МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
УМАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ САДІВНИЦТВА
Факультет економіки і підприємництва
Кафедра інформаційних технологій

Допущено до захисту
"_____" _____ 2023 р.
Зав. кафедри, к.т.н., доцент
_____ **Ліщук Р.І.**

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ ДЛЯ ОБРОБКИ ВЕЛИКИХ ДАНИХ

Виконав – студент 2 курсу, групи 21м-кн
освітньої програми "Комп’ютерні науки"
спеціальності 122 Комп’ютерні науки
Шидловський Євген Вячеславович

Науковий керівник – к.т.н., доцент
Ліщук Роман Ігорович

Умань – 2023

Зміст

ВСТУП 7

РОЗДІЛ 1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАДАЧІ 11

1.1. Методи та техніки аналізу великих даних 11

1.2. Поняття мікросервісної архітектури 26

1.3. Постановка задачі 32

РОЗДІЛ 2 ТЕХНОЛОГІЇ ОБРОБКИ ТА АРХІТЕКТУРА ДЛЯ РОЗРОБКИ МІКРОСЕРВІСІВ З ОБРОБКИ ВЕЛИКИХ ДАНИХ 34

2.1. Технології обробки і методи аналізу великих даних 34

2.2. Алгоритми кластеризації великих даних 43

2.3. Архітектура мікросервісу з обробки великих даних 52

РОЗДІЛ 3 РОЗРОБКА МІКРОСЕРВІСІВ З ОБРОБКИ ВЕЛИКИХ ДАНИХ 55

3.1. Обґрунтування вибору мови реалізації та стеку технологій 55

3.2. Реалізація мікросервісів з обробки великих даних 56

ВИСНОВКИ 61

СПИСОК ЛІТЕРАТУРИ 62

ДОДАТОКИ 64

ПЕРЕЛІК СКОРОЧЕНЬ

IT – інформаційні технології

ПП – програмний продукт

СУБД – система управління базами даних

ЦОД – центр обробки даних

ACID – atomicity, consistency, isolation, durability

AI – artificial intelligence

ANN – approximate nearest neighbor

API – application programming interface

BI – business intelligence

CAP – consistence, availability, partition tolerance

DB – database

DevOps – development and operations

ETL – extract, transform, and load

FRiS – function of rival similarity

GQL – graph query language

HCM – Hard C-Means

HDFS – Hadoop distributed file system

HOLAP – hybrid online analytical processing

HPCC – high-performance computing cluster

HTTP – HyperText transfer protocol

JSON – JavaScript object notation

JVM – Java virtual machine

KEEL – knowledge extraction based on evolutionary learning

LSH – locality-sensitive hashing

MOLAP – multidimensional online analytical processing

NoSQL – not only SQL

OLAP – online analytical processing

PDF – portable document format

QoS – quality of service

RDF – resource description framework

REST – representational state transfer

ROLAP – relational online analytical processing

SPARQL – SPARQL protocol and RDF query language

SQL – structured query language

SSD – solid-state drive

SSH – secure shell

XML – extensible markup language

ВСТУП

В сучасних умовах розвитку інформаційних технологій та комп'ютеризації населення постає проблема збору та обробки великих даних. Проблема збору великих даних полягає в необхідності розподіленого зберігання даних без порушення їхньої цілосності. Проте її вже можна вважати вирішеною, оскільки існує достатня кількість розподілених систем керування базами даних. Зате ще довго не можна буде вважати вирішеною проблему обробки великих даних, тому що довгий час алгоритми з обробки даних створювались без передбачення можливості розподілених обчислень, а тому їхнє застосування до великих даних унеможливлювало швидку обробку цих даних.

Однозначно можна сказати, що загальна кількість даних у світі, що потребує обробки, не буде зменшуватись із часом. Відповідно поширеність систем з обробки великих даних буде лише зростати, як і попит на розробку таких систем. Також вже можна із впевненістю сказати, що архітектура монолітних систем програмного забезпечення буде відходити у минуле, а на його місце стане мікросервісна архітектура. Отже зросте потреба різних мікросервісів з обробки великих даних: від примітивних до надскладних. Наразі розробка навіть найпростішого мікросервісу з обробки великих даних не по кишені малому бізнесу, який, з огляду на його поширеність, приречений на популяризацію розробок у сфері великих даних.

Для цього потрібно спростити та пришвидшити розробку простих мікросервісів, які покликані стати початком поширення програмного забезпечення з обробки великих даних у всі галузі підприємництва.

Об’єктом дослідження є проблема обробки великих даних.

Предметом дослідження є спрощення та прискорення розробки нескладних мікросервісів з обробки великих даних.

Мета роботи – розробити бібліотеку для розробки мікросервісів з обробки великих даних.

Методами дослідження є методи обробки великих даних.

Шляхи подальшого розвитку – розробка повноцінного фреймворку для створення мікросервісно-орієнтованих додатків з обробки великих даних.

РОЗДІЛ 1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Методи та техніки аналізу великих даних

Поняття великих даних (англ. Big Data) у галузі інформаційних технологіях охоплює такі великі обсяги інформації (як структурованої, так і ні), до яких не можна застосувати класичні методи обробки, які базуються переважно на досвіді класу бізнес-аналітики. У різних галузях та контекстах терміном Big Data можуть називати не тільки значні обсяги даних, але і групу технологій та методів, які призначені для аналізу та обробки таких масивів даних, як систематизованих, так і несистематизованих, для отримання якісно нових знань. Загалом поняття Big Data означає інформацію, яка не піддається обробці традиційними способами через її величезний об'єм.

Щоб охарактеризувати великі дані, використовують ознаки "п’ять V":

1) Volume – обсяг. Об’єм інформації у накопиченій базі даних настільки значний, що здійснити його обробку або зберігати за допомогою традиційних методів практично нереально. Для таких обсягів інформації необхідні модернізовані інструменти та принципово новий підхід.

2) Velocity – швидкість. Ця ознака характеризує великі дані з двох аспектів: по-перше, швидкість накопичення інформації постійно зростає; по-друге, швидкість обробки інформації відрізняється від традиційних методів. В умовах та викликах цифрового світу все частіше потребуються такі методи технології, які дають змогу використовувати обробку даних у режимі реального часу.

3) Variety – різноманітність. Big Data охоплює структуровані, так і неструктуровані дані. Структуровану інформацію можна класифікувати за певним критерієм, відповідно, такий вид інформації легко та швидко піддається машинній обробці. Неструктурована інформація містить різні масиви та види даних: фото, аудіо, відео, тексти та інші. Один із влучних прикладів неструктурованих даних – масив інформації з соціальних мереж.

Неструктурована інформація складає близько 80 % від усієї інформації, а також вимагає додаткової обробки та аналізу перед її використанням. Відповідно технології Big Data дозволяють одночасно оброблювати як структуровані, так і неструктуровані дані.

4) Veracity – істинність. Big Data охоплює масиви даних, обсяг яких постійно збільшується. Тому важливим етапом у роботі з такими даними є виділення правдивих, достовірних даних. Якість зафіксованої інформації може мати разючі відмінності, тим самим впливаючи на точний аналіз.

5) Variability – змінність, мінливість. Невідповідність інформації в умовах постійного зростання її обсягу ускладнює та інколи перешкоджає процесам обробки та управління даними.

Головний принцип роботи Big Data пов’язує обсяг даних з точністю прогнозу: чим більшим обсягом інформації ми володіємо, тим більш точний прогноз можливо зробити. Big Data також дає змогу порівнювати певні дані, аналізувати взаємозв'язки між ними, щоб відкрити приховані до цього закономірності. Такі особливості забезпечують комплексне глибинне розуміння проблеми та, в кінцевому результаті, дозволяє прийти до необхідного рішення, або дає можливість керувати потрібними процесами.

Процес обробки величезних масивів даних переважно починається з побудови моделей. Наступний крок – запуск симуляцій. На цьому етапі постійно змінюються ключові налаштування, при цьому система відстежує, який вплив мають ці зміни на можливий результат. Ці процеси проходять в автоматичному режимі, аж доки не буде виокремлено ключовий момент, який дає змогу розв’язати поставлену задачу.

Неструктурована інформація складає переважну більшість від усіх даних (до 80 %). Для їх перетворення та подальшої обробки використовують найсучасніші методи та технології аналізу. Серед таких можна назвати машинне навчання та штучний інтелект (AI).

Поява нового поняття і нових технологій закономірно потребує розробки класифікації способів аналізу, що застосовуються в межах цього поняття. Одну з таких класифікацій пропонує Міжнародна консалтингова компанія McKinsey. Спеціалізація компанії – задачі, що стосуються стратегічного управління. У класифікації McKinsey пропонується 11 методів і технік аналізу, якими можна використовувати при роботі з Big Data.

1) Методи класу Data Mining. Охоплюють отримання даних, а також їх інтелектуальний та глибинний аналіз. Це група методів, за допомогою яких можливо виявити раніше невідомі, незвичайні корисні знання у аналізованих даних. Отримане знання в майбутньому є необхідним для прийняття рішень. До таких методів, зокрема, належать: навчання асоціативним правилам, класифікація (поділ масиву даних на класи/категорії), кластерний аналіз, регресійний аналіз, виокремлення і аналіз відхилень та інші.

2) Краудсорсинг. Краудсорсинг передбачає передачу деяких функцій і задач іншим особам без вступу у трудові взаємини. Таким чином, у Big Data завдання з класифікації і приросту даних виконує невизначене коло осіб, які здійснюють цю роботу без укладання трудового договору.

3) Злиття та інтеграція даних. Це набір технік, які дають можливість інтегрувати різнорідні дані з великої кількості джерел для подальшого глибинного аналізу. Прикладом може слугувати обробка живої людської мови, яка охоплює як семантику, так і тональність мовлення.

4) Машинне навчання. Цей метод включає процес як за участі вчителя, так і без нього, і застосовується для отримання комплексних прогнозів. Для цього використовують моделі, які були сконструйовані на базі статистичного аналізу чи машинного навчання.

5) Штучні нейронні мережі, мережевий аналіз, оптимізація, у тому числі генетичні алгоритми. Ця група методів пов’язана з т. зв. евристичними алгоритмами пошуку, які застосовують для вирішення завдань оптимізації і моделювання. При цьому використовують способи випадкового підбору,

комбінацій і варіювання необхідних показників. Використовувані у цих процесах механізми відповідають природному відбору.

6) Розпізнавання образів.

7) Прогнозна аналітика.

8) Імітаційне моделювання. Цей метод дає можливість конструювати моделі для такого опису процесів, який максимально відповідає реальним умовам. Імітаційне моделювання можна аналізувати в контексті різновиду експериментальних випробувань.

9) Просторовий аналіз. Група методів, у яких за даних виокремлюють і застосовують інформацію топологічного, геометричного і географічного характеру.

10) Статистичний аналіз, аналіз часових рядів, A/B-тестування. Цей метод стосується дослідження в галузі маркетингу. Він передбачає наявність контрольної групи елементів та декількох тестових груп. У кожній з тестових груп змінюють певні характеристики (одну чи декілька). Потім кожну з тестових груп порівнюють з контрольною групою. Таким чином виявляють, як зміни впливають на цільовий показник, і які характеристики варто змінювати для покращення результату.

11) Візуалізація аналітичних даних. Цей метод передбачає наочне оформлення інформації у схеми, діаграми або малюнки. Візуалізовані дані доповнюються анімаційними елементами і можливостями інтерактивної взаємодії. Цей метод застосовують і для того, щоб прийти до результату, і для того, щоб оформити вихідну інформації для наступного аналізу. Візуалізація і унаочнення є важливим етапом аналізу великих даних, оскільки візуальне оформлення дає змогу продемонструвати найважливіші результати проведеного аналізу у найбільш комфортній для сприйняття формі.

В умовах глобалізації та цифровізації суспільства інформація перетворилася на цінний продукт, з чого випливає, що дані сьогодні є таким же значущим чинником виробництва, як і трудові ресурси чи виробничі активи.

Зокрема, про це йдеться у звіті "Global Institute, Big data: The next frontier for innovation, competition, and productivity" компанії McKinsey. Залучення до виробництва великих обсягів даних надає компаніям такі конкурентні переваги, які можуть істотно повипливати на ситуацію на ринку.

Існує низка процесів і завдань, до розв’язання і здійснення яких ефективно залучати технології Big Data:

– прогнозувати ситуацію на ринку;
– організовувати маркетинг продукту і оптимізувати продажі;
– удосконалювати та модернізувати продукцію;
– стверджувати адміністративні рішення;
– покращувати продуктивність праці;
– підвищувати ефективність логістика;
– стежити за станом основних фондів на регулярній основі;

Генерація великих даних на виробничих підприємствах відбувається зокрема у результаті залучення технологій Промислового інтернету речей. Ця технологія полягає в тому, що ключові вузли і деталі верстатів і машин оснащуються давачами, виконавчими пристроями, контролерами та, зрідка, недорогими процесорами, які мають можливість виконувати граничні (туманні) обчислення. Таким чином, протягом виробничого процесу постійно збираються дані. За можливості оснащення виконує їх попереднє опрацювання (на кшталт фільтрації, відбору певного виду даних тощо). Аналітичні платформи опрацьовують результати у найзручнішому для сприйняття вигляді і зберігають для наступного використання для потреби. Висновки на базі проаналізованих даних використовують для оцінки стану обладнання, визначення ефективності коригування або модернізації технологічного процесу тощо.

Моніторинг та відстеження даних у режимі реального часу виконує такі завдання:

– скорочення кількості простоїв;

– підвищення продуктивності обладнання;
– зменшення витрачених коштів на експлуатацію;
– зниження ризику нещасних випадків.

Одним із найважливіших завдань є запобігання нещасним випадкам. Розглянемо роботу операторів на заводах та фабриках нафтопереробної промисловості. Вони отримують приблизно півтори тисячі аварійних повідомлень за робочу зміну, тобто два-три повідомлення на хвилину. Таке навантаження спричинює пришвидшену втомлюваність працівників. Однак операторам потрібна постійна увага і концентрація для того, щоб схвалювати миттєві рішення щодо реакції платформа на певний сигнал. При залученні до роботи аналітичної платформи другорядна і неважлива інформація буде відділена. Таким чином оператори можуть першочергово сконцентруватися на критичних ситуаціях, що дасть змогу більш ефективно виявити чи попередити аварії або нещасні випадки. Отже, ми маємо підвищений рівень надійності виробництва та промислової безпеки, а також готовності технологічного обладнання та відповідності нормативним вимогам. Також аналіз Big Data дозволяє розрахувати час, за який окупляться витрати на обладнання, визначити доцільність змін технологічних режимів тощо – тобто ухвалювати стратегічні рішення щодо розвитку підприємства.

