Міністерство освіти і науки України
Чорноморський національний університет імені Петра Могили

Методична серія
Випуск № 491

Заснована в 2016 році

Г. В. Горбань

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ОБРОБКИ ВЕЛИКИХ ОБСЯГІВ ДАНИХ

Методичні рекомендації до виконання практичних робіт

ЧНУ
ВИДАВНИЦТВО
Миколаїв – 2026

УДК 004.4:004.6 (076)
Г 67

Рекомендовано до друку радою навчально-наукового медичного інституту від 14 травня 2026 року. Витяг з протоколу № 10

Рецензенти:
Євген Сіденко, кандидат технічних наук, доцент, завідувач кафедри інтелектуальних інформаційних систем ЧНУ імені Петра Могили.
Едуард Лисенков, д-р. фіз.-мат. наук, професор, професор кафедри фізики та математики ЧНУ імені Петра Могили.

Горбань Г. В.

Г 67 Програмне забезпечення обробки великих обсягів даних : метод. рек. до виконання практ. робіт / Г. В. Горбань. – Миколаїв : Вид-во ЧНУ ім. Петра Могили, 2026. – 104 с. – (Методична серія ; вип. 491).

Методичні рекомендації містять опис практичних робіт з дисципліни «Програмне забезпечення обробки великих обсягів даних», спрямованих на опанування сучасних засобів і технологій інженерії, зберігання, обробки та аналізу даних. У практичних роботах послідовно розглянуто використання мов SQL і Python, отримання даних із Web API, автоматизацію ELT-процесів за допомогою Apache Airflow, роботу з PostgreSQL, розподіленою файловою системою HDFS, платформами Hadoop і Apache Spark, а також побудову комплексних конвеєрів і платформ обробки даних із використанням CloudberryDB. Окрему увагу приділено встановленню, налаштуванню та розгортанню відповідного програмного забезпечення в операційній системі Linux. Методичні рекомендації призначено для здобувачів вищої освіти спеціальностей галузі знань F – «Інформаційні технології», а також можуть бути корисними здобувачам інших спеціальностей, які вивчають технології обробки великих обсягів даних.

УДК 004.4:004.6 (076)

ISSN 1811-492X

© Горбань Г. В.,
© ЧНУ ім. Петра Могили, 2026

Зміст

Вступ 4
Практична робота №1. «Використання мови SQL для інженерії даних» 5
Практична робота №2. «Використання мови Python для інженерії даних шляхом отримання даних з Web API» 11
Робота з налаштування ПЗ №1. «Встановлення Apache Airflow, PostgreSQL та Supervisor в операційній системі Linux» 19
Практична робота №3. «Автоматизація ELT-процесів у Apache Airflow: робота з PostgreSQL та API» 32
Робота з налаштування ПЗ №2. «Встановлення та розгортання Hadoop в операційній системі Linux» 46
Практична робота №4. «Робота з розподіленою файловою системою HDFS» 60
Практична робота №5. «Аналіз даних CSV-файлу за допомогою Hadoop та Python» 65
Робота з налаштування ПЗ №3. «Встановлення Apache Spark в операційній системі Linux» 72
Практична робота №6. «Робота зі структурованими даними за допомогою SparkSQL» 77
Практична робота №7. «Побудова data pipeline за допомогою Spark, HDFS та Apache Airflow» 82
Робота з налаштування ПЗ №4. «Встановлення та запуск CloudberryDB зі створенням кластера» 87
Практична робота №8. «Побудова повноцінної платформи даних за допомогою HDFS, Spark та Cloudberry» 102

Вступ

Наш світ є дуже складним. Ця складність зумовлює генерування постійно зростаючого обсягу даних, які потрібно зберігати та аналізувати. Швидкість генерування даних не виявляє ознак уповільнення. Різноманітність даних поширюється на нові області, які ніколи не були доступні для аналізу. Взаємодія між людьми, що використовують медіаплатформи, автоматизація процесів та агрегація даних, що надходять з різних джерел, створюють Інтернет речей. Інтернет речей (Internet of Things, IoT) не тільки приєднує датчики до існуючих речей, він створює ринок нових пов'язаних речей. Усі ці пов'язані речі генерують дані. Це додає немисливу кількість великих даних, що називається Big Data.

Експонентне зростання даних створило нову сферу інтересів у галузі технологій та бізнесу під назвою «Big Data». Загалом, набір даних або бізнеспроблема належать до класифікації Big Data, коли її дані настільки великі або складні, що їх стає неможливим зберігати, обробляти та аналізувати, використовуючи традиційні підходи до зберігання та аналізу даних.

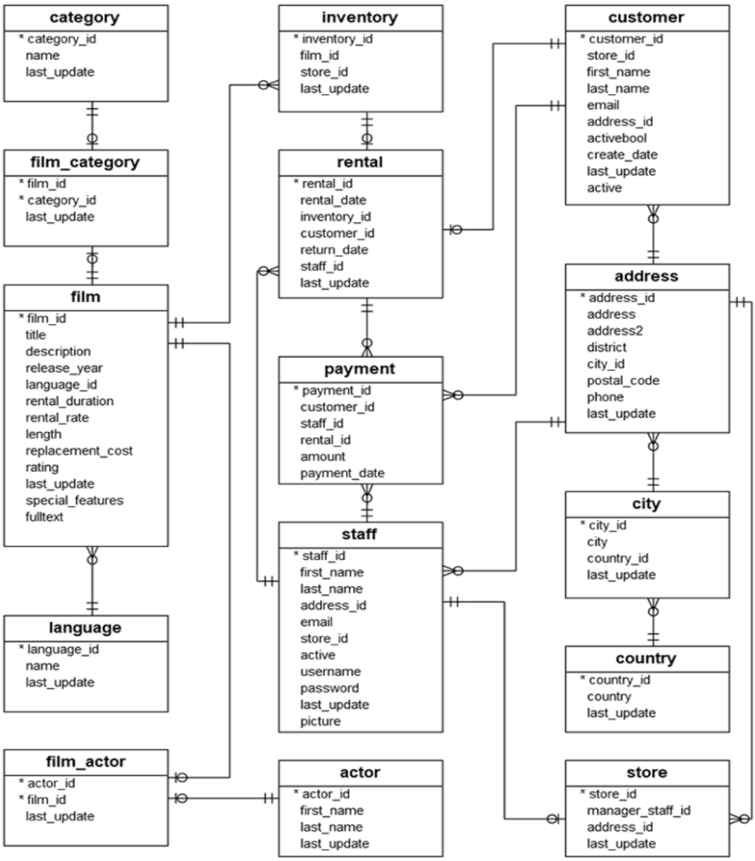
Аналітика великих даних - це систематична обробка та аналіз великих обсягів даних і складних наборів даних, відомих як «великі дані», з метою отримання цінної інформації. Аналітика великих даних дозволяє виявити тенденції, закономірності та кореляції у великих обсягах необроблених даних, щоб допомогти аналітикам приймати рішення на основі даних. Цей процес дозволяє організаціям використовувати експоненціально зростаючі обсяги даних, що генеруються з різних джерел, включаючи датчики Інтернету речей (IoT), соціальні мережі, фінансові транзакції та смарт-пристрої, для отримання дієвої інформації за допомогою передових аналітичних методів.

Основна відмінність між аналітикою великих даних і традиційною аналітикою полягає в типі даних, що обробляються, та інструментах, які використовуються для їх аналізу. Традиційна аналітика має справу зі структурованими даними, які зазвичай зберігаються в реляційних базах даних. Цей тип баз даних допомагає забезпечити, щоб дані були добре організовані та легкі для розуміння комп'ютером. Традиційна аналітика даних спирається на статистичні методи та інструменти, такі як мова структурованих запитів (SQL) для запитів до баз даних.

Аналітика великих даних включає величезні обсяги даних у різних форматах, включаючи структуровані, напівс труктуровані та неструктуровані дані. Складність цих даних вимагає більш складних методів аналізу. Аналітика великих даних використовує передові методи, такі як машинне навчання та інтелектуальний аналіз даних, для вилучення інформації зі складних наборів даних. Для управління величезним обсягом даних часто потрібні розподілені системи обробки, такі як Hadoop.

Практична робота №1.
«Використання мови SQL для інженерії даних»

Pagila – база даних прокату DVD для демонстрації можливостей PostgreSQL.



Мал. 1 – Схема бази даних Pagila

У базі даних прокату DVD є 15 таблиць:

- **actor** – зберігає дані про акторів, включаючи ім'я та прізвище;
- **film** – зберігає дані про фільми, такі як назва, рік випуску, тривалість, рейтинг тощо;
- **film_actor** – зберігає зв'язки між фільмами та акторами;
- **category** – зберігає дані про категорії фільмів;

- **film_category** – зберігає зв'язки між фільмами та категоріями;
- **store** – містить дані про магазин, включно з менеджерами та адресою;
- **inventory** – зберігає дані про облік;
- **rental** – зберігає дані про прокат;
- **payment** – зберігає платежі клієнтів;
- **staff** – зберігає дані про персонал;
- **customer** – зберігає дані про клієнтів;
- **address** – зберігає дані про адреси для персоналу та клієнтів;
- **city** – зберігає назви міст;
- **country** – зберігає назви країн.

Для імпорту бази даних необхідно, створивши попередньо порожню базу даних, запустити файл **pagila-schema.sql** в PgAdmin 4, а потім запустити **pagila-insert-data.sql**.

Приклади запитів SQL для аналізу даних

1. Обчислити кількість клієнтів у кожному місті.

```
SELECT city.city, COUNT(customer.customer_id) AS customer_count
FROM customer
JOIN address ON customer.address_id = address.address_id
JOIN city ON address.city_id = city.city_id
GROUP BY city.city
ORDER BY customer_count DESC;
```

	city	customer_count
1	London	2
2	Aurora	2
3	Tokat	1
4	Atlixco	1
5	Mukateve	1
6	Pontianak	1
7	Gatineau	1
8	Saint-Denis	1
9	Molodetno	1
10	Yingkou	
11	Changhwa	1

1-500 of 501+

Мал. 2 – Результат виконання SQL-запиту для визначення кількості клієнтів у містах

2. Обчислити загальний дохід від оренди фільмів у кожному місті

```
SELECT city.city, SUM(payment.amount) AS total_revenue
FROM payment
JOIN rental ON payment.rental_id = rental.rental_id
JOIN customer ON rental.customer_id = customer.customer_id
JOIN address ON customer.address_id = address.address_id
JOIN city ON address.city_id = city.city_id
GROUP BY city.city
ORDER BY total_revenue DESC;
```

	city	total_revenue
1	Cape Coral	221.55
2	Saint-Denis	216.54
3	Aurora	198.5
4	Molodetno	195.58
5	Apeldoorn	194.61
6	Santa Brbara dOeste	194.61
7	Qomsheh	186.62
8	London	180.52
9	Ourense (Orense)	177.6
10	Bijapur	
11	Tanza	175.58

1-500

Мал. 3 – Результат виконання SQL-запиту для визначення загального доходу від оренди фільмів у містах

3. Вивести клієнтів, які здійснили більше 15 оплат

```
SELECT customer.first_name, customer.last_name, COUNT(payment.payment_id) AS payment_count
FROM customer
JOIN payment ON customer.customer_id = payment.customer_id
GROUP BY customer.customer_id
HAVING COUNT(payment.payment_id) > 15
ORDER BY payment_count DESC;
```

	first_name	last_name	payment_count
1	ELEANOR	HUNT	46
2	KARL	SEAL	45
3	CLARA	SHAW	42
4	MARCIA	DEAN	42
5	TAMMY	SANDERS	41
6	SUE	PETERS	40
7	WESLEY	BULL	40
8	MARION	SNYDER	39
9	RHONDA	KENNEDY	39
10	TIM	CARY	39
11	ELIZABETH	BROWN	38

Мал. 4 – Результат виконання SQL-запиту для визначення клієнтів, які здійснили більше 15 оплат

4. Знайти фільми, які не були орендовані жодного разу

```
SELECT title
FROM film
WHERE film_id NOT IN (
    SELECT DISTINCT inventory.film_id
    FROM rental
    JOIN inventory ON rental.inventory_id = inventory.inventory_id
);
```

	title
1	ALICE FANTASIA
2	APOLLO TEEN
3	ARGONAUTS TOWN
4	ARK RIDGEMONT
5	ARSENIC INDEPENDENCE
6	BOONDOCK BALLROOM
7	BUTCH PANTHER
8	CATCH AMISTAD
9	CHINATOWN GLADIATOR
10	CHOCOLATE DUCK
11	COMMANDMENTS EXPRESS

Мал. 5 – Результат виконання SQL-запиту для пошуку фільмів, які не були орендовані жодного разу

5. Знайти середню тривалість фільмів у кожній категорії

```
SELECT category.name AS category_name, AVG(film.length) AS average_length
FROM category
JOIN film_category ON category.category_id = film_category.category_id
JOIN film ON film_category.film_id = film.film_id
GROUP BY category.name
ORDER BY average_length DESC;
```

	category_name	average_length
1	Sports	128.2027027027027
2	Games	127.8360655737704918
3	Foreign	121.6986301369863014
4	Drama	120.8387096774193548
5	Comedy	115.8275862068965517
6	Family	114.7826086956521739
7	Music	113.6470588235294118
8	Travel	113.3157894736842105
9	Horror	112.4821428571428571
10	Classics	111.666666666
11	Action	111.609375

16 rows

Мал. 6 – Результат виконання SQL-запиту для визначення середньої тривалості фільмів у категоріях

6. Знайти 5 акторів, які знялися в найбільшій кількості фільмів

```
SELECT actor.first_name, actor.last_name, COUNT(film_actor.film_id) AS film_count
FROM actor
JOIN film_actor ON actor.actor_id = film_actor.actor_id
GROUP BY actor.actor_id
ORDER BY film_count DESC
LIMIT 5;
```

	first_name	last_name	film_count
1	GINA	DEGENERES	42
2	WALTER	TORN	41
3	MARY	KEITEL	40
4	MATTHEW	CARREY	39
5	SANDRA	KILMER	37

Мал. 7 – Результат виконання SQL-запиту для визначення п’яти акторів із найбільшою кількістю фільмів

7. Знайти найдовший фільм у кожній категорії

```
SELECT category.name AS category_name, film.title, film.length
FROM film
JOIN film_category ON film.film_id = film_category.film_id
JOIN category ON film_category.category_id = category.category_id
WHERE film.length = (
   SELECT MAX(length)
   FROM film AS f
   JOIN film_category AS fc ON f.film_id = fc.film_id
   WHERE fc.category_id = film_category.category_id
)
ORDER BY category.name;
```

	category_name	title	length
1	Action	DARN FORRESTER	185
2	Action	WORST BANGER	185
3	Animation	POND SEATTLE	185
4	Animation	GANGS PRIDE	185
5	Children	FURY MURDER	178
6	Children	WRONG BEHAVIOR	178
7	Classics	CONSPIRACY SPIRIT	184
8	Comedy	CONTROL ANTHEM	185
9	Documentary	YOUNG LANGUAGE	183
10	Documentary	WIFE TURN	183
11	Drama	JACKET FRISCO	181

23 rows

Мал. 8 – Результат виконання SQL-запиту для визначення найдовшого фільму в кожній категорії

Завдання

Використовуючи описану вище базу даних, напишіть запити для того, щоб:

1. вивести кількість фільмів у кожній категорії, відсортувати за спаданням;
2. вивести 10 акторів, чиї фільми найбільше орендували, відсортувати за спаданням;
3. вивести категорію фільмів, яку витратили найбільше грошей;
4. вивести назви фільмів, яких немає в inventory;
5. вивести топ 3 акторів, які найбільше з'являлися у фільмах категорії “Children”. Якщо у кількох акторів однакова кількість фільмів, вивести всіх;
6. вивести міста з кількістю активних та неактивних клієнтів (активний – customer.active = 1). Відсортувати за кількістю неактивних клієнтів за спаданням;
7. вивести категорію фільмів, у якої найбільша кількість годин сумарної оренди в містах (customer.address_id в цьому city), і які починаються на букву “a”. Те саме зробити для міст у яких є символ “-”. Написати все в одному запиті.

Практична робота №2. «Використання мови Python для інженерії даних шляхом отримання даних з Web API»

Open Exchange Rates надає простий, легкий і портативний JSON API з поточними та історичними курсами іноземної валюти (форекс) через простий і легкий для інтеграції API у форматі JSON. Дані відстежуються та змішуються алгоритмічно з кількох надійних джерел, забезпечуючи справедливу та неупереджену послідовність.

Курси обміну, опубліковані через Open Exchange Rates API, збираються від багатьох надійних постачальників, змішуються разом і подаються у форматі JSON для використання всіма. Немає складних запитів, незрозумілих методів аутентифікації чи довгострокових контрактів.

Ставки на кінець дня доступні за попередні періоди для всіх днів, починаючи з 1 січня 1999 року.

Підключення до API

Дані надаються у форматі JSON через простий інтерфейс на основі URL-адреси через HTTPS, що дає змогу використовувати тарифи будь-яким способом.

Типи підключення

Будь-яка мова чи програмне забезпечення, яке може надсилати запити HTTP або отримувати веб-адреси, може отримати доступ до API

(наприклад, ви можете відвідати будь-який із маршрутів API у своєму браузері, щоб перевірити, чи вони працюють належним чином).

Для інтеграції ви можете використовувати будь-яку бібліотеку, яка вам потрібна. Це залежить від середовища розробки. Є посібники та широкий спектр доступних інтеграцій з відкритим кодом, які також описані в нашій документації.

URL-адреси (маршрути) запитуються один раз через HTTPS і доставляють усі свої дані за один раз, як і звичайний веб-запит.

Наразі ми не підтримуємо веб-сокети, веб-хуки чи будь-які інші з'єднання типу Keep-Alive або Push-повідомлення – іншими словами, якщо вам потрібні свіжі дані, ви просто запитуєте їх у нашого сервера. Ми розглядаємо ці методи для майбутньої версії нашого API, тому, будь ласка, надішліть нам електронний лист, якщо це цікаво.

Формат URL-адреси

Базовий шлях API https://openexchangerates.org/api/.

Маршрути/кінцеві точки API потім додаються до цього базового шляху, ось так:

https://openexchangerates.org/api/
latest.json
currencies.json
historical/2013-02-16.json

Параметри запиту (такі як ваш ідентифікатор програми, запитана базова валюта або зворотний виклик JSONP) додаються як параметри запиту GET, наприклад:

https://openexchangerates.org/api/latest.json
?app_id=YOUR_APP_ID
&base=GBP
&callback=someCallbackFunction

Якщо ваш запит дійсний і дозволений, ви отримаєте відповідь у форматі JSON для роботи. Якщо щось не так із запитом, ви отримаєте повідомлення про помилку.

Формати відповідей API

Відповіді доставляються через HTTPS у форматі простого тексту JSON (нотація об'єктів JavaScript), готового до використання, незалежно від того, що вимагає ваша інтеграція.

Цей формат не обмежує, як і де ви можете будь-яким чином використовувати дані: JSON – це просто швидкий, простий і легкий механізм доставки, який підтримується всіма основними мовами та фреймворками.

Ми розробили ці відповіді, щоб їх було легко інтегрувати в різні програми та програмне забезпечення. За потреби ви також можете програмно конвертувати дані JSON у формат CSV/електронну таблицю чи будь-який інший формат.

Існує кілька основних стилів/форматів відповідей: останні/історичні курси, список валют, часові ряди та конвертація валют. Вони детально описані окремо на кожній відповідній сторінці документації, і ви можете побачити приклад (для latest.json) нижче.

Ось приклад базового запиту API для всіх останніх ставок відносно доларів США (за замовчуванням):

https://openexchangerates.org/api/latest.json?app_id=YOUR_APP_ID

Під час запиту на цю URL-адресу (якщо ваш ідентифікатор програми дійсний) ви отримаєте об’єкт JSON, що містить мітку часу UNIX (UTC, секунди), базову валюту (3-літерний код ISO) і об’єкт тарифів із парами символ:значення відносно необхідна базова валюта:

```
{
  disclaimer: "https://openexchangerates.org/terms/",
  license: "https://openexchangerates.org/license/",
  timestamp: 1449877801,
  base: "USD",
  rates: {
    AED: 3.672538,
    AFN: 66.809999,
    ALL: 125.716501,
    AMD: 484.902502,
    ANG: 1.788575,
    AOA: 135.295998,
    ARS: 9.750101,
    AUD: 1.390866,
    /* ... */
  }
}
```

Формат відповіді однаковий для запитів історичних даних (historical/YYYY-MM-DD.json).

Інші маршрути API, наприклад currencies.json, time-series.json і convert/, мають інший формат запиту та відповіді. Перегляньте відповідні сторінки, щоб отримати докладні відомості та приклади.

Кінцеві точки API

1. /currencies.json (https://openexchangerates.org/api/latest.json) – отримання списку JSON усіх символів валют, доступних через Open Exchange Rates API, разом із їхніми повними назвами для використання у вашій інтеграції.

У цьому списку завжди відображатимуться валюти, доступні за останніми курсами (вказані як їхні 3-літерні коди).

Стандартною відповіддю є об'єкт JSON (наприклад, хеш/ словник/ асоціативний масив), де кожна пара ключ:значення представляє символ валюти та відображувану назву одиниці (в однині та з великої літери).

Ви можете включити список неофіційних, чорного ринку та альтернативних цифрових валют за допомогою параметрів show_alternative або only_alternative:

https://openexchangerates.org/api/currencies.json?show_alternative=1

Зверніть увагу, що альтернативні тарифи не завжди використовують трибуквений код.

Ви можете знайти більше інформації про наш неофіційний, чорний ринок та альтернативні цифрові валюти тут.

Таблиця 1 – Параметри запиту кінцевої точки /currencies.json Open Exchange Rates API

app_id	string	Ваш унікальний ідентифікатор застосунку (обов'язково)
prettyprint	boolean	Відповідь для відладки (розмір відповіді буде значно більшим)
show_alternative	boolean	Включити альтернативні валюти
show_inactive	boolean	Включити історичні/неактивні валюти

```
import requests

url = "https://openexchangerates.org/api/currencies.json?prettyprint=false&show_alternative=false&show_inactive=false&app_id=*****************"
//Реальний app_id у прикладі коду не вказано, потрібно буде зареєструватись на сервісі, для виконання завдань достатньо безкоштовного id

headers = {"accept": "application/json"}
response = requests.get(url, headers=headers)
print(response.text)
```

2. /latest.json (https://openexchangerates.org/api/latest.json) – отримання найновіші курси обміну, доступних в Open Exchange Rates API.

Найпростіший маршрут у API, latest.json, надає стандартний об’єкт відповіді, що містить усі курси конвертації для всіх доступних наразі символів/валют, позначених їхніми 3-літерними кодами валют ISO за міжнародним стандартом.

Останні тарифи завжди будуть найновішими даними, доступними у вашому плані.

Основна властивість надає 3-буквений код валюти, до якого відносяться всі надані курси обміну. Ця базова валюта також вказана в об’єкті курсів за замовчуванням (наприклад, "USD": 1).

Властивість rates – це об’єкт (хеш/словник/асоціативний масив), що містить усі конвертаційні або обмінні курси для всіх доступних (або запитуваних) валют, позначених їх міжнародними стандартними 3-літерними кодами валют. Усі значення відносяться до 1 одиниці запитаної базової валюти.

Властивість timestamp вказує час (UNIX), коли були опубліковані курси.

Таблиця 2 – Параметри запиту кінцевої точки /latest.json Open Exchange Rates API

app_id	string	Ваш унікальний ідентифікатор застосунку
base	string	Змінити базову валюту (3-буквений код, за замовчуванням: USD)
symbols	string	Обмежити результати певними валютами (список 3-буквених кодів через кому)
prettyprint	boolean	Встановіть значення false, щоб зменшити розмір відповіді (видаляє пробіли)
show_alternative	boolean	Розширити значення, що повертаються, альтернативними курсами, курсами на чорному ринку та цифровими курсами валют

3. /historical/*.json

(https://openexchangerates.org/api/historical/:date.json)– отримання

історичних курсів обміну за будь-яку дату, доступну в Open Exchange Rates API, наразі починаючи з 1 січня 1999 року.

