МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Тернопільський національний технічний університет
імені Івана Пулюя

Кафедра програмної інженерії

Конспект лекцій

з дисципліни

«Розподілені та паралельні системи обчислень»

СЕН ID: **6843**

для здобувачів третього (освітньо-наукового) рівня вищої освіти
ОНП «Інженерія програмного забезпечення»

Тернопіль — 2025

УДК 004.4(075.8)
ББК 32.973.202я73
Б83

Укладач:
В.М. Бревус, PhD

Рецензент:
В.В. Яцишин, к.т.н., доцент
завідувач кафедри систем штучного інтелекту та аналізу даних ТНТУ ім. І. Пулюя

Розглянуто й затверджено на засіданні кафедри програмної інженерії.
Протокол № 16 від 29.05.2025.

Розглянуто й затверджено на засіданні методичної комісії факультету комп'ютерно-інформаційних систем та програмної інженерії Тернопільського національного технічного університету імені Івана Пулюя.
Протокол № 5 від 05.06.2025.

Б83 Конспект лекцій з дисципліни «Розподілені та паралельні системи обчислень» для здобувачів третього (освітньо-наукового) рівня вищої освіти ОНП «Інженерія програмного забезпечення» / Укладач: Бревус В.М. – Тернопіль: ТНТУ імені Івана Пулюя, 2025 – 135 с.

Відповідальний за випуск: М.Р. Петрик, докт. фіз.-мат. наук, професор

АНОТАЦІЯ

У конспекті лекцій систематизовано сучасні підходи до розподіленої обробки великих даних з використанням екосистеми Apache Spark. Розглянуто фундаментальні принципи розподілених систем, архітектурні моделі та теорему CAP. Детально описано роботу з основними абстракціями даних RDD та DataFrame, використання Spark SQL для виконання запитів та трансформації даних. Висвітлено методи інженерії даних, включаючи очищення, підготовку та об'єднання наборів даних. Особливу увагу приділено машинному навчанню з використанням бібліотеки MLlib: побудові конвеєрів (pipelines), застосуванню алгоритмів регресії та класифікації, оцінюванню якості моделей. Окремо розглянуто питання налагодження, профілювання та оптимізації продуктивності розподілених обчислень, а також стратегії розгортання Spark-застосунків у кластерних середовищах (Standalone, YARN, Kubernetes). Матеріал орієнтований на здобувачів третього (освітньо-наукового) рівня за ОНП «Інженерія програмного забезпечення» та підтримує формування компетентностей у проєктуванні та реалізації масштабованих систем аналізу даних.

Ключові слова: розподілені системи, Apache Spark, PySpark, Big Data, машинне навчання, MLlib, RDD, DataFrame, оптимізація, розгортання.

ABSTRACT

The lecture notes systematize modern approaches to distributed big data processing using the Apache Spark ecosystem. Fundamental principles of distributed systems, architectural models, and the CAP theorem are examined. Work with core data abstractions RDD and DataFrame, as well as using Spark SQL for queries and data transformation, is described in detail. Data engineering methods, including cleaning, preparation, and merging of datasets, are covered. Special attention is paid to machine learning using the MLlib library: building pipelines, applying regression and classification algorithms, and evaluating model quality. Issues of debugging, profiling, and optimizing the performance of distributed computing, as well as strategies for deploying Spark applications in cluster environments (Standalone, YARN, Kubernetes), are separately considered. The material targets third-cycle (educational and scientific level) PhD candidates in the "Software Engineering" program and supports competence development in designing and implementing scalable data analysis systems.

Keywords: distributed systems, Apache Spark, PySpark, Big Data, machine learning, MLlib, RDD, DataFrame, optimization, deployment.

Зміст

Вступ

Конспект лекцій з дисципліни «Розподілені та паралельні системи обчислень» присвячений вивченню сучасних підходів до обробки великих даних у розподілених середовищах. Основний акцент зроблено на використанні фреймворку Apache Spark, який став де-факто стандартом для швидкої та масштабованої обробки даних.

Курс розпочинається з розгляду фундаментальних принципів розподілених систем, включаючи архітектурні моделі, теорему CAP та моделі узгодженості даних. Це створює теоретичну базу для розуміння того, як функціонують сучасні розподілені платформи.

Значна частина матеріалу присвячена архітектурі та екосистемі Apache Spark. Детально розглядаються абстракції даних, такі як RDD (Resilient Distributed Datasets) та DataFrame, а також принципи лінивих обчислень (lazy evaluation). Слухачі ознайомляться з інструментами Spark SQL для виконання запитів, трансформації даних та використання віконних функцій.

Окрему увагу приділено інженерії даних та підготовці їх до аналізу. Розглядаються методи очищення даних, робота з різними типами даних та об'єднання наборів. Ці навички є критично важливими для побудови надійних конвеєрів обробки даних (pipelines).

У рамках курсу також вивчаються можливості бібліотеки MLlib для машинного навчання. Студенти навчаться будувати конвеєри машинного навчання, застосовувати алгоритми регресії та класифікації, а також оцінювати якість побудованих моделей.

Важливим аспектом підготовки фахівців є вміння оптимізувати та розгортати розподілені застосунки. Курс охоплює методи моніторингу, профілювання та оптимізації продуктивності Spark-застосунків. Також розглядаються різні режими розгортання, включаючи Standalone, YARN та Kubernetes, а також найкращі практики конфігурації кластерів.

Протягом курсу ми розглянемо:

- Фундаментальні принципи та архітектурні моделі розподілених систем.
- Архітектуру Apache Spark та роботу з RDD і DataFrame.
- Використання Spark SQL для аналізу та трансформації даних.
- Методи інженерії ознак та очищення даних.
- Побудову моделей машинного навчання з використанням MLlib.
- Техніки налагодження, профілювання та оптимізації розподілених обчислень.
- Стратегії розгортання Spark-застосунків у кластерних середовищах.

Метою курсу є формування у здобувачів PhD професійних компетентностей для проектування, реалізації та оптимізації високонавантажених розподілених систем обробки даних.

Слухачі навчаться:

- Розуміти теоретичні основи розподілених обчислень та узгодженості даних.
- Ефективно використовувати Apache Spark для обробки великих масивів даних.
- Проектувати та реалізовувати конвеєри підготовки даних та машинного навчання.
- Виконувати оптимізацію продуктивності та виявляти «вузькі місця» у розподілених застосунках.
- Розгортати та адмініструвати Spark-застосунки у сучасних кластерних середовищах, зокрема Kubernetes.

Курс поєднує теоретичні знання з практичними навичками роботи з PySpark, що дозволяє слухачам вирішувати реальні завдання аналізу даних та машинного навчання у розподілених системах.

Лекція 1

Фундаментальні принципи розподілених систем

Архітектурні моделі, теорема CAP, моделі узгодженості.

Предмет, мета та завдання курсу

Предмет: Методи, моделі та технології для розробки програмного забезпечення, що ефективно використовує багатопроцесорні та розподілені архітектури.

Мета:

- Сформувати розуміння принципів паралелізму.
- Ознайомити з сучасними інструментами (MPI, OpenMP, CUDA).
- Навчити аналізувати та оптимізувати паралельні програми.

Завдання:

- Вивчити класифікацію архітектур.
- Розглянути моделі паралельних обчислень.
- Проаналізувати основні виклики (синхронізація, комунікація).
- Розробити та протестувати паралельні алгоритми.

Навчальні результати

Після завершення лекції студент зможе:

1. Пояснити різницю між паралелізмом та розподіленістю.
2. Класифікувати архітектури за таксономією Флінна.
3. Порівняти моделі PRAM та BSP за припущеннями і обмеженнями.
4. Обчислити прискорення та ефективність S, E для паралельного алгоритму.
5. Оцінити компроміс CAP та вибір моделі узгодженості.
6. Аргументувати вибір архітектурної моделі для заданого сценарію.

Мотиваційний приклад. Аналіз журналів запитів веб-сервісу може виконуватися паралельно (розбиття великого файлу на блоки для локальної обробки) або розподілено (координований збір даних з географічно віддалених дата-центрів). Розуміння різниці впливає на вибір інструментів.

Історичний огляд

Розвиток паралельних та розподілених систем пройшов через кілька критичних етапів:

- **1950–1960-ті рр.** Перші багатопроцесорні системи (CDC 6600).
- **1970-ті рр.** Векторні суперкомп'ютери (Cray-1), розвиток SIMD.
- **1980-ті рр.** Масово-паралельні системи (MPP), поява MPI.
- **1990-ті рр.** Кластерні системи на базі комерційних компонентів (Beowulf).
- **2000-ні рр.** Багатоядерні процесори стають стандартом, розвиток GPGPU.
- **2010-ті рр.** – наш час: Хмарні обчислення, гетерогенні системи, екзафлопсні суперкомп'ютери.

Ключові поняття

Паралелізм Одночасне виконання кількох задач або частин однієї задачі для прискорення обчислень.

Розподіленість Система, в якій компоненти розташовані на різних вузлах, що з'єднані мережею, і координують свої дії шляхом обміну повідомленнями.

Масштабованість Здатність системи збільшувати продуктивність під час додавання обчислювальних ресурсів.

Продуктивність Кількість роботи, виконаної за одиницю часу (напр., у FLOPS).

Класифікація та моделі

Класифікація паралельних архітектур (таксономія Флінна)

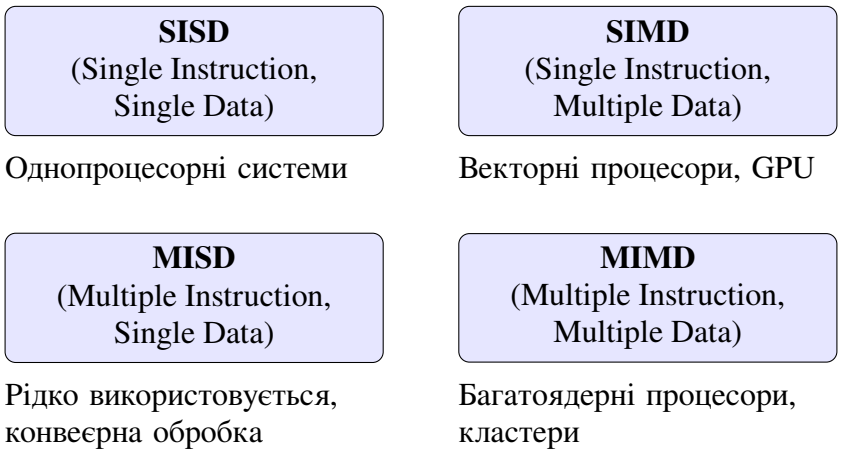

Рис. 1.1: Таксономія Флінна для класифікації паралельних архітектур

SISD (Single Instruction, Single Data) – однопроцесорні системи.

SIMD (Single Instruction, Multiple Data) – векторні процесори, GPU.

MISD (Multiple Instruction, Single Data) – майже не застосовується на практиці як загальноцільова архітектура; конвеєр інструкцій не є чистим прикладом MISD.

MIMD (Multiple Instruction, Multiple Data) – багатоядерні процесори, кластери.

Класифікація Флінна (рис. 1.1) залишається фундаментальною для розуміння паралельних архітектур, хоча сучасні системи часто поєднують кілька підходів. Кожен клас має свої особливості та області застосування:

- **SISD** архітектури домінували до появи багатоядерних процесорів
- **SIMD** стали основою для графічних процесорів (GPU) та векторних розширень
- **MISD** рідко використовується на практиці, за винятком деяких спеціалізованих систем
- **MIMD** є основою сучасних багатоядерних процесорів та розподілених систем

Моделі паралельних обчислень

PRAM (Parallel Random Access Machine)

Це абстрактна модель для теоретичного аналізу паралельних алгоритмів.

Основні характеристики:

- Система складається з N однакових процесорів.
- Всі процесори мають доступ до спільної глобальної пам'яті з однаковим часом доступу.
- На кожному кроці кожен процесор може виконати одну операцію.
- Синхронна обробка – всі процесори рухаються синхронно через кроки.

Різновиди за правилами конфліктів під час одночасного доступу:

EREW (Exclusive Read, Exclusive Write) – найбільш обмежувальна: лише один процесор може читати/писати в комірку пам'яті одночасно.

CREW (Concurrent Read, Exclusive Write) – кілька процесорів можуть одночасно читати, але лише один писати.

ERCW (Exclusive Read, Concurrent Write) – рідко використовується.

CRCW (Concurrent Read, Concurrent Write) – найменш обмежувальна, але потребує правил розв'язання конфліктів під час запису.

Переваги:

- Простота аналізу та розуміння паралельних алгоритмів.
- Дозволяє сконцентруватися на логіці алгоритму, не турбуючись про деталі реалізації.

Недоліки:

- Повністю ігнорує вартість комунікацій між процесорами.
- Припускає, що всі операції пам'яті коштують однаково, що далеко від реальності.
- Не враховує синхронізацію та накладні витрати.
- Часто прогнози часу виконання на основі PRAM далекі від реальних результатів.

BSP (Bulk Synchronous Parallel)

Це реалістичніша модель, розроблена для ближчого відображення реальних паралельних систем.

Основна ідея: Програма виконується як послідовність *суперкроків*, кожен з яких складається з трьох фаз:

1. **Обчислювальна фаза** – кожен процесор виконує локальні обчислення без комунікацій.
2. **Комунікаційна фаза** – процесори обмінюються даними через мережу.
3. **Синхронізаційна бар'єр** – все процесори чекають, поки інші завершать комунікацію. Лише після цього починається наступний суперкрок.

Параметри системи:

l Синхронна затримка – час синхронізації бар'єру.

g Коефіцієнт бісекції – відношення кількості операцій обчислення до кількості слів, які можна передати за один крок.

Час виконання суперкроку:

$$T = W + h \cdot g + l \tag{1.1}$$

де W – обчислювальна робота, h – максимальна дальність доставки повідомлень.

Переваги:

- Враховує вартість комунікацій та синхронізації, які часто домінують у паралельних системах.
- Дозволяє реалістичніше прогнозувати час виконання.
- Проста модель для аналізу – не потребує детального моделювання мережі.

Недоліки:

- Синхронна синхронізація може бути неефективною для гетерогенних систем.
- Не враховує неоднорідність затримок мережі.

Архітектурні моделі розподілених систем

Клієнт-серверна архітектура

У цій моделі централізований сервер надає ресурси та послуги множині клієнтів. Клієнти надсилають запити серверу та отримують результати.

Приклади: Веб-сервери, бази даних, системи електронної пошти.

Переваги:

- Простота управління та контролю ресурсів.
- Централізована безпека та аутентифікація.
- Простіше забезпечити узгодженість даних.

Недоліки:

- Єдина точка відмови (сервер).
- Вузьке місце – сервер може стати розподільником навантаження.
- Масштабування вимагає збільшення потужності сервера.

Peer-to-Peer (P2P) архітектура

У цій моделі всі вузли (піри) є рівноправними. Кожен вузол може виступати як клієнтом, так і сервером, надаючи ресурси іншим вузлам.

Приклади: BitTorrent, блокчейн-системи, системи спільного доступу до файлів (Napster, Gnutella).

Переваги:

- Висока масштабованість – додаванням нових вузлів система зростає.
- Відмовостійкість – немає єдиної точки відмови.
- Розподілення навантаження – кожен вузол бере участь у роботі.

Недоліки:

- Складність координації та синхронізації вузлів.
- Складніше забезпечити безпеку та аутентифікацію.
- Проблеми з узгодженістю даних.
- Важко шукати ресурси та управляти ними глобально.

Багаторівнева архітектура (N-tier)

Логіка системи розділена на кілька логічних рівнів [1], [2]:

- **Презентаційний рівень** – користувацький інтерфейс (веб-браузер, мобільний додаток).
- **Рівень бізнес-логіки** – обробка запитів, виконання бізнес-правил.
- **Рівень даних** – збереження та керування даними.

Типовим прикладом є класична 3-рівнева веб-програма (web server + application server + database server).

Переваги:

- Гнучкість – кожен рівень можна розвивати та тестувати незалежно.
- Масштабованість окремих рівнів – можна масштабувати бізнес-логіку незалежно від бази даних.
- Перевикористання компонентів.

Недоліки:

- Збільшена складність системи.
- Затримки через обмін даними між рівнями.
- Важче налагоджувати та тестувати.

Гібридна архітектура

Поєднує елементи клієнт-серверної та P2P моделей, використовуючи сильні сторони кожної.

Приклад: Skype – централізована автентифікація та управління (серверна архітектура), але P2P для безпосередніх дзвінків між користувачами.

Переваги:

- Поєднує масштабованість P2P з контролем клієнт-серверної моделі.
- Гнучкість у розподілі функцій.

Недоліки:

- Складність проектування та реалізації.
- Важче передбачити поведінку системи.

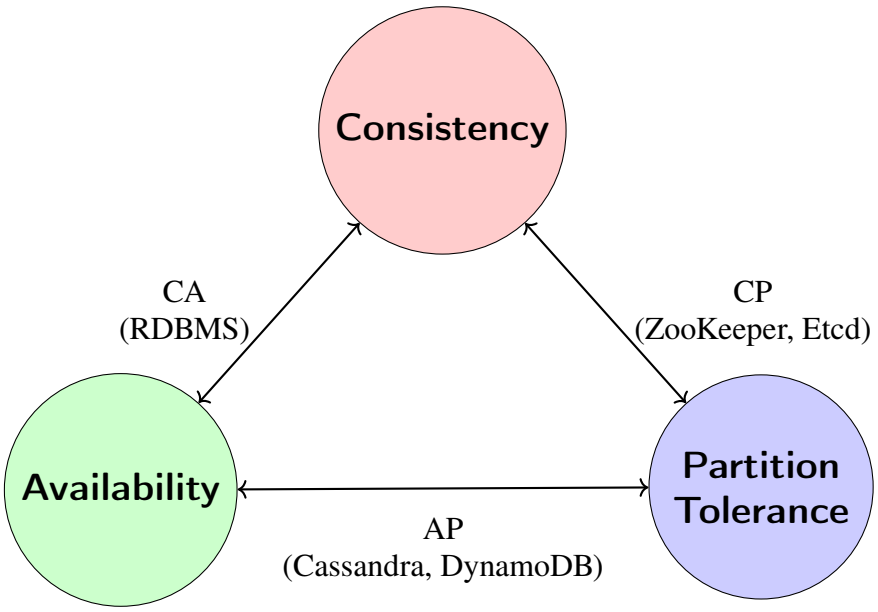

Рис. 1.2: Теорема CAP: вибір між узгодженістю, доступністю та стійкістю до розділення

Теорема CAP та моделі узгодженості

Теорема CAP (Теорема Брюера)

Розподілена система, що зберігає дані, може гарантувати лише дві з трьох властивостей одночасно [3], [4], [5]:

Consistency (Узгодженість) Усі вузли системи бачать однакові дані в один і той же час. Після успішного запису дані однозначно відбиваються на всіх репліках.

Availability (Доступність) Кожен запит отримує відповідь від системи (не обов'язково з актуальними даними), без гарантії того, що це найсвіжіші дані.

Partition Tolerance (Стійкість до розділення) Система продовжує працювати, навіть якщо виникають розриви мережі та деякі вузли не можуть спілкуватися з іншими.

Як показано на рис. 1.2, вибір між цими властивостями визначає архітектуру та поведінку розподіленої системи.

В умовах розподілених систем, де розриви мережі неминучі, стійкість до розділення (P) є обов'язковою властивістю. Тому на практиці вибір зводиться до компромісу між узгодженістю (C) та доступністю (A) (рис. 1.2):

CA системи Гарантують узгодженість і доступність, але не розраховані на розриви мережі. **Приклади:** Традиційні реляційні СУБД (Oracle, PostgreSQL) з однією точкою запису.

AP системи Гарантують доступність і стійкість до розділення, але можуть видавати застарілі дані. **Приклади:** Cassandra, DynamoDB, систем кешування.

CP системи Гарантують узгодженість і стійкість до розділення, але під час розривів мережі частина системи може бути недоступна. **Приклади:** ZooKeeper, Etcd, системи з консенсусом (Raft [6], Paxos [7]). (MongoDB та Redis мають режими, що не завжди однозначно класифікуються.)

Моделі узгодженості даних

Узгодженість даних описує, в якому порядку читачі спостерігають оновлення даних у розподілених системах.

Сильна узгодженість (Strong Consistency)

Лінеаризованість (Linearizability): (формалізовано у [8])

Найсильніша модель узгодженості. Усі операції з'являються так, ніби вони виконані миттєво (атомарно) в певний момент часу в глобальному порядку. Цей порядок узгоджується з реальним часом виконання операцій.

