Міністерство освіти і науки України
Національний університет «Києво-Могилянська Академія»
Факультет інформатики
Кафедра математики

Кваліфікаційна робота

освітній ступінь - бакалавр

на тему: **«АНАЛІЗ РЕЄСТРУ СУДОВИХ РІШЕНЬ ЗА ДОПОМОГОЮ APACHE SPARK»**

Виконав: студент 4-го року навчання
Освітньої програми «Прикладна математика», 113
Федусов С.В.
Керівник Глибовець А.М.
доктор технічних наук, доцент
Рецензент

_____
(прізвище та ініціали)

Кваліфікаційна робота захищена з оцінкою

_____

Секретар ЕК

_____

«____» _____

20____ р.

Київ - 2021

ЗМІСТ

ПЕРЕЛІК ПРИЙНЯТИХ СКОРОЧЕНЬ

API - application programming interface

RDD - resilient distributed dataset

DSM - distributed shared memory

DAG - directed acyclic graph

JVM - Java virtual machine

vCPU - virtual centralized processing unit

ВСТУП

Процеси сучасного світу дуже часто вимагають взаємодії з великою кількістю даних. Разом з тим, вимоги щодо швидкості їх опрацювання стають усе більш жорсткими. Для обробки зростаючої кількості інформації з наявними фізичними обмеженнями обчислювальних ресурсів були створені розподілені обчислювальні системи.

Окрім продуктивності, важливою компонентою будь-якої обчислювальної системи є її відмовостійкість. У випадку не розподілених систем, цей принцип може порушуватися через наявність єдиної точки відмови. В той же час, розподілені системи дозволяють мінімізувати цей ризик.

Ефективним інструментом для виконання задач аналізу великих обсягів даних є двигун для аналітики Apache Spark [1]. Завдяки розподіленій архітектурі, відмовостійкості та високій продуктивності він був обраний для виконання аналізу датасету судових рішень.

Мета роботи - проаналізувати реєстр судових рішень та на основі отриманих даних дослідити характеристичні показники процесу судочинства.

Постановка задачі:

1. Розглянути архітектуру та функціонал інструмента для аналізу Apache Spark.
2. Ознайомитися зі структурою даних реєстру судових рішень.
3. Розробити застосунок для попередньої обробки та аналізу судових рішень.
4. Виконати аналіз даних реєстру за допомогою Apache Spark.
5. Використати отримані результати для побудови візуалізації та формулювання висновків.

РОЗДІЛ 1. МОЖЛИВОСТІ ТА АРХІТЕКТУРА APACHE SPARK

Задача опрацювання великої кількості інформації вимагає ефективних способів її читання і обробки в розподіленій системі. Наведені вимоги допомагає задовольнити Apache Spark. У цьому розділі будуть розглянуті можливості та архітектура наведеного програмного інструменту.

Застосунок Spark побудований за моделлю master-worker (головний-підлеглий) (Рис. 1.1).

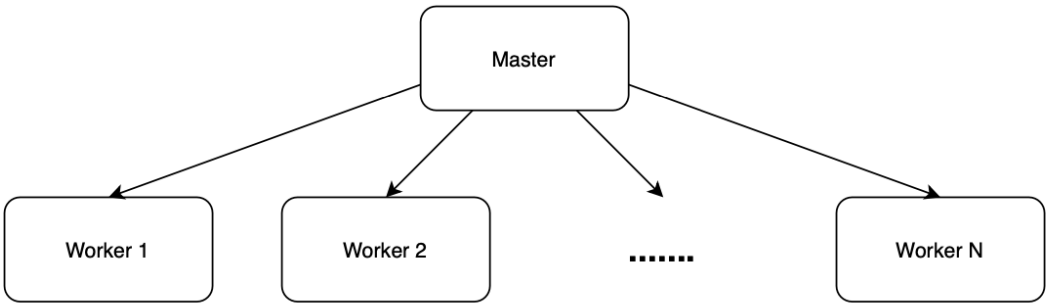


Рис. 1.1 Master-worker архітектура

Цей архітектурний шаблон передбачає наявність головного (master) вузла і кількох підпорядкованих (worker) вузлів. Хоча ці компоненти логічно відокремлені, фізично вони можуть бути розташовані на одній обчислювальній машині в якості ізольованих процесів.

Головний вузол Apache Spark кластеру є точкою входу для виконання клієнтських програм. Він виконує роль розпорядника у розподіленій системі, керуючи робочими вузлами шляхом надання їм інструкцій.

Розділ 1.1. Виконання клієнтської програми в Spark кластері

Для того, щоб мати підґрунтя для прийняття технічних рішень, пов’язаних з Apache Spark, потрібно розуміти його базову архітектуру та основні принципи виконання кластером клієнтської програми. Для цього, розглянемо схему виконання кластером Spark типового застосунку (Рис. 1.2).

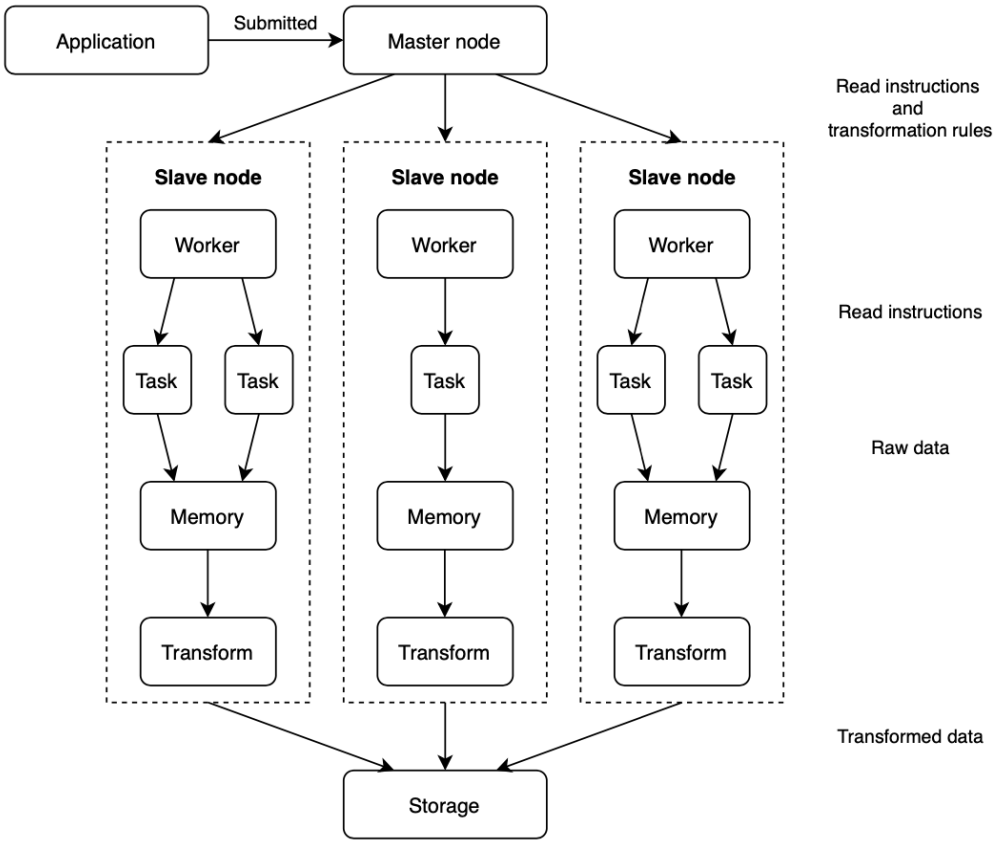


Рис. 1.2 Схема виконання користувацької програми на Spark кластері [2]

1. Клієнтська програма подається на вхід головному вузлу кластера.
2. Головний вузол надсилає підпорядкованим інструкції щодо розподіленого завантаження необхідних даних, а також набір правил для їх перетворення.

3. Для виконання інструкцій головного вузла, підпорядкований створює задачі на читання призначеної йому частини даних. Шляхом паралельного виконання задач та розподіленого читання досягається висока продуктивність.
4. Трансформація зчитаних даних. Кожен підпорядкований вузол, маючи власну частину інформації, перетворює її за правилами, попередньо визначеними клієнтським застосунком.
5. Збереження результату клієнтської програми.

