи join, які можуть бути причиною затримок.

Storage (Сховище). Відображає стан даних, що зберігаються у кластері. Дозволяє відстежувати:

– кількість кешованих даних у пам'яті та на диску;

– рівень їхньої доступності (наприклад, реплікація).

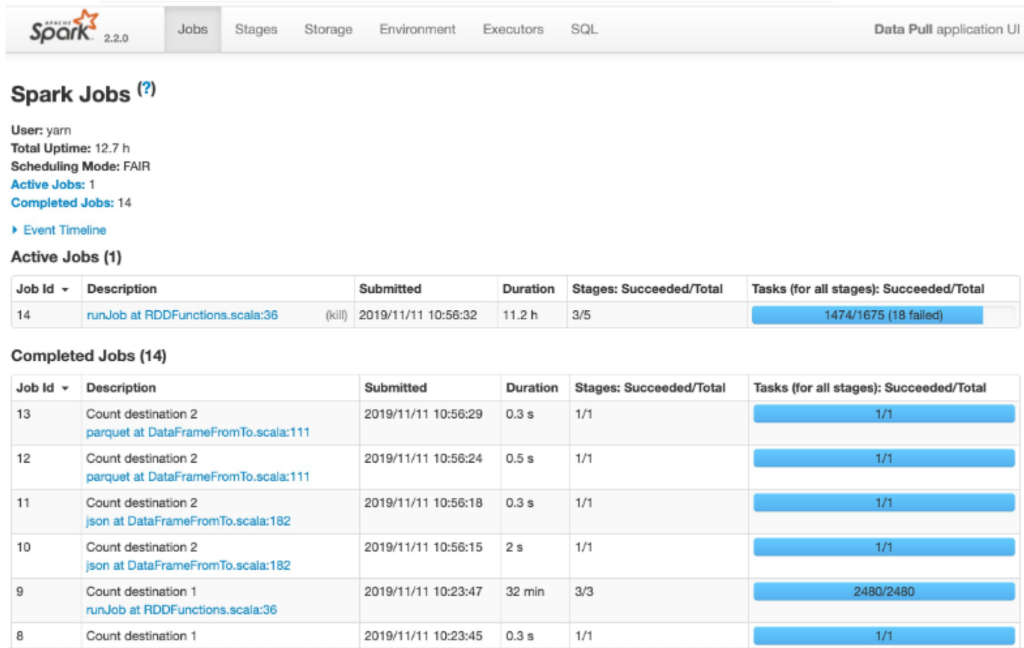

Рисунок 4.1 – Розділ «Jobs»

Розділ Executors (Виконавці).

Даний розділ надає інформацію про кожного виконавця, включаючи:

– статистику використання процесора та пам'яті;

– кількість завдань, виконаних тим чи іншим Executor’ом.

– поточний стан вузлів (активний, збій).

Environment, або Оточення (Рис.4.3).