- Після завершення операції запису всі наступні читання відбивають це оновлення.
- Гарантує, що система поводиться як єдиний процесор з глобальною пам'яттю.
- **Застосування:** Системи, де критична точність даних (фінансові транзакції, керування запасами).

Послідовна узгодженість (Sequential Consistency):

Усі процеси бачать операції в однаковому порядку, проте цей порядок може не повністю відповідати реальному часу виконання.

- Слабша від лінеаризованості, оскільки дозволяє переупорядкування операцій, якщо це не порушує послідовність окремих процесів.
- Простіша для реалізації, ніж лінеаризованість.
- **Застосування:** Системи паралельного програмування з глобальною пам'яттю (OpenMP, Pthreads).

Слабка узгодженість (Weak Consistency)

Причинна узгодженість (Causal Consistency):

Зберігається порядок лише для операцій, які мають причинно-наслідкові відносини. Операції, які не пов'язані причинно, можуть спостерігатися в різних порядках на різних вузлах.

- Якщо операція A міг потенційно вплинути на операцію B (наприклад, A завершилась раніше, ніж B почалася), то все вузли повинні спостерігати цей порядок.
- Дозволяє більше паралелізму, ніж послідовна узгодженість.
- **Застосування:** Системи обміну повідомленнями, соціальні мережі.

Узгодженість в кінцевому рахунку (Eventual Consistency):

Якщо оновлення даних припиняються, то з часом усі репліки будуть мати однакове значення [9].

- Дозволяє найбільше паралелізму, оскільки не вимагає синхронізації між оновленнями.
- На певний період користувачі можуть спостерігати різні версії даних.
- **Ризики:** Користувач може прочитати застарілі дані; конфліктуючі оновлення потребують стратегії розв'язання.
- **Застосування:** Системи, де доступність та швидкість критичні (DNS, системи кешування, соціальні мережі).

Ієрархія моделей узгодженості

Моделі узгодженості можна упорядкувати за силою гарантій, які вони надають:

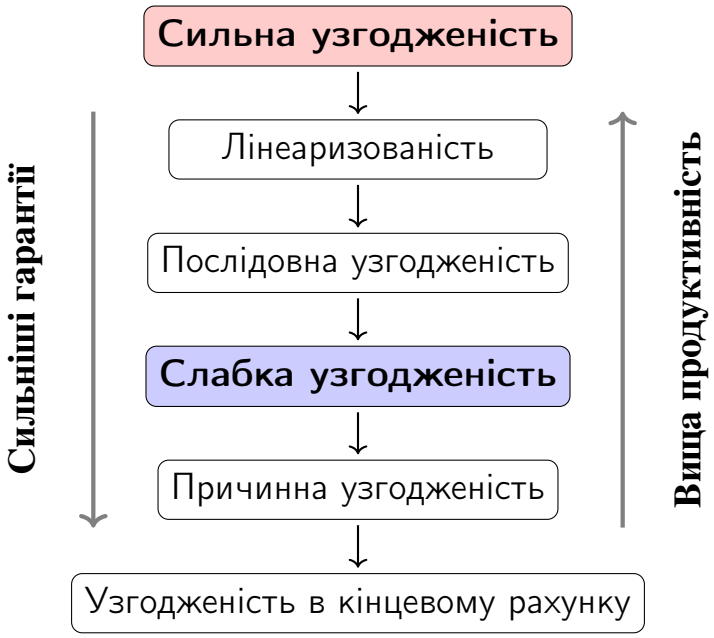

Рис. 1.3: Ієрархія моделей узгодженості: компроміс між гарантіями та продуктивністю

1. **Лінеаризованість** (найсильніша) – повна синхронізація з реальним часом.
2. **Послідовна узгодженість** – одна спільна послідовність для всіх, без привязки до часу.
3. **Причинна узгодженість** – порядок лише для пов'язаних операцій.
4. **Узгодженість в кінцевому рахунку** (найслабка) – гарантія лише в довгостроковій перспективі.

Як показано на рис. 1.3, сильніші гарантії забезпечують більшу коректність, але зменшують продуктивність і масштабованість. Вибір моделі залежить від вимог застосунку.

Проміжні та практичні моделі узгодженості

Реальні системи часто впроваджують проміжні гарантії між причинною та остаточною узгодженістю:

Read-Your-Writes Клієнт після запису бачить власні оновлення.

Monotonic Reads Новіші прочитані дані не “відкотяться” до старіших.

Monotonic Writes Записи одного клієнта застосовуються в глобальному порядку відправлення.

Session Consistency Гарантії діють у межах сеансу; поза ним можуть послаблюватися.

Ці моделі забезпечують баланс між продуктивністю та передбачуваністю для користувача.

Теоретичні межі. Результат FLP [10] доводить неможливість детермінованого консенсусу в повністю асинхронній системі з хоча б однією можливою відмовою — практичні алгоритми (Raft [6], Paxos [7]) покладаються на таймаути.

Метрики продуктивності паралельних програм

Прискорення

$$S = \frac{T_1}{T_N} \tag{1.2}$$

Ефективність

$$E = \frac{S}{N} \tag{1.3}$$

Вартість

$$C = T_N \cdot N \tag{1.4}$$

Масштабованість Збереження або покращення ефективності при збільшенні N.

Закон Амдала [11]:

$$S \leq \frac{1}{(1-p) + \frac{p}{N}} \tag{1.5}$$

Закон Густафсона [12]:

$$S(N) = (1-p) + pN \tag{1.6}$$

Комунікаційні витрати.

$$T_{msg} = t_{latency} + \frac{size}{bandwidth} \tag{1.7}$$

використовується для наближеної оцінки затримки передачі.

Типові пастки. False sharing, NUMA ефекти, незбалансованість навантаження, надмірна конкуренція за блокування.

Основні виклики

Розробка розподілених та паралельних систем стикається з фундаментальними викликами, які визначають складність проектування та реалізації:

Синхронізація

Проблема: Забезпечення правильного порядку виконання операцій та уникнення конфліктів під час одночасного доступу до спільних ресурсів.

Механізми синхронізації:

- **М'ютекси (Mutexes):** Забезпечують взаємовиключне використання критичних секцій.
- **Семафори:** Дозволяють керувати доступом до обмеженої кількості ресурсів.
- **Бар'єри (Barriers):** Синхронізують групу потоків у певній точці.
- **Умовні змінні:** Дозволяють потокам очікувати певні умови.

Типові проблеми:

- **Race Conditions (Стан гонитви):** Коли результат залежить від неконтрольованого порядку виконання операцій.
- **Deadlocks (Взаємні блокування):** Коли два чи більше потоків очікують один на одного, утворюючи циклічну залежність.
- **Livelock:** Коли процеси змінюють стан, але не роблять прогресу.

- **Starvation (Голодування):** Коли деякі потоки нескінченно довго не отримують доступу до ресурсу.

Комунікація

Проблема: Обмін даними між процесами/потоками в розподілених системах.

Моделі комунікації:

- **Спільна пам'ять:** Процеси/потоки у одній машині мають доступ до однієї адресної пам'яті. Прості для програмування, але важко масштабуються.
- **Передача повідомлень:** Процеси обмінюються явними повідомленнями через мережу. Складніше програмувати, але масштабується краще.

Характеристики комунікації:

Latency (Затримка) Час, який потрібен для доставки одного повідомлення від джерела до пункту призначення. В локальних мережах – мікросекунди, в глобальних – мілісекунди.

Bandwidth (Пропускна здатність) Кількість даних, яку можна передати за одиницю часу (наприклад, Gbps). Визначає максимальну швидкість передачі великих обсягів даних.

Вплив на проєктування:

- Затримки вимагають асинхронного програмування та пайплайнінгу запитів [13].
- Обмежена пропускна здатність вимагає ефективної упаковки даних та мінімізації обміну.

Відмовостійкість

Проблема: Система повинна продовжувати працювати коректно навіть у разі відмови окремих компонентів.

Типи відмов:

- **Crash Failures:** Компонент припиняє роботу та більше не відновлюється.

- **Arbitrary (Byzantine) Failures:** Компонент поводиться непередбачувано, може видавати неправильні результати.
- **Timeout Failures:** Компонент не відповідає протягом очікуваного часу.

Стратегії забезпечення відмовостійкості:

Реплікація Збереження копій даних на кількох вузлах. Якщо один вузол відмовляє, дані можна отримати з іншого.

Контрольні точки Періодичне збереження стану системи. У разі відмови система повертається до останньої контрольної точки.

Консенсус Алгоритми (Raft [6], Paxos [7]), які дозволяють розподіленій системі дійти згоди навіть у разі відмови деяких вузлів.

Основні виклики:

- **Виявлення відмов:** Як визначити, чи компонент дійсно відмовив або просто повільно реагує?
- **Відновлення стану:** Як відновити всю систему до консистентного стану після відмови?
- **Вартість реплікації:** Утримання кількох копій збільшує використання ресурсів та складність синхронізації.

Висновки

Основні висновки цієї лекції:

- Паралельні та розподілені системи є основною технологією обчислень, від суперкомп'ютерів до хмарних сервісів [14].
- Розуміння ключових концепцій (паралелізм, розподіленість, масштабованість) та архітектурних моделей є критичним для розробки ефективного програмного забезпечення [15].
- Теорема CAP показує фундаментальний компроміс у розподілених системах: під час розривів мережі неможливо одночасно гарантувати узгодженість та доступність.

- Вибір моделі узгодженості – критичне рішення, яке впливає на коректність, продуктивність та складність реалізації. Сильніші гарантії коштують дорожче в сенсі затримок і пропускної здатності. основні виклики – синхронізація, комунікація та відмовостійкість – вимагають ретельного проектування та ретельного аналізу [16].
- Вибір правильної архітектури (клієнт-серверна, P2P, багаторівнева) залежить від конкретної задачі, вимог до масштабованості, надійності та управління.
- На практиці розробник повинен розуміти компроміси кожного рішення і адаптувати підхід до вимог застосунку.

> *All models are wrong, but some are useful.*
> Box, G.E.P., & Draper, N.R. (1987)

Контрольні запитання

1. Які ключові результати навчання цього курсу та як вони пов'язані з практичними інструментами паралельних обчислень?
2. Чим відрізняються поняття «паралелізм» та «розподіленість» у контексті систем?
3. Які основні етапи історичного розвитку паралельних та розподілених систем і які технологічні зрушення вони принесли?
4. Поясніть різницю між продуктивністю та масштабованістю. Чому висока продуктивність не завжди означає хорошу масштабованість?
5. Наведіть приклади сучасних систем для кожного класу таксономії Флінна (SISD, SIMD, MISD, MIMD).
6. У чому абстрактність моделі PRAM і чому вона недостатня для точного прогнозування часу виконання програм?
7. Опишіть суперкрок у моделі BSP. Які витрати враховує ця модель?

8. Порівняйте архітектурні моделі: клієнт-серверна, P2P, багаторівнева та гібридна – за перевагами й недоліками.

9. Чому теорема CAP робить неможливою одночасну гарантію C, A і P? Який практичний вплив це має на проєктування систем?

10. Наведіть приклади реальних систем (СУБД/платформ) для комбінацій CA, CP, AP та аргументуйте вибір.

11. Чим лінеаризованість відрізняється від послідовної узгодженості? Наведіть сценарій, де ці моделі дають різні результати спостереження.

12. Що таке причинна узгодженість? Як визначити, що дві операції причинно пов'язані?

13. За яких умов досягається узгодженість в кінцевому рахунку? Які ризики для користувача під час періоду узгодження?

14. Розкрийте основні джерела затримок (*latency*) і фактори пропускної здатності (*bandwidth*) у розподілених системах.

15. Які механізми синхронізації запобігають станам гонитви? Чому взаємні блокування виникають і як їх уникати на рівні проєктування?

16. Порівняйте підходи «спільна пам'ять» та «обмін повідомленнями» щодо складності програмування і масштабованості.

17. Які стратегії забезпечують відмовостійкість? Поясніть роль реплікації та контрольних точок.

18. Як визначити компроміс між узгодженістю та доступністю для конкретного застосунку (наприклад, фінансові транзакції vs. соціальні мережі)?

19. Чому гетерогенність апаратних ресурсів (CPU + GPU) стала ключовою тенденцією? Які виклики для розробника це створює?

20. Запропонуйте критерії оцінки ефективності паралельного алгоритму (окрім швидкості виконання).

Лекція 2

Вступ до Apache Spark та PySpark

Архітектура Spark, RDD, ліниві обчислення.

Навчальні цілі

Після опрацювання цієї лекції студент повинен уміти:

- Пояснити архітектуру Spark (Driver, Executors, Cluster Manager) та життєвий цикл Job → Stages → Tasks.
- Відрізнити RDD, DataFrame та Dataset і обрати відповідну абстракцію для задачі.
- Описати принципи lazy evaluation, lineage та відмовостійкість.
- Налаштувати та застосувати Structured Streaming (watermark, output modes) для простого потоку.
- Аргументувати вибір формату зберігання (Parquet vs Delta vs Iceberg vs Hudi) залежно від вимог.
- Застосувати базові оптимізації (broadcast join, reduceByKey vs groupByKey, AQE) та діагностувати skew.
- Інтерпретувати Spark UI метрики (tasks, shuffle, storage) для виявлення проблем продуктивності.

Від теорії розподілених систем до практики Big Data

У попередній лекції ми розглянули фундаментальні принципи побудови розподілених систем, включаючи таксономію Флінна, моделі узгодженості та теорему CAP. Apache Spark є яскравим прикладом еволюції цих концепцій у практичний інструмент для обробки великих даних. Він є вирішенням реальних проблем, які виникають під час проєктування та реалізації масштабованих обчислювальних систем.

Розвиток паралельних та розподілених систем пройшов через кілька критичних етапів: від перших багатопроцесорних систем (CDC 6600 у 1950–1960-х рр.) до векторних суперкомп'ютерів (Cray-1 у 1970-х рр.), масово-паралельних систем (MPP) та кластерних систем на базі комерційних компонентів (Beowulf) у 1990-х рр., і нарешті до сучасних хмарних обчислень та гетерогенних систем. Apache Spark, розроблений у 2009 році в AMPLab Каліфорнійського університету в Берклі, втілює накопичені знання та досвід у інструмент, котрий робить розподілені обчислення доступними та практичними.

Як Spark реалізує концепції розподілених систем:

- **Реалізація моделі MIMD:** Spark працює на кластерах, де безліч вузлів обробляють різні частини даних паралельно, реалізуючи стратегію *Multiple Instruction, Multiple Data*.

- **Розвиток ідей BSP:** Виконання завдань розбивається на стадії (Stages), розділені бар'єрами синхронізації (Shuffle), що відповідає моделі Bulk Synchronous Parallel.

- **Відмовостійкість як альтернатива реплікації:** Замість повної реплікації стану, Spark використовує механізм Lineage (родовід трансформацій) для переобчислення втрачених партицій даних, що забезпечує Partition Tolerance з мінімальними накладними витратами.

- **Абстракція складності:** Spark приховує низькорівневі деталі комунікації та синхронізації, розглянуті у лекції 1, за простими та інтуїтивними API (RDD, DataFrame), дозволяючи розробникам зосередитись на логіці обробки даних.

Що таке Apache Spark

Визначення та контекст

Apache Spark — це високопродуктивний, уніфікований аналітичний рушій для обробки великих даних (Big Data) та машинного навчання. Розроблений в AMPLab Каліфорнійського університету в Берклі, він зараз є одним з найважливіших проєктів Apache Software Foundation та де-факто стандартом для обробки розподілених даних в індустрії.

Основні характеристики та переваги

Швидкість: Spark виконує обчислення в оперативній пам'яті (in-memory), що дозволяє йому бути до 100 разів швидшим за Hadoop MapReduce під час обробки даних в пам'яті та близько 10 разів швидшим під час роботи з диском. Це суттєво змінює можливості аналізу даних в реальному часі.

Універсальність: На відміну від MapReduce, який обмежений в типах операцій, Spark підтримує пакетну обробку (batch processing), SQL-запити, потокову обробку (stream processing), машинне навчання та графові обчислення — все в одному стеку.

Простота й доступність: Spark надає API для декількох мов програмування (Scala, Java, Python/PySpark, R), що робить його доступним для широкого кола розробників та аналітиків.

Гнучкість розгортання: Spark може працювати у Standalone режимі, на YARN (Hadoop), Kubernetes та в хмарних середовищах.

Екосистема Apache Spark

Spark Core Планування завдань, розподілені обчислення, менеджмент пам'яті, відмовостійкість. Це фундамент, на якому будуються всі інші компоненти (див. рис. 2.1).

Spark SQL Підтримка структурованих даних, Catalyst оптимізатор, інтеграція з DataFrame.

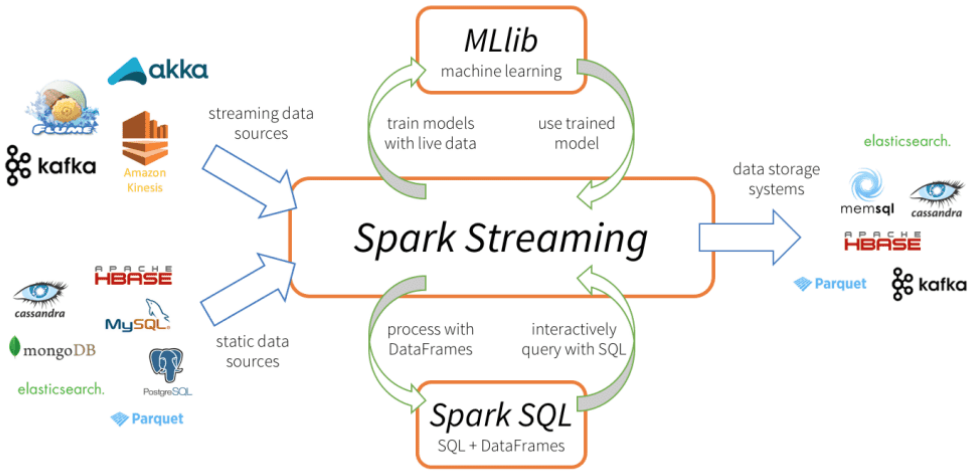

Рис. 2.1: Компоненти екосистеми Apache Spark та їх взаємозв'язки.

Spark Streaming Обробка потоків (мікробатчі) в реальному часі.

MLlib Алгоритми класифікації, регресії, кластеризації, рекомендацій.

GraphX API для графових обчислень (PageRank, трикутники, тощо).

Архітектура та режими виконання

Основні компоненти архітектури

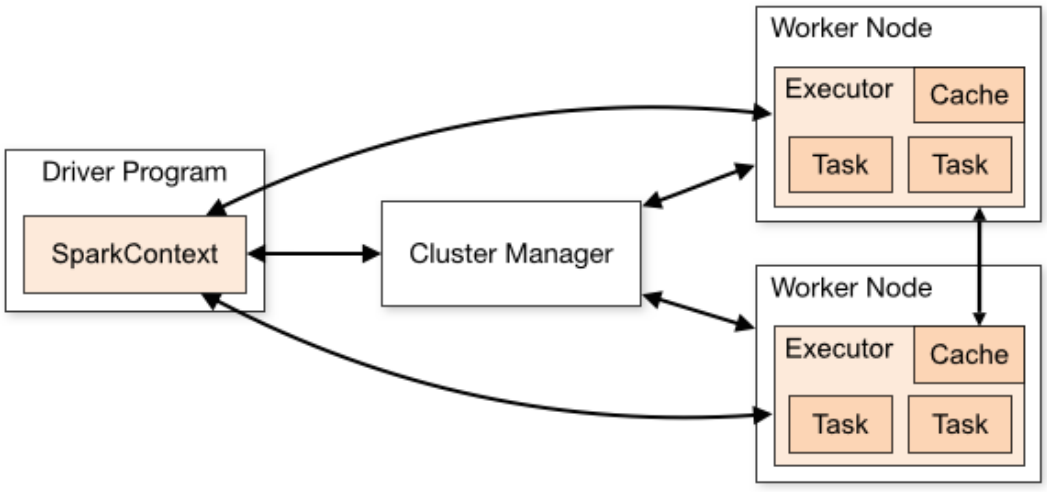

Рис. 2.2: Архітектура Spark-додатку: взаємодія Driver, Cluster Manager, Worker Nodes та Executors.

