МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 53.00.00.000 ПЗ

Група ІПм-23-2

Непорадний Тарас

2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Непорадний Тарас Михайлович

(прізвище, ім'я, по батькові)

УДК 004.942

(індекс)

МАГІСТЕРСЬКА РОБОТА

Моделі, методи та засоби фреймворків великих даних

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Непорадний Т.М.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Тимків Дмитро Федорович, д.т.н., професор

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2024 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Непорадному Тарасу Михайловичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “ Моделі, методи та засоби фреймворків великих даних”

керівник проекту (роботи) Тимків Дмитро Федорович, д.т.н., професор

затверджені наказом закладу вищої освіти від “ 22 ” листопада 2024 р. № 781/7

2. Строк подання студентом проекту (роботи) 15 грудня 2024 р.

3. Вихідні дані до проекту (роботи) Теоретичні концепції та формальні моделі побудови та функціонування інформаційних технологій обробки великих даних

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Аналіз предметної області застосування великих даних
2. Дослідження методів, архітектур та інструментів обробки великих даних
3. Дослідження методів та інструментів передачі великих об’ємів даних
4. Застосування методів та засобів фреймворків обробки великих даних

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Концепція 4 vs big data (рис. 1.1)
2. Архітектура процесу аналітики великих даних у реальному часі (рис. 1.2)
3. Архітектура Hadoop Distributed File System (HDFS) (рис. 2.1)
4. Архітектура YARN (рис. 1.4)
5. Архітектура Zookeeper (рис. 1.5)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц., к.т.н. Вовк Р.Б.	

7. Дата видачі завдання 04 вересня 2024 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури по темі магістерської роботи	15.09.2024	виконано
2	Аналіз концепцій та алгоритмів предметної області	29.09.2024	виконано
3	Дослідження предметної області застосування великих даних	15.10.2024	виконано
4	Дослідження методів, архітектур та інструментів обробки великих даних	08.11.2024	виконано
5	Дослідження методів та інструментів передачі великих об’ємів даних	20.11.2024	виконано
6	Застосування методів та засобів фреймворків обробки великих даних	01.12.2024	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2024	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 78 с., 30 рис., 50 джерел.

Тема: Моделі, методи та засоби фреймворків великих даних

Об'єкт дослідження: процеси обробки великих даних у реальному часі.

Мета роботи: оцінка продуктивності, масштабованості та гнучкості при обробці великих обсягів даних.

Предмет дослідження: моделі, методи та засоби обробки великих даних за допомогою фреймворків Apache NiFi та Apache Storm.

Результати дослідження

В роботі було розроблено методику, що включає побудову потоку даних і створення тестового сценарію для оцінки ефективності та масштабованості зазначених платформ

Висновок

Завдяки цьому дослідженню стало можливим визначити оптимальні методи інтеграції обраних платформ для забезпечення максимальної продуктивності та адаптивності при роботі з великими даними.

ВЕЛИКІ ДАНІ, APACHE NIFI, APACHE STORM, ОБРОБКА ДАНИХ У РЕАЛЬНОМУ ЧАСІ, ФРЕЙМВОРКИ, ПРОДУКТИВНІСТЬ, МАСШТАБОВАНІСТЬ, ТЕСТУВАННЯ.

ABSTRACT

Master Thesis: 78 pp., 30 fig., 50 sources.

Thesis Subject: Models, methods and tools of big data frameworks

Object of the study: real-time big data processing processes.

Purpose of the work: assessment of performance, scalability and flexibility when processing large amounts of data.

Subject of the study: models, methods and tools of big data processing using Apache NiFi and Apache Storm frameworks.

Research results

The work developed a methodology that includes building a data flow and creating a test scenario to assess the efficiency and scalability of the specified platforms

Conclusion

Thanks to this study, it became possible to determine the optimal methods of integrating the selected platforms to ensure maximum performance and adaptability when working with big data.

BIG DATA, APACHE NIFI, APACHE STORM, REAL-TIME DATA PROCESSING, FRAMEWORKS, PERFORMANCE, SCALABILITY, TESTING.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

HDFS - Hadoop Distributed File System

HDF - Hortonworks DataFlow

HDP - Hortonworks Data Platform

XBRL - eXtensible Business Reporting Language

ВСТУП

Актуальність теми.

У сучасному світі обсяг даних стрімко зростає, створюючи нові виклики для їх обробки, аналізу та зберігання. Згідно з оцінками, до 2025 року обсяг створених і спожитих даних у світі перевищить 180 зетабайтів, причому значна частина цих даних потребуватиме обробки в реальному часі. Це актуально для таких сфер, як фінанси, охорона здоров'я, енергетика, роздрібна торгівля та промисловість, де рішення, прийняті на основі оперативного аналізу даних, забезпечують конкурентні переваги.

Зростання складності обробки даних пов’язане з їх різноманітністю, швидкістю генерації та великими обсягами, що робить традиційні підходи недостатньо ефективними. Використання фреймворків обробки великих даних, таких як Apache NiFi та Apache Storm, дозволяє вирішувати ці виклики, забезпечуючи високу швидкість, масштабованість і можливість налаштування для конкретних завдань. Ці платформи активно застосовуються для аналізу потоків даних із сенсорів, моніторингу бізнес-процесів, оптимізації логістичних систем і впровадження інтелектуальних рішень в інтернеті речей (IoT).

Актуальність теми також обумовлена необхідністю оптимізації ресурсів при обробці великих даних. Розробка підходів, які дозволяють забезпечити максимальну продуктивність за мінімальних витрат, є критично важливою для ефективного використання сучасних кластерних систем. Важливо також розуміти, що обробка великої кількості невеликих файлів, зокрема у форматі XML (наприклад, XBRL), має свої специфічні виклики, зокрема оптимізацію доступу до файлів і мінімізацію затримок.

Водночас, вибір платформи для обробки даних залежить не лише від продуктивності, але й від інших факторів, таких як універсальність, зручність використання, простота інтеграції з іншими системами та можливість адаптації до різних сценаріїв. Це робить дослідження порівняльних

характеристик фреймворків критично важливим для розробників і дослідників, які шукають оптимальні рішення для своїх завдань.

Таким чином, актуальність дослідження полягає у зростаючій потребі в ефективних методах обробки великих даних у реальному часі, а також у необхідності визначення найбільш придатних інструментів для специфічних завдань, пов’язаних із їхньою обробкою. Результати роботи сприяють подальшому розвитку інформаційних систем і впровадженню інноваційних рішень у різних галузях економіки.

Метою дослідження є оцінка продуктивності, масштабованості та гнучкості при обробці великих обсягів даних.

Об'єкт дослідження - процеси обробки великих даних у реальному часі.

Предмет дослідження - моделі, методи та засоби обробки великих даних за допомогою фреймворків Apache NiFi та Apache Storm.

Задачі дослідження:

1. Проаналізувати існуючі фреймворки обробки великих даних.

2. Визначити критерії вибору фреймворків для обробки великих даних у реальному часі.

3. Розробити методику тестування продуктивності та масштабованості фреймворків.

4. Провести експериментальне дослідження з використанням Apache NiFi та Apache Storm.

5. Порівняти результати тестування з точки зору швидкості, масштабованості та зручності використання.

Методи дослідження

- Аналіз та синтез інформаційних джерел щодо платформ для обробки великих даних.

- Експериментальне тестування продуктивності фреймворків.

- Методологія бенчмаркінгу для оцінки швидкості та масштабованості обробки даних.

- Порівняльний аналіз результатів роботи фреймворків.

Наукова новизна отриманих результатів

Проведено порівняльний аналіз фреймворків Apache NiFi та Apache Storm для задач обробки великої кількості невеликих файлів у реальному часі. Розроблено методику тестування, яка включає побудову потоку даних, специфікацію кластера та оцінку масштабованості платформ.

Практичне значення результатів

Результати дослідження можуть бути використані при виборі фреймворків для обробки даних у реальному часі. Запропонована методика тестування стане корисною для проектування інформаційних систем, що обробляють великі обсяги даних, а висновки щодо продуктивності платформ можуть слугувати рекомендаціями для розробників.

Структура магістерської роботи. Робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 78 сторінок, і містить 30 рисунків, список використаних джерел із 50 найменувань.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ЗАСТОСУВАННЯ ВЕЛИКИХ ДАНИХ

1.1. Постановка проблеми дослідження

Великі дані - це концепція, яка стрімко розвивається. Оскільки генерується і збирається все більше і більше даних, зростає потреба в ефективних рішеннях, які можна використовувати для обробки всіх цих даних з метою отримання з них користі.

Зі збільшенням обсягу даних у світі зростає складність їх обробки та візуалізації. Розподілена обчислювальна платформа Hadoop включає пару фреймворків для обробки великих обсягів даних. До даних, які підлягають обробці, пред'являються різні вимоги. Деякі типи даних може знадобитися обробляти в режимі реального часу, інші можуть бути більш корисними для аналізу, обробляючи дані пакетами.

Під час написання цієї роботи не було знайдено контрольних показників ефективності цих фреймворків при обробці великих обсягів відносно невеликих файлів, таких як фінансові документи, показання датчиків тощо, які обробляються в режимі реального часу. Це ускладнює визначення ефективності різних фреймворків один проти одного.

Існує багато фреймворків для обробки великих даних у режимі реального часу, і потрібен спосіб вибрати правильний для роботи.

Великі дані не обов'язково означають, що частини даних, які надходять, є великими самі по собі. Накопичення великої кількості невеликих даних може швидко стати великими даними зі збільшенням кількості джерел. Зі збільшенням джерел має бути спосіб ефективної обробки даних у режимі реального часу. Ця робота досліджувала та перевіряла продуктивність і масштабованість під час обробки великих обсягів даних, які надходять невеликими порціями. Тестування було зроблено для того, щоб

знайти відповідного кандидата для реалізації прототипу адаптера даних для прийому фінансових даних.

Основною метою проекту було дослідження різних способів обробки великих даних і тестування аспектів швидкості та масштабованості. З огляду на це було зроблено наступні розмежування.

Оброблятиметься лише один формат, а саме XBRL, формат для передачі фінансової інформації. Передбачається, що значення, яке можна отримати з аналізу даних у форматі XBRL, уже відомо.

Тестуватиметься лише один розподіл великих даних, вибраний під назвою Hortonworks Data Platform, HDP. HDP вже був доступний на кількох вузлах кластера, а також включав численні інструменти, які могли б досягти цілей цієї роботи.

Не всі аспекти обробки великих даних перевірятимуться. Перевірятимуть лише швидкість, масштабованість і семантику доставки. Тести на масштабованість і продуктивність будуть проведені на 1, 2, 3 і 4 вузлах кластера для обробки даних XBRL. Лише два фреймворки для виконання обчислень потоку в реальному часі в HDP були перевірені через часові обмеження.

1.2. Опис та дослідження концепції великих даних

Визначення великих даних, яке іноді називають Big Data, є дещо суперечливим. Здається, загальне та загальне визначення таке: дані такого масштабу, що традиційні методи та програмне забезпечення не можуть ефективно їх обробляти. Ще в 2001 році в дослідницькому звіті [1], що стосується великих даних, було визначено великі дані як спосіб роботи з постійно зростаючими обсягами даних. Було визначено, що це спосіб роботи з обсягом даних, швидкістю передачі даних і різноманітністю даних [2].

Однак у 2011 році IDC, великий гравець у сфері великих даних, визначила великі дані як: «Технології великих даних описують нове

покоління технологій і архітектур , призначених для економічного вилучення цінності з дуже великих обсягів різноманітних даних, забезпечуючи високошвидкісне захоплення, відкриття та/або аналіз ».

Додавання четвертого аспекту, який слід було розглянути: значення [3]. Ці різні способи роботи з великими даними: обсяг, швидкість, різноманітність і цінність іноді називають 4 Vs (рис. 1.1).

Рис. 1.1. Концепція 4 vs big data

За останні 10 років кількість даних різко зросла в різних сферах [4]. За оцінками, найближчим часом кількість даних, що зберігаються, подвоюватиметься кожні два роки [3]. Все більше компаній впроваджують рішення для великих даних у свої системи [5], і потреба в аналітиках великих даних і науковцях з великих даних зростатиме [6].

Величезні обсяги даних можна використовувати для різноманітних цілей. Компанії можуть використовувати великі дані для покращення різних аспектів свого бізнесу. Наприклад, це може покращити орієнтацію на

клієнтів і оптимізувати бізнес-процеси компаній. Великі дані приносять користь не лише підприємствам і бізнесу, але обробка великих даних також може допомогти покращити науку, дослідження, спорт тощо.

Більш конкретний приклад – це приклад із відчутними результатами. Sociometric Solutions — це компанія, яка вставляє датчики в бейджі з іменами співробітників. Метою цих датчиків було виявлення соціальної динаміки. Одним із клієнтів Sociometric Solutions був Bank of America, вони виявили, що найефективніші в їхніх колл-центрах були ті, хто робив перерви разом. Започаткувавши політику групового розриву, вони підвищили ефективність робітників на 23 відсотки [7].

Робота з великими даними пов’язана з певними проблемами, які необхідно враховувати та вирішувати. Класифікуючи різні виклики за відповідними областями, можна сформувати 4 vs, більш чітку картину того, що собою представляють деякі виклики.

Велика кількість даних, які зберігаються та є легкодоступними для компаній, відкриває можливість отримання цінності з даних. Відповідно до звіту компанії, зробленого KPMG [8], випадки використання повинні бути визначені заздалегідь. Пошук шаблонів, швидше за все, призведе до знайденого шаблону, але він може не мати жодної цінності. Перш ніж починати спроби екстраполювати будь-яке значення із накопичених даних, необхідно вибрати мету машини для обробки великих даних . У деяких випадках важливо визначити значення ще до того, як дані будуть зібрані. Якщо дані зберігаються лише з метою пошуку в них цінності, може не знадобитися зберігати всі дані.

Хоча існують певні стандартні формати, такі як JSON і XML, це лише специфікації того, як дані структуровані у файлах. Немає правил щодо того, які дані містяться та як їх слід інтерпретувати. Те саме стосується схем баз даних, вони відповідають певному типу моделювання, але дані, які зберігаються, можна структурувати багатьма способами. Справитися з цими невідповідностями може бути складно [1]. Навіть дані, які використовуються

з певною метою, можуть зберігатися різними способами різними постачальниками та виробниками даних.

Сучасним підприємствам недостатньо знаходити та аналізувати свої дані. Вони повинні швидко знаходити відповідні дані та швидко їх обробляти. Одним з рішень є використання високоякісного обладнання. Використання збільшеної пам'яті та паралельної обробки може значно збільшити швидкість обробки даних. Інший підхід полягає в тому, щоб помістити дані в пам'ять, але використовувати для обробки кластер комп'ютерів [9]. Деякі дані мають оброблятися майже в режимі реального часу, щоб бути максимально корисними. Виявлення шахрайства або порушень дорожнього руху, наприклад, може втратити частину своєї цінності, якщо інформація не обробляється та не передається досить швидко [10]. Щоб використовувати великі дані та досягати значущих результатів, потрібні великі обсяги даних. Швидкості одноядерних процесорів стають плато [11], а обсяг даних масштабується швидше, ніж ресурси, які використовуються для їх обчислення. Десятиліттями жорсткі диски були нормою для зберігання даних, але жорсткі диски страждають від того факту, що їхній час довільного доступу занадто великий, щоб ефективно обробляти обсяг. Твердотільні накопичувачі, або SSD, тепер замінюють жорсткі диски. SSD, порівняно з HDD, не так сильно страждають від довільного доступу, і спосіб зберігання даних можна переглянути [10]. Інше зауваження , яке слід враховувати, полягає в тому, що багато компаній вже мають великі обсяги даних, початок аналізу цих даних може бути корисним, але знову ж таки, як згадувалося раніше; визначити цінність даних може бути важко.

1.3. Дослідження типів впливів, що здійснюють великі дані

У цьому розділі розглядається кілька впливів великих даних на суспільство в цілому, розглядаються такі теми, як соціально-економічний вплив, вплив на навколишнє середовище та вплив на суспільство.

1.3.1 Соціально-економічний вплив даних

Нещодавно було викрито одну з найбільших у світі юридичних фірм Mossack Fonesca, куди стався витік 11,5 мільйонів документів. Цей витік отримав назву The Panama Papers. Необхідно було придумати спосіб обробки всіх даних у документах, щоб мати можливість отримати уявлення про правопорушення. У статті, опублікованій The Guardian [13], стверджується, що документи розкрили багато способів, якими деякі заможні люди могли використовувати офшорні податкові режими.

Згідно зі статтею Techworld [13], щоб отримати розуміння великої мережі неправомірного використання, яку було виявлено, потрібно було обробляти дані таким чином, щоб їх можна було шукати та ефективно аналізувати. Люди самі по собі не змогли б побачити всі складні зв'язки.

Neo4j, компанія, яка розробляє та підтримує базу даних графів, випустила блог [14] про Панамські документи. У цьому блозі вони обговорюють, як було знайдено відповідні зв'язки, які призвели до оприлюднення викритих Панамських документів. Було використано деякі технології великих даних, зокрема Apache Solr і Apache Tika, Apache Solr — це платформа для великих даних, яка використовується для пошуку величезних обсягів даних, а Apache Tika — це інструментарій для аналізу вмісту.

За допомогою великих даних можна досягти соціально-економічного впливу. Про це йдеться в статті [15]. Очікується, що наслідки оприлюднення Панамських документів матимуть ряд наслідків, включаючи відставку певних політичних лідерів, зниження легітимності уряду та посилення тиску на офшорні центри.

1.3.2. Екологічний вплив даних

Великі дані мають хороші шанси стати універсальним інструментом у галузях досліджень навколишнього середовища [16] і мають хороші шанси

допомогти бізнесу зрозуміти вплив, який вони мають на навколишнє середовище, навіть поза бізнесом [17].

Деякі компанії, наприклад IKEA, намагаються зрозуміти вплив своєї роботи на навколишнє середовище. Вони намагаються зрозуміти, як різні зовнішні чинники, пов'язані з бізнесом, впливають на навколишнє середовище. Деякі з цих різноманітних зовнішніх факторів включають: як відбувається придбання сировини, як працюють їхні постачальники, як їхні працівники подорожують, як клієнти використовують їхні продукти тощо.

