Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій факультету інформатики

ДОСЛІДЖЕННЯ ТЕХНОЛОГІЙ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ

Текстова частина до курсової роботи

за спеціальністю «Інженерія програмного забезпечення» - 121

Керівник курсової роботи

Старший викладач Гречко А. В.

_____

(підпис)

“____” _____ 2022 р.

Виконав студент Янкін І.С.

_____

(підпис)

“____” _____ 2022 р.

Київ-2022

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

_____

(прізвище та ініціали)

_____

(підпис)

“____” _____ 2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Янкіну Ігорю Сергійовичу

факультету інформатики 4 р.н. бакалаврської програми

ТЕМА: Дослідження технологій розподілених обчислень

Зміст ТЧ до курсової роботи:

Календарний план

Вступ

Розділ 1. Аналіз існуючих технологій та постановка завдання

Розділ 2. Теоретичне дослідження обраних фреймворків та їх можливостей

Розділ 3. Реалізація обраних задач та аналіз отриманих результатів

Висновки

Список використаної літератури

Додатки (за необхідністю)

Дата видачі “____” _____ 2021 р. Керівник _____*(підпис)*

Завдання отримав _____*(підпис)*

Тема: Дослідження технологій розподілених обчислень

Календарний план виконання роботи:

№	Назва етапу	Термін виконання	Примітка
1.	Отримання завдання на курсову роботу	28.09.2021	
2.	Пошук інформаційних джерел за темою	01.11.2021	
3.	Ознайомлення зі знайденою літературою	01.01.2022	
4.	Обрання технологій, які будуть розглядатися	15.01.2022	
5.	Теоретичні дослідження	15.03.2022	
6.	Реалізація обраних імплементацій	15.04.2022	
7.	Аналіз отриманих результатів	01.05.2022	
8.	Написання текстової частини	20.05.2022	
10.	Завантаження роботи для перевірки на плагіат		
11.	Захист роботи		

Студент Янкін І.С.

Керівник Гречко А. В.

“___” _____

ЗМІСТ

АНОТАЦІЯ

У даній курсовій роботі досліджено існуючі технології розподілених обчислень, що дозволяють об’єднати декілька пристроїв для вирішення складних задач з використанням обчислювальних потужностей задля пришвидшення виконання розрахунків.

Проаналізовано найпопулярніші та найбільш поширені реалізації, що дозволяють виконувати розподіленні обчислення та є актуальними станом на 2022 рік. Розглянуто їх переваги, недоліки та особливості. Проведено практичні випробування на обраних фреймворках для того, щоб підтвердити або спростувати твердження, висунуті у теоретичній частині.

Отримані результати ретельно проаналізовано та на основі цього зроблено висновки стосовно використаних у роботі технологій.

Ключові слова: розподіленні обчислення, кластерні технології, паралельні обчислення, фреймворки для розподілених обчислень, розподілені данні.

ВСТУП

Опис теми дослідження

До розподілених обчислень відносяться такі розрахунки, що виконуються за допомогою двох або більше пристроїв (не обов’язково однакової продуктивності), що з'єднані в одну мережу та виконують одне завдання з використанням потужностей усіх засобів мережі.

За даними пошукової статистики Google [1,2] на сьогодні обробка та аналіз великої кількості даних є одним з найпопулярніших класів задач і можна прогнозувати, що ця тенденція буде спостерігатиметься і в майбутньому. Для вирішення задач з цієї галузі за прийнятний час вже не вистачає потужностей одного пристрою, навіть з використанням паралельного програмування, тому розподіленні обчислення, які дозволяють вирішити ці задачі – галузь, що стрімко розвивається та вдосконалюється. Наприклад, одним з перших фреймворків був MapReduce [3], який і досі залишається доволі популярним та широко використовуваним, хоч і дещо втратив свої позиції останнім часом [4]. Але на його заміну з’являється багато нових засобів, що швидко займають свою нішу, зокрема Julia [5], Dask [6], Spark [7], Dryad [8], Ignite [9], Impala [10], Drill [11], Kudu [12], Cylon [13].

Завдяки цьому розвитку у користувача існує багато варіантів, яку саме систему використати для вирішення певної задачі. Однак у вільному доступі замало інформації про переваги одних фреймворків над іншими та особливості їх роботи, що значно ускладнює обрання. У роботі пропонується завдяки теоретичним дослідженням існуючих фреймворків та практичній реалізації деяких класичних задач проаналізувати їх та на основі цього зробити висновки про їх переваги, недоліки та особливості.

Структура роботи

Робота складається з 3 розділів.

У першій частині аналізуються існуючі технології для вирішення задач розподілених обчислень, зазначаються популярні реалізації на основі цих технологій та формуються критерії для порівняння, а також визначаються задачі, на яких пропонується проводити випробування обраних фреймворків.

Друга частина містить детальній розбір обраних раніше імплементацій, визначення їх особливостей та висуваються твердження стосовно цих реалізацій, які пропонується підтвердити або спростити.

Третій розділ складається з описання практичної роботи, що була проведена, аналізу результатів, що були отримані під час цієї роботи та висновків про переваги та недоліки реалізованих фреймворків, базуючись на цьому аналізі.

1 АНАЛІЗ ІСНУЮЧИХ ТЕХНОЛОГІЙ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Обрання фреймворків

Стрімкий розвиток та дедалі більша популярність розподілених обчислень для вирішення прикладних задач пов’язані в основному з двома факторами. Перший – значне збільшення кількості даних, які потрібно обробляти для вирішення деяких класів задач [14]. З цим об’ємом один пристрій вже не здатен впоратись за прийнятний час. Другий – легкодоступність сервісів, у тому числі і хмарних [15], що дозволяють орендувати додаткові пристрої за доволі невелику ціну.

Зазвичай задачі, для яких використовують розподілене обчислення пов’язані з обробкою великої кількості неструктурованих даних у реальному часі. Більшість з них дозволяє запровадити алгоритми, спроектовані за допомогою парадигми «Розділяй та володарюй», яка передбачає розбиття початкової задачі на декілька ідентичних задач меншого розміру і комбінування їх розв’язків для отримання відповіді на початкову задачу.

У цій роботі буде розглянуто та проаналізовано такі засоби для розподілених обчислень:

- Spark
- Dask
- Flink

Їх вибір в якості предмету дослідження визначили популярність цих фреймворків, наявність достатньої кількості довідкових матеріалів та документації, а також їх активний розвиток станом на 2022 рік.

1.2 Обрання критеріїв та задач

Основними параметрами, за якими вирішено порівнювати обрані засоби паралельних обчислень в ході практичної реалізації, - це швидкість, здатність працювати з об’ємами даних, більшими за оперативну пам’ять, та легкість використання фреймворку. Швидкість важлива, адже прискорення роботи програми є однією з цілей з якою розподіленні обчислення зазвичай використовуються та є показником ефективності певної реалізації. Здатність працювати з об’ємами даних необхідна, адже більшість задач має розмір, який не дозволяє розмістити усю інформацію у пам’яті через її розмір. Не менш важливою є і легкість у використанні, тому що це впливає на кількість користувачів та за умови однакової ефективності реалізації користувачем зазвичай буде обрана та, якою легше оволодіти та для якої існує більше довідкових матеріалів та керівництв.

Для подальшої реалізації під час практичної роботи пропонуються наступні задачі:

- Сортування масиву
- Підрахунок входження кожного слова до тексту
- Пошук у структурованих логах певного поля та обрахунок його середнього значення

Вищезазначені задачі є класичними для розподіленого навчання та здатні продемонструвати різницю у використанні обраних фреймворків.

1.3 Постановка задачі

У даній курсовій роботі передбачено:

- Реалізувати перелічені вище задачі з використанням фреймворків для розподілених обчислень Spark, Dask та Flink;

- Порівняти та проаналізувати отримані реалізації з точки зору їх швидкості, здатності працювати з великими об’ємами даних, та легкості використання;
- На основі отриманих під час дослідження результатів зробити висновки стосовно переваг та недоліків розглянутих фреймворків.

2 ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ ОБРАНИХ ФРЕЙМВОРКІВ ТА ЇХ МОЖЛИВОСТЕЙ

2.1 Apache Spark

2.1.1 Основні відомості

Apache Spark – фреймворк для розподіленого обчислення з відкритим кодом, що входить до екосистеми Hadoop. Ця система містить інструменти для розробки та виконання розподілених програм та складається з 4 основних компонентів, а саме:

- Hadoop Common
- HDFS
- YARN
- Hadoop MR

2.1.2 Hadoop Common

Hadoop Common – набір загальних інструментів, що включають окремі утиліти та бібліотеки, які є допоміжними для усіх іншим модулів системи. Так, саме цей модуль містить абстракції вищого рівня, на яких базуються реалізації інших компонентів та скрипти, необідні для запуску системи. Особливість даних інструментів полягає у припущенні, щодо високої вірогідності апаратних помилок та їх відповідної автоматичної обробки.

2.1.3 HDFS

HDFS – розподілена файлова система, за допомогою якої відбувається обмін даними між її окремими елементами. Архітектура HDFS (рис. 2.1.1) передбачає наявність головного вузла, який відповідає за розміщення даних та надання шляхів для отримання цих даних та вузлів, які зберігають ці дані. Система передбачає дублювання певної частини даних за спеціальними правилами, що дозволяє говорити про високу степінь відмовостійкості у разі

виникнення проблем на одному з кінцевих вузлів [16]. Також значною перевагою є API, який дозволяє програмам виконувати свій код якомога ближче до даних, що призводить до зменшення накладних витрат на роботу з цими даними [17].

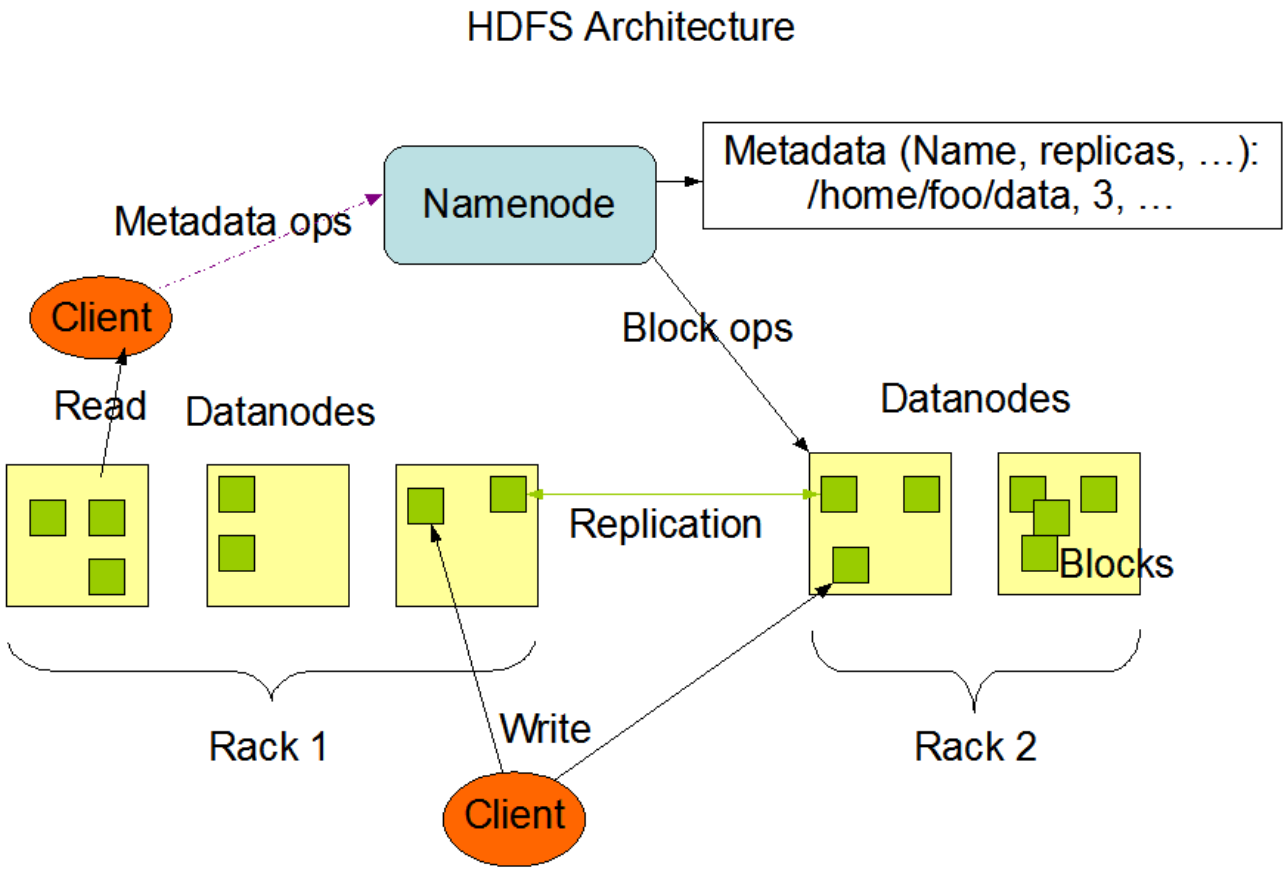

Рисунок 2.1.1 – Архітектура HDFS [18]

