УДК 004.415.2
DOI https://doi.org/10.32782/2663-5941/2023.5/29

Олещенко Л.М.
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

ОСОБЛИВОСТІ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ТЕХНОЛОГІЇ SPARK ТА МОВИ ПРОГРАМУВАННЯ R ДЛЯ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ ВЕЛИКИХ ДАНИХ

У статті розглядаються особливості програмної реалізації технології Spark та мови програмування R для розподілених обчислень для набору даних з пакету nycflights13 середовища R, який містить дані з описом 336776 авіарейсів. Розглянуто такі особливості технології Spark, як швидкодія, підтримка інструментів для аналітики та масштабованість, наведено порівняння застосування технологій Hadoop та Spark для виконання ітеративних алгоритмів PageRank, K-Means та LR. Розподілені обчислення дозволяють ефективно обробляти та аналізувати великі обсяги даних, які перевищують можливості одного сервера. Розподілені системи забезпечують горизонтальне масштабування та працюють з великими обсягами даних, використовуючи кластери з кількома вузлами. Завдяки розподіленим обчисленням різні компанії та організації можуть використовувати потужність паралельної обробки даних та приймати рішення на основі оброблених даних вчасно. Розподілена природа таких систем забезпечує стійкість до відмов та резервування, забезпечуючи неперервність обробки даних у разі відмов апаратного або програмного забезпечення. В цілому, розподілені обчислення є необхідними для вирішення проблем, що виникають при роботі з великими обсягами даних, та дозволяють організаціям використовувати їх потенціал. Технологія Apache Spark є однією з найпопулярніших платформ для розподілених обчислень великих даних. Spark використовує пам'ять для проміжних обчислень, що значно прискорює виконання операцій з даними порівняно з іншими системами розподілених обчислень. Також Spark підтримує різні мови програмування, включаючи Java, Scala, Python та R, надає високорівневі API для роботи з даними, такі як розподілена колекція даних RDD та DataFrame. Технологія Spark включає модулі для машинного навчання Spark MLlib, обробки потокових даних Spark Streaming та графових обчислень GraphX, що дозволяє виконувати різноманітні завдання аналітики на одній платформі. У статті наведено приклади роботи Spark з кластерами різного розміру з автоматичним розподіленням обчислень, зберіганням даних, що дозволяє масштабувати обробку великих обсягів даних та забезпечувати стійкість до відмов та відновлення даних у випадку неполадок апаратного забезпечення або програмних помилок.

Ключові слова: *програмні засоби, великі дані, оброблення даних, аналіз, Spark, розподілені обчислення, DataFrame, мова програмування R, HiveQL.*

Постановка проблеми. Розподілені обчислення великих даних мають велике значення у сучасному світі з огляду на нашу здатність зберігати та генерувати все більші обсяги даних. Розподілені системи дозволяють ефективно обробляти та аналізувати великі обсяги даних, які перевищують межі єдиного сервера. Завдяки розподіленим обчисленням можна використовувати кластери з багатьма вузлами, що забезпечує горизонтальне масштабування і можливість працювати з великими обсягами даних. Розподілені обчислення дозволяють паралельно обробляти дані на багатьох вузлах кластера, що дозволяє збільшити швидкість обробки. Завдяки використанню оперативної пам'яті для зберігання даних (як у випадку з Spark), можна досягти ще більшої швидкодії.

Також розподілені системи дозволяють проводити аналітику та обробку даних в реальному часі, швидко аналізувати великі обсяги даних з метою виявлення складних залежностей та патернів. Це особливо важливо для сфер, де швидкість реакції має велике значення, наприклад, у фінансовому секторі, маркетингу, телекомунікаціях та інших галузях. Це допомагає виявляти цінну інформацію та робити прогнози для економії та розподілу ресурсів.

Метою статті є дослідження особливостей практичного застосування технології Spark та мови програмування R для розподілених обчислень великих даних.

Виклад основного матеріалу

Аналіз існуючих програмних рішень для розподілених обчислень великих даних

На сьогодні найбільш поширеними програмними рішеннями для розподілених обчислень

великих даних є Apache Hadoop, Apache Spark, Apache Flink та Apache Cassandra.

Apache Hadoop є одним з найпопулярніших фреймворків для обробки великих обсягів даних, що базується на моделі розподіленого обчислення MapReduce та системі файлів Hadoop Distributed File System (HDFS). Кластер HDFS складається з одного NameNode, головного сервера, який керує простором імен файлової системи та регулює доступ до файлів клієнтами. Крім того, існує кілька DataNodes, зазвичай по одному на вузол у кластері, які керують сховищем, підключеним до вузлів, на яких вони працюють.

HDFS відкриває простір імен файлової системи та дозволяє зберігати дані користувача у файлах. Внутрішньо файл розбивається на один або кілька блоків, і ці блоки зберігаються в наборі DataNodes.

