Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Уральский федеральный университет имени первого Президента России Б. Н. Ельцина»

УТВЕРЖДАЮ

Директор по образовательной деятельности

_____ С.Т. Князев

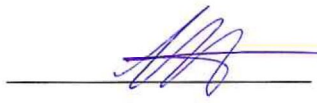
«7» октября 2023 г.

Пакетная и потоковая обработка данных

Учебно-методические материалы по направлению подготовки
09.03.03 Прикладная информатика
Образовательная программа «Прикладной искусственный интеллект»

Екатеринбург

РАЗРАБОТЧИКИ УЧЕБНО-МЕТОДИЧЕСКИХ МАТЕРИАЛОВ

Доцент кафедры информационных технологий и систем управления		М.В. Ронкин
Ассистент кафедры информационных технологий и систем управления		А.С. Аксенов

СОДЕРЖАНИЕ

1. РАСПРЕДЕЛЕННЫЕ ДАННЫЕ, РАСПРЕДЕЛЕННАЯ ОБРАБОТКА И СОВРЕМЕННЫЕ ТРЕБОВАНИЯ К СКОРОСТИ И НАДЕЖНОСТИ ВЫЧИСЛЕНИЙ

1.1 Краткая история появления инструментов для хранения и обработки больших данных. Возникновение концепции распределенных данных

Всё началось в первые годы нового тысячелетия, когда уже подросший стартап в Маунтин-Вью под названием Google пытался проиндексировать весь растущий интернет. Перед ними стояли два основных вызова, которые ранее ещё никто не решал:

Как разместить сотни терабайт данных на тысячах дисков, установленных в более, чем тысяче машин, без даунтаймов, потери информации и с сохранением её постоянной доступности?

1. Как распараллелить вычисление эффективным и отказоустойчивым способом для обработки всех этих данных на всех машинах?

2. Чтобы лучше понять всю сложность такой затеи, представьте себе кластер с тысячами машин, среди которых из-за сбоев как минимум одна всегда находится на обслуживании.

С 2003 по 2006 годы Google выпустили три исследовательские работы, объясняющие внутреннюю архитектуру данных. Эти работы навсегда изменили индустрию Big Data. Первая вышла в 2003 году под названием “The Google File System”. Вторая последовала за ней в 2004 году и называлась “MapReduce: Simplified Data Processing on Large Clusters”. Согласно Google Scholar, с тех пор её процитировали более 21 000 раз. Третий научный труд вышел в 2006 году и назывался “Bigtable: A Distributed Storage System for Structured Data”.

И пусть даже эти работы оказали решающее влияние на появление Hadoop, сама компания Google к этому появлению отношения не имела, поскольку хранила свой исходный код закрытым.

Тем временем основатель Hadoop, сотрудник компании Yahoo! Дуг Каттинг, на тот момент уже разработавший Apache Lucene (поисковая библиотека, лежащая в основе Apache Solr и ElasticSearch), работал над проектом сильно распределённого поискового модуля под названием Apache Nutch.

Подобно Google, этот проект для достижения широкого масштаба нуждался в распределённом хранилище и серьёзных вычислительных возможностях. Прочитав работы Google по Google File System и MapReduce, Дуг осознал ошибочность своего текущего подхода, а описанная в тех работах архитектура вдохновила его на создание в 2005 году дочернего проекта для Nutch, который в честь игрушки (жёлтого слона) своего сына он назвал Hadoop.

Этот проект начался с двух ключевых компонентов: распределённой файловой системы Hadoop (HDFS) и реализации фреймворка MapReduce. В отличие от Google, компания Yahoo! решила открыть исходный код проекта в рамках Apache Software Foundation. Тем самым они пригласили все остальные ведущие компании к его использованию и участию в развитии, чтобы сократить технологическое отставание от своих соседей.

Довольно скоро другие компании, начав использовать Hadoop, столкнулись с аналогичными проблемами обработки больших объёмов данных. В те времена это означало огромные обязательства, поскольку им требовалось организовать и самостоятельно управлять кластерами машин, а написание задачи MapReduce явно не представлялось лёгкой прогулкой. Попытка Yahoo! уменьшить сложность программирования этих задач реализовалась в виде Apache Pig, ETL-инструмента, способного переводить собственный язык Pig Latin в шаги MapReduce. Однако вскоре к развитию этой новой экосистемы подключились и другие.