Детальніше розглянемо процес виконання Spark клієнтських інструкцій. В основі роботи двигуна покладено ліниве виконання операцій. Це означає, що інструкції клієнтського застосунку не обов’язково будуть виконуватися в тому порядку і в той момент, що зазначений безпосередньо у коді. Наприклад, операції читання або перетворення даних не будуть розпочаті тоді, коли це явно зазначено. Лише так завані термінальні методи спричинять виконання усіх попередніх інструкцій. Лінива стратегія дозволяє виконувати лише ті операції, що дійсно впливають на кінцевий результат, а також є способом уникнути непотрібних проміжних результатів в ланцюжку перетворень.

Apache Spark перетворює код клієнтського застосунку на DAG - орієнтований ациклічний граф дій та трансформацій [4]. Для найбільш ефективного виконання, цей граф попередньо модифікується вбудованим оптимізатором Catalyst [2]. Результуючий план застосунку надалі виконується кластером.

Розділ 1.2. Spark API

Аналіз даних - задача дуже загальна, а отже вимагає гнучких інструментів для універсального застосування. Зазвичай інформація для аналізу зводиться до єдиної структури. Така структура називається схемою - це заздалегіть фіксований формат, якому дані підпорядковуються. Існування схеми дозволяє вказати тип даних кожного атрибуту елементу датасету, а також визначити, чи може атрибут мати значення null. Наявність визначеного набору обмежень дає можливість більш коректно визначати функції для поелементної трансформації даних.

Базовим інструментом для роботи клієнтського застосунку зі Spark кластером є SparkSession. Для її створення достатньо вказати url головного вузла кластеру та ім’я програми. Додатково є можливість визначити параметри конфігурації, наприклад обмеження оперативної пам’яті для процесів робочих вузлів. Об’єкт SparkSession використовується для читання даних і взаємодії з ресурсами кластеру впродовж проміжку часу роботи застосунку.

Основним компонентом Spark API для роботи з даними є Dataframe. Ця структура інкапсулює логіку представлення даних у вигляді набору рядків та колонок (Рис. 1.3), а також реалізує функції взаємодії з ними, такі як перетворення, перегляд та аналіз.

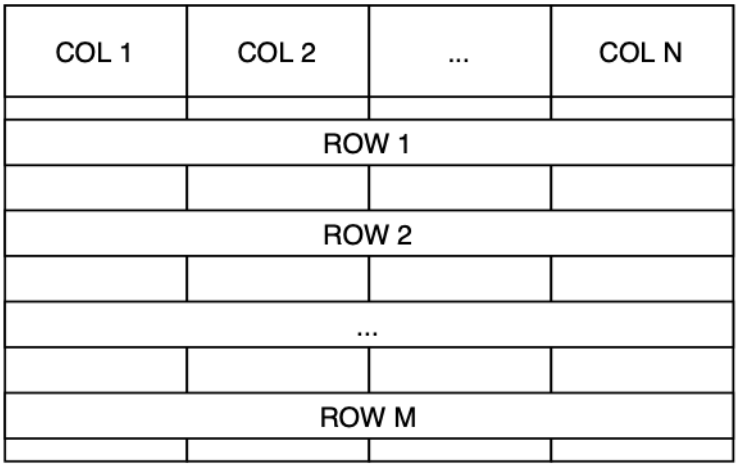


Рис. 1.3 Логічна схема Dataframe з N колонок і M рядків [2]

Датафрейм є поняттям, еквівалентним до таблиці в реляційних базах даних [2]. Ще однією концепцією, яку використовує Spark, є незмінність. Датафрейм - це незмінна структура, створення якої відбувається лише двома способами: в результаті читання даних або застосовуючи перетворення до попередньо існуючого датафрейму.

Так як операції в Spark виконуються ліниво, результатом ланцюжка трансформацій є кінцевий датафрейм. Однак, іноді виникає необхідність застосувати до нього декілька окремих операцій і отримати декілька різних результатів. В такому випадку, для кожного результуючого датафрейму буде повторно виконано всі попередні перетворення (Рис. 1.4).

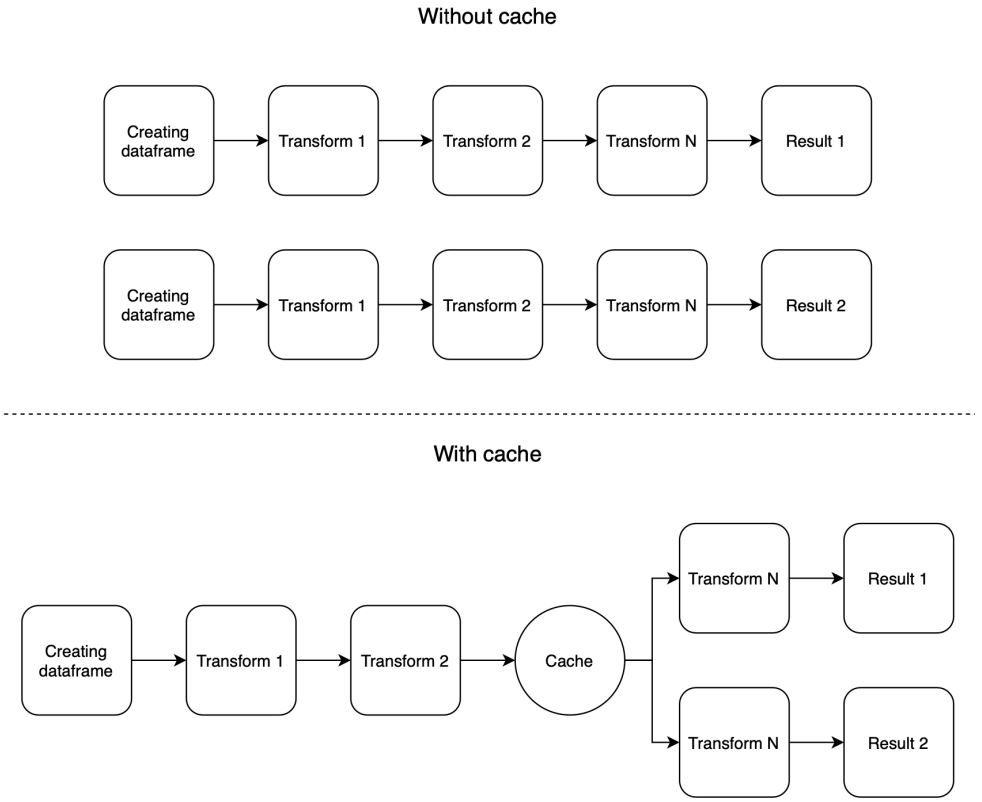


Рис. 1.4 Порівняння виконання операцій без кешування та з ним [2]

Такий підхід є зовсім неефективним. Вирішити цю задачу дозволяє функція кешування. Використовуючи методи датафрейму *cache()* або *persist()*, структура розміщується в пам'яті до моменту, поки сесія застосунку не завершиться або його явно не буде видалено методом *unpersist()* [2]. Датафрейм може бути збережено в оперативній пам'яті, на диску, або використовуючи комбінований підхід. Використання методу *cache()* є синонімом до *persist(StorageLevel.MEMORY_ONLY)* [2]. Для кешування за іншою стратегією, необхідно в методі *persist()* вказати відповідний *StorageLevel*. Таким чином, наступні операції будуть застосовані до збереженого датафрейму, дозволяючи уникнути попередньо виконаних кроків для його створення.

Попри на перший погляд тривіальний концепт, Dataframe втілює ще одну важливу особливість. Навіть прості взаємодії, такі як обрахунок кількості елементів датасету в розподіленій системі вимагають опрацювання кожної його частини. Dataframe грунтується на розподіленій пам’яті, а отже виконує операції на всіх вузлах кластеру, дозволяючи отримати результат для повного набору даних. Це є можливим завдяки ще одній структурі Spark - Resilient Distributed Dataset.

Розділ 1.3. Resilient distributed dataset

Resilient Distributed Dataset - "*is a read-only, partitioned collection of records*" [3]. Його було створено в якості альтернативи існуючим DSM рішенням, накладаючи додаткові обмеження для досягнення кращої відмовостійкості [3]. Об'єкт RDD створюється застосовуючи функцію перетворення до множини даних. Подальші операції з RDD поділяються на два типи: перетворення та дії. Перетворення визначаються як функції виду RDD[T] => RDD[R] або (RDD[T], RDD[T]) => RDD[R] (Рис. 1.5).

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.,* HDFS

Рис. 1.5 Перетворення та дії, визначені в RDD [3]

З архітектурної точки зору, RDD імплементує парадигму DSM в її широкому розумінні. Дані в RDD належать до розділів, що фізично можуть знаходитися на різних обчислювальних машинах (Рис. 1.6), а логічно належать єдиній структурі.