NameNode виконує такі операції простору імен файлової системи, як відкриття, закриття та перейменування файлів і каталогів. NameNode також визначає відображення блоків у DataNodes. DataNodes відповідають за обслуговування запитів на читання та запис від клієнтів файлової системи (рис. 1). Hadoop надає масштабовану та надійну платформу для обробки великих даних на кластері серверів [1].

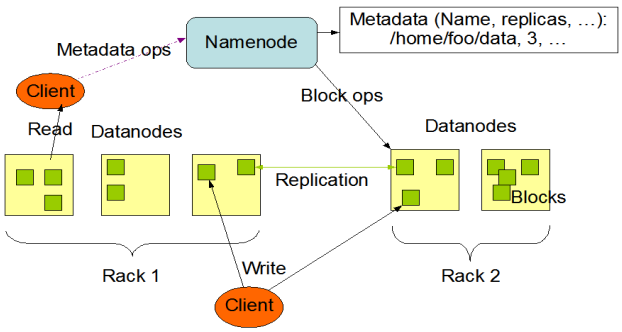

Рис. 1. Архітектура HDFS [2]

Apache Spark є відкритим фреймворком, який пропонує широкі можливості для розподіленого обчислення великих даних, включаючи обробку в пам'яті, підтримку багатьох мов програмування (Java, Scala, Python, R) та різні бібліотеки для аналізу даних, машинного навчання та графових обчислень (рис. 2). Зокрема, структура Spark включає Spark Core як основу для платформи, Spark SQL для інтерактивних запитів, Spark Streaming для аналітики в реальному часі, Spark MLlib для машинного навчання та Spark GraphX для обробки графів [3].

Однією з особливостей технології Spark для розподілених обчислень є її здатність до обробки даних in-memory. Spark зберігає дані в оперативній пам'яті (RAM) на кластері вузлів, що дозволяє значно прискорити обробку даних порівняно з традиційними системами, які зберігають дані на диску. Ця особливість Spark досягається завдяки своїй архітектурі Resilient Distributed Dataset (RDD). RDD є фундаментальним концептом Spark і представляє собою незмінний розподілений набір елементів даних, який може бути обчислений паралельно. RDD автоматично розподіляє дані між вузлами кластера і зберігає їх в оперативній пам'яті. Це дозволяє уникнути частих звернень до диску для доступу до даних і забезпечує швидку обробку даних. Ще однією особливістю Spark є можливість виконувати різні типи обчислень на одних і тих самих даних без необхідності зберігати дані в окремих системах. Spark надає багато вбудованих бібліотек для обробки даних, машинного навчання, графових обчислень тощо. Це дозволяє використовувати Spark для завдань обробки даних без необхідності переключатися між різними системами. Spark надає високу продуктивність і масштабованість для розподілених обчислень і обробки великих обсягів даних.

Apache Flink є фреймворком для потокової та пакетної обробки даних, надає потужні засоби для

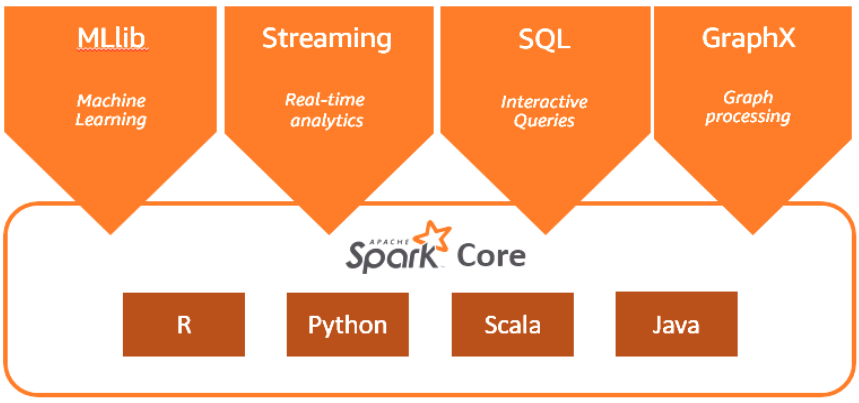

Рис. 2. Робочі навантаження Apache Spark [3]

розподіленого обчислення та аналізу великих обсягів даних в реальному часі (рис. 3). Flink підтримує потокову обробку даних (stream processing), комплексні аналітичні запити (complex event processing) та інші сценарії обробки даних [4].

Будь-які дані створюються як потік подій. Транзакції кредитних карток, вимірювання датчиків IoT, машинні журнали або взаємодії користувачів на вебсайті чи в мобільному додатку – усі ці дані генеруються як потік. Керування часом і станом дозволяє Flink запускати будь-які програми в необмежених потоках. Обмежені потоки внутрішньо обробляються за допомогою алгоритмів і структур даних, які спеціально розроблені для наборів даних фіксованого розміру, що забезпечує високу продуктивність (рис. 4).