Концепції роботи з великими даними:

Концепція NoSQL призначена для обробки як реляційних, так і нереляційних даних. Прихильниками концепції NoSQL підкреслюється, що вона не є повним запереченням мови SQL і реляційної моделі. Мова SQL – це корисний інструмент, однак має свої вади, а саме – неуніверсальність. Коли при роботі з традиційними реляційними базами даних залучають інформацію величезного обсягу або проекти з великим навантаженням, можуть виникати проблеми та збої. Таким чином, використання NoSQL розширює можливості баз даних в тих галузях, де не вистачає гнучкості. При цьому NoSQL не витісняє SQL з тих сфер, де SQL успішно порається з поставленими задачами.

Базу NoSQL складають такі чинники:

- нереляційна модель даних;
- розподіленість;
- відкритий вихідний код;
- гарна горизонтальна масштабованість.

Для того, щоб методологічно обґрунтувати конецпцію NoSQL, застосовують евристичний принцип, знаний як теорема CAP. Абревіатура CAP містить три складники: consistence – узгодженість, availability – доступність, partition tolerance – стійкість до розподілу. Теорема CAP стверджує, що неможливо одночасно забезпечити ці три складники для розподіленої системи на окремі частини. Тобто, якщо метою є високий рівень доступності та протистояння до розподілу, то при цьому неможливо сконцентруватися на засобах забезпечення узгодженості даних, які зазвичай провадяться класичними SQL-орієнтованими системами керування базами даних з транзакційними механізмами на засадах ACID.

Розглянемо нестроге доведення теореми CAP, яке базується на простих міркуваннях. Нехай розподілена система утворена з певної кількості серверів. Кожен сервер опрацьовує певну кількість клієнтських запитів. Протягом обробки конкретного запиту сервер зобов’язаний гарантувати, що у відповіді на посланий запит міститься актуальна інформація. Для цього завчасно необхідно синхронізувати вміст власної серверної бази з іншими серверами. Тобто, витрачається час на очікування повної синхронізації. Інакше сервер генерує відповідь, ґрунтуючись на несинхронізованих даних. Третій варіант передбачає часткову синхронізацію, якщо абсолютна з певних причин неможлива. Всі три випадки порушують тріаду CAP: у першому дані не доступні, у другому немає узгодженості, у третьому система не стійка до розподілу.

Виділяють чотири категорії NoSQL баз даних.

1. Key-Value (Ключ-Значення) бази даних. Ці бази вирізняються простотою ідеї. Вони являють собою дуже великі хеш-таблиці, де кожний ключ

має задане відповідне значення. Перевагою Key-Value баз є можливість працювати з будь-якими обсягами даних. Недолік – істотні обмеження в мові запитів. Серед представників баз Ключ-Значення є такі: Dynomite, Voldemort, Tokyo, Redis.

2. Клони Bigtable. Базу даних BigTable створила корпорація Google для задоволення власних внутрішніх потреб. BigTable має вигляд таблиці, де крім стовпців і рядків присутній третій вимір – часові відмітки. Така архітектура дозволяє досягти підвищеної продуктивності. Також таблиця такого типу гарно масштабується на велику кількість пристроїв. Однак варто зауважити, що така база даних є нереляційною. Відповідно вона не може забезпечити велику кількість переваг реляційних баз даних, таких як операції з'єднання чи складні запити. Google створили Bigtable для себе і не поширюють її на зовнішній ринок, що спричинило появу незалежно розроблених клонів: Hadoop, Hypertable і Cassandra.

3. Документо-орієнтовані бази даних. Вони мають деякі спільні риси із базами даних типу Ключ-Значення. Їх ключова відмінність – в метаданих чи словниках бази даних значення є чітко окресленими та визначеними. У більшості випадків значенням виступає певний документ або об'єкт, до структури якого є можливість здійснювати запити. Серед таких баз даних: CouchDB і MongoDB.

4. Бази даних, сконструйовані на графах. Вони призначені для підтримування складних зв'язків між об'єктами і базуються на теорії графів. Структура даних – набір вузлів, які пов'язані один з одним за допомогою посилань. При цьому й вузли, й посилання можуть мати певну кількість атрибутів. Серед баз даних, змодельованих на графах, є такі: Neo4j, AllegroGraph, Sones graphDB.

Виділяють також і п'яту категорію, але вона не належить до NoSQL, а саме – об'єктно-орієнтовані бази даних. Вони перш за все призначені для підтримування об'єктно-орієнтованої парадигми програмування. Такі бази

даних вирізняються простотою у використанні у мовах програмування, які підтримують цю парадигму.

Розглянемо механізми для доступу до інформації в NoSQL базах даних.

1) Restful інтерфейси

Цей підхід полягає в тому, що кожний об’єкт, на який ми маємо змогу здійснювати вплив, володіє унікальною адресою. При звертанні за цією адресою ми можемо запитувати, створювати, редагувати або видаляти вказаний об’єкт. При цьому сервер не зберігає жодного стану. Це означає, що кожний запит обробляється ізольовано від інших запитів. Ці інтерфейси мають спільні риси з основним протоколом інтернету – HTTP.

2) Мови запитів, що відрізняються від SQL

Серед них є такі: GQL – SQL-видна мова для Google Bigtable, Gremlin – мова обходу графів, SPARQL – мова запитів Семантичного Веба, Sones Graph Query Language – мова запитів до Sones Graph.

3) API запити

Google Bigtable Datastore API, Neo4j Traversal API.

Серед переваг концепції NoSQL – високий рівень експлуатаційної готовності та коректності, однак ключовим фактором називають рівень продуктивності. За цим показником NoSQL бази даних істотно перевищують реляційні бази даних у своїй ніші. Донедавна реляційні бази були єдиним видом баз даних для різноманітних ситуації. З поширенням NoSQL ситуація стає якісно іншою: тепер для кожного окремого випадку потрібно обирати власну модель даних. Інколи до роботи необхідно залучати водночас декілька баз даних, щоб використати в роботі ключові переваги кожної з них. Ще одна істотна сильна сторона NoSQL баз даних – велика кількість представників цього сімейства сховищ даних реалізовані як проекти з відкритим кодом.

Таким чином, наявність різноманітних категорій баз даних NoSQL потребує формального опису моделей даних, які ними оброблюються.

Загалом порівняння SQL і NoSQL подано в таблиці 1.1.

Таблиця 1.1 – Порівняння SQL та NoSQL

Критерій	SQL	NoSQL
Зберігання даних	Зберігаються в реляційній моделі в рядках і колонках.	У множині баз даних кожна має різноманітні моделі зберігання. Ключові: документ, графік, ключ-значення.
Схеми і гнучкість	Кожний запис відповідає визначеній схемі. При зміні схеми змінюється вся база даних. Здійснити зміну можна оффлайн.	Схеми являють собою динамічну інформацію, яку можна доповнити в будь-який момент часу.
Масштабованість	Вертикальна.	Горизонтальна.
Відповідність ACID	Більшість реляційних баз даних відповідають ACID.	Залежно від технології. Багато рішень у галузі NoSQL не дотримуються відповідності ACID, аби зберегти показники масштабованості та продуктивності.

Аналіз програмного забезпечення для роботи з великими даними

Незважаючи на те, що концепцію великих даних активно застосовують для бізнесу та на виробництві, теоретична база концепції досі не є достатньо окресленою. Проаналізувавши наявні дослідження, матеріали з теми у науково-популярних журналах, а також блогові фахові статті, можемо виокремити такі ключові точки:

- джерела великих даних;
- апаратне забезпечення та інфраструктура;
- програмне забезпечення і збереження;

– інформаційні технології (методи і засоби оброблення даних);
– застосування великих даних, бізнес-аналіз.

Джерелами Big Data можуть виступати як пристрої, так і людський ресурс. У першому випадку йдеться про дані, отримані з галузі промисловості, напрацювання масштабних проектів на кшталт Великого адронного коллайдера в ЦЕРН, тощо. Якщо говоримо про людський ресурс як джерело Big Data, то перш за все маємо на увазі дані соцмереж, охорони здоров'я тощо.

Для опрацювання масивів великих даних варто застосовувати технології хмарних обчислень. Хмарні обчислення – це новітня парадигма для розташування кластерів даних і надання різноманітних послуг місцевою або глобальною мережею Інтернет. Хостинг кластерів даних дає клієнтам можливості збереження та обчислення величезних обсягів даних у хмарі.

Позаяк обсяги певних баз даних збільшуються з величезною швидкістю швидко і долають петабайтний бар'єр (насамперед це стосується баз даних соцмереж), то онлайн-обробка в режимі OLAP таких масивів є практично нереальною (рис. 1.1).

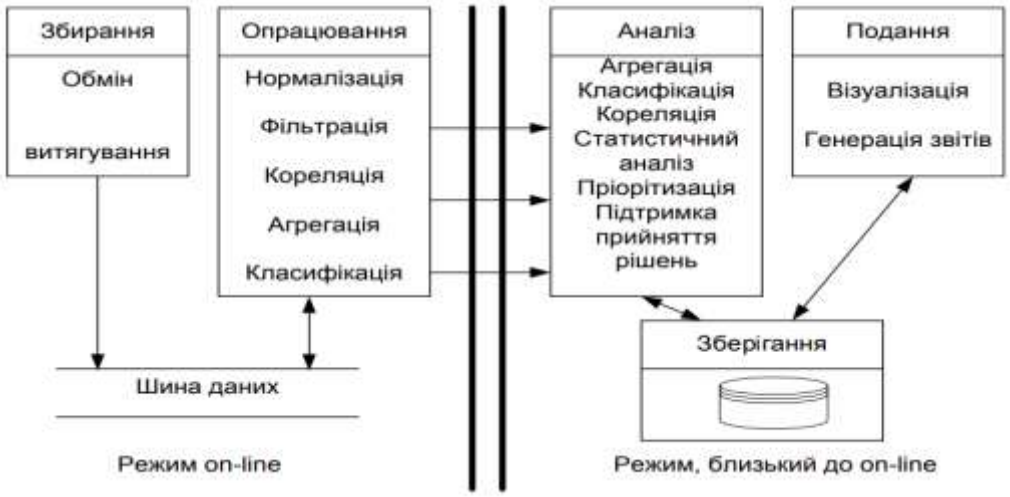


Рисунок 1.1 – Порівняльна характеристика OLAP та великих даних

В таблиці 1.2 організовано характеристику щодо низки інструментів для обробки великих даних з відкритим вихідним кодом, які поставляються через інфраструктури хмарних обчислень. Значна частина цих інструментів забезпечується Apache і випускається під ліцензією Apache. Платформи

розділені на категорії за типами завдань, що постають під час обробки великих даних.

Таблиця 1.2 – Інструменти для великих даних

Інструменти для великих даних	**Характеристика**
Аналіз даних	
Ambari http://ambari.apache.org	Інструмент web для надання послуг, керування та моніторингу Apache Hadoop кластерів
Avro http://avro.apache.org	Система серіалізації інформації
Chukwa http://incubator.apache.org/chukwa	Система колекціонування даних для управління значними за обсягом розподіленими системами
Hive http://hive.apache.org/	Інфраструктура сховища даних, яка забезпечує агрегацію даних
Pig http://pig.apache.org	Високорівнева мова потоків даних і виконуваний framework для паралельних підрахунків
Spark http://spark.apache.org	Швидкий і генеральний обчислювач для даних Hadoop. Забезпечує просту і виразну модель програмування, яка підтримує широкий спектр додатків, зокрема ETL, автоматичного навчання, опрацювання потоків
ZooKeeper http://zookeeper.apache.org/	Високопродуктивна служба координації для розподілених додатків

Продовження таблиці 1.2

Actian http://www.actian.com/	Зберігає масив первинних даних і готує дані для наступного аналізу
HPCC http://hpccsystems.com	Забезпечує швидке перетворення, одночасне опрацювання для застосувань з великими даними
Data Mining	
Orange http://orange.biolab.si	Унаочнення та аналіз даних для новачка і експертів
Mahout http://mahout.apache.org	Бібліотека засобів автоматичного навчання та отримання даних
KEEL http://keel.es	Еволюційний алгоритм для проблем отримання даних
Для соціальних мереж	
Apache Kafka	Платформа з підвищеною пропускною здатністю для обробки даних в режимі реального часу
BI	
Talend http://www.talend.com	Інтеграція даних, керування, інтеграція додатків, засоби і сервіси для великих даних
Jedox http://www.jedox.com/en	Функції аналізу, звітності, планування
Pentaho http://www.pentaho.com	Інтеграція даних, бізнес-аналіз, візуалізація даних, прогноз
Rasdaman http://rasdaman.eecs.jacobsuniversity.de/	Багатовимірні расторові дані без обмежень щодо розміру, наявності мови запитів

Продовження таблиці 1.2

Apache Lucene http://lucene.apache.org	Додаток для повнотекстової індексації та пошуку
Apache Solr http://lucene.apache.org/solr	Повнотекстовий пошук, фасетний пошук, динамічна кластеризація, формати документів типу Word, PDF, просторовий пошук
Elasticsearch http://www.elasticsearch.org	Засіб розподіленого повнотекстового пошуку з веб-інтерфейсом і JSON документами
MarkLogic http://developer.marklogic.com	NoSQL і XML база даних
mongoDB http://www.mongodb.org	Крос-платформенна документо-орієнтована система керування базами даних, що підтримує JSON та динамічні схеми
Cassandra http://cassandra.apache.org	Масштабована мультимайстерна база даних без однієї точки відмови
HBase http://hbase.apache.org	Масштабована розподілена база даних з підтримкою структурованого збереження даних значного обсягу
InfiniteGraph http://www.objectivity.com	Розподілена графова база даних

Перетворення систем баз даних NoSQL різноманітних категорій у інший формат супроводжується в результаті деякими проблемами. При чому ці

проблеми є спільними для систем різних категорій: класу ключ-значення", колонкових або документних.

Сконструювати в автоматичному режимі семантично значуще відбиття в об’єктну модель наразі фактично нереально. Причина полягає в тому, що схеми у NoSQL можуть реалізуватися у коді програми, тобто бути присутніми неочевидно. Таким чином необхідно долучати до роботи експерта для того, щоб виявити схеми даних для кожної узятої бази. Досить коректним є шлях побудови відображення на рівні додатку, що опрацьовує базою даних, пропускаючи відображення моделей системи баз даних. Проте у цьому процесі можуть трапитися деякі перешкоди. Неможливо використати цей спосіб, якщо додаток закритий для зовнішніх запитів, які не надають довільного доступу до даних. Додаток також може бути розділений на частини, кожна з яких працює з різними підмножинами даних з одних і тих самих баз. Ще одна перешкода – одночасно з однією базою може взаємодіяти декілька додатків.

Ключова причина перешкод, що постають під час семантичного відображення моделей даних NoSQL в об’єктну модель – низький або недостатній рівень структури. Такі бази переважно явно не визначають схему даних. Значення у зв’язках ключ-значення почасти є довільними і неструктурованими, а ключі виступають у ролі і елементів структури, і самих даних. Відсутність схеми може спричинити динамічну зміну структури пар. Тобто при операції оновлення є можливість у будь-який момент задати нову структуру пар ключ-значення. При цьому бази даних в більшості ситуацій не мають обмежень щодо кількості пар в структурах, довжини ключів тощо.