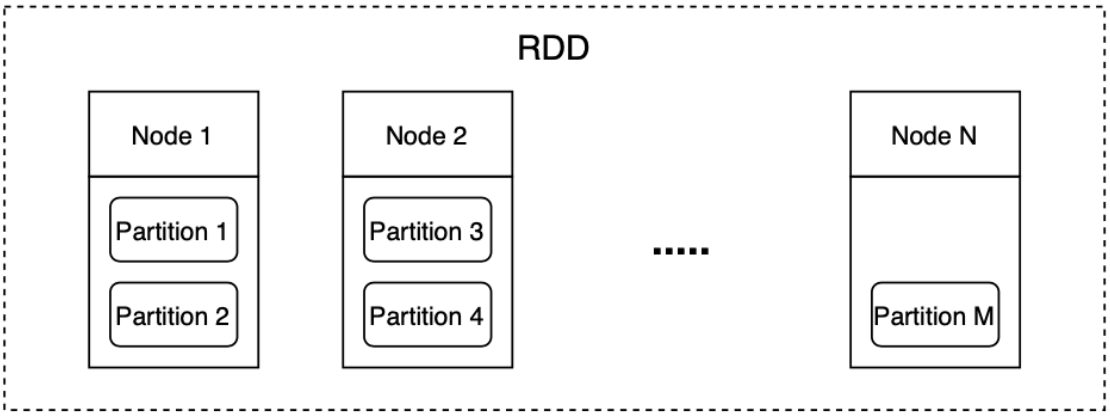


Рис. 1.6 Схема фізичного розміщення даних в RDD

Розділи RDD (partitions) - це частини структури, обмежені за розміром, призначені для розподіленого зберігання даних в системі.

Операції в RDD поділяються на два типи: narrow і wide [3]. Narrow тип операцій передбачає, що кожен розділ даних результату операції залежить від сталої кількості розділів даних, на яких виконується операція (Рис. 1.7) [3].

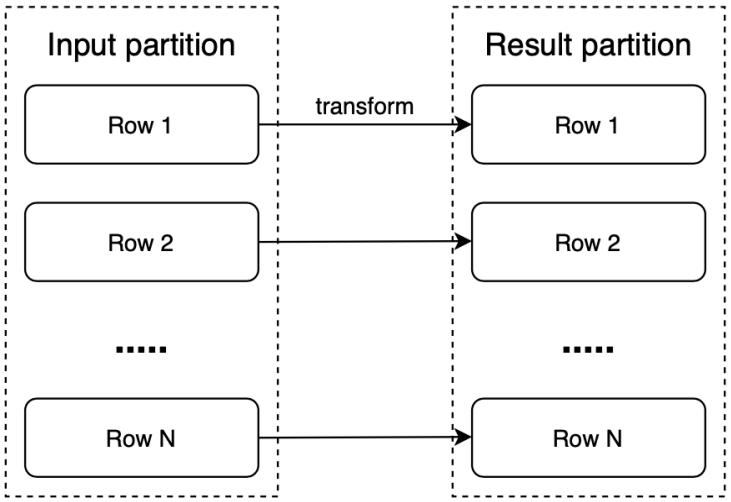


Рис. 1.7 Схема narrow операцій в RDD

Гарними прикладами narrow операцій є функції *map()* (перетворення) або *filter()* (фільтрація). Ці операції виконуються поелементно, застосовуючи задану функцію для кожного елементу розділу. Таким чином, операція перетворення кожного розділу залежить лише від його даних.

На відміну від narrow, для wide операції опрацювання кожного розділу залежить від кількох інших [3] (Рис. 1.8).

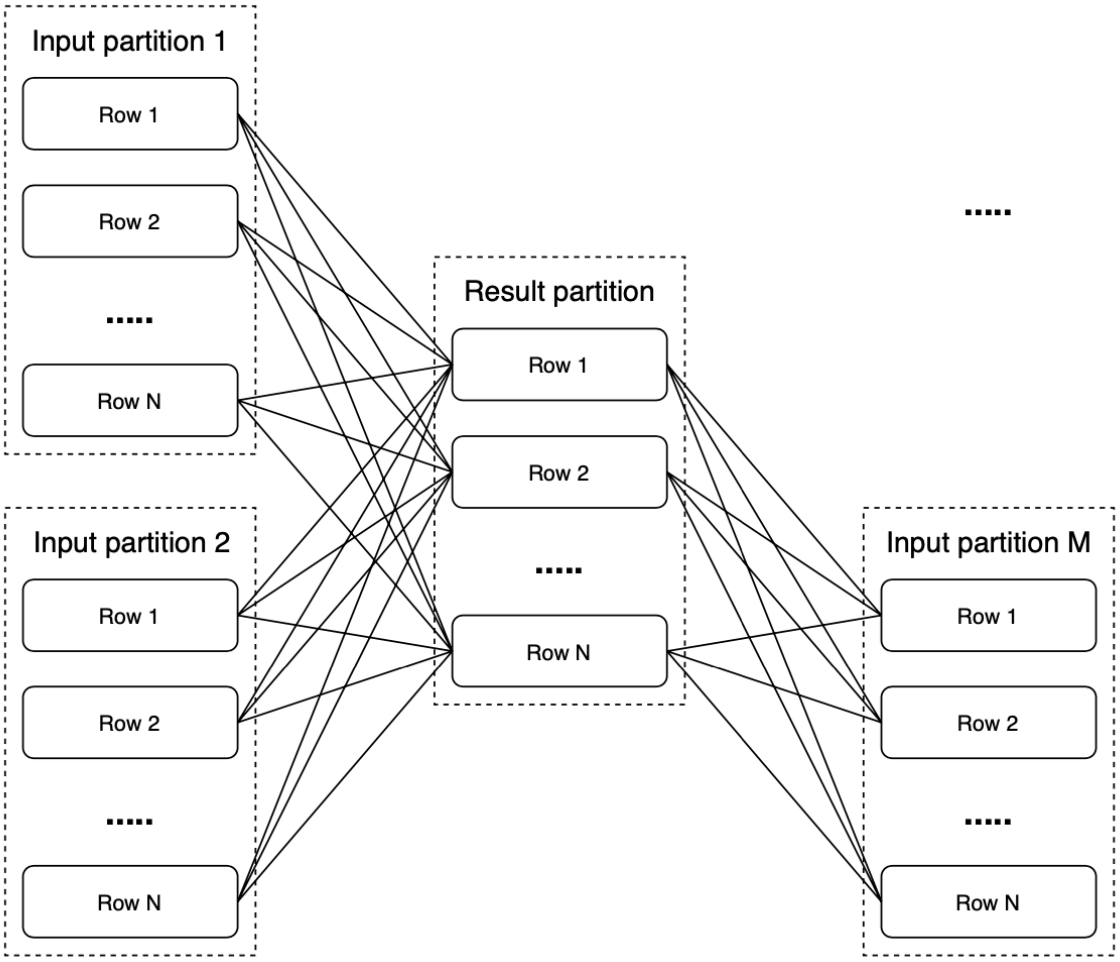


Рис. 1.8 Схема wide операцій в RDD

До цього типу операцій належать агрегатні функції і групування. "*Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary. Grouping is used to create subgroups of tuples before summarization.*" [5]. Прикладами агрегатних функцій є *avg()*

(обрахування середнє значення) або *stddev()* (обчислення стандартного відхилення). Групування виконуються за допомогою функцій *groupBy()* та *groupByKey()*.

РОЗДІЛ 2. ЗАСТОСУВАННЯ APACHE SPARK ДЛЯ АНАЛІЗУ СУДОВИХ РІШЕНЬ

Як відомо з розділу 1, Apache Spark - це розподілена система для опрацювання великої кількості даних. Така архітектура дозволяє за потреби виконувати горизонтальне масштабування - збільшення кількості обчислювальних машин в кластері, таким чином долаючи фізичні обмеження однієї обчислювальної машини і досягаючи високої продуктивності. За можливості, Spark розміщує дані в оперативній пам’яті, що дозволяє виконувати перетворення значно швидше, ніж використовуючи диск. Саме тому цю систему було обрано для аналізу реєстру судових рішень.

Судове рішення - це документ, що містить основну інформацію про судову справу та її підсумки. В цьому розділі буде розглянуто структуру даних реєстру та описано поетапний процес його попередньої обробки та аналізу.