Apache Cassandra є розподіленою системою управління базами даних (distributed database management system), яка спеціалізується на обробці великих обсягів даних та доступності. Cassandra дозволяє розподіляти дані на кластері серверів та забезпечує високу швидкодію та масштабованість для зберігання та операцій з даними [5].

Apache Hive – це розподілена відмовостійка система сховища даних, яка забезпечує аналітику у великому масштабі [6]. Hive Metastore (HMS) надає центральне сховище метаданих, які можна

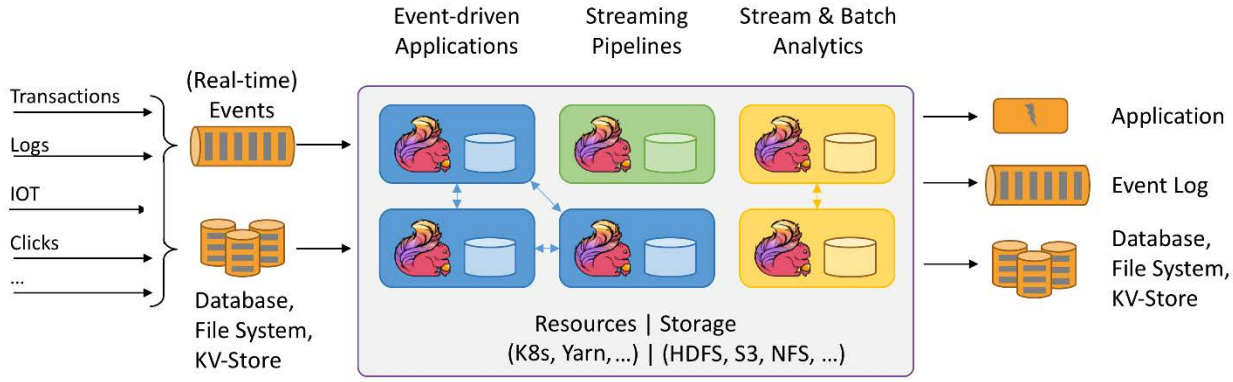

Рис. 3. Можливості Apache Flink [4]

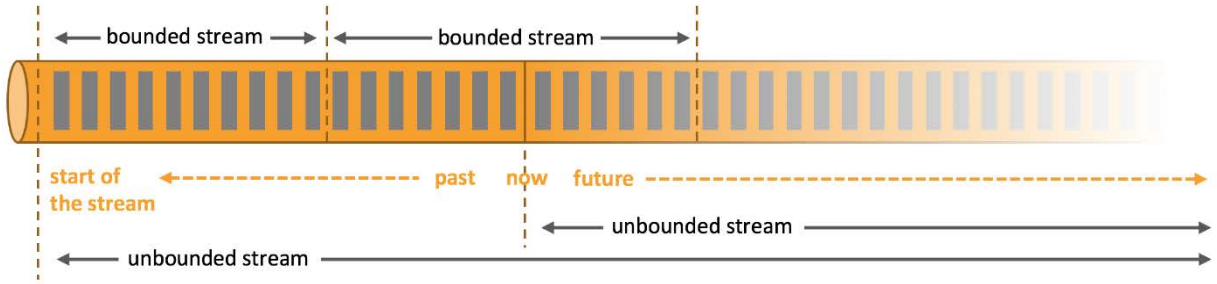

Рис. 4. Дані як необмежені або обмежені потоки [4]

Рис. 5. Порівняння застосування технологій Hadoop та Spark для виконання ітеративних алгоритмів PageRank, K-Means та LR [8]

проаналізувати для прийняття обґрунтованих рішень, керованих даними, і тому це критичний компонент багатьох архітектур централізованих сховищ даних Data Lake. Hive створено на основі Apache Hadoop і дозволяє користувачам читати, записувати та керувати петабайтами даних за допомогою SQL (HiveQL) [7].

У попередніх дослідженнях було проведене порівняння часу обробки даних для виконання ітеративних алгоритмів за допомогою технологій Hadoop та Spark (рис. 5). Згідно різних досліджень, Spark працює загалом приблизно в 100 разів швидше, ніж Hadoop [8-10].