Спарк-додаток складається з чотирьох основних компонентів, які взаємодіють для виконання розподіленої обробки даних:

Driver Program: Це головна програма, що запускає функцію `main()`. Драйвер створює об'єкт `SparkContext` (або `SparkSession` у сучасних версіях), який служить точкою входу для взаємодії зі Spark. Драйвер координує всю роботу, розбиває програму на задачі, відправляє їх виконавцям та збирає результати. Драйвер також запускає веб-інтерфейс (Spark UI) для моніторингу на порті 4040 (див. рис. 2.2).

Cluster Manager: Це зовнішній сервіс, відповідальний за виділення ресурсів кластера та управління вузлами. Spark підтримує кілька менеджерів: Standalone (вбудований простий менеджер), YARN (Hadoop Resource Manager), Kubernetes та Mesos.

Worker Node: Це будь-який вузол кластера, здатний запускати код додатку. На кожному worker вузлі бігають один або кілька процесів-виконавців (executors), які виконують задачі та утримують дані в пам'яті або на диску.

Executor: Це JVM-процес, запущений на worker вузлі, який виконує конкретні задачі (tasks) від драйвера. Executor зберігає RDD та DataFrame в пам'яті та повертає результати обчислень драйверу.

Режими роботи кластера

Вибір режиму залежить від середовища розгортання та цілей:

Local Mode: Драйвер та всі виконавці працюють в одному JVM на одній машині. Це найпростіший режим для розробки та тестування. Запуск: `spark.master('local[N]')` де N — кількість потоків.

Standalone Mode: Spark використовує власний вбудований менеджер ресурсів. Один вузол називається Master, інші — Worker вузли. Простий варіант для малих кластерів.

YARN (Yet Another Resource Negotiator): YARN є менеджером ресурсів Hadoop. Це найпоширеніший вибір для продакшену в організаціях, що

використовують Hadoop, оскільки дозволяє спільне використання ресурсів для MapReduce та Spark завдань.

Kubernetes: Сучасна система оркестрації контейнерів, що набирає популярність для розгортання Spark [17], [18]. Дозволяє еластичне масштабування та автоматичне управління ресурсами.

Mesos: Розподілений менеджер ресурсів, здатний запускати різноманітні фреймворки. Забезпечує гнучкість, але менш поширений ніж YARN або Kubernetes.

Життєвий цикл додатка

1. Відправка через `spark-submit`.
2. Ініціалізація `SparkContext` / `SparkSession` драйвером.
3. Запит ресурсів у Cluster Manager.
4. Запуск executor'ів на worker вузлах.
5. Розсилка коду та залежностей.
6. Планування: перетворення логіки у *Jobs* → *Stages* → *Tasks*.
7. Виконання, збір та повернення результатів.

Ієрархія виконання. **Job** складається зі **Stages**; кожна Stage містить множину паралельних **Tasks**. Межі Stage визначаються *wide* трансформаціями, що вимагають shuffle.

RDD: Resilient Distributed Dataset

Концепція та властивості

RDD (Resilient Distributed Dataset) є фундаментальною структурою даних в Spark та втілює ключові ідеї розподілених обчислень. Назва RDD енкапсулює три критичні властивості:

Resilient (Стійка до відмов): RDD може відновитися після відмови вузла завдяки зберіганню *lineage* (родоводу) — послідовності трансформацій, які призвели до створення RDD. Якщо партиція RDD втрачена, Spark може переобчислити її, просто перевиконавши операції.

Distributed (Розподілена): Дані в RDD розбиті на партиції, розподілені по різних вузлах кластера. Це дозволяє паралельно обробляти дані та забезпечує масштабованість.

Immutable (Незмінна): RDD неможна змінити після створення. Натомість, трансформації створюють нові RDD. Це спрощує міркування про програму та дозволяє Spark безпечно розподіляти обчислення.

Lazy Evaluation (Лініві обчислення): Трансформації над RDD не виконуються негайно. Обчислення запускаються лише тоді, коли викликається дія (action), що повертає результат користувачу.

Створення

- З колекції: `sc.parallelize(Array(1,2,3))`.
- З зовнішніх джерел: HDFS, S3, локальна ФС, JDBC, NoSQL (HBase, Cassandra).
- Через трансформації існуючих RDD.

Операції

Трансформації (lazy): `map`, `filter`, `flatMap`, `reduceByKey`, `groupByKey`.

- Narrow: не потребують shuffle (`map`, `filter`).
- Wide: потребують shuffle (`reduceByKey`, `groupByKey`).

Дії (actions): `collect`, `count`, `first`, `take`, `saveAsTextFile`.

Кешування та персистентність

Використовуються для ітеративних алгоритмів та повторного читання.

- `cache()` еквівалент `persist(MEMORY_ONLY)`.
- Рівні: MEMORY_ONLY, MEMORY_AND_DISK, MEMORY_ONLY_SER, DISK_ONLY.
- Видалення: `unpersist()`.

Лініві обчислення та DAG

Принцип лінивих обчислень

Лініві обчислення (Lazy Evaluation) — це стратегія, за якою вирази не обчислюються доти, доки їх результат не стане дійсно необхідним. У Spark всі **трансформації** (map, filter, flatMap, reduceByKey) є лінивими: вони не виконуються негайно, а лише додаються до плану виконання. Обчислення запускаються лише тоді, коли викликається **дія (action)** — операція, яка повертає результат користувачу (collect, count, take) або записує дані (saveAsTextFile).

Переваги лінивих обчислень

- **Оптимізація ланцюжка операцій:** Spark бачить весь ланцюжок операцій перед виконанням і може його оптимізувати.
- **Мінімізація проміжних матеріалізацій:** Без ленивих обчислень, кожна операція мала б записувати результат. З лінивими обчисленнями, Spark часто виконує операції без збереження проміжних результатів.
- **Економія I/O:** Мінімізується кількість записів та читань з диску, що значно прискорює обробку даних.

DAG: Спрямований Ациклічний Граф

DAG (Directed Acyclic Graph) — це спрямований граф без циклів, який представляє послідовність трансформацій до дії. Вершини DAG — це RDD, а ребра

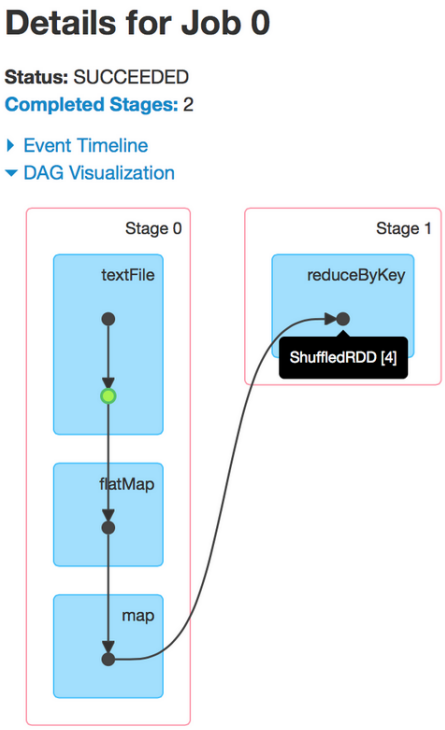

Рис. 2.3: Приклад спрямованого ациклічного графу (DAG), що показує послідовність трансформацій та їх розбиття на стадії (stages).

— це трансформації.

Коли користувач викликає дію, **DAG Scheduler** розбиває граф на **стадії (stages)**. Стадія — це набір паралельних tasks, які можна виконати разом без перемішування даних (shuffle). Межі стадій утворюються *wide* трансформаціями (reduceByKey, groupByKey, join), які потребують обміну даними між партиціями.

PySpark: Python API

Чому Python + Spark

- Лаконічний синтаксис та велика спільнота.
- Багата екосистема (NumPy, Pandas, scikit-learn).
- Низький поріг входу для аналітиків.

Архітектура PySpark

Комунікація Python $\leftrightarrow$ JVM через Py4J: драйвер керує JVM, executors запускають Python worker процеси. Накладні витрати — серіалізація та передача даних.

SparkSession та SparkContext

SparkContext Історичний вхід для RDD (доступний через `spark.sparkContext`).

SparkSession Уніфікує SQLContext, HiveContext, надає API для DataFrame/Dataset/RDD.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
  .appName("MySparkApp") \
  .master("local[*]") \
  .getOrCreate()
```

DataFrame API

Концепція та переваги

Spark DataFrame — це розподілена колекція рядків, організована в іменовані стовпці. Концептуально DataFrame еквівалентен таблиці в реляційній базі даних або pandas DataFrame в Python, але з оптимізаціями для розподілених обчислень на великих масштабах.

Кожен DataFrame має **схему (schema)**, яка визначає імена та типи даних стовпців. Spark може автоматично визначати схему (schema inference) або її можна задати вручну. Наявність схеми дозволяє Spark ефективно оптимізувати запити, застосовуючи правила перетворення операцій та виконуючи типобезпечні трансформації.

Переваги над RDD

- Catalyst оптимізація логічного плану.
- Tungsten — ефективне керування пам'яттю та генерація коду.
- Декларативний високорівневий API.

Схема

Автовизначення або явне задання; схема дозволяє типобезпечні оптимізації, відсіювання стовпців, pushdown фільтрів.

DataFrame vs Dataset vs RDD

extbfRDD: низькорівнева, незмінна колекція об'єктів JVM або Python; відсутні оптимізації Catalyst; придатна для довільних функцій над даними та роботи з некерованими типами. extbfDataFrame: таблична структура зі схемою; оптимізація через Catalyst та Tungsten; вирази і функції перетворюються у логічний/фізичний план. extbfDataset (Scala/Java): типобезпечне розширення DataFrame з підтримкою функціональних трансформацій над strongly-typed об'єктами (Encoder); у Python не підтримується окремо. extbfВибір:

- Використовуйте DataFrame для більшості аналітичних / SQL-подібних задач.
- Dataset корисний коли важлива статична типобезпека у Scala та складні бізнес-об'єкти.
- RDD застосовуйте лише коли потрібні низькорівневі операції, нестандартні формати або контроль над партиціонуванням без накладних оптимізатора.

extbfSerializaція: DataFrame/Dataset використовують внутрішній binary формат (Tungsten) ефективніший, ніж серіалізація об'єктів у RDD (Java/Python pickling). extbfОптимізації недоступні для RDD: автоматичне відсіювання колонок, predicate pushdown, AQE join selection, whole-stage codegen.

Створення

- З файлів: `spark.read.csv()`, `.json()`, `.parquet()`.
- З колекції: `spark.createDataFrame(data, schema)`.
- З RDD: `rdd.toDF()`.

Трансформації

- `select`, `filter`, `withColumn`, `groupBy().agg()`, `join`.

Дії

- `show()`, `collect()`, `count()`, `write.csv()`, `write.parquet()`.

Spark SQL та тимчасові представлення

Spark SQL дозволяє виконувати стандартні SQL-запити до даних, що зберігаються в DataFrame. Це забезпечує потужну інтеграцію між реляційним та процедурним підходами, дозволяючи користувачам, знайомим зі SQL, швидко перейти до роботи зі Spark.

Щоб виконувати SQL-запити, DataFrame потрібно зареєструвати як **тимчасове представлення (Temporary View)**:

```
df.createOrReplaceTempView('my_table')
results = spark.sql('SELECT * FROM my_table WHERE age > 30')
results.show()
```

Результатом виконання SQL-запиту завжди є новий DataFrame, що дозволяє легко поєднувати SQL з DataFrame API та ланцюжком інших операцій. Представлення є локальним для поточної SparkSession.

Structured Streaming

Мотивація та модель

extbfStructured Streaming є рекомендованим високорівневим підходом до потокової обробки у сучасних версіях Spark. На відміну від класичного Spark Streaming (мікробатчі над DStream API), Structured Streaming працює над `DataFrame`/`Dataset` і розглядає нескінченний потік даних як *таблицю, що постійно поповнюється новими рядками*. Це дозволяє застосовувати той самий оптимізатор (Catalyst) та уніфікований API. extbfВідмінності від DStream API: відсутність явного RDD-шару у користувача, декларативні агрегати, єдина семантика для batch і streaming, спільні оптимізації (Project/Filter pushdown, AQE).

Часові семантики

- **Event Time:** час події всередині даних — використовується для віконних обчислень.
- **Processing Time:** системний час обробки на драйвері.
- **Ingestion Time:** проміжний підхід (коли штамп ставиться під час прийому).

Watermark і пізні дані

Watermark визначає межу *максимально очікуваного запізнення*. Події зі значенням `eventTime < (currentWatermark - allowedLateness)` можуть бути відкинуті або не оновлювати стан агрегатів. Це обмежує нескінченний ріст стану.

State Store

Стан (наприклад, проміжні агрегати у вікнах) зберігається у state store (HDFS/локально) з періодичними snapshot і журналом. У разі збою виконується відновлення за checkpoint.

Checkpointing

extttcheckpointLocation обов'язковий для відновлюваних запитів: метадані прогресу, offsets джерел, стан агрегатів. Без нього — неможливо забезпечити exactly-once для джерел з відтворюваними offsets.

Exactly-once семантика

Досягається при: джерело з ідемпотентними/монотонними offsets (Kafka [19]), ідемпотентний sink (Delta/Parquet з merge або append + partition overwrite) або transactional sink (Delta Lake).

Підтримувані sinks

Консоль, пам'ять (debug), файлові формати (Parquet/JSON/ORC), Kafka (write to topic), Delta Lake (ACID), Foreach (кастомна логіка).

Патерни проектування

- **Late data handling:** використання watermark + allowed lateness.
- **Small batch tuning:** корекція тригера (ProcessingTime) для latency vs throughput.
- **Idempotent sinks:** уникнення дублювань під час перезапуску.

Ключові поняття

- **Trigger:** інтервал або режим обробки (`ProcessingTime`, `Once`, `Continuous`).
- **Watermark:** механізм управління пізніми подіями для стану агрегатів у вікнах.
- **Output Modes:** `append`, `update`, `complete` — визначають, які рядки виходять у sink.

- **Stateful Aggregations:** обчислення з підтримкою стану (віконні функції, поточні агрегати) у пам'яті.

Приклад потоку з Kafka

```
from pyspark.sql import functions as F
schema = "user STRING, action STRING, ts TIMESTAMP"

events = spark.readStream \
  .format("kafka") \
  .option("subscribe", "user_events") \
  .option("kafka.bootstrap.servers", "localhost:9092") \
  .load() \
  .selectExpr("CAST(value AS STRING) AS json") \
  .select(F.from_json("json", schema).alias("data")).
  select("data.*")

agg = events \
  .withWatermark("ts", "10 minutes") \
  .groupBy(F.window("ts", "5 minutes"), "action") \
  .count()

query = agg.writeStream \
  .outputMode("update") \
  .format("console") \
  .trigger(processingTime="30 seconds") \
  .start()

query.awaitTermination()
```

Переваги

- Єдиний оптимізатор для batch і streaming (спрощує підтримку).
- Автоматичне керування станом і відновлення.

- Інтеграція з форматами (Parquet, Delta) для *near-real-time* аналітики.

Adaptive Query Execution (AQE)

Призначення

extbfAQE — механізм динамічної оптимізації фізичного плану SQL/DataFrame запиту під час виконання на основі фактичних статистик (реальні розміри партицій, виявлений skew, кількість вихідних рядків). Після завершення підплану збираються статистики і можуть бути прийняті рішення про зміну стратегії join, кількості партицій чи розбиття перекошених даних.

Архітектурна ідея

Запит стартує з початкового фізичного плану. Після завершення shuffle стадії AQE аналізує фактичні метрики і перебудовує наступні стадії (re-optimization) без зміни кінцевої семантики.

Можливості

- **Coalesce Shuffle Partitions:** Автоматичне зменшення кількості дрібних партицій після shuffle.
- **Skew Join Optimization:** Виявлення перекошених ключів і розбиття великих партицій для балансування.
- **Dynamic Join Selection:** Перемикання між Sort-Merge, Broadcast Hash, Shuffle Hash Join в залежності від реальних розмірів.

Увімкнення

```
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled",
"true")
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")
```

Приклад вигоди

Якщо початково план передбачав 200 партицій після великої агрегації, AQE може зменшити їх до 40 (coalesce), знизивши накладні витрати на завдання. Broadcast join може бути обраний динамічно, якщо реальний розмір таблиці < порогового значення.

Конфігурація та пороги

- `spark.sql.autoBroadcastJoinThreshold` — байтовий поріг broadcast (дефолт 10MB).
- `spark.sql.adaptive.skewJoin.skewedPartitionFactor` — коефіцієнт виявлення skew.
- `spark.sql.adaptive.skewJoin.parallelism` — максимальна кількість частин для розбиття перекошеної партиції.

Обмеження

Працює лише для SQL/DataFrame; не застосовується до чистих RDD трансформацій. Планові підказки (hints) можуть бути змінені AQE, якщо суперечать метрикам.

Broadcast змінні та Accumulators

Broadcast змінні

extbfBroadcast Variable — одноразово серіалізована копія невеликого *довідкового* набору даних, доступна всім executor-ам без повторної передачі через мережу для кожного task. Зменшує обсяг shuffle для асиметричних join.

```
dim = spark.read.parquet("/data/dim_users")
fact = spark.read.parquet("/data/events")

joined = fact.join(F.broadcast(dim), "user_id")
```

Критерій: якщо розмір таблиці < 10 МБ (налаштовується через `spark.sql.autoBroadcastJoinThreshold`) — вигідно broadcast. Надто великі broadcast можуть викликати тиск на пам'ять executor-a.

Підводні камені broadcast

- Занадто великий об'єкт — GC та повільна десеріалізація.
- Фрагментація пам'яті off-heap використовуючи Tungsten.
- Нерівномірне використання, якщо логіка фільтрацій відкидає значну частину.

Accumulators

Лічильники для діагностики / збирання метрик. Підтримують комутативну асоціативну операцію додавання. У DataFrame плані вони використовуються рідше через декларативність (UDF може збільшувати). Значення може бути інкрементоване кілька разів під час спекулятивного виконання task.

```
from pyspark import AccumulatorParam
errors = spark.sparkContext.accumulator(0)

def parse(row):
    global errors
    try:
        # parsing logic
        return row
    except Exception:
        errors += 1
        return None

clean = rdd.map(parse).filter(lambda x: x is not None)
clean.count()
print("Errors:", errors.value)
```

extbfЗастереження: не використовуйте для бізнес-логіки; дублікація можливих інкрементів під час повторного виконання tasks.

Альтернатива

Structured Streaming: агрегації стану + sink для метрик (Delta / Prometheus exporter через foreachBatch).

Формати та Lakehouse

Мотивація

Зростання вимог до ACID транзакцій, schema evolution та ефективних апдейтів у аналітичних робочих навантаженнях привело до появи lakehouse форматів поверх data lake: Delta Lake, Apache Iceberg, Apache Hudi.

Delta Lake

Ключові можливості: ACID транзакції (transaction log), schema enforcement & evolution, time travel (версії), vacuum (очистка застарілих файлів), OPTIMIZE + Z-Ordering для покращення locality.

```
# Запис у Delta
df.write.format("delta").mode("append").save("/data/delta/events")

# Читання конкретної версії (time travel)
v3 = spark.read.format("delta") \
        .option("versionAsOf", 3) \
        .load("/data/delta/events")

# Merge (псевдокод Scala SQL):
-- MERGE INTO target t USING updates u ON t.id = u.id
-- WHEN MATCHED THEN UPDATE SET t.val = u.val
-- WHEN NOT MATCHED THEN INSERT (id,val) VALUES (u.id,u.val)
```

Apache Iceberg

Маніфести та снапшоти забезпечують розділення метаданих: приховане партиціонування (partition spec), schema evolution без повного перепису, вбудована підтримка row-level operations (delete, update) у Spark 3.x через інтеграцію.

Apache Hudi

Фокусується на частих upsert: дві стратегії зберігання — Copy-On-Write (COW) та Merge-On-Read (MOR). Metadata таблиці дозволяють інкрементальні читання.

Вибір формату

- Delta: прості транзакції, активна екосистема, потужні оптимізації (OPTIMIZE/ZORDER).
- Iceberg: масштабовані метадані, приховане партиціонування, хороша інтеграція з багатьма движками.
- Hudi: інкрементальні читання, оптимальний для streaming upsert сценаріїв (CDC).

Pushdown та статистики

Колонкові формати (Parquet/ORC) забезпечують predicate pushdown, статистики мінімум/максимум, що дозволяє скорочувати I/O. Lakehouse формати додають рівень транзакцій та еволюцію схеми поверх цих властивостей.

Pandas API on Spark та Arrow

Pandas API on Spark

У Spark доступний pandas-подібний API (pyspark.pandas або ps) для поступової міграції від локальних pandas до розподіленого виконання без повної зміни ідіом.