2.1.4 YARN

YARN – система керування розподіленими додатками, яка була розроблена з метою розділення трекера задач [19], який використовувався до цього, на дві частини, а саме на керування ресурсами та на планувальник задач. Саме так, YARN складається з двох компонентів, а саме з глобального менеджеру ресурсів та з майстра додатків. Менеджер ресурсів існує глобально та відповідає за розподілення ресурсів між окремими компонентами системи на основі запитів, які він отримує від майстра додатків. На відміну від нього, майстер додатків має окремий екземпляр на кожному з пристроїв та відповідає за надсилання менеджеру запитів на отримання необхідних ресурсів та стежить за процесом виконання задач. Під час роботи менеджер та майстер комунікують

з менеджером вузлів, який, аналогічно до майстра додатків має екземпляр на кожному пристрої. За допомогою нього відбувається керування контейнерами та відстеження поточних ресурсів на пристрої. Така архітектура (рис. 2.1.2) значно спрощує масштабування та дозволяє інтегруватись з відмінними від MapReduce парадигмами.

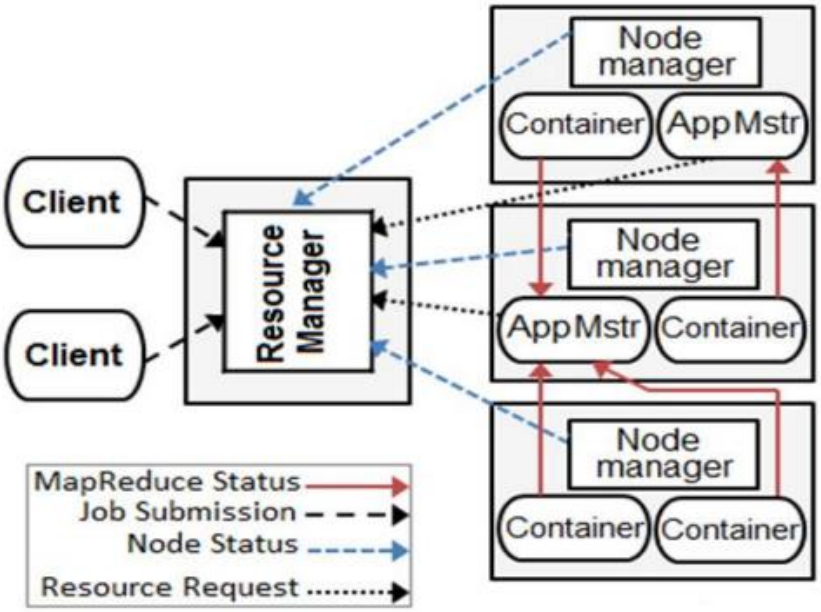

Рисунок 2.1.2 – Архітектура YARN [20]

2.1.5 Hadoop MapReduce

Hadoop MapReduce – класичний фреймворк, що відповідає за розподіленні обчислення. Його названо на честь однойменної парадигми обчислень, яку він імплементує. Ця парадигма передбачає розбиття даних на декілька частин, що обробляються незалежно одна від одної. Кожна частина проходить через декілька кроків, основними з яких є Map та Reduce. Перший крок виконує обробку даних, а другий – згортку оброблених даних. Головний вузол отримує ці результати та формує кінцеву відповідь. Цей фреймворк набув піка своєї популярності у 2014-2015 роках [21], але згодом поступився Apache Spark, хоча і досі займає власну нішу через переваги під час вирішення деяких задач [22]

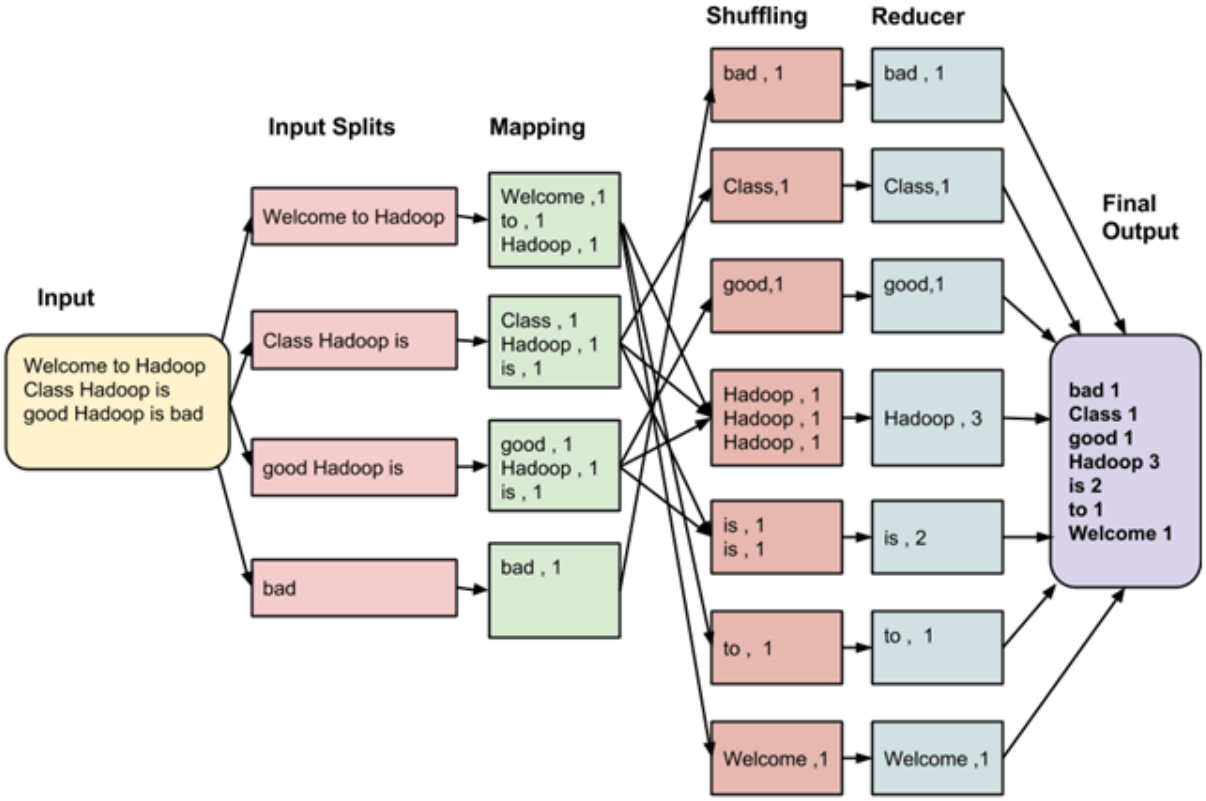

Рисунок 2.1.3 – Архітектура MapReduce [23]

2.1.6 Особливості Apache Spark

Spark, як і MapReduce, є фреймворком у екосистемі Hadoop, завдяки чому він без проблем інтегрується та взаємодіє з усіма її компонентами (рис. 2.1.4) (включно з MapReduce) та розширює її можливості, додаючи власні.

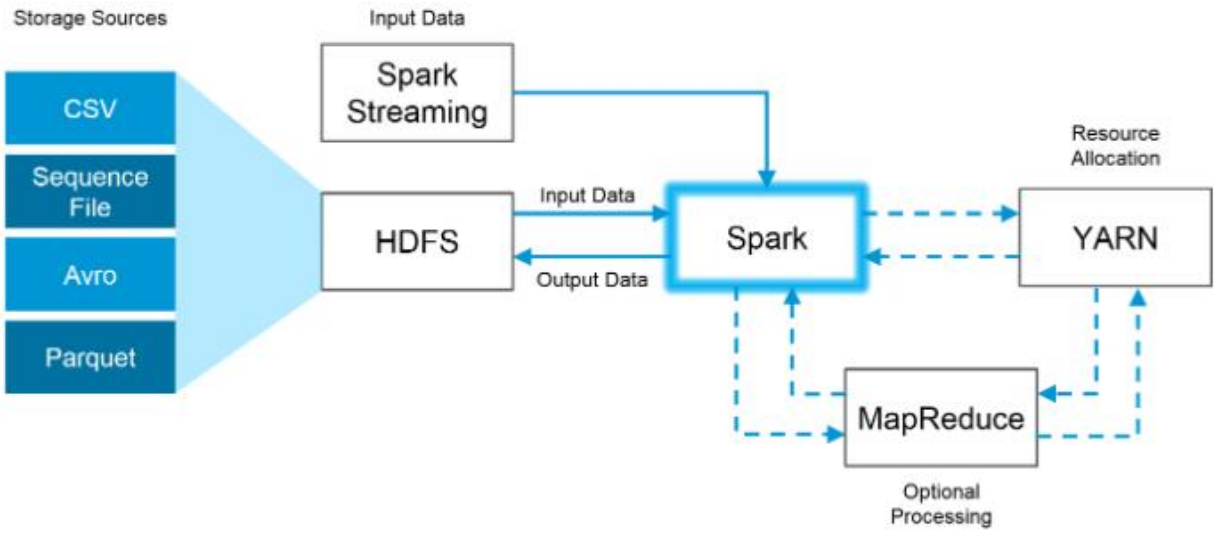

Рисунок 2.1.4 – Інтеграція Spark з екосистемою Hadoop [24]

Так, до основних особливостей можна віднести:

- Обчислення в оперативній пам’яті
- Інтерфейси для інших мов
- Бібліотеки, що значно розширюють початкові можливості фреймворку
- Можливість працювати окремо від екосистеми Hadoop

Кожна з цих особливостей є важливими та разом становлять причину, чому Spark швидко завоював популярність користувачів та досі є одним з найбільш вживаних фреймворків для розподілених обчислень. Так, завдяки підтримці таких мов, як Scala, Python, R та Java він покриває 28.8 відсотків користувачів, згідно зі статистикою 2021 року [25], а здатність працювати разом з іншими системами окремо від Hadoop дозволяє обрати потрібну конфігурацію в залежності від проекту. Разом з цим, обчислення в оперативній пам’яті дозволяють отримати значну перевагу у швидкості [26] для деяких задач (наприклад, для задач машинного навчання або при роботі з sql запитами).