Програмна реалізація розподілених обчислень великих даних

Програмна платформа Spark написана на Scala та Java і вимагає Java Virtual Machine (JVM), тому перед інсталяцією Spark потрібно переконатися, що на комп'ютері встановлено мову Java версії, не нижче 1.8, з середовища R це можна перевірити за допомогою виклику відповідної команди:

```
> system ("java -version")
java version "1.8.0_351"
Java(TM) SE Runtime Environment (build 1.8.0_351-b10)
Java HotSpot(TM) 64-Bit Server VM (build 25.351-b10, mixed mode)
[1] 0
```

Команда повернула нульовий код завершення програми, а отже, Java встановлена в системі. Як бачимо, Java має версію 1.8, що цілком відповідає вимогам платформи Spark. Після цього потрібно встановити пакет *sparklyr* для того, щоб працювати з кластером Spark, використовуючи середовище R. Підключимо даний пакет до поточного середовища. Встановимо версію Spark 3.2 за допомогою команди *spark_install()* з пакету *sparklyr*. За допомогою команди *spark_installed_versions()* перевіримо, яку версію Spark і Hadoop в результаті було встановлено на комп'ютері:

```
> install.packages("sparklyr")
package 'sparklyr' successfully unpacked and MD5 sums checked
> require(sparklyr)
Loading required package: sparklyr
Attaching package: 'sparklyr'
> spark_install("3.2")
Installing Spark 3.2.3 for Hadoop 3.2 or later.
Installation complete.
> spark_installed_versions()
```

Як можна побачити вище, пакет *sparklyr* було успішно завантажено, а платформа Spark версії 3.2.3 успішно інстальована, разом з платформою Hadoop версії 3.2. Підключимось до локального Spark-кластеру. Для цього використаємо функцію *spark_connect()* з пакету *sparklyr*:

```
> sc <- spark_connect(master = "local", version = "3.2")
```

Об'єкт, що повертається цією функцією, містить різного роду службову інформацію про кластер. У даному дослідженні було використано набір даних з пакету *nycflights13* середовища R, який містить таблиці з описом 336776 авіарейсів (рис. 6).

Рис. 6. Перегляд інформації в датасеті flights

Завантажимо таблицю *flights* в локальний Spark-кластер. Для цього використовується функція *copy_to()*, якій передається створений раніше об'єкт *sc* і таблиця *flights*. Результат відображений на рис. 7.

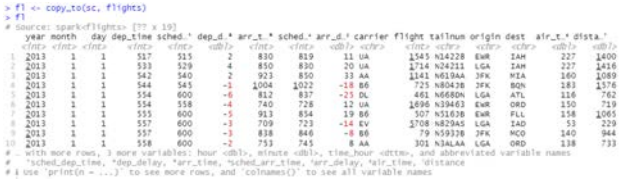

Рис. 7. Завантаження датасету flights в Spark-кластер

З'ясуємо загальну кількість рейсів з кожного аеропорту. Для цього виконаємо запит до даних в Spark-кластері з середовища R, використовуючи SQL-запит. Для виконання SQL-запиту було завантажено пакет *DBI* і використано функцію *dbGetQuery()*:

```
> require(DBI)
Loading required package: DBI
> q <- "SELECT `origin`, count(*) AS `N`
+   FROM `flights`
+   GROUP BY `origin`"
> dbGetQuery(sc, q)
  origin      N
1    JFK 111279
2    EWR 120835
3    LGA 104662
```

Як бачимо, з аеропорту JFK було виконано 111279 рейсів, з аеропорту EWR – 120835 рейсів, а з аеропорту LGA – 104662 рейсів.

Виконаємо описану вище операцію за допомогою функцій з пакета *dplyr*, а не SQL-запиту:

```
> true_result <- result %>% collect()
> true_result
# A tibble: 3 × 2
  origin      N
  <chr>   <dbl>
1 JFK    111279
2 EWR    120835
3 LGA    104662
> class(result)
[1] "tbl_spark" "tbl_sql"   "tbl_lazy"  "tbl"
> class(true_result)
[1] "tbl_df"      "tbl"         "data.frame"
```

Об'єкт *result* не є таблицею даних, він лише зберігає інформацію про те, як такі дані витягти з Spark-кластера. Для отримання даних з Spark в середовище R було використано функцію *collect()*:

```
> true_result <- result %>% collect()
> true_result
# A tibble: 3 x 2
  origin      N
  <chr>   <dbl>
1 JFK    111279
2 EWR    120835
3 LGA    104662
> class(result)
[1] "tbl_spark" "tbl_sql"   "tbl_lazy"  "tbl"
> class(true_result)
[1] "tbl_df"      "tbl"         "data.frame"
```

Після завершення роботи потрібно відключитися від кластера та видалити у ньому всі дані. Для цього була використана команда *spark_disconnect()*. Знову запускаємо локальний Spark-кластер з необхідними бібліотеками і завантажуємо в нього необхідні таблиці:

```
> require(sparklyr)
> require(nycflights13)
> sc <- spark_connect(master = "local", version = "3.2")
> flights_tbl <- copy_to(sc, flights, "flights")
> airlines_tbl <- copy_to(sc, airlines, "airlines")
```

Побудуємо модель, яка передбачає ймовірність прибуття затриманого рейсу без запізнення. Розглянемо це завдання як випадок бінарної класифікації: цікавий для нас відгук приймає значення 1, якщо затриманий рейс прибув без запізнення, і 0 у протилежному випадку.