Коли було проведено аналіз двох компаній, GSK (GlaxoSmithKline) і BT (British Telecommunications) лише 20 відсотків і 8 відсотків, відповідно, їхніх вуглецевих слідів походять із власних кордонів компаній. Решта становили викиди поза межами прямого контролю компаній [17].

Збір і розуміння даних може разом зменшити вплив підприємств на навколишнє середовище. Розуміючи весь ланцюжок операцій і вплив зовнішніх факторів на сталість, компанії можуть приймати кращі рішення, коли йдеться про екологічні проблеми.

Цікавим зусиллям у сфері великих даних, пов'язаних з проблемами навколишнього середовища, є проект Global Forest Watch Інституту світових ресурсів (WGIs). Про це йдеться в дослідженні [18]. Завдяки співпраці з різними агентствами з охорони навколишнього середовища джерела даних стали багатшими, що означає більш точні дані та кращу локалізацію забруднення в дії. Global Forest Watch також використовує супутники та краудсорсинг для створення картографічної програми, якою можуть скористатися зацікавлені сторони. Отримавши карти, ви можете збільшити область, побачити, де є проблема та хто пов'язаний.

1.3.3. Соціальний вплив великих даних

Цікаві досягнення в галузі охорони здоров'я та медицини можна знайти в програмах для великих даних. У цьому розділі буде описано кілька

способів використання великих даних у галузях охорони здоров’я або, як очікується, вони будуть добре працювати в них.

У звіті центру реформування системи охорони здоров’я США зазначено, що загальносистемне впровадження технологій великих даних може зменшити щорічні витрати уряду США на охорону здоров’я на 300–450 мільярдів доларів. Скорочення на 12-17 відсотків витрат на охорону здоров'я. Вважається, що це значне скорочення витрат на охорону здоров’я можливе завдяки застосуванню розробленої ними моделі. Цінність у розумінні цієї моделі вважається балансом витрат на охорону здоров’я та результатів лікування пацієнтів. Модель передбачає, наприклад, цілеспрямовану - профілактику захворювань, покращення координації між постачальниками медичних послуг і можливість пацієнтам швидше зустріти потрібного постачальника медичних послуг. Ці вдосконалення заохочувалися протягом деякого часу, але прогрес у великих даних може посилити ефект. Kaiser Permanente, компанія в галузі охорони здоров’я, вже почала впроваджувати частину моделі та зуміла скоротити кількість відвідувань офісу на 26.2 відсотків [19].

Ще в 2009 році був відкритий новий вірус грипу (H1N1). Він швидко поширився, і медичні установи по всьому світу побоювалися спалаху пандемії. Центр контролю захворювань (CDC - Center for Disease Control) США, звернувся до лікарів з проханням інформувати їх про нові випадки захворювання на грип. Пацієнти не завжди зобов’язані негайно звернутися до лікаря, коли почуваються погано, і передача інформації від лікарів до CDC потребувала часу. Це призводило до того, що картина грипу завжди була тижневою або двотижневою. За кілька тижнів до того, як H1N1 згадали в ЗМІ, Google опублікував статтю, в якій пояснив, як можна передбачити поширення зимового грипу в США, аж до конкретних регіонів і штатів. Їхня система базувалася на обробці величезних обсягів даних. Коли вразив вірус H1N1, він виявився набагато кориснішим індикатором поточного поширення, і він міг робити це майже в реальному часі [20].

1.4. Великі дані в режимі реального часу

Великі дані самі по собі є проблемою, а обробка великих даних у режимі реального часу ще більше. У цьому розділі обговорюється історія, проблеми та випадки використання обробки великого набору даних у реальному часі. Великі дані в режимі реального часу (real-time big data) — це концепція, яка поєднує обробку величезних обсягів інформації з надзвичайно високою швидкістю, що дозволяє аналізувати та реагувати на події фактично миттєво, як тільки вони відбуваються.

У дослідженні [21], дослідники виявили, що навіть невелике зменшення затримки може допомогти компанії, що займається торгівлею акціями, заощадити мільйони протягом року. Швидкий і надійний доступ до великої кількості бухгалтерських даних корпорацій уможливить абсолютно новий рівень фінансового аналізу. Шахрайство та помилки було б незначно виявити.

Під час обробки даних у реальному часі є два важливі параметри системи щодо продуктивності: пропускна здатність і затримка. Затримка – це час, потрібний користувачеві для взаємодії з системою та отримання відповіді. Пропускна здатність - це кількість правильних операцій, які система може згенерувати протягом певного часу. І те, і інше є міркуванням для фінансового аналізу бухгалтерських документів у реальному часі. Однак пропускна здатність може вважатися більш важливою, оскільки подача таких документів займає щонайменше кілька робочих днів, допустима затримка в пару хвилин.

Початковим варіантом використання фреймворків MapReduce була пакетна обробка збережених даних; тепер подібні фреймворки використовуються для обробки даних у міру їх надходження. Два дослідники з Twitter сказали, що у такого роду обробки є дві сторони: академічна та ділова [22]. Академічний погляд полягає в тому, щоб знайти найкращі способи аналізу однорідних даних найкращим чином. Бізнес-погляд полягає

в тому, щоб створити систему, яка може аналізувати гомо- або гетерогенні дані в надійний і масштабований спосіб.

Аналітика великих даних у реальному часі — це спосіб забезпечити миттєве реагування на високошвидкісні дані з кількох джерел. Нижче інженери з обробки даних ScienceSoft описують ключові блоки архітектури та потоки даних для рішення для аналізу великих даних у реальному часі.

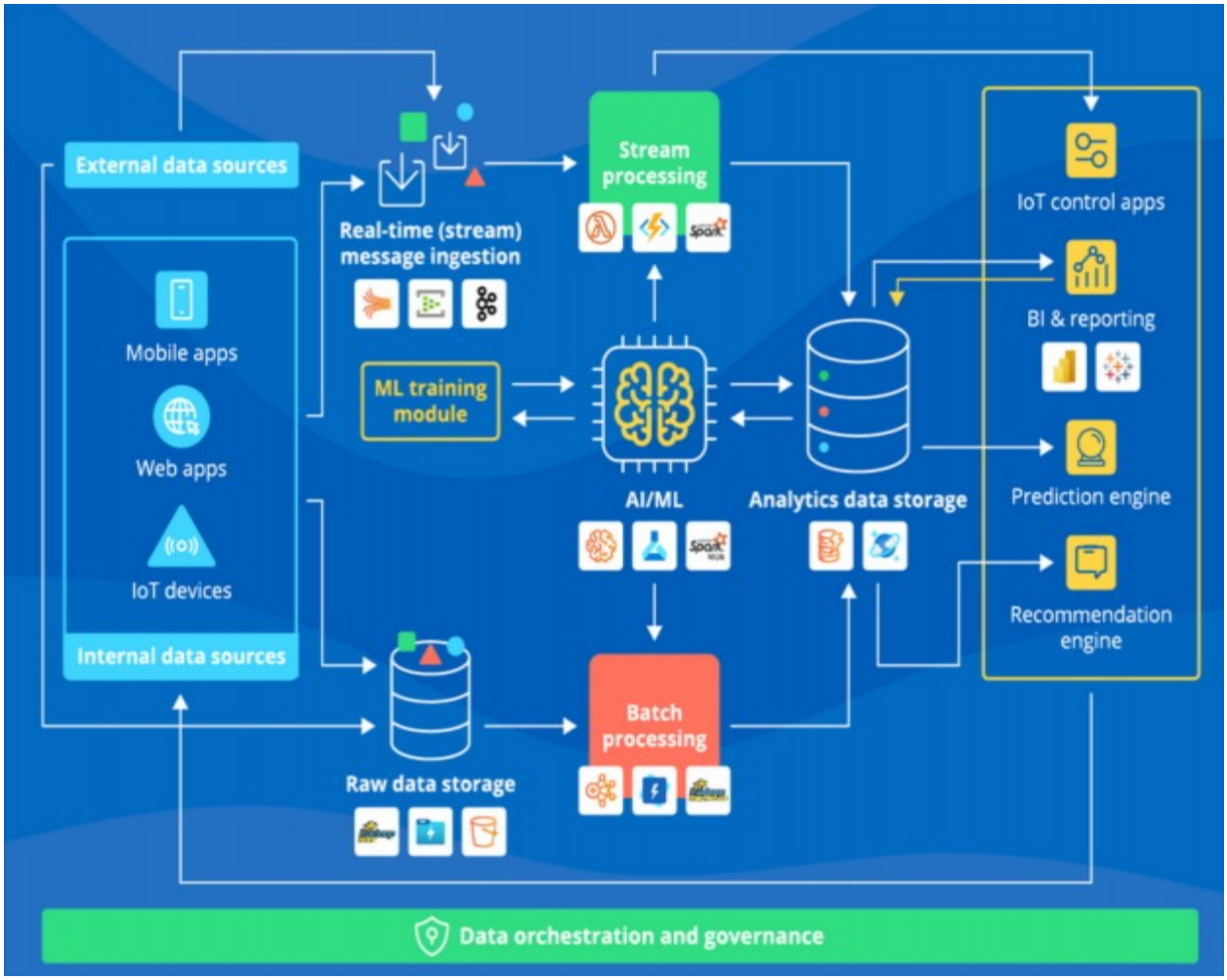

Рис. 1.2. Архітектура процесу аналітики великих даних у реальному часі

Джерела даних для рішення для аналітики великих даних у реальному часі можуть включати веб-додатки та програми для мобільних пристроїв, пристрої IoT (наприклад, датчики , переносні пристрої, приводи) та зовнішні системи (наприклад, фондові ринки, платформи соціальних мереж, системи інформації про погоду).

У більшості випадків рішення для аналітики великих даних у реальному часі матиме два рівні для обробки великих даних у реальному часі (потік) і пакетної обробки.

Шар реального часу

- Механізм прийому повідомлень у режимі реального часу отримує останні дані та надсилає їх на обробку.

- Блок потокової обробки та аналітики забезпечує реакцію на події з низькою затримкою та аналітичну інформацію в реальному часі (наприклад, персоналізовані рекомендації щодо перехресних продажів, сповіщення про патологічні показники пацієнтів).

Пакетний шар

- Сховище необроблених даних (так відоме як озеро даних) фіксує дані в початковому форматі (структурованому, неструктурованому або напівструктурованому).

- Блок пакетної обробки фільтрує, очищає, агрегує та іншим чином готує дані для аналітики відповідно до встановленого розкладу (наприклад, кожні 2 години, кожні 24 години, щотижня).

Сховище аналітичних даних (сховище даних (DWH) або база даних великих даних) зберігає уніфіковані уявлення про дані, створені блоками потокової та пакетної обробки, у високоструктурованому форматі, що відповідає вибраній моделі даних. Статистика подається до програмного забезпечення BI та бек-офісних систем. Бізнес-користувачі також можуть виконувати аналіз і дослідження даних у DWH за допомогою спеціальних запитів.

Механізм машинного навчання або штучного інтелекту (ML/AI) — це додатковий блок, який забезпечує розширену аналітику в реальному часі (наприклад, динамічне ціноутворення в електронній комерції, прогнозне технічне обслуговування у виробництві, виявлення шахрайства у фінансах). Навчальний модуль ML постійно покращує точність механізму ШІ на основі історичних даних.

Система оркестровки та керування даними автоматизує очищення даних, перетворення та інші повторювані дії з обробки даних. Він також забезпечує якість, безпеку та відповідність нормативним вимогам даних протягом усього життєвого циклу.

Висновки до розділу

У першому розділі виконано системний аналіз предметної області великих даних, зокрема їхньої концепції, впливів і можливостей використання в режимі реального часу. Розробка та впровадження великих даних є ключовим завданням для сучасних систем обробки інформації, оскільки обсяги, швидкість і різноманітність даних постійно зростають. Основними проблемами залишаються обробка, зберігання, аналіз та інтеграція даних у реальному часі.

Уточнено поняття великих даних, які визначаються як набори даних, що характеризуються трьома основними параметрами: обсяг (volume), швидкість (velocity) та різноманітність (variety). Також розглянуто параметри достовірності (veracity) та цінності (value), які підкреслюють якість і значущість отримуваної інформації. Розглянуто типи впливів великих даних.

Використання технологій реального часу дозволяє зменшити затримки в прийнятті рішень. Наприклад, аналіз даних у режимі реального часу активно використовується у фінансовій сфері, роздрібній торгівлі, системах моніторингу транспорту та охороні здоров’я.

Таким чином, перший розділ розкриває ключову роль великих даних у сучасному світі, їхній вплив на різні сфери діяльності та значення для забезпечення конкурентоспроможності організацій. Виявлені напрями дослідження створюють основу для подальшого вдосконалення моделей та методів роботи з великими даними.

РОЗДІЛ 2. ДОСЛІДЖЕННЯ МЕТОДІВ, АРХІТЕКТУР ТА ІНСТРУМЕНТІВ ОБРОБКИ ВЕЛИКИХ ДАНИХ

2.1. Огляд технологій обробки великих даних

Цей розділ має на меті представити системи, які розглядалися. Відповідно до розмежувань, розглядалися тільки системи, сумісні з Hadoop, і тестувалося лише програмне забезпечення, яке легко налаштувати та встановити на HDP (Hortonworks Data Platform). Щоб мати можливість обробляти дані, потрібно було використовувати кілька різних технологій у поєднанні одна з одною.

HDP зазвичай працює в середовищі кластера. Кластерне середовище — це налаштування, де кілька вузлів (комп'ютерів) працюють у тандемі або на локальній серверній фермі, або в одному з багатьох постачальників хмарних послуг, таких як Azure та Amazon Web Services. Для обробки великих обсягів даних одного вузла зазвичай недостатньо, тому роботу потрібно розподілити між кількома вузлами.

При аналізі великих даних потрібні різні компоненти, для конкретних цілей цієї роботи була потреба в розподіленій файловій системі, яка б дозволяла зберігати файли розподіленим способом. У кластерних середовищах зазвичай використовуються розподілені файлові системи. Їх можна використовувати, наприклад, для швидкого завантаження даних з диска [23]. Розподілена файлова система також зазвичай підтримує реплікацію, тобто якщо один із вузлів вимикається, ваші дані залишаються доступними.

Також була потреба в способі обробки великих даних у режимі реального часу. Обробку великих даних можна виконати кількома різними способами. Найпоширенішими способами є пакетна обробка, мікропакетна обробка та обробка в реальному часі. Існує багато фреймворків, які можна використовувати для виконання операцій з даними, і ці фреймворки створені

з урахуванням кластерних середовищ. Операції, які можна виконувати, включають перетворення, обчислення та аналіз даних.

Також мав бути ефективний спосіб доставки повідомлень у кластерне середовище. Дані, які ще не зберігаються в кластерному середовищі, потрібно спочатку надіслати туди для обробки. Існує багато платформ для доставки повідомлень у кластер, і дві з них описані в даному розділі.

2.1.1. Дистрибутив Hortonworks Data Platform (HDP)

HDP, Hortonworks Data Platform, є дистрибутивом Apache Hadoop з відкритим кодом. HDP включає вбудовані засоби безпеки, управління та операцій. HDP містить широкий спектр інструментів для використання в програмах для великих даних. Він збирає багато фреймів із відкритим кодом, які зазвичай використовуються, в один дистрибутив. Він складається зі способів керування даними, доступу до них, управління даними, безпеки та операцій [24]. Цей розділ має на меті дати короткий вступ до технологій у HDP, які використовувалися.

Управління даними

Частина управління даними HDP містить наріжні камені. Він складається з YARN (Yet Another Resource Negotiator) і HDFS (Hadoop Distributed File System). YARN — це те, що дає змогу виконувати кілька завдань одночасно, воно також забезпечує керування ресурсами та архітектуру, яка дозволяє підключати до нього інструменти доступу до даних. HDFS забезпечує масштабоване, відмовостійке та економічно ефективне зберігання даних [24].

HDFS

HDFS, розподілена файлова система Hadoop, є віртуальною файловою системою, яка використовує архітектуру головний-підлеглий [25, 26]. Файлова система HDFS дуже схожа на інші файлові системи, вона побудована ієрархічно і, як і інші файлові системи, дозволяє переміщувати файли між різними каталогами. Щоб керувати великими обсягами даних на

комп’ютерах, які беруть участь у кластері, потрібна структура. Структура складається з одного NameNode і кількох DataNodes, зазвичай є лише один NameNode і багато DataNodes (зазвичай по одному на учасника кластера). NameNode відповідає за забезпечення керування ієрархією файлової системи та регулювання клієнтського доступу. Клієнти, тобто DataNodes, обслуговують запити на читання та запис від клієнтів файлової системи. Вони також отримують інструкції від NameNode щодо створення, видалення та реплікації блоків даних. Програмне забезпечення NameNode і DataNode написано мовою Java, що дає змогу працювати на будь-якій системі, яка може запускати програми Java.

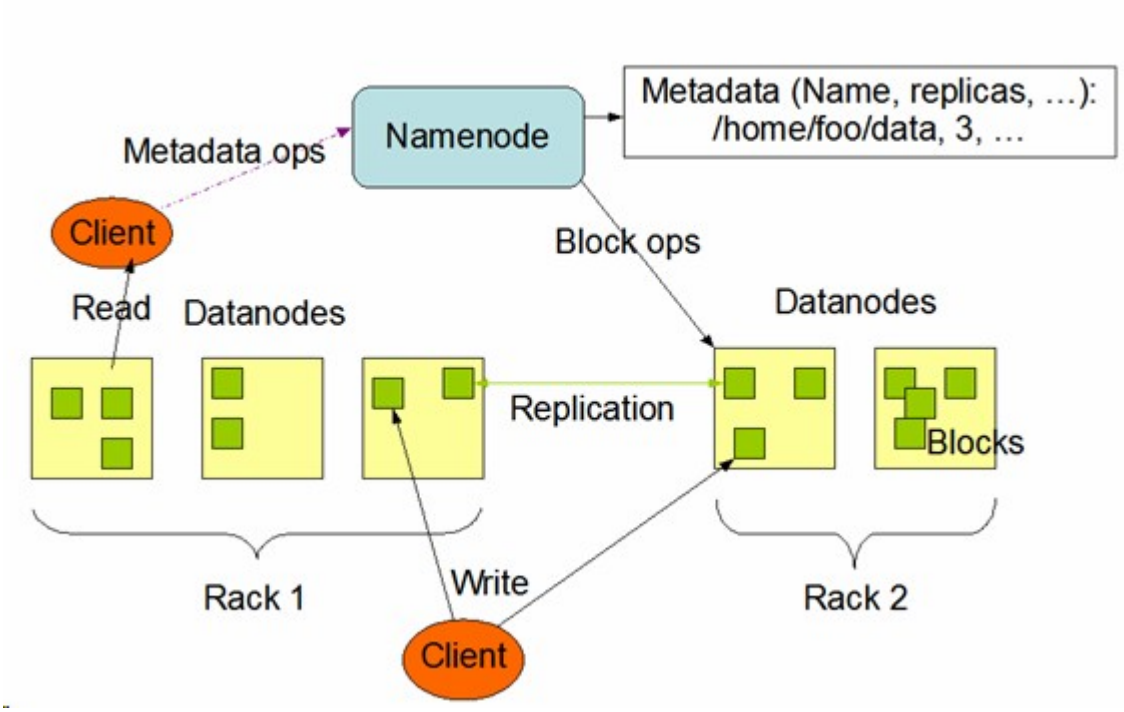

Рис. 2.1. Архітектура Hadoop Distributed File System (HDFS)

На рисунку 2.1 зображено архітектуру Hadoop Distributed File System (HDFS), розподіленої файлової системи, призначеної для зберігання великих обсягів даних на кластерах недорогих комп'ютерів.