В 2007 году молодая быстро растущая компания Facebook, под началом 23-летнего Марка Цукерберга, выпустила в открытый доступ два новых проекта под лицензией Apache: Apache Hive и через год Apache Cassandra. Apache Hive – это фреймворк, способный преобразовывать SQL-запросы в задачи MapReduce для Hadoop. При этом Cassandra является обширным столбцовым хранилищем, предназначенным для широкомасштабного распределённого доступа к контенту и его обновления. Это хранилище не требовало для своей работы Hadoop, но вскоре, когда были созданы коннекторы для MapReduce, стало частью этой экосистемы.

В то же время менее известная компания Powerset, работавшая над поисковым движком, вдохновилась работой Google по Bigtable и разработала Apache Hbase, ещё одно столбцовое хранилище, опирающееся на HDFS. Вскоре после этого Powerset была поглощена корпорацией Microsoft, которая запустила на её основе новый проект под названием Bing.

Кроме всего прочего, на быстрое внедрение Hadoop решительно повлияла ещё одна компания – Amazon. Запустив Amazon Web Services, первую облачную платформу с доступом к ресурсам по требованию, и быстро добавив поддержку MapReduce через сервис Elastic MapReduce, эта компания позволила стартапам с удобством хранить свои данные в S3, распределённой файловой системе, а также развёртывать и выполнять в ней задачи MapReduce, исключая лишнюю возню с кластером Hadoop.

В очередной раз Google косвенно оказала решающее влияние на мир больших данных, выпустив в 2010 году четвёртую исследовательскую работу под заголовком “Dremel: Interactive Analysis of Web-Scale Datasets”. В ней описывались две основных инновации:

1. Архитектура распределённых интерактивных запросов, вдохновившая появление большинства интерактивных SQL-инструментов, о которых будет сказано ниже.

2. Форма столбцового хранилища, которая легла в основу нескольких новых форматов хранения данных, таких как Apache Parquet,

совместно разработанного Cloudera и Twitter, и Apache ORC, выпущенного Hortonworks совместно с Facebook.

Вдохновлённая работой по Dremel, компания Cloudera в стремлении решить проблему высокой задержки в Hive и оторваться от конкурентов в 2012 году решила запустить новый открытый SQL-движок для интерактивных запросов под названием Apache Impala. Параллельно с этим MapR запустила собственный опенсорсный интерактивный движок, назвав его Apache Drill. А вот руководство Hortonworks, вместо создания нового движка с нуля, предпочло заняться ускорением работы Hive, запустив проект Apache Tez, некое подобие версии 2 для MapReduce, и адаптировав Hive под выполнение Tez вместо MapReduce. В основе этого решения лежало две причины: во-первых, компания располагала меньшим ресурсом сотрудников, чем Cloudera, а во-вторых, большинство её клиентов уже использовали Hive и предпочли бы ускорить его работу, а не переходить на иной SQL-движок. Как мы дальше узнаем, вскоре появилось множество других распределённых механизмов, и новым слоганом стало «Всё быстрее Hive».

Hadoop 2.0 и революция Spark

В то время как Hadoop укреплял свои позиции и был занят внедрением нового ключевого компонента YARN для управления ресурсами, с чем ранее неуклюже справлялся MapReduce, началась небольшая революция, связанная с резким ростом популярности Apache Spark. Стало очевидно, что Spark окажется отличной заменой MapReduce ввиду более широких возможностей, простого синтаксиса и высокого быстродействия, которое, в частности, обуславливалось его способностью кэшировать данные в ОЗУ. Единственным слабым местом в сравнении с MapReduce в первое время была нестабильность Spark, но эта проблема по мере развития продукта была решена.

Этот инструмент также имел высокую операционную совместимость с Hive, поскольку SparkSQL основывался на синтаксисе Hive (в действительности изначально разработчики Spark позаимствовали у Hive

лексер и парсер), что существенно упрощало переход с Hive на SparkSQL. Вдобавок к этому — Spark привлёк обширное внимание в мире машинного обучения, так как прежние попытки писать алгоритмы МО через MapReduce, например, Apache Mahout, явно проигрывали реализациям Spark.

