Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерних наук_____
(повна назва)

Кафедра_____Програмної інженерії_____

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

_____другий (магістерський)_____
(рівень вищої освіти)

Дослідження методів обробки потоків даних у Big Data
(тема)

Виконав: студент 2 курсу, групи ІПЗм-17-1___
спеціальності 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)
Освітньо-наукової програми
Інженерія програмного забезпечення_____
(повна назва освітньої програми)

___Рукавиця А.С._____
(прізвище, ініціали)
Керівник _____проф. Руткас А.Г._____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри, проф. _____ З.В.Дудар

2019 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121– Інженерія програмного забезпечення
(код і повна назва)

Освітньо-професійна програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

Студентові Рукавиці Артему Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження методів обробки потоків даних у Big Data

затверджена наказом по університету від «09» листопада 2018 р № 1592 Ст

2. Термін подання студентом роботи до екзаменаційної комісії «_» червня 2019 р.

3. Вихідні дані до роботи *алгоритми обробки потоків даних, пояснювальна записка, емулятор Bid Data екосистеми близької до реального часу, середовище об'єктно орієнтованого проектування*

4. Перелік питань, що потрібно опрацювати в роботі *позначка роботи, аналіз проблемної галузі і постановка задачі, опис проблеми обробки потоків даних, використовувані методи та алгоритми, опис розробленої програмної системи*

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) *Мета завдання, обґрунтування доцільності*

розроблення, постановка задачі, об'єктна модель системи, базові моделі, результати тестування продуктивності програмної системи, демонстраційні матеріали

6 Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	проф. Руткас А.Г.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка *
1.	Аналіз предметної галузі	10 квітня 2019р.	
2.	Огляд існуючих методів	23 квітня 2019р.	
3.	Методи підвищення ефективності веб-додатків	28 квітня 2019р.	
4.	Підготовка пояснювальної записки	18 квітня 2019р.	
5.	Спецчастина	10 травня 2019р.	
6.	Підготовка презентації та доповіді	22 травня 2019р.	
7.	Попередній захист	28 травня 2019р.	
8.	Нормоконтроль, рецензування	3 червня 2019р.	
9.	Занесення диплома в електронний архів	3 червня 2019р.	
10.	Допуск до захисту у зав. кафедри	5 червня 2019р.	
* заповнюється вручну після виконання чергового пункту			

Дата видачі завдання _____ _____2019 р.

Студент _____
(підпис)

Керівник роботи _____ __проф. Руткас А.Г._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до атестаційної роботи: 82 с., 10 рис., 2 додатки, 20 джерел.

ВЕЛИКІ ДАНІ, ОБЧИСЛЕННЯ БЛИЗЬКІ ДО РЕАЛЬНОГО ЧАСУ, ХМАРНИ ОБЧИСЛЕННЯ, HADOOP, APACHE SPARK, SPARK STREAMING, SCALA, JAVA, AMAZON WEB SERVICES.

Об'єктом дослідження є алгоритми обробки потоків даних в умовах обчислень близьких до реального часу, Big Data інфраструктура з використанням Apache Spark в хмарному середовищі.

Метою роботи є дослідження алгоритмів обробки потоків даних та будови Big Data системи Spark Streaming, порівняння патернів обробки потоку даних з інкрементальними та батч підходами.

Методи розробки базуються на мовах програмування Scala, Java та Python. Також використовуються наступні методи розробки: паралельні та розподілені обчислення, об'єктно-орієнтований та функціональний підхід розробки.

У результаті роботи були досліджені переваги та недоліки алгоритмів обробки потоків даних, структура системи Apache Spark з використанням хмарних обчислень в AWS.

BIG DATA, NEAR REAL TIME COMPUTING, CLOUD COMPUTING, HADOOP, APACHE SPARK, SPARK STREAMING, SCALA, JAVA, AMAZON WEB SERVICES.

The object of research are the algorithms of the processing of data flows in near real-time conditions and Big Data infrastructure with Apache Spark with leveraging of cloud computing.

The aim is the development of the application for research of algorithms for processing of data streams and research structure of Big Data system with applying Spark Streaming, comparison streaming processing with the incremental and batch engines.

Methods of development are based on Scala, Java, and Python languages. Also, next development approaches are used: parallel and distributed computing, object-oriented and functional programming.

As a result of the research, algorithms for processing/streaming data and the structure of Apache Spark with leveraging cloud computing with AWS were investigated,

ПЕРЕЛІК СКОРОЧЕНЬ

БД	-	база даних;
ІС	-	інформаційні системи;
ОС	-	операційна система;
AWS	-	Amazon web services;
BD	-	Big data;
SQL	-	Structured query language.

ЗМІСТ

ВСТУП

У сучасному світі, стрімко розвиваються технологій, потік інформації набирає обертів і відграє все важливішу роль, у людства як ніколи багато можливостей та інструментів для поліпшення якості власного життя, в тому числі завдяки отриманню цінної інформації з даних.

Розуміння запитів користувача і таргетинг - одні з найбільш великих і максимально освітлених широкому загалу областей застосування інструментів роботи з особистою інформацією. Методи BD допомагають аналізувати клієнтські звички, щоб в подальшому краще розуміти запити споживачів. Компанії прагнуть розширити традиційний набір даних інформацією з соціальних мереж і історією пошуку браузера з метою формування максимально повної клієнтської картини. Іноді великі організації вибирають глобальною метою створення власної рекомендаційної моделі.

Загальновідомим фактом є те, що Агентство національної безпеки Сполучених Штатів застосовує програмні інструменти, щоб запобігти терористичним актам. Інші відомства задіють прогресивну методологію, щоб запобігати більш дрібним злочини. Департамент поліції Лос-Анджелеса застосовує власну аналітичну систему для схожих цілей. Вона займається тим, що зазвичай називають проактивною охороною правопорядку. Використовуючи звіти про злочини за певний період часу, алгоритм визначає райони, де ймовірність скоєння правопорушень є найбільшою.

Використання IT-технологій в медичній сфері дозволяє лікарям більш ретельно вивчити хвороба і вибрати ефективний курс лікування для конкретного випадку. Завдяки аналізу об’ємів інформації з використанням підходів BD,

медпрацівникам легшає передбачати рецидиви і вживати превентивні заходи. Як результат - більш точна постановка діагнозу і вдосконалені методів лікування.

У всіх перерахованих прикладах, що застосовуються в найважливіших областях людської життєдіяльності, є одна спільна риса – в основі кожної лежить структурування і обробка великої кількості даних. За будь-яким спрямованим перетворенням, безсумнівно, стоїть попередній аналіз. Він дозволяє, нам планувати та прогнозувати майбутнє, аналізувати минуле, приймати рішення. На практиці, ми маємо лише список фактів і набір оціночних критеріїв, що дозволяють отримати корисну інформацію завдяки використанню алгоритмів аналізу та обробки даних. Усвідомлення останнього факту поклало початок такому дивовижному напрямку IT індустрії, як BD.

Час минав, і дана технологія знайшла своє застосування майже в кожній області людської діяльності - її популярність набирає обороти, її актуальність, і застосування не залишає сумнівів.

Саме підвищений інтерес до методів і технологій розробки програмних продуктів, що виконують обчислення на розподілених кластерах, став підказкою до вибору об'єкта дослідження для даної роботи - розподілена обробка даних, як сфера технічних наук, набирає популярність і потребує залучення нових фахівців для розвитку і прогресу. Як не була б юна дана галузь IT-індустрії, області та напрямки її застосування вже встигли значно розростися, тому, для аналізу і вивчення в контексті даної роботи була обрана більш вузька, але популярна тема, розподілена обробка потоків даних у вигляді неструктурованих або слабоструктурованих даних (прикладом можуть послужити результати соціологічних опитувань, наукових експериментів, результати роботи датчиків, безпілотних автомобілів).

1. АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Постановка задачі

Сьогодні людство живе в інформаційний вік. Не просто виміряти загальний обсяг інформаційних даних, але за оцінками IDC розмір «цифрового всесвіту» в 2006 році становив 0.18 зеттабайт, а до 2011 р повинен був досягти 1.8 зеттабайт, показавши десятикратне зростання за 5 років, ось кілька прикладів джерел таких обсягів:

- Нью-Йоркська фондова біржа виробляє близько терабайта даних на добу;
- обсяг сховища всього однієї соціальної мережі Facebook кожен день збільшується на 500 терабайт;
- Internet Archive вже сьогодні зберігає 2 петабайта даних і отримує 20 терабайт кожного місяця;
- наукові експерименти на великому адронному колайдері можуть виробляти близько петабайта даних в секунду.

Стрімке збільшення обсягу інформації ставить перед нові складні завдання по організації збереження і обробки даних.

Існує велика кількість методів класифікацій даних і методів їх обробки. Слід зрозуміти, що відрізняє характеристики і методи BD від інших. Нерідко використовується характеристика, позначена дослідницькою компанією Gartner: «BD характеризується своїм обсягом, різноманітністю і швидкістю, з якою структуровані, слабоструктуровані і неструктуровані дані надходять по мережах передачі в процесори і сховища, поряд з процесами перетворення цих даних в цінну для бізнесу або науки інформацію».

Як видно з вищевказаного визначення, великі дані мають всього чотири основні характеристики: обсяг, різноманітність, швидкість і цінність. Розглянемо докладніше ці характеристики:

- обсяг та кількість даних, вироблених як людьми, так і технікою, стрімко зростає та пред'являє до IT інфраструктури нові вимоги щодо збереження, обробки і надання доступу;
- різноманітність, дані містять цілковито різну інформацію, представлену різноманітними структурами. З усім цим, від типових логів доступу до веб-сервера до транзакцій по кредитним картам, від результатів наукових експериментів до фотографій і відео, необхідно вміти взаємодіяти;
- швидкість, важливо усвідомлювати, що під швидкістю мається на увазі не тільки швидкість, з якою дані надходять в якесь сховище, а й швидкість з якою найбільш важлива інформація з цих даних витягується;
- цінність, великі обсяги даних - це, безсумнівно, цінний ресурс, але що важливіше, він дозволяє відповідати на нагальні питання або вирішувати проблеми, які можуть виникнути в майбутньому.

У сучасному світі все частіше виникає необхідність вирішення трудомістких і ресурсоємних обчислювальних задач. Системи розподіленої обробки інформації стали тим інструментом, який підвищує ефективність вирішення подібних завдань. Під розподіленою обробкою інформації розуміється комплекс операцій з інформацією, що проводиться на незалежних, але пов'язаних між собою обчислювальних машинах, призначених для виконання спільних завдань. Системи розподіленого обробки даних легко масштабуються та дозволяють скоротити час обробки даних за рахунок високого рівня паралельної обробки. Прикладом системи розподіленої обробки даних є система Spark, це платформа з відкритим вихідним кодом, яка використовує кластерні обчислення для аналізу даних. Можливість проводити кластерні обчислення в

пам'яті, які дає Spark дозволяє більш ефективно реалізовувати ітеративні алгоритми і вирішувати завдання інтерактивної обробки даних.

Spark переважно реалізований на функціональній мові Scala і використовує його в якості середовища розробки додатків. Ця мова дозволяє легко маніпулювати розподіленими наборами даних як локальними об'єктами

Так вийшло, що інструменти, які існували до недавнього часу, виявилися не придатними для того, щоб справлятися з великими обсягами даних. Саме про проблеми, які прийшли з епохою великих даних, способи їх подолання та новітні інструменти піде мова в цьому розділі. Вимоги, що пред'являються до всіх сучасних інформаційних систем та розвиток все нових інформаційних технологій призводять до безперервного зростання складності розробки та подальшої експлуатації таких систем. Сучасні інформаційні системи характеризуються такими особливостями:

- складність опису, що передбачає ретельне моделювання та аналіз даних і процесів;
- наявність сукупності тісно взаємодіючих підсистем, що мають свої локальні підзадачі;
- необхідність наявності способів інтеграції існуючих і розроблених нових додатків;
- функціонування в неоднорідному середовищі на декількох апаратних платформах;
- різнорідність технічних засобів проектування і розробки ІС.

Аналіз сучасних тенденцій і шляхів розвитку нових інформаційних систем свідчить про значне ускладнення використовуваних в них структур даних при одночасному зростанні кількості збереженої і аналізованої інформації. Для обробки все зростаючого кількості даних потрібно нове апаратне і відповідне програмне забезпечення, оскільки існуюче вже не справляється з поставленими завданнями.

Таким чином, виникла ідея об'єднання обчислювальних потужностей двох або більше машин для вирішення складних, непосильних для кожної з них окремо завдань. Виникнення локальних мереж призвело до розвитку нової сфери розробки програмного забезпечення - створення розподілених додатків. Інтерес до таких додатків з'явилася, перш за все, у найбільших компаній, структура бізнесу яких потребувала подібних рішеннях. Саме на етапі конструювання корпоративних розподілених систем були сформовані основні вимоги і розроблено типові архітектури подібних додатків, використовувані і в даний час.

З плином часу мейнфрейми і термінали помітно еволюціонували, - в напрямку архітектури «клієнт-сервер», яка була, першим варіантом розподіленої архітектури, тобто дворівневої розподіленою системою. Так, саме в додатках, «клієнт-сервер» частина обчислювальних перетворень і «бізнес-логіки» були перенесені на сторону клієнта, що, власне, і лягло в основу цього підходу до розподіленої обробки даних.

Саме в той період стало очевидно, що головними перевагами розподілених додатків є:

– доступна масштабованість;
– інструменти управління навантаженням;
– виражена глобальність.

