Висновок до лекції 16

Розподілена потокова платформа Kafka відрізняється від традиційних посередників повідомлень використанням журналів транзакцій для передачі потокових даних у режимі реального часу між різними системами та програмами.

Cassandra – це система розподілених баз даних з відкритим кодом NoSQL. Cassandra використовує файлову систему Cassandra File System (CFS). Кожен вузол кластера має однакову однорангову реалізацію. Кластери зберігають дані в реальному часі і аналітичні операції виконуються на цих даних.

Питання для закріплення

1. Які проблеми прийому даних ви знаєте?
2. Для чого використовується Kafka?
3. Яке призначення та переваги Cassandra?

Список рекомендованої літератури

IoT Fundamentals: Big Data & Analytics // Електронний ресурс. Режим доступу: https://www.netacad.com/courses/iot/big-data-analytics

Рівні узгодженості Apache Cassandra для розподіленої обробки Big Data // Електронний ресурс. Режим доступу: https://www.bigdataschool.ru/blog/cassandra-consistency-levels.html

What is Cassandra? // Електронний ресурс. Режим доступу: https://cassandra.apache.org/

About the Cassandra File System (CFS) – deprecated // Електронний ресурс. Режим доступу: https://docs.datastax.com/en/dse/5.1/dse-dev/datastax_enterprise/ analytics/ cfsAbout.html

Лекція 17.
Платформа Apache Spark

План лекції

17.1. Проблема обчислювальної функції.
17.2. Технологія Spark.
17.3. Порівняння Spark та MapReduce.
17.4. Spark і sparklyr для роботи з великими даними в R.

17.1. Проблема обчислювальної функції

Важливим завданням, з яким стикаються обчислення Big Data, є розмір наборів даних, що використовуються у різних галузях. Наприклад, генетичне секвенування (цифрове представлення геному людини) в останні роки стало

об’єктом дослідження наукової спільноти. Це пов’язано з тим, що високоефективні обчислення (high-performance computing, HPC) у поєднанні з розвитком технологій секвенування дали змогу за 26 годин секвенувати увесь геном людини. Послідовність усього геному людини становить близько 200 ГБ даних і вимагає величезної кількості обчислювальної потужності для роботи з нею. Кількість даних та їх аналіз у майбутньому може перевищити найпотужніший HPC. Коли це трапляється, HPC потрібно або масштабувати, додаючи на комп'ютер більше процесорів і пам'яті, або масштабувати, додаючи в кластер більше комп'ютерів та з'єднуючи їх швидкісними з'єднаннями. Коли обсяг обробки даних величезний, часті переміщення даних можуть значно збільшувати затримку. Бажано мати обчислювальну систему, яка працює для подолання обмежень системи зберігання, щоб звести затримку до мінімуму.

Аналітика, проведена на Big Data, може запропонувати нові можливості та непередбачувані тенденції, забезпечивши кращий огляд клієнтів та ринку загалом. Точна аналітика клієнтів, виявлення шахрайства та аналіз ризиків – це користь від аналізу великих даних. Ці складні обчислення потребують не тільки великих сховищ даних, але й низької затримки, високопропускної спроможності потокової обробки. Це ще більше збільшує розмір та складність аналітики HPC.

Щоб зменшити затримку від частих операцій вводу / виводу даних, обчислення можна перемістити на сервер, на якому розміщені дані. У багатьох місцях виконується кілька невеликих завдань для розподілу навантаження. Модель MapReduce може певною мірою зменшити велику кількість вузьких місць вводу-виводу та мереж, але це не головна мета MapReduce.

Затримка обчислень – це також проблема великих даних. Обчислення великих даних розділяється, щоб виконувати конкретні обчислення на різних вузлах. Коли один вузол працює повільніше, ніж інші, час відгуку збільшується. Це змушує результати загальної роботи чекати, коли цей вузол виконає свою частину, перш ніж їх можна буде реалізувати. Затримка є важливою проблемою у великих центрах обробки даних, що спричиняє значний вплив на обчислення, залежні від часу, наприклад, на потоках даних.

Також графік роботи може значно збільшити затримку. Часто великі обсяги роботи виконуються, тоді як виконуються інші, менші обсяги роботи.

Менші завдання, які повинні зачекати виконання великих завдань, можуть спричинити затримку. Це може бути проблемою, коли деякі завдання потрібно обробляти в режимі реального часу.