Розділ 2.1. Датасет судових рішень

Використаний датасет був отриманий з єдиного державного реєстру судових рішень. Згідно з правилами реєстру, розміщені судові рішення є відкритими для безоплатного цілодобового доступу [8]. Датасет складається з дванадцяти розділів, кожен з яких містить файли судових рішень за відповідний місяць 2019 року. Кожен розділ є попередньо стиснутим і збереженим у форматі 7z. Загальний об’єм стиснутих розділів становить приблизно 3.2 гігабайти, а після розархівування - більш ніж 60 гігабайт. Файли судових рішень збережені у вигляді html документу, де ім’я файлу відповідає реєстраційному номеру рішення.

На жаль, структура файлів створює певні складності для їх обробки. Не існує єдиного чіткого формату, якому вони підпорядковуються, а отже, попередня обробка деяких документів може бути невдалою. Така ситуація не є рідкістю серед застосунків аналізу даних. Spark надає стандартний інструмент для вирішення цієї задачі: можливість відсіяти дані з невизначеними атрибутами або надати їм значення за замовчуванням.

Базова інформація, викладена у файлах рішень, є спільною для всіх типів документів. На Рис. 2.1 виділені основні структурні елементи судового рішення: 1 - ім’я судової установи, 2 - форма судового рішення, 3 - дата винесення рішення, 4 - номер судової справи, 5 - склад суддів.

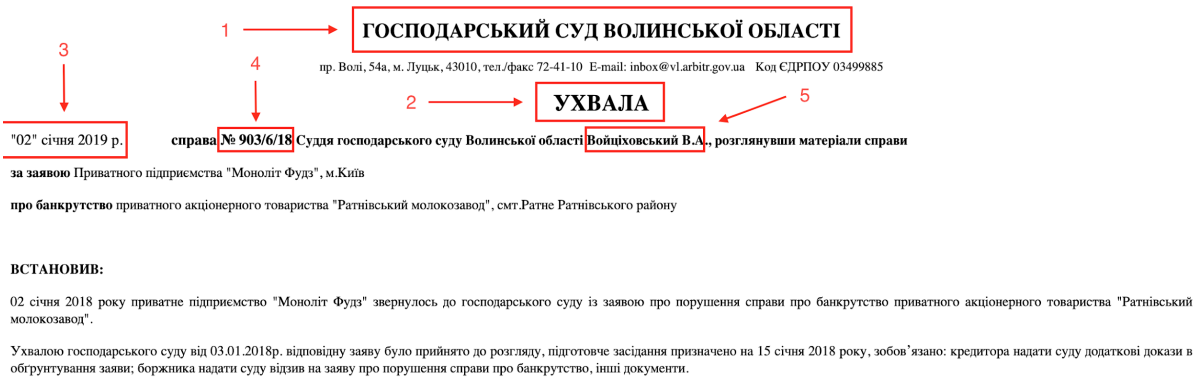
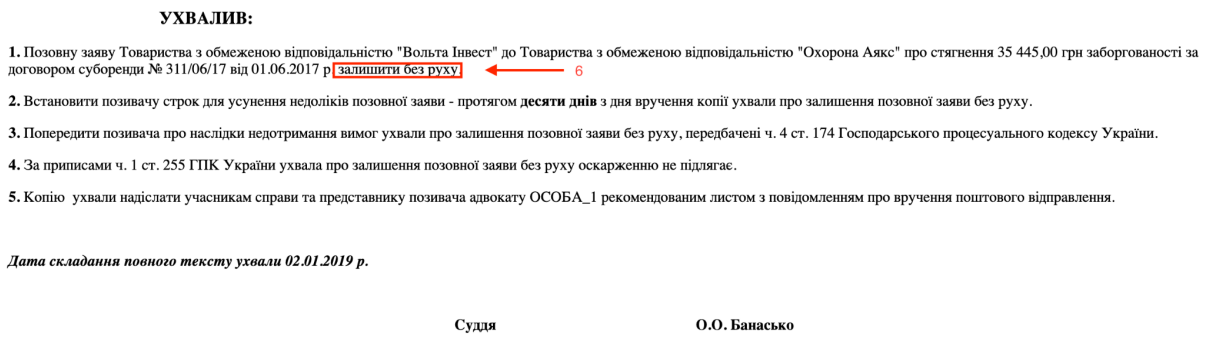


Рис. 2.1 Основні елементи судового рішення

Окрім зазначених вище атрибутів рішень, інформація, викладена в них, включає також винесений вирок. Як правило, вирок викладений у кінці документу рішення у вільному форматі (Рис. 2.2).



УХВАЛИВ:

1. Позовну заяву Товариства з обмеженою відповідальністю "Вольта Інвест" до Товариства з обмеженою відповідальністю "Охорона Аякс" про стягнення 35 445,00 грн заборгованості за договором суборенди № 311/06/17 від 01.06.2017 р **залишити без руху** 6

2. Встановити позивачу строк для усунення недоліків позовної заяви - протягом **десяти днів** з дня вручення копії ухвали про залишення позовної заяви без руху.

3. Попередити позивача про наслідки недотримання вимог ухвали про залишення позовної заяви без руху, передбачені ч. 4 ст. 174 Господарського процесуального кодексу України.

4. За приписами ч. 1 ст. 255 ГПК України ухвала про залишення позовної заяви без руху оскарженню не підлягає.

5. Копію ухвали надіслати учасникам справи та представнику позивача адвокату ОСОБА_1 рекомендованим листом з повідомленням про вручення поштового відправлення.

Дата складання повного тексту ухвали 02.01.2019 р.

Суддя **О.О. Банасько**

Рис. 2.2 Подання вироку в документі рішення

Попередня обробка вироків здійснювалась за принципом класифікації: було задоволено позов, чи навпаки, відхилено. Для цього, було визначено множину відповідних формулювань (Рис. 2.2 елемент 6). Вироки, формулюваннями яких є “задовольнити”, “визнати винуватим”, “визнати винним” або “стягнути” було віднесено до класу задоволених, в той час, як “залишити без руху”, “залишити без задоволення” або “відмовити” є свідченням не задоволених рішень. Вирокам, що не є фінальними у справі, такі як відкриття провадження, призначення наступного засідання або рішення про додатковий розгляд було надано клас “інше”.

Розділ 2.2. Читання та попередня обробка файлів рішень

Першим кроком аналізу даних в Spark кластері є їх зчитування. Для виконання клієнтських процесів в кластері було використано Python та бібліотеку pyspark. Apache Spark, у свою чергу, працює на платформі JVM, а отже вбудовано підтримує мови програмування Scala, Java і Kotlin. Підтримка Python реалізована додатково, що дозволяє повноцінно працювати з кластером і застосовувати всі можливості мови програмування, зокрема працювати з бібліотеками машинного навчання.

Для тестування розроблюваного застосунку було створено локальний Spark кластер, використовуючи інструмент контейнеризації Docker та публічно доступні Spark Docker образи від організації Big Data Europe [6]. На Рис. 2.3 поданий docker-compose файл конфігурації Spark кластеру. Docker-compose дає можливість визначити множину контейнерів з відповідними конфігураційними параметрами та керувати їх життєвим циклом.

docker-compose.yml

```yaml
version: '3'
services:
  spark-master:
    image: bde2020/spark-master:3.0.2-hadoop3.2
    container_name: spark-master
    ports:
      - "8080:8080"
      - "7077:7077"
    environment:
      - INIT_DAEMON_STEP=setup_spark
      - PYSPARK_PYTHON=python3
    volumes:
      - /Users/uzer/Desktop/court_data:/court_data:ro

  spark-worker-1:
    image: bde2020/spark-worker:3.0.2-hadoop3.2
    container_name: spark-worker-1
    depends_on:
      - spark-master
    ports:
      - "8081:8081"
    environment:
      - "SPARK_MASTER=spark://spark-master:7077"
    volumes:
      - /Users/uzer/Desktop/court_data:/court_data:ro

  spark-worker-2:
    image: bde2020/spark-worker:3.0.2-hadoop3.2
    container_name: spark-worker-2
    depends_on:
      - spark-master
    ports:
      - "8082:8081"
    environment:
      - "SPARK_MASTER=spark://spark-master:7077"
    volumes:
      - /Users/uzer/Desktop/court_data:/court_data:ro

```

Рис. 2.3 Docker-compose конфігурація Spark кластеру

Тестовий кластер складається з одного master та двох worker вузлів. Для кожного вузла сконфігуровано volume (розділ), що дає можливість процесам в контейнері взаємодіяти з вказаними файлами у файловій системі фізичної машини. У прикладі, наведеному вище, кожен вузол кластеру отримує доступ виключно на читання файлів судових рішень.

Розглянемо процес читання файлів судових рішень. На Рис. 2.4 наведений код простого Spark застосунку.