Під масштабованістю мається на увазі можливість збільшення обчислювальних потужностей розподіленого додатка без зміни його базової структури. Можливість контрольованого управління навантаженням дозволяє проміжним етапам розподіленого додатка управляти потоками запитів користувачів і направляти їх менш завантаженим серверам для обробки. Розподілена структура дає можливість слідувати розподілу виконанню бізнес-процесів і конструювати клієнтські робочі місця в найбільш зручних точках, що дає можливість вирішення більш глобальних завдань.

Поява мережі Internet відіграло важливу роль у розвитку розподілених обчислень і висунуло цю область розробки програмного забезпечення на новий рівень, зробивши предметом інтересу професійних програмістів. Сьогодні Internet значно розширює можливості застосування розподілених додатків, дозволяючи підключати віддалених користувачів і роблячи функціонал додатка доступним повсюдно.

Як вже позначалося вище, головними чинниками прогресу в сфері IT завжди є провідні корпорації, так і в даному випадку, кажучи про прогрес у даній області, необхідно розглянути лише конкретні шляхи розвитку декількох напрямків, визнаними світовими стандартами де-факто.

1.2 Дослідження методів аналізу даних в BD

В світі вже з'явилася компанія, яка, зі змінним успіхом, справляється з проблемою BD - Google. Жодна комерційна або некомерційна організація не оперує більшим об'ємом даних, ніж Google. Саме Google була основним контрибутором ідей платформи Hadoop, а також багатьох інших компонентів екосистеми Hadoop, таких як HBase, Apache Giraph, Apache Drill. Умовно історію розвитку BD рішень в Google можна розділити на 2 періоди: 1-ий етап (2003-2008), в цю фазу були описані набір принципів і концепцій, які сьогодні де-факто є стандартом у світі обробки величезних обсягів даних (на commodity-обладнанні), 2-ий етап (з 2009 по даний момент): були сформовані і описані технології обробки даних, які, зі значною часткою ймовірності, будуть використовуватися для рішень BD завдання вже в недалекому майбутньому. У період 2003-2008 інженерами Google були описані і опубліковані у відкритому доступі статті про трьох системах, які в Google використовують для вирішення своїх завдань:

- Google File System (GFS) - розподілена файлова система [1];
- Bigtable [1] - високопродуктивна і надійна база даних, орієнтована на зберігання петабайт даних;
- MapReduce [1] - програмна модель, призначена для якісної розподіленої обробки великих обсягів даних.

Вплив робіт, опублікованих Google, на перші кроки становлення галузі Big Data неможливо переоцінити. Найбільш популярним прикладом реалізації концепцій, описаних Google, є платформа Hadoop. Так прототипом файлової системи HDFS є GFS, ідеї покладені в основу архітектури HBase, взяті з BigTable, а технологія обчислень Hadoop MapReduce (без YARN) є реалізацією принципів, закладених в аналогічному фреймворку Google MapReduce.

Сама платформа Hadoop, починаючи з 2008 року, протягом декількох років буде набирати популярність і до 2010-2011 року по праву буде вважатися де-факто стандартом для роботи з BD. Сьогодні Hadoop вже є локомотивом в сфері BD і надає величезний вплив на цю область IT. Але колись такий ж величезний вплив на Hadoop надали сформовані в Google архітектурні підходи до побудови BD-платформ. Сама ж платформа Google вже той час росла, розвивалася, адаптувалася під все нові і нові вимоги, у пошукової системи з'являлися нові сервіси, в тому числі ті, чия природа відповідала скоріше інтерактивному режиму процесингу, ніж пакетного; розміри chunk (кластерів в GFS) підходили для ефективного, надійного зберігання не всіх типів даних, з'являлися проблеми, пов'язані з георозподіленністю і підтримкою розподілених транзакцій.

До 2009-2010 років як в самому Google, так і в академічному середовищі досить докладно вивчали переваги і обмеження комплексу підходів для побудови BD платформи, створеного інженерами Google в роки з 2003 по 2008. І сама платформа Google за період до 2009 року розвивалася і еволюціонувала.

В 2-ий етап прогресу Big Data платформи в Google - 2009-2013 - дослідниками цієї компанії з різним ступенем деталізації були описані такі

програмні системи: Colossus (GFS2) - самостійна розподілена файлова система, що є розвитком GFS [1]. Spanner - масштабування георозподілене сховище зпідтримкою версійності збережених даних, що є розвитком BigTable.

Dremel - добре-масштабируемая система обробки запитів в режимі близькому до режиму реального часу (near-real-time), призначена для аналізу пов'язаних read-only даних. Percolator - система для інкрементальною обробки великих даних, яка використовується для поновлення пошукових індексів Google. Caffeine - інфраструктура пошукових сервісів Google, яка використовує GFS2, next-generation (ітеративний) MapReduce і next-generation BigTable. Pregel це відмовостійка, масштабована і розподілена система обробки графів. Photon - відмовостійка та масштабована система обробки потокових даних [1].

Всі розробки нового покоління, поза сумнівом, перспективні, але не мають широкої користувальницької бази. що ставить під сумнів оптимальність такого вибору для справжнього проекту. У той же час, що викликала свого часу фурор модель розподілених обчислень MapReduce вже встигла застаріти - продуктивність, що забезпечується даною моделлю, для аналізу нинішніх об’ємів даних вже недостатньо.

В наші дні системна архітектура Apache Hadoop пішла далеко вперед від початково варіанту, сьогодні вона складається з бібліотек, dsl і утиліт, розподіленої файлової системи Hadoop Distributed File System (HDFS), середовища для управління ресурсами Hadoop YARN, технічної моделі Hadoop MapReduce і ще величезної кількості компонентів. У 2014 році стартувала міграція на нову версію Hadoop 2.0, що має цілий набір значних виправлень і доопрацювань, серед яких дві нові, більш сучасні моделі паралельних обчислень Apache Tez і Apache Spark.

Розробка Apache Tez і Apache Spark було ініційовано вимогами, що отримують широке поширення аналітично-орієнтованих додатків (машинне навчання, обробка текстів на природних мовах, розпізнавання людської мови, методи добування інформації з неструктурованих даних). Ітераційна схема

поведінки таких додатків суперечить організації єдиного потоку завдань в пакеті. Реалізація ітерацій засобами звичайного Hadoop MapReduce виявляється надзвичайно тривалою. Стосовно Apache Tez і Apache Spark можливо використовувати поняття «реальний час», однак мова не йде про суворе поняття реального часу в даному випадку, так як це поняття визначається готовністю очікувати результати із затримкою, що дорівнює часу доставки пакету даних.

Dremel являє собою інтерактивну систему, спеціально створену для обробки запитів, адресованих до згрупованим (nested) даних, доступним тільки для читання. Набір багаторівневих виконавчих дерев (multi-level execution trees) з колоночним позиціонуванням даних дозволяє обробляти мільярди табличних рядків в секунду.

Система масштабується до тисячі центральних процесорів, декількох петабайт даних і тисячі користувачів. Існує реалізація Dremel у відкритому коді під назвою OpenDremel, що увійшла в проект Apache Drill, що перевершує Dremel по функціональності.

Головна причина недостатності MapReduce при роботі з аналітичними додатками, в машинному навчанні лежить в застосуванні ациклічної моделі потоків даних (Directed Acyclic Graph, DAG), що є не придатною для реалізації будь-якої можливості використання циклів на протязі процесу від Map до Reduce, що суперечить основній ідеї подібних додатків. В даний момент, щоб адаптувати модель MapReduce для цілей аналізу, необхідно багаторазово повторювати основний цикл, а це призводить до помітного зниження продуктивності.

Apache Tez і Apache Spark дозволяють в десятки разів збільшити швидкість обробки: в першій це відбувається за рахунок удосконалення процедури виконання графової моделі, а в другій завдяки переходу на принципово іншу алгоритмічну основу, Tez перекладається з хінді як «швидкість», система є типовим зразком продукту, створеного в Індії, - в ньому реалізовано ефективніше рішення ніж MapReduce. Spark ж (від англійського «іскра») родом з Каліфорнійського університету Берклі, в ньому використана якісно нова

математика, і не виключено, що Spark - це саме та іскра, з якої може розгорітися полум'я нового покоління рішень по розподіленої обробки даних, і далеко не випадково, що після виходу Spark з тіні до цього рішення відразу було залучено велику увагу.

Apache Tez використовує DAG як набір завдань, інтерпретуючи обробку даних як граф потоків, де вершини - процеси, а ребра - переміщення даних між процесами. У вершинах розміщується аналітична логіка, а ребра відображають процес передачі даних і зв'язку між їх джерелами і споживачами. У Tez єдиний етап Reduce розділяється на окремі, пов'язані між собою стадії. Такий поділ стало можливим з появою конструкції YARN, що підтримує зв'язок з файлової системою HDFS і кластером.

Більш висока продуктивність алгоритмів Tez в порівнянні з Hadoop MapReduce забезпечується двома факторами. По перше, динамічним розпізнаванням графа, що відображає той факт, що зазвичай розподілені дані виявляються динамічними за своєю природою і багато з того, що з ними треба робити, визначається вже в процесі виконання. Динаміка знаходиться в протиріччі з пакетним режимом, де все слід визначити заздалегідь, а Tez включає в себе модулі модифікації вершин графа, що дозволяють знайти краще рішення. По-друге, Tez використовує оптимальне управління ресурсами. Ця функція реалізується їм спільно з YARN, де є елемент динаміки - потреба в ресурсах змінюється, і на різних фазах рішення задачі вони можуть бути різними. Система Tez не призначена для кінцевих користувачів і орієнтована на розробників додатків, підтримуючи сховище даних Apache Hive і аналітичну платформу з мовою високого рівня Apache Pig.

Одна з найсерйозніших проблем BD полягає в тому, що до парадигми тільки формуються, багато хто підходить зі старими мірками до методів BD, без розуміння цілісності картини. Серед нечисленних місць, де робляться спроби розібратися з неминучим в таких випадках клубком протиріч, є невелика дослідницька лабораторія AMPLab (Algorithms, Machines, People), в ній з 2008

року ведеться комплекс досліджень, який призвів до проекту Apache Spark, який являє собою працюючу в пам'яті програмну конструкцію, здатну скласти конкуренцію програмної моделі MapReduce. Автором даної новації став аспірант Берклі Матей Захаров, виходець з Румунії, тема дисертації якого - «Архітектура для швидкої і універсальної обробки даних на великих кластерах» (An Architecture for Fast and General Data Processing on Large Clusters). Одним з двох керівників Захаров є Скотт Шенкер - він відомий своїми роботами в області програмно-конфігуруючих мереж (SDN) і входив в число засновників компанії Nicira, купленої VMware в 2012 році.

Spark можна розглядати як подальший розвиток ідеї DAG з можливістю циклічної обробки потоків даних, розміщених в пам'яті. Відхід від пакетного режиму підтриманий новим, які не мають аналогів способом представлення даних, який отримав назву Resilient Distributed Dataset (RDD). Організація даних у вигляді RDD дозволяє розподіляти і перерозподіляти дані на різних стадіях паралельних обчислень, відкриваючи шлях ітераційним процесам, неможливим у випадку DAG. Ось чому RDD виявляється ефективним з точки зору як швидкості обробки, так і забезпечення високої надійності, необхідної при розподіленої роботі на кластерах.

Реалізація функціоналу RDD втілена в керуючу машину Spark RDD, поверх якої виконується потокова обробка, машина для обробки графів Spark Graph X, Spark SQL (аналог Hive SQL, але працює на два порядки швидше), а також аналітичні додатки, системи машинного навчання та ін. На відміну від Tez, рішення Spark розраховане не тільки на професійних розробників - воно може стати інструментом для фахівців за даними (data scientist).

Системи розподіленої обробки інформації стали тим інструментом, який підвищує ефективність вирішення подібних завдань. Під розподіленою обробкою інформації розуміється комплекс операцій з інформацією, що проводиться на незалежних, але пов'язаних між собою обчислювальних машинах, призначених для виконання спільних завдань. Системи розподіленого обробки даних легко

масштабуються та дозволяють скоротити час обробки даних за рахунок високого рівня паралельної обробки. Системи розподіленої обробки даних призначені для створення додатків, які здатні аналізувати великі обсяги інформації за досить невеликий проміжок часу.

1.3 Представлення даних

Дані - відомості, які характеризують систему, явище, процес або об'єкт, представлені в певній формі та призначені для подальшого використання. За ступенем структурованості виділяють наступні форми представлення даних:

– неструктуровані;
– структуровані;
– слабоструктуровані.

До неструктурованих відносяться дані, довільні за формою, включають тексти і графіку, мультимедіа (відео, мова, аудіо). Ця форма представлення даних широко використовується, наприклад, в Інтернеті, а самі дані представляються користувачеві у вигляді відгуку пошукових систем. Структуровані дані відображають окремі факти предметної області. Структурованими називаються дані, певним чином впорядковані і організовані з метою забезпечення можливості застосування до них деяких дій (наприклад, візуального або машинного аналізу). Це основна форма подання відомостей в базах даних. Організація того чи іншого виду зберігання даних (структурованих або неструктурованих) пов'язана із забезпеченням доступу до них. Під доступом розуміється можливість виділення елемента даних (або безлічі елементів) серед інших елементів з яких-небудь ознаками з метою виконання деяких дій над елементом. однією з найпоширеніших моделей зберігання структурованих даних

є таблиця. У ній всі дані упорядковуються в двовимірну структуру, що складається з стовпців і рядків. В осередках такої таблиці містяться елементи даних: символи, числа, логічні значення.