Процес побудови прогнозних моделей включає кілька кроків, першим з яких є підготовка та розвідувальний аналіз даних. Розвідувальний аналіз може складатися з декількох кроків, таких як виявлення і усунення проблем з якістю аналізованих даних (пропущені спостереження, викиди тощо), розрахунок описових статистик, виявлення найбільш перспективних предикторів для подальшого включення в модель тощо.

Підрахуємо загальну кількість польотів, виконаних кожною авіакомпанією та виберемо 5 авіакомпаній з найбільшим числом польотів.

```
> require(dplyr)
> flights_tbl %>%
+   group_by(carrier) %>%
+   summarise(N = n()) %>%
+   arrange(desc(N)) %>%
+   head(5)
# Source:     spark<?> [?? x 2]
# Ordered by: desc(N)
  carrier     N
  <chr>   <dbl>
1 UA      58665
2 B6      54635
3 EV      54173
4 DL      48110
5 AA      32729
```

Отже, наступні 5 авіакомпаній мають найбільшу кількість польотів за 2013 р.: UA (58665), B6 (54635), EV (54173), DL (48110), AA (32729).

За допомогою функції *show_query()* з пакету dplyr переглянемо SQL-запит, який був сформований з попередньої команди мовою R:

```
> flights_tbl %>%
+   group_by(carrier) %>%
+   summarise(N = n()) %>%
+   arrange(desc(N)) %>%
+   head(5) %>%
+   show_query()
<SQL>
SELECT `carrier`, COUNT(*) AS `N`
FROM `flights`
GROUP BY `carrier`
ORDER BY `N` DESC
LIMIT 5
```

Виконаємо LEFT JOIN таблиці flights_tbl з таблицею airlines_tbl за полем carrier. В результаті отримали нову колонку name, яка дає повну назву авіакомпанії. Результат показаний на рис. 8.

```
Rows: ??
Columns: 20
Database: spark_connection
$ year          <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, …
$ month         <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, …
$ day           <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, …
$ dep_time      <int> 629, 554, 602, 606, 615, 544, 555, 557, 558, 558, 559, 600, 601, 613, 615, 557, 622, 624, 627, 5…
$ sched_dep_time <int> 630, 600, 610, 610, 615, 545, 600, 600, 600, 600, 559, 600, 600, 610, 615, 600, 630, 630, 630, 5…
$ dep_delay     <db1> -1, -6, -8, -4, 0, -1, -5, -3, -2, -2, 0, 0, 1, 3, 0, -3, -8, -6, -3, 2, 4, -4, -2, -2, -1, 0, …
$ arr_time      <int> 721, 812, 812, 837, 833, 1004, 913, 838, 849, 853, 702, 851, 844, 925, 1039, 709, 1017, 909, 101…
$ sched_arr_time <int> 740, 837, 820, 845, 842, 1022, 854, 846, 851, 856, 706, 858, 850, 921, 1100, 723, 1014, 840, 101…
$ arr_delay     <db1> -19, -25, -8, -8, -9, -18, 19, -8, -2, -3, -4, -7, -6, 4, -21, -14, 3, 29, 0, 11, 20, 12, 7, -14…
$ carrier       <chr> "VW", "DL", "DL", "DL", "DL", "B6", "B6", "B6", "B6", "B6", "B6", "B6", "B6", "B6", "B6", "EV", …
$ flight        <int> 4646, 461, 1919, 1743, 575, 725, 507, 79, 49, 71, 1806, 371, 343, 135, 709, 5708, 245, 4626, 27,…
$ tailnum       <chr> "N273WN", "N668DN", "N971DL", "N3739P", "N326NB", "N804JB", "N516JB", "N593JB", "N793JB", "N657J…
$ origin        <chr> "LGA", "LGA", "LGA", "JFK", "EWR", "JFK", "EWR", "JFK", "JFK", "JFK", "JFK", "LGA", "EWR", "JFK"…
$ dest          <chr> "BWI", "ATL", "MSP", "ATL", "ATL", "BQN", "FLL", "MCO", "PBI", "TPA", "BOS", "FLL", "PBI", "RSW"…
$ air_time      <db1> 40, 116, 170, 128, 120, 183, 158, 140, 149, 158, 44, 152, 147, 175, 182, 53, 342, 190, 330, 227,…
$ distance      <db1> 185, 762, 1020, 760, 746, 1576, 1065, 944, 1028, 1005, 187, 1076, 1023, 1074, 1598, 229, 2133, 1…
$ hour          <db1> 6, 6, 6, 6, 6, 5, 6, 6, 6, 6, 5, 6, 6, 6, 6, 6, 6, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, …
$ minute        <db1> 30, 0, 10, 10, 15, 45, 0, 0, 0, 0, 59, 0, 0, 0, 10, 15, 0, 30, 30, 30, 15, 29, 58, 0, 0, 0, 0, 5, 7…
$ time_hour     <dttm> 2013-01-01 13:00:00, 2013-01-01 13:00:00, 2013-01-01 13:00:00, 2013-01-01 13:00:00, 2013-01-01 …
$ name          <chr> "Southwest Airlines Co.", "Delta Air Lines Inc.", "Delta Air Lines Inc.", "Delta Air Lines Inc.", "…
```

