КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теорії та технології програмування

«До захисту допущено»
Завідувач кафедри
Нікітченко М. С.

_____
(підпис)

«___» _____2022 р.

Кваліфікаційна робота

на здобуття ступеня бакалавра
за спеціальністю 122 Комп'ютерні науки
на тему:

РОЗРОБКА КОНЕКТОРІВ ДЛЯ ІНТЕГРАЦІЇ ТЕХНОЛОГІЙ ОБМІНУ ВЕЛИКИМИ ДАНИМИ

Виконав студент 4 курсу
Макарович Адальберт Віталійович

_____
(підпис)

Науковий керівник:
к.фіз.-мат. н., доцент
Панченко Тарас Володимирович

_____
(підпис)

Засвідчую, що в цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри теорії та технологій програмування «01» червня 2022 р., протокол № 10.

Київ – 2022

РЕФЕРАТ

Кваліфікаційна робота на тему: "Розробка конекторів для інтеграції технологій обміну великими даними" *містить:* 53 сторінок, 15 ілюстрацій, 20 джерел посилань.

Ключові слова: база даних, конектор, паралелізм, розподілені системи SingleStore, Spark.

Об'єктом дослідження є процес обміну даними між системами роботи з великими обсягами даних.

Предметом дослідження є конектор, що дозволяє ефективно обмінюватись даними між Spark та SingleStore.

Мета кваліфікаційної роботи полягає у дослідженні проблемних питань використання конекторів та розробці конектора між розподіленими системами обробки великих наборів даних.

Методи дослідження: В процесі дослідження в якості мови програмування для конектора використано Scala. Для тестування конектора використано системи неперервної інтеграції CircleCI та Webapp.io. Для створення демо використано мови програмування Scala, Python, SQL а також технологія Apache Zeppelin. Для компіляції проекту застосовано систему автоматичної збірки sbt. Для комунікації з базою даних використано MariaDB JDBC Driver, а також SingleStore JDBC Driver. Для реалізації пулу з’єднань застосовано бібліотеку DBCP 2. Тестування швидкодії конектора відбувалось за допомогою сервісу AWS EC2.

Результати роботи: розроблений конектор між фреймворком для реалізації розподіленої обробки даних Apache Spark та СКБД SingleStore. Конектор може бути застосований для побудови складних систем для обробки великих даних. Результати досліджень можуть бути застосовані під час створення нових конекторів між Spark та іншими розподіленими реляційними базами даних та використані для вдалого конфігурування конектора при потребі досягнути максимальної швидкодії.

ЗМІСТ

ВСТУП

Оцінка сучасного стану об'єкта розробки. На сьогоднішній день існує велика кількість систем для обробки великих наборів даних. Різні системи орієнтуються на виконання різних типів завдань і через це мають великі відмінності в архітектурі.

Конфігурування однієї розподіленої системи вже є складним завданням, для якого компанії використовують послуги DevOps інженерів. Створення конектора, який би дозволяв комунікацію двох систем і оптимально використовував особливості обох є надзвичайно складним завданням.

На даний момент, більшість систем розробляють свої власні конектори до найпопулярніших продуктів таких як Spark, Kafka, BigQuery, а також конектори, що базуються на драйверах, які здатні підтримувати великий стек технологій (JDBC, ODBC). Існують навіть компанії, що надають послуги з розробки конекторів [21].

Компанія Spark розробила власний JDBC конвектор, який можна застосовувати для з'єднання із будь-якою базою даних, що має JDBC драйвер. Хоч даний конектор і надає можливість взаємодії двох систем, але в багатьох моментах його поведінка є неоптимальною та не дозволяє використовувати розподілені властивості цих систем.

Актуальність роботи та підстави для її виконання. Кількість даних в світі постійно зростає, а їхній аналіз набуває все більшої необхідності. Через пристосованість різних систем під виконання специфічних функцій, виникає потреба у використанні багатьох систем під час роботи над одним проектом.

Відповідно, постає потреба у створенні конекторів між ними, які могли б оптимально використовувати властивості різних систем. З іншого боку, пристосованість під виконання специфічних функцій створює різниці в архітектурі, які ускладнюють процес розробки подібних конекторів.

Актуальним є розгляд розробки конектора між фреймворком для реалізації розподіленої обробки даних Apache Spark та реляційну розподілену СКБД SingleStore (більш відому як MemSQL).

Мета й завдання роботи. Мета кваліфікаційної роботи полягає у дослідженні проблемних питань використання конекторів та розробці конектора між розподіленими системами обробки великих наборів даних.

Для досягнення поставленої мети виокремленні наступні **завдання**:

- ✓ провести аналіз особливостей SingleStore та Spark і ситуацій, в яких вони можуть використовуватись разом;
- ✓ на основі проведеного аналізу сформулювати вимоги, необхідні для задоволення потреб користувачів;
- ✓ дослідити конвертацію операцій Spark в SQL запити та розширити кількість підтримуваних перетворень;
- ✓ провести аналіз та реалізацію різних способів паралельного читання та запису даних із SingleStore в Spark;
- ✓ визначити параметри конектора та навести поради щодо його конфігурації;
- ✓ здійснити тестування конектора та налаштувати систему неперервної інтеграції;
- ✓ розробити демо-проєкти, за допомогою яких можна продемонструвати роботу конектора потенційним клієнтам;

Об’єкт, методи й засоби розроблення. Об’єктом дослідження є процес обміну даними між системами роботи з великими обсягами даних.

Предметом дослідження є конектор, що дозволяє ефективно обмінюватись даними між Spark та SingleStore.

В якості мови програмування для конектора використано Scala. Для тестування конектора використано системи неперервної інтеграції CircleCI та Webapp.io.

Для створення демо використано мови програмування Scala, Python, SQL, а також технологія Apache Zeppelin.

Для компіляції проекту застосовано систему автоматичної збірки sbt. Для комунікації з базою даних використано MariaDB JDBC Driver, а також SingleStore JDBC Driver.

Для реалізації пулу з’єднань застосовано бібліотеку DBCP 2. Тестування швидкодії конектора відбувалось за допомогою сервісу AWS EC2.

Для мануального тестування складних сценаріїв, які не можуть бути протестовані в локальному середовищі, застосовувалась платформа Databricks.

Можливі сфери застосування. Конектор може бути застосований для побудови складних систем для обробки великих даних.

Результати досліджень можуть бути застосовані під час створення нових конекторів між Spark та іншими розподіленими реляційними базами даних та використані для вдалого конфігурування конектора при потребі досягнути максимальної швидкодії.

РОЗДІЛ 1.
АНАЛІЗ ЗАДАЧІ ТА ФОРМУВАННЯ ВИМОГ ДО КОНЕКТОРА

Проектування конектора потребує детального огляду систем, взаємодію між якими він буде надавати. В цьому розділі наведена інформацію про SingleStore та Spark, їхні принципи роботи та сильні сторони, сформульовано необхідний для реалізації функціонал та обґрунтовано його важливість.

1.1 Огляд бази даних SingleStore

SingleStore є розподіленою реляційною системою управління базами даних [10]. Її можна розгорнути в двох варіантах:

- Cloud
- Self-managed

При використанні Cloud – користувач вибирає конфігурацію бази даних (необхідну кількість оперативної пам’яті, пам’яті на диску, ядер, вершин) із запропонованих сервісом варіантів і сервіс створює базу даних на серверах AWS, Google cloud або Azure.

При використанні Self-Managed – користувач повинен сам встановити програму на своїх серверах, сконфігурувати її та запустити за допомогою спеціалізованих утиліти SingleStore Toolbox.

Перший варіант швидший і простіший. При ньому відповідальність за стан бази даних та її підтримку лягає на розробників SingleStore. При другому варіанті - користувач має набагато більше можливостей.

Наприклад, його вибір конфігурацій не обмежується тими варіантами, що пропонує сервіс. Також, якщо він встановлює базу даних на своїх серверах – він може використовувати спеціалізовані машини, які найкраще підходять для його потреб. Недоліком цього є необхідність самому конфігурувати базу даних, що потребує високо рівня розуміння принципів роботи бази даних.

SingleStore є розподіленою СКБД. Один SingleStore кластер складається із наступних вершин:

- одного головного агрегатора;
- жодного або декількох допоміжних агрегаторів;
- як мінімум одного листка.

Кожна вершина є незалежним процесом. В листках зберігаються сегменти баз даних із власне самими даними. До версії 7.3 всі DDL запити мали виконуватись за допомогою головного агрегатора. Допоміжні агрегатори існували для того, щоб розпаралелити DML запити. Після отримання запиту агрегатор створює план його виконання, надсилає підзапити до листків і потім обробляє їх результати і повертає остаточний результат. Починаючи з версії 7.3 виконання деяких DDL запитів стало можливим і на допоміжних агрегаторах. В останній версії майже всі запити можуть бути виконані на допоміжних агрегаторах. Основними і найбільш ресурсоємними операціями Spark коннектора буде запис та читання великих таблиць. Найголовнішим для оптимального виконання таких запитів є спроба розпаралелити їх між агрегаторами.

Ще однією особливістю SingleStore є підтримання двох типів таблиць: columnstore та rowstore.

Rowstore зберігається в оперативній пам’яті. Він призначений для швидкого виконання невеликих запитів зміни рядків в реальному часі, що чудово підходить для потреб OLTP. Користувач може використовувати таку таблицю для збереження даних, що часто оновлюються. Наприклад, повідомлення та пости в соціальних мережах, статус користувачів (онлайн/офлайн), банківські операції. Далі, щоб аналізувати збережені дані, користувач може скористатись процесом ETL - завантажити їх (повністю або частково) в OLAP сховище даних та провести аналіз.

В документації SingleStore наявна публікація, що детально описує подібний процес [8]. В цій публікації, здійснюється опис того, як дані із TPCH датасету завантажуються в сервіс AWS Glue за допомогою SingleStore AWS Glue

конектора, що базується на SingleStore Spark конекторі, розробка якого розглядається. Далі, над даними відбуваються обрахунки і результат записується в базу даних як нова таблиця.

Columnstore таблиці зберігаються на диску. Таблиці цього типу розміщують дані не рядок за рядком, а окремо кожну колонку. Це дозволяє пришвидшити операції агрегацій над таблицею, що суттєво впливає на час виконання OLAP процесів. Хоча запити вставки та зміни рядків в таких таблицях і повільніші, але різниця в швидкодії не є критичною, особливо для випадків, коли дані тільки додаються до таблиці і відсутні запитів зміни рядків. При збільшенні кількості інформації в базі даних час на виконання ETL процесу відповідно збільшується і може досягати цілих днів. Через це, є сенс розглянути можливість виконання обрахунків на стороні бази даних. Columnstore таблиці чудово для цього підходять, адже:

- ✓ columnstore оптимізований для виконання запитів, що зустрічаються в OLAP системах;
- ✓ columnstore зберігаються на диску і чудово стискає дані, що дозволяє зберігати велику кількість інформації;
- ✓ при виконанні обчислень на стороні бази даних зникає необхідність в пересиланні великої кількості інформації через мережу;
- ✓ зміна швидкодії OLTP запитів не є критичною;
- ✓ за рахунок кешування та якісно реалізованого паралелізму виконання певних запитів на стороні бази даних, може бути здійснене швидше ніж у випадку завантаження всіх даних в спеціалізовану OLAP систему;
- ✓ зникає необхідність підтримувати великі сервера для аналітичної частини процесу (всі обчислення й так відбуваються на стороні SingleStore).

Більш детально ці переваги описані в публікації одного з користувачів SingleStore [20].

Важливо відмітити, що користувачі відповідальні за те, щоб зберігати дані консистентними і відслідковувати, щоб зміни в таблиці не впливали на роботу аналітичних запитів. Для максимально вдалого застосування OLAP властивостей бази даних, конектор повинен вміти конвертувати запити Spark в SQL, який міг бути виконаний на стороні SingleStore.