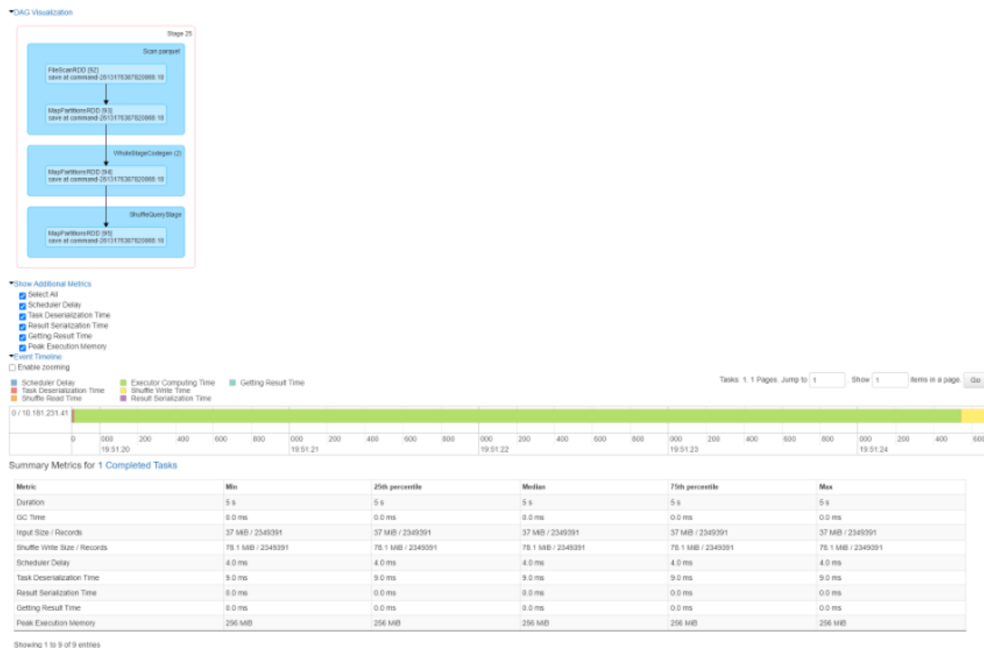

Рисунок 4.2 – Розділ «Stages»

Розділ містить системні властивості та параметри Spark, які були задані під час запуску програми.

Наприклад:

– установки пам'яті;

– кількість доступних потоків.

Рисунок 4.3 – Розділ «Environment»

Приклад використання Spark Web UI

Уявимо ситуацію, коли одне із завдань Spark-кластера виконується значно довше за інших. Через Web UI можна:

– перейти до розділу "Stages" та проаналізувати час виконання етапів;

– визначити які операції (shuffle, join) викликають затримку;

– переконатися, що навантаження рівномірно розподілено між вузлами, використовуючи розділ "Executors".

Налаштування доступу до Spark Web UI

Для локальних програм Spark Web UI доступний за адресою http://localhost:4040. У разі роботи в кластерному режимі доступ здійснюється через порт головного вузла, який вказується у логах програми.

Приклади проблем, які вирішуються за допомогою Spark Web UI

Однією з найчастіших проблем в Apache Spark є тривалі shuffle-операції, такі як join або groupBy. Ці операції вимагають передачі між вузлами і часто стають вузьким місцем продуктивності. Якщо у розділі «Stages» видно, що значний час йде на shuffle-операції, це може бути ознакою необхідності збільшення кількості партій або зміни логіки обробки даних.

Нерівномірне завантаження виконавців – ще одна поширена проблема.

Крім того, брак пам'яті, помітний у розділі «Storage», призводить до того, що дані переміщуються на диск замість зберігання в оперативній пам'яті. У таких випадках корисно збільшити виділену пам'ять для виконавців або оптимізувати обсяг даних, що кешуються.

4.1.2 Ganglia

Ganglia являє собою інструмент для моніторингу розподілених систем, який найкращим чином підходить для аналізу продуктивності великих кластерів, включаючи Apache Spark. Ganglia забезпечує візуалізацію метрик в реальному часі, масштабованість і довгострокове збереження даних для аналізу трендів [12].

Розглянемо основний функціонал Ganglia.

Даний засіб надає детальну інформацію про стан кластера. Основні функції засобу такі:

– візуалізація метрик. Відстежуються графіки завантаження процесора, використання пам'яті, мережної активності та дискового вводу-виводу для кожного вузла. Інакше кажучи, засіб дозволяє відстежувати загальну продуктивність системи;

– масштабованість. Інструмент може обробляти метрики навіть у кластерах з тисячами вузлів, забезпечуючи низький вплив на продуктивність системи;

– агрегація та збереження даних. Ganglia зберігає метрики для подальшого аналізу, що дозволяє оцінювати довгострокові тренди та виявляти закономірності.

Приклади використання Ganglia

1. Моніторинг вузлів кластера. Візуалізація допомагає швидко виявити вузли, які працюють на межі можливостей, що може бути спричинено високим завантаженням CPU або нестачею пам'яті. Це сигнал для перерозподілу завдань або додавання нових ресурсів кластер [12].

2. Аналіз мережної активності. Зі збільшенням обсягів даних, переданих через мережу (наприклад, в shuffle-операціях), графіки Ganglia дозволяють

визначити, наскільки ефективно використовується мережу, і виявити вузькі місця.

3. Довгостроковий аналіз. Збережені метрики дають можливість аналізувати тренди за тижні або місяці. Наприклад, можна побачити поступове збільшення навантаження на кластер через зростання обсягів даних.

Налаштування та інтеграція Ganglia з Apache Spark

Налаштування Ganglia для моніторингу Spark-кластера включає [12]:

1. Установку Ganglia на вузлах кластера.

2. Налаштування конфігураційного файлу gmond.conf, щоб збирати метрики Spark.