Окремо варто виділити бібліотеки, такі як: Spark SQL [27] для втілення sql запитів над даними, Spark MLlib [28] для роботи з задачами машинного навчання, Spark Streaming [29] для обробки потокових даних та GraphX [30] для обробки графів. Разом з гнучкими налаштуваннями Spark [31] – це дозволяє ефективно вирішувати практичні задачі витрачаючи на це мінімум зусиль.

2.2 Dask

2.2.1 Основні відомості

Dask – фреймворк для розподілених обчислень, який було розроблено для мови програмування Python. Завдяки цьому він чудово оптимізований для роботи зі структурами даних Python та найбільш популярними бібліотеками для роботи з даними, такими як pandas [32], scikit-learn [33] та numpy [34]. Окрім цього, самі розробники виділяють наступні переваги [35]:

- Гнучкість
- Швидкість

- Масштабованість
- Чутливість
- Легкість у розгортці
- Легкість у використанні

Більшість з цих переваг досягаються завдяки оптимізації під конкретну мову, вдалій архітектурі, що призвана посилювати особливості фреймворку та детальній документації.

Архітектура складається з двох частин, а саме з динамічного планувальника задач та колекцій для роботи з даними. Також, існує окремий модуль, який забезпечує взаємодію між різними пристроями.

2.2.2 Динамічний планувальник задач

Динамічний планувальник працює на основі графів задач. Структури даних Dask формують цей граф, у якому кожний вузол є певною функцією, а ребра між вузлами складаються з певних об’єктів Python та створюються як результат обчислень від попереднього вузла, водночас являючись вхідними даними для іншого вузла.

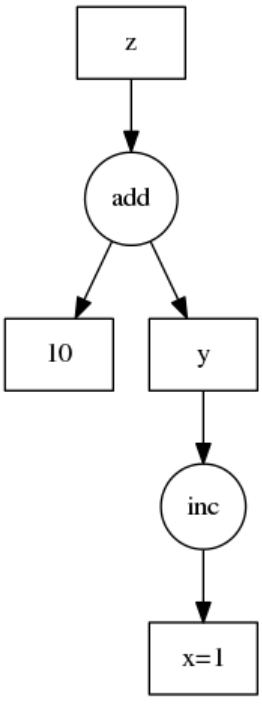

Рисунок 2.2.1 – Приклад побудови графа Dask [36]

За виконання цього графа відповідає планувальник. Планувальники бувають двох типів: одиничний та розподілений. Одиничний використовується під час локальної роботи на одному пристрої та використовує одиничний процес або пул потоків. На відміну від нього, розподілений більш складний, але дозволяє працювати з декількома пристроями, об'єднаними у кластер, одночасно.

Завдяки тому, що граф подається за допомогою структури словника Python, він може втілювати різні парадигми обчислень, такі як: приголомшлива паралельність, MapReduce, повне планування задач.

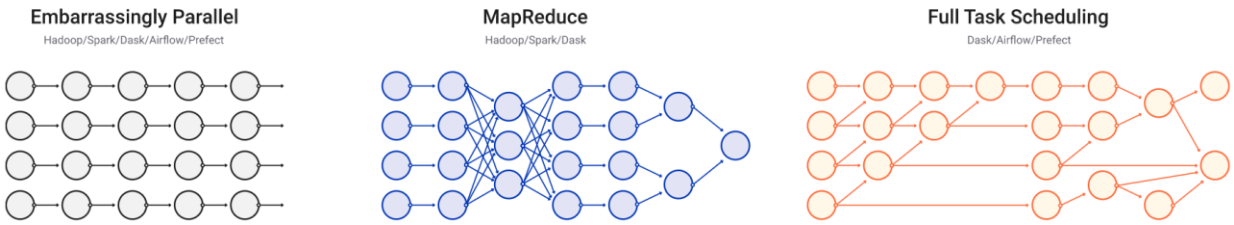

Рисунок 2.2.2 – Ілюстрація можливих парадигм обчислень у Dask [37]

2.2.3 Колекції Dask

Dask містить декілька колекцій, які використовуються для різних задач та зазвичай пов'язані з певною широко вживаною бібліотекою. Основними є:

- Array
- Bag
- DataFrame

Array імплементує інтерфейс відомої бібліотеки numpy, використовуючи алгоритми з блокуванням та розбиваючи масиви на декілька менших за розміром, що дозволяє проводити обчислення з масивами, що більше за розміром, ніж оперативна пам'ять пристрою.

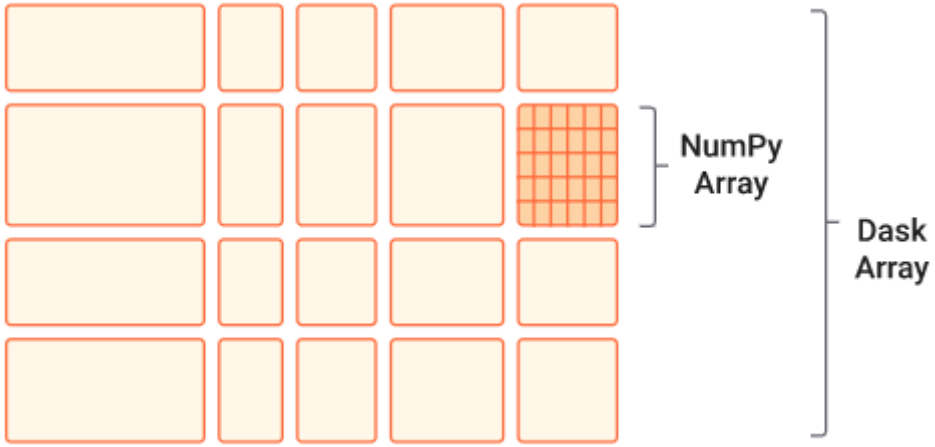

Рисунок 2.2.3 – Архітектура колекції Dask Array [38]

Bag імплементує такі операції над даними як map, filter, groupby та fold. Ця колекція основана на ітераторах та часто використовується для роботи з неструктурованими даними.

DataFrame базується на однойменній колекції з бібліотеки Pandas, яка використовується для роботи з табличними даними. На відміну від класичного, колекція здатна обробляти дані за розміром більші, ніж оперативна пам’ять. Це можливо завдяки розбиттю даних на частини, кожна з яких обробляється як Pandas DataFrame на окремому пристрої.

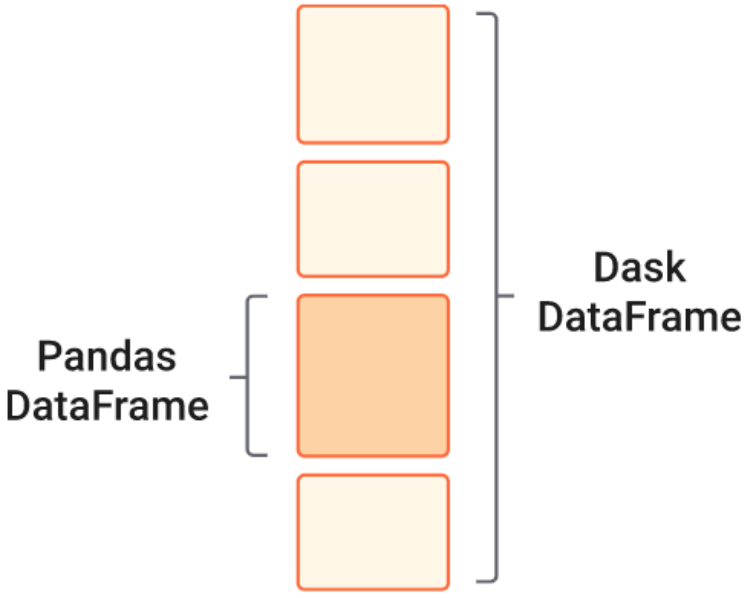

Рисунок 2.2.3 – Архітектура колекції Dask DataFrame [39]

2.2.4 Dask distributed

Dask distributed це окремий модуль, який забезпечує взаємодію між пристроями у кластері та налаштування кластера. До нього входить доступ до асинхронного API, інструменти для діагностики стану кластера та планувальник задач. Dask пропонує багато способів для розгортки, таких як розгортка з командного рядка, розгортка через ssh, розгортка за допомогою Docker, розгортка за допомогою Kubernetes та розгортка у хмарі.

2.3 Flink

2.3.1 Основні відомості

Apache Flink – фреймворк для вирішення задач розподіленого обчислення, основною перевагою якого вважається здатність інтегруватись в усі інші популярні середовища для кластерних обчислень. Інші особливості фреймворку призвані підсилити його універсальність.

2.3.2 Архітектура

Інтеграція з іншими системами досягається завдяки архітектурі Flink (рис. 2.3.1). Цей фреймворк має окремі конектори для взаємодії з обраними середовищами та обирає необхідний автоматично в залежності від менеджера ресурсів, з яким взаємодіє. Вся комунікація відбувається через універсальне REST API, що полегшує інтеграцію. Також, Flink має можливість працювати автономно, без взаємодії з іншими середовищами.

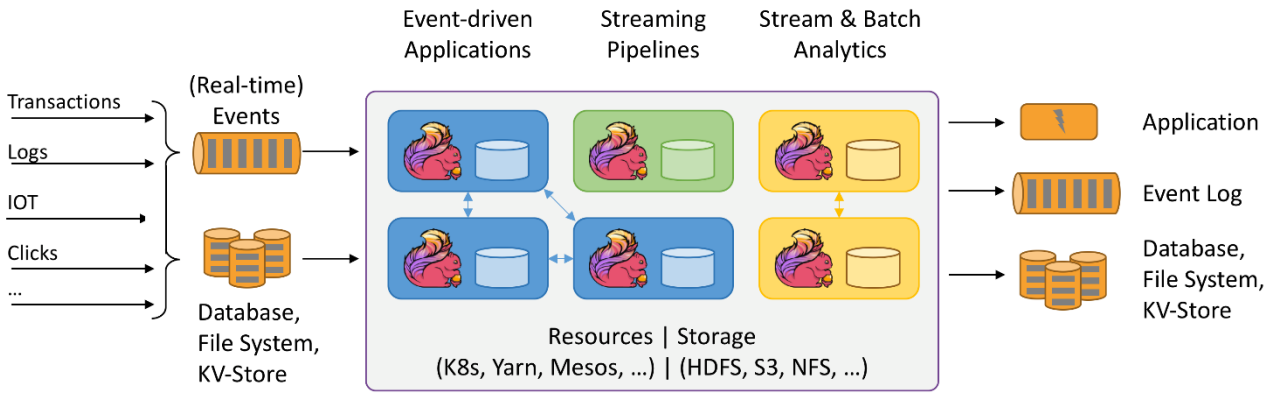

Рисунок 2.3.1 – Архітектура Flink [40]

Flink складається з двох частин, які забезпечують йог роботу: JobManager та TaskManager.

JobManager у свою чергу складається з менеджера ресурсів, що відповідає за розподіл задач та моніторинг їх виконання, диспетчера, що реалізує REST інтерфейс для взаємодії з іншими системами та з майстра задач, який відповідає за побудову графа виконання [41]. JobManager може існувати у єдиному екземплярі на кластер, але їх може бути й декілька для забезпечення відмовостійкості [42]. У випадку, якщо їх декілька, один бере на себе роль лідера, а інші стають залежними від нього.