17.2. Технологія Spark

Spark – це відкритий, розповсюджений механізм обробки даних із відкритим кодом, який використовується для великих даних. Хоча платформа Hadoop дозволила багатьом компаніям успішно застосовувати парадигму MapReduce для розподілених обчислень величезних обсягів даних, кожен раз при виникненні нового завдання потрібно написання нового коду для операцій map і reduce, що є незручним і займає багато часу.

Для вирішення цієї проблеми в 2008 р. інженери з Facebook створили Hive – систему управління базами даних на основі Hadoop. Головною особливістю Hive стала підтримка SQL -подібних запитів до даних, що зберігаються в HDFS (цей новий діалект SQL отримав назву Hive Query Language, HQL).

У 2009 р. в Каліфорнійському університеті в Берклі був запущений дослідницький проект Spark з метою підвищити ефективність розподілених обчислень методом MapReduce і створити універсальну платформу для таких обчислень. У 2010 р. Spark був опублікований як проект з відкритим кодом, а в 2013 р. переданий фонду Apache Software Foundation. Spark використовує кешування в пам'яті для досягнення швидкої продуктивності, обмежуючи читання та запис дисків.

Apache Spark – фреймворк з відкритим вихідним кодом для реалізації розподіленої обробки неструктурованих і слабоструктурованих даних, що входить в екосистему проектів Hadoop. На відміну від класичного обробника з ядра Hadoop, що реалізує дворівневу концепцію MapReduce зі зберіганням проміжних даних на накопичувачах, Spark працює згідно парадигми резидентних обчислень (In-memory computing), обробляє дані в оперативній

пам'яті, завдяки чому дозволяє отримувати значний виграш в швидкості роботи для деяких класів задач, зокрема, можливість багаторазового доступу до завантажених в пам'ять даних робить бібліотеку привабливою для алгоритмів машинного навчання.

Spark може використовувати HDFS та досягати кращих показників, ніж MapReduce. Spark підтримує такі мови програмування, такі як Python, Scala, R та Java. Spark також підтримує структуровані бази даних на SQL. Завдяки цій різноманітності підтримуваних мов та систем можна реалізувати багато різних рішень Spark.

Spark включає бібліотеки, які допомагають створювати різні типи програм:

- **Shark SQL** – бібліотека SQL, яка ефективно підтримує складну аналітику, залишаючись стійкою до відмов.
- **Spark Streaming** – бібліотека обробки потоків, яка масштабується, підтримує високу пропускну здатність та підтримує відмовостійкість.
- **MLib** – масштабована, високоефективна бібліотека машинного навчання.
- **GraphX** – бібліотека, яка містить алгоритми теорії графів.

Однією з причин того, що Spark є настільки швидким, є те, що він використовує стійкі набори розподілених даних (Resilient Distributed Datasets, RDD) для зберігання вхідних, вихідних та проміжних даних у пам'яті замість диска. Це виключає вартість вводу-виводу. RDD є стійкими, тому що вони відстежують історію, якщо їм доведеться реконструювати себе після відмови. Вони розподіляються, тому що вони розповсюджені на багатьох вузлах кластера. Це дозволяє забезпечити надмірність, а також підвищити ефективність завдяки паралельній обробці. Коли запит надходить із програми, Spark створює та завантажує дані в RDD. Дані можуть надходити з будь-якого джерела, включаючи HDFS, CFS, AWS S3 або навіть базу даних SQL. Після створення RDD Spark може виконувати два різні типи операцій на RDD (рис.17.1):

- Трансформації – це будь-яке маніпулювання даними, включаючи відображення чи фільтрування. Будь-яка трансформація спричинить створення нового RDD. Оригінальний RDD не змінюється. Це дозволяє Spark здійснювати

контроль версій на RDD. Перетворення виконуються лише тоді, коли вони потрібні, наприклад, дією.

• Дії – дії взаємодіють із даними, але не створюють змін. Приклади включають агрегування, підрахунок або отримання конкретного елемента в RDD. Після запиту про дію створюється новий RDD, за яким слід виконати дію.