Неструктуровані дані непридатні для обробки безпосередньо методами аналізу даних, тому такі дані піддаються спеціальним прийомам структуризації, причому сам характер даних в процесі структуризації може істотно змінитися. Наприклад, в аналізі текстів (Text Mining) при структуруванні з вихідного тексту може бути сформована таблиця з частотами зустрічальності слів, і вже такий набір даних буде оброблятися методами, придатними для структурованих даних. Слабоструктуровані дані - це дані, для яких визначені деякі правила і формати, але в найзагальнішому вигляді. Наприклад, рядок з адресою, рядок в прайс-листі, ПІБ і т.п. На відміну від неструктурованих, такі дані з меншими зусиллями перетворюються до структурованої формі, однак без процедури перетворення вони теж непридатні для аналізу.

У наші дні, тема BD нерозривно слідує рука об руку з поняттям слабоструктурованих даних. Все нові й нові показники, вимірювання, результати опитувань з'являються щодня з високою швидкістю, приведення хаотичного набору значень до повністю структурованого є непростою задачею. Значно менше зусиль доведеться докласти, коли дані для подальшої обробки задовольнять критеріям слабоструктурованості, в цьому випадку, робота з ними в автоматизованому режимі не на багато складніша, ніж з повністю структурованими.

Крім технічних достоїнств важливі і соціально-комерційні. Тому варто звернути увагу і на те, що проект Apache Spark привертає до себе величезну увагу, про нього написано велику кількість практичних статей, він став частиною Hadoop 2.0. Плюс він швидко обріс додатковими фреймворками, такими, як Spark Streaming, SparkML, Spark SQL, GraphX, а крім цих «офіційних» фреймворків з'явилося безліч проектів - різні коннектори, алгоритми, бібліотеки і так далі. Що говорить про його заслужено оцінений внесок в врозвиток BD.

2 ДОСЛІДЖЕННЯ МЕТОДІВ РОЗПОДІЛЕННОЇ ОБРОБКИ ДАНИХ В BD

2.1 Моделювання примітивів розподіленої системи обробки даних в BD

На високому рівні кожна програма Spark складається з програми драйвера, яка запускає основну функцію користувача і виконує різні паралельні операції на кластері. Основною абстракцією, яку надає Spark, є примітив RDD, який є сукупністю елементів, розділених по вузлах кластера, з якими можна працювати паралельно. RDD створюються, починаючи з файлу у файловій системі Hadoop (або будь-якої іншої файлової системи, що підтримується з Hadoop), або існуючої колекції Scala у програмі драйвера. Користувачі також можуть попросити Spark зберігати RDD в пам'яті, дозволяючи ефективно використовувати паралельні операції. Нарешті, RDDs автоматично відновлюються після збою вузлів.

Інша абстракція в Spark - спільні змінні, які можна використовувати в паралельних операціях. За умовчанням, коли Spark запускає функцію паралельно як набір завдань на різних вузлах, вона надсилає кожен номер кожної змінної, що використовується у функції, до кожного завдання. Іноді змінна повинна бути розділена між завданнями або між завданнями і програмою драйвера. Spark підтримує два типи спільних змінних: широкомовні змінні, які можуть використовуватися для кешування значення в пам'яті на всіх вузлах і акумулятори, які є змінними, які тільки "додаються", як лічильники і суми.

Процес побудови графа взаємозв'язків RDD можна розділити на три основних етапу:

- виділення необхідної інформації для побудови графа взаємозв'язків;
- написання методів вилучення даної інформації;
- візуалізація графа на основі зібраних даних.

Система Spark RDD представляється у вигляді деякого класу, який забезпечує доступ до інформації, що характеризує розподілений набір даних. Ця інформація складається з п'яти основних частин: блоків даних, які є одиницею інформації, що зберігається (елемент RDD), залежностей від інших RDD, функція отримання даної RDD з батьківських, метадані про розбиття на розділи і розташуванні. При побудові графа RDD в першу чергу треба звернути увагу на ту частину RDD, яка описує залежності наборів даних між собою. В системі Spark залежності між RDD можна розділити на два типи: прямолінійні (narrow) залежності, при якій з кожного елемента RDD батька генерується рівно один елемент RDD нащадка, і широкі (wide) залежності, при яких один елемент RDD може виступати в якості батька для декількох елементів RDD нащадка.

За типом залежності від батьківської RDD ми можемо розділити всі операції перетворення RDD. Операції з прямолінійною залежністю дозволяють конвеєрно обчислювати нові RDD на одному вузлі кластера. У цьому випадку кожен кластер для отримання елементів нової RDD застосовують деяку функцію до вже обчисленим на цьому ж кластері елементам RDD батька, і необхідність передачі даних між вузлами відсутня. Такі операції називають конвеєрними перетвореннями RDD. Прикладом такого перетворення може служити операція filter, результатом якої є вибірка елементів батьківського RDD, що задовольняють деякому предикату.

RDD підтримують два типи операцій: перетворення, які створюють новий набір даних з існуючого, і дії, які повертають значення програми драйвера після запуску обчислення набору даних. Наприклад, map є перетворенням, яке передає кожен елемент набору даних через функцію і повертає новий RDD, що представляє результати. З іншого боку, reduce - це дія, яка агрегує всі елементи RDD, використовуючи деяку функцію, і повертає кінцевий результат до програми драйвера (хоча існує також паралельна reductionByKey, яка повертає розподілений набір даних).

Всі перетворення в Spark є лінивими, в тому, що вони не обчислюють свої результати одразу. Натомість, вони просто пам'ятають перетворення, застосовані до деякого базового набору даних (наприклад, файл). Перетворення обчислюються лише тоді, коли дія вимагає повернення результату до програми драйвера. Ця конструкція дозволяє Spark працювати більш ефективно. Наприклад, ми можемо зрозуміти, що набір даних, створений через map, буде використовуватися у зменшенні і повертати тільки результат зменшення на драйвер, а не на більший нанесений набір даних.

За замовчуванням кожне перетворене RDD може бути перераховано кожен раз при виконанні дії на ньому. Тим не менш, можливо також зберігати RDD в пам'яті, використовуючи метод persist (або кеш), в цьому випадку Spark зберігатиме елементи навколо кластера для набагато швидшого доступу наступного разу, коли йде запит. Існує також підтримка постійних RDD на диску або реплікації на декількох вузлах.

Одна з найважчих речей у Spark - розуміння масштабу та життєвого циклу змінних та методів при виконанні коду через кластер. Операції RDD, які змінюють змінні за межами області їх застосування, можуть бути джерелом плутанини.

Операції з широкою залежністю для генерації нового блоку RDD вимагають наявності декількох або всіх елементів батьківського RDD, і, отже, не можуть бути виконані локально на одному вузлі кластера. Прикладом цього перетворення є операція groupByKey, результат якої буде являти собою групування елементів батьківської RDD по деякому ключу. Поділ на типи операцій над RDD лежить в основі поділу деякої задачі (перетворень, необхідних для вчинення дії), на окремі стадії. Кожна стадія являє собою послідовність трансформацій RDD, що мають прямолінійну залежність. На кордонах між стадіями відбуваються так звані операції перетасовки, при яких між батьківським і дочірнім RDD встановлюються широкі залежності.

Таким чином можна виділити залежність батько-нащадок між стадіями: нова стадія не може початися, поки не будуть обчислені всі елементи RDD батьківських стадій, потрібних для перетворення з широкою залежністю. стадії розбиваються на маленькі підзадачі, які віддаються в вузли кластера і в них виконуються. Кожна така підзадача (або Task) являє собою процес застосування до одного розділу початкової для даної стадії RDD всієї послідовності трансформацій, що складають цю стадію, отримуючи в результаті обчислення один з розділів заключній RDD даній стадії.

Одним з основних компонентів системи Spark є планувальник завдань. Планувальник підтримує описане вище уявлення RDD і відповідає за розподіл ресурсів між обчисленнями. При виконанні будь-якого дії в користувальницькії програмі планувальник досліджує граф взаємозв'язків RDD і будує послідовність стадій, які необхідно виконати для обчислення потрібної дії. Планувальник має інформацію про взаємозв'язки RDD і стадії виконання завдання, тому для отримання даної інформації необхідно ініціалізувати звернення до даного модулю.

Для побудови графа необхідно знати назви всіх стадій завдання і черговість з виконання. Для визначення черговості потрібно для кожної стадії зберігати посилання на їх батьківські стадії. Так само для кожної стадії необхідно мати список всіх RDD, які теж будуть характеризуватися назвою і посиланням на батьків. Візуалізацію графа можна розділити на наступні етапи:

– зчитування опису графа;
– побудова і візуалізація графа для RDD;
– поділ графа RDD на блоки, які будуть відповідати стадіями виконання.

Крім безпосередньо візуалізації графа залежностей RDD передбачається наявність відображення динаміки виконання профільованого завдання. Оскільки, на відміну від активних блоків RDD, активна стадія для всіх вузлів кластера

визначається однозначно, на графі залежностей RDD відбуватиметься відображення активної на даний момент стадії виконання завдання.

API RDD Spark також надає асинхронні версії деяких дій, наприклад foreachAsync для foreach, які негайно повертають FutureAction замість блокування після завершення дії. Це можна використовувати для керування або очікування асинхронного виконання дії.

Деякі операції в Spark викликають подію, відому як shuffle. Це механізм Spark для повторного розповсюдження даних, так що він розділений по групах. Це зазвичай передбачає копіювання даних між виконавцями та машинами, що робить процес shuffle складним і дорогим.

Щоб зрозуміти, що відбувається під час shuffle операції, можна розглянути приклад операції reduceByKey. Операція reduceByKey генерує новий RDD, де всі значення для одного ключа об'єднуються в кортеж - ключ і результат виконання функції зменшення відносно всіх значень, пов'язаних з цим ключем. Завдання полягає в тому, що не всі значення для одного ключа обов'язково знаходяться на одному розділі, або навіть на одній машині, але вони повинні бути розташовані спільно для обчислення результату.

У Spark дані, як правило, не розподілені між розділами, щоб знаходитися в необхідному місці для конкретної операції. Під час обчислень, однин Task буде працювати на одному розділі - таким чином, щоб організувати всі дані для одного reduceByKey. Він повинен читати з усіх розділів, щоб знайти всі значення для всіх ключів, а потім об'єднати значення між розділами, щоб обчислити кінцевий результат для кожної операції.

Shuffle - це дорога операція, оскільки вона включає в себе I/O диска, серіалізацію даних і мережевий I/O. Щоб організувати дані для shuffle, Spark генерує набір тасків для організації даних, а також набір тасків зменшення, щоб агрегувати його. Ця номенклатура походить від MapReduce і безпосередньо не стосується Spark.

Внутрішньо, результати окремих тасків зберігаються в пам'яті, поки вони не можуть відповідати. Потім вони будуть видалені на основі цільового розділу і запишуться в один файл. На стороні reduce читають відповідні сортовані блоки.

Деякі операції у випадковому порядку можуть споживати значну кількість пам'яті купи, оскільки вони використовують структури даних у пам'яті для організації записів до або після їх передачі. Зокрема, reduceByKey і aggregateByKey створюють ці структури на стороні map операції. Коли дані не вписуються в пам'ять, Spark може рознести ці таблиці на диск, зазнавши додаткових накладних витрат на диск та I/O і збільшення збору сміття.

Shuffle також генерує велику кількість проміжних файлів на диску. Від Spark 1.3 ці файли зберігаються доти, доки відповідні RDD більше не будуть використовуватися. Це робиться для того, щоб файли перетасовування не потребували повторного створення, якщо повторні обчислення будуть необхідні. Збір сміття може відбутися лише після тривалого періоду часу, якщо програма зберігає посилання на ці RDD або якщо GC не починає часто виникати. Це означає, що довгострокові задачі Spark можуть споживати великий обсяг дискового простору. Каталог тимчасового зберігання задається параметром конфігурації spark.local.dir під час налаштування контексту Spark.

Однією з найважливіших можливостей Spark є збереження (або кешування) набору даних у пам'яті через операції. Під час зберігання RDD, кожен вузол зберігає будь-які його розділи, які він обчислює в пам'яті, і повторно використовує їх в інших діях на цьому наборі даних (або наборів даних, отриманих з неї). Це дає змогу майбутнім діям працювати набагато швидше (часто більше 10x). Кешування є ключовим інструментом для ітераційних алгоритмів і швидкого інтерактивного використання.

Ви можете позначити RDD, який буде збережено, використовуючи методи persist або cache. Вперше він обчислюється в дії, він буде зберігатися в пам'яті на вузлах. Кеш Spark є відмовостійким - якщо будь-який розділ RDD втрачено, він

автоматично перераховується за допомогою перетворень, які його спочатку створили.

Крім того, кожен постійний RDD може зберігатися з використанням іншого рівня зберігання, що дозволяє, наприклад, зберігати набір даних на диску, зберігати його в пам'яті, але як серіалізовані об'єкти Java (для економії місця), реплікувати його по вузлах. Ці рівні встановлюються шляхом передачі об'єкта StorageLevel. Метод cache є скороченням для використання рівня зберігання за замовчуванням, який є StorageLevel.MEMORY_ONLY (зберігати десеріалізовані об'єкти в пам'яті).

2.2 Обробка даних в Spark в режимі кластера

Додатки Spark виконуються як незалежні набори процесів на кластері, координовані об'єктом SparkContext у вашій основній програмі (так званий драйвер). Кожний додаток отримує свої власні процеси виконання, які залишаються на весь термін дії програми і виконують завдання в декількох потоках. Це має вигоду ізолювати програми один від одного, як на стороні планування (кожен драйвер розробляє власні завдання), так і на стороні виконавця (завдання з різних додатків виконуються в різних JVM). Однак, це також означає, що дані не можуть бути спільно використовуються між різними програмами Spark (примірниками SparkContext) без запису до зовнішньої системи зберігання.