Для поддержки и монетизации быстрого роста Spark его создатели в 2013 году запустили Databricks. Целью этого проекта стало предоставление каждому возможности обработки огромных объёмов данных. Для этого платформа реализовала простые и эффективные API на многих языках (Java, Scala, Python, R, SQL и даже .NET), а также нативные коннекторы для многих источников и форматов данных (csv, json, parquet, jdbc, avro, etc.). Интересно здесь то, что рыночная стратегия Databricks отличалась от стратегий предшественников. Вместо предложения локального развёртывания Spark, компания заняла позицию исключительно облачной платформы, изначально интегрировавшись с сервисом AWS (который на тот момент являлся самым популярным облаком), а затем с Azure и GCP. Девять лет спустя, можно уверенно сказать, что это был грамотный ход.

Тем временем для обработки потоковых событий появлялись новые открытые проекты вроде Apache Kafka, распределённой очереди сообщений, разработанной LinkedIn, и Apache Storm [3], движка распределённых потоковых вычислений от Twitter. Оба инструмента вышли в 2011 году. В тот же период платформа Amazon Web Services достигла небывалой популярности и успеха: это можно продемонстрировать одним только резким скачком развития Netflix в 2010 году, ставшим возможным преимущественно благодаря облаку Amazon. В облачной сфере начала зарождаться конкуренция. Сначала в 2010 году появился сервис Microsoft Azure, следом за которым в 2011 родилась Google Cloud Platform (GCP).

С тех пор число проектов, являвшихся частью экосистемы Hadoop, продолжило расти экспоненциально. Большинство из них начали разрабатываться до 2014 года, и некоторые вышли под открытой лицензией также до этого момента. Какие-то проекты даже начали вносить путаницу,

поскольку была достигнута точка, когда для каждой потребности уже существовало множество программных решений.

Также начали появляться и более высокоуровневые проекты вроде Apache Apex (закрыт) или Apache Beam (в основном продвигаемый Google), нацеленные на предоставление унифицированного интерфейса для обработки пакетных и потоковых процессов поверх различных распределённых бэкендов, таких как Apache Spark, Apache Flink или Google DataFlow.

Также можно упомянуть, что на рынок, наконец, начали поступать хорошие опенсорсные планировщики — спасибо Airbnb и Spotify. Использование планировщика обычно привязано к бизнес-логике корпорации, и пишется это ПО вполне естественно без особой сложности, как минимум поначалу. Затем ты всё же понимаешь, что очень трудно сохранять его простым и понятным для других. Именно поэтому практически все крупные технологические компании написали собственные, а иногда и открытые, продукты: Yahoo! — Apache Oozie, LinkedIn — Azkaban, Pinterest — Pinball (закрыт) и многие другие.

Однако ни одно из этих решений так и не было признано лучшим, в связи с чем большинство компаний придерживались собственных. К счастью, где-то в 2015 году Airbnb запустили открытый проект Apache Airflow, а Spotify – Luigi[4]. Это были два планировщика, которые быстро завоевали интерес и были приняты на вооружение многими компаниями. В частности, Airflow сейчас в режиме SaaS доступен на Google Cloud Platform и Amazon Web Services.

Со стороны SQL тоже возникло несколько распределённых хранилищ данных, нацеленных на предоставление возможности ускоренной обработки запросов в сравнении с Apache Hive. Мы уже говорили о Spark SQL и Impala, но также стоит упомянуть Presto, открытый проект, запущенный Facebook в 2013 году. В 2016 году этот проект компания Amazon в рамках своего предложения SaaS переименовала в Athena, а когда его начальные

разработчики покинули Facebook, то они сделали отдельный форк, который назвали Trino.

В то же время появилось и несколько проприетарных распределённых хранилищ для аналитики, к которым можно отнести Google BigQuery (2011), Amazon Redshift (2012) и Snowflake (2012).

Полный список всех проектов, числящихся как часть экосистемы Hadoop, можно найти на этой странице.