Подібно до latest.json, маршрут /historical/ надає стандартний об'єкт відповіді, що містить усі курси конвертації для всіх доступних символів/валют на вказану вами дату, позначених їхніми 3-літерними кодами валют ISO за міжнародним стандартом.

Повернуті історичні курси – це останні опубліковані значення для певного дня UTC (до 23:59:59 UTC включно), за винятком поточної дати UTC.

Основна властивість надає 3-буквений код валюти, до якого відносяться всі надані курси обміну. Ця базова валюта також вказана в об'єкті курсів за замовчуванням (наприклад, "USD": 1).

Властивість rates – це об'єкт (хеш/словник/асоціативний масив), що містить усі конвертаційні або обмінні курси для всіх доступних (або запитуваних) валют, позначених їх міжнародними стандартними 3-літерними кодами валют. Усі значення відносяться до 1 одиниці запитаної базової валюти.

Властивість timestamp вказує час (UNIX), коли були опубліковані курси.

Таблиця 3 – Параметр шляху для отримання історичних курсів валют через /historical/*.json

date	date	Запитана дата у форматі РРРР-ММ-ДД (обов'язково).

Таблиця 4 – Параметри запиту кінцевої точки /historical/*.json Open Exchange Rates API

app_id	string	Ваш унікальний ідентифікатор програми (обов'язково)
base	string	Змінити основну валюту (3-буквений код, за умовчанням: USD)
symbols	string	Обмежити результати певними валютами (список із 3-літерних кодів, розділених комами)
show_alternative	boolean	Розширте повернуті значення за допомогою альтернативних курсів, курсів чорного ринку та цифрових валют

prettyprint	boolean	Зрозуміла відповідь для налагодження (розмір відповіді буде набагато більшим)

Приклад отримання даних на мові Python

Наведемо приклад реалізації отримання даних з Open Exchange Rates API. Представимо, що для аналізу динаміки змін курсу валют та подальшого їх прогнозування дані потрібно буде отримувати якщо не кожного дня, то з деякою періодичністю.

Тому важливо організувати проєкт таким чином, щоб не переписувати вручну параметри запиту, що будуть постійними. Їх можна виділити у певному файлі з певним форматом зберігання даних, який по суті буде файлом конфігурації. Наприклад, використаємо формат YAML, оскільки він є достатньо зручним для сприйняття.

Файл config.yaml

```
currency_app:
  url: "https://openexchangerates.org/api/latest.json"
  app_id: "******************"
  symbols:
    - 'GBP'
    - 'UAH'
    - 'USD'
    - 'EUR'
  directory: "./data/"
```

Далі мовою Python створимо клас, що оброблює дані конфігураційного файлу, попередньо встановивши модуль pyyaml (pip install pyyaml).

Файл config.py

```
import yaml

class Config:
    def __init__(self, path):
        with open(path, 'r') as yaml_file:
            self.__config = yaml.load(yaml_file, Loader=yaml.SafeLoader)

    def get_config(self, application):
        return self.__config.get(application)
```

Підключення до Web API, отримання даних курсу валют за поточну дату та їх зберігання у файлі формату JSON відбуватиметься в головному файлі проєкту, що має назву app.py.

Файл config.py

```
import requests
import json
import os

from datetime import date

from requests.exceptions import HTTPError

from config import Config


def app(config):
    os.makedirs(os.path.join(config['directory']), exist_ok=True)

    try:
        currencies = ",".join(config['symbols'])

        url = config['url']
        params = {'app_id': config['app_id'], 'symbols': currencies}

        response = requests.get(url, params=params)
        response.raise_for_status()

        with
open(os.path.join(config['directory']+str(date.today())+'.json'),  'w')  as
json_file:
            data = response.json()
            data = data['rates']
            json.dump(data, json_file)

    except HTTPError:
        print('Error!')


if __name__ == '__main__':
    config = Config(os.path.join('.', 'config.yaml'))
    app(config.get_config('currency_app'))
```

У результаті запуску програми ми матимемо у проєкті папку data, куди будуть додаватись файли у форматі JSON з курсами обраних валют, що вказані у конфігураційному файлі формату YAML.



Мал. 9 – Структура проєкту та приклад JSON-файлу з отриманими курсами валют

Завдання

За допомогою бібліотеки Requests мови Python витягніть курс для довільних 5 валют місячної та двомісячної давнини щодо американського долара та порівняйте, наскільки змінювався курс відносно поточного. Відповідні курси повинні бути зафіксовані у відповідних файлах, які використовуватимуться для подальшого аналізу щодо змін курсів валют.

Робота з налаштування ПЗ №1. «Встановлення Apache Airflow, PostgreSQL та Supervisor в операційній системі Linux»

Встановлення Apache Airflow

Apache Airflow – це платформа для автоматизації робочих процесів і оркестрації потоку даних. Вона дозволяє визначати, планувати, виконувати та моніторити серії завдань, які мають виконуватись у певній послідовності. Ось основні теоретичні відомості про Apache Airflow:

Основні компоненти:

1. **DAG (Directed Acyclic Graph)** – це основний елемент в Apache Airflow. Він визначає структуру робочого процесу та порядок виконання завдань.
2. **Task (Завдання)** – окремі етапи чи операції, що виконуються в рамках DAG. Завдання можуть бути різними: виконання коду, завантаження даних, запуск скриптів тощо.
3. **Scheduler (Планувальник)** – відповідальний за запуск задач відповідно до розкладу, визначеного в DAG.
4. **Executor (Виконавець)** – процес, який безпосередньо виконує завдання. Є різні типи виконувачів, такі як LocalExecutor, CeleryExecutor, KubernetesExecutor.
5. **Web UI** – графічний інтерфейс, який дозволяє відслідковувати стан завдань, переглядати логи та управляти DAG.

Особливості та переваги:

1. **Гнучкість:** Airflow дозволяє створювати складні ієрархії завдань із залежностями, що робить його потужним інструментом для складних робочих процесів.

2. **Моніторинг і відстеження:** завдяки вбудованому веб-інтерфейсу можна спостерігати за виконанням завдань, переглядати їх логи та отримувати статистику.

3. **Масштабованість:** Airflow підтримує горизонтальне масштабування через використання різних типів виконувачів, таких як Celery для виконання завдань на декількох машинах.

Інтеграція з іншими інструментами: Airflow має велику кількість операторів для інтеграції з іншими інструментами, такими як бази даних, обчислювальні платформи та зовнішні сервіси.

Щоб встановити Apache Airflow, дотримуйтесь цих кроків:

1. Підготовка середовища

Airflow найкраще встановлювати у віртуальному середовищі Python, щоб уникнути конфліктів з іншими пакетами. Переконайтеся, що Python та pip встановлені.

Перевірте, чи встановлений Python (версія 3.7–3.11):

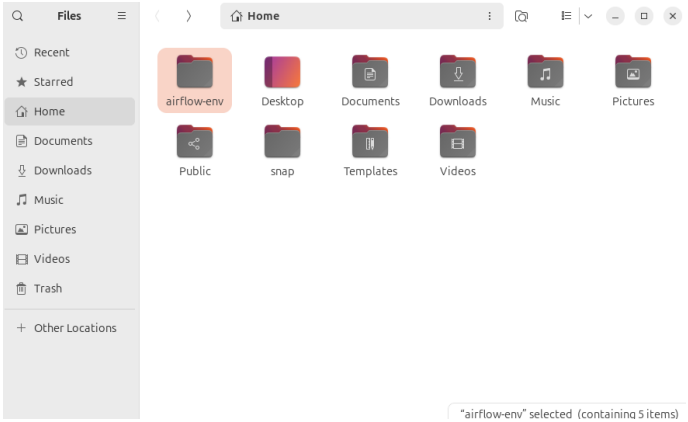
```
python3 --version
```

Встановіть pip (якщо не встановлено):

```
sudo apt install python3-pip
```

Створіть віртуальне середовище:

```
python3 -m venv airflow-env
```

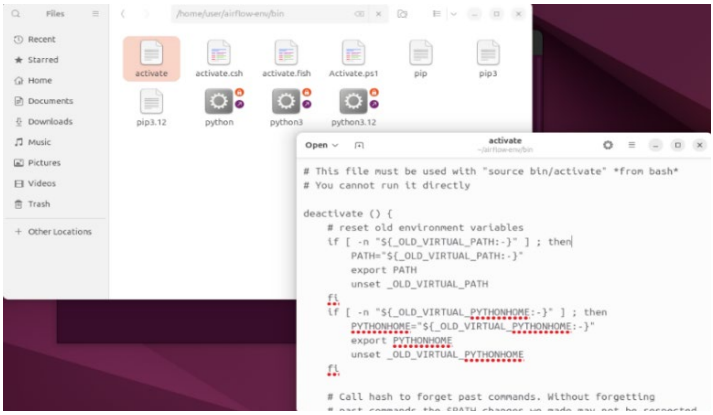


Мал. 10 – Створене віртуальне середовище airflow-env у домашньому каталозі користувача

У результаті створення віртуального середовища у домашній папці користувача буде створено папку airflow-env, що і представляє це середовище. На наступному кроці його буде необхідно активувати, щоб Airflow був встановлений у ньому.

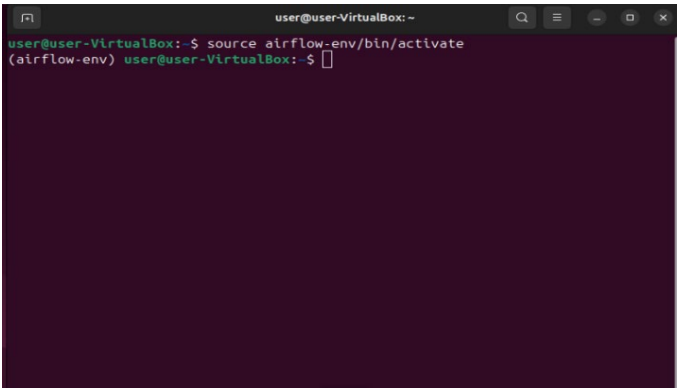
source airflow-env/bin/activate

Файл activate розташований у папці bin віртуального середовища та представляє собою bash-скрипт.



Мал. 11 – Вміст bash-скрипта activate для активації віртуального середовища

Після виконання команди source ми почнемо працювати у віртуальному середовищі airflow-env.



Мал. 12 – Активація віртуального середовища airflow-env у терміналі

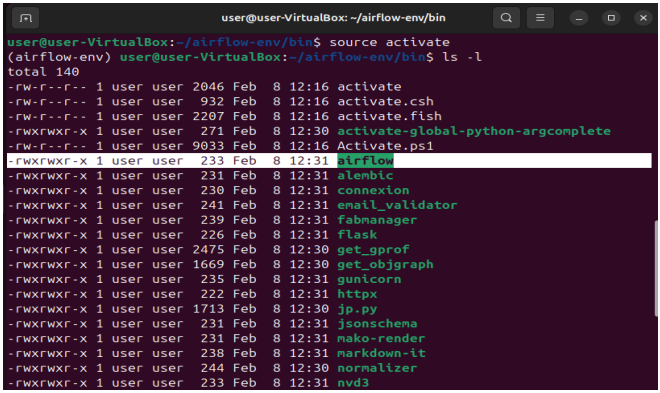
2. Встановлення Airflow

Airflow вимагає додаткових залежностей для бази даних, брокера повідомлень, тощо. Для базового встановлення використовуйте:

Встановіть Airflow через pip:

pip install apache-airflow

Після цього у папці bin віртуального середовища з'являться виконувані файли, що являють собою скрипти на мові Python, головним з яких є airflow, що і запускає сервіс.

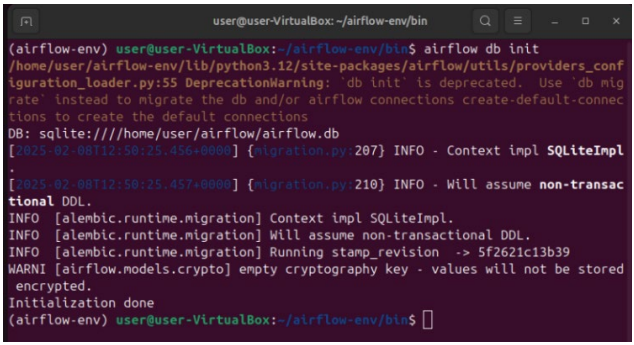


Мал. 13 – Виконувані файли у каталозі bin після встановлення Apache Airflow

3. Ініціалізація бази даних

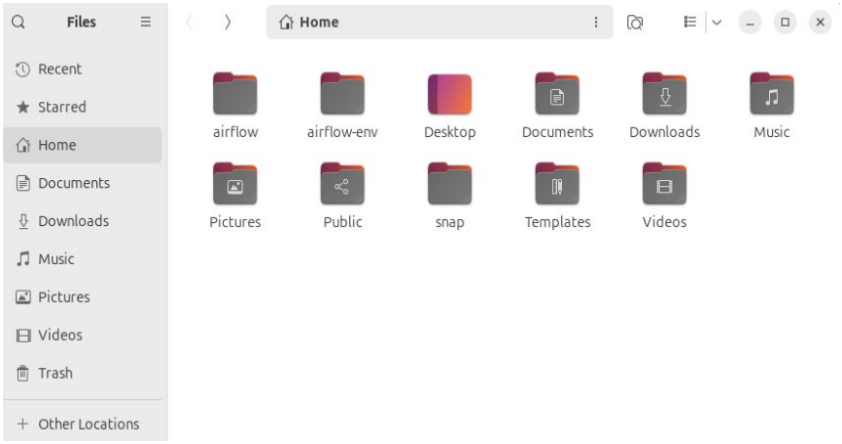
Airflow використовує базу даних для зберігання стану задач. Для першого запуску необхідно ініціалізувати базу даних:

airflow db init



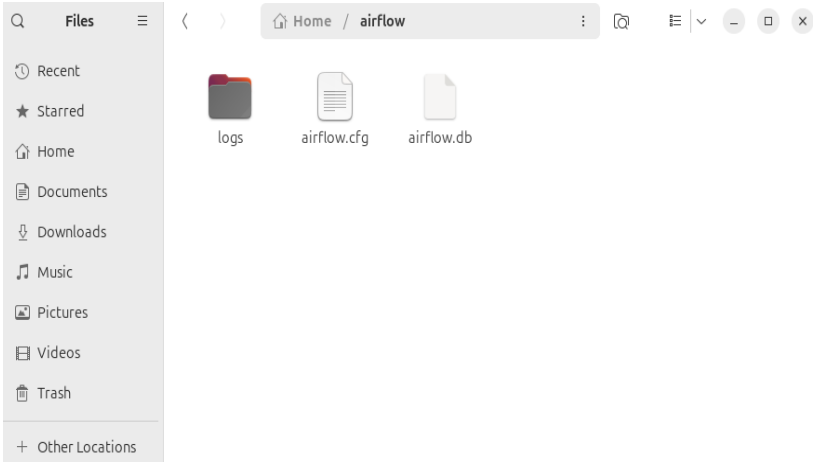
Мал. 14 – Ініціалізація бази даних Apache Airflow

При ініціалізації бази даних для збереження даних налаштувань Airflow у домашньому каталозі користувача буде створено папку airflow, що і буде робочою папкою для цього сервісу.



Мал. 15 – Створення робочого каталогу airflow у домашній папці користувача

У папці airflow зокрема є файл airflow.db, що представляє собою базу даних SQLite, та конфігураційний файл airflow.cfg, у якому збережені налаштування сервісу.



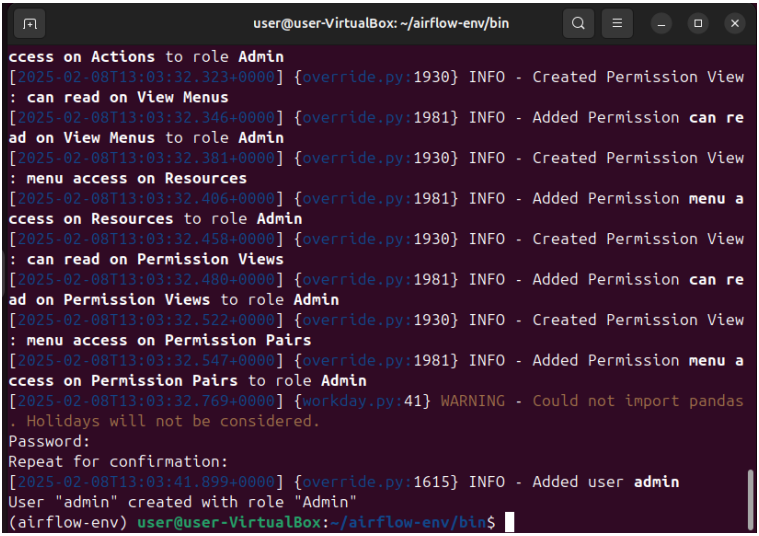
Мал. 16 – Вміст робочого каталогу airflow з файлами airflow.cfg та airflow.db

4. Створення користувача

Для доступу до веб-інтерфейсу створіть адміністративного користувача:

```
airflow users create \
    --username admin \
    --firstname Admin \
    --lastname User \
    --role Admin \
    --email admin@example.com
```

При створенні користувача необхідно ввести всю інформацію, зазначену відповідними параметрами, інакше користувач не буде створений, а система виведе відповідну підказку щодо необхідності введення повної інформації. При правильному введенні всіх даних нового користувача буде створено, а перед цим система запитає пароль для нового користувача.

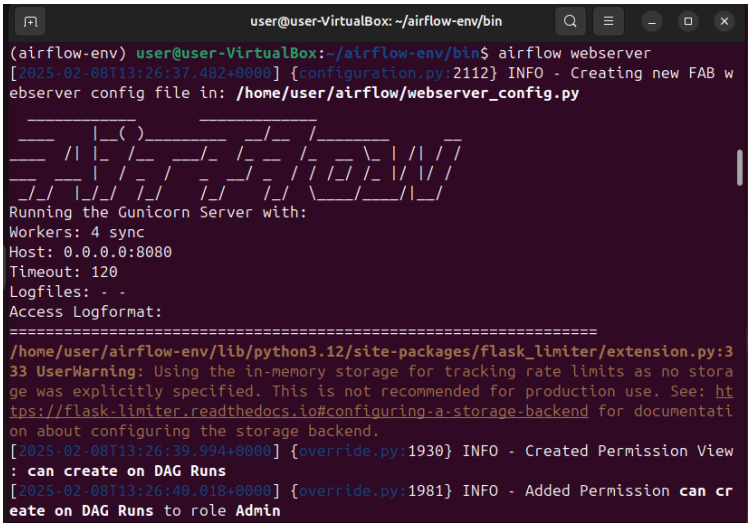


Мал. 17 – Створення адміністративного користувача Apache Airflow

5. Запуск Airflow

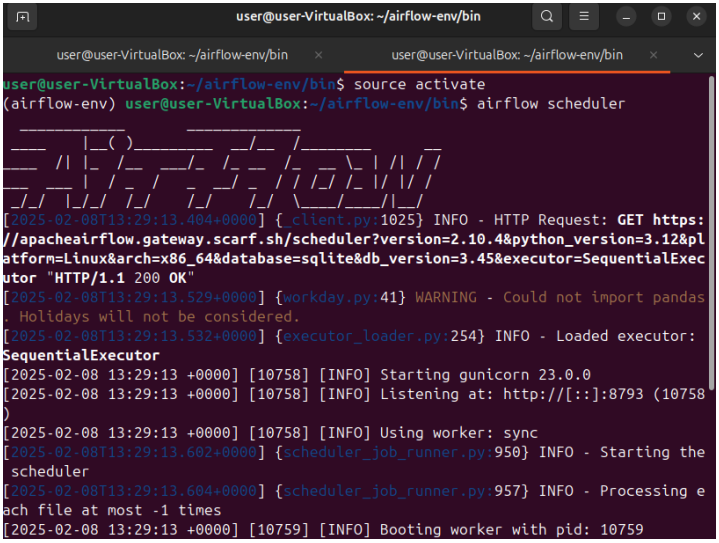
Запустіть веб-сервер:

```
airflow webserver
```



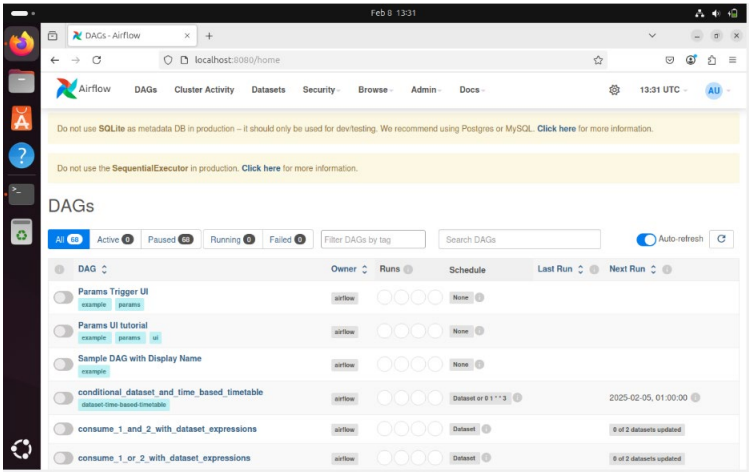
Мал. 18 – Запуск вебсервера Apache Airflow у терміналі

У іншій консолі запустіть планувальник:
airflow scheduler



Мал. 19 – Запуск планувальника Apache Airflow у терміналі

За замовчуванням веб-інтерфейс доступний за адресою: http://localhost:8080.



Мал. 20 – Головна сторінка вебінтерфейсу Apache Airflow

Якщо у Linux встановлено openssh-server, то вебінтерфейс Airflow буде доступний віддалено за ip-адресою віртуальної машини.

6. (Опціонально) Налаштування конфігурації

Файл конфігурації Airflow розташований у директорії ~/.airflow/airflow.cfg. Ви можете змінити налаштування, наприклад:

– порт веб-сервера;
– URL бази даних;
– додаткові провайдери.

Встановлення PostgreSQL

Встановлення PostgreSQL у Linux залежить від дистрибутива. Для встановлення у дистрибутивах, похідних від Debian, необхідно виконати наступні дії:

```
sudo apt update
sudo apt install postgresql postgresql-contrib
```

Після встановлення служба PostgreSQL запускається автоматично. Перевірити статус можна так:

```
sudo systemctl status postgresql
```

Або запустити вручну:

```
sudo systemctl start postgresql
```

Створення та налаштування бази даних для Airflow

1. Створіть базу даних та користувача для Airflow. Виконайте наступні команди (можна через консоль PostgreSQL):

sudo -u postgres psql #Увійти у консоль PostgreSQL під іменем користувача postgres

Усередині psql виконайте:

```
CREATE DATABASE airflow;
CREATE USER airflow WITH PASSWORD 'your_password';
ALTER DATABASE airflow OWNER TO airflow;
\q
```

Замініть 'your_password' на бажаний пароль. Назви бази даних та користувача можна змінити, але в прикладах нижче використовуються airflow.

2.Встановіть необхідний Python-пакет для PostgreSQL у віртуальному середовищі airflow-env.

Airflow підключається до PostgreSQL через драйвер psycopg2. Рекомендується встановити його як psycopg2-binary:

pip install psycopg2-binary

3. Налаштуйте з'єднання у конфігурації Airflow

Знайдіть файл airflow.cfg.

У секції [core] знайдіть параметр sql_alchemy_conn та змініть його на:

sql_alchemy_conn = postgresql+psycopg2://airflow:your_password@localhost/airflow

Тут:

– airflow – ім'я користувача,

– your_password – пароль, який ви встановили,

– localhost – адреса сервера бази даних (якщо PostgreSQL встановлено на локальній машині),

– airflow – назва бази даних.

4. Ініціалізуйте метадані Airflow

Після того, як налаштування змінено, ініціалізуйте (або перезавантажте) базу даних Airflow:

airflow db init

Встановлення supervisor

Supervisor – це система управління процесами, розроблена для Unix-подібних операційних систем, яка дозволяє централізовано контролювати, запускати, моніторити та перезапускати фонові служби та додатки. Основне призначення Supervisor полягає у забезпеченні стабільної та безперервної роботи сервісів, особливо тих, що повинні працювати постійно як фонові процеси (демони).

supervisord – це основний демон, який запускається у системі і відповідає за управління іншими процесами. Він читає конфігураційні файли, ініціалізує визначені процеси та слідкує за їхнім станом.

supervisorctl – це клієнтська утиліта, яка дозволяє адміністраторам взаємодіяти з supervisord через командний рядок, виконувати команди для запуску, зупинки, перезапуску процесів та отримувати інформацію про їхній стан.