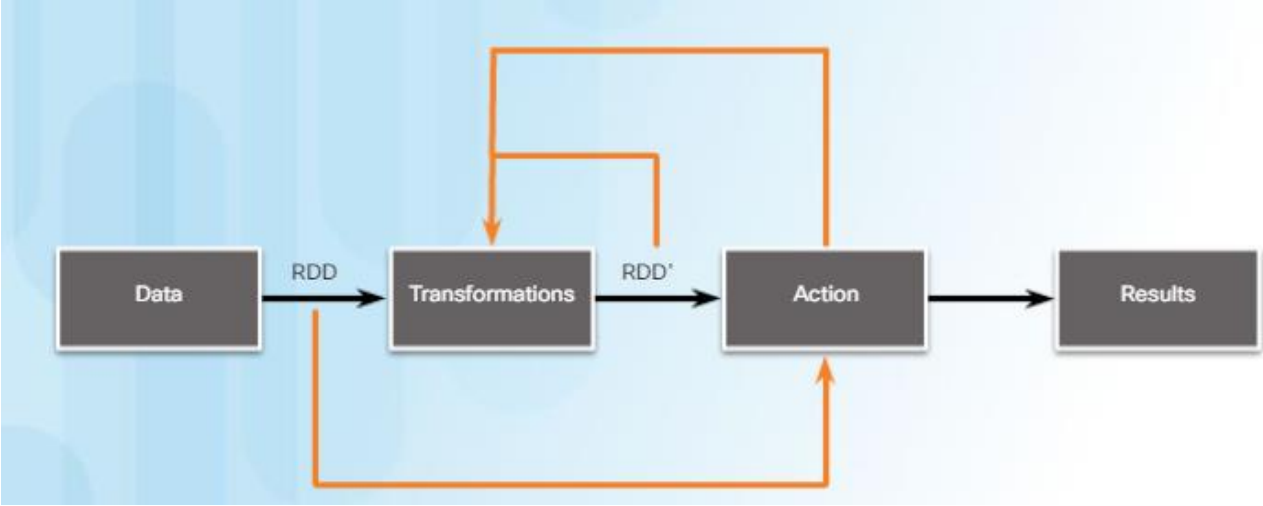

Рис. 17.1. Потік Spark [1]

17.3. Порівняння Spark та MapReduce

Spark може працювати безпосередньо над екземпляром Hadoop, використовуючи HDFS для зберігання та YARN для управління кластером.

Spark не потребує використання Hadoop. Можна використовувати інші рішення для зберігання, такі як CFS або AWS S3, а також інші менеджери кластерів, такі як Mesos.

Spark – це незалежна платформа, оскільки вона підтримує багато різних технологій та мов програмування. Це корисно, оскільки існує дуже багато різних потреб, коли йдеться про рішення та аналіз великих даних у багатьох різних сферах.

Підхід до рішення великих даних полягає у виборі правильних інструментів для роботи. Іноді поєднання Spark та MapReduce є найкращим рішенням для конкретної роботи. Використання Hadoop з MapReduce все ще є життєздатним при виконанні пакетної обробки за допомогою програми, яка використовує

виключно HDFS, або існуючі програми, які використовують цю технологію, вже у виробництві.

Spark набув великої популярності завдяки своїй продуктивності, простоті адміністрування, що при його використанні програмне забезпечення можна створювати швидше.

Розглянемо кілька причин використовувати Spark замість MapReduce при створенні програмних рішень для великих даних.

- Потокове передавання даних – Spark здатний обробляти величезні обсяги даних у режимі реального часу. Наприклад, дані можуть надходити з мобільних пристроїв, соціальних мереж або датчиків IoT.
- Неоднорідні дані – багато рішень для великих даних отримують дані з різних джерел.
- Машинне навчання – завдяки вбудованій бібліотеці машинного навчання Spark може задовольнити набагато більшу аудиторію, ніж попередні рішення.
- Додатки в режимі реального часу – обробка в пам'яті дозволяє Spark повертати результати обчислень набагато швидше, ніж MapReduce. Це важливо в усіх ситуаціях, але обов’язково, коли програма вимагає результатів у реальному часі.
- Менше коду – Spark підтримує багато різних мов програмування, а це означає, що менше коду потрібно писати та підтримувати.
- Досвід розробника – вивчати Spark набагато простіше, ніж MapReduce. Це набагато швидше приносить надійніший код для проекту.