На смену Hadoop приходит контейнеризация и глубокое обучение

Первым трендом стало массовое перемещение инфраструктур данных в облако, при котором HDFS оказалась заменена облачными хранилищами вроде Amazon S3, Google Storage и Azure Blob Storage.

Вторым трендом была контейнеризация. Вы наверняка слышали о Docker и Kubernetes. Docker – это фреймворк контейнеризации, который вышел в 2011 году и с 2013 начал резко набирать популярность. В июне 2014 Google выпустили открытый инструмент для оркестрации контейнеров под названием Kubernetes (он же K8s), который тут же взяли на вооружение многие компании для построения новых распределённых/масштабируемых архитектур. Docker и Kubernetes позволили развёртывать новые виды таких архитектур, уже более стабильные, надёжные и пригодные для множества случаев, включая событийно-ориентированные преобразования в реальном времени. Hadoop потребовалось время, чтобы поспеть за Docker, и поддержка запуска в нём контейнеров появилась лишь в версии 3.0 в 2018 году.

Третьим трендом, как уже говорилось, стало появление полностью управляемых распараллеленных хранилищ для аналитики на базе SQL. Этот момент хорошо демонстрируется формированием “Современного дата-стека” и выходом в 2016 году инструмента командной строки dbt.

Наконец, четвёртым трендом, повлиявшим на Hadoop, стало развитие глубокого обучения. Во второй половине 2010-х все уже слышали о глубоком обучении и искусственном интеллекте. AlphaGo обозначила знаковый момент, победив Ке Джи, мирового чемпиона по игре в Го, аналогично тому,

как за двадцать лет до этого программа IBM Deep Blue превзошла чемпиона по шахматам, Гарри Каспарова. Этот технологический скачок, который уже творил чудеса и обещал ещё большее – например, автономные автомобили – зачастую был связан с большими данными, поскольку модели ИИ требовали для своего обучения обработки огромных объёмов информации. Однако Hadoop и МО являлись слишком разными и подружить эти технологии было трудно. По факту глубокое обучение указало на потребность в новых подходах для обработки больших данных и подтвердило, что Hadoop не является универсальным инструментом.

Поясню вкратце: учёные по данным, работающие с глубоким обучением, нуждались в двух вещах, которые экосистема Hadoop им предоставить не могла. Во-первых, им нужны были GPU, коих в узлах кластеров Hadoop обычно не было. Во-вторых, учёным требовались последние версии библиотек для глубокого обучения, таких как TensorFlow и Keras, установить которые на весь кластер было проблематично, особенно когда многие пользователи просили разные версии одной библиотеки. Эту конкретную проблему отлично решал Docker, но интеграция этого инструмента в Hadoop требовала времени, а ждать аналитики данных были не готовы. В связи с этим они обычно предпочитали вместо использования кластера запускать одну мощную виртуальную машину с 8 GPU.

Именно поэтому, когда Cloudera в 2017 году сделала своё первое публичное размещение акций (IPO), компания уже занималась разработкой и распространением новейшего продукта, Data Science Workbench. Это решение основывалось уже не на Hadoop или YARN, а на контейнеризации с помощью Docker и Kubernetes, позволяя учёным по данным развёртывать модели в собственной среде в виде контейнеризованного приложения без риска для безопасности и стабильности.

Но для возвращения рыночного успеха этого оказалось недостаточно. В октябре 2018 года Hortonworks и Cloudera претерпели слияние, и остался только бренд Cloudera. В 2019 MapR была куплена Hewlett Packard Enterprise

(HPE), а в октябре 2021 частная инвестиционная фирма CD&R приобрела Cloudera по стоимости акций ниже изначальной.

Хотя угасание Hadoop не означает полную кончину всей экосистемы, поскольку многие крупные компании по-прежнему её используют, особенно для локальных развёртываний. Используют её и многие решения, построенные вокруг этой технологии или её частей. При этом также внедряются инновации, например, Apache Hudi, выпущенный компанией Uber в 2016 году, Apache Iceberg, запущенный Netflix в 2017, и открытый продукт Delta Lake, который разработчики Databricks представили в 2019.Интересно, что одной из основных целей этих новых форматов хранения было обойти следствия первого описанного тренда. Hive и Spark изначально создавались для HDFS, и некоторые элементы быстродействия, гарантированные этой файловой системой, при переходе в облачные хранилища вроде S3 были утрачены, что привело к падению эффективности.