Рис. 2.4 Читання файлів рішень

Першим кроком є створення об’єкту SparkSession. На рядку 7 метод *wholeTextFiles()* виконує зчитування усіх файлів у зазначеній директорії, а метод *toDf()* повертає об’єкт Dataframe, що складається з двох стовпчиків: ім’я файлу і його вміст. Метод *show()* виводить п’ять перших його рядків у консоль і застосунок завершує свою роботу (Рис. 2.5).

```
21/04/10 15:22:57 INFO Executor: Finished task 0.0 in stage 1.0 (TID 1). 19696 bytes result sent to driver
21/04/10 15:22:57 INFO TaskSetManager: Finished task 0.0 in stage 1.0 (TID 1) in 359 ms on 06f5065e1b55 (executor driver) (1/1)
21/04/10 15:22:57 INFO DAGScheduler: ResultStage 1 (showString at NativeMethodAccessorImpl.java:0) finished in 0.463 s
21/04/10 15:22:57 INFO DAGScheduler: Job 1 is finished. Cancelling potential speculative or zombie tasks for this job
21/04/10 15:22:57 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks have all completed, from pool
21/04/10 15:22:57 INFO TaskSchedulerImpl: Killing all running tasks in stage 1: Stage finished
21/04/10 15:22:57 INFO DAGScheduler: Job 1 finished: showString at NativeMethodAccessorImpl.java:0, took 0.489438 s
21/04/10 15:22:58 INFO CodeGenerator: Code generated in 52.3743 ms
+--------------------+--------------------+
|                  _1|                  _2|
+--------------------+--------------------+
|file:/court_data/...|<!DOCTYPE html ><...|
|file:/court_data/...|<!DOCTYPE html ><...|
|file:/court_data/...|<!DOCTYPE html ><...|
|file:/court_data/...|<!DOCTYPE html ><...|
|file:/court_data/...|<!DOCTYPE html ><...|
+--------------------+--------------------+
only showing top 5 rows

21/04/10 15:22:58 INFO SparkUI: Stopped Spark web UI at http://06f5065e1b55:4040
21/04/10 15:22:58 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
21/04/10 15:22:58 INFO MemoryStore: MemoryStore cleared
21/04/10 15:22:58 INFO BlockManager: BlockManager stopped
21/04/10 15:22:58 INFO BlockManagerMaster: BlockManagerMaster stopped
21/04/10 15:22:58 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
21/04/10 15:22:58 INFO SparkContext: Successfully stopped SparkContext
```

Рис. 2.5 Стандартний вивід застосунку

За замовчуванням вивід застосунку має обмежений розмір стовпчиків, через що на Рис. 2.5 значення назви файлу і вмісту показані лише частково. Метод датафрейму *show()* має параметри *truncate* і *vertical*, що дозволяють налаштувати стандартну поведінку. Однак вміст файлу є занадто об’ємним,

через що вивід застосунку з використанням зазначених параметрів читати надзвичайно складно. Імена стовпчиків в прикладі вище також генеруються за замовчуванням, однак для більшої зручності в роботі з датафреймом краще вказати їх в явному вигляді, що буде продемонстровано у наступному прикладі.

Попередня обробка файлів рішень полягає у перетворенні імені та вмісту рішення в множину його основних структурних елементів. На Рис. 2.6 наведено код застосунку, що на основі прочитаних даних виконує цю трансформацію.

decision_parser.py analyze.py

```
from pyspark.sql import SparkSession

from decision_parser import parse_reg_num, parse_court_name, parse_doctype, parse_date, parse_case_num, parse_judges,
    parse_decision

spark = SparkSession.builder \
    .appName('Court Decisions Analysis') \
    .getOrCreate()

court_data = spark.sparkContext \
    .wholeTextFiles('/court_data/') \
    .toDF() \
    .withColumnRenamed('_1', 'filename') \
    .withColumnRenamed('_2', 'content')

court_data_preprocessed = court_data \
    .withColumn('reg_num', parse_reg_num('filename')) \
    .withColumn('court_name', parse_court_name('content')) \
    .withColumn('type', parse_doctype('content')) \
    .withColumn('date', parse_date('content')) \
    .withColumn('case_num', parse_case_num('content')) \
    .withColumn('judges', parse_judges('content')) \
    .withColumn('decision', parse_decision('content')) \
    .drop('filename') \
    .drop('content') \
    .cache()

court_data_preprocessed.show(100, vertical=True, truncate=False)

spark.stop()
```

Рис. 2.6 Виділення основних структурних елементів з файлів рішень

Для зручності роботи, стовпчикам імені і вмісту файлу надаються відповідні імена, використовуючи метод *withColumnRenamed()*. Наступний крок обробки виконує додавання множини обрахованих значень до кожного елементу датафрейму. Після цього базові дані більше не потрібні і

видаляються з результату операції методом *drop()*. В цьому випадку, зручно застосувати вертикальний вивід результату застосунку для повного відображення імені установи та переліку суддів.

Щоб виділити наведені дані (Рис. 2.7) з тексту рішень, було використано Python бібліотеку beautifulsoup4 [7] для роботи з html форматом, в якому подані файли рішень.

```
-RECORD 36-----------------------------------------------------------------------
 reg_num    | 78952368
 court_name | ГОСПОДАРСЬКИЙ СУД ВІННИЦЬКОЇ ОБЛАСТІ
 type       | наказ
 date       | 02.01.2019
 case_num   | 902/638/18
 judges     | Нешик О.С.
 decision   | задовольнити
-RECORD 37-----------------------------------------------------------------------
 reg_num    | 78952371
 court_name | СХІДНИЙ АПЕЛЯЦІЙНИЙ ГОСПОДАРСЬКИЙ СУД
 type       | ухвала
 date       | 02.01.2019
 case_num   | 922/1938/18
 judges     | Л.І. Бородіна, О.В. Плахов, І.А. Шутенко, В.В. Докучаєва
 decision   | не задовольнити
-RECORD 38-----------------------------------------------------------------------
 reg_num    | 78952373
 court_name | ГОСПОДАРСЬКИЙ СУД ВОЛИНСЬКОЇ ОБЛАСТІ
 type       | ухвала
 date       | 02.01.2019
 case_num   | 903/6/18
 judges     | В. А. Войціховський
 decision   | інше
-RECORD 39-----------------------------------------------------------------------
 reg_num    | 78952376
 court_name | ЦЕНТРАЛЬНИЙ АПЕЛЯЦІЙНИЙ ГОСПОДАРСЬКИЙ
 type       | ухвала
 date       | 02.01.2019
 case_num   | 908/1475/18
 judges     | Ю.Б. Парусніков
 decision   | не задовольнити
```

Рис. 2.7 Виділені структурні елементи судових рішень

Розділ 2.3. Виконання аналізу та візуалізація отриманих результатів

В розділі 2.2 було описано процес попередньої обробки файлів судових рішень. Цей крок мав на меті перетворити різнорідні дані в єдину структуру і є необхідним для імплементації алгоритмів їх аналізу. В свою чергу, аналіз має на меті отримання характеристичних показників судового перебігу, на основі яких можна зробити висновки про процес судочинства в Україні. В цьому параграфі будуть представлені методи знаходження а також візуалізація таких даних.

Застосунок аналізу було розроблено, використовуючи сервіси хмарного провайдера Google Cloud Platform: Cloud Storage та Dataproc. Google Cloud Storage - це сховище об’єктів, яке надає можливість зберігати та ефективно отримувати доступ до файлів рішень [9]. Судові рішення розміщуються у заздалегідь створеному кошику - фундаментальному ресурсі, яким оперує об’єктне сховище. Сервіс Dataproc дозволяє швидко розгорнути та керувати життєвим циклом Apache Spark кластеру для виконання клієнтських задач [10].

Обмеження безкоштовного Google Cloud Platform аккаунту дозволяють використовувати не більше ніж 8 vCPU одночасно. Таким чином, для роботи застосунка було створено кластер з чотирьох віртуальних машин типу n2-highmem-2: однієї ведучої та трьох підпорядкованих. Цей тип машин має 2 vCPU Intel Cascade Lake та 16 гігабайт оперативної пам’яті. На Рис. 2.8 зображено параметри робочих машин для застосування в кластері.