Spark є агностиком для основного менеджера кластера. Поки він може запустити процеси виконавця, і вони взаємодіють один з одним, його можливо відносно легко запустити навіть на менеджері кластерів, який також підтримує інші програми (наприклад, Mesos/YARN). Програма драйвера повинна

прослуховувати та приймати вхідні з'єднання від своїх виконавців протягом усього терміну служби (наприклад, Spark.driver.port). Таким чином, програма драйвера повинна бути адресована мережею з робочих вузлів. Оскільки драйвер розкладає завдання на кластері, він повинен бути запущений близько до робочих вузлів, бажано в тій же локальній мережі. Для відправки запиту до кластера віддалено, краще відкрити RPC до драйвера і надати операторам поблизу дозволи, ніж запустити драйвер далеко від робочих вузлів, що зображено на рисунку 2.1.

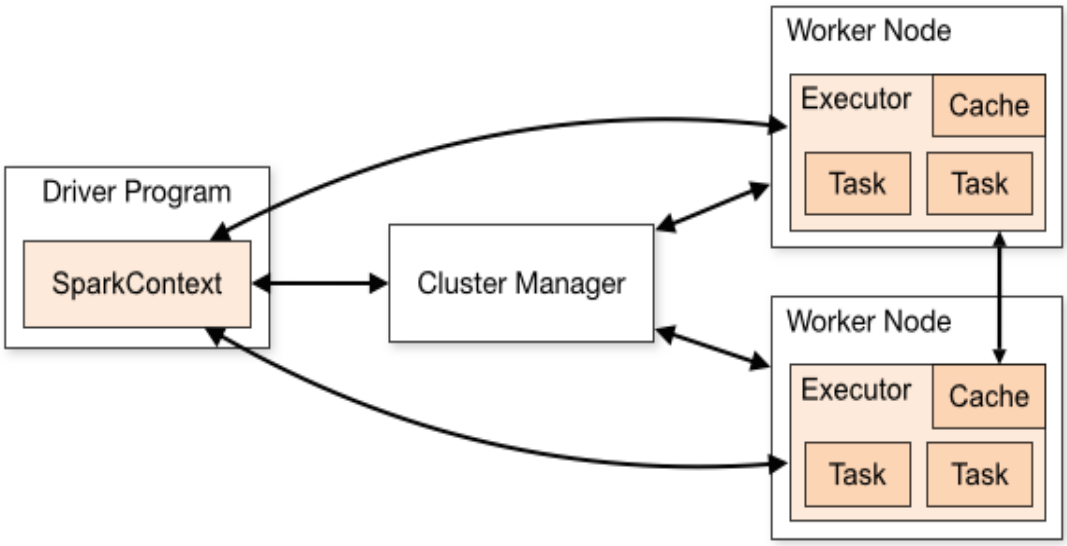

Рисунок 2.1 – Кластер менеджер Spark

Система наразі підтримує декілька менеджерів кластерів:

– автономний - простий менеджер кластера, що входить до складу Spark, що полегшує налаштування кластера;
– Apache Mesos - загальний менеджер кластерів, який також може запускати Hadoop MapReduce і сервісні програми;
– Hadoop YARN - менеджер ресурсів у Hadoop 2;
– Kubernetes - система з відкритим вихідним кодом для автоматизації розгортання, масштабування та керування контейнерними додатками.

Тепер, коли драйвер створює завдання і починає видавати завдання для планування, Mesos визначає, які машини обробляють ті завдання. Оскільки він враховує інші рамки при плануванні цих численних короткочасних завдань, кілька фреймворків можуть співіснувати на одному кластері, не вдаючись до статичного розподілу ресурсів.

Якщо увімкнено автентифікацію Mesos Framework, необхідно надати основну та секретну інформацію, за допомогою якої можна аутентифікувати Spark у Mesos. Кожна робота Spark буде зареєстрована в Mesos як окрема структура [2].

Залежно від середовища розгортання можна створити єдиний набір облікових даних, які спільно використовуються для всіх користувачів або створюють облікові дані для кожного користувача. Створення та керування обліковими даними рамки слід здійснювати за документацією автентифікації Mesos.

Коли Mesos вперше виконує завдання на підлеглому процесі, цей підлеглий процес повинен мати бінарний пакет Spark для запуску сервера виконавців Spark Mesos. Пакет Spark може бути розміщений на будь-якому доступному для Hadoop URI, включаючи HTTP через http://, Amazon Simple Storage Service через s3n://, або HDFS через hdfs://.

У fine grained режимі кожен виконавець Spark працює як одна задача Mesos. Виконавці Spark мають розміри відповідно до таких змінних конфігурації:

– пам'ять виконавця - spark.executor.memory;
– ядра виконавця - spark.executor.cores;
– кількість виконавців - spark.cores.max/spark.executor.cores.

Виконавці піднімаються під час запуску програми, доки не буде досягнуто spark.cores.max. Якщо не встановлено spark.cores.max, програма Spark буде споживати всі ресурси, запропоновані Mesos, тому слід встановити цю змінну в

будь-якому кластері з декількома процесами, включаючи кластер де є декілька одночасних додатків Spark.

Планувальник запустить виконавців розподіляючи навантаження за допомогою round robin, але не існує гарантій розповсюдження, оскільки Mesos не надає таких гарантій у потоці.

У цьому режимі виконавці Spark будуть використовувати виділений порт, якщо такий надано користувачем. Зокрема, якщо користувач визначає spark.blockManager.port в конфігурації Spark, планувальник mesos перевірить наявні пропозиції для дійсного діапазону портів, що містить номери портів. Якщо такого діапазону немає, він не запускає жодних завдань. Якщо користувачеві не накладається обмеження на номери портів, порти використовуються як звичайно. Ця реалізація виконання порту передбачає одне завдання на хост, якщо користувач визначає порт. У майбутній мережі слід підтримувати ізоляцію [2].

Перевагою fine grained режиму є набагато нижчий накладний запуск. Щоб налаштувати своє завдання на динамічне пристосування до вимог до ресурсів, слід використовувати Dynamic Allocation параметри.

Mesos підтримує динамічне виділення лише з fine grained режимом, який може змінювати кількість виконавців на основі статистики програми.

Службою зовнішнього перемикання, якою користуються Mesos, є служба Mesos Shuffle. Вона надає функцію очищення даних у випадковому порядку поверх служби Shuffle, оскільки Mesos ще не підтримує повідомлення про припинення іншої структури. Щоб його запустити, запустити скипт start-mesos.sh на всіх підлеглих вузлах, у яких spark.shuffle.service.enabled встановлено як true. Ці налаштування використовуються для запису на HDFS і підключення до ресурсу ResourceManager. Конфігурація, що міститься в цьому каталозі, буде поширюватися на кластер YARN, так що всі контейнери, що використовуються програмою, використовують ту ж конфігурацію. Якщо конфігурація посилається на властивості системи Java або змінні середовища, якими не керує YARN, вони

також повинні бути встановлені в конфігурації програми Spark (драйвер, виконавці та AM при роботі в режимі клієнта).

Є два режими розгортання, які можна використовувати для запуску програм Spark на YARN. У режимі кластера драйвер Spark запускається всередині процесу, який керується YARN на кластері, і клієнт буде запущений після запуску програми. У режимі клієнта драйвер запускається в процесі клієнта, а майстер програми використовується тільки для запиту ресурсів з YARN.

На відміну від інших менеджерів кластерів, що підтримуються Spark, в якому вказано адресу master в параметрі --master, в режимі YARN адреса ResourceManager вибирається з конфігурації Hadoop.

Kubernetes вимагає від користувачів надавати зображення, які можуть бути розгорнуті в контейнерах. Зображення побудовані для запуску в середовищі виконання, яке підтримує Kubernetes. Docker - це середовище виконання середовища, яке часто використовується з Kubernetes. Spark (починаючи з версії 2.3) постачається з файлом Docker, який можна використовувати з цією метою або налаштовувати відповідно до потреб окремої програми. Його можна знайти в каталозі kubernetes/dockerfiles/.

Spark також поставляється з сценарієм bin/docker-image-tool.sh, який можна використовувати для побудови та публікації зображень Docker для використання з сервером Kubernetes.

Починаючи з Spark 2.4.0, можна запускати програми Spark на Kubernetes в режимі клієнта. Коли ваша програма працює в режимі клієнта, драйвер може запускатися всередині пакета або на фізичному хості.

Виконавці Spark повинні мати змогу підключатися до драйвера Spark за іменем хоста та портом, який маршрутизується з виконавців Spark. Конкретна конфігурація мережі, яка буде потрібна для роботи Spark у режимі клієнта, залежить від налаштувань. Під час запуску драйвера всередині панелі Kubernetes, слід використовувати фоновий сервіс, який дозволить вашому драйверу маршрутизувати дані від виконавців. Під час розгортання служби без фонового

процесу слід переконатися, що селектор міток служби буде відповідати лише панелі драйверів та інших пакунків. Рекомендується присвоїти драйверу достатньо унікальну мітку і використовувати цю мітку в селекторі етикеток сервісу.

Якщо програма повинна взаємодіяти з іншими файловими системами Hadoop, їх URI потрібно чітко надати Spark під час запуску. Це робиться шляхом перерахування їх у властивості spark.yarn.access.hadoopFileSystems.

Планувальник Spark розгортає пакунки виконавця за допомогою OwnerReference, що, у свою чергу, гарантує, що після видалення панелі драйверів з кластера всі біни виконавців програми також будуть видалені. Драйвер буде шукати папку з даним ім'ям у просторі імен, визначеному spark.kubernetes.namespace, і OwnerReference, що вказує на цю панель, буде додано до списку OwnerReferences для кожного виконавця. Будьте обережні, щоб уникнути налаштування OwnerReference на панель, яка насправді не є такою панеллю драйверів, або ж виконавці можуть бути припинені передчасно, коли неправильна панель видалена.

Якщо spark.kubernetes.driver.pod.name не встановлено, коли ваша програма фактично запущена в паку, що пакети виконавців не можуть бути належним чином видалені з кластера, коли програма виходить. Планувальник Spark намагається видалити ці пакети, але якщо мережевий запит на сервер API не відбувся з будь-якої причини, ці біни залишаться в кластері. Процеси виконавця повинні вийти, коли вони не можуть дістатися до драйвера, тому пакети виконавців не повинні споживати обчислювальні ресурси (процесор і пам'ять) в кластері після завершення роботи програми.

Якщо залежність вашої програми розміщена у віддалених місцях, таких як HDFS або HTTP-сервери, на них можуть посилатися відповідні віддалені URI. Крім того, прикладні залежності можуть бути попередньо змонтовані в спеціально створені зображення Docker. Ці залежності можуть бути додані до classpath шляхом посилання на них з локальним: URI або встановлення змінної

середовища SPARK_EXTRA_CLASSPATH у Docker-файлах. Локальна схема також потрібна при зверненні до залежностей у спеціально створених зображеннях Docker у форматі spark-submit. Використання залежностей програми від локальної файлової системи клієнта подання наразі ще не підтримується.

Починаючи з Spark 2.4.0, користувачі можуть монтувати наступні типи томів Kubernetes до драйверів і виконавців:

– hostPath: встановлює файл або директорію з файлової системи вузла хоста в модуль;
– emptyDir: спочатку порожній том, створений, коли домену призначено вузол;
– persistentVolumeClaim використовується для монтування у паку.

Зокрема, VolumeType може бути одним з таких значень: hostPath, emptyDir і persistentVolumeClaim. VolumeName потрібно використати для тома в полі томів у специфікації пакунків.

Якщо сервер API Kubernetes відхиляє запит або з'єднання відхилено з іншої причини, логіка подання повинна вказувати на помилку. Проте, якщо під час запуску програми є помилки, найчастіше найкращим способом пошуку проблеми може бути Kubernetes CLI.

Kubernetes дозволяє використовувати ResourceQuota для встановлення обмежень на ресурси, кількість об'єктів і т.д. на окремих просторах імен. Простори імен і ResourceQuota можуть використовуватися в поєднанні з адміністратором для керування спільним доступом і розподілом ресурсів у кластері Kubernetes, де працюють програми Spark.

У кластерах Kubernetes з підтримкою RBAC користувачі можуть налаштовувати ролі Kubernetes RBAC та облікові записи служб, що використовуються різними компонентами Spark на Kubernetes, для доступу до сервера API Kubernetes, що зображено на рисунку 2.2.

Pod драйверів Spark використовує обліковий запис служби Kubernetes для доступу до сервера API Kubernetes для створення та перегляду пакунків виконавців.

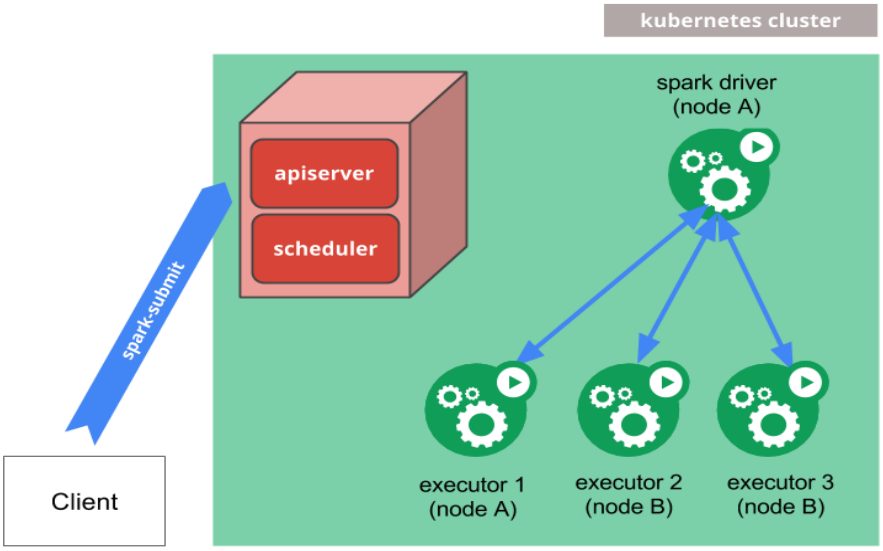

Рисунок 2.2 – Кластер Spark на Kubernetes

Обліковий запис сервісу, який використовується драйвер процесом, повинен мати відповідний дозвіл, щоб він міг виконувати свою роботу. Зокрема, обліковий запис сервісу повинен бути наданий Role або ClusterRole, що дозволяє драйверам-пачкам створювати пакети та служби. За замовчуванням панель драйвера автоматично призначається обліковому запису служби за промовчанням в просторі імен, визначеному spark.kubernetes.namespace, якщо не створено облікового запису служби під час створення пакунка [3].