Основні можливості та призначення:

1. **Автоматичний запуск та моніторинг процесів:** Supervisor дозволяє налаштувати автоматичний старт сервісів під час завантаження системи, а також автоматично перезапускати їх у разі збою або аварійного завершення.
2. **Централізоване управління:** Використовуючи один демон для контролю за багатьма процесами, Supervisor забезпечує просте адміністрування та моніторинг всіх служб, що працюють на сервері.
3. **Легка конфігурація:** Налаштування здійснюється за допомогою конфігураційних файлів у форматі .ini, де можна визначити команди для запуску, параметри логування, користувача, від імені якого має виконуватися процес, та інші параметри.
4. **Гнучкість та розширюваність:** Supervisor може використовуватися для управління різними типами процесів – від веб-серверів і баз даних до сервісів обробки даних, що робить його універсальним інструментом для системних адміністраторів та розробників.
5. **Інтеграція через API:** Supervisor надає XML-RPC API, що дозволяє інтегрувати його функціональність у інші додатки чи скрипти.

Практичне застосування:

1. Використовується для забезпечення високої доступності сервісів за рахунок автоматичного відновлення у разі збоїв.
2. Полегшує розгортання комплексних систем, де багато служб повинні працювати одночасно та взаємодіяти між собою.
3. Застосовується в середовищах розробки та продакшн, де необхідно контролювати роботу серверних додатків, баз даних, обчислювальних кластерів та інших сервісів.

4. Таким чином, Supervisor є потужним інструментом для адміністрування та моніторингу фонових процесів, що забезпечує стабільність роботи системи за рахунок автоматизації управління службами та їх перезапуску у разі необхідності.

Для встановлення supervisor у дистрибутивах, похідних від Debian, необхідно виконати команду:

sudo apt install supervisor

Налаштування автоматичного запуску сервісів Airflow у Supervisor

Головний конфігураційний файл знаходиться зазвичай у /etc/supervisor/supervisord.conf. Проте кожен процес можна додавати в окремі конфігураційні файли в директорії /etc/supervisor/conf.d/.

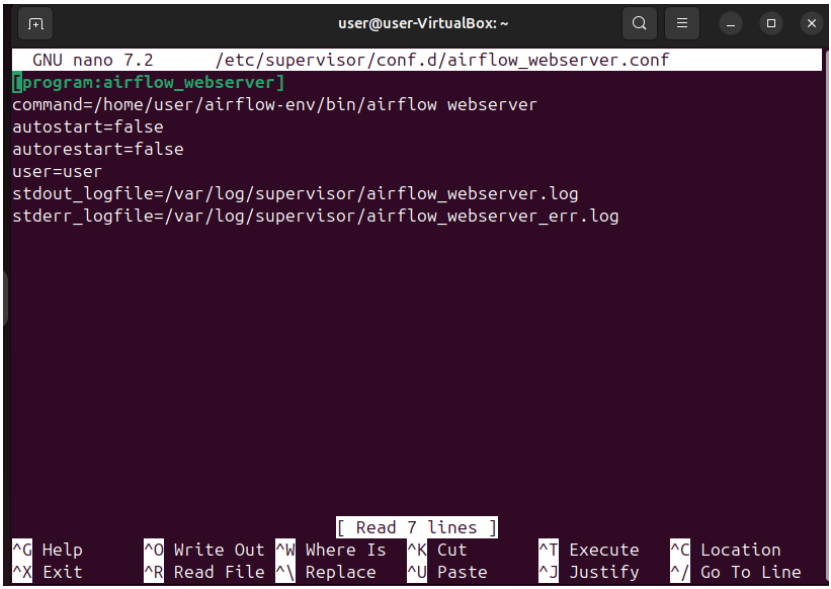
Для налаштування запуску окремого сервісу потрібно створити відповідний файл конфігурації та помісити його у папку /etc/supervisor/conf.d/. Слід пам'ятати, що для цього необхідна команда sudo, оскільки звичайний користувач немає права на запис у цій папці. Наприклад, для створення конфігураційного файлу для вебсервера Airflow за допомогою редактора nano необхідно виконати команду:

sudo nano airflow_webserver.conf

Вміст конфігураційного файлу буде наступним:

Файл airflow_webserver.conf

```
[program:airflow_webserver]
command=/home/user/airflow-env/bin/airflow webserver
autostart=false
autorestart=false
user=user
stdout_logfile=/var/log/supervisor/airflow_webserver.log
stderr_logfile=/var/log/supervisor/airflow_webserver_err.log
```



Мал. 21 – Створення конфігураційного файлу airflow_webserver.conf для Supervisor

Ім’я користувача може відрізнятись, для автоматичного запуску та перезапуску потрібно встановити autostart та autorestart у true, у тім краще запускати відповідний сервіс вручну за потребою.

Подібний конфігураційний файл створимо і для планувальника Airflow:

Файл airflow_scheduler.conf

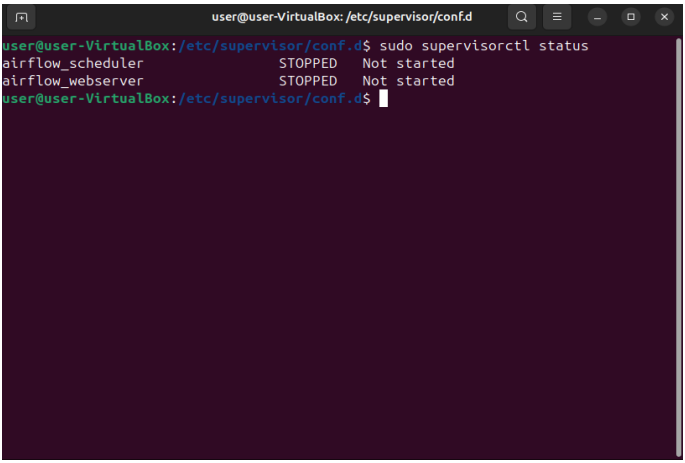
```
[program:airflow_scheduler]
command=/home/user/airflow-env/bin/airflow scheduler
autostart=false
autorestart=false
user=user
stdout_logfile=/var/log/supervisor/airflow_scheduler.log
stderr_logfile=/var/log/supervisor/airflow_scheduler_err.log
```

Після додавання всіх конфігурацій потрібно перевірити їх:

```
sudo supervisorctl reread
sudo supervisorctl update
```

Потім потрібно перевірити стан процесів:

```
sudo supervisorctl status
```

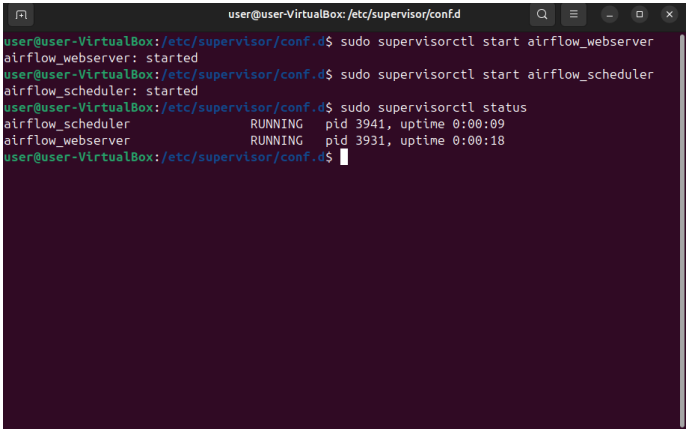


Мал. 22 – Перевірка стану процесів Airflow за допомогою supervisorctl status

Щоб запустити конкретний процес, необхідно виконати команду:
sudo supervisorctl start <process_name>

А щоб зупинити відповідний процес, необхідно виконати команду:
sudo supervisorctl stop <process_name>

Запустимо сервіси Airflow за допомогою supervisor, а потім перевіримо їх стан:



Мал. 23 – Запуск сервісів Airflow за допомогою Supervisor та перевірка їхнього стану

У подальшому будемо використовувати supervisor для налуштування інших необхідних у подальшій роботі сервісів: Hadoop, Spark та Greenplum.

Завдання

Маючи віртуальну машину Linux або локальну систему (будь-який дистрибутив) встановити Apache Airflow, PostgreSQL з налаштуванням бази даних для зберігання метаданих для нього та Supervisor з налаштуванням запуску сервісів Airflow.

Практична робота №3. «Автоматизація ELT-процесів у Apache Airflow: робота з PostgreSQL та API»

Робота з базою даних PostgreSQL мовою Python з використанням бібліотеки psycopg2

Psycopg2 – це одна з найпопулярніших бібліотек для роботи з PostgreSQL у Python. Вона дозволяє взаємодіяти з базою даних, виконувати SQL-запити, працювати з транзакціями та обробляти результати запитів.

Щоб встановити psycopg2, слід скористатись командою:

```
pip install psycopg2
```

Або встановити версію з компіляцією C:

```
pip install psycopg2-binary
```

Підключення до PostgreSQL виконується за допомогою модуля connect:

```
import psycopg2

try:
    connection = psycopg2.connect(
        dbname="your_database",
        user="your_user",
        password="your_password",
        host="localhost",
        port="5432"
    )
    print("Підключення встановлено")
except Exception as e:
    print("Помилка підключення:", e)
```

Виконання SQL-запитів:

```
cursor = connection.cursor()
cursor.execute("SELECT * FROM your_table")
```

```
rows = cursor.fetchall()
for row in rows:
    print(row)
```

Виконання INSERT:

```
cursor.execute("INSERT INTO your_table (column1, column2) VALUES (%s, %s)", (value1, value2))
connection.commit()
```

Закриття підключення:

```
cursor.close()
connection.close()
print("Підключення закрито")
```

Приклад роботи з БД pagila

Файл readFromDB.py

```
import os;
import psycopg2;

pg_creds = {
    'host': 'localhost',
    'port': '5432',
    'database': 'pagila',
    'user': 'pguser',
    'password': 'pguser'
}

def read_films_from_db():
    with psycopg2.connect(**pg_creds) as pg_connection:
        cursor = pg_connection.cursor()
        os.makedirs(os.path.join('.','data'),exist_ok=True)
        with open(file=os.path.join('.','data','film'),mode='w') as csv_file:
            cursor.copy_expert('COPY film TO STDOUT WITH HEADER CSV', csv_file)

read_films_from_db()
```

Цей Python-скрипт readFromDB.py підключається до бази pagila, експортує всю таблицю film у CSV-файл data/film та забезпечує створення необхідної директорії для збереження файлу.

Для підключення до бази даних у файлі визначаються облікові дані у вигляді словника pg_creds, у якому зазначаються хост, порт, база даних, користувач PostgreSQL та його пароль.

Слід зазначити, що для роботи з базою даних було створено користувача pguser з однойменним паролем та надано йому привілеї для роботи з БД pagila. Для цього в інтерфейсі командного рядка для взаємодії з базою даних PostgreSQL psql необхідно створити користувача pguser та надати йому пароль:

```
sudo -u postgres psql
CREATE USER pguser WITH PASSWORD 'pguser';
```

Для надання користувачеві pguser всіх привілеїв на рівні бази даних pagila у psql необхідно виконати команду:

```
GRANT ALL PRIVILEGES ON DATABASE pagila TO pguser;
```

Ця команда дозволить користувачеві створювати об'єкти (таблиці, функції) та виконувати інші операції в межах бази. В тім, ця команда не дає доступ до вже існуючих таблиць. Щоб користувач міг працювати з таблицями, потрібно надати окремі права на схеми та таблиці.

```
GRANT USAGE ON SCHEMA public TO pguser; # дозволяє користувачу працювати зі схемою public
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO pguser; # дозволяє SELECT, INSERT, UPDATE, DELETE, TRUNCATE для всіх таблиць
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO pguser; # дозволяє змінювати автоінкрементні значення (SERIAL)
GRANT ALL PRIVILEGES ON ALL FUNCTIONS IN SCHEMA public TO pguser; # дозволяє викликати всі функції
```

DAG у Apache Airflow

Apache Airflow – це платформа для створення, планування та моніторингу робочих процесів у вигляді спрямованих ациклічних графів (Directed Acyclic Graph, DAG). Airflow дозволяє автоматизувати виконання завдань, які залежать одне від одного, забезпечуючи гнучке керування та моніторинг процесів.

DAG (спрямований ациклічний граф) у Airflow – це структура, яка визначає порядок виконання завдань (tasks) і їх залежності. Основні особливості DAG:

– спрямованість – завдання виконуються в заданому порядку;

– ациклічність – немає циклічних залежностей, що запобігає нескінченному виконанню;

– гнучкість – можна задавати різні типи залежностей між завданнями.

Основні компоненти DAG

1. DAG

Це основна структура, яка описує робочий процес у Airflow. Визначається у вигляді Python-коду та містить:

– назву DAG;

– графік запуску (schedule interval);

– дефолтні параметри;

– список завдань та їх залежності.

2. Tasks (Завдання)

Усередині DAG завдання визначають конкретні дії. Завдання можуть бути:

– Operators (Оператори) – стандартні компоненти для виконання завдань (наприклад, BashOperator, PythonOperator, EmailOperator).

– Sensors (Сенсори) – спеціальні оператори, що очікують настання певної події (наприклад, перевірка наявності файлу).

– TaskFlow API – використовується для декларативного створення DAG.

3. Operators (Оператори)

Airflow надає різні оператори для виконання завдань, серед яких:

– BashOperator – виконує Bash-команди.

– PythonOperator – виконує Python-функції.

– DummyOperator – використовується як маркер або для розгалужень.

– EmailOperator – надсилає email-повідомлення.

4. Dependencies (Залежності між завданнями)

Завдання у DAG можна з'єднувати між собою, створюючи залежності:

```
task1 >> task2  # task2 виконується після task1
task1 << task2  # task1 виконується після task2
task1.set_downstream(task2)  # еквівалент task1 >> task2
task1.set_upstream(task2)  # еквівалент task1 << task2
```

Налаштування DAG

Створення DAG відбувається через Python-скрипт. Приклад базового DAG:

```
from airflow import DAG
from airflow.operators.bash import BashOperator
```

```
from datetime import datetime, timedelta

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2024, 2, 10),
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG(
    'example_dag',
    default_args=default_args,
    schedule_interval='@daily',
    catchup=False
)

task1 = BashOperator(
    task_id='print_hello',
    bash_command='echo "Hello Airflow!"',
    dag=dag
)

task2 = BashOperator(
    task_id='print_goodbye',
    bash_command='echo "Goodbye Airflow!"',
    dag=dag
)

task1 >> task2  # Визначення порядку виконання
```

Планування та виконання DAG

Airflow дозволяє задавати періодичність запуску DAG за допомогою schedule_interval. Деякі популярні значення:

– '@once' – виконати DAG лише один раз.
– '@hourly' – виконувати щогодини.
– '@daily' – виконувати щодня.
– '@weekly' – виконувати щотижня.
– '@monthly' – виконувати щомісяця.

Також можна використовувати вирази cron.

Запуск DAG у Airflow

Для додавання DAG необхідно у робочому каталозі Airflow створити підкаталог під назвою dags та помістити у нього файл Python зі спроєктованим DAG.

Для перегляду та керування DAG можна використовувати веб-інтерфейс Airflow або командний рядок:

– airflow dags list # список усіх DAG

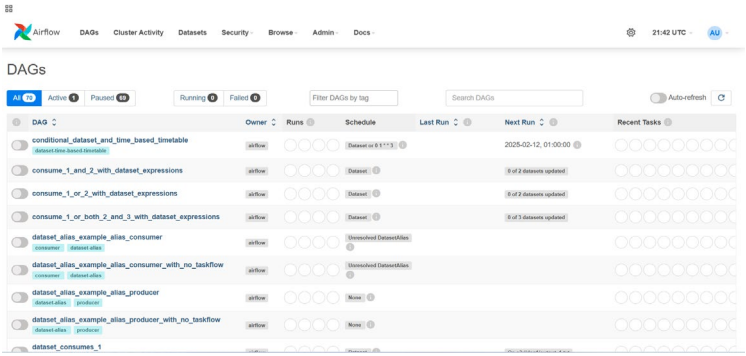
– airflow dags trigger example_dag # запуск DAG

– airflow tasks list example_dag # список завдань у DAG

airflow tasks test example_dag print_hello 2024-02-10 # тестове виконання завдання

Apache Airflow Web UI – це основний інструмент для управління DAG, завданнями та моніторингом робочих процесів. Він дозволяє запускати, призупиняти, переглядати виконання DAG, виявляти помилки та аналізувати логи.

Головна сторінка веб-інтерфейсу відображає список DAG, доступних у системі.



Мал. 24 – Список DAG на головній сторінці вебінтерфейсу Apache Airflow

Таблиця 5 – Основні елементи головної сторінки вебінтерфейсу Apache Airflow

Елемент	Опис
DAG ID	Унікальний ідентифікатор DAG (назва)
Owner	Власник DAG (зазначений у default_args)
Runs	Кількість останніх запусків DAG
Schedule	Частота запуску (наприклад, @daily, */5 * * * *)
Next Run	Час наступного запуску DAG

Last Run	Час останнього запуску DAG
Actions	Доступні дії: перегляд графіка, журналів, ручний запуск, пауза
Status	Активність DAG (On або Off)

Основні кнопки:
– Trigger DAG – ручний запуск DAG;
– Toggle DAG – увімкнення/вимкнення DAG;
– Refresh – оновлення стану DAG;
– Delete DAG – видалення DAG (видаляє з бази, але не з файлів).

Перегляд DAG

1. Graph View (Графічне представлення) - Відображає структуру DAG у вигляді спрямованого ациклічного графа.

Кожен вузол – це завдання. Колір вузлів відображає стан (running, success, failed).

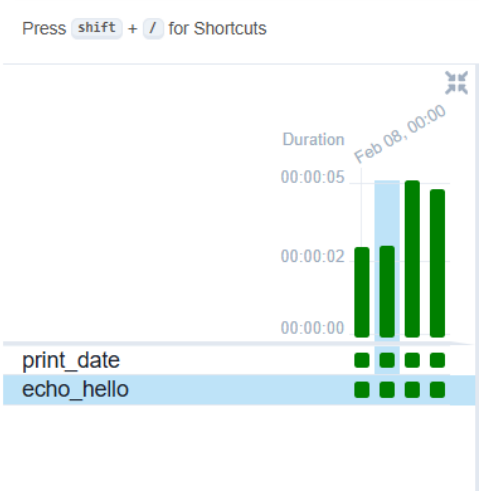


Мал. 25 – Графічне представлення структури DAG у режимі Graph View

Дії в Graph View:
– клік на вузол відкриває деталі завдання;
– натискання “Task Instance Details” – перегляд логів та деталей виконання.

2. Tree View (Дерево виконання) – відображає стан кожного DAG-запуску в табличному вигляді.

Вертикальні колонки – запуски DAG, горизонтальні рядки – завдання DAG.



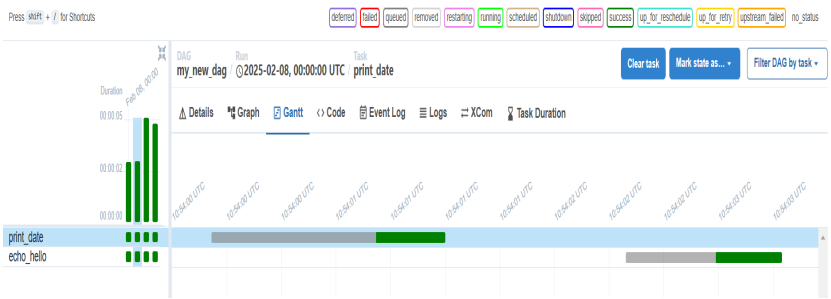
Мал. 26 – Табличне представлення запусків і завдань DAG у режимі Tree View

Клік на завдання відкриває його лог.

3. Gantt Chart (Діаграма Ганта) – візуалізує тривалість виконання завдань у DAG.

Допомагає знайти "вузькі місця" у DAG.

Часові інтервали показують початок та завершення завдань.



Мал. 27 – Відображення тривалості виконання завдань у режимі Gantt Chart

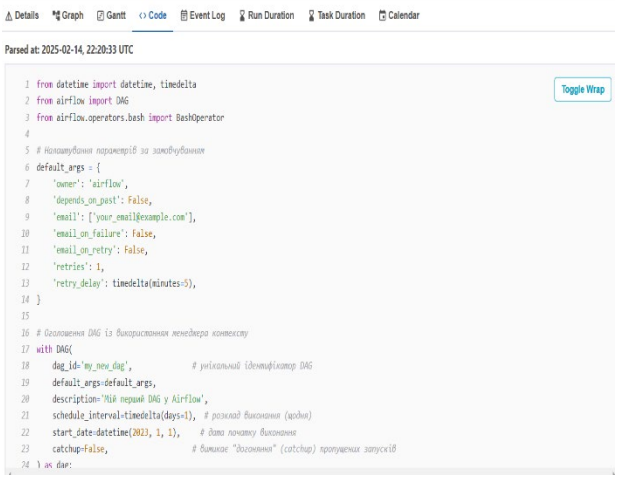
4. Task Duration (Тривалість виконання) – відображає графік тривалості виконання завдань.

Корисно для аналізу продуктивності.



Мал. 28 – Графік тривалості виконання завдань у режимі Task Duration

5. Code (Код DAG) – показує вихідний код DAG.
Дозволяє швидко перевірити логіку.



Мал. 29 – Перегляд програмного коду DAG у вебінтерфейсі Airflow

Додавання підключення PostgreSQL у Airflow

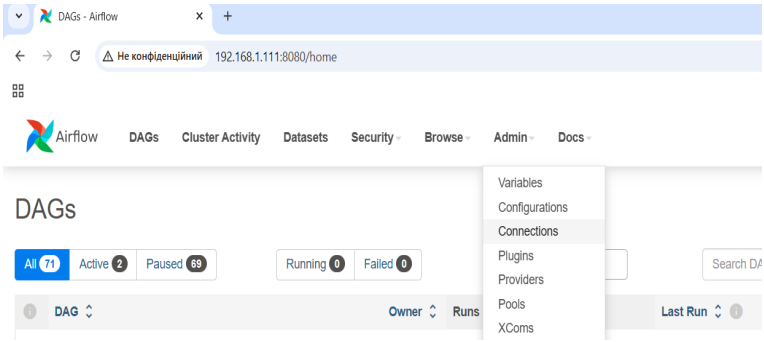
У Airflow є веб-інтерфейс, де можна переглядати виконання DAG, відстежувати помилки та змінювати статуси завдань.

Для того щоб PostgresHook міг підключатися до бази даних PostgreSQL, потрібно налаштувати PostgreSQL Connection (postgres_conn_id) у Airflow.

Додавання PostgreSQL підключення через Airflow UI

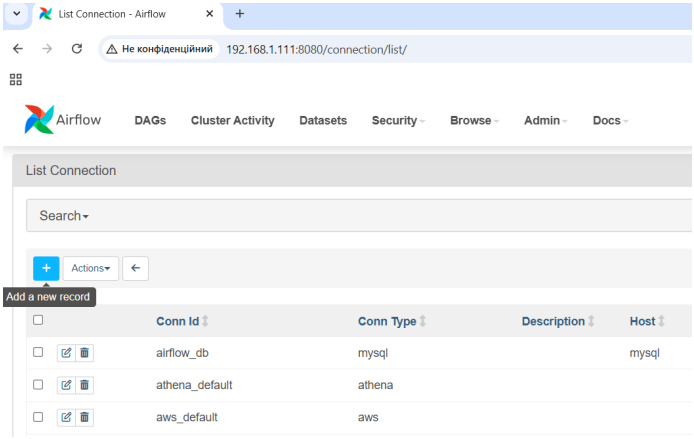
Кроки:

1. **Відкрити веб-інтерфейс Airflow.** Перейдіть за адресою: http://localhost:8080 (або відповідну для вашого середовища).
2. **Перейти до "Admin" → "Connections".** У верхньому меню натисніть Admin → Connections.



Мал. 30 – Перехід до розділу Connections у вебінтерфейсі Apache Airflow

3. **Натиснути "Create" (або редагувати існуюче підключення postgres_default).** Створіть нове підключення або редагуйте postgres_default.