Рис. 8. Результат LEFT JOIN таблиці flights_tbl з таблицею airlines_tbl

Для обчислення медіанного значення затримки рейсу у хвилинах з таблиці *flights_tbl* не можна просто скористатися базовими функціями R, тому було застосовано Hive-функцію *percentile()*:

```
> flights_tbl %>%
+   summarise(median = percentile(dep_delay, 0.5))
# Source: spark<?> [?? x 1]
  median
   <dbl>
1     -2
```

Отже, медіанне значення затримки рейсу становить 2 хвилини, тобто відправлення на 2 хв. раніше, ніж за розкладом. За допомогою функції *show_query()* переглянемо SQL-запит, який був сформований з попередньої команди мовою R:

```
> flights_tbl %>%
+   summarise(median = percentile(dep_delay, 0.5)) %>%
+   show_query()
<SQL>
SELECT percentile(`dep_delay`, 0.5) AS `median`
FROM `flights`
```

Можна побачити, що при перекладі коду R на SQL *dplyr* незнайому функцію включає в SQL-запит «як є», внаслідок чого Spark може виконати дану операцію. Hive-функція *percentile()* дозволяє одночасно обчислити кілька процентилей. Для цього в аргументах до цієї функції було передано масив *array()* з необхідними значеннями процентилей:

```
> flights_tbl %>%
+   summarise(perc = percentile(dep_delay, array(0.25, 0.5, 0.75)))
# Source: spark<?> [?? x 1]
  perc
  <list>
1 <dbl [3]>
```

Як бачимо, результатом виконання вищенаведеної команди є список з трьома значеннями. Щоб автоматично отримати ці значення зі списку, було використано Hive-функцію *explode()*:

```
> flights_tbl %>%
+   summarise(perc = percentile(dep_delay, array(0.25, 0.5, 0.75))) %>%
+   mutate(perc = explode(perc))
# Source: spark<?> [?? x 1]
   perc
  <dbl>
1    -5
2    -2
3    11
>
```

Як результат, для 25% процентиля маємо запізнення – 5 хв., а для 75% маємо 11 хв. Здійснимо розвідувальний аналіз даних в таблиці flights_tbl. Для початку з'ясуємо, чи є в наших даних пропущені значення. Виконаємо підрахунок пропущених значень для всіх стовпців таблиці flights_tbl за допомогою команди *summarise_each()* з пакету *dplyr* в поєднанні з анонімною функцією, яка задає логіку обчислень:

```
> flights_tbl %>%
+   summarise_each(list(~sum(as.integer(is.
na(.))))) %>%
+ glimpse
Rows: ??
Columns: 19
Database: spark_connection
$ year <dbl> 0
$ month <dbl> 0
$ day <dbl> 0
$ dep_time <dbl> 8255
$ sched_dep_time <dbl> 0
$ dep_delay <dbl> 8255
$ arr_time <dbl> 8713
$ sched_arr_time <dbl> 0
$ arr_delay <dbl> 9430
$ carrier <dbl> 0
$ flight <dbl> 0
$ tailnum <dbl> 2512
$ origin <dbl> 0
$ dest <dbl> 0
$ air_time <dbl> 9430
$ distance <dbl> 0
$ hour <dbl> 0
$ minute <dbl> 0
$ time_hour <dbl> 0
Warning messages:
1: 'summarise_each_()' was deprecated in
dplyr 0.7.0.
This warning is displayed once every 8 hours.
Call 'lifecycle::last_lifecycle_warnings()'
to see where this warning was generated.
2: Missing values are always removed in SQL
aggregation functions.
Use 'na.rm = TRUE' to silence this warning
This warning is displayed once every 8 hours.
```

Отже, для стовпців dep_time, dep_delay, arr_time, arr_delay, tailnum, air_time маємо пропущені значення в різних кількостях.