3. Підключення Ganglia до веб-інтерфейсу для візуалізації.

Графіки, створювані Ganglia (Рис.4.4), можна використовувати для порівняння навантаження на вузли. Наприклад, графік завантаження CPU може показати, що один із вузлів перевантажений завданнями, тоді як інші залишаються недовантаженими. Це допомагає виявити дисбаланс та оптимізувати розподіл завдань.

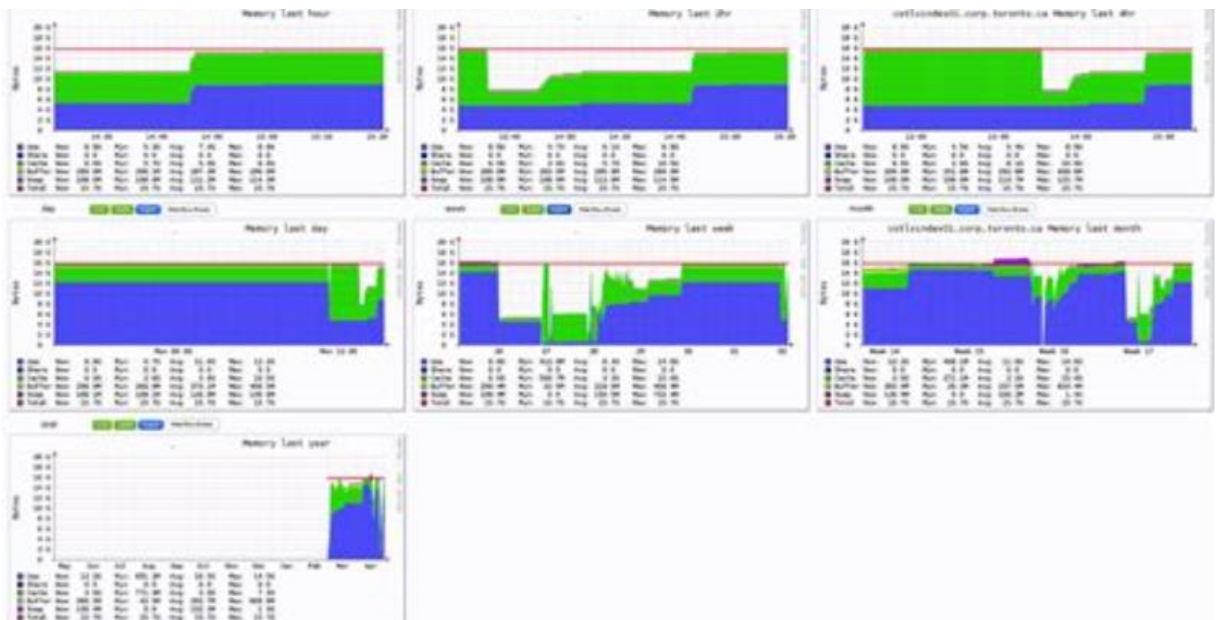

Рисунок 4.4 – Графіки Ganglia

Використання Ganglia у Spark-кластерах

Ganglia - інструмент, який надає можливості моніторингу розподілених систем, що дозволяє аналізувати продуктивність кластера на рівні метрик. Його легкість і масштабованість роблять його придатним для систем із великою кількістю вузлів. Ganglia може збирати та візуалізувати дані про стан системи,

таких як завантаження процесорів, використання пам'яті та мережної активності, а також зберігати ці метрики для подальшого аналізу.

Одним із важливих аспектів Ganglia є його здатність зберігати дані за тривалі періоди часу. Це дозволяє виявляти тренди, які важко помітити під час короткострокового моніторингу.

Наприклад, поступове збільшення навантаження може бути індикатором необхідності оптимізації shuffle-операцій в Apache Spark. Графіки, створювані Ganglia, допомагають візуалізувати ці зміни та надають інформацію для прийняття рішень.

Практичне застосування Ganglia часто пов'язані з аналізом продуктивності кластера. Наприклад, якщо один із вузлів кластера показує високе завантаження процесора або пам'яті, це може вказувати на нерівномірний розподіл завдань. У таких випадках Ganglia допомагає оперативно виявити проблему та вжити заходів, таких як зміна рівня партиціонування даних або додавання нових ресурсів у кластер.