Такі системи доцільно застосовувати для розв’язання задач, пов’язаних із використанням нефіксованих і необмежених слабоструктурованих даних. Однак значна кількість програм використовує системи баз даних класу NoSQL, щоб зберігати достатньо жорстко типізовані структури. На обидва випадки слід зважати під час відображення в об’єктну модель

У процесі аналізу семантики даних слід акцентувати на тому, як використовуються в ключах фіксовані (імена елементів структур) або нефіксовані (дані) значення. При виокремленні статичної схеми даних фіксовані значення ключа перетворюються в атрибути типів, нефіксовані – в значення атрибутів.

У процесі відображення організовуються такі структури:

– фреймові структури для вкладених пар;
– абстрактні типи даних для ситуації, якщо дані підпадають виявленню чіткої структури.

Необхідно завчасно означити та виокремити усі ймовірні варіації запитів, які застосовуються при розв'язанні задач над даними в моделях NoSQL. Ті дані, за якими слід організувати пошук, мають бути виявленими ключами в деяких парах ключ-значення, а дані, які шукають – значеннями. Щоб забезпечити такі умови, формують додаткові пари над основними даними з відповідною структурою даних. Ці пари у процесі відображення не утворюють нові структури, а відображаються в типи і класи, організовані основними парами. Відповідно до допоміжних пар у цільовій моделі з'являються вторинні ключі. Запити з порівнянням по атрибутам, що є ключами в допоміжних парах, при зворотному відображенні відображаються в операції над цими парами.

1.2. Поняття мікросервісної архітектури

Мікросервісна архітектура, або просто мікросервіси, є відмінним методом розробки програмних систем, які намагаються зосередитися на побудові однофункціональних модулів з чітко визначеними інтерфейсами та операціями. Тенденція здобула популярність в останні роки, оскільки підприємства стають більш гнучкими і рухаються до DevOps і постійного тестування.

Мікросервіси мають багато переваг для команд Agile і DevOps – як зазначає Мартін Фаулер, Netflix, eBay, Amazon, Twitter, PayPal і інші технічні

зірки еволюціонували від монолітної до мікросервісної архітектури. На відміну від мікросервісів, монолітний додаток побудовано як єдиний автономний пристрій. Це робить зміни в додатку повільними, оскільки це впливає на всю систему. Модифікація невеликої секції коду може потребувати створення і розгортання абсолютно нової версії програмного забезпечення. Масштабування певних функцій програми також означає, що необхідно масштабувати весь додаток.

Мікросервіси усувають ці проблеми монолітних систем, будучи максимально модульними. У простій формі вони допомагають створити додаток як набір невеликих сервісів, кожен з яких працює у власному процесі і може бути незалежно розгорнутим. Ці сервіси можуть бути написані на різних мовах програмування і можуть використовувати різні методи зберігання даних. У той час як це призводить до розробки систем, які є масштабованими і гнучкими. Мікросервіси часто підключаються через API і можуть використовувати багато з тих же інструментів і рішень, які виросли в екосистемі RESTful і веб-сервісів. Тестування цих API-інтерфейсів може допомогти перевірити потік даних і інформації в рамках розгортання мікросервіса.

Характеристики мікросервісів:

1) Наявність декількох компонентів

Програмне забезпечення, створене у вигляді мікросервісів, за визначенням можна розбити на багатокомпонентні сервіси. Таким чином, кожен з цих сервісів можна розгорнути, налаштувати, а потім повторно розгорнути незалежно від інших, не ставлячи під загрозу цілісність програми. В результаті може виникнути потреба змінити тільки один або кілька окремих сервісів замість того, щоб заново розгортати цілі програми. Але у цього підходу є свої недоліки, зокрема дорогі віддалені виклики (замість внутрішньопроцесних викликів) і підвищена складність при перерозподілі обов'язків між компонентами.

2) Бізнесорієнтованість

Стиль мікросервісів зазвичай організований навколо ділових можливостей і пріоритетів. На відміну від традиційного підходу до монолітної розробки, коли кожна команда спеціалізується на призначеному для користувача інтерфейсі, базах даних, технологічних рівнях або серверній логіці, в мікросервісній архітектурі використовуються крос-функціональні групи. Обов'язки кожної команди полягають у тому, щоб створювати конкретні продукти на основі одного або кількох окремих сервісів, що взаємодіють між собою. У мікросервісах команда володіє продуктом протягом всього терміну його служби.

3) Проста маршрутизація

Мікросервіси діють як класична система UNIX: вони отримують запити, обробляють їх і генерують відповідь. Мікросервіси оснащені інтелектуальними вузлами, які обробляють інформацію і застосовують логіку, і тривіальними каналами, якими передається інформація.

4) Децентралізованість.

Оскільки в мікросервісах використовуються різні технології і платформи, класичні методи централізованого управління не є оптимальними. Спільнота мікросервісів надає перевагу децентралізованому управлінню, оскільки його розробники прагнуть створити корисні інструменти, які потім можуть бути використані іншими для усунення тих же проблем. Як і модель децентралізованого управління, мікросервісна архітектура також сприяє децентралізованому управлінню даними. Монолітні системи використовують єдину логічну базу даних для різних додатків. У мікросервісному додатку кожен сервіс зазвичай керує своєю унікальною базою даних.

5) Відмовостійкість.

Мікросервіси створені для того, щоб виправляти помилки. Оскільки кілька унікальних і різноманітних сервісів взаємодіють один з одним, цілком можливо, що сервіс може вийти з ладу з певної причини (наприклад, коли

постачальник даних недоступний). У цих випадках пов’язані з цим сервісом сервіси мають функціонувати. Також моніторинг мікросервісів може допомогти запобігти ризику збою. Зі зрозумілих причин ця вимога збільшує складність мікросервісів в порівнянні з архітектурою монолітних систем.

6) Еволюційність

Архітектура мікросервісів є еволюційним проектом та ідеальна для еволюційних систем. Багато додатків розпочинають фунціонування на основі монолітної архітектури, але в міру виникнення деяких непередбачених вимог можуть бути поступово оновлені до мікросервісів.

Одна з поширених проблем полягає в спільному використанні схем даних або логіки валідації між сервісами. Те, що потрібно А для того, щоб визначити деякі дані як валідні, не завжди доцільно та можливо застосувати до В, якщо у В інші потреби. Одна з найбільш вдалих порад – застосовувати версіонування і поширювати схеми даних в поділюваних бібліотеках. Зміни в бібліотеках стають дискусіями між командами. Крім того, за строгого керування версіями з'являються залежності, які можуть викликати додаткові витрати. Найбільш ефективний спосіб для подолання цієї проблеми – запланувати забезпечення сумісності і прийняття регресійних тестів від зовнішніх служб або команд. Це породить обговорення проблем до того, як вони порушать певний бізнес-процес, а не після.

Як і в усьому іншому, відповідність мікросервісної архітектури розробці залежить від конкретних вимог, тому що всі архітектури мають свої переваги і недоліки.

Переваги мікросервісної архітектури:

- Проста інтеграція і автоматичне розгортання (з використанням інструментів безперервної інтеграції з відкритим вихідним кодом).
- Мікросервісна архітектура дає розробникам свободу самостійно розробляти і розгортати сервіси.
- Мікросервіс може бути розроблений досить невеликою командою.

- Код для різних сервісів може бути написаний різними мовами.
- Розробникам легко зрозуміти і модифікувати систему, що може допомогти новому члену команди швидко включитися в роботу.
- Легко масштабується та інтегрується зі сторонніми сервісами.
- Розробники можуть використовувати новітні технології.
- Краща ізоляція відмов: якщо один мікросервіс виходить з ладу, інший продовжить працювати (а одна проблемна область монолітного додатку може поставити під загрозу всю систему).
- Коли потрібна зміна в певній частині додатка, тільки відповідна служба може бути змінена і повторно розгорнута – немає необхідності змінювати і повторно розгортати весь додаток.
- Немає довгострокової прив’язки до технологічного стеку.

Недоліки мікросервісної архітектури:

- Архітектура вносить додаткову складність, оскільки розробникам доводиться знижувати відмовостійкість, затримку в мережі і працювати з різними форматами повідомлень, а також з балансуванням навантаження.
- Через розподілене розгортання тестування може стати складним і забрати багато ресурсів.
- Розробникам доводиться мати справу з додатковою складністю розподіленої системи.
- Розподілена система може спричинити дублювання зусиль.
- Коли кількість послуг збільшується, інтеграція і управління цілими продуктами можуть ускладнитися.
- Розробники повинні докласти додаткових зусиль для реалізації механізму зв'язку між сервісами.
- Збільшення кількості сервісів може призвести до інформаційних бар'єрів.

– Обробка сценаріїв використання, які охоплюють більше одного сервісу, без використання розподілених транзакцій, не тільки важка, але і вимагає спілкування і співпраці між різними командами.

Згідно з предметом дослідження, нескладний мікросервіс з обробки великих даних складається всього із трьох компонентів:

1) мова програмування, яка використовується для його розробки;
2) фреймворк кластерних обчислень для роботи з великими даними;
3) реалізоване API для отримання вхідних даних та видачі результату обробки даних.

Для розробки будь-яких програм для обробки великих даних найкраще підходять мови, що створені для підтримки функційної парадигми. Таких мов не так багато, найпоширенішими з них є Haskell, Erlang, F# та Scala.

Найпоширенішим фреймворком для кластерних обчислень є Apache Spark. Сам він написаний на мові Java, а отже, найкраще взаємодіє саме з мовами, які також запускаються у Java Virtual Machine. З наведених вище мов, що підтримують функційну парадигму, лише Scala є такою мовою.

Якщо у випадку з мовою, якою найдоцільніше обрати Scala для такої задачі, немає особливих складнощів (її синтаксис максимально простий та доступний, хоча і незвичний для розробників, що використовують інші парадигми та мови), то у випадку із фреймворком для кластерних обчислень, яким найдоцільніше обрати Apache Spark, все складніше, якщо говорити про такий мікросервіс, який було описано у предметі дослідження: нескладний, швидкий у розробці.

Безумовно, Apache Spark у загальному випадку дасть розробникам великий приріст швидкості розробки і, якщо дані, які необхідно оброблювати, дійсно великі, то його використання є абсолютно доцільним. Проте складність його налаштування у залежності від поставленої задачі часом робить недоцільним та нерентабельним його створення взагалі. Нескладним мікросервісом з обробки великих даних можна вважати саме такий, у якому

налаштування Apache Spark може зайняти більше, ніж вся інша частина його розробки.

Отже, постає необхідність створити часткову обгортку над Apache Spark, яка дозволить зосередитись безпосередньо на логіці обробки даних та послідовності їх руху в системі мікросервісів замість повторення одних і тих самих дій з самого початку для кожного мікросервісу.

Що стосується технології передачі повідомлень, то найчастіше використовуються такі рішення: REST API, меседж-брокерні системи (наприклад, Apache Kafka або RabbitMQ), трігери (застосовується у найбільш поширеній системі хмарних обчислень Amazon Web Services і передбачає запуск мікросервісу окремо для кожної події, що стається в його інформаційному середовищі та несе певну інформацію). Зважаючи на поставлену проблему, нескладний мікросервіс має бути схильним до нескладних, поширених та перевірених часом технологій та рішень. Отже, реалізації REST API для нескладного мікросервісу має бути більш ніж достатньо, оскільки він не буде потребувати обробки великих об’ємів повідомлень (для цього потрібні меседж-брокери) та не завжди буде окуповувати витрати на системи хмарних обчислень (такі, як Amazon Web Services), щоб додавати підтримку трігерів певних систем хмарних обчислень.

1.3. Постановка задачі

Підсумовуючи вищенаписане, можна стверджувати, що на даний момент розробка навіть найпростіших мікросервісів з обробки великих даних є дороговартісним та ресурсозатратним процесом, хоча для швидкого старту невеликі мікросервіси можуть бути побудовані швидко за таким шаблоном:

– REST API для отримання повідомлення про запуск виконання завдання, що містить всі його деталі;
– Черга завдань;

– Мінімально налаштоване середовище виконання на боці кластера Apache Spark;
– Налаштовувані callback-функцій для обробки результату виконання завдань (наприклад, надсилання його іншому мікросервісу).

Таким чином, виникає задача розроблення бібліотеки для шаблонного створення таких мікросервісів, що може застосовуватись у тих галузях підприємництва, в яких до моменту появи такої необхідності не було подібних систем та мікросервісів, і допоможе швидко ініціювати вплив Data Science на бізнес-процеси підприємства.

РОЗДІЛ 2 ТЕХНОЛОГІЇ ОБРОБКИ ТА АРХІТЕКТУРА ДЛЯ РОЗРОБКИ МІКРОСЕРВІСІВ З ОБРОБКИ ВЕЛИКИХ ДАНИХ

2.1. Технології обробки і методи аналізу великих даних

Основа технологій великих даних з погляду обробки базується на таких ключових принципах:

1. Принцип розподіленого зберігання. Технології великих даних дають змогу організовувати сховище з вільної кількості окремих простих носіїв, таким чином усуваючи проблему величезного об’єму даних. При цьому організація зберігання може здійснюватися з різним ступенем надмірності, що постачає стійкість до збоїв окремо узятих носіїв.

2. Принцип розподіленої обробки, зважаючи на локальність даних. Обчислювач, на який поставляється програма обробки, перебуває у максимальній близькості до оброблюваних даних. Класичний підхід передбачає схему, в якій обчислювальні потужності і підсистема зберігання є розділеними, а дані необхідно поставити на обчислювач. У цьому і полягає принципова відмінність технологій великих даних, які опираються на обчислювальні кластери з необмеженої кількості обчислювачів, які забезпечуються місцевою підсистемою зберігання.

Опрацювання та доступ до даних забезпечує окреме програмне забезпечення. У цьому аспекті варто назвати Apache Hadoop – найвідоміший проект, який активно розвивається і назва якого на сучасному ринку інформаційних систем стала синонімом великих даних [10,11]. Технологія цього проекту – це програмний фреймворк, завдяки якому збереження і оброблення даних відбувається із використанням комп'ютерних кластерів, залучаючи парадигму MapReduce.

Головні складники платформи Hadoop є:

– відмовостійка розподілена файлова система Hadoop Distributed File System (HDFS), яка забезпечує збереження даних;

– програмний інтерфейс Map Reduce, що виступає базою для вироблення такого програмного забезпечення, якому під силу паралельно оброблювати на кластері з тисяч машин значні об'єми як структурованих, так і неструктурованих даних;
– Apache Hadoop YARN, що реалізує завдання керування даними.

Підхід MapReduce (на рисунку 2.1) передбачає два етапи у обробці даних: Map і Reduce відповідно. Етап Map характеризується попереднім обробленням даних, яке реалізується одночасно на різних вузлах кластера.