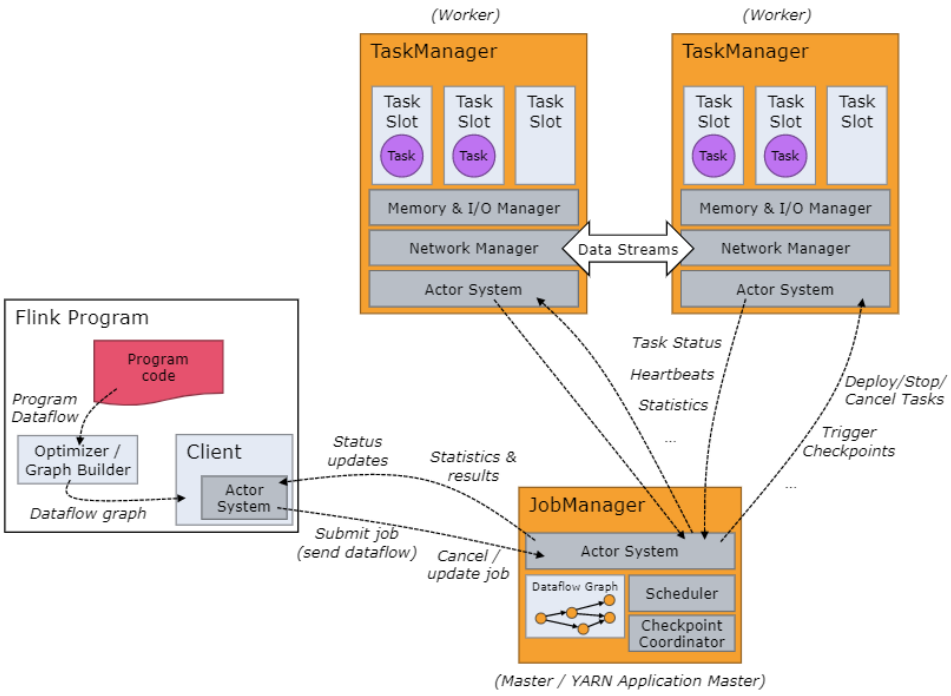

Рисунок 2.3.2 – Взаємодія JobManager з іншими компонентами [43]

TaskManager виконує задачі, обробляючі дані, що надходять. Кожен екземпляр цього менеджеру має певну кількість слотів задач, які говорять про максимальну кількість можливих одночасних процесів. Кожний процес виконується у окремому потоці, але у слоті може бути більш ніж одна задача, що дозволяє краще розподілити ресурси за допомогою групування (рис. 2.3.3) легких задач у одному слоті.

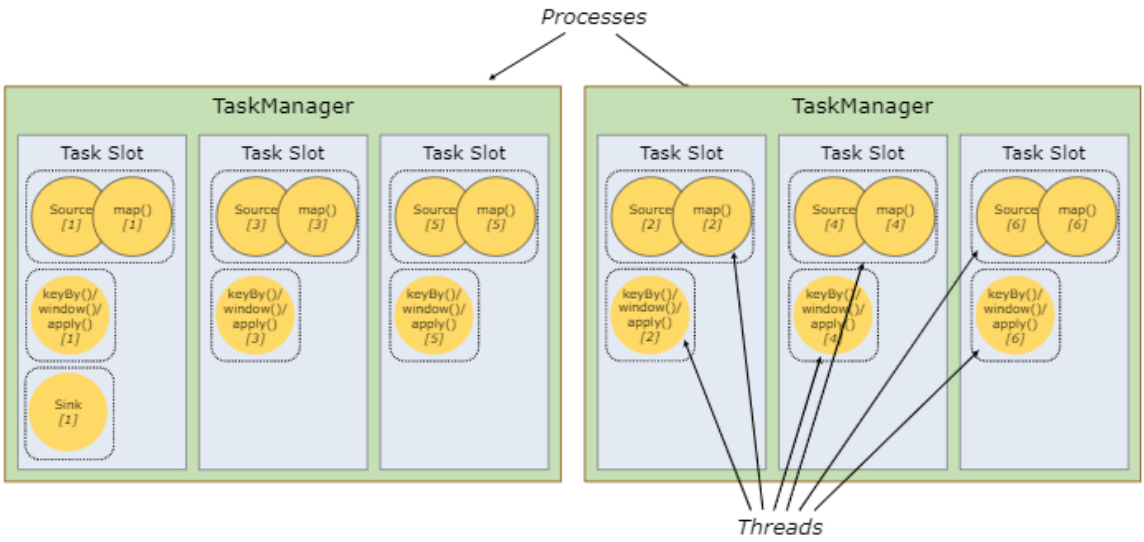

Рисунок 2.3.3 – Приклад групування задач у слотах TaskManager [44]

2.3.3 Робота з даними

Flink має можливість працювати з різними типами потоків даних. Ця можливість сильно посилює універсальність фреймворку. Існує два основних типів потоку: обмежений та необмежений. Обмежений дозволяє завантажити усі дані, що необхідно обробити до початку обробки. Необмежений потік на відміну від нього не має визначеного закінчення даних та обробляє дані по мірі їх надходження.

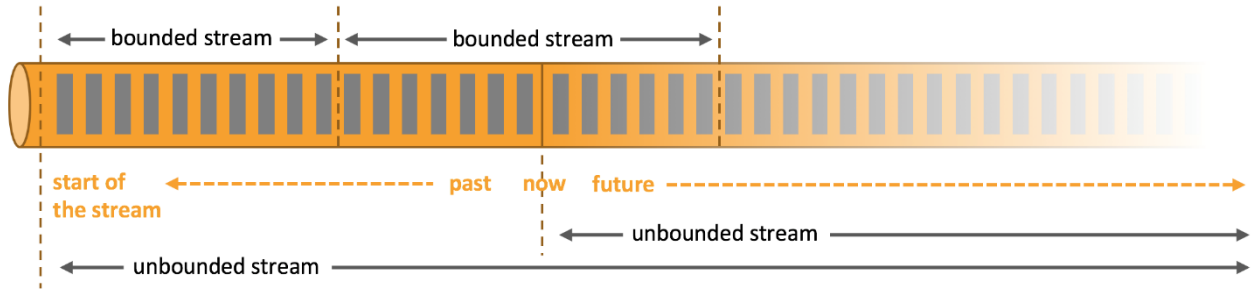

Рисунок 2.3.4 – Ілюстрація відмінностей обмеженого та необмеженого потоку [45]

З усіма отриманими даними Flink дозволяє працювати за допомогою абстракцій різного рівня.

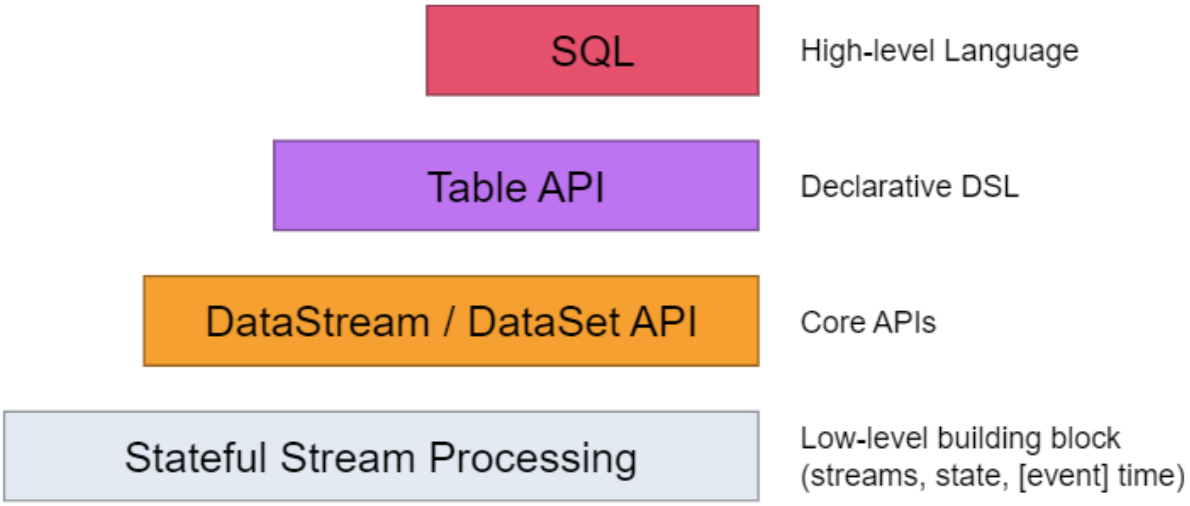

Рисунок 2.3.5 – Абстракції над даними у Flink [46]

3 РЕАЛІЗАЦІЯ ОБРАНИХ ЗАДАЧ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

3.1 Використані інструменти та технології

Для реалізації була обрана мова Python, тому що усі фреймворки підтримують її та це надає можливість провести порівняння зі сторони синтаксису, що використовується у різних фреймворках для вирішення однакової задачі.

Усі задачі були виконані на пристрої під керуванням ОС Windows з наступними характеристиками заліза:

- Процесор Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.70 GHz (2 фізичних та 4 логічних ядра)
- Оперативна пам’ять: 16Gb

Для розгортки фреймворків було використано Docker Compose [47], який дозволяє автоматизувати розгортку у відокремлених контейнерах. Завдяки цьому можна обмежити ресурси, що використовуються фреймворками та емулювати створення кластеру на декількох пристроях. Відповідно до фізичних обмежень пристрою, що використовується, під час роботи були створені кластери з кількістю виконуючих вузлів від одного до чотирьох.

У якості файлової системи була використана локальна система пристрою, на якому проводилась робота. Для вдалого зчитування файлу з локальної файлової системи усі виконуючі вузли мають мати доступ до цільового файлу. У даній роботі це було реалізовано за допомогою прикріплення розділу з файлами до сервісу у docker compose. Таким чином, знижуються до мінімуму накладні витрати на читання файлів, які могли б з'явитися у випадку використання розподіленої файлової системи на кшталт HDFS.

Для проведення замірів часу та моніторінгу виконання задач були використані вбудовані можливості фреймворків, які надаються у вигляді веб інтерфейсів для спостереження за роботою кластеру.

Під час розгортки виконуючі вузли були обмежені у використанні оперативної пам’яті до одного гігабайта та одного ядра процесору для забезпечення однакової кількості ресурсів для рівнозначного подальшого порівняння у швидкості.

3.2 Використані дані

Вхідні дані для обраних задач були згенеровані випадковим чином за допомогою скриптів на мові Python (додаток А, Б, В). Для кожної задачі було згенеровано декілька файлів різного розміру. Об’єми створених файлів дорівнюють 15 мегабайтів, 150 мегабайтів та 1.5 гігабайти. Це дозволяє простежити як змінюється швидкість обробки файлів в залежності від розмірів та здатність фреймворку обробити дані, що своїм розміром перевищують оперативну пам’ять виконуючого вузла.

Для задачі з сортування масиву було створено файли, що складаються з рядків чисел зі знаком довжиною чотири байти (рис. 3.2.1).

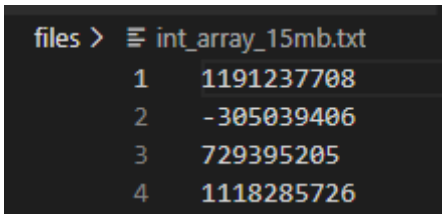

Рисунок 3.2.1 – Приклад вмісту файлу для задачі з сортування масиву

Для задачі підрахунку кількості слів було використано словник англійських слів, що складається з 61569 загальновживаних слів з маленьких літер без знаків пунктуації. Кожне слово розташоване на окремому рядку (рис. 3.2.2).