Для інтеграції Ganglia із Spark необхідно налаштувати конфігураційний файл gmond.conf. Цей файл визначає параметри збору метрик та взаємодії вузлів. Приклад конфігурації може включати вказівку адреси кластера, порту для передачі даних і списку метриків, що збираються [14]:

```
cluster {
name = "SparkCluster"
owner = "Admin"
url = "http://localhost/ganglia"
}
udp_send_channel {
host = 239.2.11.71
port = 8649
}
udp_recv_channel {
port = 8649
}
```

Після встановлення, Ganglia надає веб-інтерфейс для аналізу даних. Наприклад, графіки використання ресурсів допоможуть побачити аномалії, такі, як навантаження вузлів або збільшення часу виконання завдань. Це робить Ganglia зручним інструментом для моніторингу поточного стану системи та довгострокового аналізу.

4.1.3 Prometheus та Grafana

Prometheus та Grafana - це зв'язка інструментів, що широко використовується для моніторингу та візуалізації метрик (Рис.4.6).

Так, Prometheus відповідає за збирання та зберігання даних, а Grafana надає зручний інтерфейс для створення користувацьких графіків та панелей моніторингу. Разом ці інструменти забезпечують гнучкість та масштабованість, що робить їх популярним вибором для моніторингу Apache Spark [13].

Особливості Prometheus

Даний засіб надає потужну модель збору даних, що базується на pull-механізмі. Це означає, що інструмент самостійно опитує цілі (targets), збираючи метрики через HTTP. У випадку Spark метрики можуть включати використання пам'яті, процесорів, мережної активності, а також дані про завдання та етапи виконання.

Для налаштування Prometheus потрібен файл конфігурації prometheus.yml, у якому вказуються цілі збору даних. Приклад налаштування для моніторингу Spark-кластера [14]:

```
global:
scrape_interval: 15s
scrape_configs:
- job_name: 'spark'
static_configs:
- targets: ['localhost:4040']
```

Після налаштування Prometheus автоматично збирає метрики із зазначених цілей, та зберігає їх у своїй базі даних.

Рисунок 4.5 – Інтерфейс Prometheus

Особливості Grafana

Grafana надає візуальний інтерфейс для аналізу даних, зібраних Prometheus. За допомогою цього інструменту можна створювати динамічні графіки, комбінувати дані з різних джерел та налаштовувати алерти.

Інтерфейс Grafana дозволяє користувачам взаємодіяти з метриками, змінюючи часові рамки, додаючи фільтри а також створюючи складні дашборди.

Приклад налаштування панелі моніторингу Grafana включає вибір джерела даних (Prometheus) і додавання графіків для ключових метрик, таких як завантаження CPU і затримки у виконанні завдань.

Приклади застосування Prometheus та Grafana

Інструменти Prometheus та Grafana дозволяють вирішувати широкий спектр завдань моніторингу, наприклад [13]:

1. Відстеження стану кластера у реальному часі. Grafana надає графіки, оновлювані у реальному часі, які показують стан вузлів, завантаження ресурсів та прогрес виконання завдань.

2. Виявлення вузьких місць. Використовуючи метрики з Prometheus, можна визначити, які вузли чи операції викликають затримки. Наприклад, високе навантаження на мережу в shuffle-операціях може вказувати на необхідність оптимізації партиціонування.

3. Налаштування алертів. Grafana дозволяє налаштувати сповіщення для критичних ситуацій, таких як перевищення рівня завантаження CPU або нестача пам'яті на вузлах.

Інтеграція Prometheus та Grafana зі Spark

Інтеграція починається з налаштування експортерів метрик у Spark, щоб дані могли бути зібрані Prometheus. Потім Grafana підключається до Prometheus як джерело даних. Після налаштування стає можливим створювати дашборди та візуалізувати метрики Spark, такі як затримки етапів виконання, використання пам'яті та процесорів.

4.2 Тестування продуктивності системи

Тестування продуктивності - це методологічний процес, спрямований на вимірювання та аналіз ключових характеристик системи, таких як час виконання завдань, використання ресурсів та поведінка в умовах високих навантажень. Основна мета — визначити, чи відповідає система заявленим вимогам та очікуванням у реальних робочих умовах [10].

4.2.1 тапи тестування продуктивності

1. Визначення цілей. Необхідно зрозуміти, які метрики є критичними для оцінки (наприклад, час виконання завдань, обсяг shuffle-операцій, використання пам'яті).