Основні компоненти:

- NameNode – головний вузол, який керує файловою системою HDFS. Він зберігає метадані файлів, такі як імена, дозволи, розташування блоків

даних. NameNode відповідає за операції з файлами, такі як створення, видалення, перейменування. Він також відстежує, де зберігаються блоки даних на DataNode.

- DataNode – це вузли, які фактично зберігають блоки даних. Кожен DataNode зберігає частину файлу. DataNode відповідають за обслуговування запитів на читання і запис блоків даних від клієнтів. Вони також періодично надсилають звіти про стан NameNode.

- Client - це програми, які взаємодіють з HDFS для доступу до даних. Клієнти спочатку звертаються до NameNode, щоб отримати метадані файлу, а потім безпосередньо взаємодіють з DataNode для читання або запису даних.

Проблема з HDFS полягає в тому, що вона зберігає метадані про файли окремо, а це означає, що дані, розділені на багато файлів, споживають більше пам’яті як метаданих, ніж ті самі дані в одному файлі.

Обробка великих файлів у HDFS потребує набагато менше ресурсів, ніж обробка тієї самої інформації, розділеної на багато менших файлів, оскільки HDFS NameNodes зберігають свої метадані в пам’яті. Наприклад: NameNode читає файл розміром 1 Гб

100 байт метаданих про цей файл. Наявність 10 000 файлів по 100 КБ, що приблизно дорівнює 1 МБ, споживає 150 МБ пам’яті, більш ніж у 100 разів більше метаданих для того самого обсягу інформації. Ці розрахунки базуються на блозі Cloudera [27] і статті Microsoft [28].

2.1.2. Опис ядром Apache Hadoop YARN

YARN, Yet Another Resource Negotiator, є ядром Apache Hadoop, отже, також ядром HDP [29]. Він підтримує різні механізми обробки даних: інтерактивний SQL, потокове передавання в реальному часі, аналіз даних, пакетна обробка. Ці двигуни здатні обробляти дані, що зберігаються на одній платформі. Його основна увага полягає в тому, щоб забезпечити управління ресурсами та спосіб виконання операцій, які є послідовними, безпечними, а також дозволяють використовувати інструменти керування даними.

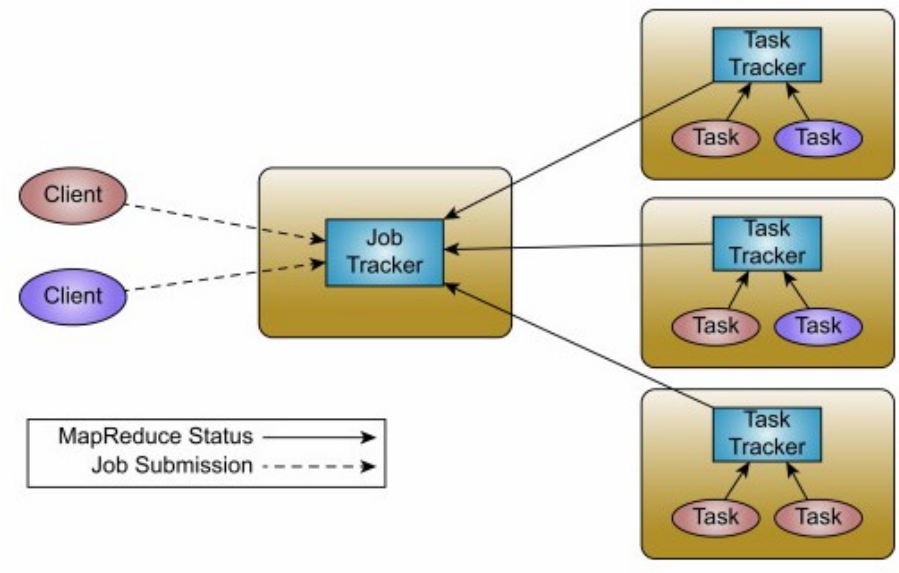

Рис. 2.2. Архітектура YARN

Основна ідея YARN полягає в тому, щоб мати можливість розділити функціональні можливості, пов’язані з керуванням ресурсами та плануванням/моніторингом завдань, на окремі фонові процеси [30]. Це досягається завдяки наявності менеджерів для арбітражу ресурсів, планування, моніторингу та запуску програм у контейнерах.

2.1.3. Архітектура служби синхронізації Zookeeper

Zookeeper — це служба синхронізації для розподілених систем із підтримкою реплікації та кінцевої узгодженості [31]. Він використовується для синхронізації менших обсягів даних між вузлами в кластері, таких як конфігурації та серцеві сигнали. Він складається з окремих вузлів, які згруповані під динамічно вибраним майстром. Дані записуються в зноди, які є масивами байтів, які можуть містити інші зноди, утворюючи структуру дерева, порівнянну з файловою системою.

Записи в Zookeeper плануються через єдиний головний вузол і тому - гарантовано є послідовними. Читання виконуються на окремих підлеглих вузлах, що полегшує багато одночасних читань на одному або кількох вузлах. Однак ці зчитування не гарантують надання найновішої інформації, оскільки головний може мати запис, який ще не було відтворено на підлеглі.

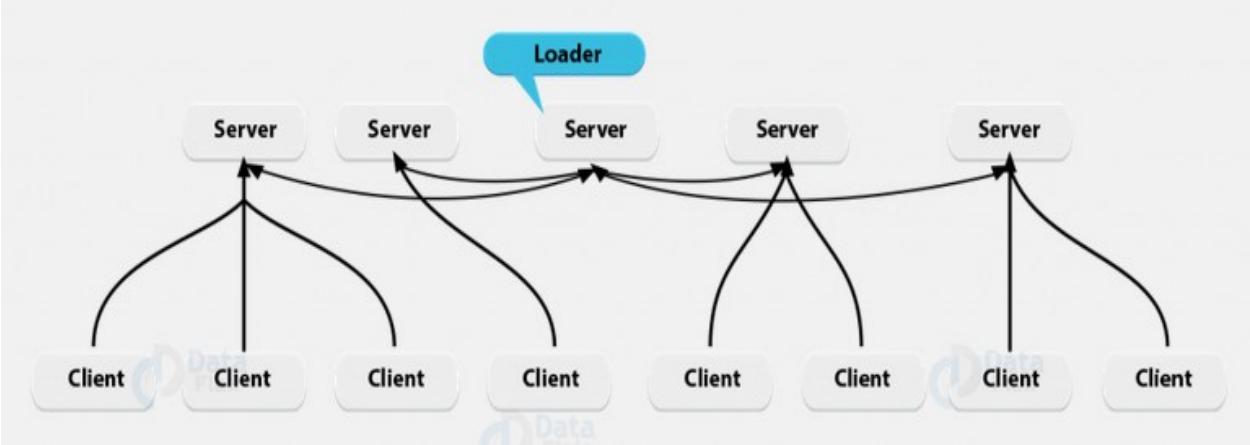

Рис. 2.3. Архітектура Zookeeper

2.2. Способи, методи та фреймворки обробки великих об’ємів даних

HDP містить багато способів обробки даних різними способами [24]. Частини доступу до даних HDP дозволяють обробляти дані пакетно, використовувати інтерактивний SQL або швидкий доступ за допомогою NoSQL. За допомогою фреймворків, таких як Apache Spark, Storm і Kafka, можна обробляти дані різними способами.

2.2.1. Архітектура фреймворку Apache Spark

Apache Spark — це загальний механізм обробки, який сумісний з Hadoop [32]. Він може працювати в кластерах Hadoop через YARN, але також може працювати в автономному режимі. Spark також має перевагу в тому, що він може виконувати пакетну обробку (наприклад, Hadoop MapReduce), обробляти потокові дані (наприклад, NiFi і Storm), а також може полегшити машинне навчання.

Apache Spark - це потужний фреймворк з відкритим кодом для розподіленої обробки великих даних. Він вирізняється високою швидкістю роботи, простотою використання та універсальністю. Spark може працювати як самостійно, так і в поєднанні з Hadoop.

Spark має багаторівневу архітектуру, що складається з наступних компонентів:

1. Driver Program - це головний процес застосунку, який запускає основну функцію Spark та створює SparkContext. SparkContext відповідає за підключення до кластера Spark та координацію виконання завдань.

2. Cluster Manager - це компонент, який відповідає за розподіл ресурсів кластера між застосунками Spark. Spark підтримує різні Cluster Manager, такі як Standalone, Apache Mesos та Hadoop YARN.

3. Executors - це процеси, які виконуються на вузлах кластера та відповідають за виконання завдань. Кожен Executor має певну кількість ядер процесора та пам'яті, які виділяються для виконання завдань.

4. RDD (Resilient Distributed Dataset) - це основна структура даних Spark, яка представляє собою незмінну розподілену колекцію об'єктів. RDD можуть створюватися з різних джерел даних, таких як файли, бази даних та потоки.

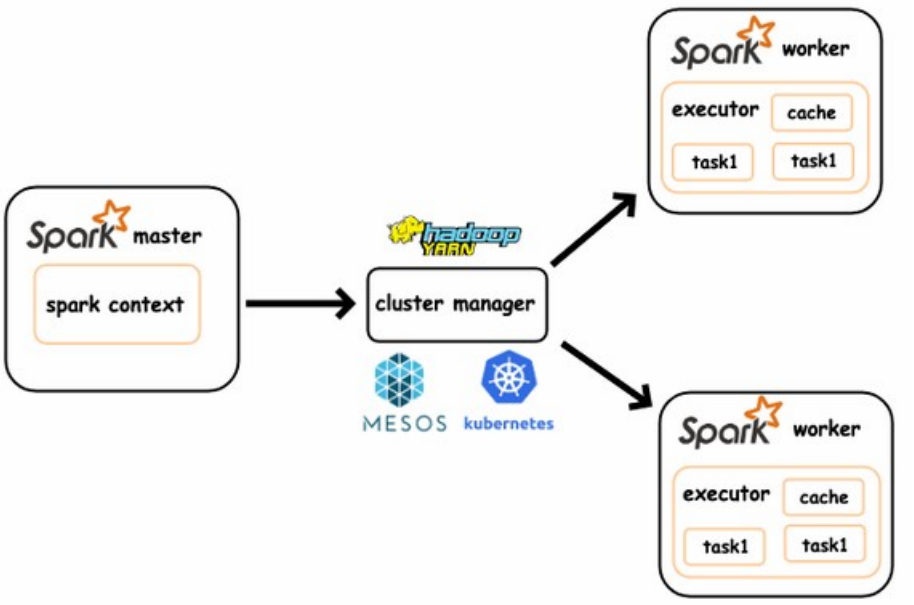

Рис. 2.3. Архітектура Apache Spark

Алгоритм роботи Spark:

- Driver Program створює SparkContext та визначає операції з даними.

- SparkContext надсилає застосунок Cluster Manager для розподілу ресурсів.

- Cluster Manager запускає Executors на вузлах кластера.

- Driver Program розбиває завдання на менші підзадачі та розподіляє їх між Executors.

- Executors виконують підзадачі та повертають результати Driver Program.

2.2.2. Опис та принцип роботи Spark Streaming

Spark Streaming — це частина Apache Spark, яка використовується для обробки даних у реальному часі. Дані, які надходять, можуть бути негайно оброблені шляхом виконання пакету невеликих завдань, що називається мікропакетуванням [33]. Spark Streaming забезпечує масштабовану, високопродуктивну та відмовостійку обробку даних у реальному часі. Ці дані можуть надходити, наприклад, з Kafka або інших сокетів TCP [34].

Spark Streaming - це розширення фреймворку Apache Spark, призначене для обробки потоків даних у режимі реального часу. Воно дозволяє аналізувати та обробляти безперервні потоки даних, такі як дані з сенсорів, соціальних мереж, фінансових транзакцій тощо.

Основні принципи Spark Streaming:

- Мікропакетна обробка: Spark Streaming розбиває потік даних на невеликі пакети (мікропакети) фіксованого розміру за часом. Кожен пакет обробляється як окремий RDD (Resilient Distributed Dataset) в Spark.

- Інтеграція з Spark: Spark Streaming тісно інтегрований з іншими компонентами Spark, такими як Spark SQL, MLlib та GraphX. Це дозволяє легко комбінувати потокову обробку з іншими типами обробки даних.

- Відмовостійкість: Spark Streaming забезпечує відмовостійкість завдяки використанню RDD, які є незмінними та розподіленими структурами даних.

На рисунку 2.4 показано принцип роботи Spark Streaming, компонента Apache Spark, призначеного для обробки потоків даних у режимі реального часу.

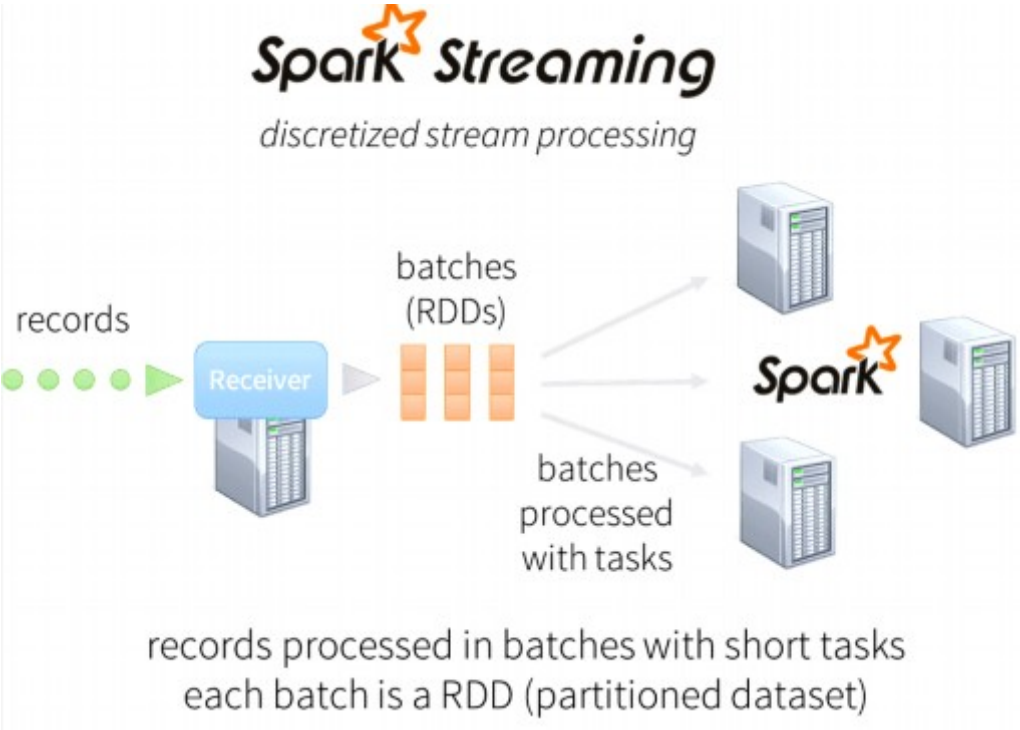

Рис. 2.4. Принцип роботи Spark Streaming

Ось як це працює, згідно з рисунком 2.4:

1. Надходження даних (records): Безперервний потік даних (records) надходить з різних джерел (наприклад, сенсори, веб-сайти, соціальні мережі).

2. Приймач (Receiver): Спеціальний компонент, "приймач" (Receiver), збирає ці дані та буферизує їх.

3. Формування пакетів (batches): Приймач накопичує дані протягом короткого проміжку часу і формує з них пакети (batches). Кожен пакет - це набір даних, зібраних за певний часовий інтервал.

4. RDD (Resilient Distributed Dataset): Кожен пакет даних перетворюється на RDD (Resilient Distributed Dataset) - незмінну, розподілену колекцію об'єктів, яка є основною структурою даних у Spark.

5. Обробка пакетів (batches processed with tasks): Spark обробляє кожен пакет (RDD) окремо, використовуючи розподілені задачі (tasks). Це дозволяє паралельно обробляти дані на кластері, забезпечуючи високу швидкість.

Як результат оброблені дані можуть бути збережені в різних сховищах (бази даних, файлові системи) або використані для інших цілей.

2.2.3. Фреймворк обробки даних в реальному часі Apache Storm

Apache Storm — це фреймворк обробки даних у реальному часі [35], яку також називають потоковою структурою. Запуск Apache Storm на YARN дозволяє проводити аналітику в реальному часі, машинне навчання та моніторинг операцій у реальному часі. Apache Storm може інтегруватися з іншими технологіями в рамках HDP, такими як Kafka [36]. Налаштувавши топологію Apache Storm, можна споживати потоки даних і обробляти їх у режимі реального часу, потоки також можна перерозподіляти між обчислювальними іонами, щоб змінити їх структуру.

Apache Storm - це розподілена система обробки потоків даних у режимі реального часу з відкритим вихідним кодом. Він розроблений для обробки необмежених потоків даних з високою пропускною здатністю та низькою затримкою. Storm ідеально підходить для завдань, де потрібна миттєва реакція на події, таких як аналіз соціальних мереж, виявлення шахрайства, моніторинг мережевого трафіку та обробка даних з сенсорів.