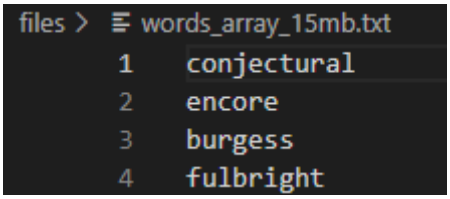

Рисунок 3.2.2 – Приклад вмісту файлу для задачі з підрахунку кількості слів

Для задачі обробки логів були створені логи наступного формату «Рівень логування|Час логування|Пріоритет|Повідомлення» (рис.3.2.3). Рівень логування обирається випадковим чином з «ERROR», «WARNING», «INFO» та «DEBUG». Час випадковий у проміжку від 25 грудня 2018 року до 24 лютого 2022 року. Пріоритет зазначається лише для логів з рівнем «ERROR» та може приймати значення 1, 2, 3, 4, 5 або бути відсутнім. Саме середнє значення пріоритету потрібно знайти під час виконання цієї задачі. Повідомлення містить випадкове англійське слово.

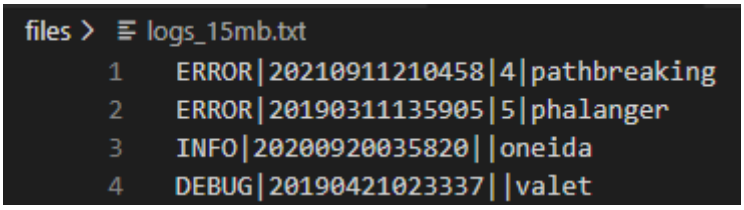

Рисунок 3.2.3 – Приклад вмісту файлу для задачі з обчислення середнього значення з логів

3.3 Spark

3.3.1 Особливості розгортки та розробки

Для розгортки кластера у докері Spark використовує один образ, але з різними налаштуваннями змін середовища. Так сервіси працівники створюються з налаштуванням SPARK_MODE=worker та посилаються на сервіс майстра, який створений з налаштуванням SPARK_MODE=master. Є можливість зазначити, скільки пам’яті та яку кількість ядер потрібно виділити кожному вузлу.

Незважаючи на популярність використання з різними мовами програмування, образи не містять встановлених конекторів для мови Python, хоча сама мова присутня.

3.3.2 Система моніторінгу

Spark надає веб інтерфейс для моніторінгу, що розгортається на майстер сервісі. Він містить основні відомості про кластер та список існуючих та виконаних задач (рис. 3.3.1). Додаткової інформації про порядок виконання задачі або про її розбиття на кроки не має, а з керування задачами є можливість лише аварійно закінчити виконання.

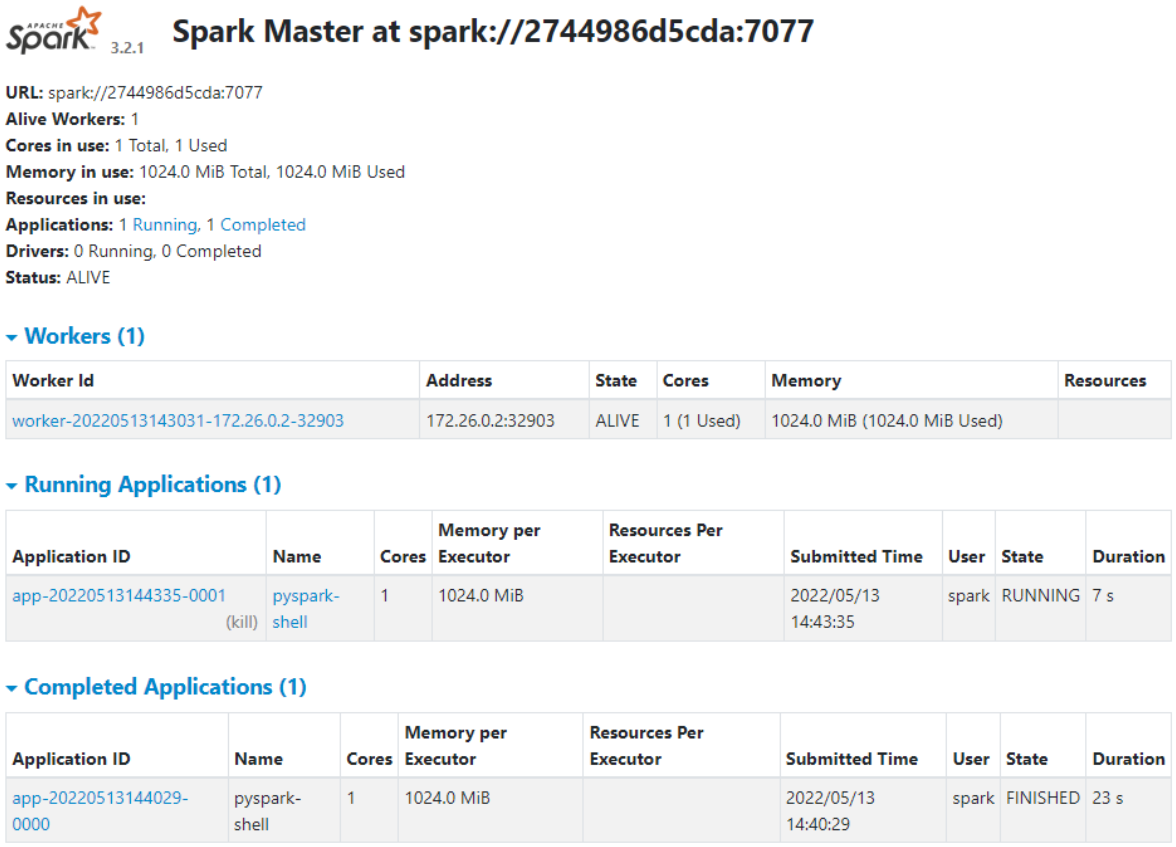

Рисунок 3.3.1 – Веб інтерфейс Spark

3.3.4 Документація

Spark має гарно прописану документацію, яку можна знайти на офіційному сайті. На жаль, більшість документації стосується мови Java, яка є основною для цього фреймворку. Також, документація для розгортки кластера переважно стосується запуску додатків на Java, що призводить до необхідності модифікувати цей процес для інших мов. Незважаючи на це, для програмування на інших мовах особливих перешкод немає, адже інтерфейси ідентичні для усіх

варіантів, а через поширеність існує велика кількість неофіційної документації та порад, створених користувачами.

3.3.5 Аналіз отриманих результатів

Задача з сортування масиву була виконана двома різними способами. З використанням DataFrame (додаток Г) та RDD (додаток Ґ). Ці структури значно відрізняються одна від іншої як у застосуванні, так і за результатами. RDD найпростіша колекція Spark, що представляє масив, а DataFrame складніша структура, що надає змогу працювати з табличними даними, використовуючи sql операції та має значну оптимізацію.

З результатів RDD (таблиця 3.3.1) бачимо, що при обробці даних невеликого розміру швидкість навіть погіршується через необхідність комунікації між робочими вузлами. А найбільший приріст швидкості можемо помітити при роботі з великими об’ємами даних. Особливо сильна різниця між виконанням на 1 та 2 вузлах для 1.5 гігабайтного файлу, адже пристрій має два фізичних ядра.

Кількість вузлів / розмір файлів	15МБ	150МБ	1.5Г
1	23	156	1200
2	21	108	720
3	24	90	660
4	28	100	600

Таблиця 3.3.1 – Результати швидкості Spark для задачі з сортування масиву з використанням RDD (с.)

Порівнюючи ці результати з результатами DataFrame (таблиця 3.3.2) можемо побачити, що загальна тенденція така ж сама, але ключова відмінність полягає у обробці великого файлу. DataFrame робить це набагато швидше. Це

досягається завдяки значної оптимізації та лінивим обчисленням. Також бачимо, що на відміну від RDD приріст швидкості з’явився лише для найбільшого файлу, а для файлу 150МБ можемо спостерігати сповільнення, як і для файлу 15МБ

Кількість вузлів / розмір файлів	15МБ	150МБ	1.5Г
1	19	27	102
2	21	28	60
3	33	38	60
4	35	45	70

Таблиця 3.3.2 – Результати швидкості Spark для задачі з сортування масиву з використанням DataFrame (с.)

Для виконання обчислень за допомогою DataFrame було використано метод OrderBy, але існує ще один метод, який називається sortWithPartitions. Згідно документації він працює швидше, але не гарантує повне сортування, через відсутність синхронізації між розділами даних, тому його не було використано під час роботи.

Для підрахунку кількості слів у файлі також було використано два різних метода. Перший, з використанням MapReduce RDD колекції (додаток Д). Бачимо (таблиця 3.3.3), що MapReduce демонструє таку ж поведінку, як і сортування та показує результати починаючи з 150МБ з найбільшим прискоренням для файлу 1.5Г при переході з 1 на 2 вузли.

Кількість вузлів / розмір файлів	15МБ	150МБ	1.5Г
1	17	90	840
2	23	84	594
3	26	72	486
4	36	70	470

Таблиця 3.3.3 – Результати швидкості Spark для задачі з підрахунку кількості слів з використанням RDD (с.)

Для реалізації на DataFrame була використана функція групування, з подальшим підрахунком кількості входжень (додаток Е). Можемо побачити, що результати (таблиця 3.3.4) ідентичні до попередньої задачі.

Кількість вузлів / розмір файлів	15МБ	150МБ	1.5Г
1	15	25	120
2	26	30	84
3	30	38	84
4	39	43	90

Таблиця 3.3.4 – Результати швидкості Spark для задачі з підрахунку кількості слів з використанням DataFrame (с.)

Для підрахунку середнього значення з рядків логів було використано лише DataFrame (додаток Є), адже RDD не дуже підходить для обробки табличних даних. Проте DataFrame навпаки, створений саме для цих задач та здатен показати найкращі результати (таблиця 3.3.5). На відміну від попередніх результатів, тенденція прискорення для обробки великого файлу зберігається не лише при переході з 1 вузла на 2, але й для подальшого збільшення кількості

вузлів, що свідчить про ефективність вирішення таких задач з використанням DataFrame та гарну оптимізацію.

Кількість вузлів / розмір файлів	15МБ	150МБ	1.5Г
1	17	32	162
2	22	33	102
3	28	33	96
4	36	39	90

Таблиця 3.3.5 – Результати швидкості Spark для задачі з обчислення середнього значення з логів (с.)

3.3.3 Висновки

Spark є вдалим фреймворком для роботи з великими даними, але його не рекомендується використовувати для обробки малих об’ємів, адже це лише сповільнить обробку. Spark легко розгортається та надає веб інтерфейс, для спостереження за основними даними кластера. Для вирішення більшості задач можна використати різні методи, що надає користувачу гнучкість у розробці. Робота з даними є гарно оптимізованою та більшість інтерфейсів є знайомими для користувачів, адже схожі на роботу з sql. Це значно спрощує опанування фреймворку та дозволяє пришвидшити процес розробки.

3.4 Dask

3.4.1 Особливості розгортки та розробки

Розгортка Dask за допомогою Docker є простою та зрозумілою. Офіційна документація пропонує усі необхідні файли та детально описує кроки, що потрібно виконати. Dask може бути розгорнутим за допомогою єдиного образу, що запускається у різних режимах: dask-scheduler для планувальника задач та dask-worker для виконуючого вузла. Усі параметри передаються як аргументи команди запуску та можуть мають змогу сконфігурувати вузли необхідним

чином. Також, існує окремий образ, що дозволяє розгорнути сервіс для розробки. Він пропонує jupiter notebook, що вже налаштований для роботи з Dask, що дозволяє одразу перейти до розробки.

3.4.2 Система моніторінгу

Веб інтерфейс Dask (рис.3.4.1) надає не лише основну інформацію про кластер, але й може отримувати багато детальної інформації про усі компоненти кластера.