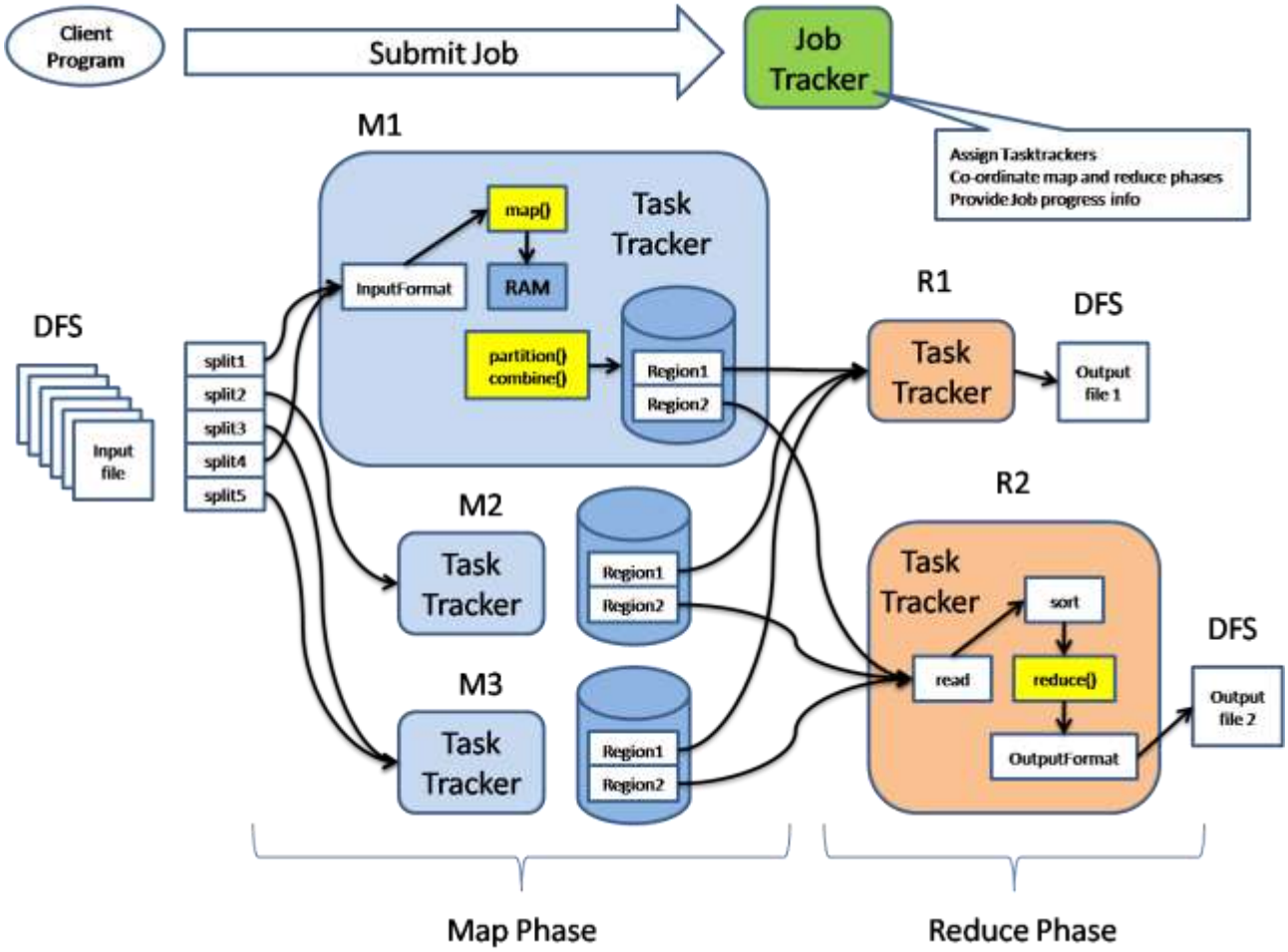


Рисунок 2.1 – Концептуальна модель MapReduce

Під час етапу Reduce зводяться завчасно оброблені дані в один загальний результат. Модель роботи Apache Hadoop базується на трьох ключових положеннях:

– дані рівномірно розподіляються на внутрішніх дисках необмеженої кількості серверів, пов’язаних HDFS;
– програма передається до даних, а не навпаки;
– дані опрацьовуються паралельно – така характеристика закладена архітектурно у програмному інтерфейсі Map Reduce.

Отже, замість традиційної моделі "база даних + сервер" технології великих даних передбачають кластер з необмеженої кількості недорогих вузлів. Кожен з цих вузлів водночас виступає і як сховище, і як оброблювач даних. Поняття "база даних" у моделі немає.

Основні переваги застосування Hadoop:

– скорочення часу на обробку і підготовку даних;
– розширення можливостей щодо аналізу;
– можливість оперування новою інформацією та неструктурованими даними.

Компанія Oracle розділяє життєвий цикл опрацювання інформації на три кроки. При цьому кожному крокові відповідає власне рішення:

1. Збір, обробка та структурування даних.

На цьому етапі залучають Oracle Big Data Appliance – встановлений Hadoop-кластер, Oracle NoSQL Database і засоби інтеграції з різноманітними сховищами даних. Функція Oracle Big Data Appliance – зберігати та попередньо опрацьовувати неструктуровану або частково структуровану інформацію.

2. Агрегація і аналіз даних.

Комплекс Oracle Exadata застосовується для процесів, пов’язаних зі структурованими даними. Модулі інтеграції Oracle Big Data Appliance дають можливість швидко завантажувати дані в Oracle Exadata, а також забезпечують миттєвий одночасний доступ до даних з Oracle Exadata.

3. Аналітика даних в режимі on-line.

Oracle Exalytics Database Machine забезпечує оперативний аналіз отриманих даних, а також дає можливість розв’язувати аналітичні завдання у режимі реального часу. Існує величезна кількість різних методів аналізу обсягів даних, які базуються майже на ідентичному наборі інструментів аналізу даних [11]: регресія, класифікація, кластеризація і пошук закономірностей. Деякі з цих методів не є винятковими лише для великих даних, а також успішно залучаються до значно менших обсягів інформації.

Основа методу багатовимірного аналізу – конструювання багатовимірного куба і отримання його різних зрізів. Зазвичай результати аналізу організовано в таблицю, де у осередках розташовані агреговані показники (кількість, середнє, мінімальне або максимальне значення тощо). За критерієм реалізації виділяють три категорії системи багатовимірного аналізу (OLAP): багатовимірна OLAP (Multidimensional OLAP – MOLAP); реляційна OLAP (Relational OLAP – ROLAP); гібридна OLAP (Hybrid OLAP – HOLAP). На сьогодні найбільш дослідженими вважаються ROLAP-системи, тому що вони базуються на поширених реляційних СУБД. Внутрішнє облаштування MOLAP і HOLAP є традиційно більш закритим і стосується галузі "ноу-хау" окремо узятих комерційних продуктів.

MOLAP поставляє інформацію у вигляді "чесної" багатовимірної моделі, але у внутрішньому устрої застосовуються спільні з ROLAP підходи: схеми "зірка" та "сніжинка". З аспекту систем управління базами даних база даних ROLAP – це класична реляційна база, яка потребує підтримки повного переліку операцій. Проте існує чотири стоп-фактори:

- неможливість жорстко контролювати етапи введення даних;
- неможливість збору статистики і підбору оптимальних структур для зберігання індексів;
- відсутність оптимізації розташування даних на диску для підвищеної швидкості введення / виводу;

– відсутність можливості провести глибокий статистичний аналіз і виробити оптимальний план виконання під час аналітичних запитів, що спричинено занадто високими вимогами до швидкодії.

ROLAP застосовують власні споріднені реляційні оптимізатори запиту, які жодним чином не зважають на "багатовимірність" бази даних. MOLAP вирізняються тим, що зазначені недоліки у них відсутні, що дає можливість максимально пришвидшити аналіз.

На доцільність залучення до аналізу великих даних MOLAP / ROLAP / HOLAP впливає фактор частоти оновлення бази даних. Якщо виступати з аспекту розпаралелювання обробки, то будь-який багатовимірний куб можна розділити за розподілами одного з вимірів та розподілити між деякою кількістю серверів за певною ознакою: часовою або регіональною тощо. Фактором для поділу куба є такий принцип: багатовимірний запит повинен буди здійснений не одним сервером, а декількома, а отримані результати у підсумку об’єднуються.

Однак у роботі існує дві перешкоди:

– складнощі при завчасному визначенні оптимального розподілу даних на певну кількість серверів;
– для певних аналітичних запитів може бути відсутньою інформація про те, які дані і з яких серверів будуть необхідні.

При роботі з великими даними наявні підходи для багатовимірного аналізу мають гарні можливості масштабування. Також ці підходи дають змогу кожному окремому серверу незалежно збирати та очищувати інформацію, а потім завантажувати в локальну базу.

Регресія – це побудова параметричної функції, яка описує зміну певної числової величини за певний відрізок часу. В основу функції лягають вже відомі дані. Саму функцію застосовують для прогнозування наступних можливих значень цієї ж величини. На вході поставляють послідовність пар на кшталт "час – значення", які характеризують зміни цієї величини у певних

умовах. На виході отримуємо параметри функції, яка характеризує поведінку величини, яку досліджують. Для підбору значень параметрів застосовують один і той самий спосіб, на вибір якого не впливає тип використовуваної параметричної функції. Розраховують загальну різницю спостережуваними значеннями та значеннями функції при поточних значеннях її параметрів. Далі з'ясовують, яким чином необхідно скоректувати значення параметрів, щоб поточна загальна різниця стала меншою, і процес починається спочатку. Кінцева мета – необхідний мінімум загальної різниці або неможливість додатково зменшити цю різницю. Головними операціями при проведенні регресійного аналізу з аспекту обробки даних виступають вирахування поточної загальної різниці і коректування значень параметрів. Поточну загальну різницю вираховують частинами на кожному сервері, а після цього підсумовують на центральному. Зазвичай під час процесу зміни та коректування значень параметрів застосовують загальновідомий математичний факт: функція певної кількості параметрів зростає в напрямку градієнта і спадає в протилежному. Таким чином щоб порахувати градієнт, необхідно порахувати приватні похідні функції за кожним з обраних параметрів, що зводиться до розділеного диференціювання, яке базується на обчисленні зважених сум. А отже, коректування значень параметрів також зводиться до підсумовування, яке можна розпаралелити.

Оскільки в основі суті регресійного аналізу лежить вирахування зважених сум, то регресійний аналіз застосовують при роботі з великими даними так само, як і багатовимірний. Це забезпечує системам регресійного аналізу хвилі можливість масштабування і роботи в умовах розподіленого збору інформації.

Завдання класифікації дещо збігаються із завданнями регресії: побудувати і використати залежність однієї змінної від певної кількості інших. Однак ці два методи мають принципову відмінну характеристику. Під час регресії здійснюють аналіз не тимчасового ряду, оскільки немає жодної

можливості чи способу впорядкувати значення на вході. Сьогодні серед науково-практичного доробку ми маємо величезну кількість методів класифікації (такі як функції Байеса, нейронні мережі), кожен з яких послуговується певною авторитетною та детальною науковою теорією. Однак незважаючи на таке різноманіття, всі методи класифікації мають у своїй основі однакову схему. На першому етапі на порівняно невеликій вибірці навчають алгоритм. Після цього отримані правила застосовують до іншої вибірки. Початковий етап передбачає можливість копіювати обсяг даних на один сервер для запуску традиційного алгоритму навчання без розпаралелювання роботи. Але на другому етапі існує можливість незалежного опрацювання даних. Це здійснюється таким чином: система правил, яку вивели за підсумками самонавчання, копіюють на кожний окремий сервер та пропускають через неї весь обсяг збережених на ньому даних. Отримані результати можуть або зберігатися на цьому сервері, або відправляються на інший для наступного опрацювання. Що маємо у підсумку: неможливо говорити про опрацювання великих даних на етапі навчання класифікаторів, оскільки не існує, по-перше, вибірок такого об’єму, по-друге, немає підготовлених для навчання систем, по-третє, на етапі класифікації деякі частини масивів даних опрацьовуються окремо та ізольовано один від одного.

Кластеризацію застосовують для того, щоб розподілити велику кількість інформаційних об'єктів на заздалегідь визначені групи. При цьому повинна зберігатися умова, що члени однієї групи схожі між собою більшою мірою, ніж члени з різних. Як фактор схожості застосовують функцію-відстань, на вхід якої надходять два об'єкта, а на вихід – міра їх схожості.

Застосування методу кластеризації у роботі з великими даними стикається з певною перешкодою: існуючі алгоритми припускають можливість безпосередньо звертатися до будь-якого інформаційного об'єкта у вихідних даних, а спрогнозувати завчасно, які саме об'єкти будуть залучені

до алгоритму, неможливо. Іншою перешкодою є можливість розподілу вихідних даних на різні сервери і відсутність гарантії, що кожен кластер зберігається тільки на одному сервері. Робити розподіл даних на окремі серверах прозорим абсолютно недоцільно, оскільки це спричинить таку ситуацію, за якої великі масиви даних скопіюються з одного сервера на інший.

Для того, щоб нівелювати вищеперераховані перешкоди, пропонується запуск на кожному із серверів власного алгоритму, який буде оперувати лише даними цього сервера, а на виході – давати параметри знайдених кластерів і їх ваги, які оцінюють, опираючись на число внутрішніх елементів кластера. Отриману у підсумку інформацію об’єднують на центральному сервері, після чого здійснюють метакластеризацію, тобто виділяють групи близько розміщених кластерів, зважаючи на їх ваги. Серед переваг такого методу можемо виділити універсальність застосування, задовільний рівень розпаралелювальності і можливість використання будь-яких інших алгоритмів кластеризації. Проте для цього методу необхідно здійснити ґрунтовні наукові дослідження, тестувати із залученням реальних даних і порівнювати результати з іншими місцевими методами. Отже, можемо зробити висновок, що застосування в галузі великих даних більшої частини методів кластеризації є недоцільним, оскільки це потребує додаткових досліджень і наукової підготовки, а це відповідно витрата різних видів ресурсів.

Під час застосування методу пошуку закономірностей визначають та окреслюють правила, які характеризують залежності між внутрішніми елементами даних. Одним із традиційних прикладів є аналіз покупок в супермаркеті і виділенні правил на кшталт "якщо покупець придбав відеокамеру, то разом із нею в чеку буде акумулятор і карта пам'яті". Як відбувається процес застосування методу: на вхід постачається неврегульована безліч об’єктів, кожному з яких притаманна певна кількість

присутніх інформаційних ознак. Таким чином, головною задачею для дослідника є виокремити та визначити правила на кшталт "якщо присутні ознаки А1, А2, А3..., Аn, то присутні і ознаки Б1; Б2, Б3..., Бn". Слід зважати, що для кожного правила існує ймовірність спрацьовування та підтримки. За ймовірністю спрацьовування визначають власне частоту виконання цього правила. За іншим параметром визначають частоту можливого застосування цього правила, тобто як часто трапляється поєднання ознак А1, А2, А3..., Аn.