Настоящее время

В настоящее время облачные развёртывания Hadoop уже в основном заменены приложениями Apache Spark или Apache Beam [5] (преимущественно на GCP) в пользу Databricks, Amazon Elastic Map Reduce (EMR), Google Dataproc/Dataflow или Azure Synapse. Я также видел, как многие молодые компании стремятся использовать «Современный дата стек», построенный вокруг хранилищ для аналитики, таких как BigQuery, Databricks-SQL, Athena или Snowflake, сопряжённых с бескодовыми (или малокодовыми) инструментами доставки данных и организованных с помощью dbt, исключая потребность в решениях для распределённых вычислений вроде Spark.

Естественно, компании, которые по-прежнему предпочитают развёртывать свои проекты локально, продолжают использовать Hadoop и другие открытые проекты, такие как Spark и Presto. Но с каждым годом доля переезжающих в облако данных увеличивается, и я не вижу причин для изменения этой тенденции.

По мере развития индустрии данных появлялось всё больше инструментов для управления и каталогизации. В этом отношении стоит упомянуть опенсорсные решения вроде Apache Atlas, выпущенного Hortonworks в 2015 году, Amundsen, открытого компанией Lyft в 2019, и DataHub, который LinkedIn открыла в 2020. Немало в этом сегменте возникло и закрытых стартапов.

Новые компании также строились и вокруг последних технологий реализации планировщика. Здесь можно назвать Prefect, Dagster и Flyte, которые запустили свои открытые репозитории в 2017, 2018 и 2019 годах соответственно и сегодня уже бросают вызов царящему превосходству Airflow.

Наконец, начала формироваться концепция Lakehouse. Lakehouse представляет собой платформу, совмещающую преимущества озера данных и хранилища данных[6]. Это позволяет учёным по данным и бизнес-аналитикам работать в рамках одной платформы, упрощая управление, обмен информацией, а также повышая безопасность. Активнее всех в этом сегменте себя проявила компания Databricks благодаря совместимости Spark как с SQL, так и с DataFrames. За ней последовало предложение Snowpark от Snowflake, затем Azure Synapse от Microsoft и пока самой последней подключилась корпорация Google, запустив BigLake. Если же смотреть в сторону опенсорса, то здесь с 2017 года аналогичную платформу предлагает Apache Dremio.

С самого начала истории Big Data количество открытых проектов и стартапов год за годом лишь продолжало расти (только оцените масштабы этой индустрии в 2021). Я помню, как в районе 2012 года звучали прогнозы, что новые SQL-войны закончатся и объявятся истинные их победители. Пока что этого не произошло, и сложно прогнозировать, как ситуация будет развиваться дальше. Потребуется ещё не один год, чтобы осела вся поднятая пыль. Хотя если всё же пытаться строить догадки, то я бы спрогнозировал следующее:

1. Как уже говорили другие, основные платформы данных (Databricks, Snowflake, BigQuery, Azure Synapse) продолжат совершенствоваться и добавлять новые возможности для восполнения пробелов между друг другом. Я ожидаю увидеть всё бóльшую связность компонентов, в том числе между языками для обработки данных вроде SQL и Python.

2. В течение пары следующих лет может замедлиться появление новых проектов и компаний, хотя причиной тому скорее станет недостаток финансирования после взрыва очередного пузыря доткомов (если такой вообще произойдёт), чем недостаток желания или идей.

3. С самого начала наиболее дефицитным ресурсом являлись квалифицированные кадры. Это означает, что для большинства компаний7 проще вложить дополнительные средства в решение проблем производительности или перейти на более рентабельные продукты, чем тратить лишнее время на оптимизацию. Особенно это актуально теперь, когда стоимость услуг распределённых хранилищ стала столь низкой. Но, возможно, в какой-то момент вендорам станет сложно продолжать ценовое соперничество путём демпинга, и цены пойдут вверх. Хотя даже в этом случае объём сохраняемых бизнесом данных продолжает год за годом расти, а с ним и сопутствующие финансовые потери из-за неэффективности. Быть может, однажды возникнет тенденция, по которой люди начнут искать новые, более дешёвые, опенсорсные альтернативы, что приведёт к возрождению нового витка Hadoop-технологий.