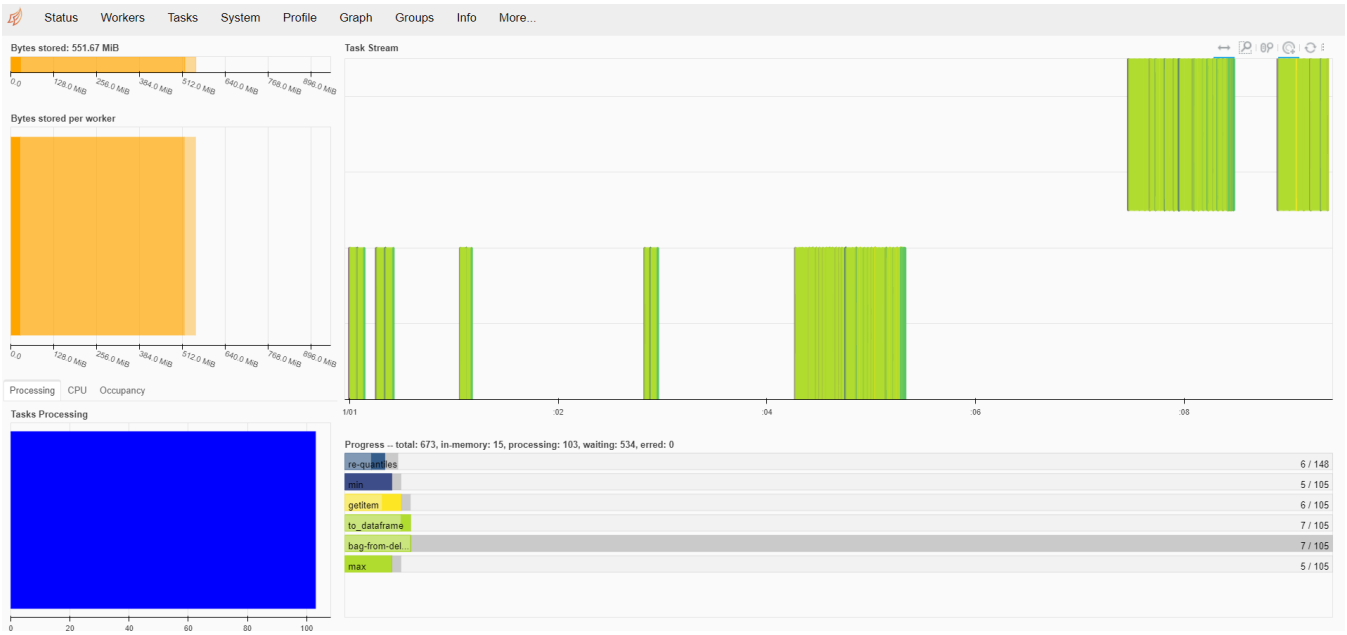

Рисунок 3.4.1 – Веб інтерфейс Dask

Так, є можливість дізнатися стан кожної задачі, вузла, подивитися на побудований граф для вирішення задачі (рис. 3.4.2), навантаження та детальний час виконання кожної операції.

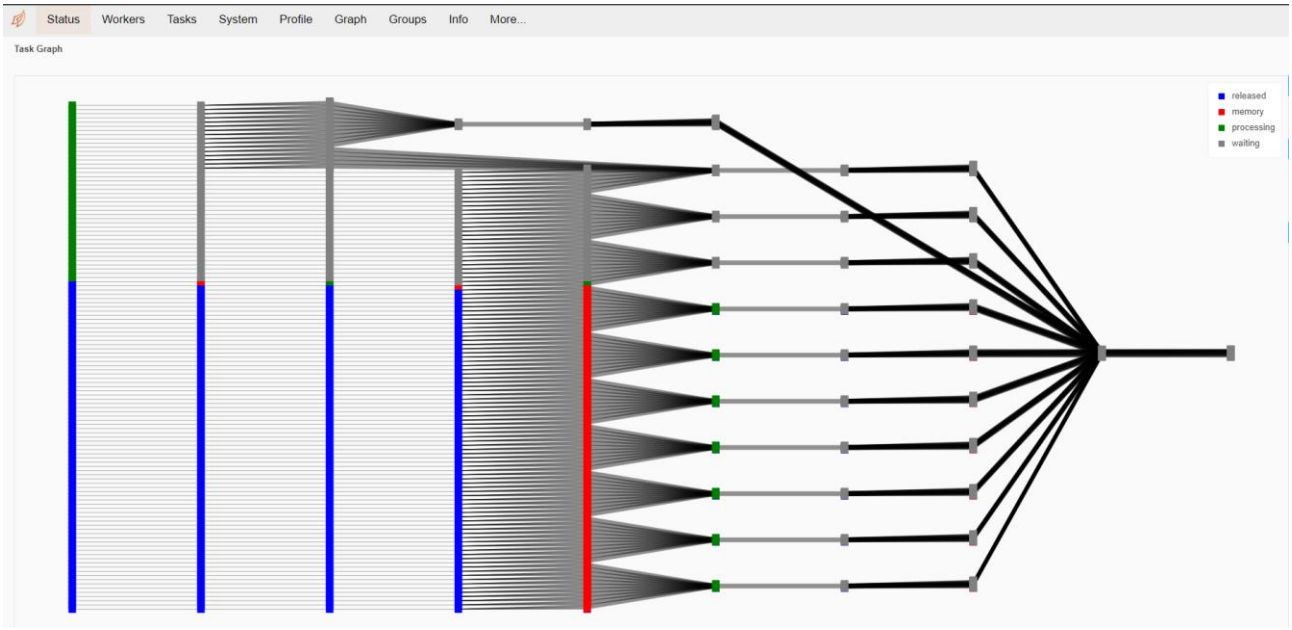

Рисунок 3.4.2 – Граф виконання для задачі у веб інтерфейсі Dask

Це надає змогу значно полегшити розробку, розуміючи вузькі місця програми та маючи усю інформацію про кластер.

3.4.4 Документація

Dask має найкращу документацію. Офіційна сторінка пропонує користувачу допомогу як у розгортці на різних системах, так і має приклади виконання деяких задач. API гарно задокументований та має відсилки на класичні бібліотеки Python, що полегшує процес розробки. На жаль, незважаючи на обширну офіційну документацію, фреймворку не вистачає неофіційних керівництв та відповідей на питання, що змушує розробників витрачати час на вирішення дрібних питань, читаючи документацію, що трохи погіршує досвід користування системою.

3.4.5 Аналіз отриманих результатів

На жаль, Dask не має вбудованої функції для сортування, та єдиним способом відсортувати дані є встановити індекс (додаток Ж), що однак є дуже важкою і не оптимізованою операцію. Тому, для цієї задачі Dask показав найгірші часові результати (таблиця 3.4.1). Хоч для маленького файлу результат не змінюється, для середнього він значно погіршується з збільшенням кількості

вузлів, а для найбільшого файлу, незважаючи на значний приріст, результати все одного набагато гірші, ніж у інших фреймворках

Кількість вузлів / розмір файлів	15МБ	150МБ	1.5Г
1	8	28	1470
2	7	42	670
3	8	65	646
4	7	70	640

Таблиця 3.4.1 – Результати швидкості Dask для задачі з сортування масиву (с.)

Для підрахунку кількості слів було використано вбудований метод для знаходження частоти входжень (додаток З). Це дало гарні результати (таблиця 3.4.2) для 1 вузла, але при масштабуванні можемо побачити погіршення результатів для усіх файлів, на відміну від інших фреймворків.

Кількість вузлів / розмір файлів	15МБ	150МБ	1.5Г
1	1	11	106
2	7	49	120
3	5	37	125
4	7	40	118

Таблиця 3.4.2 – Результати швидкості Dask для задачі з підрахунку кількості слів (с.)

Для задачі логування було використано перетворення табличних даних до масиву з подальшим знаходженням середнього (додаток І). Результати (таблиця 3.4.3) показують гарну швидкість для файлів малого розміру, але поступаються

фреймворку Spark, коли йде мова про великі файли, хоч і досягає значного прискорення.

Кількість вузлів / розмір файлів	15МБ	150МБ	1.5Г
1	2	19	245
2	2	12	168
3	3	13	160
4	3	11	156

Таблиця 3.4.3 – Результати швидкості Dask для задачі з обчислення середнього значення з логів (с.)

3.4.6 Висновки

Dask є фреймворком з найкращим офіційним супроводом. Їм легко користуватись, через схожість інтерфейсів до знайомих користувачам мови Python та повністю задокументований API. Веб інтерфейс дає багато можливостей для керування кластером та значно покращує досвід використання системи. Незважаючи на ці переваги, існують і дуже важливі недоліки, такі як відсутність конекторів для інших мов та найменша ефективність, серед інших фреймворків, що значно звужує можливості фреймворку.

3.5 Flink

3.5.1 Особливості розгортки та розробки

Flink як і попередні фреймворки має єдиний образ для розгортки, а усі налаштування кластеру зберігаються у одному файлі. У Flink найскладніша система з налаштувань пам’яті, що може ускладнити розгортку у разі, якщо необхідно встановити значення, відміні від тих, що йдуть за замовченням.

На відміну від інших фреймворків, для запуску додатку Flink потрібно виконати команду «flink run» з зазначенням мови застосунку.

3.5.2 Система моніторінгу

Flink має зручний інтерфейс для відстеження стану кластеру (рис. 3.5.1). Є доступ як до основної інформації, так і до детальної про задачі та про виконуючи вузли. Хоч користувачу і не надається можливість переглянути граф виконання, але для кожної задачі присутній узагальнений порядок її виконання та розбиття на дрібніші задачі.

Окремо хочеться відзначити можливість додати задачу з веб інтерфейсу. Для цього лише потрібно перетягнути файл, що потрібно виконати на відповідну вкладку. На жаль, ця можливість присутня лише для мови Java та файлів з розширенням .jar.

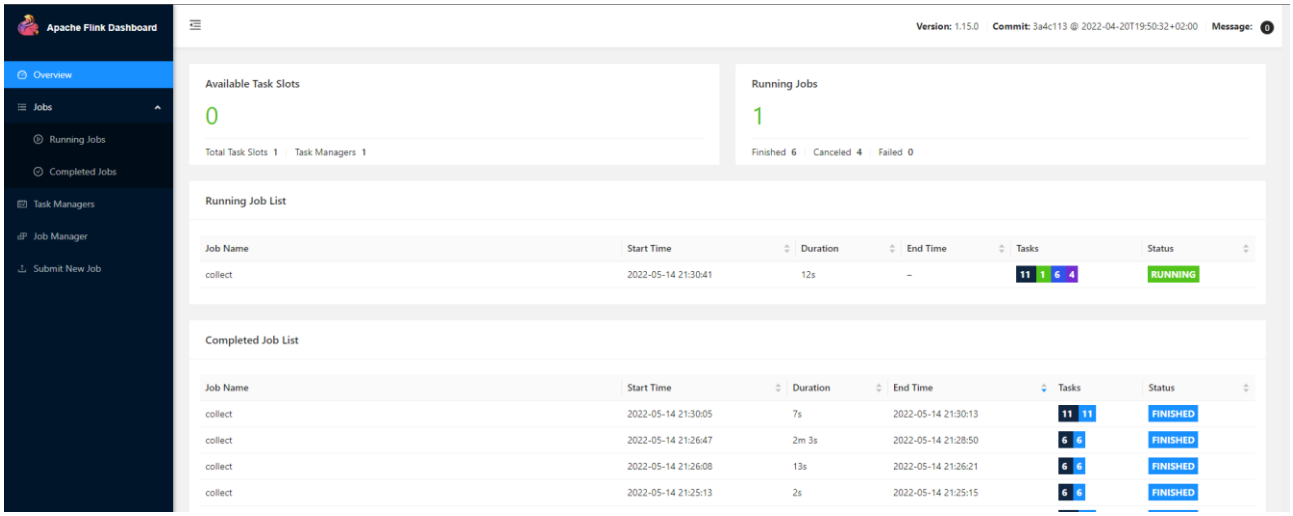

Рисунок 3.5.1 – Веб інтерфейс Flink

3.5.4 Документація

З поміж інших, Flink має найменш детальну та зрозумілу документацію. Більшість методів не є задокументованими, або задокументовані недостатньо. Документація неповна та в ній важко орієнтуватись. Це ускладнюється тим, що API Flink не є інтуїтивно зрозумілим та часто не має відповідників у класичних колекціях. Також, цей фреймворк є найменш поширеним та через це майже немає неофіційних порад та керівництв, що іноді призводить до необхідності шукати відповіді у вихідному коді навіть коли річ йде про відносно класичні задачі та проблеми.