Мал. 31 – Створення нового підключення PostgreSQL у вебінтерфейсі Airflow

4. Заповнити поля підключення:

– **Connection Id**: postgres_default (або інше ім'я, якщо бажаєте, але тоді потрібно вказати його в DAG).

– **Connection Type**: Postgres;

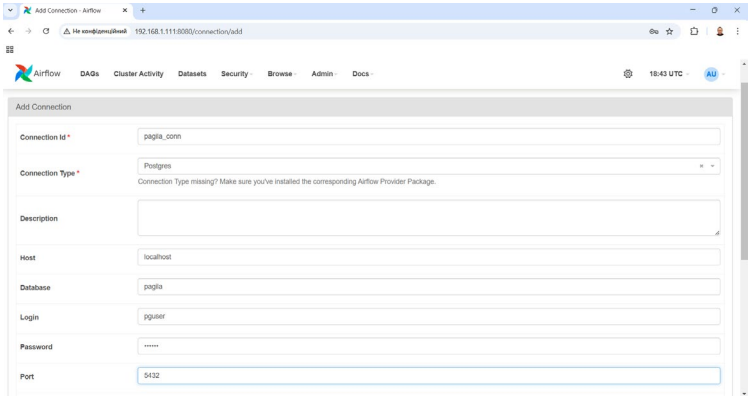
– **Host**: localhost (або IP-адреса сервера з PostgreSQL);

– **Database**: pagila (або інша назва бази даних);

– **Login**: pguser (або ваш користувач PostgreSQL);

– **Password**: pguser (пароль від PostgreSQL);

– **Port**: 5432 (типовий порт PostgreSQL).



Мал. 32 – Заповнення параметрів підключення PostgreSQL в Apache Airflow

5. Натиснути "Save".

Додавання через командний рядок (CLI)

Якщо Airflow встановлено локально, можна створити підключення через CLI:

```
airflow connections add postgres_default \
    --conn-type postgres \
    --conn-host localhost \
    --conn-schema airflow_db \
    --conn-login airflow \
    --conn-password airflow_password \
    --conn-port 5432
```

Використання PostgresHook у DAG

Після додавання підключення можна використовувати його у DAG:

```
from airflow.providers.postgres.hooks.postgres import PostgresHook
```

```
def test_connection():
    pg_hook = PostgresHook(postgres_conn_id="postgres_default")
    conn = pg_hook.get_conn()
    cursor = conn.cursor()
    cursor.execute("SELECT version();")
    result = cursor.fetchone()
    print(f"PostgreSQL Version: {result[0]}")
    cursor.close()
    conn.close()
```

Приклад DAG із завданнями по роботі з PostgreSQL

```
import os
from airflow.hooks.postgres_hook import PostgresHook
from airflow.operators.python_operator import PythonOperator
from airflow.operators.dummy_operator import DummyOperator
from airflow import DAG
from datetime import datetime

def copy_data(table):
    pg_hook = PostgresHook(postgres_conn_id='pagila_conn')

    connection = pg_hook.get_conn()
    cursor = connection.cursor()

    os.makedirs('/home/user/pg_data',exist_ok=True)
    with open(file='//home/user/pg_data/' + table + '.csv', mode='w') as csv_file:
        cursor.copy_expert('COPY' + table + 'TO STDOUT WITH
HEADER CSV', csv_file)

default_args = {
    'owner': 'airflow',
    'email': ['airflow@airflow.com'],
    'email_on_failure': False,
    'retries': 2
    }

dag = DAG(
    'pagila_dag',
    description='The dag for the pagila database in PostgreSQL',
    schedule_interval='@once',
    start_date=datetime(2025,2,14,15,0),
```

```
    default_args=default_args
)

tables = ['film','category','actor','customer']
copy_tasks = []

for table in tables:
    copy_tasks.append(PythonOperator(
        task_id = 'copy_' + table + '_task',
        python_callable = copy_data,
        dag = dag,
        op_kwargs={'table': table}
    ))

dummy1 = DummyOperator(task_id="dummy1",dag=dag)
dummy2 = DummyOperator(task_id="dummy2",dag=dag)

dummy1 >> copy_tasks >> dummy2
```

Цей код є DAG (Directed Acyclic Graph) для Apache Airflow, який використовується для автоматизації ETL-процесів у базі даних PostgreSQL.

У функції copy_data об’єкт pg_hook створює підключення до PostgreSQL через збережене з'єднання pagila_conn.

Метод makedirs модуля os створює каталог pg_data у домашньому каталозі користувача user, якщо його ще немає.

Метод copy_expert об’єкта cursor виконує команду COPY, щоб експортувати вміст таблиці PostgreSQL у CSV-файл.

Далі створюється DAG з назвою pagila_dag. Значення властивості schedule_interval='@once' визначає, що DAG виконується лише один раз, а start_date=datetime(2025,2,14,15,0) – що DAG починає працювати з 14 лютого 2025 року о 15:00.

Після створення DAG створюються завдання для копіювання таблиць. Список таблиць tables містить film, category, actor, customer. Для кожної таблиці створюється окреме завдання PythonOperator, яке викликає copy_data(table).

op_kwargs={'table': table} передає назву таблиці у функції copy_data у якості параметру.

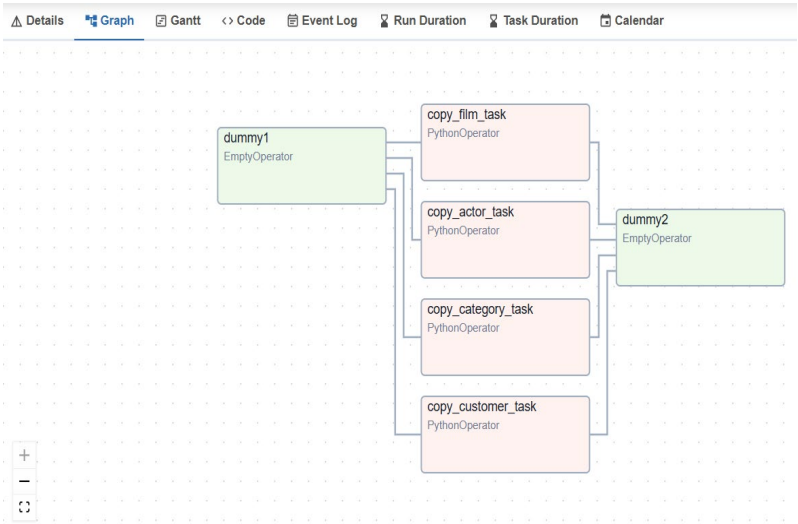
У структурі DAG dummy1 і dummy2 – це пусті оператори, що позначають його початок і кінець.

dummy1 >> copy_tasks >> dummy2 – задає послідовність виконання:

1. Починається із dummy1.

2. Далі виконуються всі задачі copy_tasks (експорт таблиць).
3. Завершується dummy2.

В результаті отримується такий граф



Мал. 33 – Граф DAG для експорту таблиць із бази даних PostgreSQL

Можна також скопіювати всі таблиці з бази даних, знайшовши їх назви. Для цього слід написати метод, що і повертає список цих таблиць.

```
def get_tables():
    pg_hook = PostgresHook(postgres_conn_id='pagila_conn')

    connection = pg_hook.get_conn()
    cursor = connection.cursor()

    cursor.execute("""
        SELECT table_name
        FROM information_schema.tables
        WHERE table_schema = 'public'
    """)
    tables = cursor.fetchall()

    return [table[0] for table in tables]
```

Завдання

Використовуючи Apache Airflow PythonOperator, реалізуйте наступний ELT процес:

– організувати щоденне отримання даних курсів валют на поточну дату за допомогою Apache Airflow у вигляді json-файлів;

– вивантажити всі дані з бази даних dshop у PostgreSQL (файл dshop_dump.sql) на диск за допомогою Apache Airflow.

Робота з налаштування ПЗ №2. «Встановлення та розгортання Hadoop в операційній системі Linux»

Hadoop – це платформа програмного забезпечення з відкритим кодом, яка використовується для зберігання та обробки великих обсягів даних у розподіленому обчислювальному середовищі. Він розроблений для обробки великих даних і базується на моделі програмування MapReduce, яка дозволяє паралельно обробляти великі набори даних.

Hadoop складається з двох основних компонентів:

– HDFS (розподілена файлова система Hadoop) – це компонент зберігання даних Hadoop, який дозволяє зберігати великі обсяги даних на кількох машинах. Він призначений для роботи з товарним обладнанням, що робить його економічно вигідним.

– YARN – це компонент керування ресурсами Hadoop, який керує розподілом ресурсів (таких як процесор і пам'ять) для обробки даних, що зберігаються в HDFS.

Ось детальні кроки для встановлення Hadoop в операційній системі Linux:

1. Підготовка середовища

Перед тим як встановлювати Hadoop, потрібно переконатись, що у вас є:

– **Java**: Hadoop вимагає Java для роботи. Зазвичай рекомендується використовувати Java 8.

– **SSH**: Для конфігурації кластеру Hadoop зазвичай використовують SSH для безпарольного доступу між вузлами.

2. Встановлення Java

1. Перевірте, чи встановлено Java:

```
java -version
```

2. Якщо Java не встановлено, ви можете встановити її за допомогою:

```
sudo apt update
sudo apt install openjdk-8-jdk
```

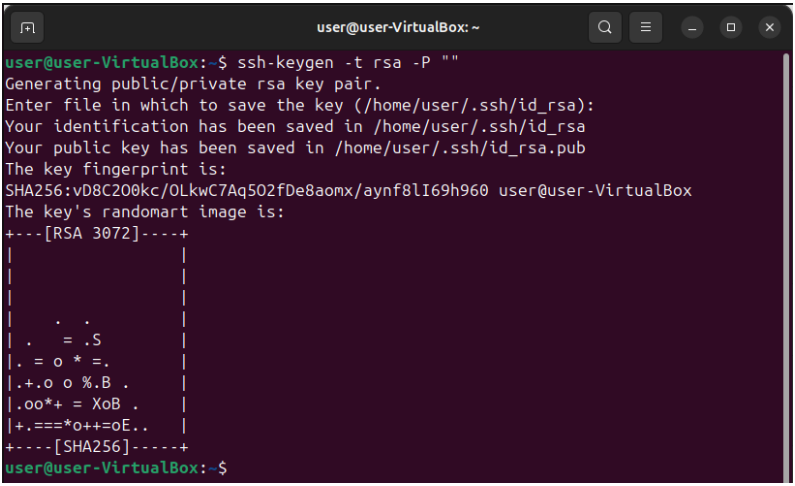
3. Налаштуйте змінні середовища для Java:
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export PATH=$PATH:$JAVA_HOME/bin

4. Додайте ці рядки в файл ~/.bashrc для автоматичного застосування змін після перезапуску:
echo "export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64" >> ~/.bashrc
echo "export PATH=$PATH:$JAVA_HOME/bin" >> ~/.bashrc
source ~/.bashrc

3. Встановлення SSH (якщо ще не встановлений)
1. Встановіть SSH:
sudo apt install openssh-client openssh-server

2. Переконайтесь, що сервіс SSH працює:
sudo systemctl enable ssh
sudo systemctl start ssh

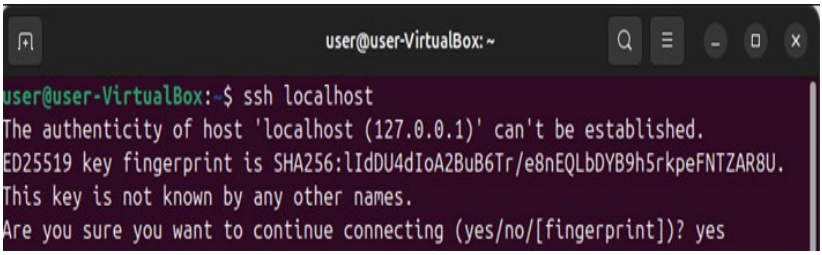
3. Налаштуйте безпарольний доступ SSH:
- Створіть ключі SSH:
  ssh-keygen -t rsa -P ""



Мал. 34 – Створення SSH-ключа для безпарольного доступу під час налаштування Hadoop

○ Додайте публічний ключ до authorized_keys:
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

4. Перевірте, чи можна підключитись без пароля:
ssh localhost



Мал. 35 – Перевірка безпарольного підключення до localhost через SSH

4. Завантаження та встановлення Hadoop

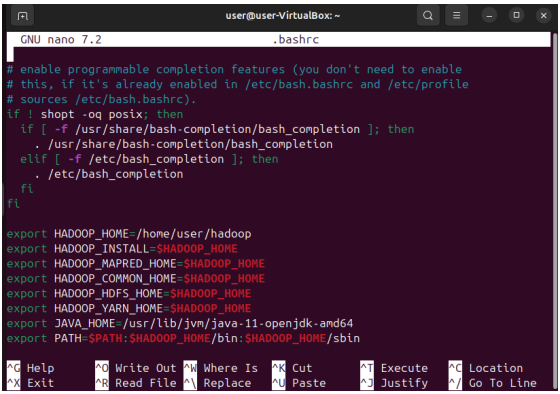
1. Перейдіть на сайт Hadoop та завантажте потрібну версію.
2. Наприклад, для завантаження останньої стабільної версії Hadoop 3.3.6:
wget https://downloads.apache.org/hadoop/common/hadoop-3.3.6/hadoop-3.3.6.tar.gz

3. Розпакуйте архів:
tar -xzf hadoop-3.3.6.tar.gz

4. Перейменуйте розпаковану директорію у hadoop:
mv hadoop-3.3.6 /home/user/hadoop

5. Налаштування змінних середовища для Hadoop

1. Відредагуйте файл ~/.bashrc і додайте наступні рядки:
export HADOOP_HOME=/home/user/hadoop
export HADOOP_INSTALL=$HADOOP_HOME
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export HADOOP_YARN_HOME=$HADOOP_HOME
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin



Мал. 36 – Налаштування змінних середовища Hadoop у файлі .bashrc

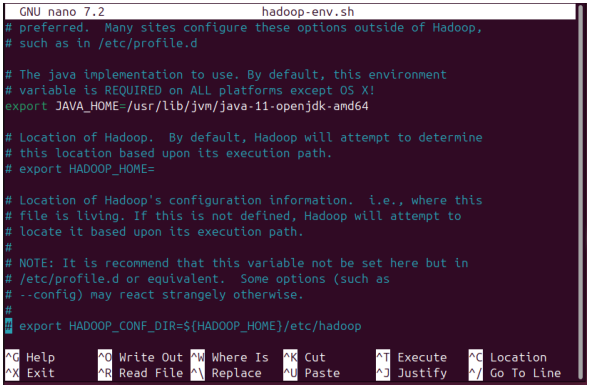
2. Завантажте зміни:
source ~/.bashrc

6. Налаштування конфігураційних файлів

1. Перейдіть в директорію Hadoop:
cd /home/user/hadoop/etc/hadoop (або ж ~/hadoop/etc/hadoop)

2. Відредагуйте файл hadoop-env.sh:
nano hadoop-env.sh

Додайте або оновіть рядок:
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64



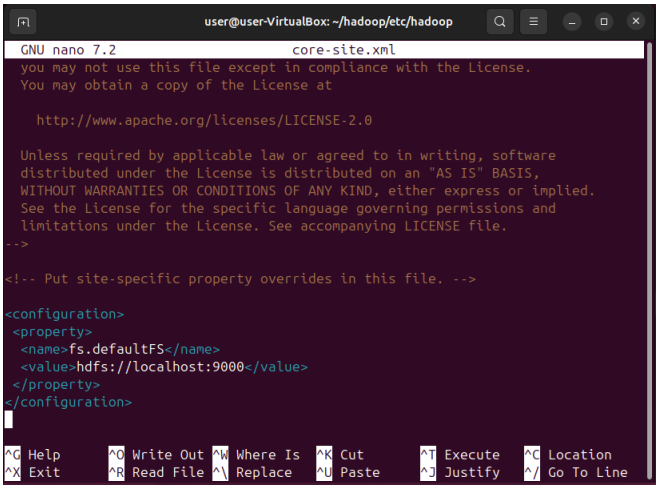
Мал. 37 – Налаштування змінної JAVA_HOME для роботи Hadoop

3. Відредагуйте файл core-site.xml:
nano core-site.xml

Додайте наступні налаштування:

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

Цей параметр вказує на HDFS, яка працює на порту 9000.



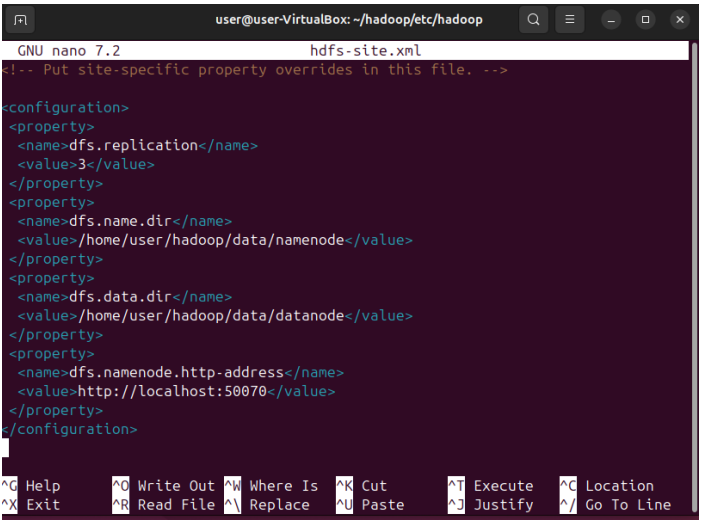
Мал. 38 – Редагування конфігураційного файлу core-site.xml

4. Відредагуйте файл hdfs-site.xml:
nano hdfs-site.xml

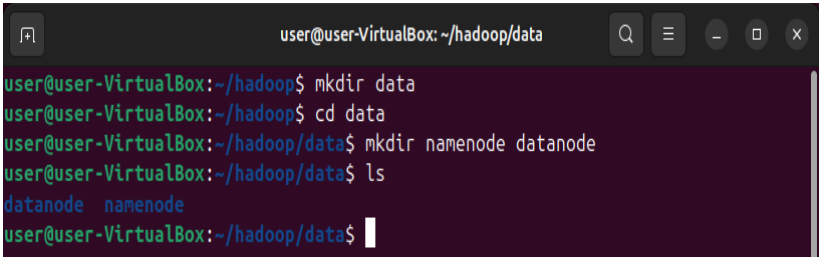
Додайте наступні налаштування:

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>3</value>
  </property>
  <property>
    <name>dfs.name.dir</name>
```

```
  <value>/home/user/hadoop/data/namenode</value>
 </property>
 <property>
  <name>dfs.data.dir</name>
  <value>/home/user/hadoop/data/datanode</value>
 </property>
 <property>
  <name>dfs.namenode.http-address</name>
  <value>http://localhost:50070</value>
 </property>
</configuration>
```



Мал. 39 – Редагування конфігураційного файлу hdfs-site.xml

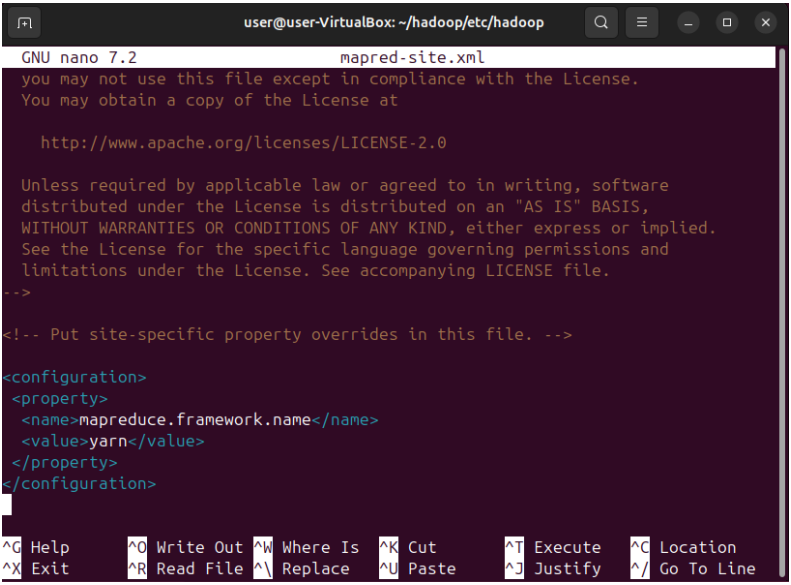


Мал. 40 – Налаштування параметрів NameNode та DataNode у файлі hdfs-site.xml

Цей файл визначає, де зберігаються дані NameNode і DataNode, а також налаштовує доступ до веб-інтерфейсу NameNode.

5. Відредагуйте файл mapred-site.xml:
nano mapred-site.xml

Додайте:
```
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>
```



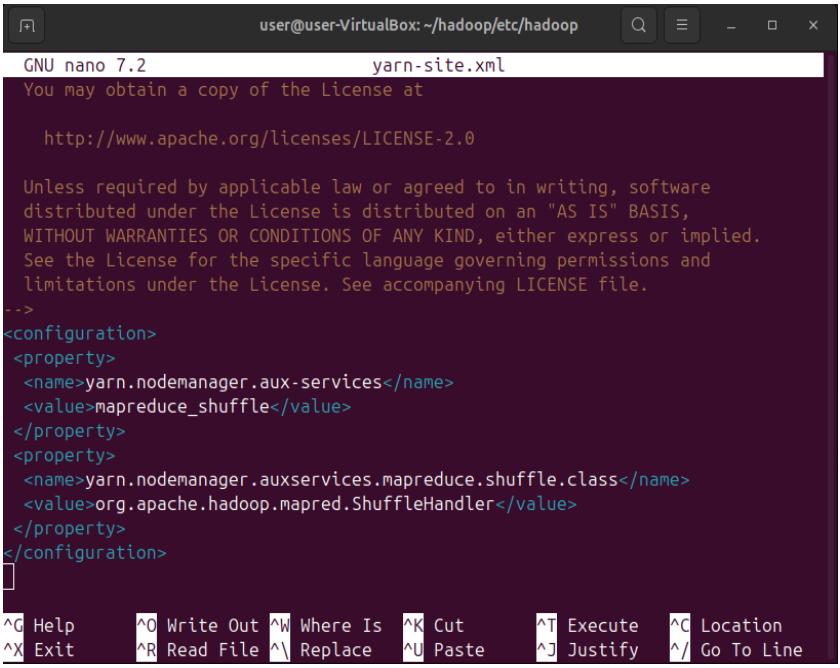
Мал. 41 – Редагування конфігураційного файлу mapred-site.xml

6. Відредагуйте файл yarn-site.xml:
nano yarn-site.xml

Додайте:
```
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
```