Залежно від версії та налаштування Kubernetes, цей обліковий запис за промовчанням може мати або не мати роль, що дозволяє драйверам-бінам створювати пакети та служби під правилами RBAC за умовчанням Kubernetes. Інколи може знадобитися вказати обліковий запис спеціальної служби, який має належну роль. Spark на Kubernetes підтримує визначення користувальницького

облікового запису служби, який буде використовуватися роботою драйвера через властивість конфігурації.

2.3 Обробка даних з Spark SQL

Spark SQL є модулем Spark для структурованої обробки даних. На відміну від основного API RDD Spark, інтерфейси, надані Spark SQL, надають Spark більше інформації про структуру даних і обчислення, що виконуються. Внутрішньо Spark SQL використовує цю додаткову інформацію для виконання додаткових оптимізацій. Існує кілька способів взаємодії з Spark SQL, включаючи SQL і API. При обчисленні результату використовується той самий механізм виконання, незалежно від того, який API використовується для вираження обчислень. Це об'єднання означає, що розробники можуть легко перемикатися між різними API, на основі яких забезпечується найбільш природний спосіб висловити дану трансформацію.

Одним із застосувань Spark SQL є виконання SQL-запитів. Spark SQL також може використовуватися для зчитування даних з існуючої інсталяції Hive. Під час запуску SQL з іншої мови програмування результати будуть повернуті у вигляді DataSet/DataFrame. Також можете взаємодіяти з інтерфейсом SQL за допомогою командного рядка або над JDBC/ODBC.

Spark SQL підтримує роботу з різними джерелами даних через інтерфейс DataFrame. DataFrame може працювати з використанням реляційних перетворень і може також використовуватися для створення тимчасового перегляду. Також можна вручну вказати джерело даних, яке буде використовуватися разом з будь-

якими додатковими параметрами, які можна передати передати джерелу даних. Джерела даних визначаються їх повним іменем до пакету, але для вбудованих джерел можна також використовувати їх короткі назви (json, паркет, jdbc, orc, libsvm, csv, text). DataFrames, завантажені з будь-якого типу джерела даних, можуть бути перетворені в інші типи за допомогою цього синтаксису.

Розбиття таблиць є загальним підходом оптимізації, що використовується в таких системах, як Hive. У секціонованій таблиці дані зазвичай зберігаються в різних каталогах, при цьому значення стовпців розбиття кодуються в шляху до кожної директорії розділів. Всі вбудовані джерела файлів (включаючи Text/CSV/ JSON/ORC/Parquet) здатні автоматично виявляти та виводити інформацію про розділення.

Подібно Protobuf, Avro і Thrift, Parquet також підтримує еволюцію схеми. в В міру необхідності починаючи з простої схеми і поступово додаючи додаткові стовпці до схеми можливо виконувати еволюцію схеми. Таким чином, можливо мати декілька файлів Parquet з різними, але взаємно сумісними схемами. Джерело даних Parquet може автоматично визначати цей випадок і об'єднувати схеми всіх цих файлів [4].

Оскільки об'єднання схем є відносно дорогою операцією і в більшості випадків не є необхідністю, тому вона була вимкнута за замовчуванням, починаючи з 1.5.0

Spark SQL також містить джерело даних, яке може зчитувати дані з інших баз даних за допомогою JDBC. Це пояснюється тим, що результати повертаються як DataFrame, і їх можна легко обробити в Spark SQL, що дозволяє обробляти дані в реальному часі з різних джерел. Оброблені дані можуть бути витіснені до файлових систем, баз даних і панелей BI. Джерела даних такі як JDBC або черги повідомлення також легше використовувати з Java або Python що забезпечує

масштабовану, високопродуктивну, стійку до відмов, оскільки не вимагається від користувача надання ClassTag, що зображено на рисунку 2.3

Рисунок 2.3 – Spark Streaming та джерела даних

Слід зауважити, що це відрізняється від сервера JDBC Spark SQL, який дозволяє іншим програмам запускати запити за допомогою Spark SQL.

2.4 Обробка потоків даних з Spark Streaming

Apache Spark Streaming є масштабованою відмовостійкою потоковою системою обробки даних, яка спочатку підтримує пакетне та потокове завантаження. Spark Streaming є розширенням базового API Spark, що дозволяє обробляти дані в реальному часі з різних джерел, включаючи (але не обмежуючись) Kafka, Flume і Kinesis. Його ключовою абстракцією є дискретизований потік або, коротко, DStream, який представляє потік даних, розділений на невеликі партії. DStreams побудовано на RDD, основній абстракції

даних Spark. Це дозволяє Spark Streaming плавно інтегруватися з будь-якими іншими компонентами Spark, такими як MLlib і Spark SQL [5].

Spark Streaming відрізняється від інших систем, які мають механізм обробки, призначений тільки для потокового передавання та має подібні пакетні і потокові API, що зображено на рисунку 2.4.

Spark Streaming є розширенням ядра Spark API, що забезпечує масштабовану, високопродуктивну, стійку до відмов потокової обробки потоків даних. Дані можуть надходити з багатьох джерел, таких як Kafka, Flume, Kinesis або TCP sockets, і можуть бути оброблені за допомогою складних алгоритмів, виражених функціями високого рівня.

Рисунок 2.4 – Spark Streaming та мікробатчі

Нарешті, оброблені дані можна виштовхнути до файлових систем, баз даних і панелей BI. Також можна застосувати алгоритми машинного навчання і обробки графіків Spark на потоках даних.

Ця уніфікація різнорідних можливостей обробки даних є основною причиною швидкого зростання популярності Spark Streaming. Це дозволяє легко розробникам використовувати єдиний фреймворк, щоб задовольнити всі їхні потреби обробки.

Дискретизований потік або DStream є основною абстракцією, що надається Spark Streaming. Вона являє собою безперервний потік даних, або вхідний потік

даних, отриманий від джерела, або оброблений потік даних, генерований шляхом перетворення вхідного потоку. Внутрішньо DStream представлений безперервним рядом RDD, що є абстракцією Spark незмінного, розподіленого набору даних [6].

Внутрішньо він працює наступним чином. Spark Streaming отримує потоки живих вхідних даних і розділяє дані на частини, які потім обробляються двигуном Spark для генерації остаточного потоку результатів у партіях. Spark Streaming забезпечує абстракцію високого рівня, що називається дискретним потоком або DStream, який представляє безперервний потік даних. DStreams можуть бути створені або з потоків вхідних даних з таких джерел, як Kafka, Flume і Kinesis, або шляхом застосування операцій високого рівня на інших DStreams. Внутрішньо DStream представлено у вигляді послідовності RDD.

3 ПОРІВНЯННЯ АЛГОРИТТМИВ ОБРОБКИ ПОТОКИВ ДАНИХ

3.1 Обробка потоків даних з Spark Streaming

З самого початку Apache Spark забезпечив уніфікований движок, який спочатку підтримує як пакетні, так і потокові навантаження. Це відрізняється від інших систем, які або мають механізм обробки, призначений тільки для потокової передачі, або мають подібні пакетні і потокові API, але компілюються всередині різних двигунів. Єдиний механізм виконання Spark та уніфікована модель програмування для пакетного та потокового доступу призводять до деяких унікальних переваг перед іншими традиційними потоковими системами. Зокрема, чотири головні аспекти:

- швидке відновлення від збоїв і відсталих;
- краще балансування навантаження та використання ресурсів;
- об'єднання потокових даних зі статичними наборами даних та інтерактивними запитами;
- рідна інтеграція з передовими бібліотеками обробки (SQL, машинне навчання, обробка графіків).

Сучасні розподілені алгоритми обробку потоку даних виконуються наступним чином:

- отримання потоку даних з джерел (наприклад, живі журнали, дані системної телеметрії, дані пристрою IoT тощо) в деякі системи збору даних, такі як Apache Kafka, Amazon Kinesis і т.д.
- обробка даних паралельно на кластері.
- вивід результатів до таких систем, як HBase, Cassandra, Kafka та ін.

Для обробки даних більшість традиційних систем потокової обробки розроблені з безперервною моделлю оператора, яка працює наступним чином:

– існує набір робочих вузлів, кожен з яких виконує один або більше безперервних операторів;
– кожен безперервний оператор обробляє потокову інформацію по одному запису за раз і пересилає записи іншим операторам в конвеєрі;
– існують source оператори для прийому даних від систем прийому та sink оператори, які виводять дані на нижні системи, що зображено на рисунку 3.1.

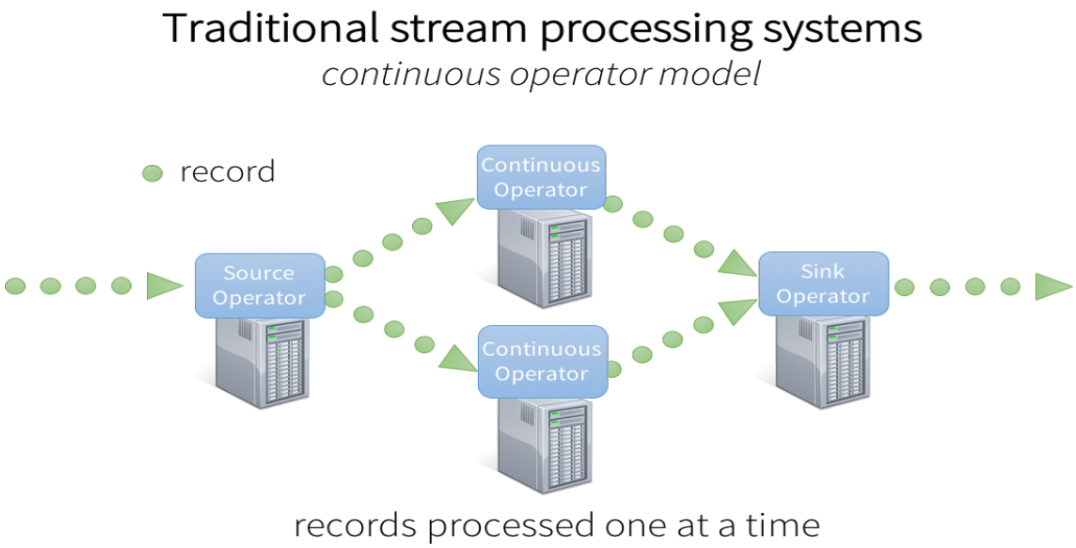

Рисунок 3.1 – Архітектура традиційних систем потокової обробки

Однак із сьогоднішньою тенденцією до більш масштабних і більш складних аналітичних даних у реальному часі, ця традиційна архітектура також зіткнулася з деякими викликами. На жаль, статичне виділення безперервних операторів до робочих вузлів ускладнює традиційні системи та швидке відновлення від несправностей і відсталих повідомлень. Найчастіше це

відбувається у великих кластерах та динамічно змінюються навантаженнях. Система повинна мати можливість динамічно адаптувати розподіл ресурсів на основі робочого навантаження [7].

Уніфікація потокових, пакетних та інтерактивних навантажень - у багатьох випадках використання також є привабливим для інтерактивного запиту потокових даних (зрештою, потокова система тримає всі дані в пам'яті), або комбінувати її зі статичними наборами даних (наприклад, попередньо обчислені) моделей). Для цього потрібний єдиний механізм, який може поєднувати пакетні, потокові та інтерактивні запити, що зображено на рисунку 3.2.

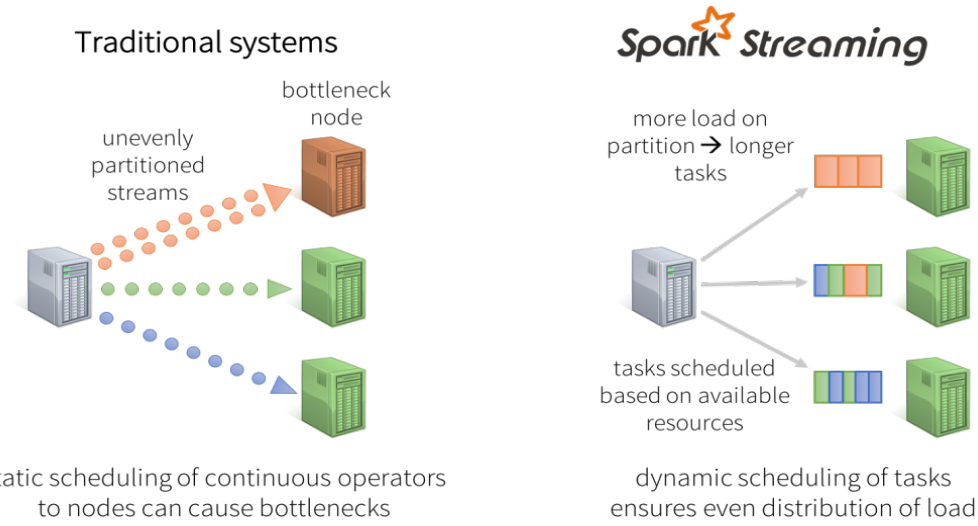

Рисунок 3.2 – Динамічне вирівнювання навантаження

Обробка потоку даних з Spark задовольняє наступні вимоги: швидка відмова та відновлення відставання, існує більш висока ймовірність того, що вузол кластера вийде з ладу або непередбачувано уповільниться. Система повинна мати можливість автоматично відновлюватися від збоїв і відсталих повідомлень, щоб забезпечити результати в реальному часі.