Основні характеристики Apache Storm:

- Простота використання. Storm надає простий API, який дозволяє легко розробляти та розгортати застосунки для обробки потоків даних.

- Масштабованість. Storm може обробляти величезні обсяги даних на кластерах з багатьох вузлів.

- Відмовостійкість. Storm гарантує, що кожне повідомлення буде оброблене принаймні один раз, навіть у разі відмови вузлів.

- Низька затримка: Storm може обробляти дані з дуже низькою затримкою, що робить його ідеальним для застосунків реального часу.

На рисунку 2.5 зображено архітектуру кластера Apache Storm, розподіленої системи обробки потоків даних у реальному часі.

Розглянемо компоненти кластера Storm.

- Nimbus - головний вузол, який відповідає за розподіл коду, призначення завдань робочим вузлам та моніторинг стану кластера. На

рисунку 2.5 показано три вузли Nimbus для забезпечення відмовостійкості. Один з них є лідером (Master Node - Leader).

- Supervisor - робочий вузол, який запускає та керує робочими процесами (workers), що виконують обробку даних. Кожен Supervisor може запускати декілька workers.

- ZooKeeper cluster - розподілена система координації, яка використовується Storm для зберігання інформації про конфігурацію кластера, стану топологій та розташування вузлів. ZooKeeper забезпечує узгодженість даних та допомагає Storm відновлюватися після відмов.

- Storm UI - веб-інтерфейс для моніторингу та керування кластером Storm. Дозволяє переглядати інформацію про топології, Spouts, Bolts, кількість оброблених повідомлень тощо.

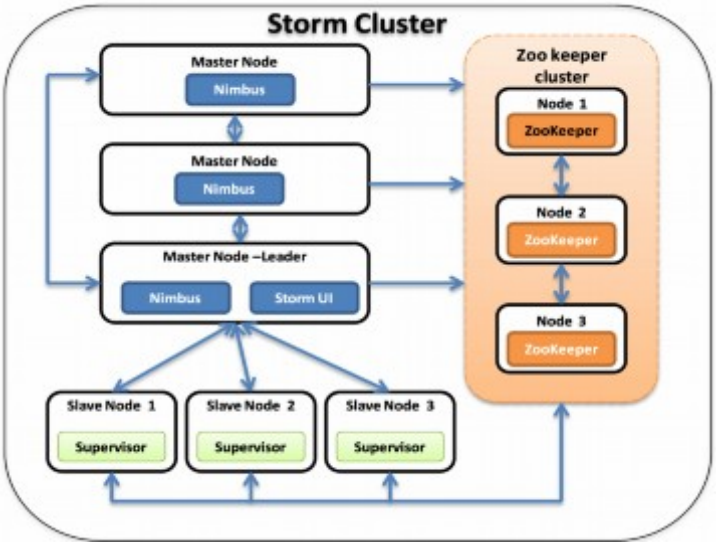

Рис. 2.5. Архітектура кластера Apache Storm

Взаємодія компонентів:

1. Nimbus розподіляє код топології між Supervisor.

2. Supervisor запускають workers, які виконують Spouts та Bolts.

3. Spouts отримують дані з зовнішніх джерел та передають їх Bolts.

4. Bolts обробляють дані та передають їх іншим Bolts або виводять результати.

5. ZooKeeper зберігає інформацію про стан кластера та топологій.

6. Storm UI дозволяє моніторити та керувати кластером.

Загалом, архітектура Storm забезпечує високу масштабованість, відмовостійкість та ефективність обробки потоків даних у реальному часі.

2.2.4. Архітектура фреймворку структурування та обробки потоків даних Apache NiFi

Apache NiFi — це структура, створена для структурування та обробки потоків даних [37]. Він має графічний веб-інтерфейс для налаштування, як називається NiFi, процесорів. Ці процесори можуть виконувати такі операції, як отримання даних, перетворення даних і передача даних. Перевіряючи процесори, можна отримати зворотний зв’язок у реальному часі, оскільки - робота процесорів візуалізується в графічному веб-інтерфейсі користувача. NiFi також дозволяє в режимі реального часу реструктуризувати операції та потоки даних, дозволяючи користувачеві перетягувати нові процесори, переміщувати існуючі або змінювати з’єднання між ними. Apache NiFi - це потужний інструмент з відкритим кодом для автоматизації потоків даних між системами. Він надає візуальний інтерфейс для проектування, моніторингу та керування потоками даних, що робить його легким у використанні навіть для не-програмістів. NiFi особливо корисний для обробки великих обсягів даних, ETL-процесів та інтеграції різних систем.

Основні можливості Apache NiFi:

- Візуальне програмування: NiFi дозволяє створювати потоки даних за допомогою графічного інтерфейсу, перетягуючи компоненти.

- Масштабованість: NiFi може обробляти величезні обсяги даних та масштабуватися для задоволення потреб будь-якої організації.

- Розширюваність: NiFi має модульну архітектуру, яка дозволяє легко додавати нові компоненти та розширювати його функціональність.

- Безпека: NiFi підтримує різні механізми безпеки, такі як SSL/TLS, аутентифікація та авторизація.

- Моніторинг: NiFi надає детальну інформацію про стан потоків даних та дозволяє відстежувати продуктивність системи.

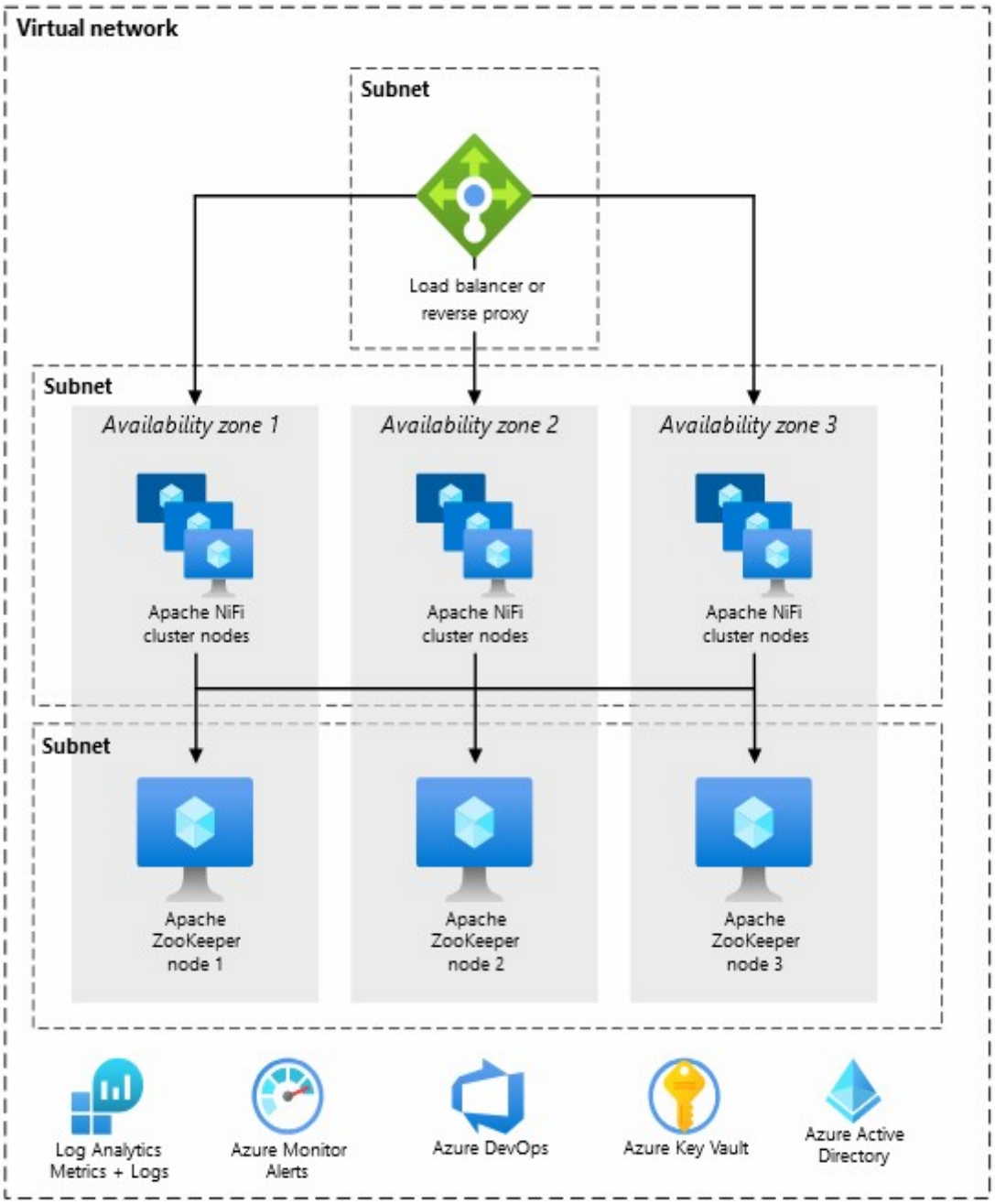

Рис. 2.6. Архітектура Apache NiFi

Архітектура NiFi базується на концепції FlowFile, який представляє собою одиницю даних, що проходить через систему. FlowFile складається з атрибутів (метаданих) та вмісту (даних). NiFi має кілька ключових компонентів:

- Процесори (Processors): Це основні будівельні блоки потоків даних. Процесори виконують різні операції над FlowFiles, такі як читання даних з джерел, перетворення даних, маршрутизація та запис даних у сховища.

- З'єднання (Connections): Це канали, які з'єднують процесори та визначають напрямок руху FlowFiles.

- Контролер потоку (Flow Controller): Це компонент, який керує потоком FlowFiles та розподіляє їх між процесорами.

- Групи процесів (Process Groups): Це логічні групи процесорів та з'єднань, які дозволяють організовувати складні потоки даних.

2.3. Дослідження методів та інструментів передачі великих об’ємів даних

Передача даних стосується способу завантаження та надсилання даних. HDP містить багато інструментів для обробки потоків даних, одним із яких є Apache Kafka. Використання інструментів потоку даних може допомогти контролювати потік даних і порядок операцій.

2.3.1.Розподілена потокова платформа Apache Kafka

Apache Kafka — це система обміну повідомленнями, яка створена як швидка, масштабована та стійка до збоїв [38]. Він використовує модель публікації-підписки, яка дозволяє програмам отримувати повідомлення вибірково. Kafka добре працює в поєднанні з такими фреймворками, як Apache Storm, Apache Spark і Apache NiFi, дозволяючи надсилати дані, які повинні передаватися фреймворками.

Щоб добре виконувати ці завдання, Apache Kafka реалізовано наступним чином [39]. Kafka підтримує різні канали повідомлень, упорядковані за назвами тем. Програми, які створюють вміст для публікації (надсилання) в Apache Kafka, називаються виробниками , тоді як програми, які підписуються (читають) на теми, називаються споживачами. Apache

Kafka зазвичай працює як кластер, що забезпечує ефективну масштабованість, і кожен із цих вузлів кластера, на якому запущено Apache Kafka, називається брокером.

Apache Kafka - це розподілена потокова платформа з відкритим кодом, розроблена LinkedIn. Вона призначена для обробки потоків даних у реальному часі з високою пропускною здатністю та низькою затримкою. Kafka використовується для різних завдань, таких як обмін повідомленнями, відстеження активності веб-сайтів, збір метрик, обробка логів, потокова обробка даних та інтеграція застосунків.

Основні характеристики Apache Kafka:

1. Висока пропускна здатність. Kafka може обробляти величезні обсяги даних з високою пропускною здатністю завдяки своїй розподіленій архітектурі та ефективній обробці даних.

2. Низька затримка. Kafka забезпечує низьку затримку при обробці повідомлень, що робить її ідеальною для застосунків реального часу.

3. Масштабованість. Kafka може легко масштабуватися горизонтально шляхом додавання нових брокерів до кластера.

4. Відмовостійкість. Kafka має високу відмовостійкість завдяки реплікації даних та можливості автоматичного відновлення після відмов.

5. Збереження даних. Kafka зберігає дані на диску, що дозволяє зберігати великі обсяги даних та забезпечує довговічність.

Kafka має розподілену архітектуру, яка складається з наступних компонентів:

- Топіки (Topics) – це категорії або канали, в яких зберігаються повідомлення. Кожен топік розділений на партиції (partitions) для розподілу даних та паралельної обробки.

- Брокери (Brokers) – це сервери, які зберігають дані топіків та обслуговують запити від продюсерів та споживачів.

- Продюсери (Producers) – це застосунки, які надсилають повідомлення до топіків Kafka.

- Споживачі (Consumers) - це застосунки, які отримують повідомлення з топіків Kafka.

- Кластер ZooKeeper - це розподілена система координації, яка використовується для керування кластером Kafka та зберігання метаданих.

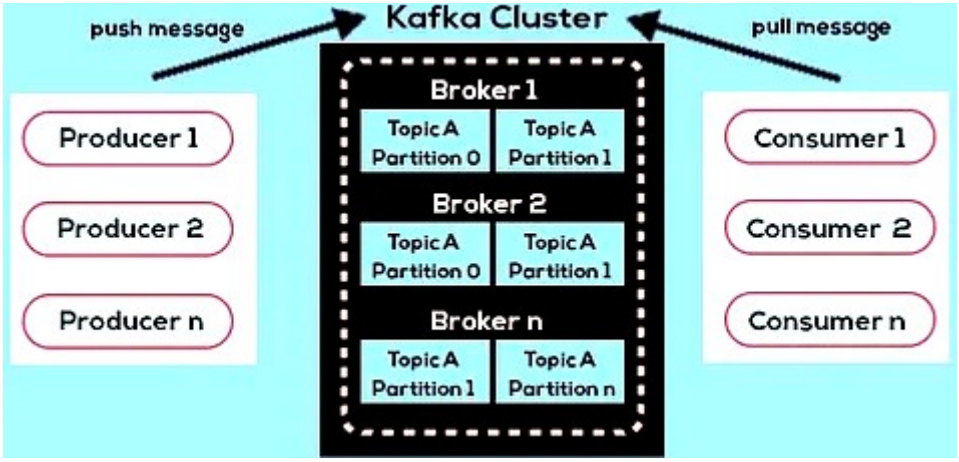

Рис. 2.7. Архітектура Apache Kafka

Алгоритм роботи наступний:

1. Продюсери надсилають повідомлення до топіків Kafka.

2. Брокери отримують повідомлення та зберігають їх у відповідних партиціях.

3. Споживачі підписуються на топіки та отримують повідомлення з партицій.

4. Kafka гарантує порядок повідомлень в межах партиції.

5. ZooKeeper використовується для керування кластером та зберігання метаданих.

2.3.2. Особливості розподіленої системи обміну повідомленнями Apache Flume

Apache Flume — це розподілена система обміну повідомленнями, яка створена як надійна з високою доступністю [40]. Flume була розроблена Apache Software Foundation і є популярним вибором для обробки логів, подій та інших типів потокових даних. Flume збирає, агрегує та переміщує дані, як

правило, дані журналів. Він має гнучку архітектуру, яка базується на потоках даних у реальному часі. Він також підтримує механізми відновлення після збою.

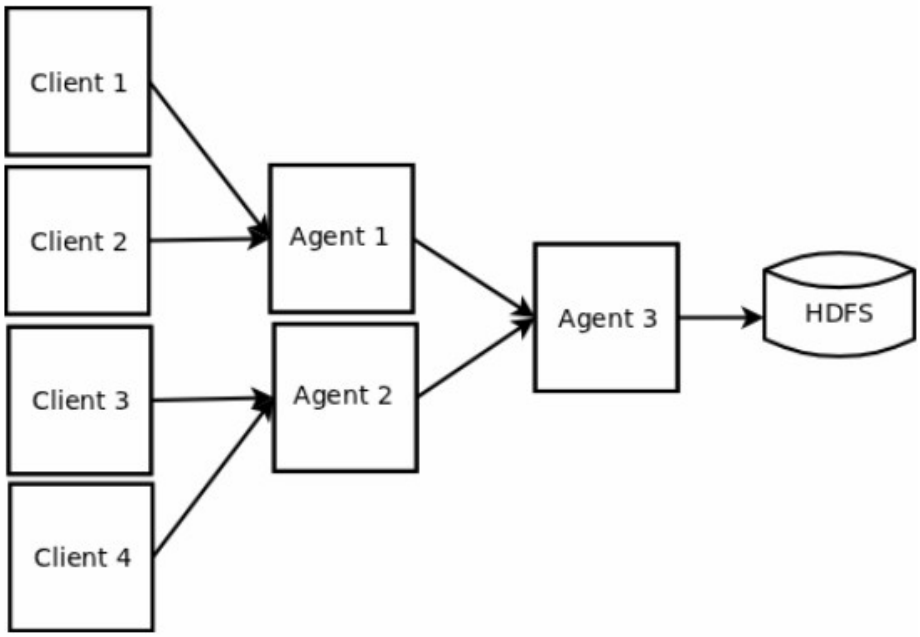

Рис. 2.8. Діаграма суміщених потоків Flume

Flume базується на концепції агентів (agents), які відповідають за збір, агрегацію та переміщення даних. Кожен агент складається з трьох основних компонентів:

- Джерела (Sources) - це компоненти, які отримують дані з зовнішніх джерел, таких як файли, мережеві сокети, API або бази даних.

- Канали (Channels) - це буфери, які тимчасово зберігають дані, отримані від джерел, доки вони не будуть передані до сховищ.

- Сховища (Sinks) - це компоненти, які записують дані в цільові сховища, такі як HDFS, HBase, Hive, Kafka або бази даних.

У Flume потік запускає клієнт. Потім клієнт передає певний тип повідомлення джерелу, тобто джерело, звідки дані надходять до агента Flume (машина, на якій працює Flume) [41]. Потім джерело передає повідомлення на один або кілька каналів, канал – це те, де дані переміщуються між джерелом і приймачем. Сховище — це компонент Flume, який осушує канали

та пропускає його вперед. У каналах використовується модель виробник-споживач, тобто якщо Flume перевантажується вхідними даними, канал зростає та може пристосуватися до сплеску даних. Приймачі можуть бути об’єднані разом з іншими джерелами, щоб забезпечити більш складні потоки даних.