Columnsotre таблиці часто можуть використовуватись разом із rowstore в одній системі. В columnstore таблиці може зберігатись велика кількість інформації потрібної для швидкої аналітики. Рядки, при цьому, мають не часто змінюються (наприклад операції купівлі в онлайн магазинах, адже перед збереженням такої операції і так є затримка в декілька секунд на її опрацювання банком). В rowstore таблицях може міститись інформація, що часто змінюється і швидкодія зміни якої, повинна бути достатньо високою (наприклад статус користувача - онлайн чи офлайн).

Ще однією особливістю SingleStore є наявність типу JSON, та доволі великої кількості спеціалізованих функцій для роботи з ним. Це дає можливості зберігати неструктуровані дані. Здається, що реляційні й документальні бази даних з часом стають все більш і більш подібними і це добра річ: різні моделі даних доповнюють одна одну. Якщо база даних буде спроможна опрацьовувати документоподібні дані, а також робити реляційні запити над ними, застосунки зможуть використовувати комбінації особливостей, що найкраще підходять для їхніх потреб [5].

Серед недоліків SingleStore можна відмітити відсутність зовнішніх ключів. Хоч, об’єднання таблиць по ключу досі залишається доволі швидкою операцією, але користувач повинен слідкувати, щоб його дані залишались консистентними.

Таким чином, для ETL процесів найважливіше, щоб конектор міг розпаралелити читання даних між вершинами кластера. Тоді масштабуючи SingleStore кластер можна досягнути необхідної швидкодії та відповідно масштабувати й всю систему.

Для процесів, в яких запити повинні виконуватись на стороні бази даних, використовуючи OLAP властивості SingleStore – найважливішим є вміння конектора генерувати SQL запити для необхідних обчислень.

Ще однією важливою властивістю є вміння швидко записувати інформацію в базу даних, так само розпаралелюючи запити між вершинами кластера. Вона може використовуватись для збереження результату або ж для додавання в базу даних таблиць із інших джерел для майбутніх обрахунків (наприклад можна завантажити excel файл в Spark, далі зберегти його в SingleStore і потім провести необхідні обчислення разом з іншими таблицями, що вже містяться в базі даних).

1.2 Огляд механізму Spark

Apache Spark – це уніфікований аналітичний механізм для великомасштабної обробки даних. Він надає API високого рівня на мовах Java, Scala, Python, R. Також, він підтримує велику множину високорівневих застосунків, таких як: Spark SQL для SQL і обробки структурованих запитів, MLlib для завдань машинного навчання, GraphX для обробки графів та Structured Streaming для інкрементального виконання потокових процесів [16].

Spark має відкритий код і є безкоштовним для використання. Ядро цієї технології реалізоване на функціональний мові програмування Scala.

Зазвичай, Spark кластер складається із головної вершини та вершин робітників. Існує ще спосіб розгортання кластера, при якому існує тільки одна вершина, що виконує обидві ролі. Такий спосіб використовується в основному для локального тестування.

На високому рівні кожна програма Spark складається з програми-драйвера, яка запускає головну функцію користувача та виконує різні паралельні операції в кластері. Програма драйвер створює підзавдання, які виконуються на вершинах робітниках.

Основною абстракцією, яку надає Spark, є стійкий розподілений набір даних (resilient distributed dataset - RDD). RDD є відмовостійкою колекцією елементів, з якими можна працювати паралельно.

Існує два способи створення RDD: розпаралелювання наявної колекції у програмі драйвера або посилання на набір даних у зовнішній системі зберігання, наприклад, спільна файлова система, HDFS, HBase або будь-яке джерело даних.

RDD підтримують два типи операцій: перетворення, які створюють новий набір даних із існуючого, і дії, які повертають значення програмі-драйверу після виконання обчислень над набором даних.

Наприклад, map — це перетворення, яке виконує функцію над кожним елементом набору і повертає новий RDD, що містить результати виконання. З іншого боку, reduce – це дія, яка об’єднує всі елементи RDD за допомогою деякої функції та повертає кінцевий результат програмі-драйверу (хоча існує також паралельна версія цієї операції reduceByKey, яка повертає RDD).

Усі перетворення в Spark ліниві, оскільки вони не обчислюють свої результати відразу. Замість цього, вони просто запам’ятовують перетворення, застосовані до деякого базового набору даних (наприклад, файлу). Перетворення обчислюються лише тоді, коли дія вимагає повернення результату програмі драйвера. Така конструкція дозволяє Spark працювати більш ефективно. Наприклад, програма може помітити, що набір даних, створений за допомогою map, буде використаний в reduce, і повернути драйверу лише результат reduce, а не весь набір даних.

За замовчуванням кожен перетворений RDD може перераховуватися щоразу, коли ви виконуєте з ним дію. Однак ви також можете зберігати RDD в пам’яті за допомогою методу persist (або кешу), у цьому випадку Spark збереже елементи в кластері для набагато швидшого доступу під час наступного запиту до них. Існує також підтримка постійних RDD на диску або реплікованих на кількох вершинах [18].

Spark конектор повинен реалізовувати інтерфейс, що дозволяє системі створювати об’єкти, з якими можна працювати так само як із звичайними RDD. При цьому, він повинен намагатись розбити всі складні операції на незалежні підзавдання, щоб вони могли бути розпаралелені та виконані на вершинах-робітниках. Оскільки, Spark запам’ятовує всі перетворення і виконує їх ліниво, з’являється можливість конвертувати всі запам’ятовані перетворення в SQL запити і виконати їх на стороні бази даних. Детальніше цей процес буде описано в наступних розділах.

Spark здатен вдало розпаралелювати операції над RDD між вершинами-робітниками, але в більшості випадків виконання аналогічних обчислень на стороні SingleStore відбувається швидше (за рахунок кешування та індексів). Також, при виконанні обчислень на стороні бази даних зменшується необхідність обміну великої кількості даних між Spark кластером та SingleStore кластером.

За рахунок відкритого коду та вдало підібраних абстракцій, під які легко підлаштувати будь-яку систему для Spark створено безліч конекторів. Це робить Spark чудовою основою для побудови сховищ даних. Дані можуть завантажуватись з різних джерел а потім паралельно оброблятись на кластері.

Наприклад, користувач може створити RDD, що містить інформацію з Excel таблиці, RDD що містить інформацію з Amazon S3 і RDD, що містить інформацію із бази даних SingleStore. Далі відфільтрувати ці дані, привести їх в однаковий формат та виконати обчислення, що використовують всі три RDD. При цьому, частина операцій може бути конвертована в SQL запит і виконана в SingleStore. Також, користувач може записати перші два RDD в SingleStore та спробувати виконати всі обчислення на стороні бази даних.

Ще однією перевагою використання Spark є те, що він надає високорівневий API на багатьох популярних мовах. Писати код на Java або Python набагато зручніше ніж формулювати ту саму логіку за допомогою складних SQL запитів.

Висновки до розділу 1

За результатами дослідження окреслено особливості SingleStore та Spark і ситуації, в яких користувачу доведеться використовувати їх одночасно та сформулювано вимоги до конектора, необхідні для задоволення потреб користувача.

Отже, SingleStore є розподіленою реляційною системою управління базами даних. Її можна розгорнути в двох варіантах: cloud та self-managed.

Один SingleStore кластер складається із наступних вершин: одного головного агрегатора; жодного або декількох допоміжних агрегаторів; як мінімум одного листка.

Ще однією особливістю SingleStore є підтримання двох типів таблиць: columnstore та rowstore. Rowstore призначений для OLTP процесів, коли columnstore краще підходить для OLAP, хоча незважаючи на це має досить високу швидкодію вставки нових рядків в таблицю.

Spark кластер складається із: однієї головної вершини та декількох вершин-робітників.

За рахунок відкритого коду та вдало підібраних абстракцій, під які легко підлаштувати будь-яку систему для Spark створено безліч конекторів. Це робить Spark чудовою основою для побудови сховищ даних. Дані можуть завантажуватись з різних джерел а потім паралельно оброблятись на кластері.

Проведено аналіз ситуацій, в яких в користувача може виникнути потреба для одночасного використання обох технологій.

Основними вимогами до конектора, необхідні для задоволення потреб користувача є:

- ✓ паралельне читання даних із SingleStore (особливо важливо для ETL процесів);
- ✓ конвертація перетворень здійснених над RDD в SQL запит і виконання їх на стороні бази даних (особливо важливо для випадків, коли SingleStore використовується як OLAP база даних);
- ✓ паралельний запис RDD в SingleStore.

РОЗДІЛ 2.
ЗАГАЛЬНА АРХІТЕКТУРА КОНЕКТОРА

Для створення конектора важливо розуміти яким саме чином Spark виконує операції над RDD. Цей розділ містить загальний опис механізму, що використовується в Spark для того, щоб паралельно обчислити результат певного запиту, а також опис основних класів конектора.

2.1. Процес обробки запитів в Spark

Процес обробки запитів в Spark може бути умовно поділений на чотири частини:

- створення логічного плану;
- оптимізація логічного плану;
- перетворення логічного плану в фізичний план;
- виконання завдань фізичного плану на кластері.

Більш детальний опис процесу обробки запитів, зображений на рис.2.1.

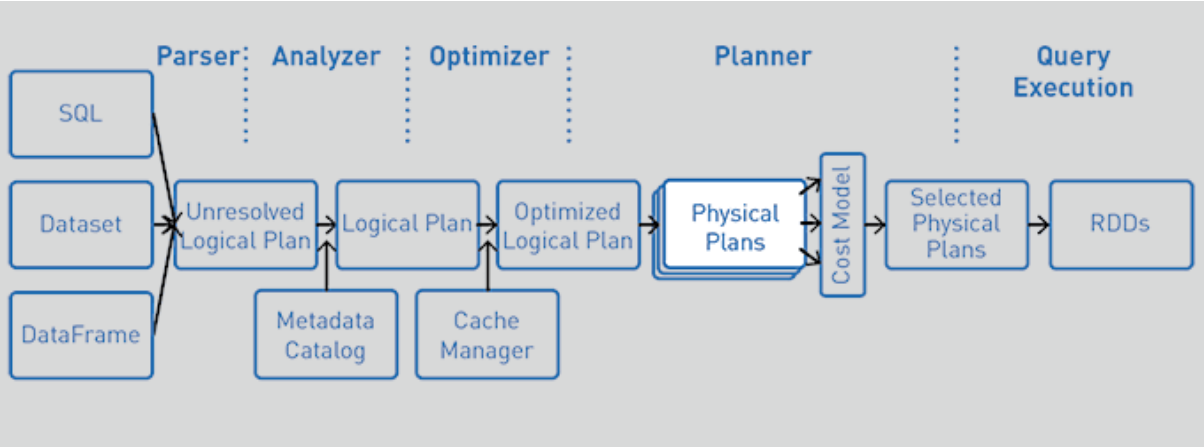

Рис 2.1. Процес обробки запитів в Spark [3]

На першому етапі створюється логічний план. Це план, який показує, які кроки будуть виконані, коли буде застосовано дію. При цьому варто враховувати, що коли застосовується перетворення до набору даних,

створюється новий набір даних. Коли це станеться, новий набір даних вказує назад на батьківський. Таку структуру можна зобразити у вигляді орієнтованого ациклічного графа, що містить інформацію про всі перетворення.

Spark містить певну множину оптимізаторів. Кожен з них приймає логічний план і перебудовує його, повертаючи новий план. Всі оптимізатори почергово застосовуються до плану. Користувач має можливість додати свої власні оптимізатори. Саме ця властивість може бути використана для перетворення Spark операцій в SQL запит.

На наступному етапі відбувається перетворення оптимізованого логічного плану в фізичний план. Всі трансформації (ребра) логічного плану можуть бути поділені на два типи: вузькі та широкі трансформації.