4. В долгосрочной же перспективе, на мой взгляд, реальными победителями окажутся облачные провайдеры Google, Amazon и Microsoft. Для этого им достаточно просто подождать и оценить, в каком направлении дует ветер, после чего в удачный момент приобрести (или просто воссоздать) наиболее оптимальные технологии. Каждый инструмент, интегрируемый в их облако, существенно упрощает жизнь пользователей, особенно когда дело касается безопасности, руководства, контроля доступа и управления

расходами. И при условии отсутствия в процессе серьёзных организационных ошибок я не вижу для этих компаний реальных конкурентов.

1.2 Потоковые данные

Под потоковыми данными понимаются непрерывно поступающие и быстро изменяющиеся информационные потоки, генерируемые различными источниками, такими как сенсоры, логи приложений, социальные сети, интернет вещей и многие другие. Эти данные часто характеризуются высокой скоростью поступления, большим объемом и коротким временем жизни.

Важность потоковых данных заключается в их способности предоставлять актуальную информацию в режиме реального времени. Они позволяют нам вовремя реагировать на события, принимать обоснованные решения и адаптироваться к меняющимся условиям. Эта актуальность имеет особенное значение во многих областях, начиная от бизнеса и финансов и заканчивая медициной и научными исследованиями.

С каждым днем интерес к обработке потоковых данных становится все более заметным. зовами, связанными с обработкой данных в высокоскоростных потоках.

Основные принципы потоковой обработки данных

Потоковая обработка данных — это метод обработки информации, при котором данные анализируются по мере их непрерывного поступления, обеспечивая оперативное реагирование и принятие решений. В сравнении с традиционной пакетной обработкой, где данные группируются и обрабатываются в определенные интервалы времени (пакеты), потоковая обработка работает в режиме реального времени, что позволяет извлекать ценную информацию из потока данных немедленно.

Принцип работы потоковой обработки данных:

1. Захват данных: Процесс начинается с захвата данных из источников. Это могут быть сенсоры, приложения, веб-сервисы и другие

источники, генерирующие потоки данных. Важно обеспечить надежный и эффективный сбор данных для последующей обработки.

2. Агрегация и преобразование: По мере поступления, данные агрегируются и преобразуются для более удобной обработки. Это может включать в себя фильтрацию, преобразование формата, агрегацию и другие операции, необходимые для получения информации в желаемом виде.

3. Обработка: Затем данные направляются в процесс обработки. Здесь применяются алгоритмы и логика, которые анализируют данные, выявляют закономерности, аналитические тренды и аномалии. Это может быть что-то такое, как выявление атак в сети, мониторинг состояния оборудования или анализ поведения пользователей.

4. Принятие решений: После обработки данных система может принимать решения на основе полученных результатов. Это может быть автоматическое действие, например, автоматическая коррекция производственного процесса при выявлении несоответствий, или предоставление информации человеку для принятия решения, например, в случае анализа финансовых данных.

5. Действие и хранение: В зависимости от результата обработки, система может выполнять дополнительные действия. Это может быть отправка уведомлений, запись данных в хранилище, архивирование информации или внесение изменений в рабочие процессы.

Пример реального сценария:

Представьте систему мониторинга транспорта для городского управления. Сенсоры на автомобилях непрерывно передают информацию о скорости, местоположении, состоянии транспортных потоков и дорог. Эти данные немедленно передаются в систему потоковой обработки.

На этапе обработки система анализирует данные, выявляет заторы движения, определяет области повышенного трафика и выявляет аномалии в движении. На основе этих данных система автоматически рассчитывает

оптимальные маршруты, предупреждает об аварийных ситуациях и предоставляет водителям рекомендации для избежания заторов.

Таким образом, потоковая обработка данных в данном сценарии позволяет городским органам управления реагировать на транспортные ситуации мгновенно, оптимизировать движение и повышать безопасность дорожного движения.