Висновки до розділу

У другому розділі детально досліджено сучасні методи, архітектури та інструменти, які використовуються для обробки великих даних. Початково було проведено огляд технологій, що дозволяють ефективно працювати з великими обсягами інформації. Однією з ключових платформ визнано Hortonworks Data Platform (HDP), яка інтегрується з Apache Hadoop і пропонує широкий спектр інструментів для аналітики. Ядро Apache Hadoop YARN показало свою здатність оптимально розподіляти обчислювальні ресурси, а Zookeeper забезпечує стабільну координацію системи, критичну для її надійності.

Особливу увагу приділено фреймворкам, які використовуються для обробки даних. Apache Spark виділяється своєю потужністю, дозволяючи обробляти дані в пам’яті, а його модуль Spark Streaming ідеально підходить для роботи з потоковими даними. Для задач реального часу також було розглянуто Apache Storm, відомий своїми низькими затримками і високою продуктивністю. Apache NiFi, зі своєю здатністю візуалізувати та структурувати потоки даних, демонструє зручність у керуванні складними інформаційними потоками.

Методи передачі даних також отримали значну увагу. Apache Kafka, завдяки своїй масштабованості і надійності, стала ключовим інструментом для потокової передачі даних у великих системах. Apache Flume, у свою чергу, є ефективним рішенням для збору та переміщення даних з різноманітних джерел до централізованих сховищ.

Таким чином, у розділі представлено комплексний аналіз рішень, що дозволяють ефективно управляти обробкою та передачею великих даних. Це дослідження підкреслює актуальність використання цих технологій для розв’язання сучасних завдань, таких як бізнес-аналітика, прогнозування та створення адаптивних додатків, які працюють у режимі реального часу.

РОЗДІЛ 3. ЗАСТОСУВАННЯ МЕТОДІВ ТА ЗАСОБІВ ФРЕЙМВОРКІВ ОБРОБКИ ВЕЛИКИХ ДАНИХ

3.1. Вибір інструментів для дослідження

У цьому розділі буде пояснено інформацію про те, які інструменти було обрано та чому. Він починається з порівняння технологій, описаних у попередньому розділі, і продовжується розділом про те, які фреймворки були обрані для використання під час тестування.

Цей розділ має на меті пояснити основні відмінності між фреймворками стосовно нашого тестового випадку. Порівнювалися не всі аспекти, а лише ті, які, швидше за все, були точками уповільнення або прискорення.

3.1.1. Порівняння фреймворків

Spark Streaming базується на концепції під назвою мікропакетування. Це означає, що Spark Streaming може безперервно отримувати дані з джерела даних, а потім ділить ці дані на пакети, які надсилаються на механізм Spark як невеликі пакети [34]. Це викликає затримку, оскільки дані очікуються, перш ніж їх буде надіслано для обробки [42]. Spark Streaming також розроблено для підтримки одноразової обробки, це означає, що дані будуть оброблені лише один раз [43].

Storm виконує почергову обробку, як тільки він отримує деякі дані, він може почати їх обробку [44]. Якщо не групувати дані, як це робить Spark Streaming, можна досягти меншої затримки. Час, який Spark Streaming витрачає на буферизацію вхідних даних, видаляється. Storm, однак, не підтримує семантику точного разу, і тому не може гарантувати, що дані оброблятимуться лише один раз [45]. Однак він підтримує обробку щонайбільше та щонайменше один раз, ці режими не можна запускати одночасно. Однак ці два режими чутливі до різних помилок. Обробка

щонайменше один раз може призвести до втрати даних, тоді як обробка щонайменше один раз може призвести до дублювання даних [46].

NiFi працює інакше, ніж Spark Streaming і Storm, це в основному інструмент, який використовується для керування потоками даних. Таким чином, NiFi за своєю суттю не забезпечує механізмів для семантики щонайбільше один раз, щонайменше один раз і точно один раз. Тому вони повинні бути реалізовані, якщо вони потрібні, розробниками та менеджерами потоку даних NiFi.

Flume тісніше пов'язаний з екосистемою Hadoop, ніж Kafka, він підтримує приймачі, які підтримуються з коробки для таких завдань, як запис у HDFS [47]. Щоб додати споживачів до Flume, необхідно змінити конвеєр і відтворити канал, що призведе до простою. Стійкість повідомлень Flume базується на каналах на основі файлів і каналів на основі пам'яті; це має зворотну сторону: повідомлення недоступні, доки не буде відновлено агент, який завершив роботу.

Kafka — це система обміну повідомленнями більш загального призначення, яка не створена спеціально для Hadoop, тобто Kafka легше переходити з однієї платформи на іншу. Однією з переваг Kafka є те, що він дуже масштабований, дозволяє додавати споживачів без простоїв і без впливу на продуктивність. Це пов'язано з тим, що Kafka просто зберігає всі повідомлення, доки не закінчиться термін їх дії, закінчення терміну дії здійснюється встановленням певного часу, протягом якого повідомлення зберігаються до їх видалення з Kafka.

3.1.2. Вибір та обґрунтування фреймворків

Вибір фреймворків для тестування базувався на кількох різних факторах. По-перше, NiFi суттєво відрізняється від Storm і Spark Streaming, як згадувалося в попередньому розділі. Не вдалося знайти відповідні тести NiFi, адже це не система, призначена для виконання тих самих завдань, які можуть виконувати Spark Stream ing і Storm. NiFi також масштабується

іншим способом: кластер вузлів NiFi працює, запускаючи весь потік даних на кожному вузлі, тоді як Spark Streaming і Storm масштабуються, розподіляючи підзавдання на інші вузли. Досліджуючи структуру , яка не була створена спеціально для аналізу великих даних, було корисно порівняти її продуктивність із системою, створеною для обробки великих даних у режимі реального часу. Порівняння фреймворків, створених для різних цілей, мало цінність, оскільки давало можливість побачити, чи потрібні фреймворки для великих даних у типі тесту, який проводився.

Через обмежений час, було неможливо реалізувати багато готових функцій, які підтримує Spark Streaming.

Оскільки NiFi є інструментом, який дуже відрізняється від Spark Streaming і Storm, більшість функцій, які надають Spark Streaming і Storm, потрібно було реалізувати, щоб дати більш справедливе порівняння. Оскільки Spark Streaming надає такі функції, як семантика точного разу та мікропакетування, їх потрібно було б реалізувати в NiFi. Storm, з іншого боку, використовує почергову обробку, тобто процесори NiFi працюють за замовчуванням. Storm також не підтримує точно один раз, як і NiFi.

Через цю різницю в функціях вибрані фреймворки для тестування один з одним були NiFi і Storm.

Природа систем передачі повідомлень призвела до вибору Apache Kafka як системи, що використовується для передачі даних. Kafka є більш довговічною та масштабованою, якщо вхідні дані збільшуються, у Kafka легко пристосуватися до цих сплесків, оскільки вона не потребує простою для масштабування. Він також підтримує довговічність у кращий спосіб, оскільки завжди зберігає записи отриманих повідомлень.

3.2 Моделі та налаштування фреймворків обробки великих даних

Порівняно з другим розділом цей розділ має на меті надати детальніший опис роботи обраних фреймворків. Він також містить

детальнішу інформацію про термінологію, що використовується різними фреймворками.

3.2.1. Особливості Apache Kafka

Як зазначено в другому розділі, Kafka - це система обміну повідомленнями типу "видавник/підписник". У цій дисертації не були потрібні всі функції Kafka, але цей підрозділ має на меті представити різні функції Kafka, які використовувалися в тестах.

Коли повідомлення надсилаються до Kafka, необхідно вказати топік. Топік - це спосіб для Kafka визначити, куди повідомлення мають бути надіслані далі. Коли відправник надсилає повідомлення до Kafka, він називається продюсером. Коли продюсер надсилає повідомлення до Kafka, вони зберігаються протягом налаштованого часу. Вони залишатимуться в Kafka доти, доки хтось не підпишеться на цей топік і не отримає повідомлення, які були надіслані до нього. Той, хто отримує повідомлення, називається споживачем. Топік має можливість використовувати розділи. Розділ - це підмножина топіка. Розділення топіка на чотири розділи, наприклад, дозволило б чотирьом різним споживачам отримувати повідомлення з одного і того ж топіка залежно від номера розділу. Ця техніка використовується в тестах, щоб мати змогу рівномірно розподіляти повідомлення з Kafka.

3.2.2. Налаштування Apache NiFi

Apache NiFi - це система керування потоками даних з відкритим кодом, де потоки даних створюються та керуються через веб-інтерфейс. Для наших цілей не були потрібні всі функції NiFi, але основні з тих, що використовувалися, описані в цьому підрозділі.

NiFi містить безліч готових процесорів. Процесор у NiFi - це компонент, який певним чином взаємодіє з даними, що надсилаються через потік даних. Деякі з включених процесорів були тими, що були потрібні для

тестів. Процесори, що постачаються з NiFi, є досить налаштовуваними, і було можливо досягти мети потоку даних, включивши деякі з готових компонентів з тими, що були написані з урахуванням конкретного завдання. Процесори, які не доступні відразу після нової інсталяції NiFi, можна додати. Це було необхідно зробити, щоб мати змогу писати власні процесори для потоку даних. Потік даних створюється шляхом перетягування компонентів у веб-інтерфейс, а потім з'єднання їх разом на основі різних зв'язків. Приклад того, як виглядає веб-інтерфейс NiFi, можна побачити на рисунку 3.1.

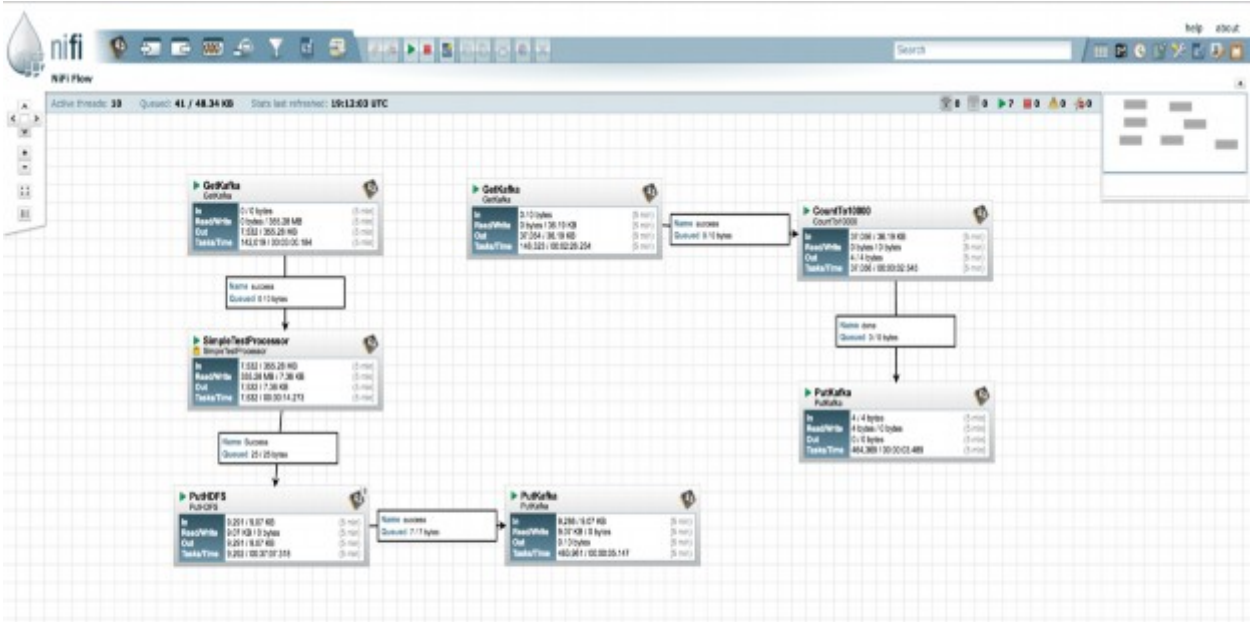

Рис. 3.1. Приклад налаштування в графічному веб-інтерфейсі NiFi

На рисунку 3.1 більші квадрати представляють процесори NiFi, маленькі квадрати представляють черги між ними, а стрілки – напрямки.

3.2.3. Особливості та модель роботи платформи Apache Storm

Apache Storm - це фреймворк з відкритим кодом для обробки великих даних у режимі реального часу. Він зазвичай використовується там, де дані потрібно витягти та обробити, наприклад, виконати певний тип підрахунку або обчислити значення на основі вхідних даних. Багато функцій Storm не мали значення для процесу тестування, і вони не будуть розглянуті.

Storm використовує абстракції, які називаються Spouts та Bolts, для передачі даних в межах топології. Топологія - це назва структури ланцюга обробки [48]. Spout - це компонент, який вносить дані в топологію ззовні кластера [48]. Наприклад, компонент з назвою KafkaSpout - це Spout, який вносить дані з Kafka в топологію. Після того, як дані потрапили до spout, а отже, і в топологію, він може розподіляти дані до Bolts. Bolt - це компонент, який використовується для обробки даних, отриманих від Spout або від іншого Bolt [48]. Bolt може, наприклад, як HDFSBolt, пояснений пізніше, отримати деякі дані, обробити їх та передати ці дані іншому Bolt. Bolts є універсальними і не завжди повинні щось випромінювати, вони можуть бути кінцевими точками, де дані просто обробляються, а потім, по суті, видаляються з топології. Існує багато Spouts та Bolts з відкритим кодом, два з яких були включені в процес тестування.

Налаштування, що використовувалося для тесту, включало один екземпляр Nimbus, один екземпляр Storm UI Server та до чотирьох екземплярів Supervisors. Nimbus - це фреймворк, який можна використовувати разом зі Storm, це фреймворк, який відповідає за призначення завдань, моніторинг кластера на предмет збоїв та розподіл коду (раніше згаданої топології) по вузлах кластера [49]. Storm UI Server - це веб-сервер, який обслуговує веб-інтерфейс, де можна моніторити топологію майже в реальному часі, тут можна побачити такі параметри, як затримка Spout та Bolt, а також підказки щодо того, яким Spouts або Bolts слід надати вищий ступінь паралелізму. Приклад частини Storm UI можна побачити на рисунку 3.2.

Рис. 3.2. Приклад частини того, як виглядає Storm UI.

Кожен вузол, який виконує роботу для Storm, запускає те, що називається Supervisor. Supervisor відповідає за прослуховування Nimbus та запуск/зупинку робочих процесів за потребою [49].

Топологія Storm покладається на розділення своєї роботи на робочі процеси. Робочий процес - це одиниця виконання, яка виконує підмножину топології, вона належить до певної топології та може запускати один або кілька виконавців для Spouts або Bolts. Виконавець - це потік, який породжується робочим процесом, кожен виконавець завжди має один потік і може запускати кілька завдань. Завдання - це назва одиниці, яка виконує фактичну роботу та запускається в потоці свого виконавця. За замовчуванням виконавець запускає лише одне завдання [50]. Ілюстрацію моделі виконання Storm можна побачити на рисунку 3.3.

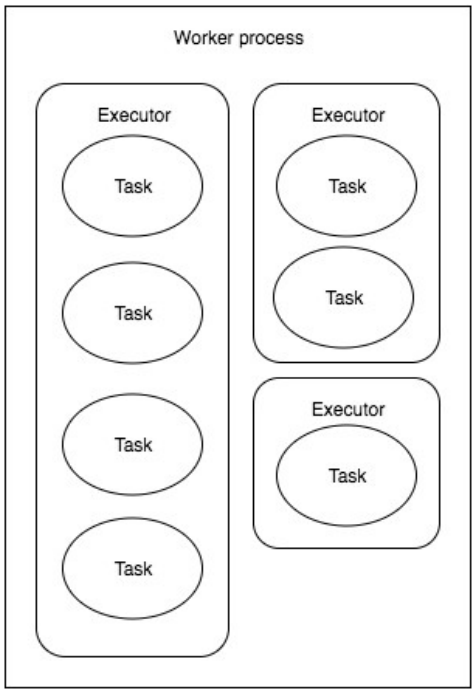

Рис. 3.3. Ілюстрація моделі виконання Storm

3.3. Вибір та опис формату даних

Для кращого аналізу проблеми цієї роботи та тестування платформи розподіленої обробки в реалістичних сценаріях було обрано eXtensible

Business Reporting Language (XBRL) як відповідний формат даних для обробки. Характер бізнес-звітності означає, що корпорація надсилає свої звіти раз на рік, і кількість звітів для обробки може бути обмежена географічним розташуванням. Існують інші формати, які можна обробляти таким же чином, це рішення зосереджено на XML, але йому байдуже, який саме формат XML. XBRL був запропонований Granditude.

3.3.1 Формат даних XBRL

eXtensible Business Reporting Language, або XBRL, який аналізувався, використовується як носій інформації для фінансів. Це, по суті, розмічена бізнес-інформація, яка використовується для створення фінансової звітності, бізнес-документів та бухгалтерських документів.

Звіт XBRL базується на двох документах: документі екземпляра XBRL, який є фактично розміченою інформацією, та одній або кількох таксономіях. Таксономії - це описові документи, які містять мітки, що використовуються в документі екземпляра, а також метадані, такі як правила форматування та зв'язки між мітками. Документ екземпляра містить факти, які є значеннями в поєднанні з контекстом. Контексти надають значенням сенс: значення 1 000 не має великого значення, якщо ви не знаєте, що це кількість товарів, проданих у квітні 2015 року, наприклад. Контекст зазвичай включає час, що описує значення, та організацію-власника. Таким чином, контекст у звичайному тексті може бути "долари, зароблені корпорацією приклад станом на 01.01.2016".

XBRL Sweden, станом на дату написання цієї роботиї, має три офіційні таксономії. Вони відповідають різним законам про бухгалтерський облік. Відображення концепцій таксономії є повним, але його слід використовувати лише як довідник, а не в офіційних бухгалтерських документах.

Остання версія XBRL на момент написання - 2.1, яка була випущена в грудні 2003 року з останньою зміною 2013. Дані в цьому форматі можна отримати від європейських корпорацій з Європейського бізнес-реєстру, EBR.

3.4. Представлення методології тестування

Цей розділ містить вступ до того, що тестувалося та як це тестувалося. Він також пояснює налаштування, що використовувалися під час тестування. У цьому розділі також буде інформація про те, як було налаштовано кластер і як різні компоненти були створені для спільної роботи. Ілюстрації налаштування кластера також будуть знайдені як для NiFi, так і для Storm.