```
import pyspark.pandas as ps
psdf = ps.read_parquet("/data/parquet/users")
result = psdf.groupby("country").agg({"id":"count"})
print(result.head())
```

Перевага: знайомий синтаксис; обмеження: не всі pandas операції ідентичні по продуктивності — слід уникати дрібногранульованих операцій у великій кількості.

Конвертація між Spark DataFrame та pandas

```
spark_df = spark.range(0, 1000).toDF("id")
pdf = spark_df.toPandas()        # Використовує Arrow (за потреби)
back = spark.createDataFrame(pdf)
```

Apache Arrow інтеграція

Прискорює серіалізацію між JVM та Python за рахунок колонки в пам'яті (zero-copy). Активується опцією:

```
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", "true")
```

Обмеження: типи, які не підтримуються Arrow (деякі комплексні), можуть викликати fallback.

Рекомендації

Використовуйте Arrow для масових перетворень; уникайте багаторазових дрібних toPandas() — краще агрегувати перед вивантаженням.

Висновки

Apache Spark сформував стандарт де-факто для розподіленої обробки даних завдяки in-memory парадигмі, уніфікованій екосистемі та потужним оптимізаторам. Розуміння архітектури, RDD vs DataFrame, лінивих обчислень, механізму shuffle та інструментів моніторингу є критичним для побудови ефективних

і масштабованих аналітичних рішень. PySpark демократизує доступ до цих можливостей для спільноти Python. Сучасні підходи до розгортання інфраструктури (IaC) також стають невід'ємною частиною екосистеми [20].

Контрольні запитання

1. Різниця між Driver, Executor та Cluster Manager у життєвому циклі Spark додатка.
2. Значення ієрархії Job → Stages → Tasks; приклади wide і narrow трансформацій.
3. Як lineage забезпечує відмовостійкість? Відновлення втраченої партиції.
4. Порівняйте RDD і DataFrame (оптимізації, типобезпека, сценарії).
5. Відмінності між `reduceByKey` та `groupByKey` (мережеві витрати).
6. Механізм lazy evaluation та оптимізація ланцюжка операцій.
7. Межі розбиття на stages DAG Scheduler; приклад послідовності.
8. Рівні персистентності та коли використовувати MEMORY_AND_DISK.
9. Чому `collect()` небезпечний на великих даних? Альтернативи.
10. Різниця між `cache()` і `persist()` (серіалізація).
11. Фази Catalyst Optimizer та вибір фізичного плану.
12. Відмінність плану DataFrame від RDD операцій.
13. Shuffle: причини, операції та мінімізація.
14. Критерії вибору формату (CSV vs Parquet vs JSON).
15. Py4J інтеграція та накладні витрати.
16. Приклад створення DataFrame з явною схемою та переваги.
17. Коли застосовувати UDF; ризики для оптимізації.

18. Narrow vs wide трансформації (`map` vs `reduceByKey`).

19. Еволюція SparkSession vs SparkContext.

20. Стратегія діагностики повільного task у стадії (метрики, логування).

Лекція 3

Робота з DataFrame та Spark SQL

Трансформації, дії (actions), SQL-запити, оптимізація та віконні функції

Навчальні цілі

Після опрацювання цієї лекції студент повинен уміти:

- Розрізнити трансформації та дії, застосовувати лінивість для оптимізації.
- Виконувати базові та розширені операції (select, filter, groupBy, join, window).
- Писати й виконувати SQL-запити до DataFrame, розуміючи еквівалентність з API.
- Пояснити фази Catalyst Optimizer та як вони покращують план виконання.
- Діагностувати та мінімізувати проблеми з join (розмір, стратегія, skew).
- Застосовувати window функції для рейтингування, аналізу тенденцій та порівнення.
- Обрати правильний формат запису (CSV, JSON, Parquet, ORC).
- Розуміти вплив партиціонування та skew на продуктивність.

Від базових концепцій до практичної аналітики

У попередній лекції ми розглянули фундаментальну архітектуру Spark, RDD та принципи лінивих обчислень. Тепер переходимо до практичної роботи зі структурованими даними через DataFrame та Spark SQL [21].

DataFrame є головним інструментом сучасного Spark-розробника: він поєднує простоту SQL з потужністю розподілених обчислень, одночасно дозволяючи оптимізатору Catalyst трансформувати декларативні запити в ефективні фізичні плани. У цій лекції ми дослідимо, як писати, оптимізувати та діагностувати DataFrame запити, розуміючи водночас внутрішні механізми, які роблять Spark таким швидким.

Що таке Apache Spark?

Визначення та основні характеристики

- **Розподілена платформа для обробки великих даних**
- **Швидка обробка в пам'яті** з використанням RDD та DataFrame
- **Підтримує**: Scala, Python, Java, SQL
- **Компоненти**: Spark Core, SQL, Streaming, MLlib, GraphX

Детальне обговорення архітектури, компонентів та режимів виконання див. у Lecture 2.

Концепція DataFrame

Що таке DataFrame?

- Структурована колекція розподілених даних
- Організована в іменовані колонки та рядки
- Аналог табличних даних у Python (Pandas) та R
- Побудована на RDD, але з оптимізацією

Переваги DataFrame перед RDD:

- Кращі оптимізації (Catalyst Optimizer)
- Зрозумілий API для роботи з даними
- Підтримка SQL запитів
- Менша потреба пам'яті

Порівняння із RDD

RDD (Resilient Distributed Dataset)	**DataFrame**
Низькорівневе API	Високорівневе API
Більша гнучкість	Простіше в використанні

Рекомендація: Використовувати DataFrame для більшості задач.

Створення DataFrame

Способи створення DataFrame

1. З Pandas DataFrame
2. З послідовності даних
3. З зовнішніх джерел
4. З RDD

Читання даних з різних джерел Побудова через `DataFrameReader`:

- `spark.read.csv('path/to/file.csv')`
- `spark.read.json('path/to/file.json')`
- `spark.read.parquet('path/to/file.parquet')`
- `spark.read.option('header', True).csv(...)`

Інші форми:

- JDBC (база даних)
- ORC, Avro, XML
- JDBC для SQL Server, PostgreSQL, MySQL

Spark автоматично визначає типи даних.

Структура та схема

Схема DataFrame (Schema) Схема (Schema): Визначає структуру DataFrame.

Компоненти схеми:

- Назва колонки (column name)
- Тип даних (data type): `IntegerType`, `StringType`, `DoubleType`, ...
- Допускає null значення (nullable)
- Метадані

Приклад схеми:

```
StructType([
    StructField('id', IntegerType, False),
    StructField('name', StringType, True),
    StructField('salary', DoubleType, True)
])
```

Специфікація схеми дозволяє оптимізувати обробку.

Інспекція структури Основні методи для інспекції:

- `df.printSchema()` — вивести схему форматовано
- `df.dtypes` — список типів колонок
- `df.columns` — список назв колонок
- `df.describe()` — статистика по колонках

- `df.info()` — інформація про DataFrame
- `df.show()` — перші рядки даних
- `df.count()` — загальна кількість рядків

  Приклад: `df.printSchema()`, `df.show(5)`

Трансформації DataFrame

Базові операції

Select та фільтрація

- **Select:** вибір конкретних колонок
  - `df.select('col1', 'col2')`
  - `df.select(col('name'), col('salary'))`
  - `df['col1']` — альтернативний синтаксис
- **Фільтрація:** відбір рядків за умовою
  - `df.filter(col('age') > 30)`
  - `df.where(col('department') == 'IT')`
  - `df.filter((col('salary') > 50000) & (col('age') < 40))`

  Select та filter можна комбінувати.

Where та умовна фільтрація

- **Where:** аналогічна `filter`, але зазвичай використовується в SQL контексті.
  - `df.where(col('status') == 'active')`
  - `df.where(col('date') >= '2024-01-01')`
- **Складні умови:**

  - `&` — логічне І (AND)
  - `|` — логічне АБО (OR)
  - `~` — логічне НІ (NOT)
  - `isin([...])` — перевірка входження

- **Приклад:** `df.where((col('salary') > 50000) & (col('department').isin(['IT', 'HR'])))`

GroupBy та агрегація

- **GroupBy:** групування даних за колонками
  - `df.groupBy('department')`
  - `df.groupBy('department', 'year')` — множинне групування
- **Функції агрегації:**
  - `.count()` — кількість рядків
  - `.sum('salary')` — сума значень
  - `.avg('salary')` — середнє значення
  - `.max('salary')` — максимальне значення
  - `.min('salary')` — мінімальне значення
- **Приклад:** `df.groupBy('department').agg(avg('salary'), count('*'))`

Розширені операції

Join операції

Join: об'єднання двох DataFrame за спільною колонкою.

- `df1.join(df2, 'id')` — inner join (за замовчуванням)
- `df1.join(df2, 'id', 'left')` — left outer join
- `df1.join(df2, 'id', 'right')` — right outer join

- `df1.join(df2, 'id', 'outer')` — full outer join
- `df1.join(df2, 'id', 'cross')` — декартів добуток

Приклад inner join: `employees.join(departments, 'dept_id', 'inner')`

Join — дорога операція, потребує shuffle.

Union та розширення даних

- **Union:** об'єднання рядків двох DataFrame
  - `df1.union(df2)` — додає рядки df2 до df1
  - `df1.unionByName(df2)` — union за назвами колонок
  - `df1.except(df2)` — рядки з df1, які не в df2
  - `df1.intersect(df2)` — рядки, які є в обох
- **Додавання колонок:**
  - `df.withColumn('new_col', expr)`
  - `df.drop('col_to_remove')`
  - `df.rename('old_name', 'new_name')`
- **Приклад:** `df.withColumn('salary_increased', col('salary') * 1.1)`

Window функції

Window функції: обчислення за вікном рядків.

- `row_number()` — номер рядка в вікні
- `rank()` — ранг з пропусками
- `dense_rank()` — ранг без пропусків
- `lag(col, offset)` — значення з попередніх рядків
- `lead(col, offset)` — значення з наступних рядків

Визначення Window:

```
from pyspark.sql.window import Window
w = Window.partitionBy('department').orderBy('salary')
```

Window функції корисні для рейтингування та аналізу тенденцій.

Дії (Actions)

Типи дій (Actions)

Actions: операції, що повертають результати до драйвера або записують дані.

Основні категорії:

1. **Дані до драйвера:** `collect()`, `first()`, `take()`
2. **Відображення:** `show()`, `display()`
3. **Запис:** `write()`, `save()`
4. **Статистика:** `count()`, `describe()`

Важливо: Actions запускають обчислення!

- Трансформації (select, filter, . . .) — ліниві
- Actions (collect, show, . . .) — активні

Використовуйте actions обережно, особливо з великими даними.

`collect()`, `show()`, `write()`

- `collect()`: повернення всіх рядків до драйвера
  - `data = df.collect()`
- `show()`: відображення рядків
  - `df.show()`, `df.show(5)`, `df.show(truncate=False)`

- `write()`: запис даних
  - `df.write.mode('overwrite').parquet('path')`
  - `df.write.mode('append').csv('path', header=True)`
  - Mode: overwrite, append, ignore, error (за замовчуванням)

`count()`, `first()`, `take()`

- `count()`: кількість рядків
  - `total = df.count()`
- `first()`: перший рядок
  - `row = df.first()`
- `take(n)`: перші n рядків
  - `rows = df.take(10)`
- **Для отримання статистики:**
  - `df.describe('age', 'salary').show()`
  - Це забезпечує count, mean, stddev, min, max

Spark SQL

SQL запити

Реєстрація таблиць та view

- **Створення тимчасової таблиці:**
  `df.createOrReplaceTempView('employees')`
- **Типи view:**
  - `createOrReplaceTempView()` — сессійна таблиця
  - `createGlobalTempView()` — глобальна таблиця

- `createOrReplaceView()` — постійна таблиця

- **Виконання SQL запиту:**
  - `result = spark.sql('SELECT * FROM employees WHERE salary > 50000')`
  - Результат SQL запиту — це DataFrame.

SQL SELECT запити

- **Базовий SELECT:** `SELECT id, name, salary FROM employees WHERE department = 'IT' ORDER BY salary DESC`
- **JOIN в SQL:** `SELECT e.name, d.dept_name FROM employees e INNER JOIN departments d ON e.dept_id = d.id`
- **GROUP BY та агрегація:** `SELECT department, AVG(salary), COUNT(*) FROM employees GROUP BY department`

  SQL часто простіший за DataFrame API для складних запитів.

Складні SQL операції

- **Window функції в SQL:**

```
SELECT name, salary, ROW_NUMBER() OVER (PARTITION BY department
ORDER BY salary DESC) as rank FROM employees
```

- **Підзапити та CTE:**

```
WITH high_earners AS (
  SELECT * FROM employees WHERE salary > 100000
)
SELECT department, COUNT(*) FROM high_earners
GROUP BY department
```

CTE (Common Table Expressions) роблять код більш читаємим.

Оптимізація

Catalyst optimizer

Catalyst: оптимізаційний двигун Spark SQL.

Фази оптимізації:

1. **Аналіз** — розроблення план дерева
2. **Логічна оптимізація**
3. **Фізична оптимізація**
4. **Генерація коду** — компіляція в Java

Дозволяє Spark автоматично оптимізувати запити.

Best practices для SQL запитів

Рекомендації для оптимізації:

- Фільтрувати дані якомога раніше
- Обирати тільки потрібні колонки
- Уникати декартова добутку
- Використовувати правильні типи даних
- Кешувати часто використовувані таблиці
- Уникати операцій, що призводять до shuffle

Перевірка плану виконання: `df.explain(extended=True)`
Catalyst часто вже добре оптимізує, але розуміння плану допомагає.

Практичні приклади

Сценарій 1: Аналіз даних

Задача: Знайти найбільших розпорядників в компанії.

- **DataFrame підхід:**
  `employees.select('name', 'salary').where(col('salary') > 100000).sort(col('salary').desc()).show()`
- **SQL підхід:**
  `SELECT name, salary FROM employees WHERE salary > 100000 ORDER BY salary DESC`

  Обидва підходи дають однаковий результат.

Сценарій 2: Трансформація та агрегація

Задача: Розрахувати середню зарплату по відділам.

- **DataFrame підхід:**
  `employees.groupBy('department').agg(avg('salary').alias('avg_salary'), count('*').alias('count')).sort(col('avg_salary').desc()).show()`
- **SQL підхід:** `SELECT department, AVG(salary) as avg_salary, COUNT(*) as count FROM employees GROUP BY department ORDER BY avg_salary DESC`

  GroupBy та агрегація — часті операції.

Сценарій 3: Роботи з різнорідними форматами

Задача: Читання з CSV, трансформація та запис у Parquet.

Повний цикл:

```
df = spark.read.option('header',True).csv('employees.csv')
df_processed = df.filter(col('status')=='active').withColumn(
  'salary_increased', col('salary')*1.1)
df_processed.write.mode('overwrite').
parquet('output/processed_employees')
```

Parquet формат більш ефективний для великих даних.

Catalyst Optimizer та оптимізація запитів

Фази Catalyst

Catalyst — це оптимізаційний компонент Spark SQL, який трансформує вхідні вирази у оптимізований фізичний план виконання [22]. Процес містить чотири критичні фази:

1. **Аналіз (Analysis):** Розв'язання імен стовпців та функцій, перевірка типів, побудова логічного плану. На цьому етапі виявляються синтаксичні помилки.

2. **Логічна оптимізація (Logical Optimization):** Rule-based трансформації:

   - *Predicate Pushdown*: фільтри зміщуються вниз плану для раннього відсівання даних
   - *Column Pruning*: видалення непотрібних стовпців на ранніх етапах
   - *Constant Folding*: обчислення виразів з константами під час компіляції
   - *Join Reordering*: переупорядкування з'єднань для зменшення проміжних результатів

3. **Фізична оптимізація (Physical Planning):** На основі логічного плану вибираються конкретні реалізації операторів. Для join: Sort-Merge, Broadcast Hash, Shuffle Hash. Вибір залежить від розміру таблиць та доступних статистик.

4. **Генерація коду (Code Generation):** Whole-stage codegen fusionує оператори (filter → project → aggregate) в один оптимізований Java метод, зменшуючи віртуальні виклики.

Best Practices для оптимізації

- **Фільтрувати дані якомога раніше:** Умови WHERE роблять фільтрацію ефективнішою
- **Вибирати тільки потрібні колонки:** Column Pruning скорочує I/O
- **Уникати декартова добутку:** Перевіряйте умови join на корректність
- **Використовувати правильні типи даних:** Числові типи швидші ніж рядки у агрегаціях
- **Кешувати часто використовувані таблиці:** `df.cache()` для повторних операцій

Перевірка плану виконання

`explain()` **та** `explain(extended=True)`:

```
df.explain()              # Фізичний план
df.explain(extended=True) # Логічний + фізичний план
```

План показує порядок операцій, типи join, і кількість розділів. Аналіз плану допомагає виявити неоптимальні стратегії.

Оптимізація Join

Типи Join у Spark

Spark підтримує кілька стратегій реалізації join:

- **Broadcast Hash Join:** Одна таблиця розглошується усім executor-ам. Оптимально для асиметричних join, коли одна таблиця < 10 MB

- **Sort-Merge Join:** Обидві таблиці сортуються та об'єднуються. Default для великих таблиць
- **Shuffle Hash Join:** Часи обидві таблиці хешуються та розміщуються одна до однієї (рідко вибирається)
- **Cartesian (Cross) Join:** Декартів добуток — УНИКАЙТЕ для великих даних

Критерії вибору стратегії

Broadcast Hash: Розмір меншої таблиці < autoBroadcastJoinThreshold (дефолт 10MB), достатньо пам'яті

Sort-Merge: Великі таблиці, дані вже розпарцейовані або потребують глобального сортування

Shuffle Hash: Середні за розміром таблиці, коли пам'ять обмежена

Явне керування стратегією

```
# Suggest broadcast join
df_small_hint = df_small.hint("BROADCAST")
result = df_large.join(df_small_hint, "id")

# Available hints: BROADCAST, MERGE, SHUFFLE_HASH,
# SHUFFLE_REPLICATE_NL
```

Приклади та розповсюджені помилки

- **Забув broadcast для малої таблиці:** Без hint, Spark може вибрати неоптимальну стратегію
- **Узлід на шпалерах ключиків:** Одна сторона join має шпалери, інша — ні (див. розділ Data Skew)
- **Join без умови:** Cross join на великих даних призводить до OOM

Partitioning та Data Skew

Партиціонування

Розподіл даних на логічні частини (партиції) визначає паралелізм. Для RDD застосовується **Partitioner** (HashPartitioner, RangePartitioner). Для DataFrame — непрямо через shuffle boundary. `repartition(n)` створює повний shuffle; `coalesce(n)` — зменшує кількість партицій без повного shuffle (коли можливе злиття). Під час запису `df.write.partitionBy(col)` — фізична партиціонована організація на диску (прискорює predicate pushdown).

Bucketing

extttbucketBy() (Scala/SQL) створює фіксовану кількість бакетів за хешем стовпця для оптимізації join (уникнення повного shuffle при сумісному bucketing). Вимагає збереження як таблиці Hive/часто у форматі з метаданими.

Вибір кількості партицій

Правило: орієнтовно 2–4 партиції на кожен CPU core executor-a для compute-heavy операцій; більше для I/O bound. Для великих shuffle — контроль через `spark.sql.shuffle.partitions` (дефолт часто 200). `spark.default.parallelism` — дефолт для RDD (часто = кількість cores).

Операції що викликають shuffle

Distinct, groupBy/agg, join по ключу, repartition, sort, orderBy, coalesce (іноді), reduceByKey (менше даних ніж groupByKey завдяки combiner).

Data Skew (перекіс даних)

Перекіс виникає коли кілька ключів мають непропорційно великий об'єм, викликаючи повільні tasks. Наслідок — збільшений час завершення стадії, spill to disk, знижена ефективність паралелізму.

- **Діагностика:** Spark UI — Stages → Task Time; великі відхилення, підвищені spill-и.
- **Метрики:** Середній vs максимальний час task; кількість записів у проблемних партиціях.

Стратегії пом'якшення

- **Salting ключів:** Додавання штучного суфікса (наприклад випадкового) до гарячого ключа перед агреграцією, потім об'єднання.
- **Broadcast малих таблиць:** Зменшення shuffle для join.
- **Попереднє групування:** Локальна агрегація на mapper рівні (reduceByKey замість groupByKey).
- **AQE Skew Optimization:** Автоматичне розбиття великих партицій (якщо увімкнено).

Приклад salting