```
        <value>mapreduce_shuffle</value>
      </property>
      <property>

<name>yarn.nodemanager.auxservices.mapreduce.shuffle.class</name>
        <value>org.apache.hadoop.mapred.ShuffleHandler</value>
      </property>
    </configuration>
```



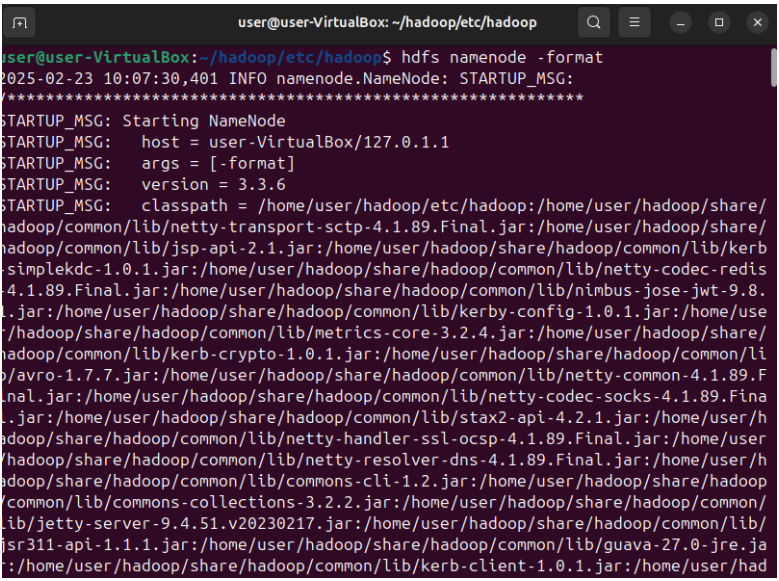
Мал. 42 – Редагування конфігураційного файлу yarn-site.xml

7. Форматування HDFS

Після того як ви налаштували конфігураційні файли, виконайте форматування NameNode:

hdfs namenode -format

Це створить нову мета-структуру для файлової системи.



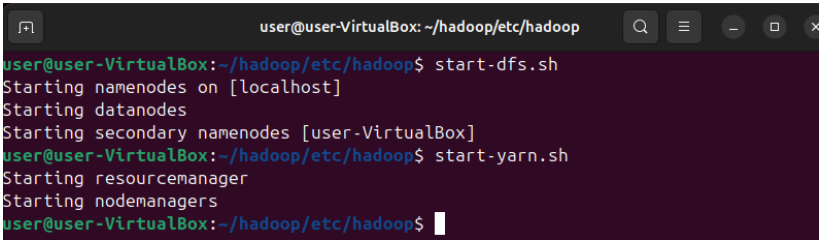
Мал. 43 – Форматування NameNode у HDFS

8. Запуск Hadoop

1. Запустіть Hadoop:

start-dfs.sh

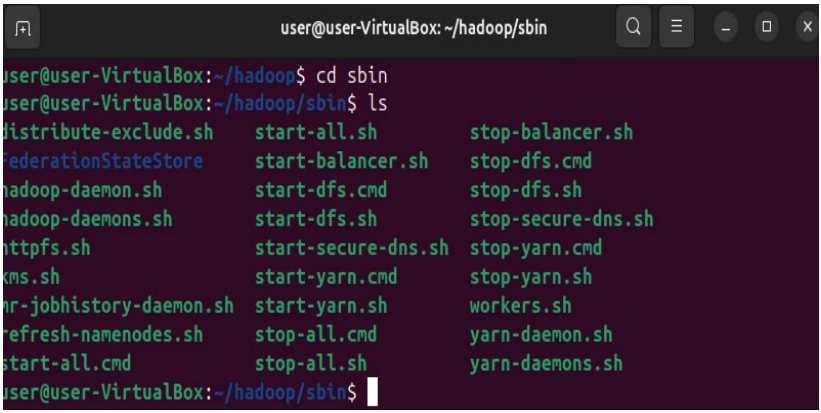
start-yarn.sh



Мал. 44 – Запуск сервісів Hadoop за допомогою скриптів start-dfs.sh та start-yarn.sh

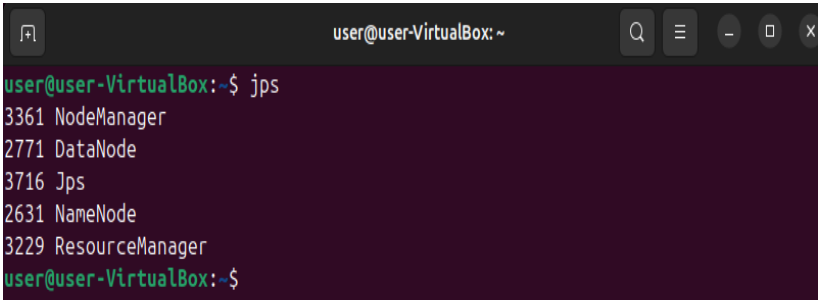
Замість виконання двох зазначених скриптів по черзі можна запустити всі сервіси Hadoop разом, запустивши скрипт start-all.sh.

Всі скрипти знаходяться у папці sbin, що розташована у папці, де встановлено Hadoop.



Мал. 45 – Вміст каталогу sbin зі скриптами запуску сервісів Hadoop

2. Перевірте, чи працюють сервіси, виконавши команду:
jps



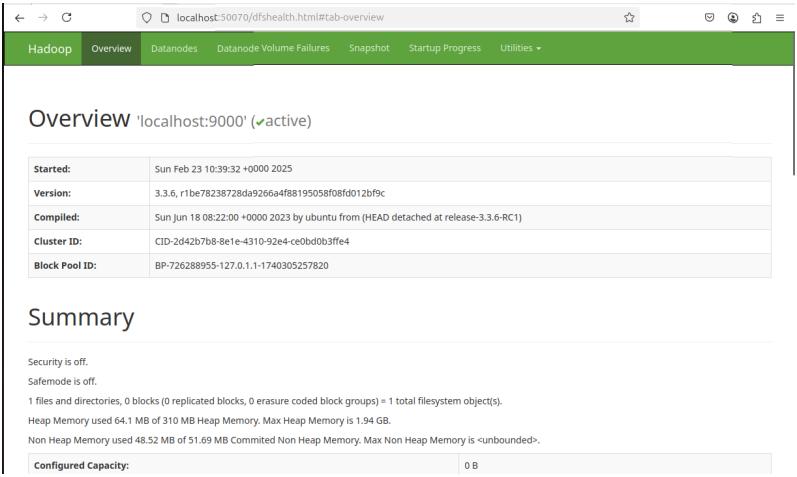
Мал. 46 – Перевірка запущених процесів Hadoop за допомогою команди jps

Ви повинні побачити процеси NameNode, DataNode, ResourceManager, NodeManager в списку.

9. Доступ до Hadoop через веб-інтерфейс

Після запуску сервісів ви можете отримати доступ до веб-інтерфейсу NameNode за адресою:

http://localhost:50070



Мал. 47 – Вебінтерфейс NameNode для перегляду стану кластера Hadoop

Це дозволить вам переглядати стан кластеру, управління файлами і метаданими.

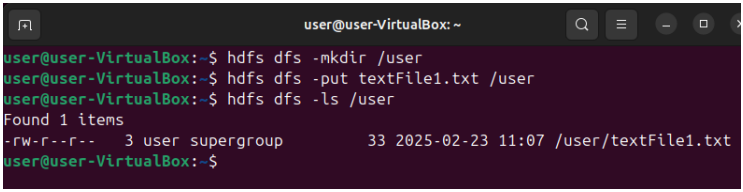
9. Тестування HDFS

Для перевірки, чи правильно працює HDFS, спробуйте виконати наступні команди:

1. Створіть директорію в HDFS:
hdfs dfs -mkdir /user

2. Завантажте файл у HDFS:
hdfs dfs -put /path/to/local/file /user

3. Перевірте вміст HDFS:
hdfs dfs -ls /



Мал. 48 – Створення каталогу, завантаження файлу та перегляд вмісту HDFS

Налаштування автоматизованого запуску Hadoop за допомогою Supervisor

Щоб налаштувати автоматизований запуск Hadoop за допомогою Supervisor (інструмент для управління процесами в Linux), вам потрібно створити конфігураційні файли для сервісів Hadoop. Нижче наведено покрокову інструкцію.

Для того, щоб не створювати конфігураційний файл для кожного сервісу Hadoop окремо, можна створити один конфігураційний файл, у якому вказати групу сервірів, що будуть запускатись.

Файл hadoop.conf

```
[group:hadoop]
programs=namenode,datanode,resourcemanager,nodemanager,secondar
ynamenode

[program:namenode]
command=/home/user/hadoop/bin/hdfs--config/home/user/hadoop/etc/hadoop/
namenode
user=user
autostart=false
autorestart=false
stdout_logfile=/var/log/hadoop_namenode.log
stderr_logfile=/var/log/supervisor/hadoop_namenode_err.log

[program:datanode]
command=/home/user/hadoop/bin/hdfs--config/home/user/hadoop/etc/hadoop/
datanode
user=user
autostart=false
autorestart=false
stdout_logfile=/var/log/hadoop_datanode.log
stderr_logfile=/var/log/supervisor/hadoop_datanode_err.log

[program:resourcemanager]
command=/home/user/hadoop/bin/yarn--config/home/user/hadoop/etc/hadoop/
resourcemanager
user=user
autostart=false
autorestart=false
stdout_logfile=/var/log/hadoop_resourcemanager.log
stderr_logfile=/var/log/supervisor/hadoop_resourcemanager_err.log
```

```
[program:nodemanager]
command=/home/user/hadoop/bin/yarn--config/home/user/hadoop/etc/hadoop/
nodemanager
user=user
autostart=false
autorestart=false
stdout_logfile=/var/log/supervisor/hadoop_nodemanager.log
stderr_logfile=/var/log/supervisor/hadoop_nodemanager_err.log

[program:secondarynamenode]
command=/home/user/hadoop/bin/hdfs--config/home/user/hadoop/etc/hadoop/
secondarynamenode
user=user
autostart=false
autorestart=false
stdout_logfile=/var/log/supervisor/hadoop_secondarynamenode.log
stderr_logfile=/var/log/supervisor/hadoop_secondarynamenode_err.log
```

Так само можна налаштувати і запуск сервісів Airflow, вже налаштовані у попередніх роботах.

Файл airflow.conf

```
[group:airflow]
programs=webserver,scheduler

[program:webserver]
command=/home/user/airflow-env/bin/airflow webserver
autostart=false
autorestart=false
user=user
stdout_logfile=/var/log/supervisor/airflow_webserver.log
stderr_logfile=/var/log/supervisor/airflow_webserver_err.log

[program:scheduler]
command=/home/user/airflow-env/bin/airflow scheduler
autostart=false
autorestart=false
user=user
stdout_logfile=/var/log/supervisor/airflow_scheduler.log
stderr_logfile=/var/log/supervisor/airflow_scheduler_err.log
```

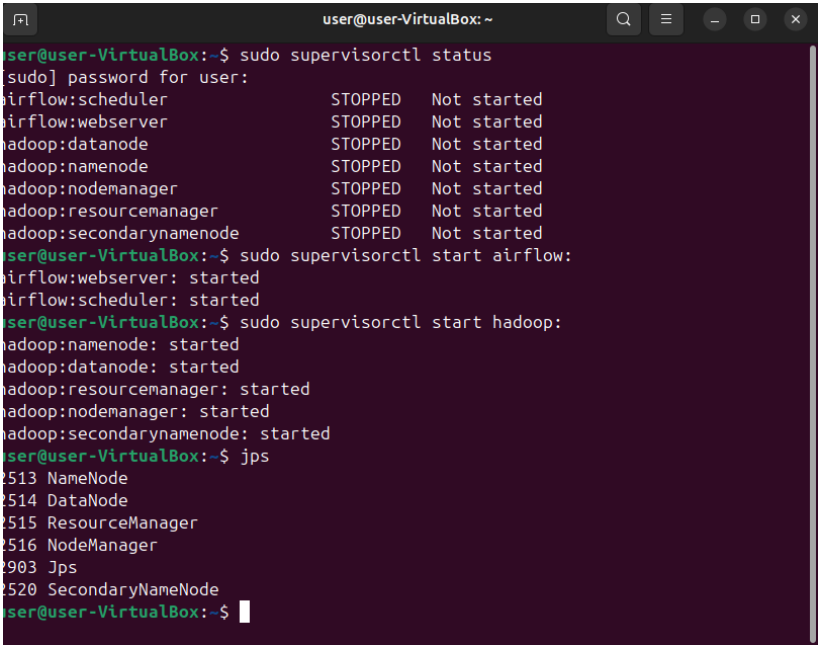
Після того як конфігураційні файли створені, потрібно перезавантажити Supervisor і запустити сервіси Hadoop.

Перезавантажте Supervisor:

sudo supervisorctl reread

sudo supervisorctl update

Перевірте статус і запустіть сервіси:



Мал. 49 – Перевірка стану та запуск сервісів Hadoop через Supervisor

Для запуску всієї групи сервісів необхідно задати назву цієї групи та двокрапку після:

sudo supervisorctl start airflow:

sudo supervisorctl start hadoop:

Завдання

Маючи віртуальну машину Linux або локальну систему (будь-який дистрибутив) встановити та розгорнути Hadoop та налаштувати автоматизований запуск його сервісів через Supervisor.

Практична робота №4. «Робота з розподіленою файловою системою HDFS»

Модуль hdfscli дозволяє взаємодіяти з файловою системою HDFS (Hadoop Distributed File System) за допомогою Python. Цей модуль забезпечує простий інтерфейс для виконання операцій, таких як читання, запис, копіювання файлів і перегляд вмісту директорій.

Для початку потрібно встановити бібліотеку:

```
pip install hdfs
```

Імпорт модуля та налаштування з'єднання

Для того, щоб працювати з HDFS, потрібно створити екземпляр клієнта, вказавши адресу вашого HDFS-сервера (за замовчуванням це буде локальний сервер).

```
from hdfs import InsecureClient

client = InsecureClient('http://localhost:50070', user='user')
```

Основні операції з файловою системою HDFS

1. Перевірка існування файлу або директорії

Для перевірки, чи існує файл чи директорія в HDFS, використовується метод status.

```
client.status('/path/to/file_or_directory')
```

2. Завантаження файлу в HDFS

Щоб завантажити файл з локальної файлової системи в HDFS, використовується метод upload.

```
client.upload('/path/to/hdfs/directory', '/path/to/local/file')
```

3. Завантаження файлу з HDFS на локальну файлову систему

Для завантаження файлу з HDFS на локальну систему використовується метод download.

```
client.download('/path/to/hdfs/file', '/path/to/local/directory')
```

4. Читання вмісту файлу в HDFS

Щоб прочитати файл з HDFS, використовується метод open, який повертає об'єкт типу io.BufferedReader.

```
with client.read('/path/to/file') as reader:
    content = reader.read()
    print(content)
```

5. Створення директорії в HDFS

Для створення нової директорії в HDFS використовується метод makedirs.

client.makedirs('/path/to/new/directory')

6. Видалення файлу або директорії

Щоб видалити файл чи директорію, можна використати метод delete.

client.delete('/path/to/file_or_directory', recursive=True)

7. Перевірка розміру файлу або директорії

Метод status також повертає розмір файлу або директорії в байтах.

status = client.status('/path/to/file_or_directory')

print(status['size'])

8. Перелік вмісту директорії

Метод list дозволяє отримати список файлів і директорій вказаної директорії HDFS.

files = client.list('/path/to/directory')

print(files)

9. Переміщення або перейменування файлів

Для переміщення або перейменування файлу використовується метод rename.

client.rename('/path/to/old/file', '/path/to/new/file')

Представлений нижче програмний код на мові Python виконує наступні дії:

- створення підключення до сервера HDFS через клієнт InsecureClient;
- створення директорії /test у файловій системі HDFS;
- перегляд вмісту кореневої директорії HDFS;
- запис даних у файл в форматі JSON на HDFS;
- завантаження файлу з HDFS на локальний диск;
- завантаження файлу з локальної файлової системи на HDFS.

```
import json
import os

from hdfs import InsecureClient

def main():
```

```
    client = InsecureClient(f'http://127.0.0.1:50070/', user='user')

    # create directory in HDFS
    client.makedirs('/test')

    #list content
    ll = client.list('/')
    print(ll)

    # create file in HDFS
    data = [{"name": "Anne", "salary": 10000}, {"name": "Victor",
"salary": 9500}]
    with client.write('/test/sample_file.json', encoding='utf-8') as
json_file_in_hdfs:
        json.dump(data, json_file_in_hdfs)
    # OR
    client.write(os.path.join('/', 'test', 'sample_file2.json'), data=json.dumps(data),
encoding='utf-8')

    # download file from HDFS
    client.download('/test/sample_file.json', './file_from_hadoop.json')

    # upload file to HDFS
    client.upload('/test/local_file_in_hadoop.json', './file_from_hadoop.json')

if __name__ == '__main__':
    main()
```

client – створення об'єкта InsecureClient, що дозволяє підключитись до HDFS-сервера за вказаною URL-адресою (http://127.0.0.1:50070/) та використовувати вказаного користувача (user).

http://127.0.0.1:50070/ – URL для доступу до веб-інтерфейсу Hadoop NameNode (HDFS-сервера). В даному випадку сервер працює локально.

user='user' – вказується ім'я користувача для доступу до HDFS.

makedirs('/test') – створює нову директорію /test в HDFS. Якщо така директорія вже існує, вона не буде створена знову.

client.list('/') – отримує список вмісту кореневої директорії HDFS (/). Результат зберігається в змінній ll та виводиться за допомогою print(ll).

client.write('/test/sample_file.json', encoding='utf-8') – відкриває файл sample_file.json в директорії /test для запису. Використовується кодування UTF-8.

json.dump(data, json_file_in_hdfs) – серіалізує список Python в формат JSON та записує його у файл в HDFS.

client.download('/test/sample_file.json', './file_from_hadoop.json') – завантажує файл sample_file.json з HDFS в поточну директорію локальної файлової системи під ім'ям file_from_hadoop.json.

client.upload('/test/local_file_in_hadoop.json', './file_from_hadoop.json') – завантажує файл з локальної файлової системи (файл file_from_hadoop.json) до HDFS у директорію /test під ім'ям local_file_in_hadoop.json.

Наступний код у якості прикладу демонструє:

– роботу з HDFS за допомогою бібліотеки hdfscli: створення директорій, запис і завантаження файлів, перейменування та видалення файлів;

– взаємодію з PostgreSQL через бібліотеку psycopg2: підключення до бази даних і експорт даних у формат CSV прямо в HDFS;

– основні операції з базою даних PostgreSQL та файловою системою HDFS за допомогою Python, що дозволяє обробляти великі обсяги даних з використанням цих технологій.

```
import json
import os

import psycopg2
from hdfs import InsecureClient # library docs https://hdfscli.readthedocs.io/en/latest/index.html

pg_creds = {
    'host': localhost'
    , 'port': '5432'
    , 'database': 'postgres'
    , 'user': 'pguser'
    , 'password': 'secret'
}

def main():

    client = InsecureClient(f'http://127.0.0.1:50070/', user='user')

    # create directory in HDFS
    client.makedirs('/test')
```

```
        #list content
        ll = client.list('/')
        print(f"list of files: {ll}")

        # create file in HDFS
        data = [{"name": "Anne", "salary": 10000}, {"name": "Victor",
"salary": 9500}]
        with client.write('/test/sample_file.json', encoding='utf-8') as
json_file_in_hdfs:
            json.dump(data, json_file_in_hdfs)
        # OR
        client.write(os.path.join('/', 'test', 'sample_file2.json'), data=json.dumps
(data), encoding='utf-8')

        # download file from HDFS
        client.download('/test/sample_file.json', './file_from_hadoop.json')

        # upload file to HDFS
        client.upload('/test/local_file_in_hadoop1.json','./file_from_hadoop.json')
        client.upload('/test/local_file_in_hadoop2.json','./file_from_hadoop.json')
        client.upload('/test/local_file_in_hadoop3.json','./file_from_hadoop.json')

        # rename file
        client.rename('/test/local_file_in_hadoop2.json',
'/test/local_file_in_hadoop22222.json')

        # delete file from HDFS
        client.delete('/test/local_file_in_hadoop3.json', recursive=False)

        # copy table's data from PostgreSQL to HDFS
        with psycopg2.connect(**pg_creds) as pg_connection:
            cursor = pg_connection.cursor()
            with client.write('/test/dump.csv',) as csv_file:
                cursor.copy_expert('COPY users(id, name) TO STDOUT WITH
HEADER CSV', csv_file)


    if __name__ == '__main__':
        main()
```

client.rename(...)–перейменовує файл у HDFS з /test/local_file_in_hadoop2 .json на /test/local_file_in_hadoop22222.json.

client.delete(...) – видаляє файл з HDFS (опція recursive=False вказує, що видаляти потрібно лише файл, а не вміст директорії).

cursor.copy_expert(...) – використовує SQL команду COPY, щоб експортувати дані з таблиці users до CSV-файлу. Данні копіюються безпосередньо в файл, що створюється в HDFS.

Завдання

Напишіть Airflow DAG, який вивантажує дані з PostgreSQL бази даних у файлову систему HDFS в CSV форматі.

Також переробіть всі Airflow DAG з попередньої практичної роботи так, щоб замість диска дані зберігалися у файловій системі HDFS.

Airflow DAG має виконуватися щодня, організуйте зберігання файлів відповідно. Зберігання даних у HDFS також повинно мати організацію.

Практична робота №5. «Аналіз даних CSV-файлу за допомогою Hadoop та Python»

MapReduce – це модель паралельного програмування, яка дозволяє обробляти та генерувати великі обсяги даних шляхом розподілу роботи на незалежні завдання та паралельного виконання їх у кластері машин.

Програма MapReduce складається з двох основних етапів: *Map (відображення)* та *Reduce (зведення)*. На етапі Map вхідні дані обробляються шматками (зазвичай по рядку) і перетворюються у проміжні пари ключ-значення. На етапі Reduce ці проміжні дані агрегуються за ключами для отримання кінцевого результату.

Hadoop надає спеціальну утиліту **Streaming**, що дозволяє запускати користувацькі скрипти (наприклад, на Python) в ролі маппера та редуктора у задачах MapReduce. Таким чином, Python-скрипти можуть бути інтегровані в Hadoop і виконувати розподілену обробку даних.

Альтернативно можна використати бібліотеку **mrjob** – це Python-фреймворк від Yelp, який обгортає Hadoop Streaming та дозволяє писати багатокрокові MapReduce-завдання повністю на Python. Використовуючи mrjob, можна розробляти та відлагоджувати MapReduce-завдання локально, а потім запускати їх на кластері Hadoop або в хмарі (наприклад, Amazon EMR) без внесення змін до коду.

Приклад CSV-файлу

Для демонстрації розглянемо спрощений приклад CSV-файлу, на якому буде виконано аналіз. Структура файлу складається з наступних колонок:

– Category (текст) – категорія товару або група даних;

– Item (текст) – назва товару (необов'язково використовується в аналізі, додаткове поле);

– Amount (число) – числове значення, наприклад сума продажу товару в доларах.

Реальний CSV-файл може містити набагато більше полів і мільйони рядків. Для простоти приклад показує невеликий фрагмент даних. В лабораторній роботі слід протестувати розроблений код як на малому прикладі (для перевірки правильності), так і на великому файлі, щоб оцінити продуктивність MapReduce.

Фрагмент прикладу CSV-файлу (перші кілька рядків):

```
Category,Item,Amount
Electronics,TV,400
Clothing,Shirt,20
Electronics,Phone,300
Grocery,Milk,5
Clothing,Jeans,40
Electronics,Laptop,800
...
```

У наведених даних присутні три категорії товарів: "Electronics", "Clothing", "Grocery". Стовпець Amount містить числове значення (наприклад, ціна товару). Перший рядок є заголовком (назви колонок).

Перед запуском MapReduce на повному наборі даних переконайтеся, що формат файлу відповідає очікуваному (колонки розділені комами, символи нового рядка коректно відділяють записи, немає проблем з кодуванням файлу і т.д.). У разі потреби, можна провести попередню обробку CSV (наприклад, видалити або екранувати проблемні символи) або протестувати мапер на кількох рядках, щоб упевнитись у відсутності помилок парсингу.

Очікуваний результат

Очікуваним результатом виконання MapReduce-завдання є агреговані дані, отримані після обробки CSV-файлу. Для заданого прикладу (сума значень Amount по кожній Category) вихідні дані мають вигляд:

```
Clothing    60.0
Electronics 1500.0
Grocery     5.0
```

Кожен рядок результату містить ключ (назву категорії) та підсумкове значення (загальна сума для цієї категорії). Формат розділення такий самий, як і використовувався у виводі редуктора: ключ і значення розділені символом табуляції. У даному прикладі:

– для категорії Clothing загальна сума становить 60.0 (два товари: Shirt за 20 + Jeans за 40);

– для Electronics – 1500.0 (три товари: TV 400 + Phone 300 + Laptop 800);

– для Grocery – 5.0 (один товар: Milk за 5).

Результати агрегування підтверджують правильність роботи мапера і редуктора на тестових даних. На великому CSV-файлі програма аналогічно поверне сумарні значення (або інші потрібні метрики) для кожного ключа.

Створення завдання MapReduce

Нижче наведено послідовність кроків, які потрібно виконати, щоб проаналізувати CSV-файл за допомогою Hadoop MapReduce та Python:

1. **Підготовка середовища**. Перед початком переконайтеся, що Hadoop-кластер запущений. Якщо ви використовуєте локальний кластер (псевдорозподілений режим), стартуйте сервіси HDFS та YARN (наприклад, командами start-dfs.sh та start-yarn.sh). Перевірте, що HDFS працює (наприклад, командою hdfs dfs -ls / можна отримати список директорій в HDFS). Також впевніться, що Python та необхідні бібліотеки встановлені.

2. **Завантаження CSV-файлу в HDFS**. Створіть в HDFS каталог для вхідних даних (якщо ще не створений) і завантажте туди CSV-файл. Це можна зробити за допомогою команд HDFS. Наприклад, виконайте в консолі:

```
$ hdfs dfs -mkdir -p /user/hadoop/input
$ hdfs dfs -put /path/to/local/data.csv /user/hadoop/input/data.csv
```

У наведеному прикладі локальний файл data.csv буде скопійовано в HDFS за шляхом /user/hadoop/input/data.csv. Після виконання команди перевірте, що файл успішно завантажено (наприклад, hdfs dfs -ls /user/hadoop/input має відобразити файл).

3. **Написання коду Mapper та Reducer на Python**. Необхідно розробити два скрипти: mapper.py і reducer.py, які реалізують функції **Map** та **Reduce** відповідно. Ці скрипти будуть запускатися Hadoop Streaming на кожному вузлі для обробки частин даних.

Mapper (mapper.py): Мапер читає вхідний CSV-файл построчно із стандартного вводу (stdin), розбирає кожен рядок та генерує проміжні пари *ключ-значення*. Ключем зазвичай виступає деяка категорія або ідентифікатор, за яким будуть групуватися дані, а значенням – величина, яку потрібно агрегувати (число, лічильник тощо).

У нашому прикладі (див. розділ про приклад CSV), припустимо, потрібно обчислити загальну суму продажів по категоріях товарів. Тоді:

– ключем буде назва категорії;

– значенням–сума (або окремий запис про ціну товару, яку будемо сумувати).

При реалізації мапера зручно використовувати модуль Python csv для парсингу рядка CSV у список полів. Також потрібно пропустити рядок заголовків (header), якщо він є в CSV. Після виділення потрібних полів мапер виводить результат у стандартний вихід (stdout) у вигляді рядків "ключ<TAB>значення" – це важливо, оскільки Hadoop Streaming очікує, що ключ і значення розділені символом табуляції.

Нижче наведено приклад коду мапера на Python:

```
import sys, csv

# Створимо об'єкт читача CSV, який читатиме з stdin
reader = csv.reader(sys.stdin)
for row in reader:
    # Припустимо, CSV має колонки: Category, Item, Amount
    # Пропускаємо рядок заголовків (за деякими ознаками, наприклад, якщо перша колонка не число)
    try:
        amount = float(row[2])  # третя колонка - сума продажу
    except ValueError:
        # Якщо не вдалося конвертувати в число, ймовірно це заголовок
        continue
    category = row[0]         # перша колонка - назва категорії
    # Виводимо ключ-значення
    print(f"{category}\t{amount}")
```

У цьому коді кожен рядок CSV розбивається на поля. Якщо рядок не є заголовком, мапер виводить пару "Category<TAB>Amount". Наприклад, рядок Electronics,TV,400 буде перетворено на вихід Electronics 400. Hadoop збирає такі проміжні результати і групує їх за ключем (категорією) перед передачею редуктору.

Reducer (reducer.py): Редуктор отримує відсортовані за ключем дані (усі значення, що мають однаковий ключ, надходять послідовно). Його завдання – зібрати всі значення для кожного ключа та обчислити підсумковий результат. У нашому прикладі редуктор підсумує всі суми продажів для кожної категорії.

При читанні зі stdin редуктор також може використовувати модуль csv, але оскільки на вході будуть прості текстові пари "ключ табуляція значення", простіше розбивати рядок методом split("\t"). Необхідно відслідковувати зміну ключа: коли ключ змінюється, редуктор виводить накопичений результат для попереднього ключа і починає накопичення для нового.

Приклад коду редуктора на Python:

```
import sys

current_category = None
total = 0.0

for line in sys.stdin:
    line = line.strip()
    if not line:
        continue
    category, value = line.split("\t", 1)
    try:
        value = float(value)
    except ValueError:
        # Некоректне числове значення, пропускаємо
        continue

    # Якщо зустріли нову категорію (і не перший випадок), вивести результат для попередньої
    if current_category is not None and category != current_category:
        print(f"{current_category}\t{total}")
        total = 0.0
    current_category = category
    total += value

# Виводимо результат для останньої категорії, якщо вона існує
if current_category is not None:
    print(f"{current_category}\t{total}")
```

Цей редуктор обробляє відсортований потік пар "Category Amount". Змінна current_category відслідковує поточний ключ (категорію), а total накопичує суму. Коли категорія змінюється, редуктор виводить підсумок для попередньої категорії. Наприкінці він також виводить суму для останньої категорії. В результаті вихід редуктора також має формат "ключ<TAB>значення" –у нашому випадку "Category<TAB>TotalAmount".

Якщо замість Hadoop Streaming використовується бібліотека **mrjob**, то мапер і редуктор визначаються як методи класу-нащадка MRJob. Наприклад, можна оголосити клас class SalesSum(MRJob) з методом mapper(self, _, line) для обробки рядка (аналогічно функціоналу mapper.py) та методом reducer(self, category, values) для агрегування сум.

Запуск завдання mrjob здійснюється через інтерпретатор Python, вказуючи режим виконання. Для локального режиму достатньо запустити скрипт, а для Hadoop-кластера використовується опція -r hadoop. Наприклад: python sales_sum.py -r hadoop hdfs:///user/hadoop/input/data.csv запустить обробку на кластері з використанням HDFS-входу.

Виконання завдання MapReduce в Hadoop.

Після того як скрипти mapper.py та reducer.py готові і розміщені у локальній файловій системі, можна запустити MapReduce-завдання на Hadoop-кластері. Для цього використовується Hadoop Streaming утиліта. Суть виконання: Hadoop запускає на кластері багато копій наших скриптів, автоматично передаючи кожному маперу різні частини вхідного файлу з HDFS, а вихід маперів пересилає відповідним редукторам.

Команда для запуску Hadoop Streaming може виглядати так (все в одному рядку, тут подано з переносами для читання):

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -files mapper.py,reducer.py \
    -mapper "python mapper.py" -reducer "python reducer.py" \
    -input /user/hadoop/input/data.csv -output /user/hadoop/output
```

Пояснення параметрів:

– files mapper.py,reducer.py – вказує Hadoop скопіювати наші скрипти на кожен вузол (розділяючи кількома файлами через кому).

– mapper "python mapper.py" – команда для запуску мапера (наш Python-скрипт).

– reducer "python reducer.py" – команда для запуску редуктора.

– input ... – шлях в HDFS до вхідного файлу (наш CSV в HDFS).

– output ... – шлях в HDFS для вихідних даних (цей каталог **не повинен існувати** до запуску, Hadoop створить його).

Залежно від конфігурації Hadoop, шлях до hadoop-streaming.jar може відрізнятися. У наведеному прикладі використано змінну оточення $HADOOP_HOME. Переконайтеся, що вказуєте правильний шлях до jar-файлу Hadoop Streaming у вашій системі. Команду можна виконати в Linux-консолі прямо на вузлі, де встановлено Hadoop (або через SSH, якщо це віддалений кластер).

Після запуску, Hadoop створить завдання, розподілить її на задачі та відслідковуватиме виконання. В консолі ви побачите хід виконання (прогрес маперів/редукторів у відсотках). Якщо завдання завершилось успішно, вказаний вихідний каталог з'явиться в HDFS.

Коли MapReduce-завдання відпрацює, результати будуть збережені у вихідному каталозі в HDFS (той, що зазначено у параметрі -output). За

замовчуванням результати зберігаються в файлах з іменами на кшталт part-00000, part-00001 тощо (по одному файлу на кожен редуктор).

Для перегляду результатів можна виконати команду:

$ hdfs dfs -cat /user/hadoop/output/part-00000 | head -20

Ця команда виведе перші 20 рядків результатів на екран. Якщо редукторів було декілька і, відповідно, є кілька part-файлів, їх всі потрібно переглянути або об'єднати. У простому випадку можна скопіювати результуючі файли на локальний комп'ютер для подальшого аналізу:

bash

Копіювати

$ hdfs dfs -get /user/hadoop/output/part-00000 ./result_part0.txt

(Аналогічно для інших part-файлів, або використати -getmerge для злиття всіх частин у один локальний файл.)

Отримані результати слід проаналізувати: вони відповідають тим агрегованим показникам, які обчислював редуктор. У нашому прикладі це буде сума продажів по кожній категорії товарів. Переконайтеся, що результати виглядають коректно (наприклад, для тестового набору даних можна вручну чи за допомогою скрипту перевірити деякі підсумки).

Завдання

Виконати MapReduce-завдання для даних з БД Pagila, попередньо експортувавши у файли CSV дані з відповідних таблиць та завантаживши їх у файлову систему HDFS (можна скористатись програмним кодом з методичної частини до практичної роботи №3):

1. Підрахувати кількість оренд для кожного фільму.
   - Mapper: зчитує дані з файлу rental.csv та видає пари (film_id, 1).
   - Reducer: групує film_id та підсумовує кількість оренд.
2. Підрахувати загальну суму платежів кожного клієнта.
   - Mapper: зчитує дані з файлу payment.csv та видає пари (customer_id, amount).
   - Reducer: групує платежі customer_id та підсумовує їх.
3. Підрахувати кількість фільмів у кожній категорії.
   - Mapper: зчитує дані з файлу film.csv та видає пари (category, 1).
   - Reducer: групує категорії та підраховує кількість фльмів.

В контексті даної практичної роботи можна обрати будь-який підхід (Streaming або mrjob), але для простоти далі показано виконання із використанням Hadoop Streaming.

Робота з налаштування ПЗ №3. «Встановлення Apache Spark в операційній системі Linux»

Apache Spark – високопродуктивне рішення для обробки даних, що зберігаються в кластері Hadoop. У порівнянні з наданим у Hadoop механізмом MapReduce, Spark забезпечує у 100 разів більшу продуктивність при обробленні даних в пам'яті й 10 разів при розміщенні даних на дисках. Рушій може виконуватися на вузлах кластера Hadoop як за допомогою Hadoop YARN, так і у відокремленому режимі. Підтримується оброблення даних у сховищах HDFS, HBase, Cassandra, Hive та будь-якому форматі введення Hadoop.

Spark може використовуватися як у типових сценаріях оброблення даних, схожих на MapReduce, так і для реалізації специфічних методів, таких як потокове оброблення, SQL, інтерактивні та аналітичні запити, рішення задач машинного навчання і робота з графами. Програми для оброблення даних можуть створюватися на мовах Scala, Java, Python та R.

Встановлення Apache Spark у Linux включає кілька кроків.

1: Перевірка та встановлення Java

Spark потребує встановленої Java (версія 8 або новіша).

Перевірте, чи встановлено Java:

```
java -version
```

Якщо Java не встановлена, встановіть її:

```
sudo apt update
sudo apt install default-jdk -y
```

Перевірте версію після встановлення:

```
java -version
```

2: Встановлення Scala (за потреби)

Spark використовує Scala для своєї внутрішньої реалізації. Хоча Scala зазвичай не обов’язкова для запуску Spark, її можна встановити для розробки.

Перевірте, чи встановлено Scala:

```
scala -version
```

Встановіть Scala, якщо її немає:

```
sudo apt install scala -y
```

3: Завантаження Apache Spark

Завантажте останню версію Spark із офіційного сайту Apache Spark.

Використайте команду wget для завантаження:
wget https://downloads.apache.org/spark/spark-<version>/spark-<version>-bin-hadoop3.tgz

Замініть <version> на актуальну версію (наприклад, 3.5.0). Розпакуйте архів:
tar -xvzf spark-<version>-bin-hadoop3.tgz

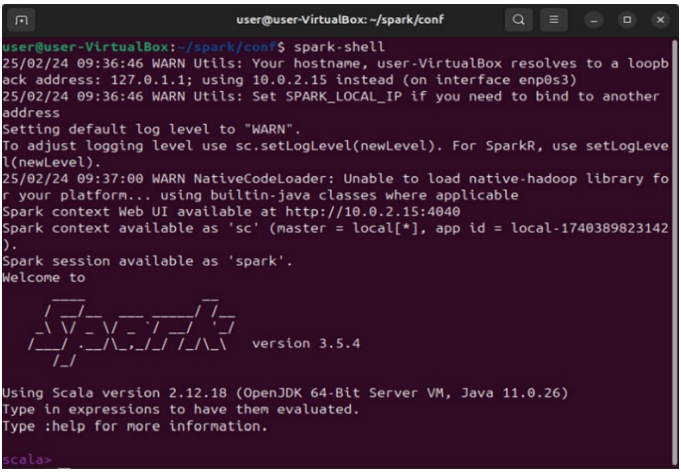
Перемістіть Spark у зручне місце:
sudo mv spark-<version>-bin-hadoop3 /opt/spark

4: Налаштування змінних середовища
Відредагуйте файл ~/.bashrc або ~/.zshrc (залежно від вашого shell):
nano ~/.bashrc

Додайте такі рядки в кінець файлу:
export SPARK_HOME=/opt/spark
export PATH=$SPARK_HOME/bin:$SPARK_HOME/sbin:$PATH
export JAVA_HOME=/usr/lib/jvm/java-<version>-openjdk

Замініть <version> на встановлену версію Java.
Оновіть змінні середовища:
source ~/.bashrc

5: Перевірка встановлення
Запустіть Spark shell, щоб перевірити роботу:
spark-shell



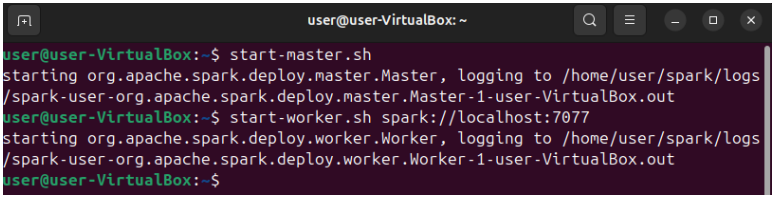
Мал. 50 – Перевірка встановлення Apache Spark через запуск spark-shell

Ви повинні побачити консоль Scala з повідомленням про запуск Spark.
Вийдіть з консолі Spark:
:quit

6: Запуск Spark
Для запуску Spark у режимі standalone:
start-master.sh

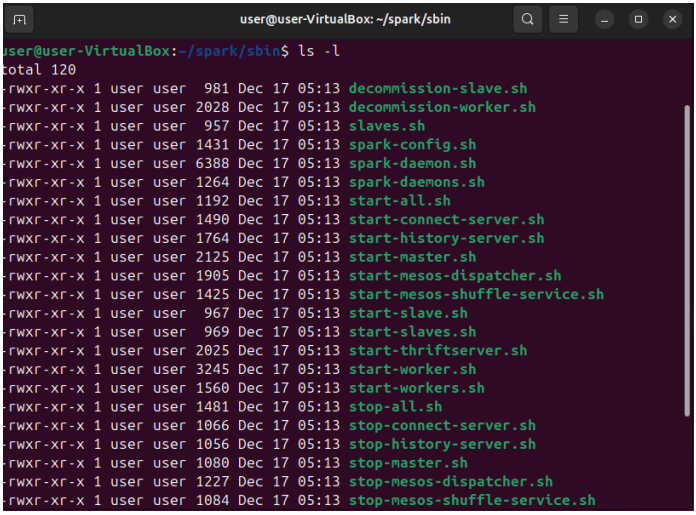
Це запустить Master Node.
Потім запустіть Worker Node:
start-worker.sh spark://<master-hostname>:7077

Замініть <master-hostname> на ваш локальний хост або IP-адресу.



Мал. 51 – Запуск Master Node та Worker Node в Apache Spark

Необхідні скрипти знаходяться у папці $SPARK_HOME/sbin:



Мал. 52 – Скрипти керування службами Apache Spark у каталозі sbin

Служби Apache Spark

Apache Spark складається з кількох ключових компонентів (служб), що працюють разом для розподілених обчислень. Основні служби Spark включають:

1. **Spark Master:** це головна служба, яка координує весь кластер. Вона відповідає за розподіл ресурсів, планування завдань та моніторинг виконання додатків;
2. **Spark Worker:** запускається на робочих вузлах кластера. Кожен Worker отримує завдання від Master та виконує їх за допомогою процесів, які називаються executors;
3. **Driver Program:** це програма, яку запускає користувач (ваш код Spark). Driver координує виконання завдань, розподіляючи їх між executors, відстежує стан завдань та збирає результати.
4. **Executors:** це процеси, які працюють на Worker-вузлах і виконують обчислення за завданнями, отриманими від Driver. Вони відповідають за виконання конкретних частин обчислювального процесу.
5. **History Server:** служба для зберігання та перегляду історії виконання додатків. Вона дозволяє аналізувати минулі запуски Spark через веб-інтерфейс.

Крім того, Spark може інтегруватися з різними менеджерами кластерів (наприклад, YARN, Apache Mesos, Kubernetes), які допомагають у розподілі ресурсів і управлінні кластерами.

Ці служби разом забезпечують гнучкість та масштабованість Spark, дозволяючи ефективно виконувати великі обчислювальні завдання.

Налаштування запуску Spark у supervisor

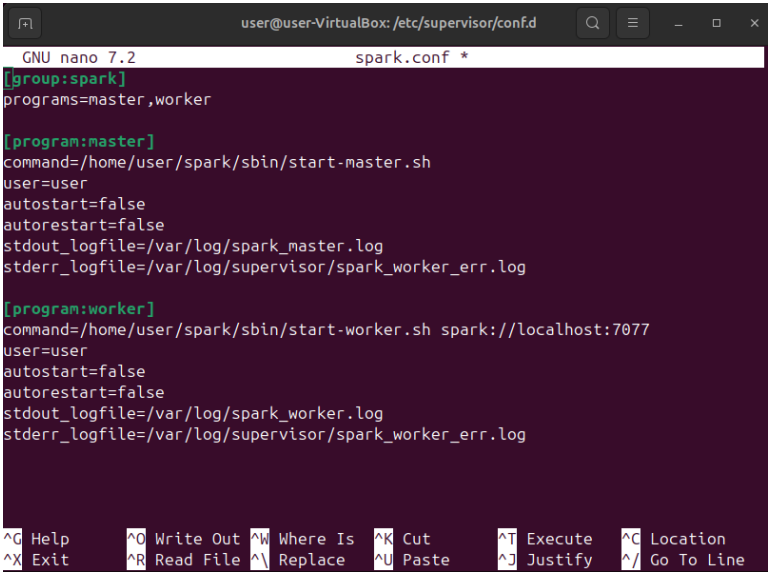
Нижче наведено приклад того, як можна налаштувати автоматичний запуск Spark за допомогою Supervisor. Припустимо, що Spark встановлено в каталозі /opt/spark, а користувач для запуску – sparkuser. Також важливо, щоб процеси Spark не від'єднувалися у фоновий режим (daemonize). Для цього використовується змінна середовища SPARK_NO_DAEMONIZE=1.

Створіть файл, наприклад, /etc/supervisor/conf.d/spark.conf з наступним змістом:

```
[group:spark]
programs=master,worker

[program:master]
command=/home/user/spark/sbin/start-master.sh
user=user
autostart=false
autorestart=false
stdout_logfile=/var/log/spark_master.log
stderr_logfile=/var/log/supervisor/spark_master_err.log
```

```
[program:worker]
command=/home/user/spark/sbin/start-worker.sh spark://localhost:7077
user=user
autostart=false
autorestart=false
stdout_logfile=/var/log/spark_worker.log
stderr_logfile=/var/log/supervisor/spark_worker_err.log
```



Мал. 53 – Перевірка стану сервісів Apache Spark через Supervisor

Перезавантажте конфігурацію Supervisor:
sudo supervisorctl reread
sudo supervisorctl update

Перевірте статус процесів:
sudo supervisorctl status

Потім запустіть групу сервісів spark:
sudo supervisorctl start spark:

Таким чином, Supervisor автоматично запустить Spark Master та Worker при старті системи, а у разі збою процеси будуть перезапускатися.

Завдання

Маючи віртуальну машину Linux або локальну систему (будь-який дистрибутив) встановити Spark та налаштувати автоматизований запуск його сервісів через Supervisor.

Практична робота №6. «Робота зі структурованими даними за допомогою SparkSQL»

Apache Spark – це розподілена обчислювальна система з відкритим кодом, розроблена для швидкої обробки великих обсягів даних. Spark підтримує паралельні обчислення та використовується для аналізу даних, машинного навчання, обробки потоків та інтерактивного аналізу.

Spark має перевагу над традиційним Hadoop MapReduce завдяки ефективному використанню пам'яті та оптимізованій моделі обчислень, що дозволяє досягати значно вищої продуктивності.

Apache Spark містить кілька основних бібліотек:

– **Spark Core** – базовий компонент для обробки даних та керування виконанням задач.

– **Spark SQL** – модуль для роботи з даними у форматі SQL.

– **Spark Streaming** – компонент для обробки потокових даних у реальному часі.

– **MLlib** – бібліотека для машинного навчання.

– **GraphX** – модуль для обробки графових структур.

RDD (Resilient Distributed Dataset) – це основна абстракція в Apache Spark, яка представляє розподілену колекцію об'єктів. Основні властивості RDD:

– розподіленість – дані розподілені між вузлами кластера;

– стійкість (resiliency) – автоматичне відновлення даних у разі збою вузла;

– незмінність (immutability) – RDD не змінюється після створення, всі операції створюють нові RDD;

– трансформації та дії;

– трансформації (map, filter, flatMap, groupByKey) – створюють нові RDD;

– дії (collect, count, reduce, saveAsTextFile) – ініціюють виконання обчислень.

Таблиця 6 – Порівняння Apache Spark і Hadoop MapReduce

Характеристика	Apache Spark	Hadoop MapReduce
Швидкість	Висока (in-memory computing)	Низька (запис на диск)
Гнучкість	Підтримка SQL, потокової обробки, ML	Лише пакетна обробка

Програмування	Простий API (Scala, Python, Java)	Складний API (Java)
Управління помилками	Відновлення RDD	Запуск повторних задач

Apache Spark підтримує кілька мов програмування:

– Scala – основна мова розробки для Spark;
– Python (PySpark) – популярна для аналітики та ма шинного навчання;
– Java – використовується у корпоративних рішеннях;
– R – підходить для статистичних обчислень.

PySpark

PySpark – це Python API для Apache Spark, що дозволяє взаємодіяти з розподіленою обчислювальною системою Spark, використовуючи мову Python. Він забезпечує потужні можливості для обробки великих даних, машинного навчання та аналізу даних.

PySpark підтримує всі основні компоненти Spark:

– **PySpark Core** – базові операції з RDD;
– **PySpark SQL** – робота з даними у вигляді таблиць та SQL-запитів;
– **PySpark Streaming** – обробка потокових даних у реальному часі;
– **MLlib (PySpark ML)** – бібліотека машинного навчання;
– **GraphFrames** – обробка графових структур.

Основні компоненти PySpark

– **SparkContext** – основний об'єкт для взаємодії зі Spark-кластером.
– **SparkSession** – спрощений інтерфейс для роботи з даними.
– **RDD** – основна структура даних у Spark.
– **DataFrame** – оптимізоване табличне представлення даних, аналогічне Pandas.
– **Transformations** та **Actions** – дві категорії операцій у PySpark.

Методи DataFrame у Spark, що є аналогами SQL-запитів

Spark SQL надає API для роботи з DataFrame, який дозволяє виконувати операції, аналогічні SQL-запитам. Нижче наведено основні методи Spark DataFrame разом із їхніми SQL-еквівалентами.