Преимущества и недостатки по сравнению с пакетной обработкой

Преимущества потоковой обработки данных:

1. Реальное время: Одним из наиболее значимых преимуществ потоковой обработки является способность анализировать данные практически в режиме реального времени. Это особенно важно для сценариев, требующих мгновенной реакции на события, таких как финансовые операции или мониторинг состояния оборудования.

2. Актуальность: Потоковая обработка позволяет получать актуальную информацию о состоянии системы или событиях, происходящих в окружающем мире. Это важно для принятия оперативных решений и реагирования на изменения в реальном времени.

3. Эффективность ресурсов: Обработка данных по мере их поступления позволяет избежать накопления больших объемов данных и ресурсоемких операций в памяти. Это способствует более эффективному использованию вычислительных ресурсов.

4. Снижение задержек: За счет анализа данных непосредственно при их поступлении можно снизить задержки в реакции на события. Это особенно важно для систем, где даже небольшая задержка может иметь критические последствия.

Недостатки потоковой обработки данных:

1. Сложность обработки: Обработка данных в реальном времени может быть сложной задачей, требующей оптимизации алгоритмов и вычислительных процессов для достижения приемлемой производительности.

2. Управление задержками: В некоторых сценариях задержки между поступлением данных и их обработкой могут быть недопустимо большими. Управление этими задержками требует специализированных методов.

3. Сложность отладки: Отладка и тестирование потоковых систем может быть более сложными, чем в случае пакетной обработки, из-за динамичной природы данных и асинхронности процессов.

2. SPARK. ОСНОВНЫЕ ПАРАДИГМЫ. СТАДИИ ОБРАБОТКИ. ТИПЫ ОПЕРАЦИЙ. API ВЗАИМОДЕЙСТВИЯ

Фреймворк Apache Spark — высокопроизводительная универсальная распределенная система вычислений, самая активная часть проекта с открытым исходным кодом Apache более чем с 1000 участников. Spark обеспечивает возможность обработки больших массивов данных, помимо тех, что могут уместиться на одной машине, с помощью высокоуровневого, относительно простого в использовании API. Spark — одна из самых быстрых систем среди аналогов, его архитектура и интерфейс уникальны. Это единственная система, которая позволяет описывать логику преобразований данных и алгоритмов машинного обучения так, чтобы не зависеть от системы, но сохранить возможность параллельного выполнения. Поэтому данный фреймворк зачастую используют для написания вычислений, которые будут работать быстро в распределенных системах хранения различных видов и размеров.

Однако, несмотря на множество преимуществ Spark и шумиху вокруг него, простейшие реализации многих распространенных стандартных операций науки о данных на Spark будут работать медленнее и менее надежно, чем оптимальные версии. А в силу того, что интересующие нас вычисления касаются обработки данных в огромном масштабе, выгоды от тонкой настройки производительности кода могут быть колоссальными. Производительность — это не только быстрая работа, в подобном масштабе

зачастую речь идет о том, чтобы вообще работать хоть как-то. Можно создать запрос Spark, который не будет выполняться при гигабайтных объемах данных, но после рефакторинга и внесения поправок в структуру конкретных данных и требования кластера прекрасно заработает в той же системе с терабайтами данных. Мы встречались в нашей практике написания промышленного кода Spark с тем, что одни и те же задачи на тех же кластерах работали в сотни раз быстрее после описанных здесь оптимизаций. Говоря языком обработки данных, время — деньги, и мы надеемся, эта книга окупит себя снижением стоимости информационной инфраструктуры и экономией времени разработчиков.

Далеко не все из этих методик применимы для каждого сценария использования. Spark является чрезвычайно гибким и более высокоуровневым, чем другие сравнимые по возможностям вычислительные фреймворки. Именно это позволяет извлечь колоссальную выгоду просто из более точной подгонки под форму и структуру данных. Некоторые из методов будут хорошо работать с определенными объемами данных или даже при определенных распределениях ключей, но не все. Простейший пример: во многих задачах использование функции groupByKey в Spark может с легкостью привести к ужасающим исключительным ситуациям из-за нехватки памяти. Но в отношении данных с незначительным дублированием эта операция будет работать столь же быстро, как и ее альтернативы, с которыми мы вас познакомим. Глубокое понимание конкретного сценария применения и системы, а также взаимодействие с ними фреймворка Spark совершенно необходимы для решения с его помощью наиболее сложных задач науки о данных.