Це важко в безперервних операторних системах, оскільки вони не призначені для динамічного введення нових операторів для спеціальних запитів. Розширена аналітика, така як машинне навчання та SQL-запити призводить до більш складних робочих навантаження, що вимагають постійного навчання та оновлення моделей даних, або навіть запит «останнього» перегляду повідомлення в потоці даних з SQL-запитами. Знову ж таки, наявність загальної абстракції через ці аналітичні завдання значно полегшує роботу розробника.

Розділення даних на малі мікро-партії дозволяє спростити API. Наприклад, розглянемо просте навантаження, в якому потік вхідних даних необхідно розділити за допомогою ключа і обробити. У традиційному підході під час запису, якщо один з розділів є більш обчислювально інтенсивним, ніж інші, вузол статично призначений для того, щоб цей розділ став вузьким місцем і сповільнив би конвеєр. У Spark Streaming навантаження є збалансованим між екзекюторами деякі єкзекютори оброблятимуть декілька більш довгих завдань ніж інші та будуть обробляти більш короткі завдання.Балансування навантаження це нерівномірний розподіл навантаження на обробку між робочими може призвести до вузьких місць у безперервній системі оператора.

3.2 Відмовостійкість обробки потоків даних з Spark Streaming

У випадку збою вузлів, традиційні системи повинні перезапустити невдалий оператор на іншому вузлі і відтворити деяку частину потоку даних для перерахунку втраченої інформації. Зверніть увагу, що тільки один вузол обробляє повторну обчислення, і конвеєр не може продовжуватися, поки новий вузол не наздожене після відтворення обчислень. В Spark обчислення вже поділено на

дрібні, детерміновані завдання, які можуть виконуватися в будь-якому місці, не впливаючи на правильність. Таким чином, невдачі можуть бути відновлені паралельно на всіх інших вузлах кластера, щоб рівномірно розподілити всі перерахування на багатьох вузлах, і відновлюватти їх швидше, ніж традиційний підхід.

Додаток для потокової передачі має працювати 24/7 і, отже, має бути стійким до збоїв, не пов'язаних з логікою програми (наприклад, збої системи, аварії JVM і т.д.), що зображено на рисунку 3.2.

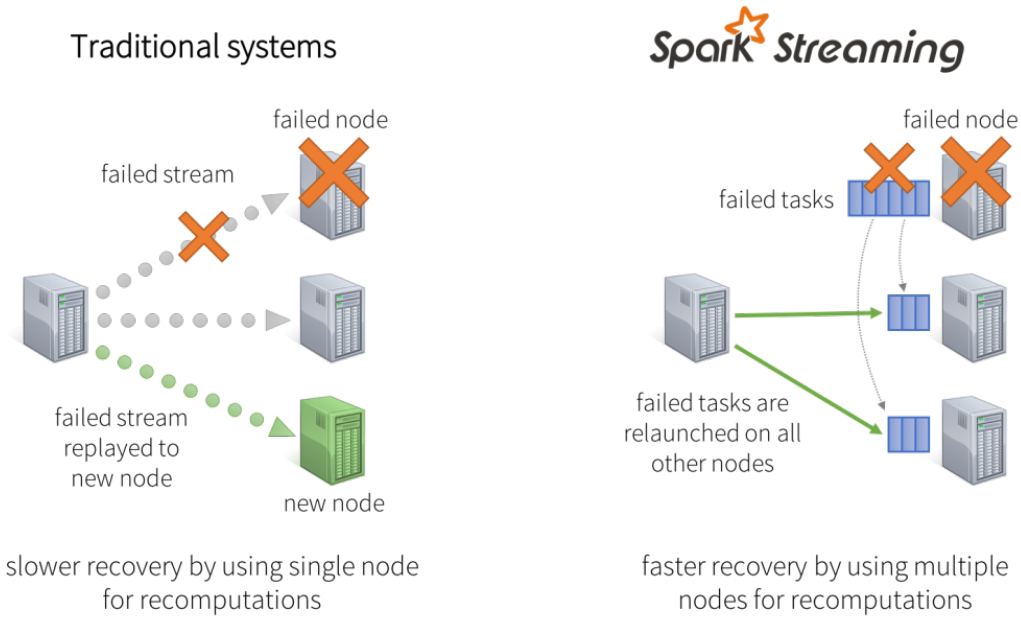

Рисунок 3.3 – Швидше відновлення відмови з перерозподілом обчислень

Щоб це було можливим, Spark Streaming має достатньо інформації для перевірки відмовостійкої системи зберігання даних, щоб вона могла відновитися після збоїв. Є два типи даних, які перевіряються на контрольні точки.

Checkpointing метаданих - збереження інформації, що визначає потокове обчислення для відмовостійкого зберігання, як HDFS. Це використовується для відновлення після збою вузла, що запускає драйвер потокової програми. Метадані включають:

– конфігурація - конфігурація, яка використовувалася для створення потокової програми;
– операції DStream - набір операцій DStream, які визначають потокове додаток;
– неповні пакети, завдання яких ставляться в чергу, але ще не завершені.

Перевірка даних - рбереження генерованих RDD в надійне сховище. Це необхідно в деяких трансформаціях, які поєднують дані в декількох RDD. У таких перетвореннях генерованих RDD, що залежать від RDD попередніх партій, що зумовлює збільшення довжини ланцюга залежностей з часом. Щоб уникнути такого необмеженого збільшення часу відновлення (пропорційно ланцюжку залежностей), проміжні RDD перетворень, що характеризуються станом, періодично перевіряються на надійне зберігання (наприклад, HDFS) для відключення ланцюгів залежностей.

Контрольна точка може бути включена, встановивши каталог у відмовостійкій, надійній файловій системі (наприклад, HDFS, S3 тощо), до якої буде збережена інформація про контрольні точки. Це робиться за допомогою streamingContext.checkpoint. Це дозволить вам використовувати вищезгадані перетворення стану. Крім того, щоб програма відновлювалася від збоїв драйвера, вам слід переписати потокове додаток, щоб мати наступну поведінку.

Підводячи підсумок, перевірка метаданих необхідна в першу чергу для відновлення збоїв драйвера, тоді як дані або контрольна точка RDD необхідні навіть для базового функціонування, якщо використовуються перетворення стану. Це загальне уявлення дозволяє безперешкодно взаємодіяти з пакетними та потоковими робочими навантаженнями. Користувачі можуть застосовувати довільні функції Spark на кожному пакеті потокових даних

3.3 Уніфікація обробки потоків даних з Spark Streaming

Ключовою абстракцією в Spark Streaming є DStream, або розподілений потік. Кожна партія потокових даних представлена RDD, що є концепцією Spark для розподіленого набору даних. Тому DStream - це лише ряд RDD. Це загальне уявлення дозволяє безперешкодно взаємодіяти з пакетними та потоковими робочими навантаженнями. Користувачі можуть застосовувати довільні функції Spark на кожному пакеті потокових даних: наприклад, легко виконати join до DStream з попередньо обчисленим статичним набором даних (як RDD). У коді ця виглядає наступним чином:

```
val dataset = sparkContext.hadoopFile("file")
//Join each batch in stream with the dataset
kafkaDStream.transform { batchRDD =>
batchRDD.join(dataset).filter(...)
}
```

Оскільки пакети потокових даних зберігаються в робочій пам'яті Spark, їх можна інтерактивно запитувати за запитом. Наприклад, можна виставити весь потоковий стан через сервер Spark SQL JDBC. Такий тип уніфікації пакетних, потокових і інтерактивних робочих навантажень у Spark дуже простий, але його важко досягти в системах без загальної абстракції для цих навантажень [8].

RDD, сформовані за допомогою DStreams, можуть бути перетворені в DataFrames (програмний інтерфейс до Spark SQL) і запитуватися за допомогою SQL. Наприклад, використовуючи JDBC-сервер Spark SQL, можливо виставити

стан потоку будь-якій зовнішній програмі, яка розмовляє SQL. У коді ця виглядає наступним чином:

```
val hiveContext = new HiveContext(sparkContext)
// ...
wordCountsDStream.foreachRDD { rdd =>
  // Convert RDD to DataFrame and register it as a SQL table
  val wordCountsDataFrame = rdd.toDF("word", "count")
  wordCountsDataFrame.registerTempTable("word_counts")
}
// ...
// Start the JDBC server
HiveThriftServer2.startWithContext(hiveContext)
```

На практиці здатність Spark Streaming до оброби пакетних даних і використання Spark призводить до порівнянної або більшої пропускної здатності з іншими потоковими системами [9]. З точки зору затримки, Spark Streaming може досягти затримок до декількох сотень мілісекунд. На практиці затримка пакетної обробки є лише малим компонентом затримки наскрізного конвеєра. Наприклад, багато додатків обчислюють результати за часовим вікном, і навіть в безперервних операторних системах, це вікно оновлюється лише періодично (наприклад, 20-секундне вікно, яке ковзає кожні 2 секунди). Багато систем збирають записи з декількох джерел і чекають короткий період часу на обробку даних із затримкою або поза замовленням. Нарешті, будь-який алгоритм автоматичного запуску має тенденцію чекати деякий період часу для запуску

тригера. Пропускна здатність DStreams часто означає, що вам потрібно менше машин для обробки одного і того ж робочого навантаження.

3.4 Уніфікація SS та виндовинг

У структурованому потоковому обчисленні вирішується проблеу ма семантики завдяки надійній гарантії щодо системи: у будь-який час вихід програми еквівалентний виконанню пакетного завдання [10].

Перевагою Structured Streaming є те, що API дуже прості у використанні. Користувачі просто описують запит, який вони хочуть запустити, вхідні та вихідні місця, а також додаткові деталі. Потім система запускає свій запит поступово, підтримуючи достатню кількість ресурсів для відновлення після збою, збереження результатів у зовнішньому сховищі тощо. У коді ця виглядає наступним чином:

```
// Read data continuously from an S3 location
val inputDF = spark.readStream.json("s3://logs")
// Do operations using the standard DataFrame API and write to MySQL
inputDF.groupBy($"action", window($"time", "1 hour")).count()
       .writeStream.format("jdbc")
       .start("jdbc:mysql//…")
```

Цей код практично ідентичний нижній версії пакетної:

```
// Read data once from an S3 location
val inputDF = spark.read.json("s3://logs")

// Do operations using the standard DataFrame API and write to MySQL
inputDF.groupBy($"action", window($"time", "1 hour")).count()
       .writeStream.format("jdbc")
       .save("jdbc:mysql//…")
```

Концептуально Structured Streaming обробляє всі дані, що надходять як необмежена таблиця вводу.

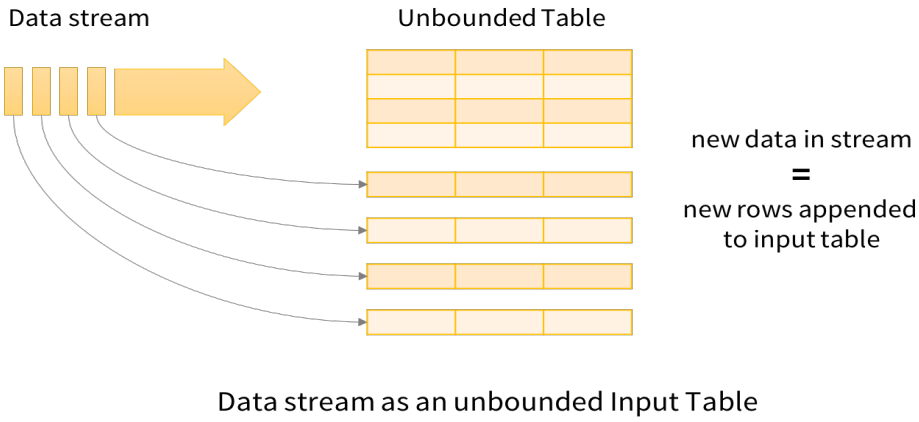

Рисунок 3.4 – Необмежений потік даних та його обробка

Ми фактично не збережемо весь вхід, але наші результати будуть еквівалентними тому, що всі вони і виконують пакетне завдання.

Розробник потім визначає запит на цій вхідній таблиці, так якщо б він був статичною таблицею, щоб обчислити кінцеву таблицю результатів, яка буде записана у вихідну таблицю. Spark автоматично перетворює цей запит, подібний

до партії, до поточного плану виконання. Це називається інкременталізацією: Spark визначає, який стан потрібно підтримувати, щоб оновити результат кожного разу, коли надходить запис. Визначаються тригери для управління, для оновлення результатів. Кожного разу, коли тригер спрацьовує, Spark перевіряє нові дані (новий рядок у вхідній таблиці) і поступово оновлює результат.