3.4.1. Тестовий випадок

Тестовий випадок, який розглядався, був гіпотетичним, припущенням, що всі компанії в Європі мали надсилати свою фінансову звітність до центральної організації. В Європі налічується 25,6 мільйона підприємств. Якби всі ці підприємства надсилали свою фінансову звітність раз на рік, це призвело б до обробки в середньому приблизно 70 000 документів XBRL на день. Тести були проведені, щоб побачити, як швидко можна підрахувати певний тег, у цьому випадку "se-gen-base:Soliditet", у 10 000 документах у режимі реального часу з приблизним розміром файлу 50 КБ. Тест був комплексним, від моменту надсилання повідомлень до часу, який знадобився відправнику, щоб отримати підтвердження про те, що всі повідомлення були збережені в HDFS. 10 000 було обрано замість 70 000, щоб тести могли виконуватися протягом розумного часу, але все ж давали гарне уявлення про те, скільки часу знадобиться для 70 000 повідомлень.

3.4.2 Специфікація кластера

Для проведення тестів використовувався кластер, розгорнутий на AWS, Amazon Web Services. На всіх машинах у кластері було запущено HDP 2.4.

Було доступно чотири машини, і машини були типу r3.large. Amazon вважає їх машинами з оптимізованою пам'яттю. Вони працювали на двох процесорах типу Intel Xeon E5-2670 v2 (Ivy Bridge), 15,25 ГБ оперативної пам'яті та 50 ГБ жорстких дисків.

3.4.3. Бенчмаркінг

У цьому розділі обговорюються різні методи тестування фреймворків. Буде інформація про різні способи проведення тестів, а також інформація про те, чому був обраний певний метод.

Вибір методу

Багато різних методів було розглянуто, перш ніж прийняти рішення про конкретну методологію. Через різну природу фреймворків довелося створити узагальнений підхід, який можна було б легко та надійно протестувати на подібній основі. У цьому розділі наведено різні методи, які розглядалися, та причини, чому був обраний певний метод.

Один з методів, який розглядався, полягав у вимірюванні часу кожного окремого компонента, щоб побачити, де присутні потенційні вузькі місця. По-перше, це дозволяє легко ідентифікувати вузькі місця, по-друге, це вимагає, щоб кожен компонент записував індивідуальний час. Це вимагало б більш складного процесу для реалізації. Через обмеження в часі та той факт, що весь потік даних мав бути протестований, цей метод не був обраний.

Інший метод полягав би в тому, щоб не враховувати час, який знадобився для надсилання повідомлень, це дозволило б тестам включати лише час, який повідомлення провели на кластері, а не час, який знадобився для надсилання повідомлень до кластера. Цей метод був перспективним, але в кінцевому підсумку не був обраний через те, що, знову ж таки, метою було знайти час, який знадобився для надсилання повідомлень до кластера та отримання відповіді про те, що вся обробка завершена.

Остаточний, і обраний, метод полягав у тому, щоб надіслати всі повідомлення до кластера, а потім сидіти та чекати відповіді, яка означає, що процес завершено. Цей метод має перевагу в тому, що фактично можна побачити, скільки часу знадобиться одному відправнику, щоб надіслати певну кількість повідомлень, включаючи час, який знадобився для відповіді відправнику, що все було оброблено успішно. Деякі недоліки цього підходу включають той факт, що мережева затримка та інший час передачі включені

в тести. Також було зроблено висновок, що це буде найшвидший підхід до реалізації, який буде працювати подібно в обох фреймворках.

Додаткова примітка щодо тестування

Щоб уникнути впливу на тести шляхом оптимізації конфігурації використовуваних інструментів, всі інструменти тестувалися з їхніми стандартними налаштуваннями або якомога ближче до цих налаштувань. Не налаштовуючи конфігурації, зрозуміло, що оптимальної продуктивності може бути не досягнуто, але фреймворки будуть протестовані на рівних умовах. Важливо також зазначити, що фреймворки призначені для досягнення успіху в різних речах, тому залишення стандартних конфігурацій може вплинути на результати тестів продуктивності.

3.4.4. Налаштування конфігурації компонентів

Цей розділ має на меті надати огляд налаштувань, що використовуються в різних компонентах, він не включатиме налаштування, які є стандартними для HDP, але включатиме налаштування, що використовуються виробником повідомлень, та налаштування, які були змінені на HDP під час виконання тестів.

Відправник

Відправник був програмою Java, що складається з виробника Kafka та споживача Kafka, метою якої було надсилання повідомлень та очікування відповіді, яка вказує на те, що всі файли були успішно збережені в HDFS. Після отримання цієї остаточної відповіді можна було бути впевненим, що всі повідомлення пройшли обробку без помилок, тобто можна було обчислити час від початку передачі до отримання останнього повідомлення. Щоб видалити якомога більше накладних витрат від виробника, один і той же файл надсилався знову і знову. Надсилання різних файлів означало б, що виробник також постійно мав би читати новий файл, перш ніж надсилати його до кластера. Другою причиною надсилання одного і того ж файлу знову і знову є те, що XBRL не гарантує, що кожен тег XML присутній у кожному

документі XBRL. Оскільки завданням обробки в режимі реального часу було підрахувати кількість входжень певного тега в певному документі, було прийнято рішення надсилати один і той же файл знову і знову. Цей компонент використовувався як для тестування Storm, так і для NiFi. Відправник мав вказати налаштування Kafka, щоб мати змогу спілкуватися з рештою кластера. Один набір налаштувань для частини виробника Kafka та інший для частини споживача Kafka. Усі ці налаштування були залишені за замовчуванням, крім налаштувань, специфічних для кластера, таких як IP-адреси, порти, ідентифікатори груп Kafka тощо.

XBRLParser

Було створено клас Java, який можна використовувати як у Storm, так і в NiFi. Назва цього класу була XBRLParser, і його метою було розібрати рядок у XML-документ і зберегти його в об'єкті. Об'єкт надає метод, який підраховує теги залежно від вхідного рядка, який він отримує. Внутрішньо клас XBRLParser використовував існуючі класи Java, створені для розбору XML. Цей компонент, як і відправник, використовувався під час тестування обох фреймворків.

Компоненти, що використовуються у фреймворках

Цей підрозділ описує компоненти, що використовуються для виконання тестів. Було потрібно кілька різних компонентів, але вони досить схожі один на одного, тому вони описуються лише один раз, з зазначенням відмінностей.

Повідомлення, що містять XML, мали бути отримані кластером певним чином, і оскільки Kafka був обраним фреймворком для цього, використовувалися компоненти, сумісні з Kafka. Повідомлення отримувалися за допомогою готових рішень з відкритим кодом як для NiFi, так і для Storm, які називаються GetKafka та KafkaSpout відповідно.

Тест включав підрахунок певної кількості тегів у XML-документах, які надсилалися до кластера. Це було зроблено за допомогою спеціально створених рішень для обох фреймворків. Він отримує XML від компонента,

згаданого в попередньому абзаці, підраховує кількість певного тега та надсилає результат далі. Назви, що використовуються для цього компонента, були TagCounter та TagCounterBolt для NiFi та Storm відповідно.

Для збереження даних у HDFS вже існували рішення з відкритим кодом, які використовувалися в обох випадках. Цей компонент отримує результат від попередньо згаданого компонента та зберігає дані в HDFS, після чого передає його наступному компоненту. Різниця між рішеннями з відкритим кодом NiFi та Storm полягає в тому, що рішення Storm за замовчуванням не передає повідомлення далі. Це було зроблено шляхом модифікації коду з відкритим кодом. PutHDFS - це назва реалізації NiFi, а HDFSBolt - назва реалізації Storm.

Далі потрібно було підрахувати кількість повідомлень, які були успішно збережені. Це було зроблено за допомогою спеціально створених рішень, які отримували повідомлення від компонента в абзаці вище, щоб підрахувати кількість успішних збережень у HDFS. Це було названо Counter та CounterBolt для NiFi та Storm відповідно.

Нарешті, був потрібен компонент для доставки повідомлення назад до клієнта. Знову ж таки, існували готові рішення з відкритим кодом, які використовувалися в обох фреймворках. Ці рішення отримують повідомлення, коли попередньо описаний компонент досягає цільового значення, і надсилають повідомлення до Kafka, що означає завершення процесу. Цей компонент називався PutKafka у випадку NiFi та KafkaBolt у випадку Storm.

3.4.5 Налаштування кластера

Довелося зробити два різних налаштування, оскільки виконувалися тести як для Storm, так і для NiFi. Налаштування складалися з набору компонентів, які були пояснені в попередньому розділі. Цей розділ присвячений поясненню та візуалізації того, як потік даних був організований у двох різних фреймворках.

Відправник

Відправник був програмою Java, яка надсилала дані та чекала, поки отримає відповідь, що дані були оброблені та успішно збережені. Відправник також відповідав за вимірювання часу процесу від першого надісланого повідомлення до отримання повідомлення, яке підтверджує, що всі дані були оброблені правильно. Програма Java надсилала та отримувала повідомлення через Kafka.

Відправник надсилав дані до певного топіка Kafka, який був налаштований спеціально для даного тестового випадку. Оскільки тести мали виконуватися на різній кількості вузлів кластера, необхідно було створити певні топіки з різною кількістю розділів. Усі топіки були створені з коефіцієнтом реплікації один. Ці топіки потім споживалися двома різними налаштуваннями, описаними далі.

Важливо зазначити, що вибір методу (необхідність чекати відповіді, яка означає, що все завершено) накладає обмеження: тести ніколи не завершаться швидше, ніж найповільніший Bolt або Processor у відповідних фреймворках, всі вони повинні чекати найповільнішого.

На всіх вузлах було запущено не тільки необхідне програмне забезпечення. На вузлах було запущено інше програмне забезпечення, таке як Zookeeper, YARN, HDFS та Kafka, а також NiFi або Storm. Див. Додаток А для переліку програмного забезпечення, яке було запущено на кожному вузлі відповідно. Зверніть увагу, що під час запуску Storm та NiFi фреймворк, який наразі не тестувався, був вимкнений, обидва вони з'являться в Додатку А, але не запускалися одночасно один з одним.

Налаштування Storm

Налаштування Storm складалося з одного Spout та чотирьох Bolts. KafkaSpout відповідав за підписку на топік Kafka, до якого відправник надсилав повідомлення, а потім передавав їх далі до наступного кроку в ланцюжку обробки, а саме до TagCounterBolt. TagCounterBolt потім отримував повідомлення, що містить дані XBRL, підраховував кількість тегів

"se-gen-base:Soliditet" у документі та передавав результат наступному болту, HDFSBolt. HDFSBolt зберігає повідомлення та передає його до CounterBolt після того, як воно було збережено. CounterBolt потім продовжує підраховувати кількість повідомлень, які були збережені в HDFS, і коли воно досягає попередньо налаштованого значення (кількість повідомлень, використаних у тесті), воно передає повідомлення до KafkaBolt. Після того, як KafkaBolt отримує повідомлення, він передає повідомлення до топіка Kafka, на який підписаний відправник Java. Масштабування налаштування здійснюється шляхом додавання більшої кількості екземплярів Bolts та Spout. Використовувався один KafkaSpout на кожен розділ Kafka. Зверніть увагу, що спосіб розподілу компонентів Storm здійснюється автоматично, тому ілюстрація налаштування є логічною схемою. Це налаштування проілюстровано на рисунку 3.4.

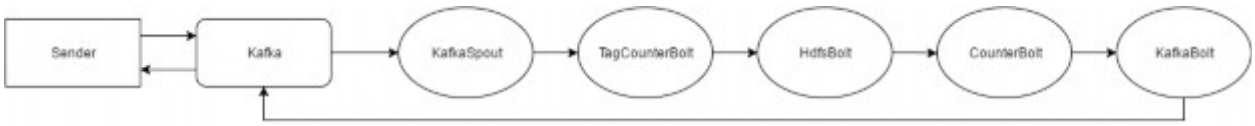

Рис. 3.4. Ілюстрація логічної схеми, що використовується для Storm

Налаштування NiFi

NiFi було налаштовано подібно до Storm. На всіх вузлах запускалися ті самі компоненти, а на одному вузлі також запускалися додаткові компоненти (див. Вузол та Додатково на рисунку 3.5 відповідно). Повідомлення надходили на вузли через Kafka, їх отримував процесор GetKafka, він підписувався на певний топік, і коли повідомлення були отримані, він передавав їх до процесора TagCounter. TagCounter отримував повідомлення, що містить дані XBRL, і підраховував кількість тегів "se-gen-base:Soliditet" у документі.

Після завершення підрахунку він передавав результат до процесора PutHDFS. Коли результат був отриманий у процесорі PutHDFS, він зберігався в HDFS, а потім збережені дані передавалися до процесора Counter. Процесор

Counter відповідав за відстеження того, скільки файлів було успішно збережено в HDFS на поточному вузлі, і коли він досягав кількості повідомлень, включених у тест, він надсилав повідомлення, яке означає, що всі файли були успішно збережені в HDFS. Це повідомлення надсилалося до процесора PutKafka, і коли процесор PutKafka отримував це повідомлення, він надсилав його до топіка Kafka, на який був підписаний інший процесор GetKafka.

Після того, як цей процесор GetKafka отримував повідомлення, він передавав їх до іншого процесора Counter, який підраховував кількість вузлів, які були завершені. Після того, як всі вузли були завершені, він надсилав повідомлення до іншого процесора PutKafka. Цей процесор PutKafka, у свою чергу, надсилав повідомлення до топіка Kafka, на який була підписана програма Java. Як пояснено в цьому тексті та проілюстровано на рисунку 3.5, це налаштування дещо відрізняється. Через природу того, як NiFi зазвичай кластеризується (один і той же потік даних зазвичай розміщується на кожному вузлі), необхідно було додати додаткові компоненти, щоб мати змогу тестувати його продуктивність. Поле, позначене як Додатково на рисунку 3.5, - це набір компонентів, необхідних для тестування.

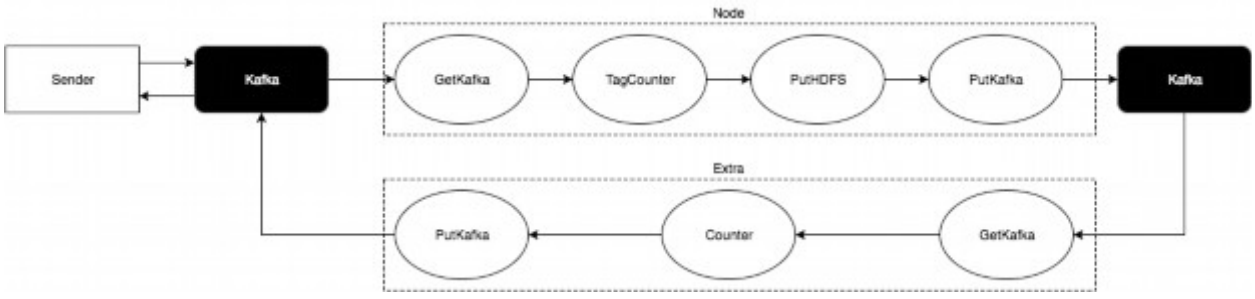

Рис. 3.5. Ілюстрація логічної схеми, що використовується для NiFi

На рисунку 3.5 чорний компонент позначає той самий компонент, намальований таким чином для простоти.

3.4.6. Проектування тестової програми

Як згадувалося, обраний метод полягав у надсиланні всіх повідомлень та очікуванні відповіді, яка означає успіх процесу надсилання. Цей розділ описує, як була створена програма для полегшення цього процесу.

Тестова програма мала одного виробника та одного споживача. Вона надсилала 10 000 повідомлень, а потім чекала, поки не отримає відповідь. Відповідь, яку вона отримає, не буде надіслана, доки всі 10 000 повідомлень не будуть оброблені, і споживач чекав у циклі, доки не буде отримано повідомлення. Це гарантувало, що споживач не завершить роботу передчасно. Коли тестова програма отримувала підтвердження про успішну передачу, вона перезапускала виробника та надсилала ще 10 000 повідомлень. Це повторювалося 40 разів для кожного тесту, щоб досягти більш точних результатів тестування.

Тестова програма також мала власний розподільник. Розподільник - це клас, який використовується для розподілу повідомлень по розділах Kafka способом, який не є стандартним. Цей розподільник був створений для того, щоб виробляти однакову кількість повідомлень для кожного розділу у відповідному топіку.

3.4.7. Налаштування фреймворку

У цьому розділі будуть розглянуті налаштування, специфічні для фреймворку, що стосуються запуску та роботи фреймворків. Оскільки не всі налаштування були змінені, будуть згадані лише ті, які були змінені. Налаштування, специфічні для кластера, тобто IP-адреси та інші налаштування на основі розташування, не будуть згадані, оскільки їх потрібно налаштовувати по-різному залежно від IP-адрес кластера, портів тощо. Також обговорюється внутрішнє тестування, яке було проведено для визначення найкращих параметрів паралелізму. Якщо не було внесено жодних змін до налаштувань за замовчуванням, встановлених у дистрибутиві HDP, вони не будуть представлені в цьому розділі.

NiFi

Щоб налаштувати паралелізм NiFi, довелося провести деякі внутрішні тестування. Цей розділ має на меті розглянути налаштування, що використовуються для NiFi, та налаштування, що використовуються для налаштування ступеня паралелізму.

Усі налаштування були залишені за замовчуванням; це включає всі процесори, які використовувалися в тестах. Однак було змінено поле під назвою "одночасні завдання". Це значення, яке можна встановити для процесора NiFi, щоб вибрати кількість паралельних завдань, які процесор може виконувати.

Веб-інтерфейс NiFi показує багато релевантної статистики при визначенні того, де знаходяться вузькі місця. Завдяки деяким тестуванням було швидко виявлено, що процесор PutHDFS був вузьким місцем, це можна було визначити за часом, який знадобився на файл, та дослідженням черги, яка веде до процесора PutHDFS. Ця черга була єдиною, де накопичувалося кілька повідомлень, що свідчить про те, що цей процесор не міг ефективно обробляти навантаження. Для визначення найкращого налаштування використовувалися різні значення одночасних завдань. Вибір низького значення для цього призводив до ще більшої черги, тоді як вибір занадто великого значення, здавалося, просто призводив до накладних витрат. Були проведені тести, і було виявлено, що встановлення одночасних завдань для процесора XBRLParser та процесора PutHDFS на два та дев'ять одночасних завдань відповідно дало найкращі результати.