Застосування технік змішування і інтеграції даних дає можливість інтеграції різнорідних дані з різних джерел для глибокого аналізу.

Краудсорсінг передбачає наявність у зборі даних величезного числа джерел. При цьому до процесів категоризації і поповнення даних залучають третіх осіб, кількість яких є невизначеною/необмеженою.

Машинне навчання із залученням штучного інтелекту Аі передбачає побудову алгоритмів самонавчання на базі статистичного аналізу даних або машинного навчання для отримання комплексних прогнозів.

Метод А / В тестування передбачає почергове порівняння контрольної вибірки з усіма іншими. У результаті виокремлюють оптимальну комбінацію показників для того, щоб досягти поставлених цілей, наприклад, найкращої відповіді споживачів на маркетингову пропозицію. Великі дані дають можливість багаторазово повторювати цю операцію, змінюючи показники, і отримати статистично вірогідний результат.

Методика генетичних алгоритмів передбачає представлення ймовірних рішень у вигляді "хромосом", які мають здатність до комбінації і мутації. При цьому здійснюється аналогія до природнього відбору: у результаті залишається одна, найбільш пристосована та сильна особина. Метод оптимізації охоплює набір чисельних методів для зміни та оновлення дизайну складних систем і процесів з метою поліпшити певний показник/показники. Цей метод доцільно застосовувати при ухваленні стратегічних рішень. Метод візуалізації аналітичних даних полягає в

унаочненні інформації і подачі її у вигляді малюнків, графіків, схем і діаграм з використанням інтерактивних можливостей та анімації як для результатів, так і для використання в якості вихідних даних.

2.2. Алгоритми кластеризації великих даних

Одна з головних проблем під час здійснення кластеризації – визначення кількості кластерів. Існує багато методів розбиття множини на кластери, в масиві яких виділяють неієрархічні та ієрархічні.

Характер роботи та фактори зупинки для неієрархічних алгоритмів потрібно завчасно визначати за допомогою значної кількості параметрів. Ця умова ускладнює процес, зокрема на перших ступенях вивчення матеріалу. Однак серед переваг цього алгоритму є можливість досягти гнучкої варіації кластеризації, а також визначити кількість кластерів. Однак значна кількість параметрів об'єктів вимагає якісно закрити завдання групування ознак. Тому при роботі з ієрархічними алгоритмами кількість кластерів не визначають, конструюючи натомість повне дерево вкладених кластерів (дендрограму). Кількість кластерів вираховують на базі незалежних від роботи алгоритмів факторів, наприклад, за динамікою варіації порога поділу/злиття кластерів. Перешкоди для цих алгоритмів досліджені належним чином: вибір заходів близькості кластерів, проблема інверсій індексації в дендрограмах, негнучкість ієрархічних класифікацій, яка може зашкодити та уповільнити процес. Проте представлення кластеризації у виді дендрограми забезпечує найповніше уявлення про структуру кластерів.

Алгоритм k-середніх

Алгоритм k-середніх передбачає максимально великі відстані між кластерами. Алгоритм k-середніх дає змогу уточнити кількість або зробити прогноз щодо кількості кластерів зі збереженням умови їх максимальної різноманітності. Принцип роботи алгоритму k-середніх: визначена фіксована

кількість до кластерів спостереження зіставляється кластерам таким чином, що середні показники в кластері мають максимально можливі розбіжності.

Алгоритм працює за такими стадіями [12]:

На першій стадії здійснюється первинний розподіл об'єктів за кластерами. Визначається кількість до, і на першому етапі ці точки розуміють як "центри" кластерів, тобто кожний кластер має один власний центр. Існує три варіанти для відбору первинних центроїдів:

– вибір k-спостережень для максимально можливої первинної відстані;
– випадковий вибір k-спостережень;
– вибір перших k-спостережень.

У кінці цієї стадії кожен об'єкт пов'язаний із конкретним кластером.

На другій стадії визначають центри кластерів, під якими надалі розуміють покоординатно середні кластерів. Відбувається черговий перерозподіл об'єктів, який спільно із процесом визначення центрів триває, допоки не буде реалізована одна з умов: стабілізація кластерних центрів (всі спостереження стосуються того ж кластеру, якого стосувалися до нинішньої ітерації); кількість ітерацій досягла максимальної верхньої межі.

Існують складнощі щодо початкового вибору кількості кластерів. Якщо гіпотези щодо цієї кількості відсутні, то доцільно створити 2 кластера, а потім поступово збільшувати їх кількість, порівнюючи отримані у процесі результати.

Серед переваг використання методу k-спостережень називають простоту та швидкість використання, а також зрозумілий і прозорий алгоритм. Однак алгоритм k-спостережень вирізняється і певними слабкими сторонами, а саме: він дуже чутливий до викидів, які можуть викривлювати середнє значення; довгий час роботи на великих базах даних; необхідність задавати кількість кластерів.

Алгоритм HCM (Hard C-Means)

Метод Hard C-Means використовують переважно під час кластеризації значних масивів числових даних. Алгоритм працює за такими стадіями [13]:

Перша стадія. Здійснюється ініціалізація кластерних центрів c_i (i = 1,2, ..., c). Один із варіантів – обрати випадковим чином c-вектори з вхідного набору.

Друга стадія. Обчислення матриці M, яка утворена з елементів m_{ik}:

$$m_{ik} = \begin{cases} 1, \|u_k - c_i\|^2 \leq \left\|u_k - c_j\right\|^2, \text{для віх } i \neq j, \\ i = 1,2, \dots C, k = 1,2, .., K \\ 0, \text{інше} \end{cases}, \quad (2.1)$$

де К – кількість елементів у вхідному наборі даних.

Матриця M вирізняється характеристикою:

$$\sum_{i=1}^{c} m_{ij} = 1, \sum_{j=1}^{K} m_{ij}. \quad (2.2)$$

Третя стадія. Визначення об'єктної функції:

$$J = \sum_{i=1}^{c} J_i = \sum_{i=1}^{c} \left(\sum_{k, u_k \in C_i} \|u_k - c_i\|^2 \right). \quad (2.3)$$

Четверта стадія. Перерахунок кластерних центрів рівнянням:

$$C_i = \frac{1}{|C_i|} \sum\nolimits_{k, u_k} \in C_i u_k, \quad (2.4)$$

де $|C_i|$ – кількість елементів в і-м кластері.

П’ята стадія. Повертаємося на другу стадію.

Цей метод легко здійснити та просто обчислити, проте необхідно задавати кількість кластерів. Також метод не гарантує отримання оптимального рішення.

Алгоритм нечіткої кластеризації Fuzzy C-means

Цей метод, як і попередній, переважно використовують для кластеризації великих масивів першою чергою числових даних.

Алгоритм працює за таким принципом:

Нехай нечіткі кластери задаються матрицею розбиття:

$$F = [\mu_{ki}], \quad \mu_{ki} \in [0,1], \quad k = \overline{1,M}; i = \overline{1,c},$$

де μ_{ki} – ступінь належності об'єкта до до кластеру k;

c – кількість кластерів;

M – кількість елементів.

Водночас:

$$\begin{cases} \sum_{i=1}^{c} \mu_{ki} = 1, k = \overline{1,M}; \\ 0 < \sum_{k=1}^{M} \mu_{ki} < M, i = \overline{1,c} \end{cases}. \tag{2.5}$$

Перша стадія. Означити параметри алгоритму: c – кількість кластерів; m стадія експонентна вага, яка безпосередньо впливає на нечіткість кластерів ($m \in [1,\infty)$); ε – параметр зупинки алгоритму.

Друга стадія. Випадкова генерація матриці нечіткого розбиття з виконанням умов пункту (2.5).

Третя стадія. Визначення центрів кластерів

$$V_i = \frac{\sum_{k=1}^{M} \mu_{ki}^m * |X|}{\sum_{k=1}^{M} \mu_{ki}^m}, i = \overline{1,c}. \tag{2.6}$$

Четверта стадія. Визначення відстані між об'єктами X і центрами кластерів:

$$D_{ki} = \sqrt{\|X_k - V_i\|^2}, k = \overline{1,M}; \ i = \overline{1,c}. \tag{2.7}$$

П’ята стадія. Перерахунок елементів матриці розбиття з виконанням наступних умов:

$$\text{якщо } D_k > 0: \mu_k = \frac{1}{\left(D_{jk}^2 * \sum_{j=1}^{c} \frac{1}{D_{jk}^2}\right)^{\frac{1}{(m-1)}}}, j = \overline{1,c}, \qquad (2.8)$$

$$\text{якщо } D_k = 0: \mu_k = \begin{cases} 1, j = i \\ 0, j \neq i \end{cases}, j = \overline{1,c} \ .$$

Шоста стадія. Перевірити виконання умови || F – F* || < ε, де F* – матриця нечіткого розбиття на попередній ітерації алгоритму. Якщо умова виконується, то перехід до сьомої стадії, інакше – третьої.

Сьома стадія. Завершення.

При використанні цього методу завдяки нечіткості при визначенні об'єкта в кластер є можливість визначати об'єкти, які розташовані на межі, в кластери. Однак цей метод вирізняється складністю обчислень, необхідністю задавати число кластерів, невизначеністю з об'єктами, які віддалені від центрів всіх кластерів.

Алгоритм FRiS-Cluster

Проблема визначення ступеня схожості є однією з ключових під час класифікації будь-яких об'єктів. Алгоритм FRiS-Cluster базується на використанні FRiS-функції (Function of Rival Similarity – функція конкурентного подібності). FRiS – ступінь схожості двох об'єктів відносно певного третього об'єкта. На відміну від інших критерії подібності, FRiS дає змогу оцінювати поняття "далеко" або "близько", "схожий" або "не схожий" не лише у якісному, а і кількісному вимірі. Цей підхід дає можливість спиратися на більшу кількість факторів під час сортування об’єктів по класам. FRiS-функція має здатність імітувати людський механізм сприйняття схожості та відмінності з високою точністю. Тому FRiS-функцію можна застосовувати як

базовий елемент для різноманітних задач, зокрема завдання кластеризації документів.

Принцип роботи алгоритму кластеризації полягає у послідовному пошуку розв'язку задачі кластеризації для значень k = 1, ..., K, де K – визначена користувачем максимальна кількість кластерів. Потім з масиву отриманих рішень обирають найбільш доцільне. В описі алгоритму застосовується FRiS-функція:

$$F(a,S) = \frac{r_2(a,S) - r_1(a,S)}{r_2(a,S) + r_1(a,S)}, \tag{2.9}$$

і скорочена FRiS-функція:

$$F^*(a,S) = \frac{r_2^* - r_1(a,S)}{r_2^* + r_1(a,S)}, \tag{2.10}$$

де r_1 – мінімальна відстань до найближчого стовпа спорідненого кластера;

r_2 – мінімальна відстань до найближчого стовпа кластера-конкурента;

r_2^* – відстань до віртуального стовпи (застосовується при k = 1, коли дійсні конкуренти ще не були виокремлені).

Перша стадія. При k = 1 для кожного об'єкта a навчальної вибірки окреслюється значення середньої скороченої FRiS-функції

$$\overline{F^*}(S) = \frac{1}{M}\sum_{a\in A} F^*(a,S). \tag{2.11}$$

Друга стадія. Об'єкт з максимальним значенням середньої скороченої FRiS-функція приймається за перший стовп - s_1.

Третя стадія. Кожен об'єкт a, за винятком s_1, приміряється на роль другого стовпа, стовпом s_1 окреслюється об'єкт, для якого значення функції $\overline{F^*}\{(a,s_1)\}$ перебуває на відмітці максимуму.

Четверта стадія. Всі об'єкти вихідної вибірки сортуються між стовпами s_1, s_2, організовуючи кластери A_1, A_2 відповідно. Об'єкт a належить до кластеру A_1, якщо відстань від a до s_1 менша, ніж для будь-якого іншого s_i.

П’ята стадія. Після додавання кластера A_2 еталонний об'єкт для кластера A_1 визначається вдруге, проте цього разу серед множини об'єктів кластера A_1Столпом s_{12} визначається об'єкт, на якому функція $\overline{F^*}\{(a, s_2)\}$ сягає максимальної відмітки. Аналогічно визначають стовп s_{22} для другого кластера. Отримана кластеризація є розв’язанням задачі для k = 2:

$$\bar{F}(S) = \frac{1}{M}\sum_{a\in A} F(a, S). \qquad (2.12)$$

Шоста стадія. Далі для кожного нового стовпа повторюються ті ж операції, що і під час додаванні об'єкта два. П’ята стадія виконується для всіх кластерів цієї кластеризації.

Ускладнює застосування цього алгоритму висока трудомісткість $O(n^3)$ і необхідність більшої кількості осередків n^2 пам'яті, що спричинено необхідністю зберігати матрицю парних відстаней.

Алгоритм FRiS-Stolp

FRiS-функцію також використовують у алгоритмі відбору еталонних прикладів для метричного класифікатора, який має назву FRiS-Stolp. Вибір еталонів реалізується із залученням алгоритму FRiS-Stolp, який базується на тому, що всі об'єкти першого способу почергово стають еталонами. Алгоритм орієнтований на вибір мінімальної кількості стовпів, які захищають не лише самих себе, а усі об'єкти, що входять до навчальної вибірки. Першими обираються стовпи, що захищають максимально можливу кількість об'єктів із визначеним рівнем надійністю. Тому за нормального розподілу насамперед будуть обрані стовпці, що перебувають в точках математичного очікування[14].

Стадії алгоритму FRiS-Stolp:

Перша стадія. Перевіряємо варіант, за якого перший обраний випадковим чином об'єкт a_i є єдиним стовпом образу S; а всі інші мають власні об'єкти. Для всіх об'єктів $a_i \neq a_j$ першого образу вираховуємо відстань r_{1j} до власного стовпа a_i і відстань r_{2j} до найближчого об'єкта іншого образу. За отриманими величинами вираховуємо значення FRiS-функції для кожного об'єкта a_j першого образу. Визначаємо ті m_i об'єктів першого образу, значення функцій приналежності F яких перевищує визначений поріг F^*. За цим m_i об'єктам вираховуємо суму значень FRiS-функції F_j, яка описує придатність об'єкта a_j на роль стовпа.

Друга стадія. Повторюємо ті ж самі дії, що і на першій стадії, призначаючи як стовпи все M об'єктів першого образу по черзі.

Третя стадія. Знаходимо об'єкт a_i з максимальним значенням F_i і визначаємо його як перший стовп A_{11} першого кластера C_{11} першого образу S_1.

Четверта стадія. Виключаємо з першого образу m_i об'єктів, що належать до першого кластеру.

П’ята стадія. Для інших об'єктів першого образу визначаємо наступний стовп повторенням стадій з першої по четверту.