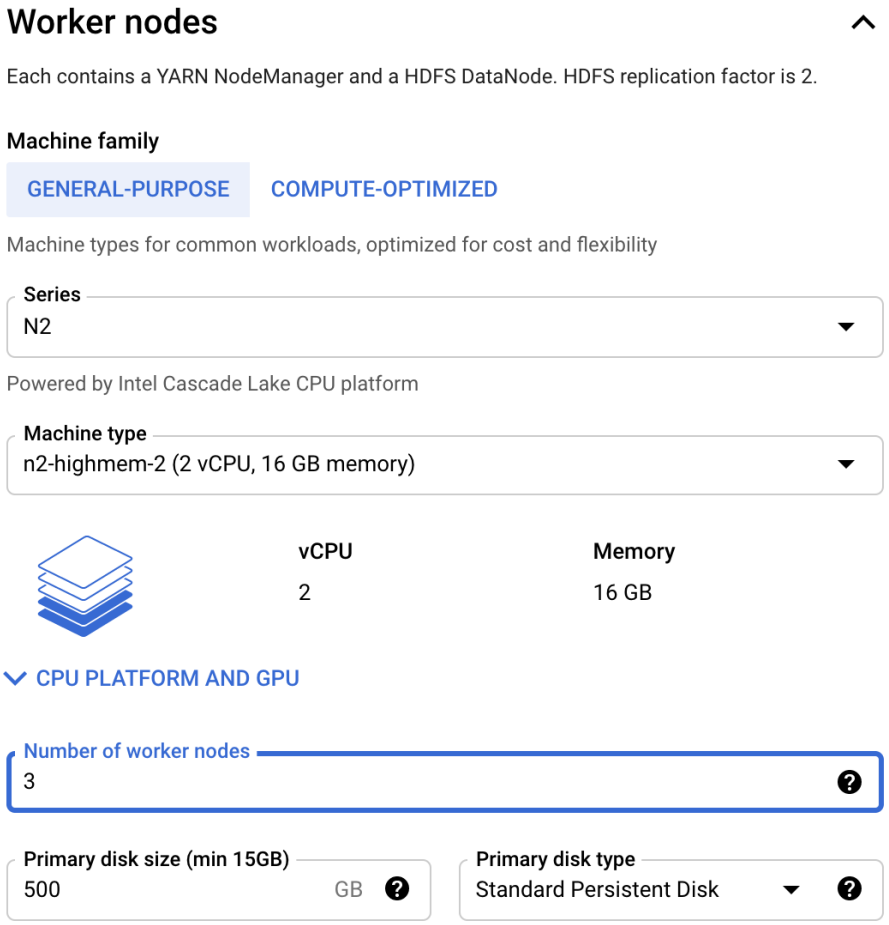


Рис. 2.8 Конфігурація робочих вузлів Dataproc кластера

Альтернативно є можливість обрати інші типи віртуальних машин: n1-highmem-2 (2 vCPU на базі Intel Skylake, 13 гігабайт оперативної пам’яті) та n2d-highmem-2 (2 vCPU на базі AMD EPYC Rome, 16 гігабайт оперативної пам’яті). В порівнянні з n1-highmem-2, n2-highmem-2 має новішу архітектуру процесора та більший обсяг оперативної пам’яті, а отже є більш продуктивним, але дещо дорожчим.

Розглянемо задачу побудови розподілу судових рішень за типом на створеному кластері. Судові рішення в датасеті поділяються на кілька основних типів: постанова, ухвала, рішення і наказ. На Рис. 2.9 наведено код застосунку, що будує розподіл рішень за типом.

```
def cases_by_type(data: DataFrame):
    data \
        .groupBy('type') \
        .count() \
        .coalesce(1) \
        .write \
        .csv('gs://court_data_registry/cases_by_type')
```

Рис. 2.9 Побудова розподілу рішень за типом

Спочатку попередньо оброблені файли судових рішень групуються за типом. Після цього, підрахувавши кількість елементів в кожній групі, результат у форматі csv зберігається у кошик court_data_registry в Google Cloud Storage. Окремо слід зазначити роль методу *coalesce()* в описаному алгоритмі. Згідно з інформацією, викладеною в розділі 1, датафрейм рішень фізично складається з множини його частин, розташованих на різних вузлах кластеру. Запис результатів підрахунку рішень за типом виконується для кожної з частин датафрейму, а отже до кошику буде збережено n файлів, де n = кількості частин. Отримати результат аналізу в єдиному файлі дозволяє застосування *coalesce(1)*, що попередньо зменшує кількість частин до однієї.

Вихідні дані розподілу рішень за типом було використано для побудови кругової діаграми. Це потрібно для того, щоб якнайкраще оцінити частку одного типу рішень відносно інших. Цю та подальші візуалізації було створено засобами Python з використанням бібліотеки matplotlib. З Рис. 2.10 зрозуміло, що найбільш поширеним типом судових рішень є “ухвала” - 68.1% усіх проаналізованих рішень.

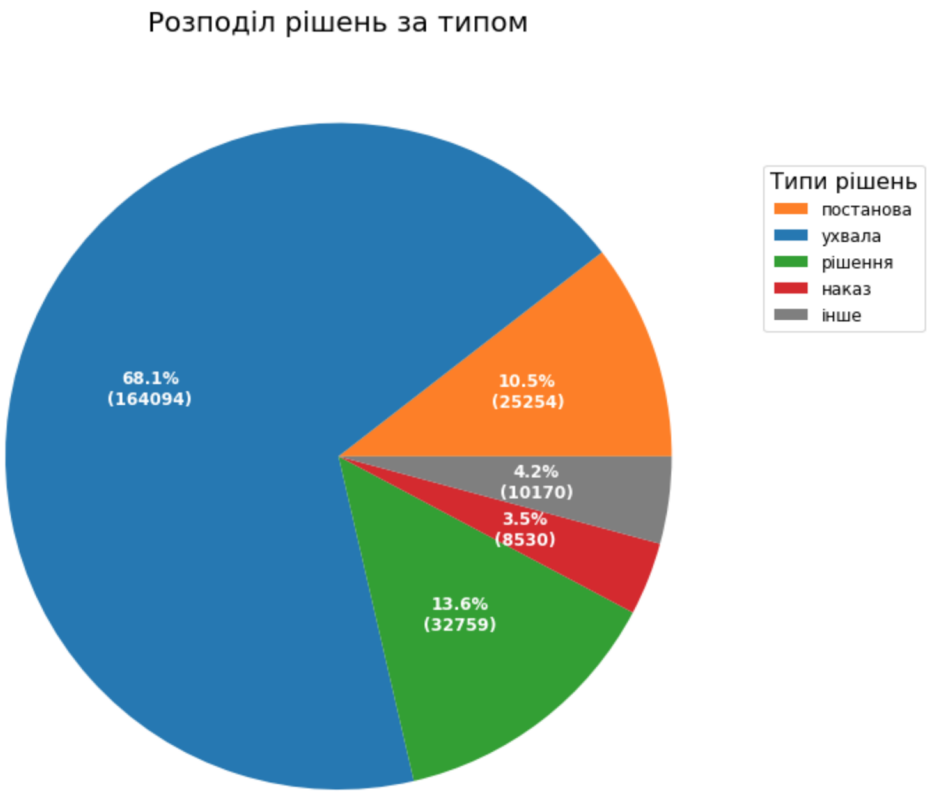


Рис. 2.10 Кругова діаграма розподілу рішень за типом

Наступними типами за частотою є “рішення”, “постанова” та “наказ” - 13.6%, 10.5% і 3.5% відповідно. Судові рішення іншого типу або рішення, що не були корректно оброблені потрапляють до категорії “інше”. Як було зазначено у розділі 2.1, структура файлів рішень дуже різниться для кожної судової установи, а отже випадки помилок попередньої обробки можуть бути присутні.

Наступним методом аналізу датасету рішень є побудова розподілу за датою винесення рішення. Завдяки цьому можна зробити припущення про коректність даних, а також оцінити темп роботи судочинних установ. Побудова розподілу за датою виконується аналогічно попередньому прикладу, виконуючи групування попередньо оброблених результатів за атрибутом date. Графічне представлення результату аналізу було виконано у вигляді стовпчикової діаграми (Рис. 2.11).