Максимальна кількість пропущених значень становить 9430 в стовпцях arr_delay та air_time. Всього в таблиці 336776 записів. 9430 від 336776 це 2.8%. Тобто максимальна кількість пропущених значень від загального числа спостережень становить 2.8%. Оскільки частка пропущених значень невелика, ми можемо видалити відповідні рядки з таблиці без особливого ризику вплинути на якість подальшого аналізу. Для цього було використано базову функцію *na.omit()*:

```
> flights_full <- flights_tbl %>% na.omit()
* Dropped 9430 rows with 'na.omit' (336776 => 327346)
```

Тож, було видалено 9430 рядків з пропущеними значеннями, загальна кількість рядків в таблиці з 336776 зменшилась до 327346. Знайдемо розмірність нової таблиці після вищенаведених операцій:

```
> flights_full %>% sdf_dim()
[1] 327346     19
```

Відповідно, маємо 19 стовпців та 327346 рядків. Оскільки нас цікавлять рейси, затримка яких склала від 15 до 30 хв. включно, то далі нам потрібно відфільтрувати дані відповідним чином. Додамо новий стовпець target зі значеннями залежної змінної:

```
> flights <- flights_full %>%
+   filter(dep_delay >= 15, dep_delay <= 30) %>%
+   mutate(target = as.integer(arr_delay <= 0))
> flights %>% sdf_dim()
[1] 24515    20
```

Отже, отримано таблицю з розмірністю в 20 стовпців та 24515 рядків. Спробуємо з'ясувати, які з наявних змінних корелюють із залежною змінною target. Логічно очікувати, що ймовірність прибуття затриманого рейсу за розкладом в значній мірі визначається відстанню між аеропортом вильоту і аеропортом прибуття (стовпець distance, який виражається в милях). Розрахуємо медіанне значення цієї відстані для обох класів залежною змінною:

```
> flights %>%
+   group_by(target) %>%
+   summarise(median_dist = percentile(distance, 0.5))
# Source: spark<?> [?? x 2]
  target median_dist
   <int>       <dbl>
1      1        1089
2      0         820
```

Чим більша відстань між аеропортами, тим більше шанс у затриманого рейсу надолужити згаяний час і прибути без запізнення. Можливість надолужити згаяний час може визначатися різними факторами, пов'язаними з авіакомпанією, яка виконує той чи інший рейс (досвід пілотів, характеристики літака тощо). Тому категоріальна змінна carrier, що містить скорочені назви авіакомпаній, також може виявитися корисним предиктором.

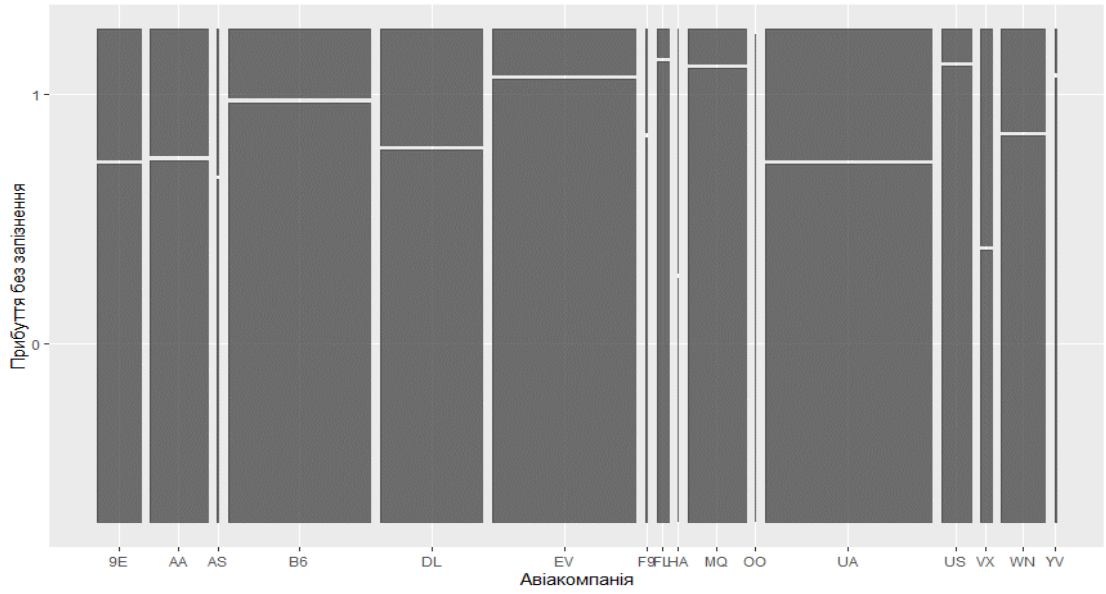

Рис. 9. Мозаїчна діаграма залежності затримки рейсів від авіакомпанії

Щоб зрозуміти, чи це так, згрупуємо дані за рівнями змінної carrier і підрахуємо кількість затриманих рейсів, які прибули із запізненням та без запізнень. Візуалізуємо дані з отриманої таблиці спряженості за допомогою мозаїчної діаграми, яка показана на рис. 9.