```
import random
salted = rdd.map(lambda x: ((x[0], random.randint(0,9)), x[1])) \
            .reduceByKey(lambda a,b: a+b) \
            .map(lambda x: (x[0][0], x[1])) \
            .reduceByKey(lambda a,b: a+b)
```

Компроміс: збільшення кількості ключів vs балансування навантаження.

Додаткові техніки

- **Pre-aggregation:** локальна агрегація перед join (зменшення об'єму).
- **Two-phase aggregation:** map-side + reduce-side (Spark робить автоматично для деяких функцій).
- **RangePartitioner:** для числових ключів з нерівномірним розподілом — іноді кращий за hash.
- **Hybrid join strategy:** розбити skew ключі і broadcast другу частину.

Висновки

Ключові концепції лекції:

- DataFrame — основний інструмент для структурованої аналітики; простіший та швидший ніж RDD
- Трансформації ліниві (lazy), дії активні (eager) — цей механізм дозволяє оптимізатору бачити весь ланцюжок операцій
- Catalyst Optimizer трансформує декларативні запити у ефективні фізичні плани через predicate pushdown та інші оптимізації
- Join — дорога операція; правильна стратегія (broadcast vs sort-merge) критична для продуктивності
- Data Skew є частою проблемою; techniques як salting та pre-aggregation допомагають
- Партиціонування впливає на паралелізм; правильна кількість партицій — баланс між паралелізмом та накладними витратами
- DataFrame та SQL — еквівалентні; виберіть стиль за розміром команди та задачі

Коли використовувати які підходи:

- **DataFrame API:** Коли потрібна гнучкість, низькорівневий контроль, интегарація з Python функціями
- **Spark SQL:** Коли запити стандартні, команда знайома з SQL, читаємість коду пріоритет
- **Window функції:** Для рейтингування, часових рядів, лагів та лідів без складних self-join

Контрольні запитання

1. Поясніть різницю між логічним і фізичним планом виконання в Spark SQL.
2. Як Catalyst Optimizer застосовує predicate pushdown і column pruning? Наведіть приклад запиту.
3. Порівняйте DataFrame API і SQL-підхід: критерії вибору для складних ETL конвеєрів.
4. Чим window функції відрізняються від звичайних агрегацій? Наведіть приклад комбінованого використання `rank()` і `groupBy`.
5. Які типові причини появи shuffle у join та groupBy і як його мінімізувати?
6. Поясніть різницю між `union` і `unionByName` та сценарії помилок для різних схем.
7. Чому `collect()` небезпечний для великих DataFrame? Які альтернативи для часткового аналізу?
8. Як працює `explain(extended=True)` і які артефакти допомагають діагностувати неоптимальний план?
9. Поясніть різницю між `inner`, `left`, `right`, `outer` та `cross` join з точки зору семантики та витрат.
10. Коли доцільно використати window функцію `lag()` замість self-join?
11. Яка різниця між broadcast join, sort-merge join та shuffle hash join? Коли кожен підходит?
12. Як діагностувати і усунути data skew в DataFrame? Опишіть техніку salting.
13. Які розміри вибору partitionBy та репартиціонування? Як впливає на продуктивність?
14. Опишіть процес читання даних із CSV з явною схемою: які переваги порівняно з автоматичним визначенням?

15. Як DataFrame зберігає інформацію про nullable і як це впливає на логічні оптимізації?

16. Поясніть різницю між фізичною операцією sort і логічним `orderBy`. Коли сортування обов'язкове?

17. Чому формат Parquet є переважним для аналітики? Поясніть columnar storage та compression.

18. Наведіть приклад комбінованого використання `withColumn`, `filter` і `groupBy` з метою побудови метрики.

19. Чим відрізняється `describe()` від специфічних агрегатів (наприклад, `avg`, `stddev`)? Коли який підхід доречний?

20. Як реалізувати розрахунок ковзного середнього (rolling average) із window функціями?

21. Поясніть відмінність між `row_number()` і `dense_rank()` на прикладі повторюваних значень.

22. Запропонуйте стратегію оптимізації запиту з багатьма join та фільтрами для великого DataFrame.

23. Як правильно вибрати підказку join (BROADCAST, MERGE, SHUFFLE_HASH)?

24. Яка роль Tungsten у підсистемі виконання та як whole-stage codegen покращує продуктивність?

Лекція 4

Інженерія ознак та очищення даних

Робота з різними типами даних, об’єднання наборів, очищення даних

Навчальні цілі

Після опрацювання цієї лекції студент повинен уміти:

- Розрізнити та застосовувати методи масштабування та нормалізації числових даних.
- Трансформувати категоріальні дані за допомогою One-Hot Encoding та Label Encoding.
- Здійснювати об’єднання наборів даних (конкатенацію та merge).
- Виявляти та обробляти пропущені значення (видалення, імпутація).
- Виявляти та мінімізувати вплив викидів (outliers).
- Виявляти та видаляти дублікати в наборах даних.
- Розуміти стратегії дискретизації та створення нових ознак.
- Застосовувати загальний робочий процес очищення даних.

Чому інженерія ознак та очищення даних критичні

Якість моделей машинного навчання залежить не тільки від алгоритмів, але й від якості та репрезентативності вхідних даних. У реальних проектах до 80% часу витрачається на підготовку та очищення даних. Розуміння того, як трансформувати сирі дані у придатні для аналізу ознаки, є фундаментальною навичкою інженера даних та дата-сайентиста [21].

У цій лекції ми розглянемо методи для роботи з різними типами даних, вивчимо проблеми та їх вирішення: від простого видалення пропусків до складних трансформацій через дискретизацію та взаємодійні ознаки.

Інженерія ознак (Feature Engineering)

Визначення та основні завдання

Інженерія ознак — це процес трансформації вихідних даних у набір ознак (features), які краще представляють проблему для моделі машинного навчання, що призводить до покращення точності та продуктивності.

Основні завдання:

- **Створення нових ознак:** Комбінування або перетворення існуючих ознак для отримання більш інформативних. Наприклад, з дати народження можна отримати вік, з повної адреси — регіон.
- **Трансформація даних:** Зміна формату або розподілу даних, щоб вони відповідали вимогам певних алгоритмів.
- **Вибір ознак:** Відбір найважливіших ознак для уникнення надмірної складності, шуму та перенавчання (overfitting).

Залежно від типу даних та алгоритму, стратегія інженерії ознак може істотно відрізнятися.

Робота з числовими даними

Масштабування та нормалізація

Багато алгоритмів машинного навчання (наприклад, лінійна регресія, SVM, кластеризація K-means) чутливі до масштабу даних. Якщо одна ознака має діапазон [0, 100000], а інша — [0, 1], перша домінуватиме у обчисленнях.

Min-Max Scaling (нормалізація)

Min-Max Scaling приводить значення до діапазону [0, 1]:

```
X_scaled = (X - X_min) / (X_max - X_min)
```

Переваги:

- Гарантований обмежений діапазон
- Зберігає форму розподілу

Недоліки:

- Чутлива до викидів (X_max та X_min мають великий вплив)
- Не працює добре для нових даних з значеннями поза діапазоном

Рекомендується для нейронних мереж та алгоритмів, чутливих до масштабу.

Standardization (Z-score нормалізація)

Standardization трансформує дані до розподілу з середнім 0 і стандартним відхиленням 1:

```
X_scaled = (X - mean) / std_dev
```

Переваги:

- Менш чутлива до викидів ніж Min-Max
- Працює добре з алгоритмами, що припускають нормальний розподіл

Недоліки:

- Не обмежує діапазон (може бути < -1 або > 1)

Переважна для регресії, класифікації та PCA.

Вибір метода

Min-Max: Коли потрібен обмежений діапазон (наприклад, вірогідність, пікселі в зображенні)

Standardization: Коли дані мають викиди або припускається нормальний розподіл

Дискретизація (Binning)

Дискретизація — це перетворення неперервних числових даних на категоріальні шляхом поділу їх на інтервали (біни).

Приклад: Замість точного віку (наприклад, 34 роки), створюємо категорії:

- “молодий” (від 18 до 30 років)
- “середній” (від 30 до 50 років)
- “старший” (від 50 до 70 років)

Переваги:

- Допомагає моделі вловлювати нелінійні залежності
- Робить дані більш інтерпретабельними
- Зменшує вплив викидів

Недоліки:

- Втрачається інформація всередині бінів
- Потрібно обирати розмір та межі бінів

Корисна у випадку, коли залежність вік-дохід має форму: молоді та старші люди заробляють менше, середнього віку — більше.

Робота з категоріальними даними

One-Hot Encoding

One-Hot Encoding створює нову бінарну ознаку для кожної унікальної категорії.

Приклад:

- Вихідна колонка: `color` = [“red”, “green”, “blue”]
- Результат:
  - `color_red` = [1, 0, 0]
  - `color_green` = [0, 1, 0]
  - `color_blue` = [0, 0, 1]

Коли використовувати:

- Номінальні дані, де порядок неважливий (колір, країна, тип товару)
- Переважна для більшості алгоритмів машинного навчання

Проблеми:

- Висока кардинальність (багато унікальних значень) призводить до експоненціального зростання ознак (”curse of dimensionality”)
- Мультиколінеарність: суми всіх колонок = 1

Розв’язання: Drop first category (видалити першу колонку) або використати Label Encoding для великої кількості категорій.

Label Encoding

Label Encoding присвоює кожній категорії унікальне числове значення (0, 1, 2, . . .).

Приклад:

- Вихідна колонка: `education` = [“bachelor”, “master”, “phd”]

- Результат: [0, 1, 2]

Коли використовувати:

- Порядкові дані, де є логічний порядок (освіта: бакалавр, магістр, доктор)
- Категорії з великою кардинальністю
- Tree-based моделі (які розуміють порядок)

Небезпека:

- Лінійні моделі можуть неправильно інтерпретувати числа як впорядковані (наприклад, blue=0, red=1, green=2 не означає, що red — це "більше" ніж blue)

Рекомендація: Використовувати Label Encoding лише для явно упорядкованих категорій або tree-based моделей.

Порівняння методів

Аспект	One-Hot Encoding	Label Encoding
Кількість ознак	Дорівнює кількості категорій	1 ознака
Допущення порядку	Ні	Так
Лінійні моделі	Хороший	Небезпечний
Tree-based моделі	Хороший	Дуже хороший
Висока кардинальність	Проблема	Гарно

Об'єднання наборів даних

Конкатенація

Конкатенація — це об'єднання таблиць по одній з осей (вертикально або горизонтально).

Вертикальна конкатенація

Об'єднання рядків з різних таблиць (той же набір стовпців):

```
df_combined = pd.concat([df1, df2, df3], axis=0)
# або
df_combined = df1.union(df2).union(df3)  # PySpark
```

Умови:

- Стовпці мають збігатися (за назвою та типом)
- Порядок стовпців не обов'язково однаковий у PySpark

Горизонтальна конкатенація

Об'єднання колонок з різних таблиць (той же набір рядків):

```
df_combined = pd.concat([df1, df2], axis=1)
# або
df_combined = df1.join(df2, on=index)  # або інші методи
```

Небезпека:

- Рядки мають бути вирівняні за індексом
- Якщо індекси не збігаються, можна отримати NaN значення

Merge (Join)

Merge — це об'єднання таблиць на основі спільних стовпців (ключів), подібно до JOIN в SQL.

Типи merge

- **Inner join:** Тільки рядки, присутні в обох таблицях
  - `pd.merge(df1, df2, on='id', how='inner')`
  - `df1.join(df2, 'id')` (PySpark за замовчуванням)

- **Left join:** Все рядки з лівої таблиці, відповідні з правої (якщо є)
  - `pd.merge(df1, df2, on='id', how='left')`
  - `df1.join(df2, 'id', 'left')` (PySpark)
- **Right join:** Все рядки з правої таблиці, відповідні з лівої (якщо є)
- **Outer join:** Все рядки з обох таблиць

Важливо: Merge може дублювати рядки, якщо ключ не унікальний у одній або обох таблицях!

```
# Приклад дублювання
df1: id=[1,2,3], name=['A','B','C']
df2: id=[1,1,2], dept=['IT','IT','HR']
merge: id=[1,1,1,2], name=['A','A','B','C'],
       dept=['IT','IT','HR','NaN']
```

Рекомендація: Перевіряйте унікальність ключів перед merge!

Очищення даних (Data Cleansing)

Обробка пропущених значень

Пропущені значення (missing values, NaN) виникають через помилки введення, технічні збої, або природні причини. Необхідно визначити, як із ними працювати.

Видалення

Метод: Видалити рядки або стовпці з пропущеними значеннями.

```
# Pandas
df_clean = df.dropna()  # видалити рядки з NaN
df_clean = df.dropna(subset=['col1', 'col2'])
# видалити за конкретним стовпцям
df_clean = df.dropna(thresh=0.8*len(df), axis=1)
```

```
# видалити стовпці з >20% пропусків

# PySpark
df_clean = df.dropna()  # видалити рядки з будь-яким NaN
df_clean = df.dropna(subset=['col1', 'col2'])
# видалити за конкретним стовпцям
```

Коли використовувати:

- Пропусків небагато (< 5% даних)
- Пропуски випадкові (MCAR — Missing Completely At Random)
- Стовпець не критичний для аналізу

Імпутація (заповнення)

Імпутація — це заміна пропущених значень певним значенням.

Методи імпутації:

- **Середнім (mean):** Для числових даних без викидів

  ```
  df['salary'].fillna(df['salary'].mean())
  ```

  - Швидко й просто
  - Зменшує дисперсію, може спотворити розподіл

- **Медіаною (median):** Для числових даних з викидами

  ```
  df['salary'].fillna(df['salary'].median())
  ```

  - Більш стійка до викидів ніж середнє

- **Модою (mode):** Для категоріальних даних

```
df['city'].fillna(df['city'].mode()[0])
```

  - Найпоширеніше значення

- **Константою:** Заповнення нулем, -1 або спеціальним значенням

```
df['status'].fillna('Unknown')
df['age'].fillna(-1)
```

  - Явно позначає пропущені дані
  - Добре для категоріальних даних

- **Forward/Backward fill:** Для часових рядів (останнє відоме значення)

```
df['value'].fillna(method='ffill')  # forward fill
df['value'].fillna(method='bfill')  # backward fill
```

- **Інтерполяція:** Для часових рядів (лінійна або інша)

```
df['value'].interpolate(method='linear')
```

Як обрати метод:

Числові дані без викидів: Mean

Числові дані з викидами: Median

Категоріальні дані: Mode або спеціальна константа

Часові ряди: Forward fill або інтерполяція

Критичні ознаки: Спеціальна модель для передбачення пропусків

Обробка викидів (Outliers)

Викиди — це значення, які значно відрізняються від решти даних. Вони можуть бути результатом помилки виміру або справжнього явища.

Виявлення викидів

Правило трьох сигм (3-sigma rule) Для даних з нормальним розподілом, значення за межами $[\text{mean} - 3\sigma, \text{mean} + 3\sigma]$ вважаються викидами.

```
mean = df['salary'].mean()
std = df['salary'].std()
outliers = df[(df['salary'] < mean - 3*std) |
(df['salary'] > mean + 3*std)]
```

Міжквартильний розмах (IQR) Більш стійкий до екстремальних викидів:

```
Q1 = df['salary'].quantile(0.25)
Q3 = df['salary'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
outliers = df[(df['salary'] < lower_bound) |
(df['salary'] > upper_bound)]
```

Інтерпретація: Значення поза діапазоном $[\text{Q1} - 1.5 \times \text{IQR}, \text{Q3} + 1.5 \times \text{IQR}]$ — потенційні викиди.

Візуалізація Box plots (діаграми розмаху) наочно показують викиди.

Обробка викидів

- **Видалення:** Якщо викид — це помилка або надзвичайне явище

  ```
  df_clean = df[(df['salary'] >= lower_bound) &
  (df['salary'] <= upper_bound)]
  ```

- **Заміна:** На найближче “нормальне” значення (upper/lower bound)

```
df['salary'] = df['salary'].
clip(lower=lower_bound, upper=upper_bound)
```

- **Трансформація:** Логарифмічна або інша трансформація для зменшення впливу

```
df['log_salary'] = np.log(df['salary'])
```

  - Корисна для асиметричних розподілів
  - Викиди стають менш екстремальними у логарифмічній шкалі

- **Ігнорування:** Якщо викид є справжнім явищем та репрезентативним
  - Наприклад, дохід мільйонера в датасеті про доходи

Коли НЕ видаляти викиди:

- Викид є справжнім явищем, важливим для вашої задачі
- Дані зі сфери, де екстремальні значення природні (спорт, фінанси)
- Видалення призведе до зміщення вибірки (bias)

Дедуплікація даних

Дублікати — це однакові або майже однакові рядки у наборі даних. Вони можуть виникати через помилки ETL, багаторазове введення або злиття таблиць.

Виявлення точних дублікатів

```
# Pandas
duplicates = df[df.duplicated(subset=['id', 'name'], keep=False)]
# keep=False показує ВСІ дублікати,
# keep='first' --- лише другі та далі
```

```
# PySpark
df_distinct = df.dropDuplicates(['id', 'name'])
df.groupBy('id', 'name').count().filter(col('count') > 1).show()
```

Видалення дублікатів

```
# Видалити рядки, дублюючі перший
df_clean = df.drop_duplicates(subset=['id', 'name'], keep='first')

# PySpark
df_clean = df.dropDuplicates(['id', 'name'])
```

Нечіткі дублікати (Fuzzy Matching)

Для викриття близьких, але не ідентичних рядків (наприклад, “John Smith” vs “Jon Smit”) використовуються міри подібності:

- **Levenshtein distance:** Кількість редагувань для трансформації одного рядка в інший
- **Cosine similarity:** Подібність текстових векторів

  Потребує спеціальних бібліотек (fuzzywuzzy, textdistance).

Практичні приклади та workflow

Приклад 1: Очищення та трансформація датасету з тестування

Задача: Підготувати датасет тестів співробітників для навчання моделі.