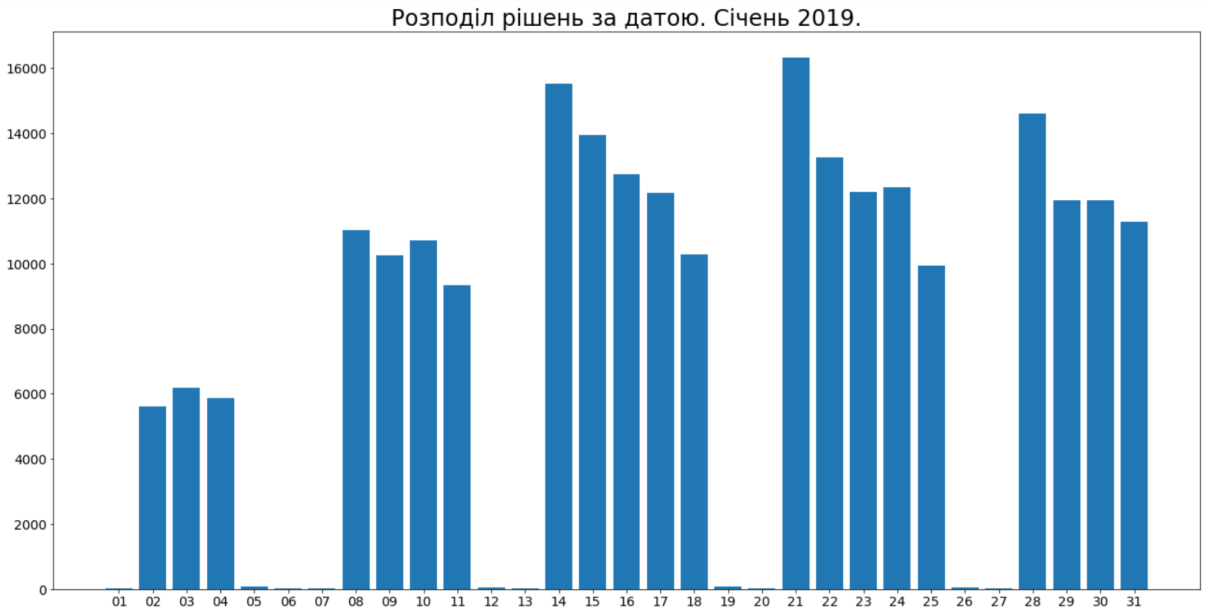


Рис. 2.11 Стовпчикова діаграма розподілу рішень за датою

З діаграми, зображеної на рисунку, можна зробити декілька припущень. Перше, на що варто звернути увагу, це кількість рішень, датованих вихідними і святковими днями. Загалом вона дуже мала у порівнянні з робочими, але не є нульовою. Виконавши подальше дослідження причин цього результату, а саме виключивши з вибірки рішення, датовані робочими днями, було встановлено, що в датасеті дійсно присутні документи рішень, що були винесені у вихідний день. Це означає, що результат попередньої обробки і побудови розподілу за датою є корректним.

Наступне припущення, яке можна сформулювати на основі розподілу рішень за датою стосується динаміки винесення рішень. На графіку чітко простежується зменшення кількості винесених рішень впродовж тижня. Розглянемо задачу формулювання та обґрунтування статистичної гіпотези. Сформулюємо нульову гіпотезу наступним чином: динаміка винесення рішень на початку тижня не відрізняється від показників в кінці тижня. Альтернативна гіпотеза представлена зворотним твердженням: динаміка винесення рішень на початку та в кінці тижня є

різною. Для доведення або спростування сформульованих гіпотез корисно отримати більшу кількість даних: результати аналізу січневих рішень не є показовими, адже на цей місяць припадає святковий період. З графіку також можна спостерігати, що перші два робочі тижні відрізняються за динамікою від наступних, післясвяткових. Додатково було проведено аналіз рішень, винесених у квітні, липні та жовтні. Розподіл рішень за датою у липні наведено на Рис. 2.12.

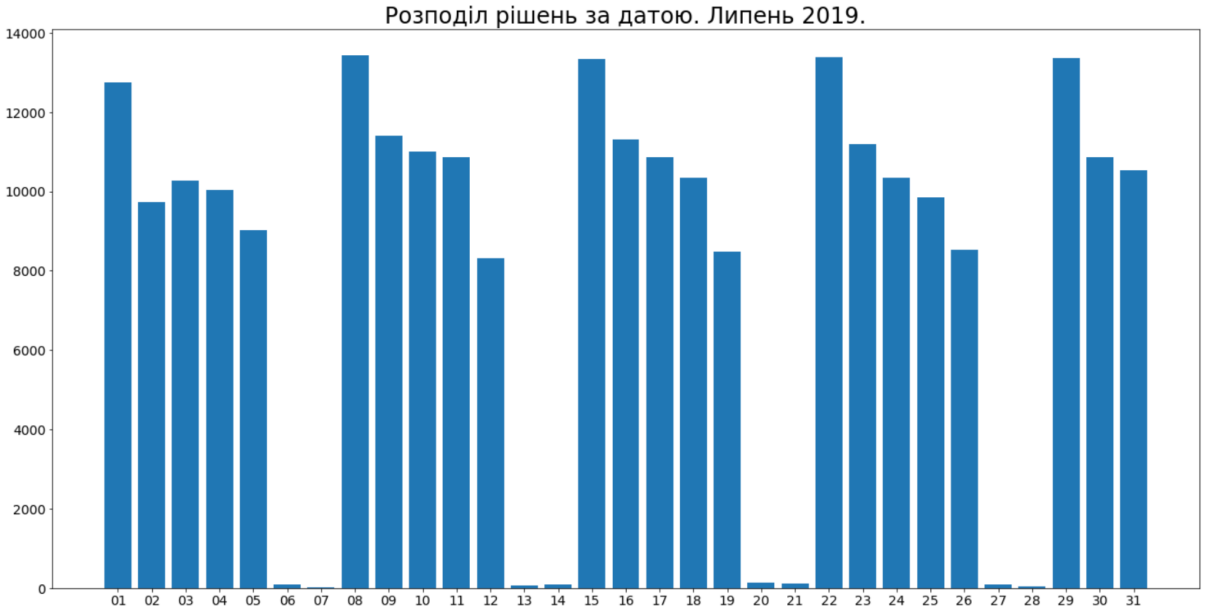


Рис. 2.12 Результат додаткового аналізу розподілу рішень

На графіку спостерігається динаміка винесення рішень, аналогічна післясвятковим даним у січні, отже попередньо можна зробити висновок про істинність альтернативної гіпотези.

Розглянемо формальне доведення істинності альтернативної гіпотези. Для цього, дані кількості рішень, винесених у понеділок та п’ятницю, було об’єднано у дві вибірки. В якості додаткового метода візуалізації вибірок було застосовано boxplot (Рис. 2.13).

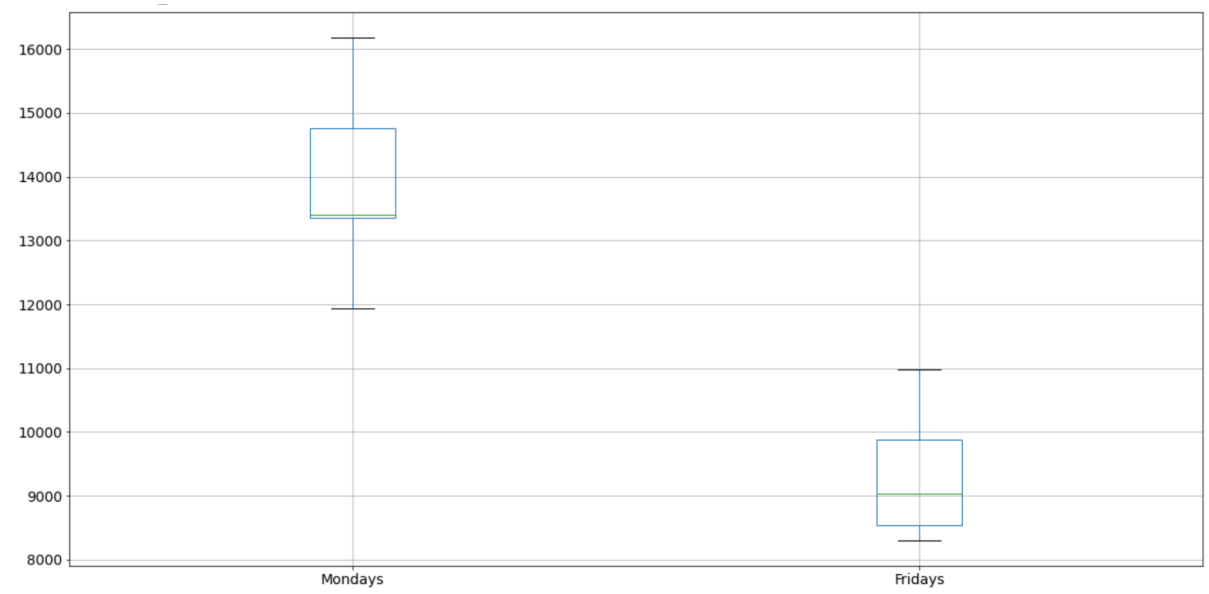


Рис. 2.13 Boxplot візуалізація вибірок кількості рішень на початку і в кінці тижня

Порівняння отриманих коробчастих діаграм лише підтверджує попередньо сформульоване припущення.