2. Створення тестового сценарію. Такий сценарій повинен моделювати реальні робочі умови, включаючи тип завдань, структуру даних та передбачувані навантаження.

3. Збір метрик. Під час тестування необхідно збирати дані про стан системи, такі як використання CPU, обсяг пам'яті, затримки виконання завдань.

4. Аналіз результатів. На основі зібраних даних виявляються вузькі місця, проблеми з масштабованістю чи продуктивністю.

5. Оптимізація та ретест. Після усунення проблем тести повторюються, щоб підтвердити покращення.

4.2.2 Роль моніторингу у тестуванні продуктивності

Розподілені системи, такі як Apache Spark, обробляють великі обсяги даних, розподіляючи завдання між множиною вузлів. Однак, їхня продуктивність може залежати від багатьох факторів: від конфігурації параметрів кластера до структури вхідних даних [10].

Тестування допомагає відповісти на важливі питання:

– як система поводиться під навантаженням?

– які вузькі місця існують у поточній конфігурації?

– наскільки ефективно працюють алгоритми, що використовуються?

Один з типів тестування - навантажувальне тестування.

Наприклад, обробка RDD з мільйонами записів або виконання множини паралельних завдань дозволяє зрозуміти, як кластер справляється з піковим навантаженням. Особливу увагу варто приділяти метрикам:

– середня та максимальна затримка виконання завдань;

– використання пам'яті та процесорів на вузлах;

– обсяги даних, що передаються через мережу (shuffle).

Розглянемо приклад коду для тестування навантаження [10]:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("LoadTest").getOrCreate()
rdd = spark.sparkContext.parallelize(range(1, 1000000), numSlices=100)
result = rdd.map(lambda x: x ** 2).reduce(lambda x, y: x + y)
```

Далі виконаємо огляд ще одного виду тестування - перевірка відмовостійкості.

Це тестування спрямоване на вивчення поведінки системи при збоях. Наприклад, штучне відключення вузла, або перевищення ліміту пам'яті на одному з виконавців дозволяє перевірити, як швидко кластер перерозподіляє завдання та відновлює свою працездатність. Розробниками рекомендується моделювати збої, такі, як:

– вимкнення вузла через кластерний менеджер (Spark Standalone, YARN);

– генерація помилок усередині завдання.

– Окремо слід розглянути тестування алгоритмів та конфігурації.

– У ході даного тестування перевіряється вплив змін параметрів на продуктивність, наприклад:

– вплив кількості партицій на продуктивність. Слід брати до уваги, що збільшення кількості партій даних може прискорити shuffle-операції, але збільшити навантаження на мережу;

– налаштування параметрів пам'яті (spark.executor.memory, spark.driver.memory). Це дозволяє уникнути помилки OutOfMemoryError;

– порівняння продуктивності RDD і DataFrame, що показує, який підхід ефективніший у конкретній задачі.

Приклад [10]:

```
# Тест зі зміною числа партицій
rdd = spark.sparkContext.parallelize(range(1, 1000000)).repartition(200)
result = rdd.map(lambda x: x ** 2).reduce(lambda x, y: x + y)
```

У ході тестування необхідно ураховувати наступне:

– При тестуванні навантаження важливо враховувати реальні сценарії використання. Наприклад, якщо завдання включають складні трансформації (join, groupBy), тести мають це імітувати.

– Перевірка стійкості до відмов вимагає моделювання реальних збоїв, таких як відключення вузлів або перевищення лімітів пам'яті.

– Тестування алгоритмів слід проводити зі зміною ключових параметрів Spark, таких як spark.sql.shuffle.partitions або executor.memory, щоб знайти оптимальні значення для різних завдань.

4.3 Оптимізація та аналіз результатів моніторингу

Моніторинг відіграє ключову роль у підтримці високої продуктивності Apache Spark. Дані, отримані під час моніторингу, надають повну картину про стан кластера, використання ресурсів та ефективність виконання завдань.

Однією з частих проблем є навантаження окремих вузлів, що може бути викликане нерівномірним розподілом завдань [10].

Наприклад, один вузол може обробляти занадто багато даних, тоді як інші залишаються недовантаженими. Моніторинг дає змогу своєчасно виявити такі ситуації. Інша поширена проблема - тривалі shuffle-операції, пов'язані з великим обсягом даних, що передаються між вузлами. Це може бути викликане неправильним партиціонуванням даних або недостатньою кількістю ресурсів для обробки.

Дані моніторингу допомагають у виявленні проблем, а також у оцінці ефективності внесених змін.

Наприклад, після оптимізації параметрів системи можна перевірити, чи скоротився час виконання завдань, чи знизилося навантаження на мережу. Таким чином, моніторинг стає невід'ємним інструментом постійного поліпшення продуктивності системи.

Методи оптимізації продуктивності

Оптимізація продуктивності Apache Spark базується на правильному використанні ресурсів та ефективній обробці даних. Дані, зібрані під час моніторингу, дозволяють вибирати найбільш підходящі методи оптимізації для усунення виявлених вузьких місць [10].

1. Кешування та використання persist()

Кешування (cache) та persist() дозволяють зберігати проміжні результати обчислень у пам'яті чи диску. Це особливо корисно, якщо дані використовуються багаторазово в рамках однієї програми. Наприклад, під час обробки великих RDD кешування скорочує час повторних обчислень.

Приклад використання кешування [10]:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("CachingExample").getOrCreate()
rdd = spark.sparkContext.parallelize(range(1, 1000000))
cached_rdd = rdd.map(lambda x: x * 2).
print(cached_rdd.count()) # Перше обчислення
```

```
print(cached_rdd.sum()) # Повторне використання кешованих даних
```