Для цього спочатку з середовища R надсилаємо інструкції для виконання обчислень у Spark-кластері, а потім імпортуємо отриманий результат в середовище R і вже продовжуємо аналіз звичайними для R засобами, у даному випадку – зображуємо дані графічно за допомогою *ggplot2* і *ggmosaic*:

```
> flights %>%
+   group_by(target, carrier) %>%
+   tally() %>%
+   collect %>%
+   ggplot() +
+   geom_mosaic(aes(product(target, carrier), weight = n)) +
+   labs(x = "Авіакомпанія", y = "Прибуття без запізнення")
```

Як бачимо вище, авіакомпанії можуть відрізнятися за часткою рейсів, які прибули без запізнення, в зв'язку з чим змінну carrier можна розглядати як потенційно корисний предиктор.

Висновки та подальша робота. У статті наведено особливості програмної реалізації технології Spark для розподілених обчислень з використанням мови програмування R. У даному дослідженні було встановлено Spark на локальній машині та виконано розподілені обчислення для набору даних пакету nycflights13 з використанням Spark-кластера у середовищі R як за допомогою запитів HiveQL, так і за допомогою можливостей бібліотеки R dplyr. Отримано дані після їх обробки в Spark кластері для виконання подальших обчислень, виконано розвідувальний аналіз засобами мови R для даних в датасеті, який містить таблиці з описом 336776 авіарейсів. Задача розподілених обчислень великих даних полягала у побудові моделі, яка передбачає ймовірність прибуття затриманого рейсу без запізнення. В рамках розвідувального аналізу було вилучено рядки з пропущеними значеннями. Дані були відфільтровані відповідно до умови задачі та додано змінну, яка позначає бінарну класифікацію відносно запізнення рейсу. Було встановлено, що відстань маршруту та авіакомпанія корелюють з тим, чи запізниться рейс. Для цього було обраховано медіану та побудовано мозаїчну діаграму. У подальшому планується дослідження аналогічним чином перспективних на думку автора програмних методів оброблення великих даних та прогнозування в режимі реального часу з використанням технологій та методів машинного навчання.

Список літератури:

1. Apache Hadoop. https://hadoop.apache.org/
2. HDFS Architecture. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html
3. Apache Spark. https://aws.amazon.com/big-data/what-is-spark/
4. Apache Flink. https://flink.apache.org/
5. Apache Cassandra. https://cassandra.apache.org/doc/latest/

6. Apache Hive. https://hive.apache.org/
7. Jesús Camacho-Rodríguez, et al. Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing. International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands, p. 1773–1786, https://doi.org/10.1145/3299869.3314045.
8. Samadi Y., Zbakh M., Tadonki C. Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks. Concurrency Computat: Pract Exper. 2017; 43-67. https://doi.org/10.1002/cpe.4367
9. Gu L., Li H. Memory or time performance evaluation for iterative operation on Hadoop and Spark. In: IEEE 10th International Conference on HighPerformance Computing and Communications; 2013; Zhangjiajie.721-727.
10. Lin X, Wang P, Wu Log B. Analysis in cloud computing environment with Hadoop and Spark. In: 5th IEEE International Conference on Broadband Network and Multimedia Technology; November 2013; Guilin China. 273-276.

Oleshchenko L.M. SOFTWARE IMPLEMENTATION FEATURES OF SPARK TECHNOLOGY AND R PROGRAMMING LANGUAGE FOR BIG DATA DISTRIBUTED COMPUTING

The article discusses the features of the software implementation of Spark technology and the R programming language for distributed computing for the data set from the nycflights13 package of the R environment, which contains data with a description of 336,776 flights. Such features of the Spark technology as speed, support for analytics tools and scalability are considered, and a comparison of the use of Hadoop and Spark technologies for the implementation of iterative PageRank, K-Means and LR algorithms is given. Distributed computing allows efficient processing and analysis of large volumes of data that exceed the capabilities of a single server. Distributed systems enable horizontal scaling and work with large amounts of data using multi-node clusters. With distributed computing, organizations can leverage the power of parallel processing for complex calculations and make data-driven decisions in a timely manner. The distributed nature of such systems provides fault tolerance and redundancy, ensuring continuity of data processing even in the event of hardware or software failures. In general, distributed computing is necessary to solve the problems that arise when working with large volumes of data and allows organizations to use their potential. Apache Spark technology is one of the most popular platforms for distributed big data computing. Spark technology uses memory for intermediate calculations, which significantly speeds up data operations compared to other distributed computing systems. Spark also supports various programming languages, including Java, Scala, Python, and R, and provides high-level APIs for working with data, such as RDD (Resilient Distributed Dataset) and DataFrame. Spark includes modules for machine learning (Spark MLlib), streaming data processing (Spark Streaming), and graph computing (GraphX), which allows perform a variety of analytics tasks on a single platform. The article provides examples of how Spark works on clusters of various sizes with automatic computation distribution and data storage, which allows for easy scaling of processing large volumes of data. Spark provides mechanisms for fault tolerance and data recovery in the event of hardware failures or software errors.

Key words: *software, big data, data processing, analysis, Spark, distributed computing, DataFrame, R programming language, HiveQL.*