Шоста стадія. Процес зупиняється за умови, що всі об'єкти першого образу належать до своїх кластерів.

Сьома стадія. Відновлюємо всі об'єкти образу S_1 і для образу S_2 виконуємо стадії з першої по шосту.

Восьма стадія. Повторюємо стадії з першої по сьому для всіх інших образів. Ця стадія – завершення першого етапу пошуку стовпів. Кожен стовп A_j захищає підмножина об'єктів m_j власного кластера C_j.

Алгоритм LSH (Locality-sensitive hashing)

Locality-sensitive hashing (LSH) – це імовірнісний метод знизити розмірності багатовимірних даних. його основу складає підбір хеш-функцій для

деяких вимірювань, щоб подібні об'єкти з високим ступенем ймовірності потрапляли до того самого кошика. За збільшення розмірності вихідних даних пошук за індексом реалізується з більшими похибками, аніж послідовний перегляд. Метод дає змогу створити структуру для швидкого ймовірнісного пошуку n-мірних векторів, подібних до шуканого шаблону.

Сьогодні LSH здобув найбільшу популярність серед наявних наближених алгоритмів пошуку найближчих сусідів (Approximate Nearest Neighbor, ANN). LSH у такому аспекті перетворює величезну кількість точок у високорозмірному просторі на безліч бінів, тобто на хеш-таблицю. Відмінність LSH від традиційних хеш полягає у чутливості до місця розташування (locality-sensitive hash), завдяки чому здатний розташовувати сусідні точки в один і той самий бін.

LSH простий у застосуванні, підкріплюється строгою теорією, яка означає високу продуктивність алгоритму. Також LSH сумісний з будь-нормою L_p при $0 < p \leq 2$. Є можливість застосовувати LSH з евклідовою метрикою, з манхеттенською відстанню. Існують також ситуації для відстані Хеммінга і косинусного коефіцієнта.

В умовах тотальної глобалізації та цифровізації суспільства можна стверджувати говорити про настання епохи великих даних оскільки наука, бізнес, виробництво та управлінні ІТ оперують величезними масивами даних, які збільшуються у експоненціальній залежності. Однак існують проблеми щодо наявних програм обробки великих даних: користувачі не мають можливостей керувати та управляти етапами введення даних, збирати статистику і підбирати оптимальні структури для зберігання індексів. Наразі також неможливо оптимізувати розташування даних на диску, щоб пришвидшити введення/виведення. Також неможливо здійснити глибинний статистичний аналіз і розробити оптимальний план для здійснення аналітичних запитів.

Найбільш важливу роль у процесах масштабування і швидкодії додатків відіграють характеристики мереж ЦОД – оптимізація якості обслуговування (QoS) потребує інтенсивного обміну інформацією між обчислювальними вузлами. Однак значна частина наявних ЦОД не може достатньо швидко переносити дані у порівнянні з показниками комунікаційних мереж високопродуктивних комп'ютерів.

Серед сучасних методів аналізу для опрацювання великих даних не існує інструментів, які могли б стовідсотково задовольнити вимоги додатків опрацювання великих даних. Деякі з традиційних методів не підходять для специфіки роботи з опрацювання великих даних. Для інших методів немає можливості застосовувати їх у "чистому" вигляді, тому алгоритм вирізняється підвищеною трудомісткістю. Також певні класичні методи не можна залучати при побудові автоматичної класифікації великої кількості об'єктів в умовах відсутності апріорної інформації про число класів.

2.3. Архітектура мікросервісу з обробки великих даних

У бібліотеці передбачено створення синхронних та асинхронних мікросервісів. Синхронним у данному випадку є такий мікросервіс, який у процесі виконання обробки даних не створює додаткових потоків у драйвері (самій програмі), а лише використовує кластерні обчислення. Асинхронний – це такий, який під час обробки даних використовує дані, що були отримані асинхронно (наприклад, зчитаний файл з іншого мікросервісу за допомогою HTTP-протоколу з використанням неблокуючого завантаження), а типом результату обробки даних є тип Future, що буде містити або результат обробки даних, або інформацію про помилку. Загалом структура цих сервісів однакова, а розділення було здійснено виключно для зручності розробника, який буде використовувати цю бібліотеку.

Функціональну схему наведено на рисунку 2.2.

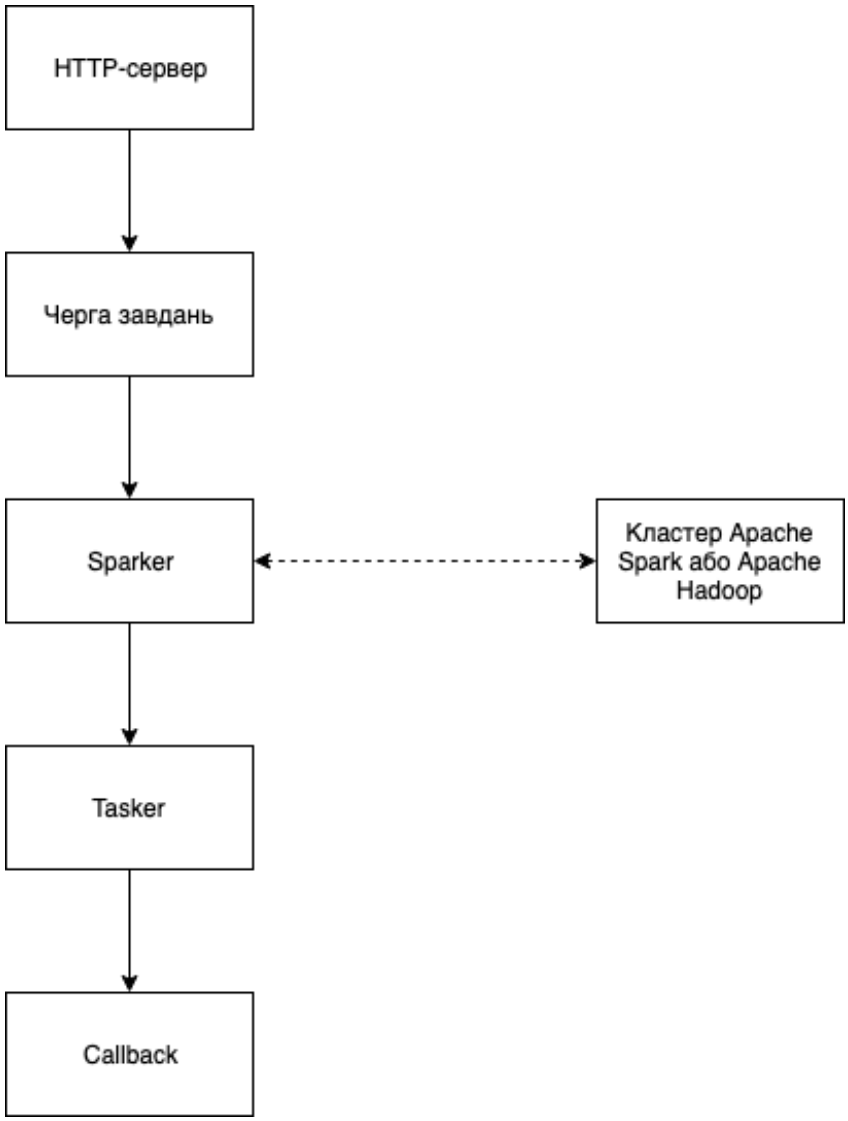


Рисунок 2.2 – Функціональна схема бібліотеки для розробки мікросервісів з обробки великих даних

Пояснення до функціональної схеми:

1) HTTP-сервер – це API, що реалізоване розробником та вимагає визначення значення типу Route з бібліотеки Akka HTTP.
2) Черга завдань – це синхронізатор багатопотокового доступу за допомогою API, що по черзі запускає усі поставлені завдання з обробки даних на виконання.
3) Sparker – це модуль, що відповідає за роботу з Apache Spark або Apache Hadoop кластером шляхом управління SparkSession. Він реєструє завдання у кластері перед початком його виконання (або ж ставить його у

чергу на самому кластері, що регулює послідовність виконання завдань від різних мікросервісів), а після закінчення виконання завдання закриває сесію для звільнення резервування ресурсів кластера для мікросервісу.

4) Tasker – це модуль, що відповідає за відкладений запуск завдання (коли настала його черга), що було сконструйоване ще перед потраплянням до черги.

5) Callback – це функція, яку буде автоматично викликано для обробки результатів виконання завдання після його завершення. Наприклад, можна відправити повідомлення іншому мікросервісу про те, що він може починати оброблювати результат, або надіслати електронного листа користувачу, який надсилав це завдання, про успішне його завершення.

РОЗДІЛ 3 РОЗРОБКА МІКРОСЕРВІСІВ З ОБРОБКИ ВЕЛИКИХ ДАНИХ

3.1. Обґрунтування вибору мови реалізації та стеку технологій

У якості мови для розробки будь-яких програм для обробки великих даних найкраще підходять мови, що створені для підтримки функційної парадигми, оскільки функційна парадигма наголошує на використанні "чистих" функцій, відсутності перезаписуваних змінних, рекурсійних обчисленнях та в цілому призначена для розподілених обчислень. Таких мов не так багато, найпоширенішими з них є Haskell, Erlang, F# та Scala.

Найпоширенішим фреймворком для кластерних обчислень є Apache Spark. Сам він написаний на мові Java. Отже найкраще взаємодіє саме з мовами, які також запускаються у Java Virtual Machine. З наведених вище мов, що підтримують функційну парадигму, лише Scala є такою мовою.

Для роботи зі змінними конфігураціями було обрано бібліотеку Lightbend Config, що була розроблена компанією, заснованою творцем мови Scala, а отже гарантує сумісність зі Scala різних версій.

Для логування подій програми було обрано бібліотеку Lightbend scala-logging, тому що вона дозволяє її використовувати, використовуючи множинне наслідування trait у Scala (зокрема містить передвизначений trait LazyLogging) замість конфігурування логера у коді.

Для створення REST API використовується бібліотека Akka HTTP, оскільки вона дозволяє створювати повністю неблокуючі веб-сервери з повною асинхронністю обчислень.

Для синхронізації паралельних потоків (а конкретно створення черги для завдань з обробки великих даних) використано абстракцію акторів, а зокрема її імплементацію для мови Scala у бібліотеці Akka Actors.

Внаслідок такого підбору мови та стеку технологій стало можливим реалізувати бібліотеку у функційній парадигмі програмування з використанням об’єктно-орієнтованих модифікацій, які дозволяє зробити Scala. Це забезпечує

суттєве пришвидшення розробки таких мікросервісів, а також забезпечило пришвидшення розробки самої бібліотеки.

Мікросервіс з обробки великих даних – це серверний додаток. Для його запуску мікросервіс має бути сконфігурований для доступу до кластеру Apache Hadoop або Apache Spark, що складається принаймні з однієї машини. Кожна з машин має мати доступ до всіх машин кластера та до машини з драйвером (самим мікросервісом, якщо його встановлено, згідно рекомендацій, не на сервері з кластеру) за протоколом SSH без паролю (за файлом з ключем).

Рекомендовані системні вимоги до усіх машин, включно з машиною з драйвером:

- Пропускна здатність мережі не менше 1 Гбіт/с
- Багатоядерний процесор
- SSD-диск для файлів кешу Apache Spark

Вимоги до об’єму оперативної пам’яті, об’єму SSD-диску для файлів кешу Apache Spark та потужності центрального процесора встановлюються виходячи з особливостей кожних окремих даних, для обробки яких створюється мікросервіс.

3.2. Реалізація мікросервісів з обробки великих даних

У функції main необхідно створити один екземпляр класу, що є нащадком trait BigDataMicroserviceSync (для синхронного мікросервісу) або BigDataMicroserviceAsync (для асинхронного мікросервісу). Пропонується у головному конструкторі класу перевизначити поля host та port.

Обидва (асинхронний та синхронний) trait для створення мікросервісу вимагають визначення двох типів: Input та T. Input – це тип тих даних, які передаються із веб-сервера до таскера, щоб із них сконструювати завдання для подальшого відкладеного запуску. T – це тип результату. У випадку асинхронного мікросервісу типом результату буде Future[T].

Через порядок ініціалізації значень при наслідуванні та використанні не визначених значень у самій бібліотеці перевизначення наступних полів мають бути обов'язково позначені як lazy: actorSystem, mat, ec, partitioner, route, config.

При визначенні поля route необхідно використати передвизначену в бібліотеці функцію queueSender, щоб надіслати вхідні дані у чергу.

Наприклад:

```
override lazy val route: Route = path("api") {
  parameters('path) { path =>
    queueSender(path)
    complete(HttpResponse(status = StatusCodes.OK))
  }
}
```

Partitioner – це такий trait, у якому визначена абстрактна функція parts, що приймає аргументом той самий тип Input, що формується веб-сервером та надсилається у чергу. Результатом виконання цієї функції є натуральне число, що вказує на кількість партицій, на які необхідно розбити ці дані на кластері для розподіленої обробки. Мінімальна рекомендована кількість – це сумарна кількість логічних ядер процесора на всіх машинах кластера. У світовій практиці це число визначається експериментально.[15]

Tasker – це така функція, що приймає аргументом значення типу Input, а повертає функцію, що приймає аргументом SparkSession, а повертає завдання (у випадку із синхронним мікросервісом це екземпляр класу TaskSync, у випадку із асинхронним – екземпляр класу TaskAsync).

Функція sparkConf приймає два аргументи: конфігурацію типу Config бібліотеки Lightbend Config і вхідні дані типу Input (для передачі у функцію parts визначеного екзмепляру класу Partitioner). Ця функція повертає екзмепляр класу SparkConf (пакет org.apache.spark бібліотеки spark-core).