17.4. Spark і sparklyr для роботи з великими даними в R

В екосистемі R є багато пакетів, що дозволяють користувачам працювати з віддаленими базами даних і виконувати масштабовані розподілені обчислення за допомогою призначених для цього програмних платформ, або фреймворків. Усі обчислення в системі R виконуються в оперативній пам'яті комп'ютера. Обсяг багатьох сучасних наборів даних (наприклад, зібраних

високонавантаженими веб-додатками і промисловими системами) набагато перевищує розмір оперативної пам'яті, доступний на окремому комп'ютері. Такі дані зазвичай зберігаються у віддаленій базі даних або в сховищі іншого типу. Залежно від завдання, для обробки подібних даних за допомогою R можна скористатися однією з наступних стратегій.

1. Створення репрезентативної вибірки обмеженого розміру. Практично усі класичні методи статистики припускають, що дослідник працює з репрезентативною вибіркою з деякою генеральною сукупністю. Тому для застосування таких методів цілком можна випадковим чином вибрати кілька тисяч або навіть сотень тисяч записів з бази даних, а потім локально виконати необхідний аналіз на комп'ютері користувача.

Такий підхід особливо корисний на початкових стадіях проекту, коли потрібно швидко ознайомитися з властивостями даних або створити прототип моделі. При цьому користувачеві буде доступно все розмаїття існуючих для R додаткових пакетів. Недолік цього підходу полягає в тому, що іноді створити репрезентативну вибірку буває складніше, ніж здається. Працюючи ж з вибіркою, дослідник ризикує зробити невірні висновки щодо властивостей генеральної сукупності.

2. Розбиття даних на частини. Рішення ряду завдань передбачає обробку логічно розділених наборів даних, наприклад, відповідних певним часовим періодам, окремим географічним областям, компаніям, користувачам тощо. Такі набори можна легко витягти з бази даних і далі послідовно (іноді паралельно) обробити за допомогою R на локальному комп'ютері дослідника. У цій стратегії аналізу піддаються усі наявні дані. Однак ці дані повинні бути нероздільні на окремі частини, що не завжди можливо або не завжди має сенс. Більш того, в залежності від обсягу даних, такий підхід може зайняти занадто багато часу і обчислювальних ресурсів.

3. Виконання ресурсномістких обчислень на стороні бази даних. Часто перед виконанням статистичного аналізу або побудовою прогнозної моделі до даних необхідно застосувати такі операції, як фільтрація, групування, агрегування

тощо. Більшість баз даних чудово справляються з такими операціями і тому має сенс спочатку зробити подібні обчислення саме на стороні бази даних, а потім вже завантажити отриманий набір даних меншого розміру на комп'ютер користувача і провести його подальшу обробку за допомогою R.

До цієї стратегії варто віднести також можливість побудови деяких прогнозних моделей з використанням обчислювальних ресурсів бази даних. Наприклад, пакет **modeldb** дозволяє таким чином створювати моделі лінійної регресії. Тут же варто згадати і спроби деяких компаній реалізувати можливість виконання R-скриптів повністю на стороні бази даних. Зокрема, така можливість є в Microsoft SQL Server. Залежно від використовуваної бази даних, необхідна користувачу функціональність іноді може бути недоступна. Більш того, не всі бази даних однаково ефективні: виконання ресурсоємних запитів може привести до істотного уповільнення деяких з них, що, в свою чергу, може викликати і інші небажані наслідки (наприклад, уповільнення роботи веб-сайту, який обслуговується подібною базою даних).

4. Використання спеціалізованих програмних платформ для роботи з великими даними. Якщо описані вище стратегії з тих чи інших причин не підходять, варто звернутися до спеціалізованих програмних платформ, призначених для організації і виконання розподілених обчислень над великими даними. Найбільш відомими і широко використовуваними серед таких платформ є Hadoop і Spark (обидві входять до складу проектів фонду Apache Software Foundation і тому їх також часто називають Apache Hadoop і Apache Spark).

В R є кілька пакетів (зокрема, **sparklyr**), що надають зручний інтерфейс для роботи з цими платформами. Spark є однією з найбільш використовуваних платформ для роботи з великими даними, яка характеризується швидкодією за рахунок виконання обчислень в оперативній пам'яті великої кількості комп'ютерів, об'єднаних в один кластер, а також завдяки ефективним протоколам передачі даних по мережі. Пакет **sparklyr** надає зручний інтерфейс для роботи зі Spark-кластерами з середовища R. Зокрема, з його допомогою можна встановлювати з'єднання з кластером; виконувати операції перетворення,

фільтрації і агрегування даних з використанням синтаксису **dplyr**; будувати прогнозні моделі з використанням алгоритмів машинного навчання, реалізованих в бібліотеці MLlib для Spark; працювати з іншими R-пакетами, які використовують Spark для виконання розподілених обчислень.