Вузькі трансформації не переміщують дані між сегментами множини даних. Прикладами таких трансформацій є filter або select. Декілька вузьких трансформацій можуть бути виконані над частиною даних, що знаходяться на одній із вершин-робітників послідовно і без комунікації з іншими вершинами.

Широкі трансформації спричиняють обмін інформації між сегментами. Прикладами таких трансформацій є groupBy, agg, sortBy. На рис 2.2 зображено як дані переміщуються між сегментами під час обробки вузьких та широких трансформацій в Spark.

Оптимізований логічний план розбивається на етапи таким чином, щоб всередині одного етапу були тільки вузькі трансформації. Далі кожен етап розбивається на окремі завдання для сегментів. Такі завдання надсилаються із програми-драйвера на вершини-робітники і виконуються там [3].

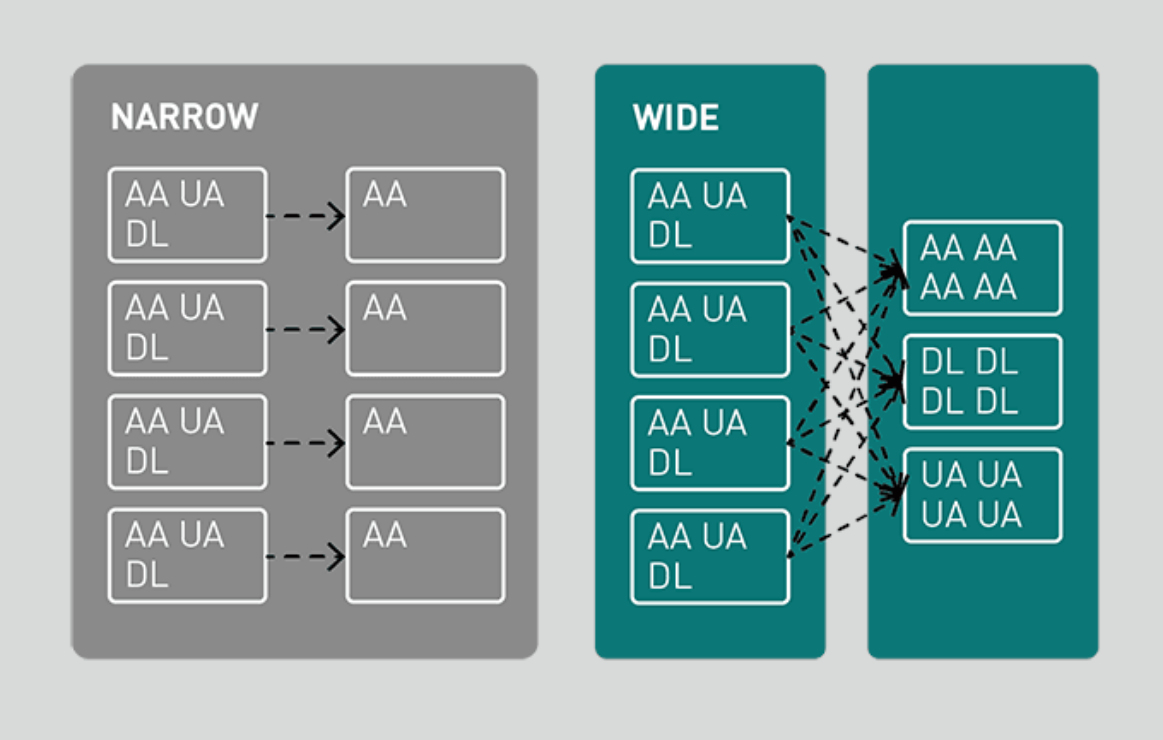

Рис 2.2. Вузькі та широкі трансформації в Spark [3]

Отже, операції читання даних із сховища знаходяться в листках логічного плану. При паралельному читанні Spark надсилає завдання, що читають дані із сховища та виконують всі вузькі трансформації до вершин-робітників. Кількість таких завдань рівна кількості сегментів у базі даних.

2.2 Основні класи конектора

Головним класом конектора є клас DefaultSource. Він реалізує три інтерфейси:

1. RelationProvider;
2. CreateRelationProvider;
3. DataSourceRegister.

Інтерфейс RelationProvider реалізується об’єктами, що створюють реляції для певного сховища даних. Він містить єдиний метод - createRelation, що приймає глобальну інформацію про сесію Spark програми у вигляді об’єкта SQLContext та словник параметрів. Метод createRelation повертає об’єкт, що

реалізує інтерфейс BaseRelation (колекцію кортежів із відомою схемою). Цей метод викликається при читанні із сховища даних. Об’єкт, який він повертає зберігається в листках логічного плану. В нашому випадку, це буде об’єкт класу SinglestoreReader. Цей клас містить метод buildScan, що повертає RDD для читання даних із сховища.

Розглянемо приклад коду для читання даних.

```
df:DataFrame = spark
  .read
  .format(“com.singlestore.spark.DefaultSource”)
  .option(“optionName1”, “optionValue1”)
  …
  .option(“optionNameN”, “optionValueN”)
  .load(“databaseName.tableName”)
```

Рис. 2.3. Приклад коду для читання даних.

В даній ситуації spark – Spark сесія застосунку. Наступним кроком є застосування операції, що має бути виконана. У нашому випадку, це читання. Після цього, вказується клас, що повинен бути використаний для читання із сховища даних. Далі йде множина опцій, для конфігурації операції читання. Вкінці вказується база даних та таблиця, з яких власне і мають бути завантажені дані.

Spark, виконуючи даний запит, викличе функцію createRelation класу DefaultSource передавши їй всі вказані параметри. Результатом виконання цього коду буде об’єкт типу DataFrame – колекція кортежів, які можуть бути перетворені паралельно використовуючи функції та реляційні операції.

Spark не завантажить всі дані відразу. Замість цього, даний об’єкт буде містити тільки інформацією про те, як їх дістати. Створений об’єкт просто читає дані із сховища, але якщо застосувати до нього певні операції реляційної алгебри, можна отримати об’єкти типу DataFrame із складнішим планом виконання.

Інтерфейс CreateRelationProvider реалізується об’єктами, що зберігають DataFrame в сховище даних. Цей інтерфейс також складається з єдиного методу

createRelation, але він приймає ще і DataFrame, який має бути збережений. Клас, що реалізує інтерфейс відповідає за оптимальний запис даних. Для цього можна використати метод класу DataFrame foreachPartition. Цей метод приймає функцію і запускає її на вершинах-робітниках. Кожен виклик функції приймає ітератор на рядки одного сегмента об'єкту DataFrame.

Розглянемо приклад коду для збереження даних.

```
df.write
  .format("com.singlestore.spark.DefaultSource")
  .option("optionName1", "optionValue1")
  …
  .option("optionNameN", "optionValueN")
  .save("databaseName.tableName")
```

Рис. 2.4. Приклад коду для збереження даних.

В даному прикладі df – DataFrame, що має бути збережений. На наступному етапі використовується функція для збереження write. Потім, вказується клас конектора, що має бути використаний для збереження, конфігурації, та місце, куди DataFrame має бути збережений.

Останнім інтейфейсом є DataSourceRegister. Він потрібний, щоб надати можливість звертатись до конектора за допомогою короткого імені і за рахунок цього, зробити його використання простішим. Для цього, потрібно реалізувати фунцію shortName. Наприклад, після реалізації цього інтерфейсу читання може бути виконане за допомогою наступного коду:

```
df:DataFrame = spark
  .read
  .format("singlestore")
  .option("optionName1", "optionValue1")
  …
  .option("optionNameN", "optionValueN")
  .load("databaseName.tableName")
```

Рис. 2.5. Приклад коду із використанням короткого імені конектора.

Отже, для створення робочого конектора достатньо реалізувати декілька інтерфейсів.

2.3 Спосіб комунікації конектора із базою даних

Швидкий та якісно реалізований спосіб передачі даних є одним з найголовніших аспектів конектора. SingleStore підтримує протокол MySQL і через це з'єднання з ним можна встановити за допомогою багатьох вже існуючих драйверів. Основними особливостями цього протоколу є:

- ✓ прозоре шифрування за допомогою SSL;
- ✓ стиснення даних;
- ✓ фаза з'єднання під час якої відбувається обмін параметрами та даними аунтентифікації;
- ✓ фаза команд, яка приймає команди від клієнта та виконує їх [7].

Для версій конектора 2.x та 3.x використано MariaDB Connector/J – JDBC драйвер для MariaDB, що працює на основі згаданого вище протоколу.

Використання запозиченого драйвера значно полегшило розробку на початкових етапах, але спричинило певні проблеми в кінці.

Через це, в ході здійснення нашого дослідження драйвер було змінено на SingleStore JDBC Driver аналогічний драйвер, але розроблений саме для SingleStore. Оскільки підтримкою коду цього драйвера займається команда SingleStore - компанія має можливість виправляти помилки в ньому та додавати нові особливості не залежачи при цьому від MariaDB. Починаючи з версії 4.x – конектор використовує новий драйвер.

Для встановлення з'єднання конектор надає наступні параметри.

- ✓ ddlEndpoint - ім'я хоста або IP адреса головного агрегатора бази даних в форматі host[:port]. Даний параметр є обов'язковим. Приклад: master-agg.foo.internal:3308 або master-agg.foo.internal.

- ✓ dmlEndpoints - ім’я хостів або IP адрес вершин для надсилання запитів у форматі host[:port],host[:port],... (порт не є обов’язковим, декілька адрес розділені комою). Якщо цей параметр не вказаний, то замість нього використовується ddlEndpoint. Приклад: child-agg:3308,child-agg2.
- ✓ user - ім’я користувача SingleStore, що використовується для встановлення з’єднань. За замовчуванням – root.
- ✓ password - пароль до бази даних. За замовчуванням порожній рядок
- ✓ query - запит, що надсилається до бази даних. Не може використовуватись разом з параметром dbtable.
- ✓ dbtable – таблиця, над якою виконуються запити. Не може використовуватись разом з параметром query.
- ✓ database - ім’я бази даних, до якої встановлюється з’єднання.

Якщо користувач вказує параметр, що не є серед перелічених – він буде напряму переданий до драйвера під час створення з’єднання.

Є два способи передати параметри конектору. Перший з них був продемонстрований раніше (рис. 2.3, 2.4, 2.5). Крім нього, параметри можуть бути встановлені глобально в об’єкті Spark сесії. В такому випадку, перед іменем параметра має йти префікс spark.datasource.singlestore.

Приклад:

```
spark.conf.set("spark.datasource.singlestore.ddlEndpoint", "singlestore-master.cluster.internal")
spark.conf.set("spark.datasource.singlestore.dmlEndpoints", "singlestore-master.cluster.internal,singlestore-child-1.cluster.internal:3307")
spark.conf.set("spark.datasource.singlestore.user", "admin")
spark.conf.set("spark.datasource.singlestore.password", "s3cur3-pa$$word")
```

Рис. 2.6. Приклад встановлення глобальних параметрів.

Параметри з’єднання найчастіше встановлюються саме таким чином, адже вони доволі громіздкі і рідко змінюються впродовж виконання застосунку. Якщо

ж користувач хоче одночасно звертатись до декількох баз даних в межах одного застосунку він може використати попередній спосіб.

На перший погляд, може здатись дивним наявність двох IP адрес для з'єднання. Два параметри були запропоновані через те, що SingleStore до версії 7.3 міг виконувати DDL запити тільки за допомогою головного агрегатора. При цьому інші агрегатори необхідні для вдалого розпаралелення всіх інших запитів. При створенні з'єднання до бази даних для виконання складних операцій використовується випадковий адрес із перелічених в параметрі dmlEndpoint.