Оскільки стандартна модель кластера NiFi полягає в тому, щоб розмістити один і той же потік на кожному вузлі, який має бути включений у ланцюжок обробки, для потоку даних кожного вузла використовувалися ті самі налаштування.

Storm

Цей підрозділ описує, як було налаштовано паралелізм для Storm та які налаштування за замовчуванням були змінені під час виконання тестів. Він

також надасть інформацію про тестування, яке було проведено для отримання цих налаштувань.

Усі налаштування для Storm були залишені за замовчуванням, однак значення для паралелізму не мають значень за замовчуванням і тому мали бути налаштовані під час створення топології. Досліджуючи Storm UI, можна було визначити, де відбуваються вузькі місця. Storm UI показує таку інформацію, як затримка обробки, використовуючи це значення, можна було визначити, якою буде відповідна кількість виконавців.

Спочатку було виявлено, що HDFSBolt був вузьким місцем. Поле в Storm UI під назвою "Ємність" є індикатором, який допомагає розробнику визначити, яким компонентам потрібна більша обчислювальна потужність (більше виконавців). Кількість виконавців для цього болта була збільшена, оскільки було зазначено, що цьому компоненту потрібна більша обчислювальна потужність. Зі збільшенням кількості виконавців затримка зменшувалася, однак це було вірно лише до певного моменту. Коли кількість виконавців перевищувала приблизно 14 на активний вузол кластера, затримка зростала. Це свідчило про те, що накладні витрати на обробку стають занадто великими. Через це було вирішено, що приблизно 14 HDFSBolt будуть використовуватися на кожен тест.

Усі інші компоненти мали відносно хороше використання (знову ж таки, це вказується полем "Ємність" у Storm UI) і не потребували налаштування, на них не витрачалося багато часу.

3.5. Представлення результатів проведеного тестування

Цей розділ присвячений результатам проведених тестів. Він міститиме діаграми, що відображають результати тестів, а також висновки, які можна зробити з результатів. Діаграми базуються на 160 тестах, які були проведені для Storm та NiFi, по 40 на кожне налаштування вузла, яких було чотири. На діаграмах будуть відображені дані, що стосуються швидкості, з якою Storm

та NiFi обробляли 400 000 файлів, та дані щодо середнього часу на файл. Розділ завершується порівнянням двох фреймворків.

3.5.1. Storm

Першими зібраними даними були дані про те, скільки часу в середньому витрачалося на файл. Діаграму цих даних можна побачити на рисунку 3.6. Вона показує, скільки мілісекунд в середньому займав один файл, щоб завершити свій прохід через конвеєр обробки. Смуги похибки представляють 95% впевненість у тому, що файл буде оброблено протягом цього часу.

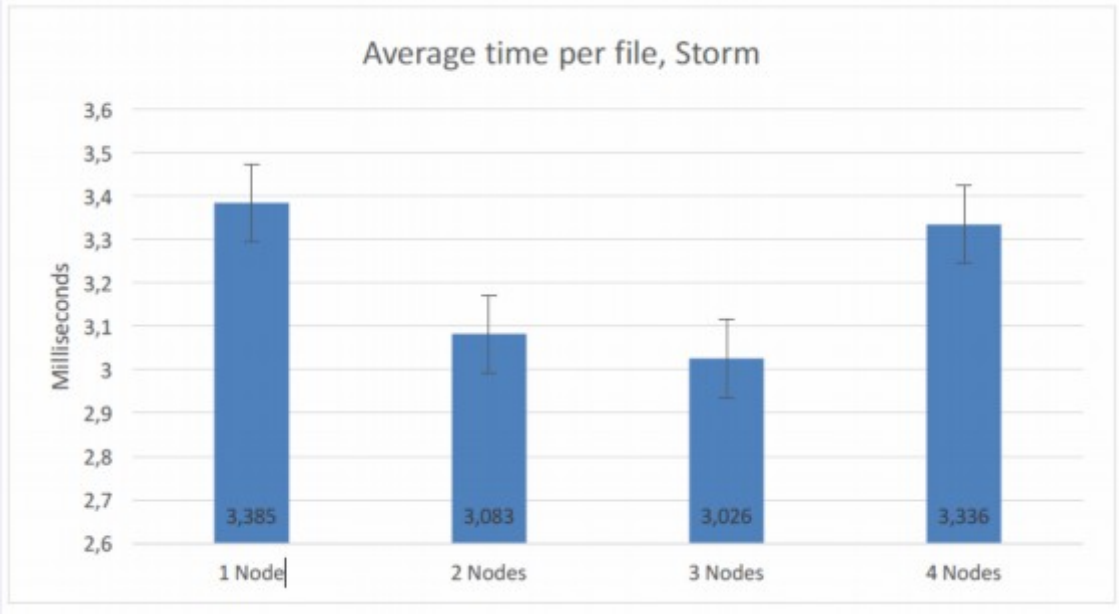

Рис. 3.6. Діаграма, що відображає середній час на файл для Storm, на різній кількості вузлів

Графік на рисунку 3.7 ілюструє часову шкалу того, скільки часу знадобилося кожній конфігурації, відповідно, щоб завершити всі свої тести. Вісь X позначає кількість оброблених файлів, починаючи з нуля і до 400 000 (40 тестів, 10 000 файлів на кожен тест). Вісь Y позначає час, у секундах, який знадобився для завершення тестів.

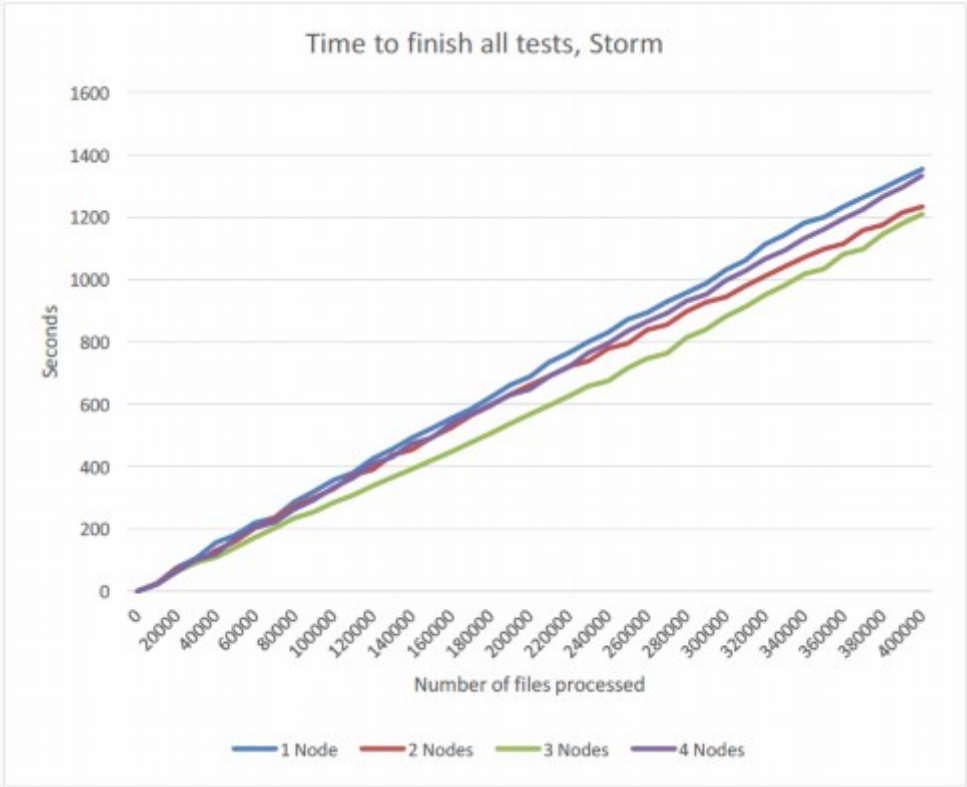

Рис. 3.7. Ілюстрація того, скільки часу знадобилося Storm, щоб завершити всі тести для кожної конфігурації вузлів відповідно

Як можна було зрозуміти з рисунка 3.6, конфігурація з трьома вузлами є оптимальною кількістю вузлів для виконання цього процесу найефективнішим чином, без оптимізації будь-яких налаштувань, крім тих, що стосуються ступеня паралелізму. Цікаво, що на цьому графіку, навіть з секундами на осі Y, лінії відносно нерівні. Це свідчить про великі коливання різниці в часі від тесту до тесту. З цього можна зробити ще один висновок: процес не дуже стабільний. Іноді тести виконувалися набагато швидше, ніж інші, навіть якщо всі параметри були однаковими.

3.5.2. NiFi

Як і в попередньому розділі, перші дані стосуються того, скільки часу в середньому витрачалося на файл. Діаграма на рисунку 3.8 побудована так само, як і в попередньому розділі, і, знову ж таки, смуги похибки позначають 95% впевненість у тому, що результат потрапить у цей інтервал.

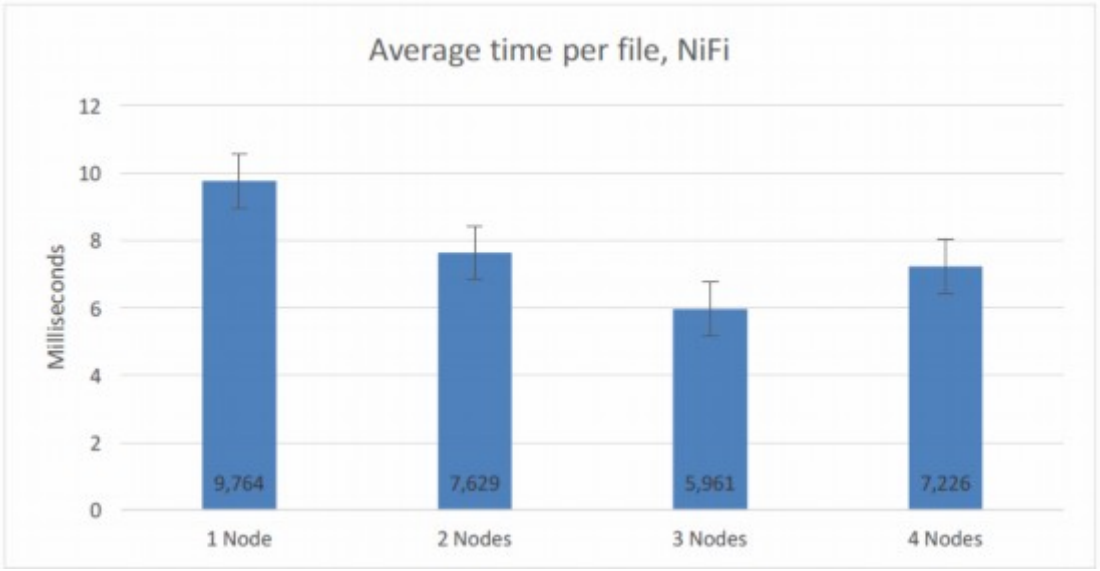

Рис. 3.8. Діаграма, що відображає середній час на файл для NiFi, на різній кількості вузлів

Діаграма, аналогічна діаграмі на рисунку 3.6, показана на рисунку 3.9, вона побудована на основі тих самих параметрів, що й рисунок 3.7, але відображає результати для NiFi.

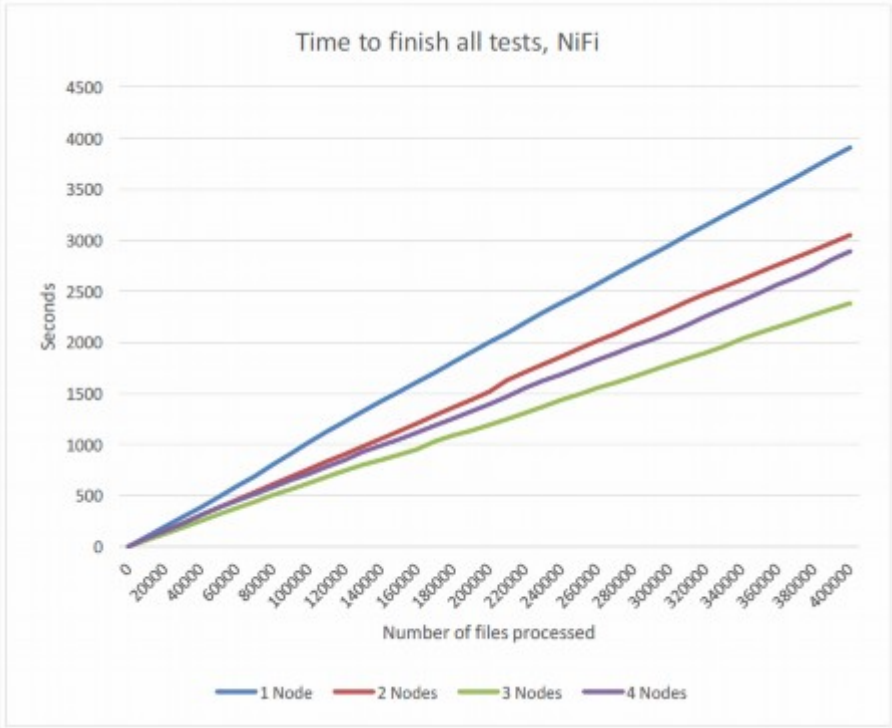

Рис. 3.9. Ілюстрація того, скільки часу знадобилося NiFi, щоб завершити всі тести для кожної конфігурації вузлів відповідно

Результати, показані на цій діаграмі, як і на рисунку 3.7, можна екстраполювати з діаграми, показаної на рисунку 3.8. Конфігурація з трьома вузлами знову є переможцем тесту. Лінії відносно прямі, що свідчить про те, що немає великих коливань від тесту до тесту.

3.5.3. Порівняння результатів

Цей розділ містить порівняння результатів, зіставлення двох фреймворків та дослідження відмінностей. Розділ також містить висновки, які можна зробити з порівняння.

Діаграма нижче, рисунок 3.10, є комбінацією діаграм, показаних на рисунках 3.6 та 3.8. Об'єднавши ці дві діаграми, можна легше візуалізувати інформацію про те, який фреймворк є найшвидшим.

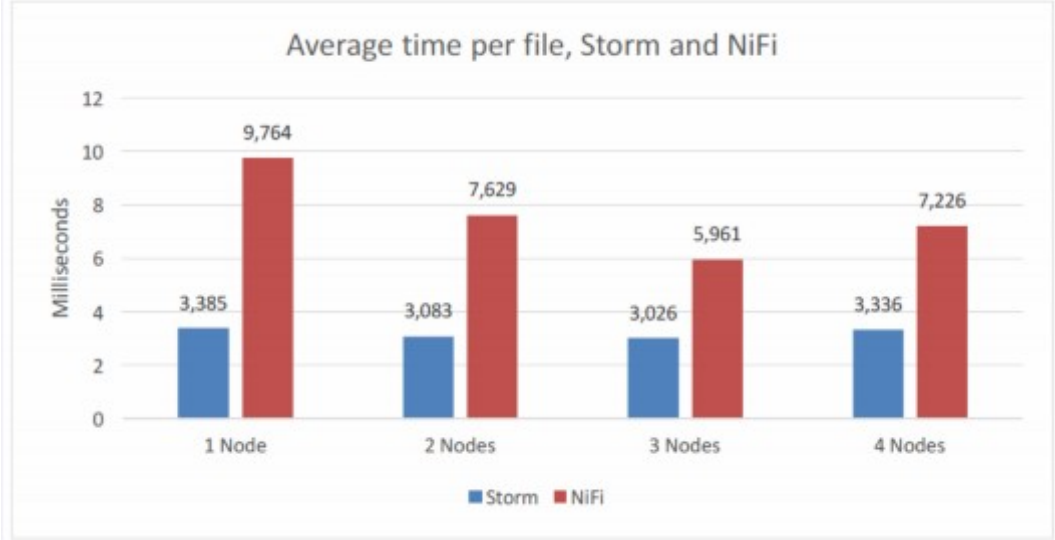

Рис. 3.10. Діаграма для порівняння NiFi та Storm за швидкістю середнього часу обробки на файл

Як видно з рисунка 3.10, Storm є переможцем цього тесту. При налаштуванні на один вузол він майже втричі швидший за NiFi. При трьох вузлах, найкращій конфігурації для обох фреймворків, Storm все ще майже вдвічі швидший. Можна зробити ще один цікавий висновок: NiFi набагато більше виграє від збільшення кількості вузлів у проведених тестах. Також

дуже цікаво, що конфігурація з одним вузлом все ще майже перевершує найкращу конфігурацію NiFi.

На рисунку 3.11 було зроблено комбінацію діаграм, показаних на рисунках 3.9 та 3.7. Він витягнув найкращі результати тестів з обох, а саме конфігурації з трьома вузлами. Для обробки 400 000 файлів Storm витратив приблизно 1200 секунд, тоді як NiFi витратив приблизно 2400 секунд. Ці результати можна було екстраполювати з інших діаграм, але це дає простіший огляд результатів.

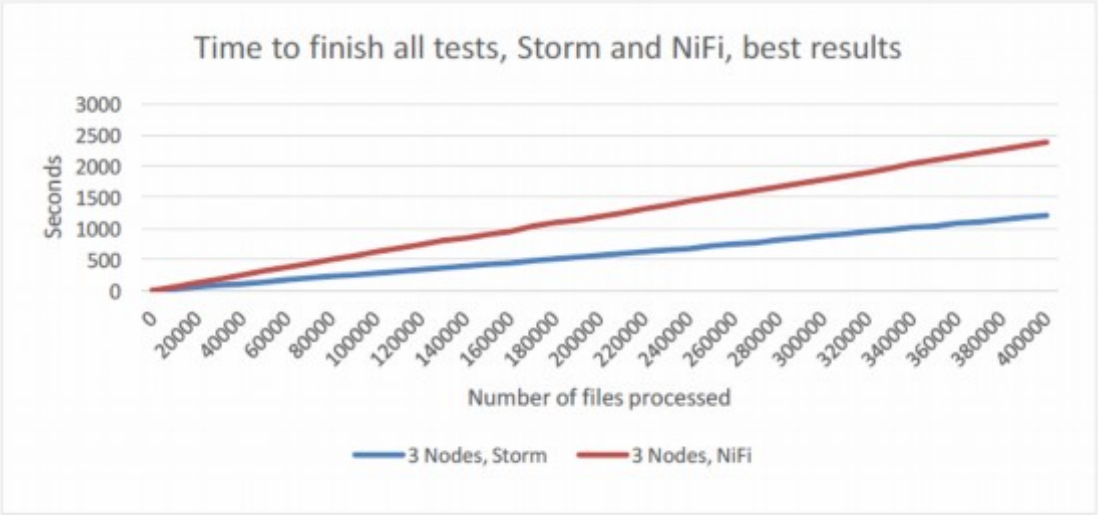

Рис. 3.11. Діаграма, що відображає найкращі конфігурації NiFi та Storm поруч

3.6. Аналіз отриманих результатів

У цьому розділі аналізуються результати та обговорюються причини, чому результати могли виявитися такими, якими вони є. Відмінності між NiFi та Storm будуть обговорені в цьому розділі, оскільки вони мають велике значення для розуміння отриманих результатів.