Завдяки таким пакетам, як **sparklyr**, ми можемо використовувати R в якості клієнта, який відправляє обчислювальні завдання (Push compute) на Spark-кластер, а потім збирає результати обчислень (Collect results) для подальшого аналізу вже в самій системі R. Є два способи формального уявлення подібних обчислювальних задач перед їх відправкою на кластер: або з використанням SQL-команд, або за допомогою функцій з пакета **dplyr**. Хоча SQL (точніше, Spark SQL, який, в свою чергу, заснований на діалекті HiveQL) дозволяє формулювати як завгодно складні операції над даними, в більшості стандартних випадків **dplyr** більш зручний, оскільки він добре знайомий більшості сучасних користувачів R і володіє більш лаконічним синтаксисом.

У наведених нижче прикладах використані дані з пакета nycflights13, який містить кілька таблиць з описом 336776 авіарейсів з аеропортів Нью-Йорка в 2013 р. Запустимо локальний Spark-кластер і завантажимо в нього необхідні таблиці [4]:

```
require(sparklyr)
require(nycflights13)
# підключення до кластеру:
sc <- spark_connect(master = "local", version = "2.3")
# завантаження даних в кластер:
flights_tbl <- copy_to(sc, flights, "flights") # дані по рейсам
airlines_tbl <- copy_to(sc, airlines, "airlines") # дані по авіакомпаніям
```

Припустимо, що перед нами стоїть завдання побудувати модель, яка передбачає ймовірність прибуття затриманого рейсу без запізнення (за умови затримки вильоту на 15-30 хвилин). Будемо розглядати це завдання як випадок бінарної класифікації: цікавий для нас відгук приймає значення 1 якщо затриманий рейс прибув без запізнення, і 0 якщо немає.

Процес побудови прогнозних моделей включає кілька кроків, першим з яких є підготовка та розвідувальний аналіз даних. Сам розвідувальний аналіз також 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 і тощо.

Подивимося, як з цим можуть допомогти команди з пакета **dplyr**.

За допомогою пакета **dplyr** можна виконувати такі стандартні типи обчислень на Spark-кластері:

- вибір, фільтрація і агрегування змінних;
- використання віконних функцій;
- об'єднання декількох таблиць за допомогою join-операторів;
- імпорт результатів обчислень з Spark в середовище R.

До найбільш важливих команд **dplyr** відносяться select(), filter(), mutate(), summarise() і arrange(). Крім того, для виконання групових операцій використовується команда group_by().

Ці та інші команди dplyr можна об'єднувати в "ланцюжки" за допомогою оператора %>% з пакета **magrittr** (підвантажується одночасно з dplyr і тому окремо його викликати не потрібно). Підрахуємо загальну кількість польотів, виконаних кожною авіакомпанією за 2013 р., а потім вибираємо 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
```

Оскільки Spark "розуміє" тільки SQL, то в дійсності усі обчислення, задані в R за допомогою команд **dplyr** перед відправкою на Spark-кластер автоматично переводяться на SQL наступним чином:

select () -> SELECT

filter () -> WHERE

mutate () -> оператори + , - , / , * , log та ін.

summarise () -> функції агрегування SUM , MIN , MAX і ін.

arrange () -> ORDER

group_by () -> GROUP BY

Крім того, dplyr автоматично переводить на SQL такі базові команди R:

Математичні оператори:

+, -, *, /, %%, ^

Математичні функції:

abs, acos, asin, asinh, atan, atan2, ceiling, cos, cosh, exp,

floor, log, log10, round, sign, sin, sinh, sqrt, tan, tanh

Логічні оператори:

<, <=, !=, >=, >, ==, %in%

Булеві оператори:

&, &&, |, ||, !

Функції для роботи с символьними рядками:

paste, tolower, toupper, nchar

Функції для перетворення типу змінної:

as.double, as.integer, as.logical, as.character, as.date

```
# Функції агрегування:
mean, sum, min, max, sd, var, cor, cov, n
```

Функція show_query () з пакету **dplyr** дозволяє переглянути SQL-запит, який формується з відповідного коду R. Для наведеного вище прикладу отримуємо:

```
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
```

Як і у випадку з базами даних, при роботі з Spark **dplyr** дотримується принципу "ледачих обчислень" (Lazy evaluation). Це означає, що всі обчислення на Spark-кластері відкладаються до останнього моменту, поки не знадобиться їх результат. Крім того, підсумковий результат обчислень буде імпортовано в середовище R тільки якщо користувач запросить його в явному вигляді. Наприклад, наведену вище послідовність команд ми могли б прописати в такий спосіб:

```
command_1 <- group_by(flights_tbl, carrier)
command_2 <- summarise(command_1, N = n())
command_3 <- arrange(command_2, desc(N))
command_4 <- head(command_3, n = 5)
```

Жодна з цих послідовних команд не буде виконана до тих пір, поки ми не запитаємо результат обчислень. Як зазначалося раніше , в об'єктах command_1,

command_2 ... command_4 зберігається тільки інформація, необхідна для підключення до Spark-кластеру, а також інструкції для відповідних обчислень.

Під "запитом результату обчислень в явному вигляді" розуміються дві ситуації: або висновок результату на екран, або збереження його в локальний об'єкт R:

```
# вивід на екран:
command_4
## # 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
# імпорт результату та збереження в об'єкт R:
result <- collect(command_4)
result
## # A tibble: 5 x 2
##   carrier     N
##   <chr>   <dbl>
## 1 UA      58665
## 2 B6      54635
## 3 EV      54173
## 4 DL      48110
## 5 AA      32729
```

У другому випадку ми застосували спеціальну команду collect () з пакету dply. У певному сенсі collect () протилежна copy_to (), яка була використана вище для імпорту даних з R в Spark.

У пакеті **dplyr** є кілька функцій для виконання стандартних JOIN -операцій над двома таблицями: inner_join () , left_join () , right_join () , full_join () , semi_join (), nested_join () і anti_join (). У наступному прикладі виконаний LEFT JOIN таблиці flights_tbl з таблицею airlines_tbl за полем carrier:

```
flights_tbl %>%
left_join(airlines_tbl, by = "carrier") %>%
glimpse()
## Observations: ??
## Variables: 20
## Database: spark_connection
## $ year          <int> 2013, 2013, 2013, 2013, 2013, 201...
## $ month         <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ day           <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ dep_time      <int> 517, 533, 542, 544, 554, 554, 555...
## $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600...
## $ dep_delay     <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, ...
## $ arr_time      <int> 830, 850, 923, 1004, 812, 740, 91...
## $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 85...
## $ arr_delay     <dbl> 11, 20, 33, -18, -25, 12, 19, -14...
## $ carrier       <chr> "UA", "UA", "AA", "B6", "DL", "UA...
## $ flight        <int> 1545, 1714, 1141, 725, 461, 1696,...
## $ tailnum       <chr> "N14228", "N24211", "N619AA", "N8...
## $ origin        <chr> "EWR", "LGA", "JFK", "JFK", "LGA"...
## $ dest          <chr> "IAH", "IAH", "MIA", "BQN", "ATL"...
## $ air_time      <dbl> 227, 227, 160, 183, 116, 150, 158...
## $ distance      <dbl> 1400, 1416, 1089, 1576, 762, 719,...
## $ hour          <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, ...
## $ minute        <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0...
## $ time_hour     <dttm> 2013-01-01 10:00:00, 2013-01-01 ...
## $ name          <chr> "United Air Lines Inc.", "United ...
```

Хоча стандартні команди dplyr і деякі базові функції R автоматично перетворюються на SQL, їх буде недостатньо для розробки більш складних обчислень над даними за допомогою Spark.

Spark SQL заснований на мові запитів Hive Query Language (HiveQL) і всі функції цієї мови можна використовувати в поєднанні з командами **dplyr**.

Наприклад, для обчислення медіанного значення змінної dep_delay (затримка рейсу, хв.) з таблиці flights_tbl ми не можемо скористатися базовими функціями R median () або quantile (), це призведе до помилки. Але ми можемо застосувати Hive-функцію percentile ():

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

Коли при перекладі коду R на SQL dplyr зустрічає незнайому функцію, то він просто включає її в SQL-запит "як є":

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

Hive-функція percentile() дозволяє одночасно обчислити кілька процентилей. Для цього на неї потрібно подати 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 <list [3]>
```

Результатом виконання наведеної команди є список з трьома значеннями. Щоб автоматично отримати ці значення зі списку, використовується Hive-функція 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
```

Застосуємо отримані знання в ході розвідувального аналізу даних з таблиці flights_tbl. З'ясуємо, чи є в наших даних пропущені значення. Це можна зробити декількома способами. Нижче підрахунок пропущених значень виконаний 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
## Observations: ??
## Variables: 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> 124
```

Як бачимо, пропущені значення дійсно мають місце в деяких змінних. Максимальна кількість пропущених значень досягає 9430, що становить менше 3% від загального числа спостережень в таблиці (336776). Розмірність таблиці можна з'ясувати за допомогою команди sdf_dim () з пакету **sparklyr**, яка є аналогом базової R-функції dim ().