```
# Читання
df = pd.read_csv('test_scores.csv')

# Видалення рядків з critical пропусками
df = df.dropna(subset=['employee_id', 'score'])
```

```
# Заповнення категоріальних пропусків
df['department'].fillna('Unknown', inplace=True)

# Виявлення і обробка викидів (сум > 100 або < 0 --- помилка)
df = df[(df['score'] >= 0) & (df['score'] <= 100)]

# One-Hot Encoding для department
df = pd.get_dummies(df, columns=['department'], drop_first=True)

# Масштабування оцінок
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df['score_scaled'] = scaler.fit_transform(df[['score']])

# Видалення дублікатів по employee_id та date
df = df.drop_duplicates(subset=['employee_id', 'date'],
keep='first')
```

Приклад 2: Merge таблиць з обережністю

```
employees = pd.DataFrame({
    'emp_id': [1, 2, 3],
    'name': ['Alice', 'Bob', 'Charlie']
})

salaries = pd.DataFrame({
    'emp_id': [1, 1, 2, 3],  # emp_id=1 дублюється!
    'salary': [50000, 55000, 60000, 70000]
})

# Inner join без перевірки унікальності призведе до дублювання
result = pd.merge(employees, salaries, on='emp_id', how='left')
# Результат: Alice з'явиться двічі
```

```
# Розв'язання: агрегувати salary перед merge
salaries_agg = salaries.groupby('emp_id')['salary'].mean().
reset_index()
result = pd.merge(employees, salaries_agg, on='emp_id',
how='left')
```

Загальний робочий процес очищення даних

Рекомендована послідовність кроків:

1. **Дослідження (Exploratory Analysis)**
    - `df.info()`, `df.describe()`, `df.isnull().sum()`
    - Візуалізація розподілів, кореляцій
2. **Обробка пропущених значень**
    - Аналіз причин пропусків (MCAR, MAR, MNAR)
    - Вибір методу: видалення або імпутація
3. **Виявлення дублікатів**
    - Перевірка унікальних ключів
    - Видалення точних дублікатів
4. **Обробка викидів**
    - Виявлення (IQR, 3-sigma)
    - Визначення: помилка чи реальне явище
    - Обробка: видалення, заміна або трансформація
5. **Трансформація категоріальних даних**
    - One-Hot Encoding для номінальних
    - Label Encoding для порядкових
6. **Масштабування числових ознак**

- Вибір метода (Min-Max vs Standardization)
- Застосування до всіх числових колонок

7. **Інженерія нових ознак**
    - Створення дерівованих ознак (вік з дати народження)
    - Дискретизація числових ознак
    - Взаємодійні ознаки (feature interactions)
8. **Перевірка якості**
    - Кількість та розподіл пропусків
    - Розподіл кожної ознаки після трансформацій
    - Кореляція між ознаками

Висновки

Ключові концепції лекції:

- **Масштабування та нормалізація** — критичні для чутливих алгоритмів; вибір методу залежить від дистрибуції та наявності викидів
- **One-Hot Encoding** — стандартний метод для номінальних категорій, але потребує обережності при великій кардинальності
- **Label Encoding** — ефективний для tree-based моделей, але небезпечний для лінійних алгоритмів з неупорядкованими категоріями
- **Пропущені значення** — обробка залежить від механізму пропусків (MCAR, MAR, MNAR); імпутація часто переважна над видаленням
- **Викиди** — не завжди помилки; рішення залежить від домену та мети аналізу
- **Дублікати** — фундаментальна проблема якості; видалення поперед моделюванням

- **Merge операції** — потребують уважності щодо унікальності ключів; моніторинг дублювання рядків
- **Дискретизація та feature interactions** — покращують інтерпретабельність та можуть підняти точність нелінійних залежностей

Практичні рекомендації:

- Спочатку розберіть дані: кількість пропусків, розподіл, викиди, дублікати
- Документуйте кожен крок очищення: які дані видалені, чому
- Не залишайте дані на автомат; завжди розумійте наслідки трансформацій
- Перевіряйте результати на кожному етапі: розподіли, статистика, логічні помилки
- Збереження нативних даних для можливості повернення до них

Контрольні запитання

1. Поясніть різницю між масштабуванням (standardization) та нормалізацією (min-max scaling). Коли кожне застосовується?
2. Які ризики дискретизації (binning) неперервних ознак для моделей? Наведіть приклад корисного застосування.
3. Порівняйте One-Hot Encoding і Label Encoding: вплив на модель та випадки некоректного застосування.
4. Які стратегії вибору ознак допомагають уникати надмірної кореляції та шуму?
5. Як виявити пропущені значення, що маскуються специфічними константами (наприклад, -999)?
6. Поясніть різницю між середнім, медіаною і модою як методами імпутації. Як обрати правильний?

7. Що таке IQR і як його використати для виявлення викидів? Наведіть формулу меж.

8. У яких випадках викиди не слід видаляти? Приклад доменної логіки.

9. Поясніть підхід логарифмічної трансформації та її вплив на асиметричні розподіли.

10. Розгляньте задачу створення ознаки ”вік клієнта” з дати народження: які потенційні джерела помилок?

11. Поясніть різницю між вертикальною та горизонтальною конкатенацією таблиць.

12. Коли merge (join) призводить до дублювання рядків і як цього уникнути?

13. Які критерії для вибору видалення рядків з пропусками vs імпутації?

14. Розкрийте ризики надмірної кількості One-Hot колонок (висока кардинальність). Альтернативні методи.

15. Як можна автоматично виявляти дублікати при неідеальній відповідності (fuzzy matching)?

16. Поясніть підхід створення взаємодіючих ознак (feature interactions) та його вплив на модель.

17. Чому надмірна інженерія ознак може призвести до overfitting? Ознаки цього.

18. Які метрики якості даних ви б відстежували у продакшн конвеєрі (data pipeline)?

19. Опишіть процес аудиту джерел даних перед інженерією ознак.

20. Запропонуйте послідовність кроків для систематичного очищення сирого набору даних перед моделюванням.

Лекція 5

Вступ до машинного навчання з MLlib

Побудова конвеєрів (pipelines), основні алгоритми машинного навчання

Що таке Apache Spark та MLlib?

Apache Spark — це високопродуктивний, розподілений обчислювальний фреймворк з відкритим кодом, призначений для обробки великих даних. Він надає прості у використанні API для програмування на Java, Scala, Python та R.

MLlib — це бібліотека машинного навчання, вбудована в Apache Spark. Вона надає інструменти для:

- **Інженерії ознак:** трансформація, вибір та вилучення ознак.
- **Моделювання:** реалізація поширених алгоритмів, таких як класифікація, регресія, кластеризація.
- **Побудови конвеєрів (Pipelines):** об'єднання кількох етапів обробки даних та моделювання в єдиний робочий процес.
- **Збереження та завантаження моделей:** для повторного використання.

MLlib працює з `DataFrame` зі Spark SQL, що дозволяє легко інтегрувати машинне навчання в ETL-процеси.

Основні концепції MLlib

MLlib API базується на кількох ключових абстракціях:

- **DataFrame:** Основна структура даних у Spark, що є розподіленою колекцією даних, організованою в іменовані стовпці. MLlib використовує `DataFrame` для зберігання ознак, міток та передбачень.
- **Transformer:** Алгоритм, який перетворює один `DataFrame` на інший. Наприклад, `Tokenizer` — це `Transformer`, який перетворює стовпець з текстом на стовпець з масивом слів (токенів). Кожен `Transformer` має метод `.transform()`.
- **Estimator:** Алгоритм, який навчається на `DataFrame` для створення `Transformer`. Наприклад, `LogisticRegression` — це `Estimator`, який навчається на даних і створює `LogisticRegressionModel` (який є `Transformer`). Кожен `Estimator` має метод `.fit()`.
- **Pipeline (Конвеєр):** Об'єднує послідовність `Transformer`-ів та `Estimator`-ів у єдиний робочий процес. Це дозволяє автоматизувати всі етапи — від очищення даних до навчання моделі.

Побудова конвеєрів машинного навчання

Конвеєр (Pipeline) — це потужний інструмент для організації та автоматизації складних робочих процесів машинного навчання. Він складається з послідовності етапів (stages), де кожен етап є або `Transformer`, або `Estimator`.

Переваги використання конвеєрів:

- **Структурованість:** Чітко визначає послідовність кроків обробки.
- **Автоматизація:** Весь процес, від підготовки даних до налаштування гіперпараметрів, можна виконати одним викликом.
- **Відтворюваність:** Конвеєр можна зберегти та завантажити, що гарантує однакову обробку даних як під час навчання, так і під час використання моделі.

Приклад етапів у конвеєрі для аналізу текстів:

1. `Tokenizer`: Розбиває речення на слова.
2. `HashingTF`: Перетворює слова на вектори ознак фіксованої довжини.
3. `LogisticRegression`: `Estimator`, що навчає модель класифікації.

Коли ми викликаємо `.fit()` для `Pipeline`, дані проходять через кожен етап послідовно. Всі `Transformer`-и викликають `.transform()`, а `Estimator`-и — `.fit()`. Результатом є `PipelineModel`, який є `Transformer`-ом і може бути використаний для передбачень на нових даних.

Огляд основних алгоритмів MLlib

MLlib підтримує широкий спектр алгоритмів для різних завдань.

Класифікація (Classification)

Завдання полягає у віднесенні об'єкта до однієї з кількох категорій.

- **Логістична регресія (Logistic Regression):** Популярний алгоритм для бінарної та багатокласової класифікації.
- **Дерева рішень (Decision Trees):** Нелінійний алгоритм, що легко інтерпретується.
- **Випадковий ліс (Random Forest):** Ансамбль дерев рішень, що забезпечує вищу точність.

Регресія (Regression)

Завдання полягає у передбаченні неперервного значення.

- **Лінійна регресія (Linear Regression):** Прогнозує значення на основі лінійної комбінації вхідних ознак.
- **Дерева рішень для регресії:** Аналогічні до класифікаційних, але прогнозують числове значення.

Кластеризація (Clustering)

Завдання полягає у групуванні схожих об'єктів без попередньої розмітки.

- **K-Means:** Розділяє дані на k кластерів на основі відстані до центроїдів.
- **Latent Dirichlet Allocation (LDA):** Популярний для тематичного моделювання текстів.

Практичний приклад: класифікація текстів

Розглянемо спрощений приклад побудови конвеєра для класифікації текстових документів (наприклад, спам/не спам).

Вхідні дані: `DataFrame` з двома стовпцями: `text` (текст повідомлення) та `label` (0 або 1).

```
# Приклад на PySpark
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer

# 1. Створення етапів конвеєра
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(),
                      outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)

# 2. Створення конвеєра
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# 3. Навчання моделі (trainingData - це наш DataFrame)
model = pipeline.fit(trainingData)

# 4. Використання моделі для передбачень на нових даних
predictions = model.transform(testData)
```

Цей приклад демонструє, як легко можна об'єднати інженерію ознак (`Tokenizer`, `HashingTF`) та навчання моделі (`LogisticRegression`) в єдиний, відтворюваний процес.

MLlib: Еволюція та Практика

Перехід до DataFrame API

Початкове MLlib (RDD-based) надавало низькорівневі алгоритми (LinearRegression, KMeans) поверх RDD; сучасний акцент — DataFrame Pipelines: **Estimator**, **Transformer**, **Pipeline**, **Evaluator**. RDD-алгоритми поступово вважаються застарілими для нових проєктів.

Pipeline концепти

- Estimator (навчання) → fit() повертає Transformer.
- Transformer → transform(df) додає стовпці (features, predictions).
- ParamGridBuilder + CrossValidator для тюнінгу гіперпараметрів.

Особливості

VectorAssembler, StringIndexer, OneHotEncoder, StandardScaler — забезпечують відтворювані кроки. Модель серіалізується у каталозі.

Приклад (скорочено)

```
from pyspark.ml.feature import VectorAssembler, StringIndexer
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline

indexer = StringIndexer(inputCol="country", outputCol="country_idx")
assembler = VectorAssembler(inputCols=["country_idx","age"],
outputCol="features")
lr = LogisticRegression(featuresCol="features", labelCol="label")
```

```
pipe = Pipeline(stages=[indexer, assembler, lr])
model = pipe.fit(train_df)
pred = model.transform(test_df)
```

DataFrame переваги для ML

Оптимізації Catalyst (фільтри, проєкції), ефективні формати зберігання фіч, інтеграція з Spark SQL для попередньої обробки.

Обмеження

Деякі алгоритми менш багаті на опції порівняно з спеціалізованими бібліотеками (XGBoost, LightGBM). Інтеграція через сторонні пакети.

GraphX та GraphFrames

GraphX

API на Scala для розподілених графових обчислень (RDD-based): оператори mapVertices, joinEdges, алгоритми PageRank, ConnectedComponents. Переваги: масштабованість, гнучкість. Недолік: менше оптимізацій та відсутність повного SQL інтегрування.

GraphFrames

Побудовані поверх DataFrame: вершини (vertices DataFrame) і ребра (edges DataFrame); підтримка motif queries (шаблонів), інтеграція зі Spark SQL. Алгоритми: PageRank, BFS, ConnectedComponents, TriangleCount.

```
from graphframes import GraphFrame
g = GraphFrame(v_df, e_df)
tri = g.triangleCount()
tri.show()
```

Вибір

GraphFrames для потреб SQL/ML інтеграції та швидкої аналітики; GraphX для низькорівневих трансформацій та коли потрібен контроль у Scala.

Обмеження

GraphFrames не завжди покриває всі високоспеціалізовані сценарії; для великих графів можливі обмеження пам'яті.

Висновки

Apache Spark та його бібліотека MLlib є потужним інструментом для побудови масштабованих систем машинного навчання. Використання `DataFrame` та `Pipeline` дозволяє створювати гнучкі, структуровані та ефективні робочі процеси для обробки великих обсягів даних, що робить MLlib стандартом у галузі Big Data.

Ключові переваги MLlib:

- **Масштабованість:** Розподілені алгоритми, що можуть обробляти дані, які не вміщуються в пам'ять однієї машини.
- **Інтеграція:** Вбудована в Spark, що дозволяє легко комбінувати з обробкою даних, SQL-запитами та потоковими обчисленнями.
- **Стандартизація:** Pipeline API забезпечує узгоджену термінологію та інтерфейс для всіх алгоритмів.
- **Гнучкість:** Підтримка різних типів алгоритмів: класифікація, регресія, кластеризація, обробка текстів.
- **Відтворюваність:** Моделі та конвеєри легко зберігаються та завантажуються для подальшого використання.

При виборі MLlib слід враховувати, що для спеціалізованих завдань (наприклад, глибинного навчання) можуть бути більш придатними вузькоспеціалізовані бібліотеки (наприклад, Ray [23]). Однак для задач обробки великих даних

з класичними алгоритмами машинного навчання MLlib залишається відмінним вибором.

GraphX та GraphFrames розширюють можливості Spark для роботи з графовими структурами, забезпечуючи ефективні алгоритми для аналізу мереж та взаємозв'язків у великих графах.

Контрольні запитання та завдання

1. Чим відрізняється Transformer від Estimator у MLlib? Наведіть приклад пари Estimator→Model.

2. Поясніть переваги використання Pipeline порівняно з «ручним» викликом послідовних трансформацій.

3. Опишіть типову структуру конвеєра для задачі класифікації тексту (етапи та їхні ролі).

4. Як забезпечити відтворюваність моделі під час повторного використання PipelineModel у продакшн середовищі?

5. Порівняйте логістичну регресію і дерева рішень для класифікації: сильні та слабкі сторони.

6. Поясніть різницю між бінарною та багатокласовою класифікацією з точки зору підходів оцінки.

7. Які виклики виникають під час використання HashingTF (колізії) і як їх мінімізувати?

8. Коли доцільно використати CountVectorizer замість HashingTF?

9. Поясніть принцип роботи K-Means: роль центроїдів і критерій зупинки.

10. Як інтегрувати інженерію ознак (наприклад, нормалізацію) у Pipeline без втрати узгодженості між train/test?

11. Що таке регуляризація в логістичній регресії і які гіперпараметри її контролюють?

12. Поясніть різницю між fit() конвеєра та fit() окремого Estimator. Що повертається у кожному випадку?

13. Як масштабованість Spark MLlib впливає на вибір алгоритмів порівняно з локальними бібліотеками (scikit-learn)?

14. Опишіть стратегію збереження та завантаження навченого PipelineModel для подальших передбачень.

15. Які критерії вибору алгоритму кластеризації (K-Means vs LDA) для текстових даних?

16. Як додати етап вибору ознак (feature selection) у існуючий конвеєр?

17. Поясніть відмінність між трансформацією ознак і створенням ознак (feature extraction vs feature engineering).

18. Які можливі джерела data leakage у побудові конвеєра і як їх уникнути?

19. Чим відрізняється підготовка даних для регресії від класифікації у MLlib (вимоги до типів даних, масштабування)?

20. Запропонуйте підхід до комбінування декількох текстових полів в єдиний вектор ознак у конвеєрі.

Лекція 6

Застосування моделей регресії та класифікації

Практичні приклади та оцінка моделей

Огляд завдань регресії та класифікації

Як ми дізналися з попередньої лекції, машинне навчання з учителем (Supervised Learning) вирішує два основні типи завдань:

- **Регресія (Regression):** Прогнозування неперервного значення. Наприклад, прогнозування ціни на будинок, температури повітря або попиту на товар.
- **Класифікація (Classification):** Прогнозування дискретної мітки або категорії. Наприклад, визначення, чи є лист спамом, класифікація новин за темами або діагностика захворювання.

Після побудови моделі критично важливо оцінити, наскільки добре вона працює. Для цього існують спеціальні метрики.

Практичний приклад регресії

Завдання: Спрогнозувати вартість автомобіля на основі його характеристик (рік випуску, пробіг, об'єм двигуна).

Етапи (з використанням PySpark MLlib):

1. **Підготовка даних:** Завантажуємо `DataFrame` з даними про автомобілі.
2. **Інженерія ознак:**
    - `VectorAssembler`: Об'єднуємо всі числові ознаки (`year`, `mileage`, `engine_volume`) в один вектор `features`.
3. **Побудова моделі:**
    - `LinearRegression`: Створюємо екземпляр моделі лінійної регресії, вказуючи стовпець ознак (`features`) та цільовий стовпець (`price`).
4. **Навчання та передбачення:**
    - Навчаємо модель на тренувальних даних (`.fit()`).
    - Робимо передбачення на тестових даних (`.transform()`).

```
# Приклад коду на PySpark
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorAssembler

# 1. Підготовка даних (припустимо, 'dataset' вже завантажено)
assembler = VectorAssembler(
    inputCols=["year", "mileage", "engine_volume"],
    outputCol="features")
output = assembler.transform(dataset)

# 2. Розділення даних
(trainingData, testData) = output.randomSplit([0.8, 0.2])

# 3. Побудова та навчання моделі
lr = LinearRegression(featuresCol="features", labelCol="price")
lr_model = lr.fit(trainingData)

# 4. Передбачення
predictions = lr_model.transform(testData)
```

Метрики для оцінки моделей регресії

Після отримання `predictions` ми можемо оцінити якість моделі.

- **Root Mean Squared Error (RMSE):** Корінь із середньоквадратичної помилки. Показує, наскільки в середньому передбачення моделі відрізняються від реальних значень. Чим менше RMSE, тим краще.
- **R-squared (R^2):** Коефіцієнт детермінації. Показує, яку частку варіації цільової змінної пояснює модель. Значення R^2 лежить в діапазоні від 0 до 1. Чим ближче до 1, тим краще модель описує дані.
- **Mean Absolute Error (MAE):** Середня абсолютна помилка. Схожа на RMSE, але менш чутлива до великих помилок (викидів).

  У Spark для цього є `RegressionEvaluator`.

```
from pyspark.ml.evaluation import RegressionEvaluator

evaluator_rmse = RegressionEvaluator(labelCol="price",
                                     predictionCol="prediction",
                                     metricName="rmse")
rmse = evaluator_rmse.evaluate(predictions)
print(f"RMSE: {rmse}")

evaluator_r2 = RegressionEvaluator(labelCol="price",
                                   predictionCol="prediction",
                                   metricName="r2")
r2 = evaluator_r2.evaluate(predictions)
print(f"R-squared: {r2}")
```

Практичний приклад класифікації

Завдання: Класифікувати відгуки на фільми як «позитивні» (1) або «негативні» (0).

Етапи (з використанням PySpark MLlib):

1. **Підготовка даних:** `DataFrame` з колонками `text` та `label`.
2. **Інженерія ознак (конвеєр):**
    - `Tokenizer`: Розбиває текст на слова.
    - `HashingTF`: Перетворює слова на вектори ознак.
3. **Побудова моделі:**
    - `LogisticRegression`: Створюємо модель для бінарної класифікації.
4. **Побудова конвеєра та навчання:**
    - `Pipeline`: Об'єднуємо всі етапи в єдиний процес.
    - Навчаємо конвеєр на тренувальних даних.