Ще одним важливим аспектом комунікації з базою даних є наявність пулу з'єднань. Створення нового з'єднання є дорогою операцією, адже вона блокує будь-які інші операції на кластері. Особливо це стає помітно, якщо користувач намагається виконувати багато швидких операцій (наприклад додавання невеликої кількості рядків до бази даних в режимі реального часу). Якщо для кожної такої операції буде відбуватись створення нового з'єднання, то це в рази сповільнює швидкодію програми. Щоб уникнути таких випадків, у ході виконання нашого дослідження до конектора було додано пул з'єднань. Його реалізовано за допомогою бібліотеки DBCP 2 [2].

В кожному процесі конектор зберігає об'єкт BasicDataSource бібліотеки DBCP 2 для кожної множини параметрів з'єднань.

Цей об'єкт відповідає за створення з'єднань і їх зберігання для повторного використання. Кожного разу, коли створюється нове з'єднання, конектор перевіряє чи для параметрів з'єднання вже є відповідний об'єкт BasicDataSource . Якщо його немає – створюється новий.

На наступному етапі використовується його метод getConnection для того, щоб отримати з'єднання. Бібліотека DBCP 2 кешує створені з'єднання і якщо якісь з них вільні, то вони використовуються повторно. Після того, як з'єднання не використовувалось певний проміжок часу воно автоматично видаляється з кешу. Під час створення нового з'єднання відбувається перевірка всіх створених об'єктів BasicDataSource. Якщо якийсь з них не містить жодного активного з'єднання - він видаляється.

Конектор надає велику кількість параметрів для конфігурації пула з’єднань. Оскільки запити, що надсилаються з головного процесу Spark кластера та запити, що надсилаються із процесів-робітників мають різну структуру, то користувачу надається можливість конфігурувати їх незалежно одне від одного. Користувач навіть має можливість виключити використання пулу з’єднань. Вони перераховані нижче.

- driverConnectionPool.Enabled – включає використання пулу з’єднань для запитів в головному процесі Spark кластера. Значення за замовченням – true.
- driverConnectionPool.MaxOpenConns – максимальна кількість активних з’єднань з однаковими параметрами, що можуть бути створені в головному процесі. Від’ємне число у випадку, коли обмежень немає. Значення за замовченням (-1).
- driverConnectionPool.MaxIdleConns – максимальна кількість активних з’єднань з однаковими опціями, що можуть залишатись незадіяними в пулі з’єднань головного процесу. Від’ємне число у випадку, коли обмежень немає. Значення за замовчуванням – 8.
- driverConnectionPool.MinEvictableIdleTimeMs – мінімальна кількість часу, впродовж якого, з’єднання з пулу головного процесу може залишатись незадіяним перед тим, як воно буде видалене. Значення за замовчуванням - 30000 (30 секунд).
- driverConnectionPool.TimeBetweenEvictionRunsMS – час між запусками перевірок з’єднань на необхідність видалення з пулу головного процесу. Якщо цей параметр не додатній – з’єднання не будуть видалятись. Значення за замовчуванням – 1000 (1 секунда).
- driverConnectionPool.MaxWaitMS – максимальна кількість часу, яку пул головного процесу буде чекати для створення з’єднання перед поверненням помилки. Значення за замовченням – (-1).
- driverConnectionPool.MaxConnLifetimeMS – час життя з’єднання в пулі головного процесу. Після проходження цього часу, перевірка на

необхідність видалення буде спрацьовувати для цього з’єднання і воно більше не буде використане.

- executorConnectionPool.Enabled – включає використання пулу з’єднань для запитів в процесах-робітниках Spark кластера. Значення за замовченням – true.
- executorConnectionPool.MaxOpenConns - максимальна кількість активних з’єднань з однаковими опціями, що можуть бути створені в процесах-робітниках. Від’ємне число у випадку, коли обмежень немає. Значення за замовченням (-1).
- executorConnectionPool.MaxIdleConns - максимальна кількість активних з’єднань з однаковими опціями, що можуть залишатись незадіяними в пулі з’єднань процесів-робітника. Від’ємне число у випадку, коли обмежень немає. Значення за замовчуванням – 8.
- executorConnectionPool.MinEvictableIdleTimeMs - мінімальна кількість часу, яку з’єднання з пулу процеса-робітника може залишатись незадіяними перед тим, як воно буде видалене. Значення за замовчуванням - 2000 (2 секунд).
- executorConnectionPool.TimeBetweenEvictionRunsMS – час між запусками перевірок з’єднань на необхідність видалення з пулу процеса-робітника. Якщо цей параметр не додатній – з’єднання не будуть видалятись. Значення за замовчуванням - 1000 (1 секунда).
- executorConnectionPool.MaxWaitMS - максимальна кількість часу, яку пул процеса-робітника буде чекати для створення з’єднання перед поверненням помилки. Значення (-1) означає чекати поки з’єднання не буде створене. Значення за замовчуванням (-1).
- executorConnectionPool.MaxConnLifetimeMS – час життя з’єднання в пулі процеса-робітника. Після проходження цього часу, перевірка на необхідність видалення буде спрацьовувати для цього з’єднання і воно більше не буде використане.

Як бачимо, для процесів-робітників за замовчуванням кількість часу, впродовж якого незадіяне з'єднання залишається в пулі набагато менша ніж для головного процесу.

Це пов'язане з тим, що процеси-робітники зазвичай створюють багато з'єднань для паралельного надсилання запитів читання та запису даних. З іншого боку, головний процес надсилає в основному, послідовні запити для створення таблиць та запити для визначення оптимальної поведінки конектора (наприклад визначення версії бази даних, схеми таблиці, плану виконання складніших запитів і т. д.).

Відповідно, головний процес створює набагато менше нових з'єднань і може дозволити зберігати їх довший час.

Потрібно бути надзвичайно обережним при використанні параметрів driverConnectionPool.MaxOpenConns та executorConnectionPool.MaxOpenConns. За замовчуванням вони не встановлюють ніяких обмежень.

Якщо встановити занадто малі обмеження – є ризик, що під час паралельного читання або запису даних конектор не зможе створити необхідну кількість з'єднань. В кращому випадку це спричинить зменшення швидкодії, а в гіршому може навіть спричинити помилку читання або запису.

Висновки до розділу 2.

Цей розділ містить загальний опис механізму, що використовується в Spark для того, щоб паралельно обчислити результат певного запиту. Механізм обробки запитів в Spark може бути умовно поділений на чотири частини:

- ✓ створення логічного плану;
- ✓ оптимізація логічного плану;
- ✓ перетворення логічного плану в фізичний план;
- ✓ виконання завдань фізичного плану на кластері.

Також, розділ містить опис головного класу конектора – DefaultSource. Цей клас є вхідною точкою роботи конектора. Він реалізує три інтерфейси:

- ✓ RelationProvider;

- ✓ CreateRelationProvider;
- ✓ DataSourceRegister;

Інтерфейс RelationProvider реалізується об’єктами, що створюють реляції для певного сховища даних. Він дозволяє читати дані із SingleStore в Spark.

Інтерфейс CreateRelationProvider реалізується об’єктами, що зберігають DataFrame в сховище даних. Він дозволяє записати дані із Spark в SingleStore.

Інтейфейс DataSourceRegister потрібний, щоб надати можливість звертатись до конектора за допомогою короткого.

Швидкий та якісно реалізований спосіб передачі даних є одним з найголовніших аспектів конектора. SingleStore підтримує протокол MySQL і через це з’єднання з ним можна встановити за допомогою багатьох вже існуючих драйверів.

Для версій конектора 2.x та 3.x використано MariaDB Connector/J - JDBC драйвер для MariaDB, що працює на основі згаданого вище протоколу. В ході виконання цієї роботи конектор було змінено і пристосовано для використання SingleStore JDBC Driver. Ця зміна дає розробникам компанії SingleStore більше контролю над продуктом.

Ще одним важливим аспектом комунікації з базою даних є наявність пулу з’єднань. Створення нового з’єднання є дорогою операцією, адже вона блокує будь-які інші операції на кластері. Особливо це стає помітно, якщо користувач намагається виконувати багато швидких операцій (наприклад додавання невеликої кількості рядків до бази даних в режимі реального часу). Якщо для кожної такої операції буде відбуватись створення нового з’єднання, то це в рази сповільнює швидкодію програми. Щоб уникнути таких випадків, у ході виконання дослідження до конектора було додано пул з’єднань. Його реалізовано за допомогою бібліотеки DBCP 2.

РОЗДІЛ 3.
ОСОБЛИВОСТІ КОНЕКТОРА

В розділі 1 сформульовано та обґрунтовано функціонал, який повинен підтримувати конектор для того, щоб задовольнити потреби користувачів. В цьому розділі описано ці особливості конектора. Вони є основною причиною для його розробки, адже стандартний Spark JDBC конектор їх не підтримує.

3.1 Паралельне читання даних

Версії конектора 2.x та 3.x мали підтримку паралельного читання з листків SingleStore.

За замовчуванням цей функціонал вимкнений. Щоб його ввімкнути користувачу потрібно встановити параметру enableParallelRead значення true. Якщо паралельне читання ввімкнене, то конектор надсилав до бази даних запит EXPLAIN JSON <SQL> із SQL запитом, що мав би бути виконаний [11].

Із результату запиту EXPLAIN конектор міг дізнатись, яким чином база даних планувала виконувати основний запит. Якщо план виконання не мав ніяких обчислень, що мають бути здійснені на агрегаторах, а тільки такі, що відбуваються на листках, то конектор міг надіслати запити напряму до листків. В окремих випадках це може значно покращити швидкодію виконання запиту. Якщо запит не міг бути прочитаний напряму з листків, то конектор повертався до звичної поведінки і виконував його надсилаючи до агрегаторів і не розпаралелюючи.

Така реалізація мала значну кількість недоліків:

- ✓ при створенні кластера за допомогою Managed Service користувачі не мають доступу до листків, а отже не можуть застосовувати паралельне читання;

- ✓ конектор під’єднується до листків за допомогою тих самих параметрів, що використовуються при з’єднанні із агрегаторами, а отже авторизація має бути однаковою у всьому кластері (однакові паролі);
- ✓ складні запити зазвичай частково обробляються на агрегаторах, а отже не можуть бути прочитані цим способом;
- ✓ даний процес пропускає транзакційну лінію, яка керується агрегаторами і через це, при паралельному виконанні інших запитів в базі даних, користувач ризикує отримати неконсистентні данні.

Для вирішення подібних проблем SingleStore реалізував можливість паралельного читання на стороні бази даних [14].

Результат запиту SELECT можна читати паралельно декількома читачами. Кожен читач звертається до одного сегмента бази даних, що містить частину фінального результату. Для виконання паралельного запиту необхідно виконати наступні кроки:

- ✓ Створити спеціальну таблицю результату за допомогою запиту:
  CREATE [MATERIALIZED] RESULT TABLE <result_table_name> AS SELECT ... FROM <table(s)>;
- ✓ Кожен читач повинен виконати запит для читання відповідного відділу:
  SELECT * FROM ::<result_table_name> WHERE partition_id() = <partition_id>;
- ✓ Після закінчення читання видалити таблицю результату:
  DROP RESULT TABLE <result_table_name>;

Під час першого кроку можна створювати таблиці двох типів:

- MATERIALIZED RESULT TABLE;
- RESULT TABLE.

При створенні MATERIALIZED RESULT TABLE весь результат обраховується і зберігається в оперативній пам’яті бази даних. Якщо вільної пам’яті недостатньо – є ризик, що запит поверне помилку. При цьому сама база даних залишається в робочому стані. З такої таблиці результат можна читати

декілька разів і кожен запит читання із одного сегмента відбувається незалежно від інших.

При створенні RESULT TABLE - результат обчислюється під час читання. За рахунок цього, такі запити використовують набагато менше пам’яті та є швидшими. Їхнім недоліком є те, що результат можна прочитати тільки один раз. Також, читання з одного сегмента може залежати від читання з іншого сегмента. Таким чином, щоб гарантувати відсутність тупикових ситуацій всі користувачі повинні почати читати в один час. Останнє обмеження є дуже важливим для Spark конектора, адже за замовчуванням система Spark очікує, що операції над різними сегментами відбуваються повністю незалежно.