2. Налаштування параметрів кластера

Для підвищення продуктивності важливо коректно налаштувати параметри Spark. До ключових параметрів відносяться:

– spark.executor.memory та spark.driver.memory:збільшують обсяг пам'яті, доступної виконання завдань. Це допомагає уникнути OutOfMemoryError;

– spark.sql.shuffle.partitions:визначає кількість партій для shuffle-операцій. Збільшення значення покращує паралелізм, але збільшує навантаження на мережу.

Розглянемо приклад налаштування параметрів [10]:

```
spark = SparkSession.builder \
.appName("TuningExample") \
.config("spark.executor.memory", "4g") \
.config("spark.sql.shuffle.partitions", "200") \
.getOrCreate()
```

3. Оптимізація даних

Ефективна обробка даних безпосередньо впливає на продуктивність:

– партиціювання. Зменшення обсягу даних у кожній партиції допомагає уникнути навантаження окремих вузлів. Проте надто маленькі партиції можуть збільшити накладні витрати;

– зменшення обсягів shuffle. Використання функцій map-side join або попередньої агрегації допомагає мінімізувати обсяг даних, що передаються між вузлами.

– Розглянемо приклад прискорення shuffle-операцій.

– Shuffle-операції, такі, як join і groupBy, часто стають вузьким місцем у продуктивності через обсяг даних, що передаються між вузлами. Щоб зменшити час виконання shuffle-операцій, необхідно:

– збільшити кількість партицій, щоб знизити навантаження кожен вузол;

– використовувати map-side join, які мінімізують обсяги даних, що передаються.

На рівні записів конфігурації тут маємо наступне [10]:

```
rdd1 = spark.sparkContext.parallelize([(1, "a"), (2, "b"), (3,
"c")])
```

```
rdd2 = spark.sparkContext.parallelize([(1, "x"), (2, "y"), (3,
"z")]))
# Використання join (shuffle)
result = rdd1.join(rdd2)
print(result.collect())
```

У свою чергу, для уникнення помилки OutOfMemoryError, яка часто виникає через брак пам'яті у виконавців.

Рішення включають:

- збільшення пам'яті для виконавців за допомогою spark.executor.memory;
- кешування даних скорочення обсягів повторних обчислень.

На рівні записів файлу конфігурації це може бути показано наступним чином [10]:

```
spark = SparkSession.builder \
.appName("MemoryOptimization") \
.config("spark.executor.memory", "4g") \
.getOrCreate()
data = spark.range(1, 1000000).repartition(100).cache()
print(data.count()) # Перше обчислення
```

У свою чергу, окремо виконується усунення «вузьких місць» у мережевій активності.

Так, високе навантаження на мережу часто пов'язане з shuffle-операціями. Зменшення обсягів переданих даних досягається рахунок попередньої агрегації чи фільтрації.