До вибірок було застосовано t-test Стьюдента для незалежних вибірок та критерій Манна-Уітні. P-value критеріїв становить 6.728557279081126e-13 та 1.6959106954125473e-06 відповідно і не перевищує рівень значущості $\alpha = 0.025$. Виходячи з цього, можна впевнено стверджувати, що нульова гіпотеза є хибною, тобто динаміка винесення судових рішень зменшується впродовж тижня.

Ще одним корисним методом аналізу є побудова розподілу рішень за складом суддівської колегії та вироком. В такий спосіб можна дослідити загальну тенденцію рішень, а також виявити суддів, результати роботи яких з неї виокремлюються. Для цього, попередньо оброблені документи було згруповано за двома атрибутами: склад суддів та вирок. Далі, використовуючи отриманий розподіл, обчислено різницю між кількістю задовільних та незадовільних вироків. Як і в попередньому прикладі,

результати обчислення візуально представлено у вигляді коробчастої діаграми (Рис. 2.14).

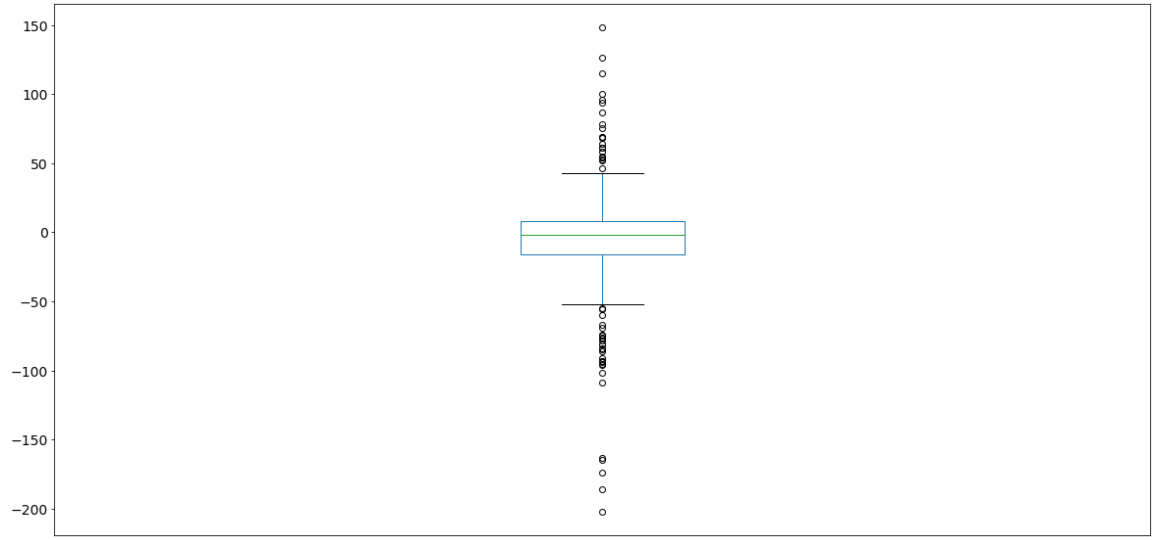


Рис. 2.14 Boxplot представлення різниці між задовільними та незадовільними вироками

На Рис. 2.14 простежується, що частина обрахованих значень виокремлюється з загальної тенденції. Викиди з додатним значенням параметру відповідають кількості задовільних вироків, що значно переважає кількість незадовільних. Для викидів з від’ємним значенням параметру справедливе зворотне твердження. На Рис. 2.15 наведено приклад знаходження суддів, вироків одного типу яких в декілька разів більше, ніж іншого.

```
[100] comparison[(comparison['2_x']/comparison['2_y'] > 5) | (comparison['2_y']/comparison['2_x'] > 5)]
```

	0	1_x	2_x	1_y	2_y
32	В.В. Хохуляк	задовольнити	16	не задовольнити	97
61	Л.О. Маруліна	задовольнити	16	не задовольнити	125
92	А.Ю. Кучма, Н.В. Безименна, В.О. Аліменко	задовольнити	12	не задовольнити	97
112	В.Г. Костенко	задовольнити	108	не задовольнити	14
150	Е.Г. Казначеєв, Л.В. Ястребова, І.Д. Компанієць	задовольнити	15	не задовольнити	109
194	Чаплицький В. В.	задовольнити	144	не задовольнити	18
197	О.П. Стародуб, В.М. Кравчук, Т.О. Анцупова	задовольнити	14	не задовольнити	200
210	В.М. Бевзенко, В.М. Шарапа, Н.А. Данилевич	задовольнити	16	не задовольнити	218
225	Т.О.Каращук	задовольнити	80	не задовольнити	11
270	Н.П. Побережна	задовольнити	92	не задовольнити	14
292	О.М. Чудак	задовольнити	20	не задовольнити	106
318	О.П. Стародуб, Т.О. Анцупова	задовольнити	13	не задовольнити	87
322	Іванчук В. М.	задовольнити	164	не задовольнити	16
350	І.Г. Бичков	задовольнити	135	не задовольнити	20
354	П.П. Чеберяк	задовольнити	72	не задовольнити	11
359	Т.М. Шипуліна, Л.І. Бившева, В.В. Хохуляк	задовольнити	25	не задовольнити	188

Рис. 2.15 Знаходження суддів, вироки яких значно виокремлюються з загальної тенденції

Отримані результати аналізу можна застосовувати для подальшого предметного дослідження результатів роботи конкретних представників суддівських колегій.

ВИСНОВКИ

В рамках кваліфікаційної роботи було детально розглянуто можливості та архітектуру двигуна для аналізу Apache Spark. Розподілена структура даних Resilient Distributed Dataset, якою оперує система, дозволяє забезпечити надійність та відмовостійкість при виконанні клієнтських процесів, що, в свою чергу, позитивно впливає на продуктивність роботи застосунку, а також дозволяє зменшити витрати на інфраструктуру при роботі з хмарними провайдерами.

Застосування розподіленої системи Apache Spark зробило можливим опрацювання кількості даних, що перевищує розмір доступної оперативної пам’яті на одній обчислювальній машині і дозволило ефективно виконати процес аналізу реєстру судових рішень.

Вирішення задачі попередньої обробки різнорідних даних продемонструвало ефективність запропонованого підходу. Отримані результати аналізу дозволили сформулювати декілька тверджень, що характеризують процес судочинства в Україні. Першим висновком став той факт, що переважна більшість винесених рішень має тип “ухвала”.

Наступні спостереження стосуються динаміки винесення рішень. Розглянувши кількісні дані впродовж декількох робочих тижнів була сформульована і доведена статистична гіпотеза про зниження темпу прийняття рішень впродовж тижня. Іншими словами, найефективнішими органи судочинства є на початку тижня, а з наближенням вихідних днів їх продуктивність стрімко знижується. Схожий висновок було зроблено, спостерігаючи дані кількості судових рішень у святковий період на початку року. В цей час, кількість рішень майже вдвічі менша, ніж у інші робочі дні, що, скоріш за все, зумовлено зростаючою кількістю відпусток серед працівників судової системи.

Проаналізувавши різницю задоволених та не задоволених позовів було виявлено загальну тенденцію та отримано перелік суддівських колегій, що з неї виокремлюються. Підсумки цього дослідження в подальшому можуть бути використані для фактичного розслідування результатів роботи суддів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Apache Spark
   https://spark.apache.org/
2. Jean-Georges Perrin “Spark in Action”, May 2020
3. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”, University of California, Berkeley, July 19, 2011.
4. Matei Zaharia “An Architecture for Fast and General Data Processing on Large Clusters”, University of California, Berkeley, February 3, 2014
5. Ramez Elmasri and Shamkant B. Navathe “Fundamentals of Database Systems (7th. ed.)”, June 2015.
6. Spark docker
   https://github.com/big-data-europe/docker-spark
7. Beautifulsoup4
   https://pypi.org/project/beautifulsoup4/
8. Єдиний державний реєстр судових рішень
   https://reyestr.court.gov.ua/
9. Google Cloud Storage
   https://cloud.google.com/storage
10. Google Cloud Dataproc
    https://cloud.google.com/dataproc