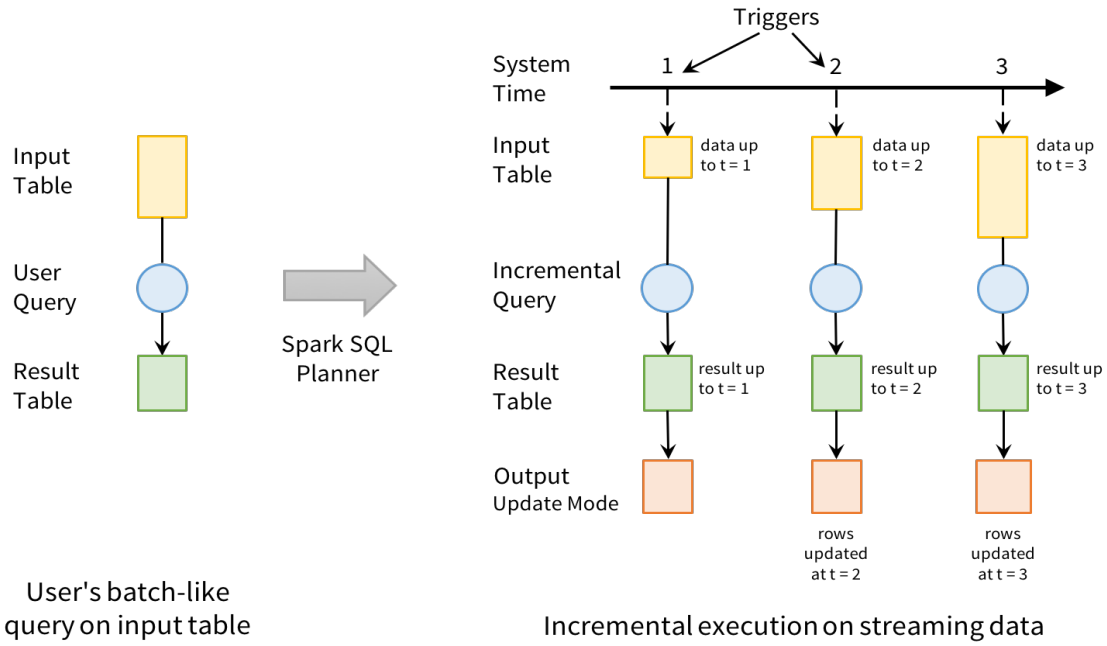

Рисунок 3.5 – Модель обробки струтурованого потоку даних

Остання частина моделі є режимами виводу. Кожного разу, коли таблиця результатів оновлюється, розробник хоче записати зміни у зовнішню систему, наприклад, S3, HDFS або базу даних. Зазвичай ми хочемо писати висновок поступово.

Для цього у Structured Streaming передбачено три стана:

– додавання, тільки нові рядки додаються до таблиці результатів, оскільки останній тригер буде записано у зовнішню пам’ять. Це стосується лише запитів, де існуючі рядки у таблиці результатів не можуть змінюватися (наприклад, карта на вхідному потоці);
– завершення, всю оновлену таблицю результатів буде записано у зовнішню пам’ять;
– оновлення, у зовнішній пам'яті буде змінено лише рядки, які були оновлені в таблиці результатів з моменту останнього тригера. Цей режим працює для вихідних поглиначів, які можуть бути оновлені на місці, такі як таблиця MySQL.

Пакетний запит відповідає за обчислення кількості дій, згрупованих за період часу. Щоб запустити цей запит поступово, Spark зберігає стан з підрахунками для кожної пари і оновлюється після надходження нових записів. Для кожного зміненого запису виводяться дані відповідно до режиму виводу.

У кожній точці тригера ми приймаємо попередні згруповані лічильники і оновлюємо їх новими даними, які надходять з моменту останнього тригера, щоб отримати нову таблицю результатів. Потім ми випускаємо лише зміни, що вимагаються нашим виходом, тут ми оновлюємо записи для пар, які змінилися під час цього тригера в MySQL (показано червоним кольором) [11].

Зауважемо, що система також автоматично обробляє пізні дані. На малюнку вище, "відкрита" подія для phone3, яка сталася в 1:58 на телефоні, тільки потрапляє в систему в 2:02. Незважаючи на це, незважаючи на те, що минуло 2 години, ми оновлюємо запис на 1:00 в MySQL. Однак гарантія цілісності префікса в структурованому потоці гарантує, що ми обробляємо записи з кожного джерела в порядку їх надходження.

Однак гарантія цілісності префікса в структурованому потоці що зображена на малюнку 3.6 нижче показано це виконання за допомогою режиму виводу оновлення:

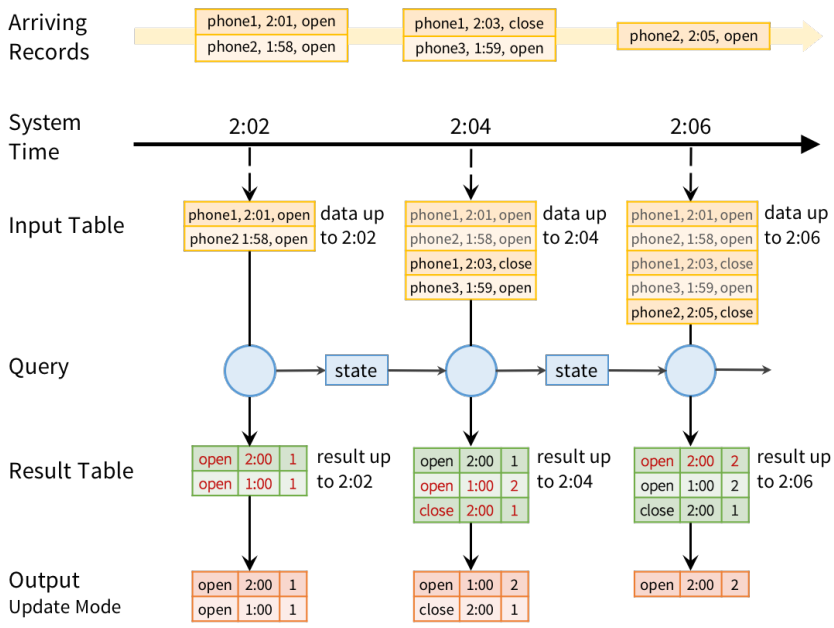

Рисунок 3.6 – Модель режиму виводу оновлення потоку даних

Наприклад, оскільки подія «закриття» телефону1 надходить після його «відкритої» події, ми завжди будемо оновлювати «відкриті» підрахунки, перш ніж оновити кількість «закритих».

Структурований потік інтегрований у API даних набору та інтерфейсу DataFrame, у більшості випадків потрібно лише додати кілька викликів методів для запуску потокового обчислення. Він також додає нові оператори для агрегованого вікна та для налаштування параметрів моделі виконання (наприклад, режими виводу). У Apache Spark 2.0 створено альфа-версію системи з основними API [12].

Програми структурованого потокового передавання можуть використовувати існуючі методи DataFrame і Dataset для перетворення даних, включаючи map, filter, join та інші. Крім того, запущені (або нескінченні) агрегації, такі як кількість від початку часу, доступні через існуючі API. Це те, що ми використовували в нашому моніторинговому додатку вище.

Потокові програми часто повинні обчислювати дані на різних типах вікон, включаючи ковзаючі вікна, які перекриваються один з одним (наприклад, 1-годинне вікно, що просувається кожні 5 хвилин), і обвалювальні вікна, які не мають цієї характеристики (наприклад, щогодини). У структурованому потоці вікно просто представлено як груповий оператор, це показано малюнку 3.6 нижче

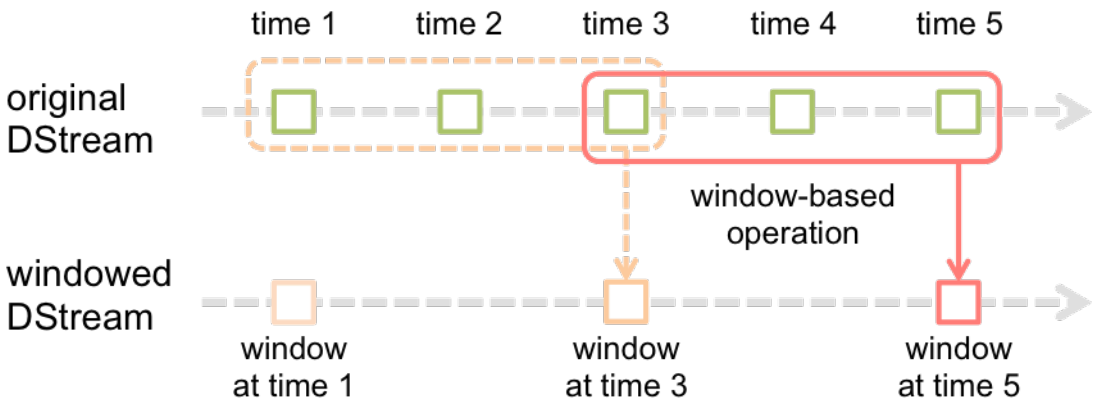

Рисунок 3.7 – Модель обробки вікна в Spark

Кожна вхідна подія може бути зіставлена з одним або кількома вікнами і просто призводить до оновлення одного або декількох рядків таблиці результатів [13]. Як показано на малюнку, кожен раз, коли вікно ковзає над джерелом DStream, джерела RDD, які потрапляють у вікно, об'єднуються і працюють для створення RDD з DStream [14]. У цьому конкретному випадку операція застосовується протягом останніх 3 одиниць часу даних і слайдів по 2 одиниці часу. Це показує, що будь-яка операція з вікном повинна вказати два параметри.

У той час як наша попередня програма виводила результати форми (час, дія, кількість), ця нова буде виводити результати форми (вікно, дія, кількість), наприклад ("1: 10-2: 10", "open") , 17). Якщо прибуде пізій запис, то будуть оновлені всі відповідні вікна в MySQL. І на відміну від багатьох інших систем, вікно не просто спеціальний оператор для потокових обчислень; мождливо запускати той самий код у пакетному завданні, щоб групувати дані таким же чином. Windows можна вказати за допомогою віконної функції в DataFrames.

За замовчуванням, Structured Streaming з використанням файлів на основі джерела у вигляді брокера повідомлень вимагає, щоб схема була указана, Spark, не може вивести автоматично схему в даному випадку. Це обмеження гарантує, що послідовна схема буде використовуватися для потокового запиту, навіть у випадку збоїв. Якщо стовпці з'являються в наданій користувачем схемі, вони будуть заповнені Spark на основі шляху до файлу, що читається. Каталоги, які складають схему розділення, повинні бути присутніми, коли запит починається, і повинні залишатися статичними [15].

3.5 Порівняння з Flink

Apache Flink - це платформа з відкритим вихідним кодом, яка надає вам величезні можливості для запуску конвеєрів обробки даних у реальному часі у відмовостійкий спосіб, в масштабі мільйонів подій в секунду.

Ключовим моментом є те, що він робить все це, використовуючи мінімально можливі ресурси при латентності в декілька мілісекунд.

Flink базується на моделі DataFlow, тобто обробляє елементи, як і коли вони приходять, а не обробляють їх у мікропартіях (що робиться потоковим використанням Spark).

Мікро-партії можуть містити величезну кількість елементів, а ресурси, необхідних для обробки цих елементів одночасно, що можуть бути істотними. У випадку розрідженого потоку даних (в якому можна отримати тільки пакет даних на нерегулярних інтервалах), це стає основною проблемою.

Також не потрібно проходити випробування та помилки налаштування розміру мікро-партії, щоб час обробки пакету не перевищував час накопичення. Якщо це трапиться, то партії починають ставати в чергу і в кінцевому підсумку вся обробка припиниться. Dataflow дозволяє Flink обробляти мільйони записів на хвилини з затримкою в декілька мілісекунд на одній машині.

Flink забезпечує надійну відмовостійкість від несправностей за допомогою контролю збоїв (періодично зберігаючи внутрішній стан до зовнішніх джерел, таких як HDFS).

Однак механізм контрольно метаданих Flink може бути зроблений інкрементно (за винятком лише змін, а не всього стану), що дійсно зменшує обсяг даних у HDFS і тривалість введення-виведення. Накладні витрати на контроль метаданих майже незначні, що дозволяє користувачам мати постійні стани в програмах Flink [16].

Flink також забезпечує високе налаштування доступності через Zookeeper. Це призначено для повторного виникнення завдання у випадках, коли драйвер (який називається JobManager в Flink) вийшов з ладу.

Іноді для виконання операцій потрібна деяка конфігурація або дані з іншого джерела. Простим прикладом може бути підрахунок кількості записів типу Y в потоці X. Цей лічильник буде відомий як стан операції.

Flink надає прості API для взаємодії, це нагадує взаємодію з об'єктом в Java. Стани можуть бути підкріплені пам'яттю, файловою системою або RocksDB, які перевіряються і, отже, є стійкими до помилок. Що стосується наведеного вище прикладу, у разі перезапуску вашої програми значення лічильника буде збережено [17].

Apache Flink забезпечує exactly one delivery семантику, подібну до Kafka версії 0.11 і вище, з мінімальними накладними витратами і нульовим зусиллям. Це не тривіально зробити в інших потокових рішеннях, таких як Spark Streaming і Storm, і зовсім не підтримується в Apache Samza.

Завдання у Flink можна запускати в розподіленій системі або в локальній машині. Програма може працювати на Mesos, Yarn, Kubernetes, а також в автономному режимі (наприклад, в контейнерах докера). Hadoop не є місцем, яке дає ряд можливостей, де можна запустити додаток Flink.

3.6 Інтеграція з Apache Kafka

Підхід положлений в Apache Kafka використовує приймач для отримання даних. Приймач реалізований з використанням високоякісного споживчого API Kafka. Як і всі приймачі, дані, отримані від Kafka через Receiver, зберігаються у виконавцях Spark, а потім завдання, запущені в Spark Streaming, обробляють дані.

Однак, за умовчанням, цей підхід може втратити дані при. Щоб забезпечити нульову втрату даних, потрібно додатково ввімкнути журнали запису в потоці Spark Streaming (введені в Spark 1.2), що одночасно зберігає всі

отримані Kafka дані в журналах попереду запису на розподіленій файловій системі (наприклад, HDFS).

Topics є основними примітивом в Kafka, вони не співвідносяться з розділами RDD, що генеруються в Spark Streaming. Таким чином, збільшення кількості topics завдяки KafkaUtils.createStream, тільки збільшує кількість потоків, що використовують їх, які споживаються в межах одного приймача. Це не збільшує паралелізм Spark при обробці даних.