3.6.1. Застосування результатів

На початку третього розділу було запропоновано гіпотетичний варіант використання, який передбачав обробку 70 000 файлів протягом дня для

обробки фінансової інформації. У цьому розділі аналізується, що ймовірно станеться, коли буде надіслано 70 000 повідомлень замість 10 000, як це було в тестах.

Використовуючи найкращі конфігурації вузлів, які були знайдені для NiFi та Storm відповідно, середній час на повідомлення залишався досить стабільним, і можна очікувати, що він залишиться таким і при збільшенні навантаження повідомленнями.

Якщо припустити, що це так, і використовуючи найкращі налаштування для NiFi та Storm (конфігурація з трьома вузлами), час для надсилання всіх 70 000 повідомлень, ймовірно, буде в межах, наведених нижче.

Середній час обробки 70 000 повідомлень:

- Storm – 3,53 хвилини
- NiFi – 6,96 хвилини

95% довірчий інтервал часу обробки:

- Storm – від 3,12 до 3,94 хвилини
- NiFi – від 6,64 до 7,23 хвилини

Це означає, що навіть якщо використовувати NiFi, цілком можливо обробляти 70 000 файлів на день, оскільки для цього, ймовірно, знадобиться лише близько семи хвилин. Схоже, що навіть у найгіршому випадку це не займе набагато більше семи хвилин.

Проводячи такі вимірювання та тести, можна визначити, чи дійсно кластер повинен бути таким потужним, яким він є, або чи дійсно йому потрібно стільки вузлів, скільки він має. У цьому випадку метою було обробляти 70 000 файлів на день. Це завдання легко досяжне, і якби це було єдине завдання, яке мав виконувати кластер, він був би надмірно потужним. Якщо він надмірно потужний, можливо, варто спробувати знайти інші, додаткові завдання для кластера або просто зменшити його масштаб, щоб знизити споживання енергії та заощадити кошти.

3.6.2. Аналіз результатів та платформ

Storm позиціонує себе як розподілену систему обчислень у реальному часі, тоді як NiFi позиціонує себе як систему для маршрутизації даних, перетворення та логіки посередництва систем [37]. Порівнюючи те, чим вони обидва себе позиціонують, результати очікувано виявилися такими, якими вони були для нас.

Цікавим є те, що середній час обробки на файл для Storm зменшується, доки рішення не масштабується до чотирьох вузлів. Відомо, що HDFSBolt мав найбільшу затримку з усіх компонентів, ймовірно, це вузьке місце, яке неможливо подолати шляхом горизонтального масштабування. З цього можна зробити висновок, що додавання вищого ступеня паралелізму до рішення підвищує продуктивність, але лише до певної міри. Після цього, ймовірно, доведеться вжити інших заходів для скорочення часу обробки.

Цікаво, що на рисунку 3.8 показано ту саму закономірність, що й у третьому розділі. Три вузли, здається, працюють найкраще для цього тестового випадку. Це призводить до того самого висновку, але цього разу для процесора PutHDFS, а не для HDFSBolt. Після певного моменту додавання більшої кількості вузлів, здається, не допомагає і в NiFi.

Storm зосереджений на аналітиці та обчисленнях у реальному часі, тоді як NiFi створений як інтуїтивно зрозумілий спосіб розподілу даних через потік даних. Storm виграв з досить великим відривом, але для даного випадку використання NiFi був більш ніж достатнім.

На нашу думку, з обома платформами було дуже приємно працювати, хоча стало очевидно, що вони заповнюють різні ніші; це зробило їх цікавими для тестування один проти одного. NiFi легко переналаштовувався після розгортання, нічого не потрібно було перекомпілювати, якщо розгорнуті процесори мали поля, що налаштовуються, для всіх значень, які потребували б зміни на льоту. Це зробило тестування NiFi дуже плавним при зміні конфігурації вузла. Storm, з іншого боку, якщо дані потрібно маршрутизувати

по-іншому або обробляти іншим чином, вимагає перепакування рішення в новий JAR-файл, розгортання його в кластері та перезапуску.

Ми дійшли висновку, що обидві платформи є дуже цінними самі по собі, і що використання правильного інструменту для роботи не завжди є строго необхідним. Ми також дійшли висновку, що вибір правильних інструментів для правильної роботи є складним, коли немає легкодоступних тестів для конкретного випадку використання. У цій дисертації не розглядалося завдання налаштування платформ оптимальним чином, що може дати різні результати.

Пошук інших способів підвищення продуктивності HDFS вимагатиме заглиблення в безліч налаштувань, які можна змінити під час налаштування кластера HDFS. На це не вистачило часу, і тому ми можемо лише здогадуватися про причини вузького місця.

Існує безліч налаштувань для обох платформ, починаючи від таких речей, як кількість файлів, дозволених у системі, до більш специфічних налаштувань, таких як поведінка певного Processor, Bolt або Spout. Цілком ймовірно, що подальші покращення можна було б внести, дослідивши можливі налаштування для NiFi та Storm. Якби було більше часу, частина цього часу, ймовірно, була б витрачена на дослідження специфіки, що стосується покращення продуктивності обох платформ.

Приділяючи час тестуванню різних реалізацій обробки великих даних, можливо, можна приймати рішення, про які інакше не подумали б. Потенційним висновком може бути те, що кількість вузлів можна зменшити, або що потужність машин можна зменшити. Таким чином, проведення таких тестів може мати цінність у більш ніж одному аспекті. Якщо компанія використовує занадто багато машин для обробки даних, вона може витрачати ресурси у вигляді споживання енергії. Якщо є відчутні результати, на які компанія може подивитися і зрозуміти, що їй, можливо, не потрібно використовувати такі потужні машини або таку велику кількість вузлів, можна зменшити їх кількість.

Тести показали, що збільшення кількості вузлів погіршувало продуктивність, як тільки вона перевищувала три. Якби чотири вузли використовувалися в надії на кращу продуктивність, було б витрачено більше енергії без жодної користі. Знаючи оптимальну кількість вузлів для використання в конкретному випадку, можна заощадити енергію. Це приносить користь як навколишньому середовищу, так і знижує витрати для компанії, яка використовує такий кластер.

Оскільки все більше і більше соціальних даних доступно через Інтернет з таких джерел, як Facebook, Twitter та інші соціальні мережі, може з'явитися можливість отримати цінність з цих даних, якщо швидкість, з якою це робиться, достатньо висока. Якщо обробка даних такого масштабу проводиться, існує ймовірність постійного спостереження за людьми. Щоденна діяльність, місця, куди вони ходять, кого вони знають та інші знання можуть бути використані, якщо таке спостереження проводиться. Це серйозно впливає на конфіденційність, на яку люди можуть вважати себе вправі, навіть якщо вони публікують дані добровільно. Маючи доступ до даних такого масштабу, може виникнути безліч етичних дилем. Тести показують найкращу конфігурацію, яку ми змогли розробити в цій дисертації. Надаючи такі тести, їх можна використовувати не за призначенням.

Висновки до розділу

У третьому розділі детально розглянуто практичне застосування методів та засобів фреймворків для обробки великих даних. Початково було проведено вибір відповідних інструментів на основі порівняльного аналізу. Обґрунтування вибору фреймворків ґрунтувалося на таких критеріях, як продуктивність, масштабованість, адаптивність до різних сценаріїв використання та підтримка широкого спектра форматів даних. Серед обраних платформ виділялися Apache Kafka, Apache NiFi та Apache Storm.

Особливу увагу приділено налаштуванню обраних фреймворків для ефективної роботи з великими обсягами даних. Методологія тестування включала створення специфікації кластерів, налаштування конфігурацій компонентів та використання підходів бенчмаркінгу. Тести проводилися на кількох фреймворках, серед яких Apache Storm та Apache NiFi, з подальшим порівнянням результатів.

Результати показали переваги кожного з інструментів у різних аспектах. Apache Storm відзначився високою швидкістю обробки даних у режимі реального часу, тоді як Apache NiFi виявився зручним у налаштуванні та управлінні потоками. Аналіз результатів дозволив визначити найефективніші підходи для різних сценаріїв використання.

ВИСНОВКИ

У магістерській роботі досліджено моделі, методи та інструменти фреймворків обробки великих даних, зосереджуючись на розробці ефективного підходу до обробки значної кількості відносно невеликих файлів. Основною метою дослідження було тестування двох фреймворків — Apache NiFi та Apache Storm — для оцінки їхньої продуктивності та масштабованості.

Було розроблено методику, що включає побудову потоку даних і створення тестового сценарію для оцінки ефективності та масштабованості зазначених платформ. Результати тестування продемонстрували, що Apache Storm перевершує Apache NiFi за швидкістю виконання завдань, які аналізувалися в межах цього дослідження. Однак масштабованість платформ виявилася обмеженою: збільшення кількості вузлів у кластері не завжди приводило до покращення продуктивності, що свідчить про необхідність додаткових підходів до оптимізації обробки даних.

Дослідження було зосереджено на обробці XML-файлів, відповідних специфікації XBRL, середнім розміром близько 50 КБ. Обидва фреймворки виявилися придатними для цього сценарію. Apache NiFi показав себе як більш універсальний інструмент із легким налаштуванням, яке іноді можна здійснювати без зупинки кластера, що є його перевагою. Водночас Apache Storm забезпечує вищу швидкість обробки, що є важливим для великомасштабних задач.

Таким чином, дослідження підтвердило, що вибір платформи для обробки великих даних залежить від конкретного сценарію використання. Для задач, де пріоритетними є швидкість та масштабованість, Apache Storm є оптимальним вибором. У випадках, коли важливіше гнучкість налаштування та універсальність, доцільно використовувати Apache NiFi. Результати цієї роботи можуть бути корисними для прийняття рішень щодо вибору відповідних платформ у подібних проектах із обробки потоків даних.

Завдяки цьому дослідженню стало можливим визначити оптимальні методи інтеграції обраних платформ для забезпечення максимальної продуктивності та адаптивності при роботі з великими даними. Отримані результати є цінними для практичного використання в різних галузях, включаючи фінансову аналітику, IoT, бізнес-моніторинг та інші сфери, що вимагають обробки великих потоків інформації.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. P. Russom, "Big data analytics", 2011. [Online]. Available: http://www.tableau.com/sites/default/files/whitepapers/tdwi_bpreport_q411_big_data_analytics_tableau.pdf.
2. C. Min, M. Shiwen and L. Yunhao, "Big Data: A Survey", Springer Science+Business Media, New York, 2014.
3. J. Gantz and D. Reinsel, "Extracting Value from Chaos", June 2011. [Online]. Available: https://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf.
4. EY, "Under cyber attack", October 2013. [Online]. Available: http://www.ey.com/Publication/vwLUAssets/EY_-_2013_Global_Information_Security_Survey/$FILE/EY-GISS-Under-cyber-attack.pdf.
5. J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh and A. Hung Byers, "Big data: The next frontier for innovation, competition, and productivity", McKinsey Global Institute, 2011.
6. B. Marr, "How is Big Data Used in Practice?", Advanced Performance Institute, [Online]. Available: http://www.ap-institute.com/big-data-articles/how-is-big-data-used-in-practice-10-usecases-everyone-should-read.aspx.
7. KPMG, "DATA AND ANALYTICS: A New Driver of Performance and Valuation", October 2015. [Online]. Available: http://www.kpmg.com/US/en/topics/dataanalytics/Documents/DA_ADriverofPerformanceandValuation.pdf.
8. SAS, "Five big data challenges", 2013. [Online]. Available: https://www.sas.com/resources/asset/five-big-data-challenges-article.pdf.
9. D. Agrawal, E. Bertino, S. Davidson, M. Franklin, A. Halevy, J. Han, H. V. Jagadish, S. Madden, Y. Papakonstantinou, R. Ramakrishnan, K. Ross, C. Shahabi, S. Vaithyanathan and J. Widom, "Challenges and Opportunities

with Big Data", 2012. [Online]. Available: http://cra.org/ccc/wpcontent/uploads/sites/2/2015/05/bigdatawhitepaper.pdf.

10. K. Rupp, "40 Years of Microprocessor Trend Data", 25 June 2015. [Online]. Available: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/.
11. Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM, 51(1), 107–113.
12. Zaharia, M., et al. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. USENIX Symposium on Networked Systems Design and Implementation.
13. Tomar, D., & Agarwal, S. (2013). A survey on pre-processing and post-processing techniques in data mining. International Journal of Database Theory and Application, 6(6), 1–8.
14. Grolinger, K., et al. (2014). Challenges for MapReduce in Big Data. IEEE World Congress on Services.
15. Chen, M., Mao, S., & Liu, Y. (2014). Big Data: A Survey. Mobile Networks and Applications, 19(2), 171–209.
16. White, T. (2015). Hadoop: The Definitive Guide. O'Reilly Media.
17. Zaharia, M., et al. (2016). Apache Spark: A Unified Engine for Big Data Processing. Communications of the ACM, 59(11), 56–65.
18. Gorton, I., & Klein, J. (2015). Distribution, Data, Deployment: Software Architecture Convergence in Big Data Systems. IEEE Software, 32(3), 78–85.
19. Kang, S., et al. (2017). Exploring Emerging Technologies in Big Data Analytics. Journal of Big Data, 4(1), 22.
20. Song, I. Y., & Zhu, Y. (2016). Big Data and Data Science: Opportunities and Challenges. Journal of Data Science and Analytics, 1(1), 1–9.
21. Shvachko, K., et al. (2010). The Hadoop Distributed File System. IEEE 26th Symposium on Mass Storage Systems and Technologies.

22. McSherry, F., et al. (2015). Scalable, Distributed Differential Privacy. Proceedings of the VLDB Endowment, 8(14), 2080–2091.
23. Karau, H., & Warren, R. (2017). High Performance Spark. O'Reilly Media.
24. Marz, N., & Warren, J. (2015). Big Data: Principles and Best Practices of Scalable Realtime Data Systems. Manning Publications.
25. Shi, Y., et al. (2016). A Survey of Interactive Visualization for Smart Energy Systems. Big Data Research, 5, 1–9.
26. Kumar, V., et al. (2015). Frameworks for Big Data Analytics: A Systematic Literature Review. Journal of Grid Computing, 13(4), 457–492.
27. Lin, J., & Ryaboy, D. (2013). Scaling Big Data Mining Infrastructure: The Twitter Experience. ACM SIGKDD Explorations Newsletter, 14(2), 6–19.
28. Sakr, S., & Gaber, M. M. (2014). Large Scale and Big Data: Processing and Management. Chapman and Hall/CRC.
29. Alexandrov, A., et al. (2014). The Stratosphere Platform for Big Data Analytics. The VLDB Journal, 23(6), 939–964.
30. Tsai, C.-W., et al. (2015). Big Data Analytics: A Survey. Journal of Big Data, 2(1), 1–32.
31. Cuzzocrea, A., et al. (2011). Analytics over Large-Scale Multidimensional Data: The Big Data Revolution! Proceedings of the ACM International Workshop on Data Warehousing and OLAP.
32. Aridhi, S., et al. (2016). Big Data Frameworks: A Comprehensive Review. Journal of Big Data, 3(1), 1–31.
33. Rehman, M. H., et al. (2016). Big Data Analytics in Industrial IoT: A Survey. IEEE Access, 4, 141–160.
34. Chervenak, A. L., et al. (2013). Giggle: A Framework for Integrating Big Data Computing into Scientific Workflows. ACM Transactions on Intelligent Systems and Technology, 5(3), 12.
35. Kambatla, K., et al. (2014). Trends in Big Data Analytics. Journal of Parallel and Distributed Computing, 74(7), 2561–2573.

36. Nishant, R., et al. (2017). Adoption of Big Data Technologies in Organizations: A Multi-theoretic Perspective. Journal of Information Technology, 32(3), 209–228.
37. Rahm, E., & Do, H. H. (2000). Data Cleaning: Problems and Current Approaches. IEEE Data Engineering Bulletin, 23(4), 3–13.
38. Zhao, D., & Zhang, L. (2014). Distributed Feature Selection for Efficient Big Data Processing. IEEE Transactions on Big Data, 1(1), 21–33.
39. Minelli, M., et al. (2013). Big Data, Big Analytics: Emerging Business Intelligence and Analytic Trends for Today's Businesses. Wiley.
40. Baer, T., et al. (2016). Challenges in Building Big Data Architectures. IBM Systems Journal, 45(4), 451–464.
41. Molina, A. I., et al. (2016). Applications of Big Data in Knowledge Discovery. International Journal of Computer Science Issues, 13(2), 5–13.
42. Warden, P. (2011). Big Data Glossary. O'Reilly Media.
43. Marr, B. (2015). Big Data: Using Smart Big Data, Analytics and Metrics to Make Better Decisions and Improve Performance. Wiley.
44. Hashem, I. A. T., et al. (2015). The Role of Big Data in Smart City Development. Journal of Smart Cities, 1(2), 12–24.
45. Dobre, C., & Xhafa, F. (2014). Intelligent Services for Big Data Science. Future Generation Computer Systems, 37, 267–281.
46. Sarkar, D., et al. (2018). Practical Machine Learning with Hadoop and Spark. Packt Publishing.
47. Aggarwal, C. C. (2015). Data Mining: The Textbook. Springer.
48. Ranjan, J. (2009). Business Intelligence: Concepts, Components, Techniques, and Benefits. Journal of Theoretical and Applied Information Technology, 9(1), 60–70.
49. Akter, S., & Wamba, S. F. (2016). Big Data Analytics in E-commerce: A Systematic Review. Journal of Business Research, 70, 238–246.
50. Che, D., et al. (2013). From Big Data to Big Data Mining: Challenges, Issues, and Opportunities. Database Systems Journal, 4(3), 88–100.