Рекомендована імплементація функції sparkConf:

```
override def sparkConf(config: Config, input: String): SparkConf = {
  new SparkConf()
    .setMaster({
      config.getString("spark.master")
    })
    .setAppName(s"bigdata_${config.getString("branchTag")}" +

s"_${input.takeRight(10)}${scala.util.Random.alphanumeric.take(3).mkString
("")}")
    .setAll(
      {
        val parts = partitioner.parts(input)

        val l = List(
          "spark.sql.shuffle.partitions" -> s"$parts",
          "spark.default.parallelism" -> s"$parts",
          "spark.executor.memory" ->
config.getString("spark.executor.memory"),
          "spark.files.maxPartitionBytes" ->
config.getString("spark.files.maxPartitionBytes"),
          "spark.rpc.message.maxSize" ->
config.getString("spark.rpc.message.maxSize"),
          "spark.driver.port" -> config.getString("spark.driver.port"),
          "spark.driver.host" -> config.getString("spark.driver.host"),
          "spark.driver.bindAddress" ->
config.getString("spark.driver.bindAddress"),
          "spark.blockManager.port" ->
config.getString("spark.blockManager.port"),
          "spark.ui.port" -> config.getString("spark.ui.port"),
          "spark.submit.deployMode" -> "client",
```

```
        "spark.executor.extraClassPath" -> s"-
Dconfig.resource=${config.getString("configFile")}",
        "spark.driver.extraJavaOptions" -> s"-
Dconfig.resource=${config.getString("configFile")}"
      )
      if (config.getBoolean("spark.uploadJars")) {
        val additionalConfig = List(
          "spark.yarn.jars" ->
            s"target/scala-
${bigdata_info_build.BuildInfo.scalaVersion.split('.').take(2).mkString(".")}/
bigdata-example-assembly.jar",
          ("spark.hadoop.yarn.resourcemanager.hostname",
config.getString("spark.hadoop.yarn.resourcemanager.hostname")),
          ("spark.hadoop.yarn.resourcemanager.address",
config.getString("spark.hadoop.yarn.resourcemanager.address")),
          ("yarn.app.container.log.dir", "/var/log/hadoop/userLogs")
        )
        additionalConfig ::: l
      }
      else l
    }
  )
}
```

Рекомендований файл конфігурації для 8-поточного локального запуску:

```
        spark {
          executor {
            memory = "4G"
          }
          driver {
            port = "20002"
            host = "127.0.0.1"
            bindAddress = "127.0.0.1"
```

```
  }
  rpc {
    message {
      maxSize = "2047"
    }
  }
  files {
    maxPartitionBytes = "1342177280"
  }
  uploadJars = false
  master = "local[8]"
  blockManager {
    port = "20003"
  }
  ui {
    port = "4040"
  }
}

configFile = "application.conf"

branchTag = "LOCAL"
```

У даному розділі наведено функціональну схему роботи програми, розробленої за допомогою розробленої в рамках даної роботи бібліотеки для розробки мікросервісів з обробки великих даних.

Приведений приклад може бути застосовано як повноцінний мікросервіс, якщо реалізувати його DevOps частину і розгорнути у місці його подальшого прикладного використання, а також замінити орієнтованість на тестові дані обробкою реальних даних.

ВИСНОВКИ

Кваліфікаційна робота присвячена задачі розробки бібліотеки для розробки мікросервісів з обробки великих даних та демонстраційного мікросервісу з обробки великих даних з використанням цієї бібліотеки.

За допомогою мови програмування Scala та фреймворку для кластерних обчислень Apache Spark було розроблено бібліотеку для розробки мікросервісів з обробки великих даних та демонстраційний мікросервіс з кластеризації великих даних методом LSH (Locality-sensitive hashing).

У першому розділі було надано визначення великих даних, оглянуто методи та техніки аналізу великих даних, описано концепції роботи з великими даними, надано поняття мікросервісної архітектури, оглянуто мікросервісну архітектуру та надато опис її недоліків та переваг, надано огляд існуючих рішень для створення мікросервісів з обробки великих даних та поставлено задачу дослідження.

У другому розділі описано методи аналізу великих даних, зокрема наведено приклади алгоритмів кластеризації, один з яких було використано для розробки демонстраційного мікросервісу з обробки великих даних з використанням розробленої бібліотеки.

У третьому розділі наведено функціональну схему роботи програми, розробленої за допомогою розробленої в рамках даної роботи бібліотеки для розробки мікросервісів з обробки великих даних, було надано інструкції з використання цієї бібліотеки.

Четвертий розділ являє собою економічну частину.

СПИСОК ЛІТЕРАТУРИ

1. Головна сторінка репозиторію spark-hash. URL: https://github.com/mrsqueeze/spark-hash.
2. Стаття, присвячена визначенню мікросервісної. URL: https://smartbear.com/solutions/microservices/
3. Стаття, присвячена визначенню великих даних. URL: https://uk.wikipedia.org/wiki/Великі_дані
4. Стаття, присвячена великим даним. URL: http://thefuture.news/bigdata
5. Стаття, присвячена великим даним. URL: https://www.it.ua/knowledge-base/technology-innovation/big-data-bolshie-dannye
6. Болюбаш Ю. Я. Методи та засоби опрацювання інформаційних ресурсів Великих даних в системах територіального управління : дис. канд. техн. наук : 01.05.03. Львів, 2017. 165 с.
7. Алєксєєв Н. О. Система аналізу телекомунікаційних данних в Big Data середовищі : дис. магістр. Комп'ютерні науки : 122. Київ, 2018. 97 с.
8. Стаття, присвячена Locality-sensitive hashing. URL: https://ru.wikipedia.org/wiki/Locality-sensitive_hashing
9. Бутакова М.А., Климанская Е.В., Янц В.И. Мера информационного подобия для анализа слабоструктурированной информации. *Современные проблемы науки и образования.* 2013. № 6. URL: http://www.science-education.ru/113-11307
10. Is Big Data a Bubble Set to Burst? URL: https://www.datacenterknowledge.com/archives/2015/03/30/big-data-bubble-set-burst
11. Большие данные (Big Data). URL: http://www.tadviser.ru/index.php/Статья:Большие_данные (Big_Data)
12. k-means clustering. URL: https://en.wikipedia.org/wiki/K-means_clustering

13. Участник:Urandon/Строгий алгоритм С средних (Hard C-Means, HCM): http://algowiki-project.org/ru/Участник:Urandon/Строгий_алгоритм_С_средних_(Hard_C-Means,_HCM)

14. Алгоритм FRiS-СТОЛП. URL: http://www.machinelearning.ru/wiki/index.php?title=Алгоритм_FRiS-СТОЛП

15. Стаття, присвячена проблемі memoryOverhead в Apache Spark. URL: https://gsamaras.wordpress.com/code/memoryoverhead-issue-in-spark/

16. H.Karau, A. Konwinski, P. Wendell, M. Zaharia. Learning Spark: Lightning-Fast Big Data Analysis. 2015. 276 с.

ДОДАТОКИ

Б.1 Програмний код файлу ua.bigdata.example.Main.scala

```
package ua.bigdata.example
import com.typesafe.scalalogging.LazyLogging
import ua.bigdata.example.microservice.Clusterer
object Main extends LazyLogging {
  def main(args: Array[String]): Unit = {
    new Clusterer("localhost", 9090)
  }
}
```

Б.2 Програмний код файлу ua.bigdata.example.hash.Hasher.scala

```
package ua.bigdata.example.hash
import scala.util.Random
import org.apache.spark.mllib.linalg.SparseVector
/**
 * simple hashing function. defined by ints a, b, p, m
 * where a and b are seeds with a > 0.
 * p is a prime number, >= u (largest item in the universe)
 * m is the number of hash bins
 */
class Hasher(a : Int, b : Int, p : Int, m : Int) extends Serializable {
  override def toString: String = "(" + a + ", " + b + ")"
  def hash(x : Int) : Int = {
```

```
    ( ((a.longValue*x) + b) % p ).intValue % m
  }
  def minhash(v : SparseVector) : Int = {
    v.indices.map(i => hash(i)).min
  }

}
object Hasher {
   /** create a new instance providing p and m. a and b random numbers mod p */
   def create(p : Int, m : Int) = new Hasher(a(p), b(p), p, m)



  /** create a seed "a" */
  def a(p : Int) : Int = {
    val r = new Random().nextInt(p)
    if(r == 0)
      a(p)
    else
      r
  }

  /** create a seed "b" */
  def b(p : Int) : Int = {
```

```
    new Random().nextInt(p)

  }


}
```

Б.3 Програмний код файлу ua.bigdata.example.hash.LSH.scala

```
package ua.bigdata.example.hash

import org.apache.spark.mllib.linalg.SparseVector

import org.apache.spark.rdd.RDD

class LSH(data : RDD[SparseVector], p : Int, m : Int, numRows : Int, numBands : Int, minClusterSize : Int) extends Serializable {


  /** run LSH using the constructor parameters */

  def run() : LSHModel = {


    //create a new model object

    val model = new LSHModel(p, m, numRows)

    //preserve vector index

    val zdata = data.zipWithIndex().cache()


    //compute signatures from matrix

    // - hash each vector <numRows> times

    // - position hashes into bands. we'll later group these signature bins and has them as well

    //this gives us ((vector idx, band#), minhash)
```

```
    val signatures = zdata.flatMap(v => model.hashFunctions.flatMap(h => List(((v._2, h._2 % numBands),h._1.minhash(v._1))))).cache()

    //reorganize data for shuffle

    //this gives us ((band#, hash of minhash list), vector id)

    //groupByKey gives us items that hash together in the same band

    model.bands = signatures.groupByKey().map(x => ((x._1._2, x._2.hashCode), x._1._1)).groupByKey().cache()



    //we only want groups of size >= <minClusterSize>

    //(vector id, cluster id)

    model.vector_cluster = model.bands.filter(x => x._2.size >= minClusterSize).map(x => x._2.toList.sorted).distinct().zipWithIndex().map(x => x._1.map(y => (y, x._2))).flatMap(x => x.grouped(1)).map(x => x.head).cache()



    //(cluster id, vector id)

    model.cluster_vector = model.vector_cluster.map(x => x.swap).cache()



    //(cluster id, List(vector))

    model.clusters = zdata.map(x => x.swap).join(model.vector_cluster).map(x => (x._2._2, x._2._1)).groupByKey().cache()



    //compute the jaccard similarity of each cluster

    model.scores = model.clusters.map(row => (row._1, jaccard(row._2.toList))).cache()



    model

  }
```

```
  /** compute a single vector against an existing model */

  def compute(data : SparseVector, model : LSHModel, minScore : Double) : RDD[(Long, Iterable[SparseVector])] = {

    model.clusters.map(x => (x._1, x._2++List(data))).filter(x => jaccard(x._2.toList) >= minScore)

  }



  /** compute jaccard between two vectors */

  def jaccard(a : SparseVector, b : SparseVector) : Double = {

    val al = a.indices.toList

    val bl = b.indices.toList

    al.intersect(bl).size / al.union(bl).size.doubleValue

  }



  /** compute jaccard similarity over a list of vectors */

  def jaccard(l : List[SparseVector]) : Double = {

    l.foldLeft(l.head.indices.toList)((a1, b1) => a1.intersect(b1.indices.toList.asInstanceOf[List[Nothing]])).size /

    l.foldLeft(List())((a1, b1) => a1.union(b1.indices.toList.asInstanceOf[List[Nothing]])).distinct.size.doubleValue

  }



}
```

Б.4 Програмний код файлу ua.bigdata.example.hash.LSHModel.scala

```
package ua.bigdata.example.hash
```

```
import org.apache.spark.mllib.linalg.SparseVector
import org.apache.spark.rdd.RDD
import scala.collection.mutable.ListBuffer
class LSHModel(p : Int, m : Int, numRows : Int) extends Serializable {

  /** generate rows hash functions */
  private val _hashFunctions = ListBuffer[Hasher]()
  for (_ <- 0 until numRows)
    _hashFunctions += Hasher.create(p, m)
  final val hashFunctions : List[(Hasher, Int)] = _hashFunctions.toList.zipWithIndex

  /** the signature matrix with (hashFunctions.size signatures) */
  var signatureMatrix : RDD[List[Int]] = _
  /** the "bands" ((hash of List, band#), row#) */
  var bands : RDD[((Int, Int), Iterable[Long])] = _

  /** (vector id, cluster id) */
  var vector_cluster : RDD[(Long, Long)] = _

  /** (cluster id, vector id) */
  var cluster_vector : RDD[(Long, Long)] = _

  /** (cluster id, List(Vector) */
  var clusters : RDD[(Long, Iterable[SparseVector])] = _
```

```
  /** jaccard cluster scores */
  var scores : RDD[(Long, Double)] = _

  /** filter out scores below threshold. this is an optional step.*/
  def filter(score : Double) : LSHModel = {

    val scores_filtered = scores.filter(x => x._2 > score)
    val clusters_filtered = scores_filtered.join(clusters).map(x => (x._1, x._2._2))
    val cluster_vector_filtered = scores_filtered.join(cluster_vector).map(x => (x._1, x._2._2))
    scores = scores_filtered
    clusters = clusters_filtered
    cluster_vector = cluster_vector_filtered
    vector_cluster = cluster_vector.map(x => x.swap)
    this
  }

}
```

Б.5 Програмний код файлу ua.bigdata.example.microservice.Clusterer.scala

```
package ua.bigdata.example.microservice
import java.io.{BufferedWriter, FileWriter}
import akka.actor.ActorSystem
import akka.http.scaladsl.model.{HttpResponse, StatusCodes}
```

```
import akka.http.scaladsl.server.Directives.{complete, parameters, path}

import akka.http.scaladsl.server.Route

import akka.stream.ActorMaterializer

import com.typesafe.config.{Config, ConfigFactory}

import com.typesafe.scalalogging.LazyLogging

import org.apache.spark.SparkConf

import org.apache.spark.rdd.RDD

import org.apache.spark.sql.SparkSession

import bigdata.microservice._

import bigdata.partitioners.Partitioner

import bigdata.task.TaskSync

import scala.concurrent.{ExecutionContext, Future}

class Clusterer(override val host: String, override val port: Int) extends

  BigDataMicroserviceSync[String, RDD[(Long, List[List[Int]])]] with LazyLogging
{

  override implicit lazy val actorSystem: ActorSystem = ActorSystem("Clusterer")

  override implicit lazy val mat: ActorMaterializer = ActorMaterializer()

  override implicit lazy val ec: ExecutionContext = actorSystem.dispatcher

  override lazy val partitioner: Partitioner[String] = new ConstantPartitioner(100)

  override lazy val config: Config = ConfigFactory.load()

  override def callback(input: String, value: RDD[(Long, List[List[Int]])]):
Future[Unit] = Future.unit.map { _ =>

    val writer = new BufferedWriter(new FileWriter("result_" +
scala.util.Random.alphanumeric

      .take(5).mkString + ".tsv"))
```

```
    value.toLocalIterator.foreach { x =>
      x._2.foreach { elem =>
        writer.write(s"${x._1}\t${elem.mkString(",")}\n")
      }
    }
    writer.close()
    logger.info(s"Task was completed successfully!")
  }
  override def tasker(input: String): SparkSession => TaskSync[String, RDD[(Long, List[List[Int]])]] = {
    def in(sparkSession: SparkSession) = new ClusteringTask(sparkSession, input)
    in
  }
  override lazy val route: Route = path("api") {
    parameters('path) { path =>
      queueSender(path)
      complete(HttpResponse(status = StatusCodes.OK))
    }
  }
  override def sparkConf(config: Config, input: String): SparkConf = {
    new SparkConf()
      .setMaster({
        config.getString("spark.master")
      })
```

```
      .setAppName(s"bigdata_${config.getString("branchTag")}" +

s"_${input.takeRight(10)}${scala.util.Random.alphanumeric.take(3).mkString("")}")
      .setAll(
        {
          val parts = partitioner.parts(input)
          val l = List(
            "spark.sql.shuffle.partitions" -> s"$parts",
            "spark.default.parallelism" -> s"$parts",
            "spark.executor.memory" -> config.getString("spark.executor.memory"),
            "spark.files.maxPartitionBytes" -> config.getString("spark.files.maxPartitionBytes"),
            "spark.rpc.message.maxSize" -> config.getString("spark.rpc.message.maxSize"),
            "spark.driver.port" -> config.getString("spark.driver.port"),
            "spark.driver.host" -> config.getString("spark.driver.host"),
            "spark.driver.bindAddress" -> config.getString("spark.driver.bindAddress"),
            "spark.blockManager.port" -> config.getString("spark.blockManager.port"),
            "spark.ui.port" -> config.getString("spark.ui.port"),
            "spark.submit.deployMode" -> "client",
            "spark.executor.extraClassPath" -> s"-Dconfig.resource=${config.getString("configFile")}",
            "spark.driver.extraJavaOptions" -> s"-Dconfig.resource=${config.getString("configFile")}"
          )
          if (config.getBoolean("spark.uploadJars")) {
```

```
        val additionalConfig = List(

          "spark.yarn.jars" ->

            s"target/scala-
${bigdata_info_build.BuildInfo.scalaVersion.split('.').take(2).mkString(".")}/bigdata-
example-assembly.jar",

          ("spark.hadoop.yarn.resourcemanager.hostname",
config.getString("spark.hadoop.yarn.resourcemanager.hostname")),

          ("spark.hadoop.yarn.resourcemanager.address",
config.getString("spark.hadoop.yarn.resourcemanager.address")),

          ("yarn.app.container.log.dir", "/var/log/hadoop/userLogs")

        )

        additionalConfig ::: l

      }

      else l

    }

  )

 }

}
```