Приклад [10]:

```
rdd = spark.sparkContext.parallelize([(1, 2), (1, 3), (2, 4),
(3, 5)])
aggregated_rdd = rdd.reduceByKey(lambda x, y: x + y)
# Агрегація до передачі
```

4.4 Автоматизація тестування та моніторингу

Для спрощення аналізу та підвищення точності рекомендується використовувати автоматизацію, зокрема:

– інструменти, такі як pytest-spark, допомагають створювати та виконувати тести для перевірки продуктивності;

– системи автоматичного моніторингу (Prometheus) з налаштуванням алертів дозволяють оперативно реагувати на критичні зміни у системі.

Окрім цього, збереження та аналіз історичних даних дозволяють виявляти тренди, такі як збільшення навантаження на кластер або зниження продуктивності додатків. Ці дані допомагають приймати обґрунтовані рішення щодо масштабування системи або зміни архітектури обробки даних.

ВИСНОВКИ

У відповідності з завданням, у ході виконання кваліфікаційної роботи було досліджено:

– функціонал та архітектурні особливості фреймворку розподілених обчислень Apache Spark;

– порядок розгортання Apache Spark, як базису для організації обчислювального кластеру;

– специфіку обробки Великих даних в обчислювальному кластері за принципом MapReduce.

Зокрема, було виявлено, що:

продуктивність MapReduce-обчислень на базі Apache Spark для випадків обробки складних масивів даних є вищою порівняно з середовищем Apache Hive;

Apache Spark може функціонувати у режимі Standalone, як окрема платформа, та у режимі Hadoop YARN у випадках, коли у кластері вже присутній Hadoop.

Окрім цього, детально було розглянуто приклади виконання завдань MapReduce у Apache Spark. При цьому, рішення розглядалися для середовищ Python та Scala.

У рамках дослідження підходів до оптимізації Apache Spark-кластерів було розглянуто:

інструменти моніторингу процесів, які мають перебіг у кластері на його окремих вузлах;

показники, які мають відстежуватися;

стратегії оптимізації Apache Spark.

Було показано, що збереження та наступний аналіз даних моніторингу за тривалий час дозволяє виявляти тренди, такі, як збільшення навантаження на кластер або зниження продуктивності додатків. Ці дані допомагають приймати обґрунтовані рішення щодо масштабування системи або зміни архітектури обробки даних.

Отже, усі завдання до кваліфікаційної роботи виконано у повній мірі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Introduction to Distributed Computing Environment (DCE) [Електронний ресурс] – Режим доступу: https://www.geeksforgeeks.org/introduction-to-distributed-computing-environment-dce/?ysclid=m6202tdfxz772020523

2. Parallel and Distributed Computing Concepts [Електронний ресурс] – Режим доступу: https://weshaan.hashnode.dev/parallel-and-distributed-computing-concepts

3. Big Data Processing 101: The What, Why, and How [Електронний ресурс] – Режим доступу: https://www.dataversity.net/big-data-processing-101/

4. Balusamy B., Abirami N., Kadry S., Gandomi A. Big Data: Concepts, Technology, and Architecture. - Wiley, 2021. - 368 p. ISBN: 978-1-119-70185-9

5. Maas G., Garillot F. Stream Processing with Apache Spark: Mastering Structured Streaming and Spark Streaming. – O'Reilly Media, 2019. – 450 p.

6. Wenig B., Guang-Yeu Lee D., Damji J., Das T. Learning Spark: Lightning-Fast Data Analytics. 2 Ed. - O’Reilly, 2020. - 399 p.

7. Ryza S., Laserson U., Owen S., Wills J. Advanced Analytics with Spark: Patterns for Learning from Data at Scale. - O’Reilly, 2015. - 276 p.

8. Перрен Ж. Спарк у дії. - ДМК Прес, 2021. - 636 с.

9. Spark SQL & DataFrames | Apache Spark [Електронний ресурс] – Режим доступу: https://spark.apache.org/sql/

10. Monitoring and Instrumentation [Електронний ресурс] – Режим доступу: https://spark.apache.org/docs/latest/monitoring.html

11. Web UI [Електронний ресурс] – Режим доступу: https://spark.apache.org/docs/latest/web-ui.html

12. Ganglia Cluster Monitoring Made Easy [Електронний ресурс] – Режим доступу: https://softpanorama.org/Admin/Monitoring/ganglia.shtml

13. Grafana | Prometheus [Електронний ресурс] – Режим доступу: https://prometheus.io/docs/visualization/grafana/

14. Get started with Grafana and Prometheus [Електронний ресурс] – Режим доступу: https://grafana.com/docs/grafana/latest/getting-started/get-started-grafana-prometheus/