Spark дозволяє додавати функції зворотного виклику для операцій початку або кінця завдань, етапів, програм. Для цього потрібно створити об’єкт що реалізує інтерфейс SparkListener та зареєструвати його. Цей інтерфейс складається із всіх можливих функцій зворотного виклику. За допомогою них, конектор створює та видаляє спеціальні таблиці для паралельного читання. Створення та видалення завжди відбувається в головному потоці програми. Створення таблиці відбувається в момент початку етапу, що містить завдання, які мають читати дані паралельно. Знищення таблиці відбувається в кінці етапу, що містить завдання, які мають читати дані паралельно. Кожне завдання з читання посилає запити до бази даних, чекаючи поки необхідна таблиця буде створена. Якщо таблиця не створюється впродовж довгого часу, то завдання повертає помилку. За допомогою параметрів можна конфігурувати час на очікування створення таблиці.

Є невелика кількість запитів, що поки не підтримують паралельне читання. Для того, щоб перевірити, чи запит може бути прочитаний паралельно - конектор виконує запит EXPLAIN CREATE [MATERIALIZED] RESULT TABLE …

Для того, щоб гарантувати одночасний початок читання з всіх сегментів таблиці створеної за допомогою запиту CREATE RESULT TABLE … конектор використовує RDDBarier [19].

Це експериментальна функція Spark, яка змушує запускати всі завдання певного етапу виконання одночасно. Якщо рівень паралелізму в кластері замалий, щоб почати всі завдання одночасно – буде повернена помилка.

Для розширення можливостей користувача, конектор підтримує аж три види паралельного читання і велику кількість параметрів для його конфігурації. Головним із них є параметр enableParallelRead. Він може набувати наступні значення:

- ✓ disabled - не виконувати паралельне читання;
- ✓ automaticLite - перевірити чи хоча б один з способів паралельного читання вказаних в parallelRead.Features може бути використаний і якщо такий знайдеться, то використати його, інакше прочитати результат непаралельно. Коли використовується ця опція і фінальний результат має бути відсортований - результат читається непаралельно;
- ✓ automatic - перевірити чи хоча б один з способів паралельного читання вказаних в parallelRead.Features може бути використаний і якщо такий знайдеться, то використати його, інакше прочитати результат непаралельно. Коли використовується ця опція і фінальний результат має бути відсортований - сортування завжди відбувається на стороні Spark, а не SingleStore.
- ✓ forced - перевірити чи хоча б один з способів паралельного читання вказаних в parallelRead.Features може бути використаний і якщо такий знайдеться, то використати його, інакше повернути помилку. Коли використовується ця опція і фінальний результат має бути відсортований - сортування завжди відбувається на стороні Spark, а не SingleStore.

  Значенням за замовчуванням є automaticLite.

Способи паралельного читання вказуються в параметрі parallelRead.Features. Цей параметр є списком бажаних способів паралельного читання, розділених комою. Використовуватись буде перший із вказаних способів, всі умови для якого будуть задоволені. Конектор підтримує такі типи паралельного читання:

- ✓ readFromAggregators;
- ✓ readFromAggregatorsMaterialized;
- ✓ readFromLeaves.

За замовчуванням використовується спосіб readFromAggregators.

Коли використовується спосіб readFromAggregators таблиця для паралельного читання створюється за допомогою запиту CREATE RESULT TABLE. Наступні умови мають виконуватись для використання readFromAggregators:

- ✓ версія бази даних 7.5 або вище;
- ✓ параметр database встановлений, або ім’я бази даних вказане в функції load;
- ✓ база даних підтримує паралельне читання згенерованого запиту;
- ✓ рівень паралелізму в Spark кластері достатній, для того, щоб одночасно почати читати з всіх відділів бази даних.

Обмеження часу для створення спеціальної таблиці результату може бути встановлене за допомогою параметру parallelRead.tableCreationTimeoutMS. Значення за замовчуванням – 0, що вказує на відсутність обмежень.

Коли використовується спосіб readFromAggregatorsMaterialized таблиця для паралельного читання створюється за допомогою запиту CREATE MATERIALIZED RESULT TABLE. Цей спосіб використовує більше пам’яті на стороні бази даних. Наступні умови мають виконуватись для використання readFromAggregatorsMaterialized:

- ✓ версія бази даних 7.5 або вище;
- ✓ параметр database встановлений, або ім’я бази даних вказане в функції load;
- ✓ база даних підтримує паралельне читання згенерованого запиту.

Обмеження часу для створення спеціальної таблиці результату може бути встановлене за допомогою параметру parallelRead.materializedTableCreationTimeoutMS. Значення за замовчуванням – 0, що вказує на відсутність обмежень.

Коли використовується спосіб readFromLeaves – конектор намагається читати дані напряму з листків бази даних. При цьому кожен окремий запит бачить незалежну версію стану бази даних, що може спричинити неочікувані результати, якщо таблиця оновлюється в момент читання. Наступні умови мають виконуватись для використання readFromLeaves:

- ✓ параметр database встановлений, або ім’я бази даних вказане в функції load;
- ✓ однаковий спосіб авторизації для всіх вершин (однаковий пароль та користувач) та можливість отримання доступу до вершин-листків;
- ✓ запити не виконують ніякої роботи на агрегаторах.

При використанні readFromAggregators та readFromAggregatorsMaterialized є ризик, що дані між сегментами розподілені нерівномірно. Для того, щоб уникнути таких ситуацій конектор надає параметри parallelRead.repartition та parallelRead.repartitionColumns. Якщо вказаний тільки параметр repartition, то конектор розподілить результат випадковим чином.

Якщо таблиця містить індекси, то розподіл можна зробити по ним вказавши колонки, які повинні використовуватись під час розподілу в параметрі parallelRead.repartitionColumns. Коли встановлений параметр parallelRead.repartition – спеціальна таблиця для читання створюється за допомогою запиту CREATE [MATERIALIZED] RESULT TABLE <result_table_name> REPARTITION BY (...) AS SELECT ... FROM <table(s)>.

В таблиці 3.1. відображено різницю в швидкості читання таблиці паралельно і непаралельно.

Для тестування використовувався кластер SingleStore розміром S-4 (32 сегменти, 4 вершини-листки, 3 вершини-агрегатори). Spark кластер мав 7 вершин-робітників кожен, з яких мав 20 гігабайт оперативної пам’яті та 5 ядер. Таблиця, читання якої тестувалось, містила більше 80 колонок різних типів.

Таблиця 3.1.

Результати тестування швидкості читання

Кількість рядків	Непаралельне читання	Паралельне читання (readFromAggregators)
1000000	58.815 сек	14.371 сек
2000000	120.921 сек	20.893 сек
3000000	183.817 сек	28.184 сек
4000000	OOM	33.433 сек
8000000	OOM	63.595 сек
16000000	OOM	120.738 сек

При читанні 4000000 рядків і більше без використання паралельного читання була отримана помилка. Це сталось через те, що всі дані читаються однією вершиною-робітником, яка має тільки 20G оперативної пам’яті. В таблиці 3.1. цей результат помічений як OOM (Out of memory).

3.2 Конвертація операцій Spark в SQL запити

Ще однією важливою оптимізацією є можливість перетворення операцій над RDD зроблених в Spark на операції реляційної алгебри, що можуть бути виконані всередині бази даних. В деяких випадках це може значно пришвидшити їх виконання. Наприклад, замість того, щоб завантажувати гігабайти інформації в Spark, можна виконати всі обчислення на стороні SingleStore і повернути тільки результат.

Spark перед тим, як виконувати запит оптимізує логічний план. Для цього використовуються об’єкти що реалізують абстрактний клас Rule[LogicalPlan]. Вони приймають дерево логічного плану і повертають його оптимізований варіант. Spark дозволяє користувачам додавати власні оптимізатори. За рахунок цього, конектор має змогу перебудувати логічний план. Якщо користувач встановить значення параметра disablePushdown false, то конектор додасть свій

оптимізатор. Цей оптимізатор бере листки логічного плану, які мають читати із бази даних і намагається замінити їх та їх батьків на одну вершину, що читає складніший запит.

Розглянемо приклад запиту в Spark SQL

select cast(age as byte) from users where age > -129 and age < 128

Побудований для нього логічний план має наступний вигляд:

```
DeserializeToObject createexternalrow(age#214, StructField(age,ByteType,true)),
obj#243: org.apache.spark.sql.Row
+- Project [cast(age#208 as tinyint) AS age#214]
   +- Filter (isnotnull(age#208) AND ((age#208 > -129) AND (age#208 < 128)))
      +- Relation
testdb.users[id#202L,first_name#203,last_name#204,email#205,owns_house#206,fa
vorite_color#207,age#208,birthday#209]
---------------
SingleStore Query
Variables: ()
SQL:
SELECT * FROM testdb.users
```

Рис. 3.1. Логічний план запиту

Після конвертації в SQL він буде виглядати так, як зображено на рис 3.2.

Хоч результат і виглядає громіздким, але насправді він повністю відповідає початковому запиту.

Основна частина функціоналу для конвертації логічного плану в SQL запити була реалізована ще до написання цієї роботи, але нами було добавлено багато нових правил для конвертації конкретних виразів в вершинах логічного плану та виправлено помилок у вже існуючих.

Також, в ході написання роботи досліджено, що запити генерувались недетерміновано. Через це, послідовне виконання одного й того ж запиту могло генерувати різні SQL запити.

Якщо користувач виконував один запит багато разів, то кеш запитів в базі даних міг швидко збільшуватись.

```
DeserializeToObject createexternalrow(age#214, StructField(age,ByteType,true)),
obj#243: org.apache.spark.sql.Row
+- Relation testdb.users[age#214]
---------------
SingleStore Query
Variables: ()
SQL:
SELECT ( ( age#2 !:> TINYINT ) ) AS age#2
FROM (
  -- Spark LogicalPlan: Project [cast(age#208 as tinyint) AS age#214]
  SELECT ( ( age#9 !:> TINYINT ) ) AS age#2
  FROM (
    -- Spark LogicalPlan: Filter (isnotnull(age#208) AND ((age#208 > -129) AND
(age#208 < 128)))
    SELECT *
    FROM (
      SELECT ( id ) AS id#3 , ( first_name ) AS first_name#4 , ( last_name ) AS
last_name#5 , ( email ) AS email#6 , ( owns_house ) AS owns_house#7 , (
favorite_color ) AS favorite_c#8 , ( age ) AS age#9 , ( birthday ) AS birthday#10
      FROM (
        SELECT * FROM testdb.users
      ) AS a2
    ) AS a3
    WHERE ( ( ( age#9 > -129 ) AND ( age#9 < 128 ) ) AND ( age#9 ) IS NOT
NULL )
  ) AS a4
) AS a5
```

Рис. 3.2. Логічний план запиту після конвертації операцій в SQL запити

Це сповільнювало виконання запитів, адже SingleStore мав аналізувати запит кожного разу для визначення оптимального способу обчислення. Щоб уникнути цієї проблеми, алгоритм генерації SQL запитів змінено. Вкінці всі атрибути запиту нормалізувались для гарантування однакового результату при повторному виконанні.

На даний момент, конектор підтримує конвертацію наступних операцій:

- Project;
- Filter;

- Aggregate;
- Window;
- Join;
- Limit;
- Sort.

Крім цього, конектор підтримує конвертацію більше ніж 150 виразів, що можуть зустрічатись в цих операціях (порівняння, конвертація типів, математичні функції, функції для роботи з рядками, функції для роботи з часом).

В деяких випадках, конвертований запит не повністю відповідає оригінальному. Інформацію про такі випадки можна знайти на GitHub репозиторії конектора [9].