3.5.5 Аналіз отриманих результатів

Для сортування у Flink використовується функція order by (додаток Ї). З результатів (таблиця 3.5.1) можна побачити, що хоч приріст швидкості й незначний, Flink впорався з задачею на одному рівні зі Spark, а для одного вузла навіть краще за нього. Так, Flink є абсолютним лідером у швидкості для обробки на 1 вузлі і не поступається Spark при їх збільшені

Кількість вузлів / розмір файлів	15МБ	150МБ	1.5Г
1	2	13	70
2	5	14	60
3	9	15	58
4	10	14	64

Таблиця 3.5.1 – Результати швидкості Flink для задачі з сортування масиву (с.)

У Flink не має можливості запровадити патерн MapReduce, тому задача з підрахунку кількості входжень була реалізована через агрегацію табличних значень (додаток Й). Для цієї задачі результати Flink (таблиця 3.5.2) значно гірше, ніж у інших фреймворків, що свідчить про нездатність Flink ефективно виконувати задачі цього типу.

Кількість вузлів / розмір файлів	15МБ	150МБ	1.5Г
1	7	53	403
2	8	40	373
3	10	38	366
4	12	44	364

Таблиця 3.5.2 – Результати швидкості Flink для задачі з підрахунку кількості слів (с.)

Незважаючи на те, що Flink гарно пристосований до роботи з таблицями та підходить для виконання цієї задачі, його результати (таблиця 3.5.3) виявились гіршими, ніж у інших системах. Можна дійти висновку, що агрегація у Flink (додаток К) реалізована менш вдало, ніж у Spark та Dask. Можемо бачити, що незважаючи на прийнятий результат для обробки на одному вузлі, Flink майже не демонструє прискорення під час збільшення їх кількості.

Кількість вузлів / розмір файлів	15МБ	150МБ	1.5Г
1	3	26	240
2	6	32	230
3	7	34	226
4	7	33	227

Таблиця 3.5.3 – Результати швидкості Flink для задачі з обчислення середнього значення з логів (с.)

3.5.6 Висновки

Flink виявився менш результативним для двох з трьох розглянутих задач. Також, Flink набагато важчий у розробці через брак документації. Ще, дуже важливим недоліком є незначне покращення при масштабуванні, що призводить нас до висновку, що Flink може вдало працювати лише у невеликих за розмірами кластерах. Незважаючи на це, для задачі з сортування фреймворк продемонстрував набагато кращі результати, ніж інші реалізації, що свідчить про можливість його ефективного використання для деяких типів задач. Не слід забувати, що Flink є найновішим з згаданих фреймворків та ще досі розвивається, а значить може згодом позбутися деяких з зазначених у роботі проблем.

3.6 Переваги і недоліки фреймворків

Під час роботи було проаналізовано результати отримані при реалізації деяких задач із застосуванням обраних фреймворків, що дає можливість сформулювати їх переваги та недоліки.

Так, Apache Spark, хоч і є найстарішим фреймворком з усіх, одночасно з цим є й найбільш вдалим для більшості задач. Він гарно масштабується, демонструє прийняті результати для усіх задач та має велику спільноту, що спрощує процес розробки. Одночасно з цим, він є гнучким та може бути налаштований під потреби певної задачі. З недоліків можна зазначити лише веб інтерфейс, що менш інформативний, ніж у інших системах та найгірші результати для найменших файлів.

Результати Dask для більшості задач є прийнятними хоч і поступаються Spark. Одночасно з цим, Dask має інтуїтивно зрозумілий API, що ефективно взаємодіє з основними засобами для роботи з великими даними у мові Python та найбільш розвинутий інтерфейс, що допомагає розробникам ефективно використовувати його можливості. Незважаючи на брак деякого функціоналу та середню швидкість, Dask залишається вдалим фреймворком для вирішення задач з розподіленого обчислення.

Flink є найбільш неоднозначним фреймворком з усіх. Його результати в більшості випадків виявились найгіршими, а масштабування несуттєвим, що значно зменшує його можливості. Також, майже відсутній супровід обтяжує розробку. Незважаючи на це, він добре проявив себе для файлів невеликого об’єму та може бути ефективним для деяких задач. Його інтерфейс також є дуже вдалим та надає користувачам найбільш можливостей. Хоч його використання і є ситуативним, має сенс обмежено використовувати його та спостерігати за подальшим розвитком.

3.7 Висновки

Відповідно до отриманих результатів та проведеного аналізу можна зробити висновки, що у загальному випадку, найкращим рішенням буде використання фреймворку Apache Spark. Dask може бути використано для покращення існуючої кодової бази та у випадку, коли для проекту є важливим детальний моніторинг за станом кластером та процесом виконання задач. Flink для більшості випадків виявився гіршим, ніж інші фреймворки, але все ще може бути використано для деяких задач та має потенціал для майбутнього розвитку.

ВИСНОВКИ

Відповідно до поставленої задачі, було обрано фреймворки для огляду та сформовано критерії для їх подальшого порівняння. Були визначені задачі, за допомогою яких було запропоновано продемонструвати особливості обраних систем. Під час роботи були запрограмовані вирішення обраних задач для кожного з фреймворків. Отримані під час виконання результати були ретельно проаналізовані та досліджені. На основі цих результатів було сформовано уявлення про існуючі переваги та недоліки фреймворків. Було зроблено висновки про використання фреймворків, базуючись на попередньому аналізі.

Проведена робота має певний потенціал для продовження та доповнення. У роботі було проведено аналіз на основі класичних задач для трьох фреймворків. Існує як можливість вдосконалити роботу за рахунок проведення поглибленого аналізу, проводячи дослідження на інших задачах, що будуть включати в себе взаємодію з іншими системами, розподілене файлове сховище та обробку необмежених потоків даних, так і можливість додати до порівняння інші фреймворки, які хоч і не є такими поширеними, але теж використовуються та можуть мати значні переваги у певних аспектах.

Список використаної літератури

1. Популярність пошуку у Google за фразою «data analytics» [Електронний ресурс] // Режим доступу: https://trends.google.com.ua/trends/explore?q=data%20analytics&date=all
2. Популярність пошуку у Google за фразою «analytics of big data» [Електронний ресурс] // Режим доступу: https://trends.google.com.ua/trends/explore?q=analytics%20of%20big%20data&date=all
3. History and Advantages of Hadoop MapReduce Programming [Електронний ресурс] // Режим доступу: https://mindmajix.com/mapreduce/history-and-advantages-of-hadoop-mapreduce-programming
4. Популярність пошуку у Google за фразою «MapReduce» [Електронний ресурс] // Режим доступу: https://trends.google.ru/trends/explore?date=all&q=%2Fm%2F05pp4x
5. Офіційний сайт мови програмування Julia [Електронний ресурс] // Режим доступу: https://julialang.org/
6. Офіційний сайт фреймворку Dask [Електронний ресурс] // Режим доступу: https://dask.org/
7. Офіційний сайт фреймворку Spark [Електронний ресурс] // Режим доступу: https://spark.apache.org/
8. Офіційний сайт фреймворку Dryad [Електронний ресурс] // Режим доступу: https://www.dryad.net/
9. Офіційний сайт фреймворку Ignite [Електронний ресурс] // Режим доступу: https://ignite.apache.org/
10. Офіційний сайт фреймворку Impala [Електронний ресурс] // Режим доступу: https://impala.apache.org/
11. Офіційний сайт фреймворку Drill [Електронний ресурс] // Режим доступу: https://drill.apache.org/

12. Офіційний сайт фреймворку Kudu [Електронний ресурс] // Режим доступу: https://kudu.apache.org/
13. High Performance Data Engineering Everywhere / Chathura Widanage, Niranda Perera, Vibhatha Abeykoon, Supun Kamburugamuve, Thejaka Amila Kanewala, Hasara Maithree, Pulasthi Wickramasinghe, Ahmet Uyar, Gurhan Gunduz, Geoffrey Fox // 2020. – ст. 1-9 - Режим доступу: https://arxiv.org/pdf/2007.09589.pdf
14. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025 / Statista Research Department // 2022. – ст. 1 - Режим доступу: https://www.statista.com/statistics/871513/worldwide-data-created/
15. Cost of server ownership: On-premises vs. IaaS [Електронний ресурс] // Режим доступу: https://www.sherweb.com/blog/cloud-server/total-cost-of-ownership-of-servers-iaas-vs-on-premise/
16. HADOOP ARCHITECTURE AND FAULT TOLERANCE BASED HADOOP CLUSTERS IN GEOGRAPHICALLY DISTRIBUTED DATA CENTER / T. Cowsalya, S.R. Mugunthan // 2015. – ст. 1-4 - Режим доступу: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1038.365&rep=rep1&type=pdf
17. Past, Present and Future of Hadoop: A Survey / Ameneh Zarei, Shahla Safari, Mahmood Ahmadi, Farhad Mardukhi // 2022. – ст. 1-4 - Режим доступу: https://arxiv.org/ftp/arxiv/papers/2202/2202.13293.pdf
18. HDFS Architecture [Електронний ресурс] // Режим доступу: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#:~:text=HDFS%20has%20a%20master%2Fslave,access%20to%20files%20by%20clients.
19. Офіційна документація JobTracker [Електронний ресурс] // Режим доступу: https://cwiki.apache.org/confluence/display/HADOOP2/JobTracker

20. Past, Present and Future of Hadoop: A Survey / Ameneh Zarei, Shahla Safari, Mahmood Ahmadi, Farhad Mardukhi // 2022. – ст. 10 - Режим доступу: https://arxiv.org/ftp/arxiv/papers/2202/2202.13293.pdf
21. Популярність пошуку у Google за фразою «MapReduce» [Електронний ресурс] // Режим доступу: https://trends.google.ru/trends/explore?date=all&q=MapReduce
22. Spark vs. Hadoop MapReduce: Which big data framework to choose [Електронний ресурс] // Режим доступу: https://www.scnsoft.com/blog/spark-vs-hadoop-mapreduce#MapReduce-tasks
23. What is MapReduce in Hadoop? Architecture | Example [Електронний ресурс] // Режим доступу: https://www.guru99.com/introduction-to-mapreduce.html
24. Apache Spark: гайд для новачків [Електронний ресурс] // Режим доступу: https://medium.com/nuances-of-programming/apache-spark-%D0%B3%D0%B0%D0%B9%D0%B4-%D0%B4%D0%BB%D1%8F-%D0%BD%D0%BE%D0%B2%D0%B8%D1%87%D0%BA%D0%BE%D0%B2-959145ef6167
25. Рейтинг мов програмування 2021 [Електронний ресурс] // Режим доступу: https://dou.ua/lenta/articles/language-rating-jan-2021/
26. How do Hadoop and Spark Stack Up? [Електронний ресурс] // Режим доступу: https://logz.io/blog/hadoop-vs-spark/#:~:text=relational%20data%20stores.-,Performance,Naive%20Bayes%20and%20k%2Dmeans
27. Офіційна документація SparkSQL [Електронний ресурс] // Режим доступу: https://spark.apache.org/sql/
28. Офіційна документація SparkML [Електронний ресурс] // Режим доступу: https://spark.apache.org/mllib/
29. Офіційна документація Spark Streaming [Електронний ресурс] // Режим доступу: https://spark.apache.org/docs/latest/streaming-programming-guide.html