У пакеті **sparklyr** є ряд інших команд, ім'я яких починається на sdf_ (від "Spark data frame"), наприклад, sdf_nrow () , sdf_ncol () , sdf_bind_rows () , sdf_pivot (). Так як частка пропущених значень невелика, ми можемо видалити відповідні рядки з таблиці без особливого ризику вплинути на якість подальшого аналізу. Для цього скористаємоя базовою функцією na.omit ():

```
flights_full <- flights_tbl %>% na.omit()
## * Dropped 9554 rows with 'na.omit' (336776 => 327222)
flights_full %>% sdf_dim()
## [1] 327222     19
```

Оскільки нас цікавлять рейси, затримка яких склала від 15 до 30 хв. (включно), потрібно відфільтрувати дані відповідним чином.

Паралельно 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))
# розмірність таблиці:
flights %>% sdf_dim()
## [1] 24507    20
```

Ми отримали таблицю з 24507 рядками і 20 стовпцями. Це невелика таблиця і вже зараз її можна було б імпортувати з Spark в R (за допомогою команди collect ()) для подальшого аналізу. Однак з метою демонстрації можливостей роботи Spark з R ми залишимо цю таблицю в пам'яті кластера.

З'ясуємо, які з наявних змінних корелюють з залежною змінною target. Логічно було б очікувати, що ймовірність прибуття затриманого рейсу за розкладом в значній мірі визначається відстанню між аеропортом вильоту і аеропортом прибуття (стовпець distance, що 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      0         820
## 2      1        1089
```

Дійсно, чим більше відстань між аеропортами, тим більше шанс у затриманого рейсу надолужити згаяний час і прибути без запізнення.

Висновок до лекції 17

Apache Spark – це уніфікований механізм аналітики з відкритим кодом для широкомасштабної обробки даних.

Spark забезпечує інтерфейс для програмування цілих кластерів з неявною паралельністю даних та стійкістю до відмов.

Spark підтримує різні мови програмування, такі як Python, Scala, R та Java.

Apache Spark також підтримує структуровані бази даних.

Питання для закріплення

1. У чому полягає проблема обчислювальної функції?
2. Опишіть особливості технології Spark.
3. Яка відмінність між Spark та MapReduce?
4. Які пакети в R використовуються для роботи з Spark?

Список рекомендованої літератури

1. IoT Fundamentals: Big Data & Analytics // Електронний ресурс. Режим доступу: https://www.netacad.com/courses/iot/big-data-analytics

2. Apache Spark // Електронний ресурс. Режим доступу: https://spark.apache.org/

3. Spark і sparklyr для роботи з великими даними в R // Електронний ресурс. Режим доступу: https://r-analytics.blogspot.com/2020/02/spark-intro.html

4. tidyverse/nycflights13 // Електронний ресурс. Режим доступу: https://github.com/tidyverse/nycflights13

5. Локальний Spark-кластер // Електронний ресурс. Режим доступу: https://r-analytics.blogspot.com/2020/02/spark-r-connect.html

6. Аналіз даних в Spark-кластері за допомогою пакета dplyr // Електронний ресурс. Режим доступу: https://r-analytics.blogspot.com/2020/03/spark-dplyr.html

Лекція 18.

Lambda та Kappa архітектури оброблення великих даних

План лекції

18.1. Lambda - архітектура.
18.2. Переваги і недоліки Lambda -архітектури.
18.3. Kappa - архітектура.
18.4. Переваги і недоліки Kappa-архітектури.

18.1. Lambda - архітектура

Прагнучи зблизити аналіз «історичних» даних та даних, отримуваних у режимі реального часу [1], була створена архітектура Lambda (рис. 18.1).

Lambda – це архітектура обробки даних, яка використовує як обробку потоків, так і пакетну обробку для отримання точного перегляду як "живих" даних, так і пакетних даних.