1. Вибірка колонок

Spark DataFrame:

```
df.select("column1", "column2").show()
```

SQL еквівалент:

```
SELECT column1, column2 FROM table;
```

2. Фільтрація даних
Spark DataFrame:
```
df.filter(df.age > 30).show()
df.where(df.age > 30).show()
```
SQL еквівалент:
```
SELECT * FROM table WHERE age > 30;
```

3. Сортування
Spark DataFrame:
```
df.orderBy(df.age.desc()).show()
```
SQL еквівалент:
```
SELECT * FROM table ORDER BY age DESC;
```

4. Групування та агрегація
Spark DataFrame:
```
df.groupBy("department").count().show()
df.groupBy("department").agg({"salary": "avg"}).show()
```
SQL еквівалент:
```
SELECT department, COUNT(*) FROM table GROUP BY department;
SELECT department, AVG(salary) FROM table GROUP BY department;
```

5. Об’єднання таблиць (JOIN)
Spark DataFrame:
```
df1.join(df2, df1.id == df2.id, "inner").show()
```
SQL еквівалент:
```
SELECT * FROM table1 INNER JOIN table2 ON table1.id = table2.id;
```

Типи об'єднань у Spark аналогічні SQL:

- "inner" – INNER JOIN
- "left" – LEFT JOIN
- "right" – RIGHT JOIN
- "outer" – FULL OUTER JOIN

6. Лімітування кількості записів
Spark DataFrame:
```
df.limit(10).show()
```
SQL еквівалент:
```
SELECT * FROM table LIMIT 10;
```

7. Видалення дублікатів

Spark DataFrame:

```
df.dropDuplicates(["column1"]).show()
```

SQL еквівалент:

```
SELECT DISTINCT column1 FROM table;
```

8. Додавання нових колонок

Spark DataFrame:

```
df.withColumn("new_column", df["existing_column"] * 2).show()
```

SQL еквівалент:

```
SELECT existing_column, existing_column * 2 AS new_column FROM table;
```

9. Видалення колонок

Spark DataFrame:

```
df.drop("column_to_remove").show()
```

SQL еквівалент:

```
ALTER TABLE table DROP COLUMN column_to_remove;
```

10. Виконання SQL-запитів у Spark

Spark дозволяє виконувати SQL-запити безпосередньо, використовуючи тимчасові таблиці.

```
df.createOrReplaceTempView("people")
spark.sql("SELECT name, age FROM people WHERE age > 30").show()
```

Приклад

У наступному прикладі програмний код використовує Apache Spark для обробки даних про лауреатів премії "Оскар". Розглянемо його поетапно:

1. Імпорт бібліотек:

```
from pyspark.sql import SparkSession
import pyspark.sql.functions as F
```

Цей блок імпортує необхідні модулі для роботи зі Spark DataFrame та функціями обробки даних.

2. Створення SparkSession:

```
spark = SparkSession\
    .builder\
    .master('local')\
    .appName("OskarApp")\
    .getOrCreate()
```

Створюється локальна Spark-сесія з назвою " OskarApp", яка дозволяє виконувати розподілену обробку даних.

3. Завантаження даних із CSV-файлів:

```
df = spark.read\
        .option("header", True)\
        .option("inferSchema", True)\
        .csv("/oscar_age_female.csv")

oscar_female = spark.read\
        .option("header", True)\
        .option("inferSchema", True)\
        .csv("/oscar_age_female.csv")

oscar_male = spark.read\
        .option("header", True)\
        .option("inferSchema", True)\
        .csv("/oscar_age_male.csv")
```

Spark завантажує два CSV-файли:

– oscar_age_female.csv – дані про переможниць "Оскара";

– oscar_age_male.csv – дані про переможців "Оскара".

Опція header=True вказує, що перший рядок CSV містить заголовки. Опція inferSchema=True дозволяє автоматично визначити типи даних.

4. Обробка даних (очищення, перейменування колонок):

```
df = df\
   .withColumnRenamed(' "Year"', 'year')\
   .withColumnRenamed(' "Age"', 'age')\
   .withColumnRenamed(' "Name"', 'name')\
   .withColumnRenamed(' "Movie"', 'movie')\
   .withColumn('year', F.col('year').cast('int'))\
   .withColumn('age', F.col('age').cast('int'))\
   .drop('Index')
```

Аналогічні операції виконуються для oscar_female та oscar_male.
Що відбувається в цьому блоці:

– перейменовуються колонки (withColumnRenamed);

– колонки year та age перетворюються у числовий тип (cast('int'));

– видаляється колонка 'Index' (drop('Index')).

5. Об'єднання даних за роками:

```
oscar = oscar_female\
   .join(oscar_male, oscar_female.year == oscar_male.year)\
   .select(oscar_female.year
         , oscar_female.name.alias('female')
         , oscar_male.name.alias('male'))\
   .show()
```

join(oscar_male, oscar_female.year == oscar_male.year): з'єднує чоловічі та жіночі нагороди за спільним значенням year.

.select(...): вибирає потрібні колонки (year, female – переможниця, male – переможець).

.show(): виводить результати.

Завдання

З демо PostgreSQL бази даних (практична робота №1) завантажте таблиці в PySpark DataFrames.

Використовуючи ці дані (без SQL), напишіть PySpark код для того, щоб:

1. вивести кількість фільмів у кожній категорії, відсортувати за спаданням.
2. вивести 10 акторів, чиї фільми найбільше орендували, відсортувати за спаданням.
3. вивести категорію фільмів, на яку витратили найбільше грошей.
4. вивести назви фільмів, яких немає в inventory.
5. вивести топ 3 акторів, які найбільше з'являлися у фільмах категорії “Children”. Якщо у декількох акторів однакова кількість фільмів, вивести всіх.
6. вивести міста з кількістю активних та неактивних клієнтів (активний – customer.active = 1). Відсортувати за кількістю неактивних клієнтів за спаданням.
7. вивести категорію фільмів, у якої найбільша кількість годин сумарної оренди в містах (customer.address_id в цьому city), і які починаються на букву “a”. Те саме зробити для міст у яких є символ “-”.

Практична робота №7. «Побудова data pipeline за допомогою Spark, HDFS та Apache Airflow»

Медальйонна архітектура (Medallion Architecture) є загальноприйнятим підходом до організації даних у розподілених системах обробки. Вона передбачає три рівні:

- **Bronze (Сирі дані):** первинне збереження даних без змін.
- **Silver (Оброблені дані):** очищення, трансформація та агрегування.

– **Gold (Аналітичні дані):** підготовка структурованих наборів даних для аналітики та бізнес-інтелекту.

Ці рівні допомагають структурувати ETL-процеси та зробити їх більш зрозумілими та керованими.

Наступний приклад реалізує процес ETL (Extract, Transform, Load) на базі медальйонної архітектури (Bronze -> Silver -> Gold) за допомогою PySpark, PostgreSQL та HDFS. Він виконує такі основні кроки:

- завантаження сирих даних у Bronze (HDFS);
- обробка даних у Silver (Parquet-файли);
- агрегація та збереження в Gold (Greenplum SQL база).

1. Ініціалізація середовища

```
from pyspark.sql import SparkSession
import pyspark.sql.functions as F
import psycopg2
from hdfs import InsecureClient
import os
from datetime import datetime
```

- PySpark використовується для обробки великих даних.
- psycopg2 дозволяє підключитися до PostgreSQL.
- hdfs.InsecureClient використовується для взаємодії з HDFS.
- os, datetime потрібні для роботи з файлами та часом.

Налаштування з'єднань з БД

```
pg_url = "jdbc:postgresql://172.20.10.10:5432/postgres"
pg_properties = {"user": "pguser", "password": "secret"}

gp_url = "jdbc:postgresql://172.20.10.10:5433/postgres"
gp_properties = {"user": "gpuser", "password": "secret"}
```

- pg_url та gp_url – з'єднання з PostgreSQL та Greenplum.
- pg_properties та gp_properties – облікові дані.

Створення Spark-сесії

```
spark = SparkSession.builder\
   .config('spark.driver.extraClassPath','/home/user/postgresql-42.7.3.jar')\
    .master('local')\
    .appName("lesson")\
    .getOrCreate()
```

Дії:
- Ініціалізує Spark для роботи з даними.
- Додає JAR-драйвер для PostgreSQL.
- Визначає локальний режим роботи.

2. Завантаження даних у BRONZE (HDFS)

Налаштування підключення до PostgreSQL

```
pg_creds = {
   'host': 'localhost',
   'port': '5432',
   'database': 'pagila',
   'user': 'pguser',
   'password': 'pguser'
}
```

Зберігає параметри підключення до PostgreSQL.

Збереження таблиць у HDFS (Bronze Layer)

```
hdfs_url = 'http://localhost:50070/'
tables_to_load = ('film', 'film_actor', 'actor', 'film_category', 'category','language')
current_date = datetime.now().strftime("%Y-%m-%d")
client = InsecureClient(hdfs_url, user='user')
```

- hdfs_url – URL-сховище HDFS.
- tables_to_load – список таблиць, які треба скопіювати.
- current_date – додає мітку часу.
- client – клієнт HDFS.

Цикл копіювання таблиць у HDFS

```
for table_name in tables_to_load:
   with psycopg2.connect(**pg_creds) as pg_connection:
      cursor = pg_connection.cursor()
      with client.write(os.path.join('/bronze', current_date, table_name +
'.csv')) as csv_file:
         cursor.copy_expert(f"COPY {table_name} TO STDOUT WITH
HEADER CSV", csv_file)
```

Дії:
- Підключається до PostgreSQL.
- Використовує COPY TO STDOUT для збереження таблиць у форматі CSV.

- Записує файли у HDFS (/bronze/YYYY-MM-DD/).

3. Обробка даних у SILVER (Parquet-файли)

Обробка таблиці film

```
bronze_film_df = spark.read.load(f"/bronze/{current_date}/film.csv",
                    header="true", inferSchema="true", format="csv")

bronze_film_df = bronze_film_df.drop('original_language_id', 'special_features', 'fulltext')
bronze_film_df = bronze_film_df.where(F.col('length') >= 50)

bronze_film_df.write.partitionBy('rating').parquet("/silver/film", mode='overwrite')
```

Дії:
- Читає CSV-файл з Bronze.
- Видаляє непотрібні колонки.
- Фільтрує фільми, довжина яких ≥ 50 хв.
- Зберігає у Silver як Parquet, розділяючи за rating.

Обробка інших таблиць

```
film_actor_df = spark.read.load(f"/bronze/{current_date}/film_actor.csv", header="true", inferSchema="true", format="csv")
film_actor_df.write.parquet("/silver/film_actor", mode='overwrite')
```

Те саме виконується для таблиць: actor, film_category, category, language. Всі таблиці зберігаються у форматі Parquet.

4. Об'єднання та агрегація у GOLD

Завантаження даних із Silver

```
film_df = spark.read.parquet("/silver/film")
film_category_df = spark.read.parquet("/silver/film_category")
category_df = spark.read.parquet("/silver/category")
film_actor_df = spark.read.parquet("/silver/film_actor")
actor_df = spark.read.parquet("/silver/actor")
```

Дія:
- Читає Silver-файли.

Об'єднання категорій

```
film_category_df = film_category_df.join (category_df, film_category_df.category_id == category_df.category_id, 'left') .select (film_category_df.film_id, category_df.name.alias('category_name'))
```

film_df = film_df.join (film_category_df, film_df.film_id == film_category_df.film_id, 'left') .select (film_df['*'], film_category_df.category_name)

Дія:

- З'єднує film_category з category, додаючи назву категорії до film.

Об'єднання акторів

film_actor_df = film_actor_df.join(actor_df, film_actor_df.actor_id == actor_df.actor_id, 'left')\
.select(film_actor_df.film_id, F.concat(F.col('first_name'), F.lit(' '), F.col('last_name')).alias("name"))
film_df = film_df.join(film_actor_df, film_df.film_id == film_actor_df.film_id, 'left')\
.select(film_df['*'], film_actor_df.name)

Дія:

- З'єднує film_actor з actor, додаючи імена акторів.

Агрегація даних

```
films = film_df.groupBy(
  F.col('film_id'), F.col('title'), F.col('description'), F.col('rental_duration'),
  F.col('rental_rate'), F.col('length'), F.col('replacement_cost'),
  F.col('rating'), F.col('category_name')
).agg(F.collect_list(F.col('name')).alias('actors'))
films.write.parquet("/gold/films", mode='overwrite')
```

Дії:

- Групує дані за фільмами, створюючи список акторів.
- Зберігає у Gold.

Запис у Greenplum

```
films = films.select (films['*'], F.explode (F.col('actors')) .alias('actor')). drop('actors')
films.write.jdbc(gp_url, table='films', properties=gp_properties, mode='overwrite')
```

Дії:

- Розширює список акторів.
- Записує в Greenplum.

Завдання

Побудуйте data pipeline за допомогою Spark, HDFS, Airflow:

1. Вивантажте дані із dshop PostgreSQL таблиць у Bronze.

2. Вивантажте дані на поточний день із Currency API. Якщо даних на сьогоднішній день немає, то Airflow повинен видавати помилку з відповідним log message в Логах Airflow Task.

3. Дані Bronze повинні бути партиційовані за датою.

4. Перенесіть дані з Bronze до Silver, попутно очистивши їх та прибравши дублікати.

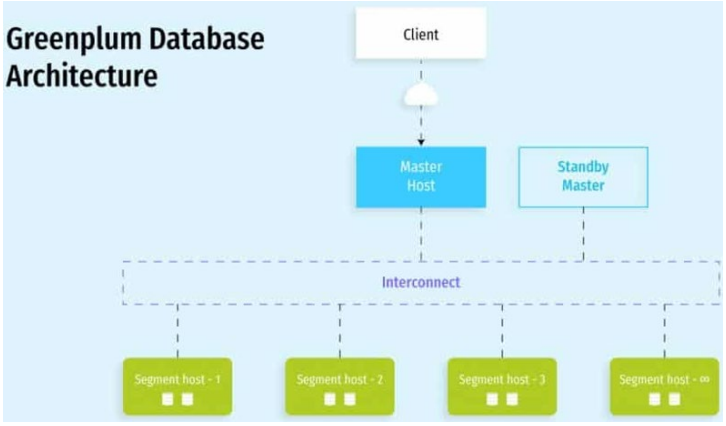
5. Дані Silver повинні зберігатися у форматі Parquet.

6. Додайте Airflow логи на кожну дію.

Обробка даних із PostgreSQL та з API повинна бути в різних Airflow DAG. Кожен окремий крок обробки – окремий Airflow Task.

Робота з налаштування ПЗ №4. «Встановлення та запуск CloudberryDB зі створенням кластера»

CloudberryDB – це сучасна система управління базами даних масового паралельного оброблення (MPP) з відкритим кодом, створена на основі PostgreSQL 14.4. Вона є відкритою альтернативою Greenplum Database і забезпечує високопродуктивний аналіз великих обсягів даних за рахунок розподілу навантаження між кількома вузлами. Архітектурно CloudberryDB складається з координатора (аналог “master” у Greenplum) і множини сегментів (вузлів з даними). Координатор приймає SQL-запити від користувачів, планує їх виконання та розподіляє завдання між сегментами, а також агрегує результати. Сегменти зберігають частини даних і виконують паралельні обчислення. За потреби CloudberryDB може мати резервний координатор (standby) для високої доступності та дзеркальні сегменти для резервування даних.



Мал. 54 – Приклад архітектури MPP-кластера CloudberryDB / Greenplum

Основне призначення CloudberryDB – це розгортання аналітичних сховищ даних та виконання складних SQL-запитів до багатотерабайтних (і петабайтних) наборів даних з максимальною продуктивністю. Далі наведено детальні кроки для встановлення CloudberryDB з нуля на Ubuntu (з примітками для CentOS/RHEL) та розгортання локального кластера бази даних.

1. Підготовка середовища

Перед встановленням CloudberryDB необхідно підготувати систему та переконатися, що присутні всі необхідні компоненти:

– Python 3.8 – CloudberryDB використовує Python для утиліт управління кластером. Рекомендовано Python версії 3.8 (деякі скрипти можуть бути несумісні з новішими версіями).

– virtualenv – ізолює Python-середовище для встановлення потрібних пакетів (paramiko, pygresql тощо).

– Компілятор GCC та інструменти збірки – потрібні для компіляції вихідного коду (GNU Compiler Collection, make та ін.).

Перевірка Python 3.8:

```
python3 --version
```

Якщо команда вище не показує Python 3.8, встановіть його (для Ubuntu 20.04+):

```
sudo apt update
sudo apt install python3.8 python3.8-venv python3.8-dev
```

Примітка: На Ubuntu 22.04 за замовчуванням Python 3.10. Ви можете встановити 3.8 паралельно (як показано вище) або скористатись pyenv для керування версіями. На RHEL/CentOS 8+ Python 3.8 встановлюється пакетами python38 та python38-devel через yum/dnf.

Створення та активація virtualenv:

```
python3.8 -m venv cloudberry-venv   # створюємо віртуальне середовище
source cloudberry-venv/bin/activate  # активуємо його
python --version          # перевіряємо, що тепер використовується Python 3.8
```

Після активації virtualenv всі потрібні Python-бібліотеки будемо встановлювати локально в цьому середовищі, щоб ізолювати їх від системних. Перевірка наявності компілятора:

```
gcc --version
```

Якщо GCC не встановлено, він буде доданий на наступному кроці разом з іншими залежностями.

2. Встановлення необхідних системних бібліотек та пакетів

Для успішної компіляції CloudberryDB з вихідного коду потрібен ряд системних бібліотек і інструментів:

– Базові інструменти розробки: build-essential (включає gcc, g++, make та інші утиліти).

– Заголовки та бібліотеки PostgreSQL: libpq-dev (містить клієнтські бібліотеки libpq, необхідні під час збірки).

– Бібліотеки для розширень: readline, zlib, OpenSSL (додатково забезпечують підтримку консолі psql, стиснення, шифрування).

– Парсер генератори: flex, bison (використовуються для побудови SQL-парсера PostgreSQL).

Виконайте команду встановлення залежностей в Ubuntu:

– sudo apt update

– sudo apt install -y build-essential libpq-dev libreadline-dev zlib1g-dev flex bison libssl-dev

Примітка для CentOS/RHEL: Рекомендовано виконати sudo dnf groupinstall -y "Development Tools" для встановлення базових інструментів. Інші пакети: sudo dnf install -y libpq-devel readline-devel zlib-devel openssl-devel flex bison. (Для CentOS Stream 9 може знадобитися спершу увімкнути репозиторій CRB для доступу до деяких пакетів.)

Пояснення:

– build-essential – містить компілятори gcc/g++ та make.

– libpq-dev – девелоперські файли клієнтської бібліотеки PostgreSQL (потрібно для компіляції механізмів доступу до БД).

– libreadline-dev – для інтерактивної консолі psql (щоб збірка підтримувала історію команд і автодоповнення).

– zlib1g-dev – для підтримки стиснення (в PostgreSQL архівування WAL тощо).

– libssl-dev – для підтримки SSL (шифрування з'єднань).

– flex і bison – генератори парсерів, потрібні щоб скомпілювати SQL-парсер, якщо він не згенерований у вихідному коді.

Встановлення Python-бібліотек: Після успішного встановлення системних залежностей, встановимо Python-модулі, необхідні утилітам CloudberryDB:

pygresql, psutil, paramiko (і інші). Ці пакети потрібні для роботи скриптів керування кластером (наприклад, gpinitsystem, gpstart, gpssh тощо):

```
pip install pygresql psutil paramiko
```

Ця команда (виконана всередині раніше створеного cloudberry-venv) встановить:

PyGreSQL – Python-модуль для взаємодії з PostgreSQL (використовується CloudberryDB для деяких внутрішніх скриптів).

psutil – для отримання системної інформації (напр., утиліта gpstate перевіряє процеси сегментів, їх стан).

paramiko – Python-бібліотека для SSH (використовується gpssh та інші інструменти для розсилання команд на вузли кластера через SSH).

Примітка: В середовищах на кшталт CentOS/RHEL можна альтернативно використовувати системні пакунки (python3-psycopg2, python3-psutil, python3-paramiko), але рекомендація – встановити через pip останні сумісні версії.

Після цього кроку середовище готове до збірки: встановлено компілятор, необхідні dev-бібліотеки та Python-пакети.

3. Клонування репозиторію CloudberryDB та збірка з вихідного коду

CloudberryDB розповсюджується як вихідний код (проект Apache Incubating). Клонування репозиторію дасть вам останню версію коду:

```
git clone https://github.com/cloudberrydb/cloudberrydb.git
cd cloudberrydb
```

Це завантажить вихідний код у директорію cloudberrydb/. Далі потрібно налаштувати збірку (configure) і скомпілювати код. Налаштування конфігурації збірки: Запустіть скрипт configure з потрібними параметрами. Основні ключі:

– --prefix – місце, куди буде встановлено CloudberryDB після компіляції.

– --disable-orca – відключити оптимізатор ORCA під час збірки (спочатку зробимо це, щоб спростити процес і уникнути можливих помилок на етапі компіляції).

– --with-python – увімкнути підтримку PL/Python (і використати python, який зараз у PATH – наш Python 3.8).

Наприклад, щоб встановити CloudberryDB в каталог /usr/local/cloudberrydb:

```
./configure --prefix=/usr/local/cloudberrydb --disable-orca --with-python
```

Пояснення: опція --disable-orca відключає збірку нового оптимізатора ORCA. Без неї збірка включатиме компоненти ORCA (написані на C++), що потребує додаткових бібліотек (наприклад, Xerces-C для роботи з XML) і може спричинити помилки компіляції на нових версіях GCC. Ми додамо ORCA пізніше за бажанням.

Опція --with-python вказує використовувати Python для підтримки процедур на Python (PL/Python) та утиліт. Скрипт configure сам знайде шлях до Python 3.8 (оскільки ми активували virtualenv з потрібною версією).

За замовчуванням також буде зібрано підтримку PL/pgSQL; при потребі можна додати --with-perl для PL/Perl (необхідний libperl-dev) чи --with-libxml для XML-функцій (libxml2-dev), але ці опції не обов'язкові для базового запуску кластера. Після успішного виконання configure в кінці з'явиться зведення знайдених параметрів та увімкнених можливостей.

Якщо якийсь із критичних компонентів відсутній, configure повідомить про помилку. У випадку успіху – переходимо до компіляції: Компіляція та встановлення:

```
make -j$(nproc)
sudo make install
```

Команда make запустить процес компіляції (параметр -j прискорює збірку, використовуючи всі доступні CPU; ви також можете вказати конкретну кількість). Цей процес може зайняти кілька хвилин (залежно від потужності системи). По завершенні (якщо все пройшло успішно) виконайте make install для встановлення зібраних бінарних файлів, бібліотек і скриптів у директорію, вказану в --prefix.

Примітка: Якщо ви виконуєте збірку під звичайним користувачем і не маєте прав запису в /usr/local, команда make install може потребувати sudo. Альтернативно, ви могли обрати префікс у домашній каталог (наприклад, --prefix=$HOME/cloudberrydb), тоді встановлення не вимагає підвищених прав, і весь кластер знаходиться в домашній теці. У цьому прикладі ми припускаємо використання /usr/local/cloudberrydb з правами адміністратора для встановлення, після чого передамо керування кластером звичайному користувачу.

Налаштування середовища CloudberryDB: Після встановлення необхідно налаштувати змінні середовища для роботи з CloudberryDB. Найпростіше – виконати скрипт greenplum_path.sh, який було встановлено в директорію prefix. Він встановлює необхідні змінні (GPHOME, PATH, LD_LIBRARY_PATH тощо):

```
source /usr/local/cloudberrydb/greenplum_path.sh
```

Ця команда встановить GPHOME=/usr/local/cloudberrydb (корінь вашої установки) і доповнить PATH шляхом до утиліт ($GPHOME/bin). Переконайтесь, що тепер утиліти доступні, наприклад:

which gpstate

має вказати шлях на /usr/local/cloudberrydb/bin/gpstate, а команда postgres --version – вивести версію PostgreSQL/CloudberryDB. Для постійного збереження цих змін додайте рядок source /usr/local/cloudberrydb/greenplum_path.sh у ваш ~/.bashrc. (Якщо плануєте керувати кластером від імені користувача gpadmin, додайте цю строку в ~/.bashrc користувача gpadmin).

4. Виправлення типових помилок при компіляції (якщо виникають)

Під час компіляції із вихідного коду можуть виникнути певні помилки, пов'язані з версіями інструментів або компонентів. Розглянемо типові проблеми та шляхи їх розв'язання:

1. **Помилка типу "overloaded virtual/Equals" (перевантаження Equals):** Якщо компілятор GCC перериває збірку з повідомленням про приховану перевантажену функцію Equals в коді ORCA, це виглядає так:

error: 'virtual gpos::BOOL gpos::CWStringBase::Equals(const gpos::WCHAR*) const' was hidden [-Werror=overloaded-virtual]

...

cc1plus: all warnings being treated as errors

Ця помилка виникає в C++-коді оптимізатора ORCA (бібліотека gpos) на нових версіях GCC. Оскільки ми відключили ORCA (--disable-orca), подібної помилки не мало б виникнути. Якщо ж вона з'явилася, переконайтеся, що конфігурація виконана з --disable-orca або вимкніть режим -Werror для цих попереджень.