```
# Приклад коду на PySpark
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import Tokenizer, HashingTF
from pyspark.ml import Pipeline

# 1. Підготовка даних (припустимо, 'reviews' вже завантажено)
# reviews - DataFrame з колонками 'text' та 'label'

# 2. Інженерія ознак
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashing_tf = HashingTF(inputCol="words", outputCol="features")

# 3. Побудова моделі
lr = LogisticRegression(featuresCol="features", labelCol="label")

# 4. Побудова конвеєра
pipeline = Pipeline(stages=[tokenizer, hashing_tf, lr])

# 5. Навчання моделі
model = pipeline.fit(reviews)
```

Метрики для оцінки моделей класифікації

Для оцінки класифікаторів використовують `BinaryClassificationEvaluator` або `MulticlassClassificationEvaluator`.

- **Accuracy:** Частка правильних прогнозів. $(\text{TP} + \text{TN})/(\text{Всі прогнози})$. Це найпростіша метрика, але вона може бути оманливою на незбалансованих наборах даних.
- **Precision (Точність):** Частка правильних позитивних прогнозів серед усіх, які модель назвала позитивними. $\text{TP}/(\text{TP}+\text{FP})$. Важлива, коли ціна хибно-позитивного результату висока (наприклад, у спам-фільтрах).
- **Recall (Повнота):** Частка знайдених позитивних об'єктів серед усіх реальних позитивних об'єктів. $\text{TP}/(\text{TP} + \text{FN})$. Важлива, коли ціна пропуску позитивного результату висока (наприклад, у діагностиці хвороб).
- **F1-score:** Гармонійне середнє між Precision та Recall. Це хороший баланс між двома метриками.

Матриця помилок (Confusion Matrix)

Це таблиця, яка візуалізує ефективність моделі класифікації. Для бінарного випадку вона має вигляд 2×2:

	Predicted: Positive	**Predicted: Negative**
Actual: Positive	True Positive (TP)	False Negative (FN)
Actual: Negative	False Positive (FP)	True Negative (TN)

- **TP:** Модель правильно передбачила позитивний клас.
- **TN:** Модель правильно передбачила негативний клас.
- **FP (Помилка I роду):** Модель помилково передбачила позитивний клас.
- **FN (Помилка II роду):** Модель помилково пропустила позитивний клас.

Матриця помилок дозволяє глибоко зрозуміти, де саме помиляється ваша модель.

Вибір правильної метрики

Вибір метрики залежить від бізнес-завдання:

- **Збалансовані дані:** `Accuracy` є хорошим початком.
- **Незбалансовані дані:** `Accuracy` не підходить. Використовуйте `Precision`, `Recall`, `F1-score` та `AUC-ROC`.
- **Важливіше не пропустити подію (напр., хворобу):** Оптимізуйте `Recall`.
- **Важливіше не робити хибних тривог (напр., спам):** Оптимізуйте `Precision`.
- **Потрібен баланс:** Використовуйте `F1-score`.

Висновки

Побудова моделі — це лише частина роботи. Правильна оцінка її ефективності за допомогою відповідних метрик є ключовим етапом, який визначає, чи готова модель до використання в реальних умовах. Вибір метрики завжди повинен ґрунтуватися на конкретних вимогах та цілях вашого проєкту.

Основні принципи оцінки моделей:

- **Завжди розділяйте дані** на навчальну та тестову частини, щоб оцінити справжню ефективність моделі на невидимих даних.
- **Використовуйте відповідні метрики** для вашого типу завдання та характеру даних.
- **Аналізуйте матрицю помилок** для глибшого розуміння поведінки моделі.
- **Слідкуйте за переобучанням (overfitting)**, порівнюючи метрики на тренувальних та тестових даних.
- **Контекст має значення** — вибір метрики повинен враховувати особливості та вимоги конкретної задачі.

Контрольні запитання та завдання

1. Поясніть різницю між регресією та класифікацією з точки зору типу вихідної змінної.

2. Чому розділення даних на train/test (або додатково validation) є критичним? Наслідки порушення.

3. Порівняйте метрики похибок (RMSE, MAE): коли кожна з них більш інформативна?

4. Що означає високий R^2, але значна систематична похибка у передбаченнях? Інтерпретація.

5. Наведіть приклад задачі, де оптимізація Precision важливіша за Recall, і навпаки.

6. Як інтерпретувати випадок, коли Precision високий, а Recall низький? Які бізнес-наслідки?

7. В чому відмінність між мікро- та макро- усередненням метрик у багатокласовій класифікації?

8. Побудуйте приклад матриці помилок для моделі з високим класовим дисбалансом. Як її аналізувати?

9. Чим F1-score відрізняється від узагальненого F-score і коли корисно налаштовувати параметри?

10. Поясніть поняття AUC-ROC та його інтерпретацію. Коли крива ROC може вводити в оману?

11. Як впливає дисбаланс класів на Accuracy і які методи корекції існують (oversampling, class weights)?

12. Поясніть покроково створення ознак для текстової класифікації у прикладі лекції.

13. Які ризики використання лінійної регресії на даних з сильними нелінійними взаємодіями ознак?

14. Як інженерія ознак може покращити Recall без значного падіння Precision?

15. Поясніть різницю між in-sample та out-of-sample помилками. Прояв overfitting.

16. Чому важливо фіксувати seed під час розділення даних? Які наслідки варіативності?

17. Як інтерпретувати випадок: низький RMSE, але низький R^2? Можливі причини.

18. Запропонуйте стратегію вибору метрики для моделі прогнозування ціни vs для виявлення шахрайства.

19. Як можна покращити стабільність метрик класифікації при сильному класовому дисбалансі?

20. Опишіть процес порівняння двох моделей: які статистичні тести або методи валідації застосуєте?

Лекція 7

Налагодження та оптимізація Spark застосунків

Моніторинг, профілювання, техніки оптимізації

Навіщо потрібна оптимізація в Spark?

Apache Spark — це потужний інструмент для розподіленої обробки даних, але його стандартні налаштування не завжди є оптимальними для конкретного завдання. **Оптимізація Spark-застосунків** — це процес налаштування коду, конфігурації та архітектури для підвищення продуктивності, зменшення часу виконання та ефективного використання ресурсів кластера (пам'яті, процесора).

Основні цілі оптимізації:

- **Швидкість:** Зменшення часу виконання завдань.
- **Ефективність:** Зменшення витрат на обчислювальні ресурси.
- **Стабільність:** Уникнення помилок, пов'язаних з браком пам'яті (`OutOfMemoryError`) або іншими вузькими місцями.

Моніторинг за допомогою Spark UI

Spark UI — це вбудований вебінтерфейс, який є головним інструментом для моніторингу та діагностики Spark-застосунків. Він доступний за замовчуванням за адресою `http://<driver-node>:4040`.

Ключові вкладки Spark UI:

- **Jobs:** Показує список всіх дій (actions), які були запущені в застосунку.
- **Stages:** Кожна робота (Job) розбивається на етапи (Stages). Тут можна побачити деталі кожного етапу, включаючи залежності та причину його створення (наприклад, через shuffle).
- **Tasks:** Кожен етап складається з завдань (Tasks), які виконуються паралельно на екзекуторах. Ця вкладка дозволяє аналізувати тривалість, метрики зчитування/запису та можливі відхилення в часі виконання окремих завдань.
- **Storage:** Відображає інформацію про RDD та DataFrame, які були закешовані в пам'яті або на диску.
- **Executors:** Надає зведену інформацію про всі екзекутори, включаючи використання пам'яті, диска та кількість виконаних завдань.
- **SQL:** Показує деталізований план виконання запитів (фізичний та логічний), що є критично важливим для оптимізації Spark SQL.

Аналіз ключових метрик

- **Job (Робота):** Створюється на кожну дію (action) в коді, наприклад `collect()`, `count()`, `save()`. Довгий час виконання роботи може свідчити про неефективну обробку.
- **Stage (Етап):** Роботи діляться на етапи на межах перемішування даних (shuffle). Велика кількість етапів або тривалі етапи часто вказують на інтенсивний shuffle.

- **Task (Завдання):** Найменша одиниця роботи, що виконується на одному ядрі екзекутора. Якщо одне завдання виконується значно довше за інші в межах одного етапу, це може свідчити про нерівномірний розподіл даних (Data Skew).

Поширені проблеми продуктивності

- **Shuffle (Перемішування):** Це процес перерозподілу даних між екзекуторами. Він є дуже витратним, оскільки вимагає дискового I/O та мережевої передачі. Мінімізація shuffle — одна з головних задач оптимізації.
- **Data Skew (Нерівномірний розподіл даних):** Ситуація, коли одна або кілька партицій значно більші за інші. Це призводить до того, що завдання, які обробляють ці партиції, виконуються набагато довше, створюючи «пляшкове горло».
- **Memory Pressure (Проблеми з пам'яттю):** Неправильне керування пам'яттю може призвести до частих збірок сміття (Garbage Collection) або помилок `OutOfMemoryError`. Це часто трапляється під час кешування великих обсягів даних або під час shuffle.

Основні техніки оптимізації

Серіалізація даних (Kryo)

За замовчуванням Spark використовує стандартну серіалізацію Java, яка є гнучкою, але повільною. **Kryo** — це альтернативний, значно швидший і компактніший фреймворк серіалізації.

```
# Увімкнення Kryo
spark.conf.set("spark.serializer",
               "org.apache.spark.serializer.KryoSerializer")
```

Керування партиціями та перемішуванням

Кількість партицій впливає на рівень паралелізму. Занадто мало партицій — ресурси кластера не використовуються повністю. Занадто багато — накладні витрати на керування завданнями зростають.

- `repartition()`: Збільшує або зменшує кількість партицій, що завжди викликає повний shuffle.
- `coalesce()`: Ефективно зменшує кількість партицій, уникаючи повного shuffle, якщо це можливо.

Кешування (Caching) та персистенція (Persistence)

Якщо ви плануєте використовувати один і той самий `DataFrame` кілька разів, його варто закешувати, щоб уникнути повторних обчислень.

```
df.cache() # або df.persist(StorageLevel.MEMORY_ONLY)
df.count()  # Перший раз df буде обчислено і закешовано
df.show()   # Другий раз дані будуть взяті з кешу
```

Використання Broadcast-змінних

При об'єднанні (join) великого `DataFrame` з маленьким, Spark може автоматично «транслювати» (broadcast) менший `DataFrame` на всі екзекутори. Це дозволяє уникнути дорогого shuffle великого `DataFrame`. Іноді це потрібно робити вручну.

```
from pyspark.sql.functions import broadcast

# small_df буде розіслано на всі вузли
large_df.join(broadcast(small_df), "common_key")
```

Оптимізація на рівні коду та запитів (Catalyst Optimizer)

Spark SQL має потужний оптимізатор запитів **Catalyst**. Він автоматично переписує ваш код (SQL-запити або операції з `DataFrame`) у більш ефективний

фізичний план виконання. Наприклад, він може:

- **Змінювати порядок фільтрації та об'єднань:** застосовувати фільтри якомога раніше (*Predicate Pushdown*).
- **Оптимізувати проєкції:** зчитувати тільки ті стовпці, які потрібні для запиту (*Column Pruning*).

Розуміння плану виконання (через `df.explain()`) дозволяє побачити, як Catalyst оптимізував ваш запит, і виявити можливі проблеми.

Профілювання та налагодження

- **Аналіз логів:** Логи екзекуторів та драйвера містять детальну інформацію про помилки та попередження (наприклад, про тривалі паузи на Garbage Collection).
- **Виявлення Data Skew:** У Spark UI на вкладці Stages можна побачити, що одне або кілька завдань виконуються значно довше за інші. Це можна виправити за допомогою «соління» ключів (salting) або використання `repartition` за іншим ключем.

Керування Ресурсами та Тюнінг

Executor параметри

spark.executor.memory, spark.executor.cores — баланс між паралелізмом та GC. Надто великі heap — тривалий GC; надто малі — часті spill.

Driver пам'ять

spark.driver.memory — збільшити якщо виконуються великі collect()/broadcast; уникати надмірних локальних операцій.

Dynamic Allocation

spark.dynamicAllocation.enabled — автоматичне додавання/видалення executor-ів залежно від черги завдань; потребує external shuffle service.

Speculative Execution

spark.speculation=true — повторний запуск повільних tasks; ефективно при розрізненій продуктивності вузлів, але збільшує ресурсні витрати.

Паралелізм

spark.sql.shuffle.partitions (SQL); spark.default.parallelism (RDD). Зменшувати для малих джерел, збільшувати для широких обчислень.

GC Тюнінг

Використання G1GC (Java 11+) або CMS для зменшення пауз; -XX:+UseG1GC -XX:InitiatingHeapOccupancyPercent; моніторинг через GC логи.

Memory Fraction

spark.memory.fraction та spark.memory.storageFraction впливають на розподіл між виконанням (execution) та кешем (storage).

Діагностика та Усунення Проблем

GC Overhead

Симптом: тривалі паузи, низька пропускна здатність. Метрики: GC time

Shuffle Spill

Спілінг на диск під час сортування або агрегації (значення у інтерфейсі: Shuffle Write, Spill). Рішення: збільшити executor memory, використати більш селективні фільтри, застосувати reduceByKey.

Data Skew

Ознаки: один task триваліший за інші, великі spill. Рішення: salting, AQE skew join, попередня агрегація.

Out Of Memory (OOM)

Driver OOM під час collect(); executor OOM через великі shuffle блоки. Рішення: уникати collect(), збільшити memory, оптимізувати партиції.

Повільні UDF

Проблема: Python UDF руйнує whole-stage codegen. Рішення: вбудовані функції, SQL вирази, pandas UDF (vectorized) за потреби.

Checkpoint vs Cache

Cache для повторних читань у job; checkpoint для усунення довгого lineage (fault tolerance міцніше). Не плутати з streaming checkpoint.

Best Practices

- Використовуйте DataFrame API замість низькорівневих RDD.
- Ранні фільтри та вибір лише потрібних стовпців.
- Кешуйте часто використовувані DataFrame.
- Уникайте `collect()` на великих даних — використовуйте `show()`, `take()`.

Типові помилки

- Надмірне використання UDF замість вбудованих функцій (втрата оптимізацій).
- Невдале налаштування кількості партицій (перевантаження / дрібність).
- Збір великих наборів на драйвер.

Оптимізація продуктивності

- Ефективні формати: Parquet, ORC (колонкові, компресія, predicate pushdown).
- Мінімізація shuffle: правильний вибір агрегацій (`reduceByKey` > `groupByKey`).
- Тюнінг пам'яті executor'ів та кількості cores.

Моніторинг

Spark UI Доступний на порті 4040: *Jobs, Stages, Executors, Storage, SQL.*

Логування Налаштування рівня: `spark.sparkContext.setLogLevel('WARN')`.

Метрики Shuffle read/write, spills, serialization/deserialization time.

Висновки

Оптимізація Spark-застосунків — це ітеративний процес, що вимагає глибокого розуміння як самих даних, так і внутрішньої роботи фреймворку. Використання Spark UI для моніторингу, виявлення вузьких місць та застосування відповідних технік оптимізації дозволяє значно підвищити продуктивність та стабільність систем обробки великих даних.

Ключові напрями оптимізації:

- **Зменшення shuffle:** Це одна з найдорожчих операцій у розподіленій обробці. Правильний вибір алгоритмів агрегації та об'єднань може мати великий вплив.
- **Ефективна серіалізація:** Перехід на Kryo замість стандартної Java серіалізації може значно пришвидшити передачу даних.
- **Кешування та персистенція:** Повторне використання проміжних результатів дозволяє уникнути пересування одних і тих самих обчислень.
- **Керування партиціями:** Правильна кількість партицій забезпечує баланс між паралелізмом та накладними витратами на керування завданнями.

- **Broadcast-змінні:** Розсилання малих наборів даних на всі вузли дозволяє уникнути дорогого shuffle під час об'єднань.
- **Використання Catalyst Optimizer:** DataFrame API та Spark SQL автоматично оптимізуються Catalyst, що дозволяє генерувати ефективніший код ніж низькорівневі RDD операції.
- **Моніторинг та діагностика:** Spark UI є незамінним інструментом для виявлення вузьких місць та розуміння характеру проблем у застосунку.

Контрольні запитання та завдання

1. Поясніть послідовність переходу від дії (action) до створення Job, Stages та Tasks. Що формує межі стадій?
2. Які показники у Spark UI сигналізують про Data Skew і як його усувати?
3. Порівняйте `repartition()` та `coalesce()`: механіка виконання і сценарії використання.
4. Чому shuffle є дорогим? Перерахуйте джерела накладних витрат (мережа, диск, сортування).
5. Як визначити оптимальну кількість партицій для великого набору даних? Критерії та емпіричні правила.
6. Поясніть різницю між caching та persistence. Коли використовувати серіалізовані рівні (`MEMORY_ONLY_SER`)?
7. Що таке broadcast join? Які умови ухвалення рішення про його застосування автоматично чи вручну?
8. Наведіть приклад негативних наслідків неефективної серіалізації (Java vs Kryo).
9. Як інтерпретувати довгий Task Deserialization Time? Причини і дії.
10. Поясніть механізм Garbage Collection впливу на продуктивність Spark. Як його діагностувати?

11. Які оптимізації Catalyst застосовує до DataFrame/SQL запитів і як їх перевірити через `explain()`?

12. Запропонуйте стратегію зменшення обсягу shuffle для багатьох join операцій.

13. Як керувати розміром executor memory та cores для балансування паралелізму і GC пауз?

14. Поясніть техніку salting ключів і приведіть приклад для усунення Data Skew.

15. Які ризики надмірного кешування і як відстежити непотрібні об'єкти у Storage вкладці?

16. Як профілювати виконання окремого запиту Spark SQL для виявлення вузьких місць?

17. Поясніть різницю між локальним режимом (local) та кластерним у контексті діагностики.

18. Які симптоми memory spill і як їх усунути (налаштування та зміни коду)?

19. Порівняйте вплив формату зберігання (Parquet vs CSV) на продуктивність читання і predicate pushdown.

20. Запропонуйте поетапний чекліст оптимізації нового ETL Spark-процесу перед продакшном.

Лекція 8

Розгортання Spark-застосунків

Режими розгортання (Standalone, YARN, Kubernetes), конфігурація, best practices

Архітектура Spark-застосунку

Будь-який Spark-застосунок складається з трьох основних компонентів, що працюють у координації:

- **Driver (Драйвер):** Процес, що виконує `main()` функцію вашого застосунку та створює `SparkContext`. Драйвер відповідає за перетворення коду користувача на завдання (tasks) та їх розподіл між виконавцями. Він також координує загальне виконання.
- **Executors (Виконавці):** Процеси, що запускаються на робочих вузлах (worker nodes) кластера. Вони виконують завдання, надіслані драйвером, і повертають результати. Кожен виконавець має власну пам'ять (для кешування даних) та ядра процесора.
- **Cluster Manager (Менеджер кластера):** Зовнішній сервіс, що відповідає за виділення ресурсів (процесорів, пам'яті) на кластері для Spark-застосунку. Spark підтримує кілька менеджерів: Standalone, Apache YARN та Kubernetes.

Режими розгортання: `client` vs `cluster`

Існує два основних режими, що визначають, де саме буде працювати драйверний процес:

Client mode (`-deploy-mode client`)

- Драйвер запускається на тій самій машині, з якої було відправлено застосунок (наприклад, ваш ноутбук або edge-node кластера).
- **Переваги:** Ідеально для інтерактивної роботи (Spark Shell, Jupyter) та налагодження, оскільки результати та логи одразу доступні на клієнті.
- **Недоліки:** Якщо клієнтська машина вимкнеться або втратить зв'язок з кластером, застосунок завершиться з помилкою. Не підходить для довготривалих завдань у продакшені.

Cluster mode (`-deploy-mode cluster`)

- Драйвер запускається як один із процесів на робочому вузлі всередину кластера. Його життєвим циклом керує менеджер кластера.
- **Переваги:** Надійність. Застосунок продовжить працювати, навіть якщо клієнт, що його запустив, відключиться. Це стандарт для продакшн-середовища.
- **Недоліки:** Налагодження складніше, оскільки логи драйвера зберігаються на одному з вузлів кластера, а не на клієнті.

Standalone режим

Це найпростіший менеджер кластера, що постачається разом зі Spark.

- **Архітектура:** Складається з одного або кількох `Master`-вузлів та `Worker`-вузлів. `Master` відповідає за розподіл застосунків по `Worker`-ах, а `Worker`-и запускають `Executor`-и.

- **Переваги:**
  - Дуже легко налаштувати.
  - Чудово підходить для невеликих кластерів, навчання та розробки.
- **Недоліки:**
  - Відсутні розширені можливості керування ресурсами, такі як черги, пріоритети або ізоляція ресурсів для різних користувачів.
  - За замовчуванням, `Master` є єдиною точкою відмови (SPOF), хоча можна налаштувати високу доступність за допомогою ZooKeeper.

Розгортання на Apache YARN

YARN (Yet Another Resource Negotiator) — це менеджер ресурсів з екосистеми Hadoop. Протягом довгого часу це був де-факто стандарт для розгортання Spark у великих корпоративних середовищах.

Як це працює:

1. Клієнт відправляє застосунок до YARN `ResourceManager`.
2. `ResourceManager` запускає `ApplicationMaster` (AM) на одному з вузлів.
3. В `cluster` режимі Spark-драйвер працює всередині AM. В `client` режимі AM лише запитує ресурси для `Executor`-ів.
4. AM веде переговори з `NodeManager`-ами на робочих вузлах для запуску контейнерів, в яких працюватимуть `Executor`-и.

Переваги:

- Надійність та зрілість.
- Централізоване керування ресурсами для різних типів застосунків (Spark, MapReduce, Hive тощо).
- Підтримка черг, пріоритетів та контролю доступу.

Розгортання на Kubernetes

Kubernetes (K8s) — це сучасна платформа для оркестрації контейнерів, яка стала популярним вибором для розгортання Spark-застосунків.

Як це працює:

1. `spark-submit` спілкується з Kubernetes API-сервером для створення Spark-драйвера, який працює в `Pod`-і.
2. Драйвер, у свою чергу, створює `Pod`-и для `Executor`-ів.
3. Kubernetes відповідає за мережеву взаємодію, виділення ресурсів та моніторинг `Pod`-ів.

Переваги:

- **Гнучкість та ізоляція:** Кожен застосунок може мати власні версії бібліотек та залежностей завдяки контейнеризації (Docker).
- **Ефективне використання ресурсів:** Kubernetes може динамічно масштабувати кількість вузлів у хмарних середовищах.
- **Єдина інфраструктура:** Дозволяє запускати Spark-завдання на тій самій інфраструктурі, що й інші мікросервіси.

Порівняння режимів та рекомендації

Критерій	Standalone	YARN	Kubernetes
Простота налаштування	Висока	Середня	Складна
Керування ресурсами	Базове	Розширене	Розширене
Ізоляція залежностей	Немає	Обмежена	Повна (Docker)
Екосистема	Тільки Spark	Hadoop	Хмарні застосунки

Рекомендоване використання:

- **Standalone:** Розробка, невеликі кластери.
- **YARN:** Великі on-premise кластери з Hadoop.
- **Kubernetes:** Хмарні середовища, мікросервіси.

Практичні аспекти: використання `spark-submit`

`spark-submit` — це основний інструмент командного рядка для запуску Spark-застосунків на кластері.

Приклад для YARN у `cluster` режимі:

```
spark-submit \
  --class com.example.MySparkApp \
  --master yarn \
  --deploy-mode cluster \
  --executor-memory 1G \
  --num-executors 10 \
  my-app.jar
```

Приклад для Kubernetes у `cluster` режимі:

```
spark-submit \
  --class com.example.MySparkApp \
  --master k8s://https://<k8s-api-server> \
  --deploy-mode cluster \
  --conf spark.kubernetes.container.image=my-spark-image:latest \
  --conf spark.executor.instances=5 \
  local:///opt/spark/jars/my-app.jar
```

Ключові параметри:

- `-master`: Вказує менеджер кластера (`yarn`, `k8s://...`, `spark://...`).
- `-deploy-mode`: `client` або `cluster`.
- `-class`: Головний клас вашого застосунку.
- `-num-executors`, `-executor-memory`, `-executor-cores`: Параметри для конфігурації ресурсів виконавців.

Вибір версій Java та Spark

Правильний вибір версій Java (JDK) та Apache Spark є критичним для стабільності, продуктивності та безпеки продакшн-середовищ.

Java LTS (Long-Term Support)

Apache Spark працює на віртуальній машині Java (JVM), тому вибір версії Java безпосередньо впливає на роботу кластера.

- **Java 8:** Довгий час була стандартом для Big Data екосистеми (Hadoop, Spark). Проте, вона застаріла і нові версії Spark поступово припиняють її підтримку.
- **Java 11 LTS:** Стандарт для більшості сучасних розгортань Spark 3.x. Забезпечує кращу продуктивність та безпеку порівняно з Java 8.
- **Java 17 LTS:** Підтримується починаючи зі Spark 3.3. Рекомендується для нових проектів, оскільки пропонує значні покращення в роботі збирача сміття (Garbage Collector), зокрема ZGC та Shenandoah, що є критичним для Spark-застосунків з великим обсягом пам'яті.
- **Java 21 LTS:** Підтримка впроваджується в найновіших версіях Spark (Spark 4.0+).

Рекомендація: Для нових розгортань Spark 3.3+ варто використовувати **Java 17**. Це забезпечує баланс між стабільністю та використанням сучасних оптимізацій JVM.

Версії Apache Spark

Spark не має офіційної політики «LTS» у тому ж сенсі, що й Java, але спільнота підтримує останні кілька мінорних релізів (наприклад, 3.3.x, 3.4.x, 3.5.x) випусками з виправленнями помилок (maintenance releases).

Рекомендації щодо вибору:

- **Використовуйте останні Maintenance Release:** Завжди обирайте останню версію в межах гілки (наприклад, 3.5.1 замість 3.5.0), оскільки вона містить критичні виправлення помилок та вразливостей.
- **Сумісність екосистеми:** Перевіряйте сумісність обраної версії Spark з іншими компонентами вашого стеку (Hadoop/YARN, Hive, Delta Lake, Iceberg, Airflow).

- **Scala та Python:** Звертайте увагу на версії Scala (2.12 vs 2.13) та Python. Для Python рекомендується використовувати актуальні версії (3.9+), оскільки старі версії швидко втрачають підтримку бібліотек (Pandas, NumPy).

Безпека та Управління Доступом

Аутентифікація

Kerberos (Hadoop/YARN інтеграція) для аутентифікації сервісів; у Kubernetes — сервіси кластеру + секрети контейнерів.

Авторизація

ACL для Spark UI (конфігурації spark.acls.enable, spark.ui.view.acls, spark.ui.admin.acls). Ролі через інтеграцію з Ranger / Sentry (в Hive/таблицях).

Шифрування

TLS між компонентами (spark.ssl.*), шифрування shuffle (spark.network.crypto.enabled), шифрування зберігання на HDFS (Transparent Data Encryption).

Audit Логування

Логи подій Spark (History Server) + системне логування операцій (YARN RM logs). Для Lakehouse: Delta transaction log — джерело аудиту змін.

Конфіденційні дані

Masking / tokenization перед завантаженням; мінімізація використання collect(); контроль доступу до таблиць external metastore.

Висновки

Вибір режиму розгортання Spark-застосунку є критично важливим архітектурним рішенням.

- **Standalone** ідеально підходить для швидкого старту та розробки.
- **YARN** залишається надійним вибором для інтеграції з існуючою Hadoop-екосистемою.
- **Kubernetes** пропонує найбільшу гнучкість, ізоляцію та інтеграцію в сучасні хмарні інфраструктури, що робить його стратегічним вибором для нових проектів.

Розуміння переваг та недоліків кожного підходу дозволяє будувати стабільні та ефективні системи обробки даних.

Контрольні запитання та завдання

1. Поясніть ролі Driver, Executor і Cluster Manager. Як вони взаємодіють під час подачі завдання?
2. Чим відрізняються режими `client` та `cluster` з точки зору надійності, логування і сценаріїв використання?
3. Опишіть послідовність кроків запуску Spark-додатку через `spark-submit` у YARN cluster mode.
4. Які обмеження Standalone режиму у корпоративних середовищах і як їх частково пом'якшити?
5. Порівняйте архітектуру розгортання на YARN і Kubernetes: розподіл відповідальностей та ізоляція.
6. Чому Kubernetes забезпечує кращу ізоляцію залежностей порівняно з YARN? Приклад практичної вигоди.
7. Поясніть вибір параметрів `-num-executors`, `-executor-cores`, `-executor-memory` для CPU-bound і IO-bound задач.

8. Як визначити потребу в спеціалізованих налаштуваннях драйвера (`-driver-memory`, `-driver-cores`)?

9. Що таке dynamic allocation і коли його варто увімкнути/вимкнути?

10. Які безпекові аспекти слід врахувати під час розгортання Spark на Kubernetes (image hardening, secrets)?

11. Поясніть різницю між фізичним масштабуванням кластера і логічним збільшенням партицій даних.

12. Як обрати між Parquet і ORC у продакшн конвеєрі для Spark SQL аналітики?

13. Що таке ApplicationMaster у YARN і як він взаємодіє зі Spark-драйвером у різних режимах?

14. Наведіть сценарій, де `client` mode є більш доцільним за `cluster` навіть у великому кластері.

15. Як забезпечити спостережуваність (observability) Spark-застосунку у Kubernetes: інструменти та метрики.

16. Поясніть механізм розподілу контейнерів у Kubernetes для executor pods та вплив на мережеву топологію.

17. Які практики управління ресурсами запобігають oversubscription ядрами на YARN?

18. Складіть чекліст налаштувань перед розгортанням критичного Spark ETL процесу у продакшн.

19. Як тестувати конфігурації (sizing) локально перед переносом у кластер? Обмеження такого тестування.

20. Запропонуйте стратегію міграції застосунку з YARN на Kubernetes з урахуванням різниць у конфігурації.

Словник ключової термінології

Apache Kafka Розподілена система керування подіями та потоками даних з високою пропускною здатністю.

Apache Spark Високопродуктивний, уніфікований аналітичний рушій для обробки великих даних та машинного навчання.

Availability (Доступність) Властивість системи (у контексті CAP-теореми), що гарантує, що кожен запит отримує відповідь.

BSP (Bulk Synchronous Parallel) Модель паралельних обчислень, що складається з суперкроків: локальні обчислення, обмін даними та бар'єрна синхронізація.

Cache (Кеш) Швидка пам'ять, що зберігає часто використовувані дані для скорочення часу доступу.

Catalyst Optimizer Оптимізатор запитів у Spark SQL, що автоматично переписує логічні плани виконання у більш ефективні фізичні плани.

Cloud Computing (Хмарні обчислення) Надання обчислювальних ресурсів, зберігання та програмного забезпечення через інтернет.

Concurrency (Конкурентність) Здатність системи виконувати кілька операцій одночасно або перепліти їх виконання.

Containerization (Контейнеризація) Упакування застосунку та його залежностей у контейнер для забезпечення портативності.

DAG (Directed Acyclic Graph) Спрямований ациклічний граф, який Spark будує для відстеження послідовності трансформацій та оптимізації плану виконання.

DataFrame Основний API в Spark для роботи зі структурованими даними; розподілена колекція даних, організована в іменовані стовпці.

Deadlock (Взаємне блокування) Стан, коли два або більше процесів чекають один на одного, не звільняючи свої ресурси.

Docker Платформа для контейнеризації, що дозволяє упаковувати застосунки та залежності в контейнери.

Durability (Стійкість) Властивість, що гарантує, що після завершення операції дані залишаються збережені навіть у разі відмов.

ETL (Extract-Transform-Load) Процес вилучення даних з джерела, їх трансформації та завантаження в пункт призначення.

GraphFrames API Spark для роботи з графами як DataFrame, що забезпечує SQL-інтеграцію.

gRPC Сучасний фреймворк для розробки розподілених систем з використанням Protocol Buffers та HTTP/2.

Hadoop Екосистема для розподіленої обробки великих даних, включаючи MapReduce та HDFS.

HDFS (Hadoop Distributed File System) Розподілена файлова система для зберігання великих файлів на кластерах.

Immutable (Незмінний) Властивість, що означає об'єкт не може бути змінений після створення.

Isolation Level (Рівень ізоляції) Ступінь, в якому одна транзакція ізольована від впливу інших одночасних транзакцій.

Join (Об'єднання) Операція об'єднання двох таблиць або наборів даних за спільним ключем.

Kubernetes (K8s) Платформа для оркестрації контейнерів, автоматизації розгортання, масштабування та керування.

Leader Election (Вибір лідера) Процес вибору одного вузла в якості координатора для управління системою.

MapReduce Парадигма та фреймворк для розподіленої обробки великих обсягів даних.

Metadata (Метадані) Дані про дані; інформація, що описує структуру, формат та атрибути основних даних.

Microservice (Мікросервіс) Архітектурний підхід, де застосунок будується як набір малих, незалежних, розподілених сервісів.

MIMD (Multiple Instruction, Multiple Data) Тип паралельної архітектури, де кілька процесорів одночасно виконують різні інструкції над різними даними.

MLlib Бібліотека машинного навчання, вбудована в Apache Spark, що надає інструменти для моделювання.

Monolithic Architecture (Монолітна архітектура) Архітектура, де весь застосунок розроблюється як єдине ціле без розділення на модулі.

Node (Вузол) Окремий комп'ютер або сервер у розподіленій системі.

Normalization (Нормалізація) Процес масштабування даних до діапазону для покращення якості навчання моделі.

Orchestration (Оркестрація) Автоматизоване управління розгортанням та виконанням складних систем.

Parquet Стовпцевий формат зберігання, оптимізований для аналітичних запитів.

Partition (Розділ) Логічна або фізична поділ даних для паралельної обробки.

Pod (у Kubernetes) Найменша розгортаємо одиниця в Kubernetes, що містить один або більше контейнерів.

PRAM (Parallel Random Access Machine) Абстрактна модель паралельних обчислень з N процесорами та спільною пам'яттю.

PySpark Python API для Apache Spark, що дозволяє розробникам використовувати Spark для розподіленої обробки.

Race Condition (Гонка даних) Стан, коли результат операції залежить від непередбачуваного порядку виконання потоків.

Raft Consensus Алгоритм консенсусу для досягнення згоди в розподілених системах.

RDD (Resilient Distributed Dataset) Фундаментальна структура даних в Spark, незмінна, відмовостійка, розподілена колекція об'єктів.

Replication (Реплікація) Копіювання даних на кілька вузлів для забезпечення відмовостійкості та доступності.

REST (Representational State Transfer) Архітектурний стиль для веб-сервісів, що використовує стандартні HTTP методи.

Schema (Схема) Структурне визначення таблиці або даних, що описує імена та типи стовпців.

Serverless Computing Модель хмарних обчислень, де розробник не керує інфраструктурою.

Shard (Фрагмент) Один з розділів розподіленої БД, що зберігається на окремому сервері.

Shuffle Процес перегрупування даних між вузлами для операцій типу join або groupBy.

SIMD (Single Instruction, Multiple Data) Тип паралельної архітектури, де одна інструкція застосовується до кількох елементів даних.

Spark Context Основний об'єкт Spark-застосунку, що представляє з'єднання з кластером.

Spark Session У Spark 2.0+, основний об'єкт для роботи з DataFrame та Spark SQL.

Spark SQL Компонент Apache Spark для роботи зі структурованими даними та виконання SQL-запитів.

SQL (Structured Query Language) Мова запитів для роботи з реляційними БД.

Stream Processing (Потокова обробка) Обробка даних у вигляді неперервного потоку замість пакетів.

Throughput (Пропускна здатність) Кількість роботи, виконаної за одиницю часу.

Transaction (Транзакція) Послідовність операцій, що виконуються як єдине атомарне дію.

Transformer (Трансформатор) У MLlib, об'єкт, що перетворює один DataFrame на інший.

YARN (Yet Another Resource Negotiator) Менеджер ресурсів з екосистеми Hadoop для розгортання Spark-застосунків.

Zookeeper Централізований сервіс координації для управління конфігурацією та синхронізацією.

Інженерія ознак (Feature Engineering) Процес трансформації вихідних даних у набір ознак, які краще представляють проблему для моделі машинного навчання.

Алгоритми консенсусу (Consensus Algorithms) Методи, які дозволяють розподіленим системам досягти згоди щодо стану навіть за наявності вад компонентів.

Відмовостійкість (Fault Tolerance) Здатність системи продовжувати роботу у разі відмови компонентів через реплікацію або контрольні точки.

Драйвер (Driver Program) Процес, що виконує main() функцію Spark-застосунку та координує виконання завдань на кластері.

Дії (Actions) У Spark, операції, що запускають обчислення та повертають результат драйверу.

Екзекутор (Executor) Процес, запущений на робочому вузлі, що виконує завдання та зберігає дані в пам'яті або на диску.

Класифікація (Classification) Завдання машинного навчання, що полягає у віднесенні об'єкта до однієї з категорій.

Кластеризація (Clustering) Завдання машинного навчання без вчителя, що полягає у групуванні схожих об'єктів без попередньої розмітки.

Конвеєр (Pipeline) У MLlib, інструмент, що об'єднує послідовність етапів обробки даних та моделювання в єдиний робочий процес.

Консистентність (Consistency) Властивість системи (у контексті CAP-теореми), що гарантує, що всі вузли бачать однакові дані в один і той же час.

Ліниві обчислення (Lazy Evaluation) Стратегія в Spark, за якою трансформації не обчислюються доти, доки їх результат не стане дійсно необхідним.

Масштабованість (Scalability) Здатність системи збільшувати продуктивність при додаванні обчислювальних ресурсів.

Паралелізм (Parallelism) Одночасне виконання кількох задач або частин однієї задачі для прискорення обчислень.

Регресія (Regression) Завдання машинного навчання, що полягає у передбаченні неперервного числового значення.

Розподіленість (Distributedness) Властивість системи, в якій компоненти розташовані на різних вузлах, що координують свої дії шляхом обміну повідомленнями.

Синхронізація (Synchronization) Забезпечення правильного порядку виконання операцій для уникнення deadlocks та race conditions.

Стійкість до розділення (Partition Tolerance) Властивість системи (у контексті CAP-теореми), що дозволяє їй працювати під час розривів мережі.

Трансформації (Transformations) У Spark, ліниві операції, що створюють новий RDD або DataFrame з існуючого.

Узгодженість в кінцевому рахунку (Eventual Consistency) Модель слабкої узгодженості, яка гарантує, що репліки даних з часом збігаються до однакового стану.

Список джерел

[1] M. Richards та N. Ford, *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, 2020, ISBN: 978-1-492-04345-4.

[2] N. Ford, M. Richards, P. Sadalage та Z. Dehghani, *Software Architecture: The Hard Parts: Modern Distributed Architectures in a Complex World*. O'Reilly Media, 2021, ISBN: 978-1-492-08689-5.

[3] E. A. Brewer, «Towards Robust Distributed Systems,» в *Proc. of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, New York, NY, USA, 2000, с. 7.

[4] S. Gilbert та N. A. Lynch, «Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services,» *ACM SIGACT News*, т. 33, № 2, с. 51—59, 2002. DOI: 10.1145/564585.564601

[5] A. Xu, *System Design Interview – An Insider's Guide: Volume 2*. Independently published, 2021, ISBN: 979-8775642875.

[6] D. Ongaro та J. K. Ousterhout, «In Search of an Understandable Consensus Algorithm,» в *USENIX Annual Technical Conference (ATC)*, Philadelphia, PA, 2014, с. 305—319.

[7] L. Lamport, «The Part-Time Parliament,» *ACM Transactions on Computer Systems*, т. 16, № 2, с. 133—169, 1998. DOI: 10.1145/279227.279229

[8] M. P. Herlihy та J. M. Wing, «Linearizability: A Correctness Condition for Concurrent Objects,» *ACM Transactions on Programming Languages and Systems*, т. 12, № 3, с. 463—492, 1990. DOI: 10.1145/78969.78972

[9] A. Bellemare, *Building Event-Driven Microservices: Leveraging Organizational Data at Scale*. O'Reilly Media, 2020, ISBN: 978-1-492-05789-5.

[10] M. J. Fischer, N. A. Lynch та M. S. Paterson, «Impossibility of Distributed Consensus with One Faulty Process,» *Journal of the ACM*, т. 32, № 2, с. 374—382, 1985. DOI: 10.1145/3149.214121

[11] G. M. Amdahl, «Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,» *AFIPS Conference Proceedings*, т. 30, с. 483—485, 1967. DOI: 10.1145/1465482.1465560

[12] J. L. Gustafson, «Reevaluating Amdahl's Law,» *Communications of the ACM*, т. 31, № 5, с. 532—533, 1988. DOI: 10.1145/42411.42415

[13] K. Indrasiri та D. Kuruppu, *gRPC: Up and Running: Building Cloud Native Applications with Go and Java*. O'Reilly Media, 2020, ISBN: 978-1-492-05833-5.

[14] A. S. Tanenbaum та M. van Steen, *Distributed Systems: Principles and Paradigms*, 3rd. Prentice Hall, 2016, ISBN: 978-0-13-439242-9.

[15] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017, ISBN: 978-1-491-92383-0.

[16] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996, ISBN: 978-1-55860-348-6.

[17] B. Burns, J. Beda, K. Hightower та L. Evenson, *Kubernetes: Up and Running: Dive into the Future of Infrastructure*, 3rd. O'Reilly Media, 2022, ISBN: 978-1-098-11020-8.

[18] B. Ibryam та R. Huß, *Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications*, 2nd. O'Reilly Media, 2023, ISBN: 978-1-492-08345-0.

[19] N. Narkhede, G. Shapira та T. Palino, *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*, 2nd. O'Reilly Media, 2021, ISBN: 978-1-492-04308-9.

[20] Y. Brikman, *Terraform: Up & Running: Writing Infrastructure as Code*, 3rd. O'Reilly Media, 2022, ISBN: 978-1-098-11674-3.

[21] B. Chambers та M. Zaharia, *Spark: The Definitive Guide: Big Data Processing Made Simple*. O'Reilly Media, 2018, ISBN: 978-1-491-91205-8.

[22] D. J. Abadi, «The Case for Serverless Data Systems,» *arXiv*, т. 1812.11582, 2018.

[23] M. Pumperla, E. Oakes та R. Liaw, *Learning Ray: Flexible Distributed Python for Machine Learning*. O'Reilly Media, 2023, ISBN: 978-1-098-11722-1.