3.3 Запис даних із Spark в SingleStore

Конектор підтримує декілька варіантів поведінки запису даних в залежності від того, чи створена таблиця, в яку мають бути додані дані і чи є в ній вже якісь дані. Конфігурувати цю поведінку можна за допомогою встановлення режиму збереження. Spark підтримує чотири режими збереження:

- Append – якщо таблиця вже існує, то дані просто додаються до неї, інакше вона створюється перед додаванням. Під час запису рядків із вже існуючим ключем генерується помилка.
- Overwrite – якщо таблиця вже існує, то дані в ній перезаписуються. Конфігурувати, яким способом дані перезаписуються можна за допомогою параметра overwriteBehavior
- ErrorIfExists – якщо таблиця вже існує, то повертається помилка.
- Ignore – якщо таблиця вже існує, то дані просто додаються до неї, інакше вона створюється перед додаванням. Рядки із вже існуючим ключем ігноруються.

Параметр overwriteBehavior дозволяє вказати, яким саме чином дані мають перезаписуватись при використанні режиму Overwrite. Він може набувати

наступні значення:

- ✓ dropAndCreate – таблиця знищується і створюється заново, після чого дані додаються до неї;
- ✓ truncate – з таблиці видаляються всі рядки, після чого дані додаються до неї;
- ✓ merge – дані з таблиці не видаляються, але якщо необхідно записати рядок з вже існуючим ключем – він перезаписується.

Крім зазначених способів запису даних в процесі виконання цієї роботи реалізовано ще один. За допомогою параметру onDuplicateKeySQL можна вказати спеціальне правило, за яким будуть перераховані рядки з вже існуючим ключем. Це правило має бути SQL виразом, який підтримується в SingleStore.

Розглянемо приклад використання цього параметра:

```
df.write
  .option("onDuplicateKeySQL", "age = age + 1")
  .option("insertBatchSize", 300)
  .mode(SaveMode.Append)
  .save("foo.bar")
```

Рис. 3.3. Приклад запису таблиці з параметром onDuplicateKeySQL

При записі рядка, ключ якого вже існує - колонка age буде перерахована. Її значення буде збільшене на 1.

Зазвичай для завантаження даних використовується запит LOAD DATA [13]. Щоб розпаралелити завантаження, цей запит викликається для кожного сегмента RDD через вершини-робітники Spark кластера. При використанні параметра onDuplicateKeySQL замість запитів LOAD DATA використовується запити INSERT [12] . В такому випадку, кількість рядків в одному запиті можна контролювати за допомогою параметру insertBatchSize. За замовчуванням, цей параметр має значення 10000.

Коли дані завантажуються за допомогою запитів LOAD DATA, користувач може встановити, в якому форматі вони завантажуються та яким чином стискаються. Для встановлення способу стиснення використовується параметр loadDataCompression. Він може набувати наступних значень:

- ✓ GZip;
- ✓ LZ4;
- ✓ Skip (дані не стискаються).

Значенням за замовчуванням є GZip.

Для встановлення формату, до якого конвертуються дані перед завантаженням використовується параметр loadDataFormat. Конектор підтримує два формати:

- ✓ Avro
- ✓ CSV

За замовчуванням використовується формат CSV. Avro дає невеликий виграш у швидкості, але при використанні цього формату неможливе завантаження типів для роботи з часом (TimestampType, DateType).

Якщо користувач планує створити таблицю під час запису даних, він може вказати її тип за допомогою параметра createRowstoreTable (за замовчуванням створюється columnstore таблиця, а при встановленні цього параметра - rowstore). Також можливе створення ключів таблиці за допомогою параметра tableKey. Розглянемо приклад запису даних при якому відбувається створення ключів для таблиці.

```
df.write
  .format("singlestore")
  .option("tableKey.primary", "id")
  .option("tableKey.key.created_firstname", "created, firstName")
  .option("tableKey.unique", "username")
  .mode(SaveMode.Overwrite)
  .save("foo.bar")
```

Рис. 3.4. Приклад запису даних із створенням ключів

В цьому прикладі буде створено три ключі:

- ✓ primary key для колонки id;
- ✓ key для колонок created, firstName, що буде мати назву created_firstname;
- ✓ unique key для колонки username.

Крім типів ключів, згаданих в прикладі, ще може бути використаний ключ для сортування даних в columnstore таблиці (tableKey.columnstore) та ключ для розбиття таблиці на сегменти (tableKey.shard).

В деяких випадках корисною є можливість ігнорувати помилки під час завантаження даних. До прикладу, користувач може завантажувати рядки і припускати, що серед них значення ключа не завжди є унікальним. Конектор підтримує параметр maxErrors. Він вказує на максимальну кількість помилок, що може виникнути під час одного запиту LOAD DATA. Якщо кількість помилок перевищує це значення, то повертається помилка, а інакше всі помилки ігноруються.

Висновки до розділу 3.

Версії конектора 2.x та 3.x підтримували паралельне читання напряму з листкових вершин SingleStore. Хоч цей спосіб і надавав велику превагу в швидкодії, але мав безліч недоліків. Основні з них:

- ✓ при створенні кластера за допомогою Managed Service користувачі не мають доступу до листків, а отже не можуть застосовувати паралельне читання;
- ✓ конектор під’єднується до листків за допомогою тих самих параметрів, що використовуються при з’єднанні із агрегаторами, а отже авторизація має бути однаковою у всьому кластері (однакові паролі);
- ✓ складні запити зазвичай частково обробляються на агрегаторах, а отже не можуть бути прочитані цим способом;

- ✓ даний процес пропускає трансзкційну лінію, яка керується агрегаторами і через це, при паралельному виконанні інших запитів в базі даних, користувач ризикує отримати неконсистентні данні.

Щоб покращити процес паралельного читання в конекторі реалізовано ще два способи паралельного читання, при яких дані читаються з агрегаторів, але за допомогою декількох запитів. На даний момент, конектор підтримує три типи паралельного читання: readFromLeaves, readFromAggregators, readFromAggregatorsMaterialized.

Для пришвидшення виконання запитів, конектор підтримує механізм, що конвертує операції Spark в SQL запити і за рахунок цього переносить обчислення на сторону бази даних. Це значно зменшує обсяг даних, що пересилаються між системами. В ході виконання роботи, було додано багато нових перетворень і виправлені помилки у вже існуючих. На даний момент, конектор підтримує конвертацію наступних операцій: Project, Filter, Aggregate, Window, Join, Limit, Sort.

Крім цього, конектор підтримує конвертацію більше ніж 150 виразів, що можуть зустрічатись в цих операціях (порівняння, конвертація типів, математичні функції, функції для роботи з рядками, функції для роботи з часом).

Конектор має велику кількість конфігурацій для запису даних. За допомогою них можна встановити, яким чином буде створюватись таблиця, як будуть зберігатись рядки з вже існуючим ключем, який формат і спосіб стиснення використовувати під час надсилання даних. В більшості випадків, конектор використовує декілька запитів LOAD DATA для кожного сегмента реляції для того, щоб надіслати дані.

Розроблено спосіб зберігання даних, при якому користувач може вказати правило, за допомогою якого при завантаженні перераховуються рядки з вже існуючим ключем. При використанні цього способу, конектор використовує запити INSERT INTO замість LOAD DATA.

РОЗДІЛ 4.
ТЕСТУВАННЯ ТА РОЗПОВСЮДЖЕННЯ КОНЕКТОРА

Даний розділ містить інформацію про процес тестування, створення релізів конектора та демо для презентації конектора потенційним користувачам. Якісне тестування допомагає підтримувати проєкт у майбутньому. Зручний спосіб встановлення і просте та інформативне демо допомагають користувачам оцінити переваги продукту.

4.1 Тестування конектора

Тестування є важливою частиною розробки. Воно дозволяє знайти недоліки в програмному забезпеченні, а також переконатись, що нові зміни не ламають вже існуючий функціонал. У випадку створення конекторів воно ще дозволяє переконатись, що введені зміни працюють коректно для різних версій систем. До прикладу, якщо розробник зробив певні зміни, то він має перевірити, що вони коректно працюють для всіх версій Spark та всіх версій SingleStore, що підтримуються на момент розробки. При релізі нової версії однієї з систем, вони часто виявляють проблеми сумісності, що мають бути вирішені, щоб користувачі змогли безперешкодно користуватись конектором.

Під час розробки SingleStore Spark конектора особлива увага надавалась тестуванню. Для тестування використовувався локальний SingleStore кластер розгорнутий в docker контейнерах [6].

Цей кластер складається із одного головного агрегатора, одного звичайного і одного листка. В кластері є спеціальна база даних та таблиці для тестування. Також, цей кластер має декількох користувачів із різними способами авторизації. Для того, щоб швидко підняти кластер і встановити всі необхідні для тестування конфігурації в проекті є спеціальний bash скрипт. З його допомогою, будь-який розробник може розгорнути в себе базу даних для тестування за лічені хвилини.

Для зручності розробки проект також тестується на сервісах неперервної інтеграції. Спочатку все тестування відбувалось на сервісі CircleCI [4]. З часом кількість тестів збільшилась до таких розмірів, що запуск всіх тестів тривав більше години. Через це систему неперервної інтеграції було змінено на webappio [22] . Це дозволило пришвидшити час виконання тестів. На даний момент виконання тестів знову збільшилось до ~40 хвилин. Тести запускаються кожного разу, коли розробник відправляє локальні зміни в віддалений репозиторій.

На даний момент, конектор має 1029 тестів. Більшість з них це тести для перетворення Spark операцій в SQL запити. В певний момент, розробники, що працювали над проектом помітили, що існуючі перетворення некоректно працювали в крайніх випадках. Для покращення якості перетворень, прийняли рішення написати по декілька тестів із крайніми значеннями для кожної операції, перетворення якої підтримується. Впродовж цього процесу знайшлась величезна кількість помилок.

Конектор тестується із наступними версіями SingleStore:

- 7.1
- 7.3
- 7.5
- 7.6
- 7.8

Конектор тестується із наступними версіями Spark:

- 3.0
- 3.1
- 3.2

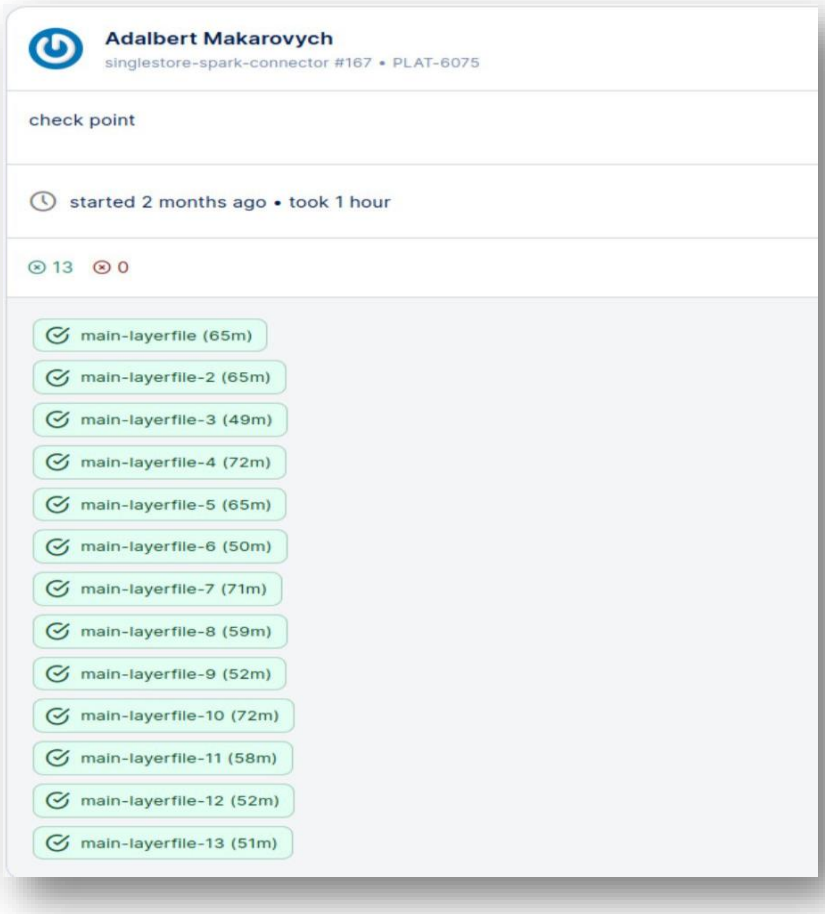

Рис 4.1. Проходження тестів конектора на Webapp.io

На зображенні продемонстровано результат виконання тестів. У випадку помилки - розробник може переглянути детально логи а також отримати SSH доступ до контейнера всередині якого виконувались тести.