Один зі способів – відредагувати файл src/backend/gporca/libgpos/include/gpos/ string/CWStringBase.h і додати ключове слово override до визначення Equals, але це складніший шлях.

Простіше – відключити трактування попереджень як помилки. Для цього можна прибрати прапор -Werror із глобальних налаштувань збірки. Наприклад, відкрийте файл ~/cloudberrydb/Makefile.global (або аналогічний глобальний Makefile, згенерований configure) і видаліть -Werror із параметрів компіляції, після чого запустіть make знову. Це дозволить ігнорувати попередження типу **overloaded**-virtual і продовжити збірку.

2. **Проблеми через -Werror (попередження, що переросли у помилку):** подібно до вищенаведеного, компіляція може зупинитися на

інших попередженнях (наприклад, stringop-overflow чи інших), якщо вони підпадають під прапор -Werror.

Сучасні компілятори можуть бути “прискіпливішими” до коду. Рішення аналогічне – тимчасово вимкнути -Werror. Це можна зробити, вказавши змінні середовища при запуску configure або make. Наприклад, перед повторним запуском configure можна встановити:

```
export CFLAGS="-Wno-error"
export CXXFLAGS="-Wno-error"
```

або відредагувати глобальний Makefile. Після цього виконати make повторно. Попередження все ще будуть видно в консолі, але не перервуть компіляцію. Звісно, в ідеалі виправити джерело попередження в коді, але для навчальної збірки це не обов’язково.

3. **Несумісність версій Python або бібліотек:** Якщо не дотримано вимогу щодо Python 3.8, можуть виникати проблеми з залежностями. Наприклад, інструменти CloudberryDB можуть очікувати певні версії модулів.

Відомий випадок – версія PyYAML: у конфігурації CloudberryDB може бути закріплена певна версія (наприклад, <7.0), і якщо у системі/venv встановлена новіша, установка залежностей може спочатку завершитися з помилкою. Ми не встановлювали PyYAML окремо, але якщо буде потрібно (наприклад, для утиліти gpupgrade чи інших), слід встановити сумісну версію, наприклад: pip install 'PyYAML<7.0'.

Також переконайтеся, що версія GCC достатньо сучасна (не старіша 7), але й не занадто нова. На практиці GCC 9–12 підходять. Якщо у вас дуже новий дистрибутив з GCC 13+, можуть з’явитися нові попередження – тоді використовуйте наведений вище підхід із вимкнення -Werror.

Загалом, дотримання рекомендованих версій (Python 3.8, GCC 9-11, актуальні залежності) забезпечує успішну збірку CloudberryDB. Після подолання можливих труднощів переходимо до ініціалізації кластера.

5. Встановлення та ініціалізація локального кластера CloudberryDB

Після успішної компіляції та встановлення CloudberryDB, можна створити локальний кластер бази даних – тобто запустити процес координатора і сегментів на вашій машині. Ми налаштуємо кластер з одним координатором та двома сегментами на одному вузлі (для демонстрації паралельної роботи).

Користувач для кластера: рекомендовано запускати СКБД від імені окремого невпривілейованого користувача (традиційно в Greenplum

використовують користувача gpadmin). У нашому випадку, якщо всі кроки вище виконувалися під звичайним користувачем (не root), ви можете продовжити ним і надалі. Якщо ж збірка виконувалась під root, створіть окремого користувача для запуску БД та змініть власника каталогу з встановленим CloudberryDB і майбутніх каталогів даних на цього користувача.

Припустимо, що ми працюємо під користувачем user (замініть на свій логін або gpadmin залежно від ситуації).

1. Налаштування SSH-доступу (безпарольного):

Утиліти CloudberryDB (Greenplum) використовують SSH для запуску процесів сегментів на всіх вузлах (навіть якщо вузол один – він підключається до себе). Тому необхідно налаштувати безпарольний SSH-вхід для вашого користувача на localhost. Якщо ви вже робили це раніше (наприклад, у лабораторній з Hadoop), то можете пропустити, інакше виконайте:

```
# Встановити SSH-сервер, якщо досі не встановлений (Ubuntu):
sudo apt install openssh-client openssh-server

# Переконатися, що служба SSH запущена:
sudo systemctl enable ssh
sudo systemctl start ssh

# Згенерувати пару ключів (натисніть Enter при запиті пароля, щоб залишити порожнім):
ssh-keygen -t rsa -N "" -f ~/.ssh/id_rsa

# Додати публічний ключ до авторизованих ключів для localhost:
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
chmod 600 ~/.ssh/authorized_keys

# Перевірити SSH до самого себе:
ssh localhost
# (при першому підключенні підтвердьте додавання хоста, після цього підключення має проходити без пароля)
exit  # вихід з сесії SSH
```

Якщо остання команда не вимагає пароль, значить все налаштовано правильно. Важливо: такий самий механізм використовується і при розгортанні на кількох вузлах – ключ користувача координатора має бути в authorized_keys на всіх сегмент-хостах.

2. Створення каталогів даних для кластера:

Потрібно підготувати окремі директорії на файловій системі для даних координатора і сегментів. Зазвичай виділяють різні шляхи (можливо, на різних дисках) для розподілу навантаження. У нашому прикладі використаємо каталог ~/cloudberry_data з підкаталогами:

master – для даних координатора (раніше званого master).

seg1, seg2 – для даних двох сегментів.

Створимо ці каталоги та забезпечимо користувачу повні права доступу до них (якщо ще не):

```
mkdir -p ~/cloudberry_data/master
mkdir -p ~/cloudberry_data/seg1
mkdir -p ~/cloudberry_data/seg2
```

Переконайтеся, що всі три каталоги належать користувачу, під яким працюватиме кластер (команда whoami повинна показувати цього користувача, і ls -ld ~/cloudberry_data/* – що власник каталогів співпадає).

3. Підготовка файлу конфігурації кластера (gpinitsystem_config):

Для ініціалізації кластера в CloudberryDB використовується утиліта gpinitsystem, якій потрібен конфігураційний файл з параметрами кластера. Встановлена система зазвичай містить шаблон цього файлу за шляхом $GPHOME/docs/cli_help/gpconfigs/gpinitsystem_config. Ми можемо скопіювати його і відредагувати. Альтернативно, створимо свій мінімальний варіант. Створіть файл gpinitsystem_config у вашому робочому каталозі (наприклад, в ~/cloudberrydb чи у будь-якому зручному місці) з таким вмістом:

```
# FILE NAME: gpinitsystem_config
# Конфігурація для ініціалізації CloudberryDB (Greenplum) кластера

### ОБОВ'ЯЗКОВІ ПАРАМЕТРИ ###

# Ім'я масиву (кластера) – довільна назва
ARRAY_NAME="Cloudberry Demo Cluster"

# Префікс імені для каталогів сегментів
SEG_PREFIX=gpseg

# Базовий порт для сегментів (первинних)
PORT_BASE=7000
```

```
# Шляхи до каталогів даних сегментів (кількість шляхів = кількість сегментів на хост)
declare -a DATA_DIRECTORY=(/home/user/cloudberry_data/seg1 /home/user/cloudberry_data/seg2)

# Хостнейм координатора
COORDINATOR_HOSTNAME=localhost

# Шлях до каталогу даних координатора
COORDINATOR_DIRECTORY=/home/user/cloudberry_data/master

# Порт для координатора (TCP-порт, на якому він слухатиме підключення)
COORDINATOR_PORT=5432

# Довірений shell для SSH (використовується для розсилки команд)
TRUSTED_SHELL=ssh

# Кодування баз даних за замовчуванням
ENCODING=UNICODE

### ІНШІ ОПЦІЙНІ ПАРАМЕТРИ (необов'язкові для простого кластера) ###
# Можна встановити параметри чекпойнтів, ім’я створюваної БД тощо.
```

Зверніть увагу: у файлі слід замінити /home/user на шлях до домашньої теки вашого користувача (або відповідно вказати інший шлях, якщо ви створювали каталоги не в домашній теці). Параметр COORDINATOR_PORT=5432 виставляє порт 5432 для координатора – переконайтесь, що цей порт зараз вільний. За замовчуванням PostgreSQL використовує 5432, тож якщо на вашій машині запущений інший сервер Postgres, виникне конфлікт. Як перевірити: виконайте sudo lsof -i :5432. Якщо вивід порожній – порт вільний. Інакше див. розділ 6 про зміну порту.

Також створимо файл зі списком хостів (hostfile) для gpinitsystem. У нашому випадку кластер одновузловий, тому просто:

```
echo "localhost" > hostfile_gpinitsystem
```

Файл hostfile_gpinitsystem містить по одному хосту на рядок для всіх сегмент-хостів. Тут лише localhost. (Для багатовузлового кластера ви б перечислили всі імена/адреси машин, які мають запускати сегменти).

4. Ініціалізація кластера командою gpinitsystem:

Переконавшись, що $GPHOME та інші змінні налаштовано (greenplum_path.sh виконано), запускаємо утиліту:

gpinitsystem -c ./gpinitsystem_config -h ./hostfile_gpinitsystem

У процесі gpinitsystem перевірить доступність вузлів через SSH, наявність каталогів, а потім виведе конфігурацію кластера і запитає підтвердження (потрібно буде ввести YES для продовження). Наприклад, ви побачите список сегментів, портів, розміщення і рядок на кшталт: "Continue with Greenplum creation? [yes]/no:" – введіть yes і натисніть Enter. Далі утиліта створить необхідні структури каталогів, згенерує конфігураційні файли PostgreSQL для кожного сегмента та координатора, запустить процеси і завершить ініціалізацію. Якщо все пройшло успішно, наприкінці з'явиться повідомлення:

CloudberryDB instance successfully created.

(Може бути зазначено як "Greenplum Database instance successfully created" – Cloudberry на даний момент сумісна з термінологією Greenplum.) Також утиліта виведе шлях до файлу конфігурації кластера і до скриптів зупинки/видалення кластера на випадок, якщо потрібно його перезапустити чи видалити.

Примітка: Під час ініціалізації можуть з'явитися попередження, наприклад, про Memory Overcommit (налаштування ядра Linux для виділення пам'яті) чи інші. Як правило, на одновузловій тестовій інсталяції ними можна знехтувати. Але для production-середовища варто скоригувати /etc/sysctl.conf відповідно до рекомендацій (наприклад, vm.overcommit_memory=2).

5. Змінні середовища для керування кластером:

Після ініціалізації, бажано встановити змінну COORDINATOR_ DATA_DIRECTORY (або стара назва MASTER_DATA_DIRECTORY) – вона вказує утилітам на розташування даних координатора за замовчуванням, щоб не передавати його в кожній команді. В нашому випадку:

export
COORDINATOR_DATA_DIRECTORY=/home/user/cloudberry_data/master/gpseg-1

Справа в тому, що всередині каталогу координатора master створено підкаталог із префіксом (gpseg-1, якщо префікс gpseg як в конфігурації). Цей шлях і є коренем даних координатора. Додайте цей export також у свій ~/.bashrc поруч із greenplum_path.sh для постійності. Для сегментів змінні не потрібні – їх керування здійснюється через координатор. На цьому установка кластеру завершена.

6. Зміна порту за замовчуванням (при необхідності)

Якщо порт 5432, який ми використали для координатора, був зайнятий (наприклад, на машині вже запущений PostgreSQL), gpinitsystem видасть помилку при спробі запустити координатор. В такому разі потрібно вибрати інший порт. Популярний варіант – 15432 (або будь-який інший вільний порт).

Для зміни порту: відредагуйте файл gpinitsystem_config перед запуском ініціалізації, встановивши, наприклад, COORDINATOR_PORT=15432. Також можна змінити PORT_BASE для сегментів, якщо раптом діапазон 6000+ конфліктує з чимось (малоймовірно). Після цього запустіть gpinitsystem знову.

У вже ініціалізованого кластера змінити порт координатора не так просто – потрібно вручну правити файл postgresql.conf координатора і сегментів і параметри запуску. Тому краще визначити правильні порти на етапі ініціалізації. Якщо ж після ініціалізації ви виявили конфлікт, найпростіше рішення – зупинити кластер (gpstop -a), видалити його (gpdeletesystem або вручну видаливши каталоги даних), виправити gpinitsystem_config і виконати gpinitsystem наново з новим портом.

Перевірка зайнятості порту: скористайтеся командою ss або lsof. Наприклад:

ss -tnlp | grep 5432

Покаже, чи є процес, що слухає 5432. Аналогічно для 15432. Це слід робити до запуску ініціалізації, щоб уникнути помилок.

7. Запуск CloudberryDB, перевірка стану кластера та підключення до бази

Після gpinitsystem кластер вже має працювати: координатор і сегменти запущені. Перевіримо це та навчимось керувати кластером:

Утиліта gpstate: покаже статус кластеру. Виконайте:

gpstate

У відповідь ви повинні побачити інформацію про сегменти. Наприклад, для двосегментного кластера на одному вузлі буде зазначено 2 primary-сегменти Up (працюють) і, якщо немає дзеркал, статус No mirror configured. Також буде виведено ім'я кластера (ARRAY_NAME), версію CloudberryDB і шлях до master data directory.

Підключення до бази даних через psql: Скористаємось клієнтом psql для виконання запитів. Підключимося до координатора:

psql -d postgres

(Опція -d postgres вказує підключитися до дефолтної БД postgres. Також можна явно зазначити хост/порт, напр.: psql -h localhost -p 5432 -

d postgres, але якщо змінні середовища PGPORT і PGHOST не задані, psql спробує localhost і порт 5432 за замовчуванням. Якщо ви використовуєте нестандартний порт, або якщо під час configure було вказано інший порт, встановіть змінну середовища, напр.: export PGPORT=15432 для сесії, щоб не писати -p щоразу.) У разі успіху ви побачите промпт postgres=#. Далі можна виконати тестовий запит, щоб перевірити версію:

SELECT version();

Ви повинні отримати рядок, що містить версію CloudberryDB. Приклад виведення:

```
                                                version
-------------------------------------------------------------------------------------------------
 PostgreSQL 14.4 (Cloudberry Database 1.0.0 build dev) on x86_64-pc-linux-gnu, compiled by gcc, 64-bit
(1 row)
```

Як бачимо, відображається версія PostgreSQL 14.4 та позначка Cloudberry Database з номером збірки. Це підтверджує, що ми підключені саме до CloudberryDB. Щоб побачити конфігурацію сегментів, виконайте запит до системної таблиці:

SELECT content, role, hostname, port, datadir FROM gp_segment_configuration;

Ви побачите записи для кожного сегмента і координатора. Для нашого кластера буде: один запис з content = -1, role = p (координатор), і два записи з content = 0,1 (сегменти) з роллю p (primary), їх хости = localhost, порти ~7000 та каталоги даних відповідно seg1 і seg2. Ця інформація підтверджує структуру кластера.

Зупинка та запуск кластера: Утиліта gpstop зупиняє всі процеси кластера, а gpstart – запускає їх. Для коректної зупинки виконайте:

gpstop -a

Ключ -a (–automatic) означає не запитувати підтвердження. Аналогічно, щоб знову запустити кластер:

gpstart -a

Переконайтеся, що перед запуском ви виконали source greenplum_path.sh та встановили COORDINATOR_DATA_DIRECTORY (інакше gpstart не знайде, що запускати). Після gpstart можна знову використати gpstate для перевірки статусу – всі сегменти мають бути в стані Up.

Важливо: За замовчуванням кластер працює під поточним користувачем і не перезапускається автоматично після перезавантаження системи. Наступний крок – налаштувати Supervisor для автоматичного старту.

8. Налаштування Supervisor для автозапуску CloudberryDB

Щоб при кожному увімкненні/перезавантаженні сервера кластер CloudberryDB запускався автоматично, зручно використати систему супервізора процесів. Тут ми розглянемо використання Supervisor (python-based, зручний для керування користувацькими процесами). Встановлення Supervisor: (Ubuntu)

```
sudo apt install -y supervisor
```

Після встановлення сервіс Supervisor зазвичай вже запущений (sudo systemctl status supervisor щоб перевірити). Конфігураційні файли програм розміщують у каталозі /etc/supervisor/conf.d/. Створимо файл конфігурації, наприклад /etc/supervisor/conf.d/cloudberry.conf з наступним вмістом:

```
[program:cloudberrydb]
# User, від імені якого запускати кластер:
user=user

# Команда для запуску координатора і сегментів (gpstart з автоматичним підтвердженням):
command=/usr/local/cloudberrydb/bin/gpstart -a

# Директорія, з якої запускати (на всякий випадок вкажемо домашню):
directory=/home/user

# Змінні середовища: GPHOME, шлях та змінна з каталогом master
environment=GPHOME="/usr/local/cloudberrydb",\
 PATH="/usr/local/cloudberrydb/bin:/usr/local/bin:/usr/bin:/bin",\

COORDINATOR_DATA_DIRECTORY="/home/user/cloudberry_data/master/gpseg-1"

autostart=true       ; запускати автоматично при старті Supervisor
autorestart=true     ; перезапускати, якщо процес gpstart (координатор) завершиться
stopsignal=INT       ; сигнал для зупинки (INT для коректного завершення PostgreSQL)
stdout_logfile=/var/log/supervisor/cloudberrydb.log
stderr_logfile=/var/log/supervisor/cloudberrydb_err.log
```

Пояснення:

Вказано запуск від користувача user (замінити на вашого користувача або gpadmin).

Команда – gpstart -a. Фактично, Supervisor запустить кластер, коли сам стартує. Варто зазначити, що gpstart після запуску процесів координатора та сегментів завершується (виходить). Але процеси постгресів залишаються працювати у фоні. Supervisor сприйматиме завдання як виконане. Щоб Supervisor не намагався "тримати" процес gpstart, ми не використовуємо режим stopasgroup чи подібні. У разі, якщо координатор завершиться (наприклад, кластер впаде), Supervisor за autorestart=true спробує знову виконати gpstart.

Змінні середовища: ми експортируємо необхідні значення, аналогічно тому, як якщо б ви вручну виконали greenplum_path.sh і встановили COORDINATOR_DATA_DIRECTORY. Важливо вказати коректний шлях до GPHOME та MASTER/COORDINATOR_DATA_DIRECTORY.

Логи: зазначені файли, куди буде записано stdout і stderr. У файлі stdout (cloudberrydb.log) після успішного запуску ви побачите вивід gpstart (список сегментів, повідомлення про успішний запуск). Лог stderr повинен бути порожнім у випадку нормальної роботи (або містити повідомлення про помилки, якщо виникли).

Після створення конфігурації, перерахуйте програми Supervisor:

```
sudo supervisorctl reread    # зчитати нові конфігурації
sudo supervisorctl update    # застосувати їх (запустити нову програму)
```

Перевірте статус:

```
sudo supervisorctl status cloudberrydb
```

Повинно показати RUNNING (і PID процесу, який Supervisor відслідковує – в нашому випадку це буде PID gpstart або, якщо він вже завершився, Supervisor може вважати задачу EXITED але не перезапускати, бо ми не вказали autorestart=unexpected. Основне – наші процеси PostgreSQL повинні працювати). Перевіримо, що кластер дійсно запустився:

```
gpstate -s          # короткий статус кластера
ps -ef | grep postgres | grep cloudberry_data
```

Перша команда (з ключем -s) покаже статус сегментів. Друга – знайде всі процеси Postgres, що працюють з нашого каталогу даних (має показати процес координатора postgres -D .../master/gpseg-1 та два процеси сегментів postgres -D .../seg1 і .../seg2). Якщо вони є – значить автозапуск спрацював.

Налаштування автозапуску Supervisor: На Ubuntu Supervisor запускається при старті системи автоматично (після встановлення він активований як служба systemd). Щоб переконатися, що Supervisor запускатиметься: sudo systemctl enable supervisor.

Таким чином, при наступному перезавантаженні сервера Supervisor підхопить нашу програму cloudberrydb і виконає gpstart. Ви можете протестувати, перезавантаживши систему, або зупинивши вручну кластер і давши Supervisor-у перезапустити його:

```
gpstop -a                    # зупинити кластер вручну
sudo supervisorctl restart cloudberrydb  # перезапустити процес через Supervisor
```

Після цього кластер знову має запуститися. В файлі /var/log/supervisor/cloudberrydb.log можна побачити лог запуску. Крім того, Supervisor логуватиме власні повідомлення в /var/log/supervisor/supervisord.log.

Завдання

1. Виконайте всі кроки, щоб зібрати CloudberryDB з вихідного коду і створити локальний кластер з двома сегментами. За результатами роботи утиліти gpinitsystem надайте фрагмент виводу, який підтверджує успішне створення кластера (рядок про successfully created).
2. Підключіться до розгорнутої бази даних за допомогою psql (використовуйте користувача за замовчуванням, створеного під час ініціалізації, зазвичай це gpadmin без пароля, або локальний trust). Виконайте команду SELECT version(); і наведіть у звіті її результат (версія повинна відображати CloudberryDB). Також виконайте SELECT * FROM gp_segment_configuration; і вкажіть, скільки сегментів показано в конфігурації і їх статус.
3. Налаштуйте Supervisor згідно інструкції вище. Перезавантажте віртуальну машину або сервер, на якому розгорнуто кластер (або зупиніть кластер і запустіть через Supervisor). Переконайтеся, що після перезавантаження процеси CloudberryDB запустилися автоматично: перевірте статус supervisorctl і виконайте запит SELECT 1; до бази через psql. Опишіть, як ви перевірили автозапуск і додайте витяги команд/виводу, які це підтверджують.

Практична робота №8. «Побудова повноцінної платформи даних за допомогою HDFS, Spark та Cloudberry»

Завдання

Застосувати отримані навички та побудувати повноцінну Data Platform з нуля.

Вимоги

Джерела

- API
  - out of stock

● Дані з таблиць PostgreSQL
○ orders
○ products
○ departments
○ aisles
○ clients
○ stores
○ store_types
○ location_areas

Складові компоненти програми
● Джерела даних:
○ PostgreSQL
○ API
● Data Lake:
○ Apache Hadoop HDFS
● Сховище даних:
○ Cloudberry
● ETL:
○ Apache Spark
● Scheduler & ETL Management:
○ Apache Airflow

ETL
Extract:
● Для кожного джерела має бути створений окремий DAG в Airflow
● Кожна таблиця PostgreSQL повинна оброблятись в окремому Airflow операторі
всередині одного DAG;
● Дані повинні зберігатися у Bronze Layer Data Lake
Transform:
● Окремі Airflow DAG, які повинні провалювати дані в Bronze. Провести перевірку даних та перенести їх у Silver
● Дані Silver повинні бути організовані
Load:
● Завантажити дані із Silver прямо в Сховище Даних, без використання проміжного сховища

ETL потрібно проектувати з урахуванням того, що джоби будуть виконуватися щодня в заздалегідь заданий час. Всі credentials та ін. інформацію необхідно зберігати в Airflow Connections.

DWH
● Сховище даних має бути реалізовано у вигляді схеми Зірки;
● Повинно бути, як мінімум, дві таблиці фактів.

Горбань Гліб Валентинович

Методична серія
Випуск № 491

**ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ
ОБРОБКИ ВЕЛИКИХ ОБСЯГІВ ДАНИХ**

Методичні рекомендації до виконання практичних робіт

Редактор Карпова К.О.
Комп'ютерна верстка Олисько Я.Є.
Друк Волинець С.Л. Фальцювально-палітурні роботи Мішалкіна О.Є.

Підп. до друку 24.09.2026.
Формат 60x84 $^{1}/_{16}$. Папір офсет.
Гарнітура «Times New Roman». Друк різограф.
Ум. друк. арк. 6,5. Обл.-вид. арк. 3,3.
Тираж 10 пр. Зам. № 7275
.
Видавець і виготовлювач: ЧНУ ім. Петра Могили.
54003, м. Миколаїв, вул. 68 Десантників, 10.
Тел.: 8 (0512) 50–03–32, 8 (0512) 76–55–81, e-mail:rector@chmnu.edu.ua.
Свідоцтво суб'єкта видавничої справи ДК № 6124 від 05.04.2018.