Багатоканальні вхідні Dfreams можуть бути створені з різними групами та темами для паралельного прийому даних з використанням декількох приймачів.

Якщо журнали запису біли ввімкнуті на початку запису з реплікаційною файловою системою, як HDFS, отримані дані вже реплікуються в журналі.

Для програм Scala та Java, слід використовувати SBT або Maven для керування проектами, то пакувати spark-streaming-kafka-0-8_2.11 і його залежності в додатковий JAR. Переконайтеся, що spark-core_2.11 та spark-streaming_2.11 позначені як надані залежності, оскільки вони вже є в установці Spark [18].

Цей новий «direct» підхід без використання одержувачів був запроваджений у Spark 1.3, щоб забезпечити посилені гарантії. Замість того, щоб використовувати приймачі для прийому даних, цей підхід періодично запитує Кафку для останніх зсувів у кожному topic, і відповідно визначає діапазони зміщення для обробки в кожному пакеті. Після запуску завдань для обробки даних, простий споживчий API Kafka використовується для зчитування визначених діапазонів відхилень від Kafka (подібно до читання файлів з файлової системи). Зауважимо, що ця функція була введена в Spark 1.3 для Scala і Java API, та у Spark 1.4 для Python, підхід без використання одержувачів був запроваджений у Spark 1.3, щоб забезпечити посилені гарантії. Замість того, щоб використовувати приймачі для прийому даних, цей підхід періодично запитує

Кафку для останніх зсувів і відповідно визначає діапазони зміщення для обробки в кожному пакеті. Після запуску завдань для обробки даних, простий споживчий API Kafka використовується для зчитування визначених діапазонів відхилень від Kafka (подібно до читання файлів з файлової системи). Зауважимо, що ця функція була введена в Spark 1.3 для Scala і Java API, у Spark 1.4 для API Python.

Спрощений паралелізм: немає необхідності створювати декілька потоків Kafka і їх об'єднання. Завдяки DirectStream, Spark Streaming створить стільки розділів RDD, скільки і розділів Kafka для споживання, які паралельно зчитують дані з Kafka. Таким чином, існує взаємне відображення між розділами Kafka і RDD, які легше зрозуміти і налаштувати.

Ефективність: Досягнення нульової втрати даних у першому підході вимагало збереження даних у журналі запису на початку запису, який додатково реплікував дані. Це насправді неефективно, оскільки дані ефективно повторюються двічі - один раз за допомогою Kafka, а другий - за допомогою журналу Write ahead. Цей другий підхід усуває проблему, оскільки немає приймача, і, отже, немає необхідності в журналах запису. До тих пір, поки у вас є достатня кількість брокерів, повідомлення можуть бути відновлені з Kafka.

Exactly one semantics це підхід використовує високорівневий API Kafka для зберігання зміщень у Zookeeper. Це традиційно спосіб споживання даних з Kafka. Хоча цей підхід (у поєднанні з логами запису вперед) може забезпечити нульову втрату даних (тобто, принаймні, exactly one semantics), існує невелика ймовірність, що деякі записи можуть споживатися двічі при збоях. Це відбувається через невідповідності між даними, які надійно отримуються в Spark Streaming, і зміщеннями, які відстежуються в Zookeeper. Отже, у цьому другому підході ми використовуємо простий Kafka API, який не використовує Zookeeper. Відхилення відслідковуються потоком Spark у межах контрольних точок. Це усуває невідповідності між Spark Streaming і Zookeeper/Kafka, і кожен запис отримується в Spark Streaming ефективно рівно один раз, незважаючи на відмови.

Для того, щоб досягти точної семантики для виведення результатів, операція виводу, яка зберігає дані у зовнішньому сховищі даних, повинна бути ідемпотентною або атомарною транзакцією, яка зберігає результати та зміщення.

Новий споживач API Kafka попередньо отримуватиме повідомлення в буфери. Тому для забезпечення продуктивності важливо, щоб інтеграція Spark зберігала кешованих споживачів на виконавцях (замість того, щоб відтворювати їх для кожної партії), і потрібно планувати розділи на місцях, де є відповідні споживачі.

Якщо екзекютори знаходяться на тих же хостах, що і ваші брокери Kafka, слід скористатися пунктом PreferBrokers, який віддасть перевагу розкладам розділів на лідера Kafka для цього розділу. Нарешті, якщо є значний перекіс у завантаженні серед розділів, то слід використати PreferFixed.

Kafka також має більш сильні гарантії цілісності, ніж традиційні системи обміну повідомленнями.

Традиційно, Kafka зберігає записи в порядку на сервері, і якщо декілька споживачів споживають з черги, то сервер роздає записи в тому порядку, в якому вони зберігаються. Однак, хоча сервер передає записи по порядку, записи доставляються асинхронно споживачам, тому вони можуть вийти з ладу на різних споживачах. Це ефективно означає, що порядок записів втрачається при наявності паралельного споживання. Системи обміну повідомленнями часто обходяться цим шляхом, маючи поняття ексклюзивний споживач, що дозволяє тільки одному процесу споживати з черги, але, звичайно, це означає, що немає паралельності в обробці.

Маючи поняття паралелізму - розділ, у межах тематики, Kafka здатна забезпечити як гарантії впорядкування, так і балансування навантаження над пулом споживчих процесів. Це досягається шляхом призначення розділів у темі споживачам у групі споживача, так що кожен розділ споживається рівно одним

споживачем в групі. Роблячи це, ми гарантуємо, що споживач є єдиним читачем цього розділу і споживає дані в порядку. Оскільки існує багато розділів, це все ще врівноважує навантаження на багато споживчих екземплярів.

У Kafka потоковий процесор - це все, що приймає безперервні потоки даних з вхідних та виконує певну обробку на цьому вході і виробляє безперервні потоки даних до вихідних тем.

Наприклад, додаток може приймати потоки вхідних даних продажів і відправлень, а також виводити потік повторних замовлень і коригування цін, виведене з цих даних [19].

Можна робити просту обробку безпосередньо за допомогою API виробників і споживачів. Однак для більш складних перетворень Kafka надає повністю інтегрований Streams API. Це дозволяє створювати програми, які роблять нетривіальну обробку і обчислюють агрегати з потоків або об'єднують потоки разом.

Ця функція допомагає вирішувати важкі проблеми, пов'язані з цим типом застосунків: обробкою даних, що вийшли з ладу, переробкою вхідних даних як змін у коді, виконанням обчислень, що виконують стан, і т.д.

Streams API побудовані на основних примітивах Kafka, що передбачає використання API виробників і споживачів для введення, Kafka використовує це для збереження стану і використовує той же механізм групи для відмовостійкості серед екземплярів потокового процесора.

Така комбінація обміну повідомленнями, зберігання та обробки потоків може здатися незвичною, але вона є важливою для ролі Kafka як платформи для потокової передачі.

Розподілена файлова система, як HDFS, дозволяє зберігати статичні файли для пакетної обробки. Ефективно така система дозволяє зберігати та обробляти історичні дані з минулого.

Традиційна система обміну повідомленнями підприємства дозволяє обробляти майбутні повідомлення, які надходять після підписки. Додатки, побудовані таким чином, обробляють майбутні дані під час їх надходження.

Kafka поєднує обидві ці можливості, і ця комбінація є критичною як для використання Kafka як платформи для потокових додатків, так і для потокових конвеєрів даних [20].

Об'єднуючи підписки на зберігання та низьку затримку, потокові програми можуть розглядати як минулі, так і майбутні дані однаковим чином. Тобто єдиний додаток може обробляти історичні, збережені дані, а не закінчуватися, коли вони досягають останнього запису, який він може продовжувати обробляти, оскільки приходять дані майбутнього. Це узагальнене уявлення про обробку потоку, що включає в себе пакетну обробку, а також додатки, керовані повідомленнями.

Аналогічно для потокових конвеєрів даних комбінація підписки на події в реальному часі дає можливість використовувати Kafka для трубопроводів з дуже низькою затримкою; але здатність зберігати дані надійно дає можливість використовувати її для критичних даних, де доставка даних повинна бути гарантована або для інтеграції з автономними системами, які завантажують дані лише періодично або можуть тривати протягом тривалого періоду часу на технічне обслуговування. Засоби обробки потоку дозволяють трансформувати дані під час їх надходження.

ВИСНОВКИ

У ході виконання атестаційної роботи було досліджено проблему обробки потоків даних у режимі близькому до реально часу з використанням технологій паралельної та розподіленої обробки даних.

У процесі дослідження в першому розділі роботи виявлено, що на даний момент існує велика кількість методів обробки великих даних, заточених на вирішення конкретних проблем. Була досліджена задача розподілення обчислень великих даних, яка використовувалася для вирішення заданої проблеми, описано особливості та проблематики задачі розподілених обчислень.

Другий розділ був присвячений вивченню існуючих методів вирішення проблеми, побудованої архітектури, описаної конфігурації системи. Проаналізовано основні компоненти Apache Spark, низькорівненеві та високорівневі примітиви даної платформи, способи завантаження додатків до хмарних середовищів та взаємозв’язок з Hadoop.

У результаті розробки вдалося досягти поставленої задачі, порівняти алгоритми обробки потоків даних в режимі близького до реально часу з урахуванням переваг та недоліків кожного з методів з урахуванням ефективного використання низькорівневих примітивів обробки великих даних.

У третьому розділі проводився аналіз отриманих результатів, а саме порівняння продуктивності Spark Streaming з Apache Flink при вирішені різних задач обробки потоку даних, взаємодія з брокерами повідомлення такими як Apache Kafka. Були розглянуті алгоритми обробки даних для обробки повідомлень, що прийшли із запізненням, алгоритми відновлення обробки

потоку даних після переривання, методи покращення швидкості обробки потоку, зменшення затримок.

Актуальність роботи полягає у тому, що розподіленні обчислення зменшують вартість та час, що необхідний для виконання завдання обробки великих масивів даних. Інтерес до методів і технологій розробки програмних продуктів на розподілених кластерах з метою обробки потоків даних стрімко збільшується різ за роком, так як є актуальним у багатьох бізнес доменах для отримання цінної інформації у реальному часі.

Шляхи подальшого розвитку предмета дослідження це використання вдосконалених методів хмарних обчислень із застосуванням більшої кількості потужних систем обчислень, реалізація алгоритмів об’єднання потоків даних.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Zaharia, Matei. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In Memory Cluster Computing. [Текст] : матеріали 2011 ACM SIGMOD/PODS міжнар. наук. форум, 12 червня 2011р. New York / редкол. : Zaharia, Matei; Mosharaf Chowdhury; Tathagata Das и др. - New York: ACM New York, 2011.-322с.

2. Zaharia, Matei. Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing. [Текст] : матеріали 2012 ACM SIGMOD/PODS міжнар. наук. форум, 14 червня 2012р. New York / редкол. : Zaharia, Matei; Mosharaf Chowdhury; Tathagata Das и др. - New York: ACM New York, 2012.-322с.

3. Система запитань та відповідей Stack Overflow [Електронний ресурс] / портал stackoverflow.com : Stack Overflow FAQ. - Режим доступу: www/URL: http://stackoverflow.com/

4. Martin Odersky. Programming in Scala - Mountain View: Aritma, 2010. – 883с.

5. Eric Summer. Hadoop Operations - Sebastopol: O'Reilly Media, 2012. – 267с.

6. Tom White. Hadoop: The Definitive Guide - Sebastopol: O'Reilly Media, 2009. – 479с.

7. Томас Х. Алгоритмы. Построение и анализ [Текст]/ Х. Томас, И. Чарльз.- М.:Вильямс, 2016.-1328с.

8. Pollard, J. M. Monte Carlo methods for index computation / Pollard, J. M. / Mathematics of Computation - 1978 – Vol. 32, no. 143. – P. 918-924.

9. Daniel G. Waddington, Nilabja Roy, Douglas C. Schmidt. Dynamic Analysis and Profiling of Multi-threaded Systems

10. М.Г. Адигеев Введение в теорию сложности [Текст] / М.Г. Адигеев.- Ростов-на-Дону:РГУ, 2004.-35с.

11. JMH [Электронный ресурс] / OpenJDK - Режим доступа: https://openjdk.java.net/projects/code-tools/jmh/ - 15.04.2019 г. - Загл. с экрана

12. Томас Х. Алгоритмы. Построение и анализ [Текст]/ Х. Томас, И. Чарльз.-М.:Вильямс, 2016.-1328с.

13. Joshua D. Suereth. Scala in Depth - Shelter Island: Manning Publications, 2012. – 304с.

14. Nilanjan Raychaudhuri. Scala in Action - Shelter Island: Manning Publications, 2013 – 416с.

15. Xin, Reynold. Shark: SQL and Rich Analytics at Scale. [Текст] : матеріали 2013 ACM SIGMOD/PODS міжнар. наук. форум, 22 червня 2013р. New York / редкол. : Xin, Reynold; Rosen, Josh; Zaharia, Matei; Franklin и др. - New York: ACM New York, 2013.-322с.

16. Adam Silberstein, Joe Hellerstein. Data Profiling in the Big Data Era, 2015

17. Приемы объектно-ориентированного проектирования. Паттерны проектирования [Текст] / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – СПб.: Питер, 2007. – 366 с.

18. Marko Bonaći, Petar Zečević, ”Spark in Action”, 2015, early access edition

19 Шилдт Г. Java: Полное руководство Java SE 7 [Текст] / Г. Шилдт. – 8-е изд. – М.: Вильямс, 2012. – 1104 с.=

20. John R. Koza, ”Genetic Programming II: Automatic Discovery of Reusable Programs”. Cambridge Massachusetts: MIT Press, may 1994.