4.2 Спосіб розповсюдження конектора та процес створення релізів

Зручний спосіб завантаження та використання конектора напряму впливає на те, скільки користувачів зможуть ним скористатись. Весь код конектора доступний у відкритому GitHub репозиторії: https://github.com/memsql/singlestore-spark-connector. Будь-хто може його переглянути та за потреби запропонувати зміни. Конектор розповсюджується на

основі ліцензії Apache 2. Ця ліцензія дозволяє іншим розробникам копіювати, перевикористовувати та змінювати код проекту.

Релізи конектора публікуються на Maven Central та Spark Packages [https://search.maven.org/artifact/com.singlestore/singlestore-spark-connector_2.12 [15], https://spark-packages.org/package/memsql/memsql-spark-connector [17].

За рахунок наявності конектора на Spark Packages, його можна додати до Spark застосунку за допомогою spark-shell, pyspark, spark-submit. Приклад коду, що запускає spark-shell із конектором до SingleStore:

$SPARK_HOME/bin/spark-shell --packages com.singlestore:singlestore-spark-connector_2.12:4.0.0-spark-3.1.0

За рахунок наявності конектора на Maven Central, його можна легко додати до будь-якого проекту, що збирається за допомогою наступних інструментів: Maven, Gradle, SBT, Ivy, Grape, Leiningen, Buildr. Приклад додавання конектора в проект, що збирається за допомогою Maven:

```
<!-- https://mvnrepository.com/artifact/com.singlestore/singlestore-spark-connector -->
<dependency>
   <groupId>com.singlestore</groupId>
   <artifactId>singlestore-spark-connector_2.12</artifactId>
   <version>4.0.0-spark-3.2.0</version>
</dependency>
```

Рис. 4.2. Приклад додавання конектора в проект, що збирається за допомогою Maven

Також, розробники мають можливість завантажити JAR файл проекту напряму із Maven Central.

Під час створення релізу нової версії конектора створюється відразу декілька білдів для кожної версії Spark, що підтримується.

Розглянемо приклад номера версії конектора: com.singlestore:singlestore-spark-connector_2.12:4.0.0-spark-3.1.0

Він вказує, що конектор створений за допомогою Scala 2.12, версія самого конектора 4.0.0, а версія Spark для якої він створений 3.1.0. Потреба створення різних білдів конектора для різних версій Spark зумовлена тим, що різні версії Spark можуть мати різні операції і відповідно, конектор повинен їх по різному перетворювати в SQL запити.

Процес створення релізів автоматизований і відбувається за допомогою CircleCI. Автоматизація цього процесу значно полегшує роботу розробників та мінімізує можливість помилок під час такого важливого процесу. Для створення релізу, розробник має створити тег у віддаленому репозиторії і підтвердити створення релізу в інтерфейсі CircleCI. Система перевірить, що новий реліз проходить всі тести, згенерує всі необхідні файли та додасть їх в Maven Packages.

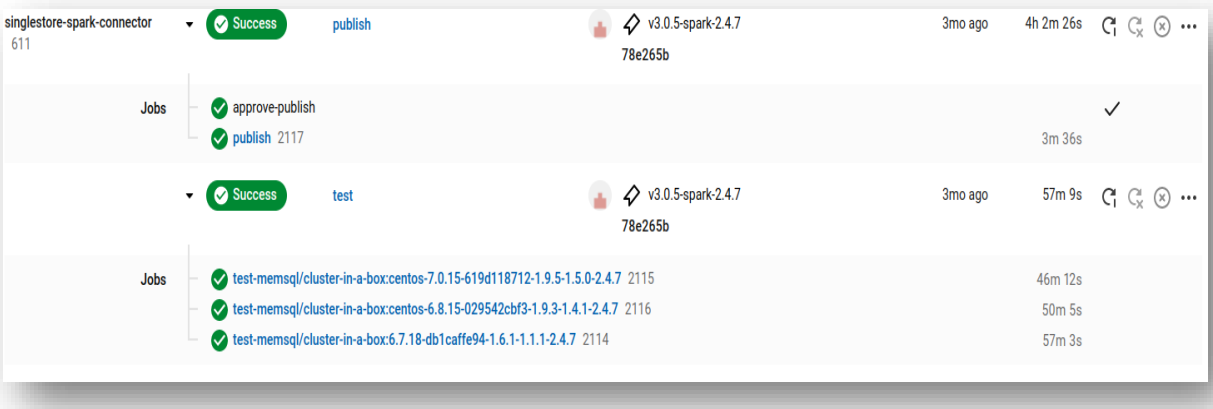

Рис 4.3. Створення релізу конектора на CircleCI

4.3 Демо для презентації конектора

Для того, щоб продемонструвати користувачам використання конектора, в ході виконання цієї роботи створено невелике демо. Основними вимогами до нього були простота запуску, відсутність потреби встановлення додаткового програмного забезпечення та наявність прикладів на декількох різних мовах. Користувач має мати можливість побачити конектор в дії не встановлюючи його для того, щоб вирішити чи конектор покриває його потреби.

Для створення демо використано технологію Apache Zeppelin. Apache Zeppelin – це веб-блокнот, що надає можливості для керування даними, інтерактивної аналітики, створення спільних документів на мовах SQL, Scala, Python, R, тощо [1]. Щоб не змушувати користувача завантажувати додаткове програмне забезпечення Apache Zeppelin запускається в docker контейнері. Крім Zeppelin, для демо необхідно запустити ще один docker контейнер із базою даних.

Кроки запуску демо:

- ✓ завантажити репозиторій конектора https://github.com/memsql/singlestore-spark-connector/tree/master/demo і перейти в каталог demo
- ✓ створити docker мережу, щоб могти з'єднати Appache Zeppelin і SingleStore

  docker network create zeppelin-ciab-network
- ✓ завантажити docker зображення із SingleStore

  docker pull memsql/cluster-in-a-box
- ✓ запустити SingleStore кластер

```
docker run -i --init \
--name singlestore-ciab-for-zeppelin \
-e LICENSE_KEY=[INPUT_YOUR_LICENSE_KEY] \
-e ROOT_PASSWORD=my_password \
-p 3306:3306 -p 8081:8080 \
--net=zeppelin-ciab-network \
memsql/cluster-in-a-box
```

```
docker start singlestore-ciab-for-zeppelin
```

- ✓ створити docker зображення для Zeppelin

  docker build -t zeppelin .
- ✓ запустити Apache Zeppelin

```
docker run -d --init \
--name zeppelin \
```

-p 8082:8082 \

--net=zeppelin-ciab-network \

-v $PWD/notebook:/opt/zeppelin/notebook/singlestore \

-v $PWD/notebook:/zeppelin/notebook/singlestore \

zeppelin

Після цих кроків користувач може відкрити будь-який браузер і побачити в ньому блокноти із прикладами використання конектора на трьох мовах:

- Scala (посилання http://localhost:8082/next/#/notebook/2F8XQUKFG);
- Python (посилання http://localhost:8082/next/#/notebook/2F6Y3APTX);
- SQL (посилання http://localhost:8082/next/#/notebook/2F7PZ81H6).

Кожен блокнот починається із встановлення параметрів з'єднання та детального їх пояснення

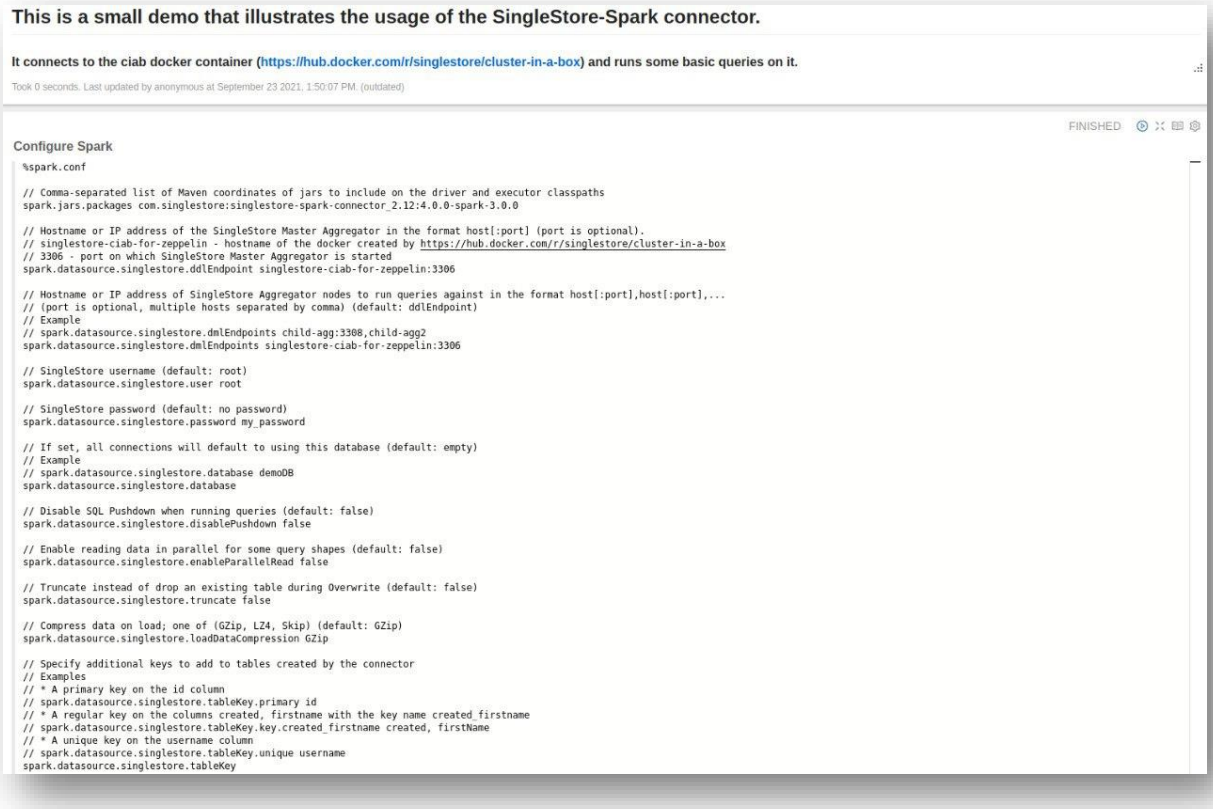

Рис. 4.4. Конфігурація конектора

Після цього створюється база даних та з'єднання

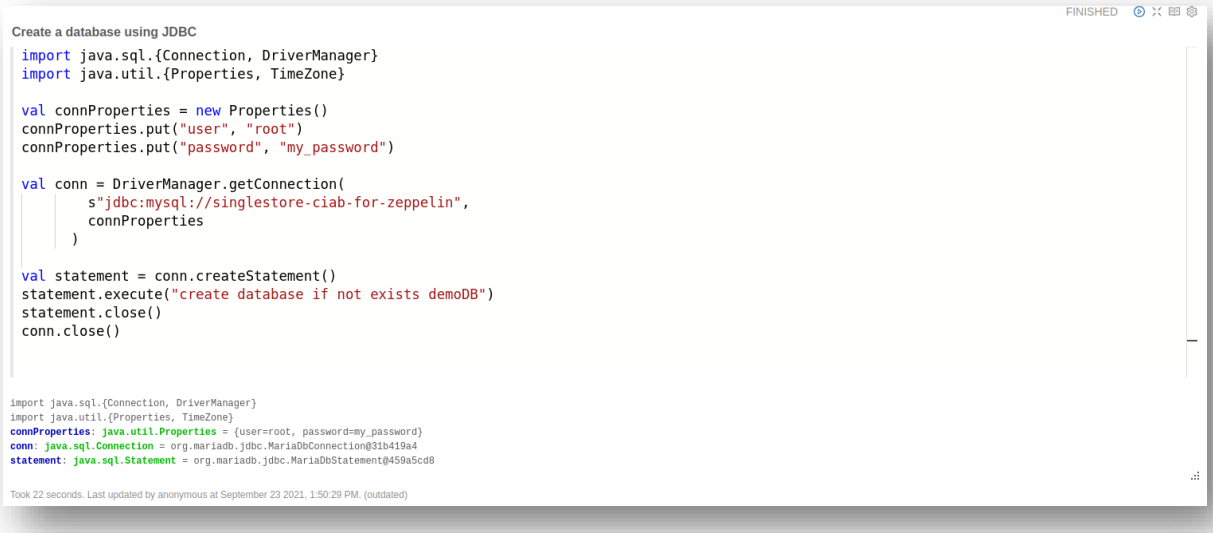

Рис. 4.5. Створення бази даних та з'єднання

Вкінці блокнота демонструється запис даних із Spark в SingleStore та читання даних із SingleStore в Spark