30. Офіційна документація Spark GraphX [Електронний ресурс] // Режим доступу: https://spark.apache.org/docs/latest/graphx-programming-guide.html
31. Auto Tuning of Hadoop and Spark parameters / Mrs. Tanuja Patanshetti, Mr. Ashish Anil Pawar, Ms. Disha Patel, Mr. Sanket Thakare // 2021. – ст. 1-12 - Режим доступу: https://arxiv.org/ftp/arxiv/papers/2111/2111.02604.pdf
32. Офіційний сайт Pandas [Електронний ресурс] // Режим доступу: https://pandas.pydata.org/
33. Офіційний сайт Scilkit-learn [Електронний ресурс] // Режим доступу: https://scikit-learn.org/
34. Офіційний сайт numpy [Електронний ресурс] // Режим доступу: https://numpy.org/
35. Офіційна документація Dask [Електронний ресурс] // Режим доступу: https://docs.dask.org/en/stable/
36. Офіційна документація Dask, приклад графу виконання [Електронний ресурс] // Режим доступу: https://docs.dask.org/en/stable/_images/dask-simple.png
37. Офіційна документація Dask, можливі парадигми [Електронний ресурс] // Режим доступу: https://docs.dask.org/en/stable/_images/map-reduce-task-scheduling.svg
38. Офіційна документація Dask, Dask Array [Електронний ресурс] // Режим доступу: https://docs.dask.org/en/stable/_images/dask-array.svg
39. Офіційна документація Dask, Dask DataFrame [Електронний ресурс] // Режим доступу: https://docs.dask.org/en/latest/_images/dask-dataframe.svg
40. Офіційний сайт Fkink [Електронний ресурс] // Режим доступу: https://flink.apache.org/
41. Flink, JobManager Data Structures [Електронний ресурс] // Режим доступу: https://nightlies.apache.org/flink/flink-docs-master/docs/internals/job_scheduling/#jobmanager-data-structures

42. Flink, High Availability [Електронний ресурс] // Режим доступу: https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/ha/overview/
43. Anatomy of a Flink Cluster [Електронний ресурс] // Режим доступу: https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture/#anatomy-of-a-flink-cluster
44. Task Slots and Resources [Електронний ресурс] // Режим доступу: https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture/#task-slots-and-resources
45. Flink, Task Slots and Resources [Електронний ресурс] // Режим доступу: https://flink.apache.org/flink-architecture.html
46. Flink’s APIs [Електронний ресурс] // Режим доступу: https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/concepts/overview/#flinks-apis
47. Офіційна документація Docker Compose [Електронний ресурс] // Режим доступу: https://docs.docker.com/compose/

Додаток А

(довідниковий)

Генерація файлу для задачі сортування масиву

```python
import os
import random
from tqdm import tqdm

random.seed(1000)
arr_len = 132000000
batch = 10000
low = -2147483648
high = 2147483647
r = range(low, high)

filename = 'int_array.txt'

with open(filename, 'w') as f:
    for i in tqdm(range(arr_len//batch)):
        s = '\n'.join(map(str, random.sample(r, batch)))
        f.write(s + '\n')

print(f'{os.path.getsize(filename)//1024/1024} MB')
```

Додаток Б

(довідниковий)

Генерація файлу для задачі з підрахунку кількості слів

```
import os
import random
from english_words import english_words_lower_alpha_set
from tqdm import tqdm

random.seed(1000)
arr_len = 17500000
batch = 10000
filename = 'words_array.txt'

with open(filename, 'w') as f:
    for i in tqdm(range(arr_len//batch)):
        b = '\n'.join(random.choices(list(english_words_lower_alpha_set), k =
batch))
        f.write(b + '\n')

print(f'{os.path.getsize(filename)//1024/1024} MB')
```

Додаток В

(довідниковий)

Генерація файлу для задачі з обчислення середнього значення з логів

```
import os
import random
from tqdm import tqdm
from english_words import english_words_lower_alpha_set
from datetime import datetime

timestamp_start = 1545730073
timestamp_stop = 1645730073

random.seed(1000)
arr_len = 52000000
batch = 10000
filename = 'logsmb.txt'

levels = ['INFO', 'DEBUG', 'ERROR']
errors_priority = [None, '1', '2', '3', '4', '5']

with open(filename, 'w') as f:
    for i in tqdm(range(arr_len//batch)):
        levels = random.choices(levels, k=batch)
        datetimes = [datetime.fromtimestamp(x).strftime('%Y%m%d%H%M%S') for x in
random.sample(range(timestamp_start, timestamp_stop), batch)]
        messages = random.choices(list(english_words_lower_alpha_set), k = batch)
        errors = random.choices(errors_priority, k=batch)
        errors = [x if y == 'ERROR' and x else '' for x,y in zip(errors,levels)]

        log = zip(levels, datetimes, errors, messages)
        log = map(lambda x: '|'.join(x), log)
        b = '\n'.join(log)
        f.write(b + '\n')

print(f'{os.path.getsize(filename)//1024/1024} MB')
```

Додаток Г

(довідниковий)

Вирішення задачі сортування масиву з використанням DataFrame Spark

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

filename = 'file:///files/int_array_150mb.txt'
spark = SparkSession.builder.master("spark://spark:7077").getOrCreate()
textFile = spark.read.text(filename)
textFile = textFile.withColumn("value",col("value").cast("int"))
textFile = textFile.sortWithinPartitions("value").first()
```

Додаток Ґ

(довідниковий)

Вирішення задачі сортування масиву з використанням RDD Spark

```
from pyspark.sql import SparkSession

filename = 'file:///files/int_array_1500mb.txt'
spark = SparkSession.builder.master("spark://spark:7077").getOrCreate()
textFile = spark.sparkContext.textFile(filename).map(int)
textFile = textFile.sortBy(lambda x: x)
```

Додаток Д

(довідниковий)

Вирішення задачі з підрахунку кількості слів з використанням RDD Spark

```
from pyspark.sql import SparkSession


filename = 'file:///files/words_array_1500mb.txt'
spark = SparkSession.builder.master("spark://spark:7077").getOrCreate()
textFile = spark.sparkContext.textFile(filename)
textFile = textFile.map(lambda word: (word, 1)).reduceByKey(lambda a,b:a
+b).first()
```

Додаток Е

(довідниковий)

Вирішення задачі з підрахунку кількості слів з використанням DataFrame Spark

```
from pyspark.sql import SparkSession

filename = 'file:///files/words_array_1500mb.txt'
spark = SparkSession.builder.master("spark://spark:7077").getOrCreate()
textFile = spark.read.text(filename)
textFile = textFile.groupBy("value").count().first()
```

Додаток Є
(довідниковий)
Вирішення задачі з обчислення середнього значення з логів з використанням Spark

```
from pyspark.sql import SparkSession
from pyspark.sql.types import IntegerType

from pyspark.sql.types import StructField, StructType, StringType,LongType


filename = 'file:///files/logs_1500mb.txt'
spark = SparkSession.builder.master("spark://spark:7077").getOrCreate()

custom_schema = StructType([
    StructField("level", StringType(), False),
    StructField("datetime", StringType(), False),
    StructField("priotity", IntegerType(), True),
    StructField("message", StringType(), False),
])
textFile = spark.read.csv(filename, sep='|', schema=custom_schema)
textFile = textFile.dropna().groupBy().avg('priotity').first()
```

Додаток Ж

(довідниковий)

Вирішення задачі сортування масиву з використанням Dask

```
from dask.distributed import Client
client = Client()
import dask.bag as b
import dask.array as a
from time import time
filename = 'file:///files/int_array_1500mb.txt'
d = b.read_text(filename, blocksize='15MB').map(lambda x: int(x[:-
1])).to_dataframe()
r = d.set_index(0).loc[0].compute()
```

Додаток З

(довідниковий)

Вирішення задачі з підрахунку кількості слів з використанням Dask

```
from dask.distributed import Client
client = Client()
import dask.bag as b
import dask.array as a
from time import time
filename = 'file:///files/words_array_1500mb.txt'
d = b.read_text(filename, blocksize='15MB')
r = d.frequencies().compute()
```

Додаток І

(довідниковий)

Вирішення задачі з обчислення середнього значення з логів з використанням Dask

```
from dask.distributed import Client
client = Client()
from time import time
filename = 'file:///files/logs_1500mb.txt'
d = dd.read_csv(filename, blocksize='15MB', sep='|', assume_missing=True,
names=('level','datetime','priority','message'))
a.average(d.dropna().c.map(int, meta=('priority', int)).to_dask_array()).compute()
r = d.compute()
```

Додаток Ї

(довідниковий)

Вирішення задачі з підрахунку кількості слів з використанням Flink

```
      from pyflink.table import EnvironmentSettings, TableEnvironment

from pyflink.common import Configuration
from pyflink.table.expressions import lit, col


configuration = Configuration()
configuration.set_string("jobmanager.rpc.address", "jobmanager")


env_settings =
EnvironmentSettings.new_instance().in_batch_mode().with_configuration(configuration
).build()
table_env = TableEnvironment.create(env_settings)
table_env.get_config().set("parallelism.default", "5")


my_source_ddl = """
    create table mySource (
        val int
    ) with (
        'connector' = 'filesystem',
        'format' = 'csv',
        'path' = '/files/int_array_1500mb.txt'
    )
"""

table_env.execute_sql(my_source_ddl)
tab = table_env.from_path('mySource')

tab.order_by(col('val').asc).fetch(1).execute().wait()
```

Додаток Й

(довідниковий)

Вирішення задачі з підрахунку кількості слів з використанням Flink

```
from pyflink.table import EnvironmentSettings, TableEnvironment
from pyflink.common import Configuration
from pyflink.table.expressions import lit, col


configuration = Configuration()
configuration.set_string("jobmanager.rpc.address", "jobmanager")


env_settings =
EnvironmentSettings.new_instance().in_batch_mode().with_configuration(configuration
).build()
table_env = TableEnvironment.create(env_settings)
table_env.get_config().set("parallelism.default", "5")


my_source_ddl = """
    create table mySource (
        word VARCHAR
    ) with (
        'connector' = 'filesystem',
        'format' = 'csv',
        'path' = '/files/words_array_1500mb.txt'
    )
"""

table_env.execute_sql(my_source_ddl)
tab = table_env.from_path('mySource')

tab.group_by('word').select(col('word'), lit(1).count).fetch(1).execute()
```

Додаток К

(довідниковий)

Вирішення задачі з обчислення середнього значення з логів з використанням Flink

```
from pyflink.table import EnvironmentSettings, TableEnvironment
from pyflink.common import Configuration
from pyflink.table.expressions import lit, col


configuration = Configuration()
configuration.set_string("jobmanager.rpc.address", "jobmanager")


env_settings =
EnvironmentSettings.new_instance().in_batch_mode().with_configuration(configuration
).build()
table_env = TableEnvironment.create(env_settings)
table_env.get_config().set("parallelism.default", "5")


my_source_ddl = """
    create table mySource (
        level VARCHAR,
        dt VARCHAR,
        priority INT,
        message VARCHAR
    ) with (
        'connector' = 'filesystem',
        'format' = 'csv',
        'csv.field-delimiter' = '|',
        'csv.ignore-parse-errors' = 'true',
        'path' = '/files/logs_1500mb.txt'
    )
"""

table_env.execute_sql(my_source_ddl)
tab = table_env.from_path('mySource')

tab.filter(col("priority").is_not_null).select(col('priority').avg).execute()
```