Б.6 Програмний код файлу ua.bigdata.example.microservice.ClustererPartitioner.scala

```
package ua.bigdata.example.microservice

import bigdata.partitioners.Partitioner

object ClustererPartitioner extends Partitioner[String] {

 override def parts(input: String): Int = {

   val min = Runtime.getRuntime.availableProcessors()

   val source = scala.io.Source.fromFile(input)
```

```
    val i = source.getLines().size
    source.close()
    math.max(min, math.round(i.toDouble/10000)).intValue()
  }
}
```

Б.7 Програмний код файлу ua.bigdata.example.microservice.ClusteringTask.scala

```
package ua.bigdata.example.microservice
import org.apache.spark.mllib.linalg.{SparseVector, Vectors}
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.SparkSession
import ua.bigdata.example.hash.{LSH, LSHModel}
import bigdata.task.TaskSync
class ClusteringTask(sparkSession: SparkSession, override val input: String) extends TaskSync[String, RDD[(Long, List[List[Int]])]] {
  val inputRDD: RDD[(List[Int], String)] = sparkSession.sparkContext.parallelize(10000000 to 10100000)
    .map(_.toString)
    .filter(_.length % 2 == 0)
    .map(x => x.sliding(2, 2).toList.map(_.toInt).distinct.sorted -> x)
    .reduceByKey((a, _) => a)
  val vctr: RDD[SparseVector] = inputRDD.map(r => r._1.map(i => (i, 1.0))).map(a => Vectors.sparse(65535, a).asInstanceOf[SparseVector])
  val lsh = new LSH(data = vctr, p = 65537, m = 1000, numRows = 2000, numBands = 100, minClusterSize = 2)
  val model: LSHModel = lsh.run()
```

```
 override val result: RDD[(Long, List[List[Int]])] = model.clusters.zipWithIndex().map(x => x._2 -> x._1._2.toList.map(x =>

   x.indices.toList))

}
```

Б.8 Програмний код файлу ua.bigdata.example.microservice.ConstantPartitioner.scala

```
package ua.bigdata.example.microservice

import bigdata.partitioners.Partitioner

class ConstantPartitioner(partsC: Int) extends Partitioner[Any] {

 override def parts(input: Any): Int = partsC

}
```

Б.9 Програмний код файлу bigdata.microservice.BigDataMicroservice.scala

```
package bigdata.microservice

import akka.actor.{ActorRef, ActorSystem, Props}

import akka.http.scaladsl.Http

import akka.http.scaladsl.server.Route

import akka.stream.ActorMaterializer

import com.typesafe.config.Config

import com.typesafe.scalalogging.Logger

import org.apache.spark.SparkConf

import org.apache.spark.sql.SparkSession

import bigdata.partitioners.Partitioner

import bigdata.queue.{QueuerAsync, QueuerSync}

import bigdata.task.{TaskAsync, TaskSync}

import scala.concurrent.{ExecutionContext, Future}
```

```
sealed trait BigDataMicroservice[Input, T] {
  val partitioner: Partitioner[Input]
  implicit val actorSystem: ActorSystem
  implicit val mat: ActorMaterializer
  protected val logger: Logger
  def sparkConf(config: Config, input: Input): SparkConf
  val config: Config
  protected lazy val sparker: Sparker[Input] = new Sparker[Input](config, sparkConf)
  def callback(input: Input, value: T): Future[Unit]
  val queueActor: ActorRef
  def queueSender(input: Input)
  val route: Route
  val port: Int
  val host: String
  Http().bindAndHandle(route, host, port)
}
trait BigDataMicroserviceAsync[Input, T] extends BigDataMicroservice[Input, T] {
  implicit val ec: ExecutionContext
  private lazy val queuer: QueuerAsync[Input, T] = new QueuerAsync[Input, T]
  final lazy val queueActor: ActorRef = actorSystem.actorOf(Props(new queuer.QueueAsync(callback, sparker)))
  def tasker(input: Input): SparkSession => TaskAsync[Input, T]
  final def queueSender(input: Input): Unit = {
    queueActor ! queuer.TaskAsyncHandler(input -> tasker(input))
```

```
  }
}
trait BigDataMicroserviceSync[Input, T] extends BigDataMicroservice[Input, T] {
  private lazy val queuer: QueuerSync[Input, T] = new QueuerSync[Input, T]
  implicit val ec: ExecutionContext
  final lazy val queueActor: ActorRef = actorSystem.actorOf(Props(new queuer.QueueSync(callback, sparker)))
  def tasker(input: Input): SparkSession => TaskSync[Input, T]
  final def queueSender(input: Input): Unit = {
    queueActor ! queuer.TaskSyncHandler(input -> tasker(input))
  }
}
```

Б.10 Програмний код файлу bigdata.microservice.Sparker.scala

```
package bigdata.microservice
import com.typesafe.config.Config
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql.{SQLContext, SparkSession}
import scala.util.Random
class Sparker[Input](config: Config, sparkConfiger: (Config, Input) => SparkConf) {
  private val random = new Random()
  def createSessionNG(input: Input): SparkSession = {
    val sparkConf: SparkConf = sparkConfiger(config, input)
    import org.apache.hadoop.security.UserGroupInformation
    import org.apache.spark.deploy.SparkHadoopUtil
```

```
UserGroupInformation.setConfiguration(SparkHadoopUtil.get.newConfiguration(spa
rkConf))

    val credentials = UserGroupInformation.getLoginUser.getCredentials

    SparkHadoopUtil.get.addCurrentUserCredentials(credentials)

    val sparkContext = new SparkContext(sparkConf)

    val sqlContext = new SQLContext(sparkContext)

    val spark = sqlContext.sparkSession

    val hadoopConfig = sparkContext.hadoopConfiguration

    hadoopConfig.set("fs.hdfs.impl",
classOf[org.apache.hadoop.hdfs.DistributedFileSystem].getName)

    hadoopConfig.set("fs.file.impl",
classOf[org.apache.hadoop.fs.LocalFileSystem].getName)

    hadoopConfig.set("dfs.replication", "1")

    spark.sparkContext.setLogLevel("ERROR")

    spark

  }

}
```

Б.11 Програмний код файлу bigdata.partitioners.Partitioner.scala

```
package bigdata.partitioners

trait Partitioner[-Input] {

  def parts(input: Input): Int

}

object DefaultPartitioner extends Partitioner[Any] {

  def parts(input: Any): Int = Runtime.getRuntime.availableProcessors()

}
```

Б.12 Програмний код файлу bigdata.queue.QueuerAsync.scala

```
package bigdata.queue

import akka.actor.Actor

import com.typesafe.scalalogging.LazyLogging

import org.apache.spark.sql.SparkSession

import bigdata.microservice.Sparker

import bigdata.task.TaskAsync

import scala.collection.immutable.Queue

import scala.concurrent.{ExecutionContext, Future}

import scala.util.{Failure, Success}

class QueuerAsync[Input, T] {

  class QueueAsync(callback: (Input, T) => Future[Unit], sparker: Sparker[Input])

                  (implicit val ec: ExecutionContext) extends Actor with
LazyLogging {

    override def receive: Receive = {

      case TaskAsyncHandler(task) =>

        context.become(handler(task, Queue.empty[(Input, SparkSession =>
TaskAsync[Input, T])]))

        self ! StartCurrent

      case any => logger.warn(s"Unknown message received:
${Option(any).map(_.toString).getOrElse("null")}")

    }

    def handler(currentTask: (Input, SparkSession => TaskAsync[Input, T]),

                queue: Queue[(Input, SparkSession => TaskAsync[Input, T])]): Receive =
{

      case TaskAsyncHandler(task) =>
```

```
      context.become(handler(currentTask, queue.enqueue(task)))
    case RemoveCurrent(sparkSession) =>
      sparkSession.stop()
      if (queue.isEmpty)
        context.become(receive)
      else {
        val (nT, nQ) = queue.dequeue
        context.become(handler(nT, nQ))
        self ! StartCurrent
      }
    case StartCurrent =>
      val spark = sparker.createSessionNG(currentTask._1)
      val mF = currentTask._2(spark).finalFuture.flatMap(callback(currentTask._1, _))
      mF.onComplete {
        case Success(_) =>
          logger.info("Task completed successfully")
        case Failure(exception) =>
          logger.error(s"Task failed with exception: ${exception.getMessage}\n" + exception.getStackTraceString)
      }
      mF.onComplete(_ => self ! RemoveCurrent(spark))
    case any => logger.warn(s"Unknown message received: ${Option(any).map(_.toString).getOrElse("null")}")
  }
```

```
  }
  case class TaskAsyncHandler(task: (Input, SparkSession => TaskAsync[Input, T]))
  private case class RemoveCurrent(sparkSession: SparkSession)
  private case object StartCurrent
}
```

Б.13 Програмний код файлу bigdata.queue.QueuerSync.scala

```
package bigdata.queue
import akka.actor.Actor
import com.typesafe.scalalogging.LazyLogging
import org.apache.spark.sql.SparkSession
import bigdata.microservice.Sparker
import bigdata.task.TaskSync
import scala.collection.immutable.Queue
import scala.concurrent.{ExecutionContext, Future}
import scala.util.{Failure, Success}
class QueuerSync[Input, T] {
  class QueueSync(callback: (Input, T) => Future[Unit], sparker: Sparker[Input])
                 (implicit val ec: ExecutionContext) extends Actor with LazyLogging
{
    override def receive: Receive = {
      case TaskSyncHandler(task) =>
        context.become(handler(task, Queue.empty[(Input, SparkSession =>
TaskSync[Input, T])]))
        self ! StartCurrent
```

```
    case any => logger.warn(s"Unknown message received: ${Option(any).map(_.toString).getOrElse("null")}")
  }
  def handler(currentTask: (Input, SparkSession => TaskSync[Input, T]),
              queue: Queue[(Input, SparkSession => TaskSync[Input, T])]): Receive = {
    case TaskSyncHandler(task) =>
      context.become(handler(currentTask, queue.enqueue(task)))
    case RemoveCurrent(sparkSession) =>
      sparkSession.stop()
      if (queue.isEmpty)
        context.become(receive)
      else {
        val (nT, nQ) = queue.dequeue
        context.become(handler(nT, nQ))
        self ! StartCurrent
      }
    case StartCurrent =>
      val spark = sparker.createSessionNG(currentTask._1)
      val mF = Future.unit.map(_ => callback(currentTask._1, currentTask._2(spark).result))
      mF.onComplete {
        case Success(_) =>
          logger.info("Task completed successfully")
        case Failure(exception) =>
```

```
        logger.error(s"Task failed with exception: ${exception.getMessage}\n" +
exception.getStackTraceString)

      }

      mF.onComplete(_ => self ! RemoveCurrent(spark))

    case any => logger.warn(s"Unknown message received:
${Option(any).map(_.toString).getOrElse("null")}")

    }

  }

  case class TaskSyncHandler(task: (Input, SparkSession => TaskSync[Input, T]))

  private case class RemoveCurrent(sparkSession: SparkSession)

  private case object StartCurrent

}
```

Б.14 Програмний код файлу bigdata.task.TaskAsync.scala

```
package bigdata.task

import scala.concurrent.Future

trait TaskAsync[Input, T] {

  val input: Input

  val finalFuture: Future[T]

}
```

Б.15 Програмний код файлу bigdata.task.TaskSync.scala

```
package bigdata.task

trait TaskSync[Input, T] {

  val input: Input

  val result: T

}
```

Б.16 Програмний код файлу build.sbt

```
import sbtbuildinfo.BuildInfoPlugin.autoImport.buildInfoPackage

lazy val commonSettings = Seq(
  organization := "ua.kpi",
  version := "1.1",
  scalaVersion := "2.12.8"
)

lazy val library = (project in file("bachelor-diploma-library")).settings(
  commonSettings,
  libraryDependencies ++= Seq(
    "org.apache.spark" %% "spark-core" % "2.4.3",
    "org.apache.spark" %% "spark-sql" % "2.4.3",
    "org.apache.spark" %% "spark-mllib" % "2.4.3"
  ),
  libraryDependencies += "com.typesafe" % "config" % "1.3.4",
  buildInfoKeys := Seq[BuildInfoKey](name, version, scalaVersion, sbtVersion),
  buildInfoPackage := "bigdata_info_build",
  assemblyJarName in assembly := "bigdata-assembly.jar",
  libraryDependencies += "ch.qos.logback" % "logback-classic" % "1.2.3",
  libraryDependencies += "com.typesafe.scala-logging" %% "scala-logging" % "3.9.2",
  libraryDependencies ++= Seq(
```

```
    "com.typesafe.akka" %% "akka-actor" % "2.5.22",
    "com.typesafe.akka" %% "akka-testkit" % "2.5.22" % Test
  ),
  libraryDependencies ++= Seq(
    "com.typesafe.akka" %% "akka-http" % "10.1.8",
    "com.typesafe.akka" %% "akka-http-testkit" % "10.1.8" % Test
  ),
  libraryDependencies ++= Seq(
    "com.typesafe.akka" %% "akka-stream" % "2.5.22",
    "com.typesafe.akka" %% "akka-stream-testkit" % "2.5.22" % Test
  ),
  libraryDependencies ~= {
    _.map(_.exclude("org.slf4j", "slf4j-log4j12"))
  }
).enablePlugins(BuildInfoPlugin)


lazy val example = (project in file("bachelor-diploma-example")).settings(
  commonSettings,
  assemblyJarName in assembly := "bigdata-example-assembly.jar"
).dependsOn(library)
```