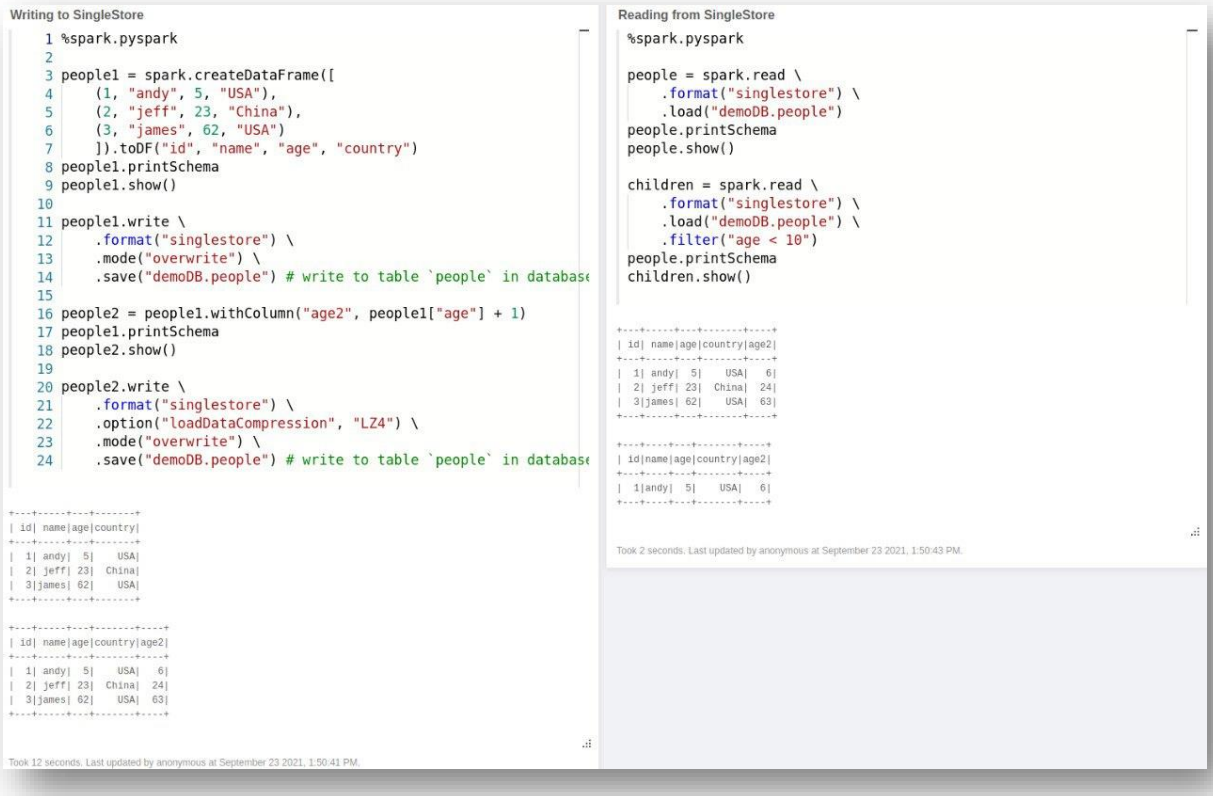

Рис. 4.6. Читання та запис даних в SingleStore

Висновки до розділу 4.

Під час розробки SingleStore Spark конектора особлива увага надавалась тестуванню. Для тестування використовувався локальний SingleStore кластер розгорнутий в docker контейнерах.

Для зручності розробки, проєкт тестується на сервісах неперервної інтеграції. Спочатку все тестування відбувалось на сервісі CircleCI. З часом кількість тестів збільшилась до таких розмірів, що запуск всіх тестів тривав більше години. Через це систему неперервної інтеграції було змінено на webappio .

На даний момент, конектор має 1029 тестів.

Весь код конектора доступний у відкритому GitHub репозиторії https://github.com/memsql/singlestore-spark-connector .

Релізи коенктора публікуються на Maven Central та Spark Packages

Під час створення релізу нової версії конектора створюється відразу декілька білдів для кожної версії Spark, що підтримується.

Розроблено демо для демонстрації можливостей конектора. Воно запускається в docker контейнерах і не потребує завантаження додаткового програмного забезпечення. Демо використовує блокноти Apache Zeppelin. Демо містить три блокноти на наступних мовах:

- ✓ Scala;
- ✓ Python;
- ✓ SQL.

Кожен блокнот демонструє конфігурацію конектора, читання та запис даних в SingleStore.

ВИСНОВКИ

В рамках кваліфікаційної роботи досліджено проблемні питання використання конекторів та розроблено конектор між технологіями SingleStore та Spark.

У роботі виконано поставлені завдання.

- ✓ Проведено аналіз особливостей SingleStore та Spark і ситуацій, в яких вони можуть використовуватись разом.
- ✓ На основі проведеного аналізу сформульовано вимоги, необхідні для задоволення потреб користувачів.
- ✓ Досліджено конвертацію операцій Spark в SQL запити та розширено кількість підтримуваних перетворень.
- ✓ Проведено аналіз способів паралельного читання із SingleStore. Додано два нові види паралельного читання.
- ✓ Специфіковано конфігурації запису даних в SingleStore. Реалізовано спосіб паралельного запису даних, який дозволяє вказувати правило, що використовується для перерахунку рядків із повторюваним ключем.
- ✓ Визначено параметри конектора та наведено поради щодо його конфігурації.
- ✓ Забезпечено покриття конектора тестами. Перенесено систему тестування із CircleCI на Webapp.io.
- ✓ Створено демо-проєкти конектора на 3 мовах (Scala, Python, SQL).

Результатом кваліфікаційної роботи є розроблений конектор та проведений аналіз пов’язаний з його розробкою та використанням.

Мета по дослідженню проблемних питання використання конекторів та розробці конектора була досягнута.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Apache Zeppelin: веб-сайт. [Електронний ресурс]. – Режим доступу: https://zeppelin.apache.org/ (дата звернення 10.05.2022).
2. Appache Commons. The DBCP Component. [Електронний ресурс]. – Режим доступу: https://commons.apache.org/proper/commons-dbcp/index.html (дата звернення 17.05.2022).
3. Carol McDonald. How Spark Runs Your Applications. [Електронний ресурс]. – Режим доступу: https://developer.hpe.com/blog/how-spark-runs-your-applications/#:~:text=The%20Spark%20execution%20model%20can,driver%2Dnode%3E%3A4040. (дата звернення 17.05.2022).
4. CircleCI. : веб-сайт. [Електронний ресурс]. – Режим доступу: https://circleci.com/
5. Martin Kleppmann. Designing Data-Intensive Applications The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. – O'Reilly Media, Inc. – Sebastopol. – 2017. – P.590.
6. MemSQL Official Docker Image. [Електронний ресурс]. – Режим доступу: https://hub.docker.com/r/memsql/cluster-in-a-box. (дата звернення 11.05.2022).
7. MySQL. [Електронний ресурс]. – Режим доступу: https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_PROTOCOL.html (дата звернення 17.05.2022).
8. SingleStore Documentation : веб-сайт. [Електронний ресурс]. – Режим доступу: https://docs.singlestore.com/managed-service/en/load-data/load-data-from-a-data-source/load-data-from-amazon-web-services--aws-/load-data-from-aws-glue.html (дата звернення 16.05.2022)
9. Singlestore spark connector. SQL Pushdown Incompatibilities. [Електронний ресурс]. – Режим доступу: https://github.com/memsql/singlestore-spark-connector#sql-pushdown-incompatibilities (дата звернення 18.05.2022).

10. SingleStore. : веб-сайт. [Електронний ресурс]. – Режим доступу: https://docs.singlestore.com/ (дата звернення 10.05.2022)
11. SingleStore. DB Software v7.8. [Електронний ресурс]. – Режим доступу: https://docs.singlestore.com/db/v7.8/en/reference/sql-reference/data-manipulation-language-dml/explain.html (дата звернення 17.05.2022)
12. SingleStore. DB Software v7.8. INSERT. [Електронний ресурс]. – Режим доступу: https://docs.singlestore.com/db/v7.8/en/reference/sql-reference/data-manipulation-language-dml/insert.html. (дата звернення 18.05.2022).
13. SingleStore. DB Software v7.8. LOAD DATA. [Електронний ресурс]. – Режим доступу: https://docs.singlestore.com/db/v7.8/en/reference/sql-reference/data-manipulation-language-dml/load-data.html. (дата звернення 18.05.2022).
14. SingleStore. Read Query Results in Parallel. [Електронний ресурс]. – Режим доступу: https://docs.singlestore.com/db/v7.8/en/query-data/query-procedures/read-query-results-in-parallel.html (дата звернення 17.05.2022).
15. Sonatype. Maven Central Repository Search. Singlestore spark connector. [Електронний ресурс]. – Режим доступу: https://search.maven.org/artifact/com.singlestore/singlestore-spark-connector_2.12
16. Spark : веб-сайт. [Електронний ресурс]. – Режим доступу: https://spark.apache.org/docs/latest/index.html (дата звернення 17.05.2022)
17. Spark Packages. [Електронний ресурс]. – Режим доступу: https://spark-packages.org/package/memsql/memsql-spark-connector (дата звернення 18.05.2022).
18. Spark. Class RDD Barrier. [Електронний ресурс]. – Режим доступу:https://spark.apache.org/docs/latest/api/java/org/apache/spark/rdd/RDDBarrier.html (дата звернення 17.05.2022).

19. Spark. RDD Programming Guide. [Електронний ресурс]. – Режим доступу: https://spark.apache.org/docs/latest/rdd-programming-guide.html (дата звернення 17.05.2022).
20. Synoptic : веб-сайт. [Електронний ресурс]. – Режим доступу: https://medium.com/synaptic-tech/why-we-chose-singlestore-as-our-analytics-database-11742643df34 (дата звернення 17.05.2022)
21. The Universal Data Connectivity Platform. [Електронний ресурс]. – Режим доступу: https://www.cdata.com/ (дата звернення 10.05.2022).
22. Webappio. [Електронний ресурс]. – Режим доступу: https://webapp.io/docs/why-us. (дата звернення 10.05.2022).