2.1 Apache Spark

Apache Spark - это мощный движок с открытым исходным кодом, построенный для скорости, простого использования, сложной аналитики, в котором есть API на Java, Scala, Python, R и SQL. Spark запускает программы до 100 раз быстрее, чем Hadoop MapReduce в памяти или в 10 раз быстрее на

диске. Его можно использовать для построения приложений данных как библиотеки или выполнения интерактивного специального анализа данных. Spark включает в себя большое количество библиотек для работы с помощью SQL с датафреймами и датасетами, MLlib для машинного обучения, GraphX для работы с графами и Spark Streaming для обработки данных в режиме реального времени. Вы можете комбинировать эти библиотеки в одном приложении. Кроме того, Spark может работать локально, на Hadoop, Apache Mesos, автономно или в облаке. Он может получить доступ к различным источникам данных, включая HDFS, Apache Cassandra, Apache HBase и S3.

Первоначально он был разработан в Калифорнийском университете Беркли в 2009 году. Обратите внимание, что создатель Spark Матей Захария с тех пор стал CTO Databricks и преподавателем Массачусетского технологического института. С момента своего выпуска Spark быстро внедряется организациями в широком спектре индустрии. Интернет-гиганты, такие как Netflix, Yahoo и Tencent стремительно быстро внедрили Spark во внутренние процессы, коллективно обрабатывая несколько петабайт данных в кластерах из более чем 8000 узлов. Быстро возникло сообщество в области больших данных с более чем 1000 участниками разработки кода и более чем 187000 участниками в 420 группах Apache Spark Meetups.

2.2. RDD

Под капотом Apache Spark лежит понятие абстракции данных как распределенного набора объектов. Эта абстракция данных, называется Resilient Distributed Dataset (RDD), позволяет писать скрипты, которые преобразуют эти распределенные наборы данных.

RDD - это неизменяемая распределённая совокупность элементов ваших данных, которые могут храниться в памяти или на диске в кластере машин. Данные распределены между машинами на вашем кластере, которые могут работать параллельно с низкоуровневым API, которые предоставляют возможность трансформации и действия. RDD отказоустойчивы, так как

отслеживают поток данных для автоматического восстановления потерянных данных в случае сбоя.

2.3. DataFrame

Как и RDD, DataFrame - это неизменяемая распределенная коллекция данных. В отличии от RDD, данные организованы в именованные столбцы, как таблица в реляционной базе данных. Разработан для того, чтобы сделать обработку больших данных еще проще, датафрейм позволяет разработчикам придавать структуру распределенному набору данных, обеспечивая абстракцию более высокого уровня. Он предоставляет возможность манипулировать данными с помощью API различных языков и делает Spark доступным для более широкой аудитории, помимо специализированных инженеров данных.

2.4. Dataset

Представленный в Spark 1.6 Spark Dataset предоставляет API, с помощью которого пользователи могут легко выражать преобразования на объектах, а также повышает производительность и преимущества надежного механизма выполнения Spark SQL.

Обратите внимание, что начиная со Spark 2.0 API DataFrame было объединено с API Dataset, что позволило использовать больше подходов в обработке данных. Из-за унификации разработчикам теперь требуется изучать меньшее количество документации, и они работают с одним высокоуровневым и безопасным API называемым Dataset. Концептуально Spark DataFrame - это псевдоним для коллекции универсальных объектов Dataset [Row], где Row - это универсальный нетипизированный объект JVM. Dataset, напротив, представляет собой коллекцию сильно типизированных объектов JVM, заданный вами классом на Scala или Java.

2.5. MLlib

Apache Spark предоставляет базовую библиотеку машинного обучения - MLlib - которая предназначена для простоты, масштабируемости и легкой интеграции с другими инструментами. Благодаря масштабируемости,

языковой совместимости и скорости Spark саентисты могут быстрее решать и воспроизводить свои задачи.

С самого начала проекта Apache Spark MLlib считался основополагающим для успеха Spark. Ключевое преимущество MLlib заключается в том, что она позволяет саентистам сосредоточиться на задачах и моделях вместо того, чтобы решать сложности, связанные с распределенными данными (такими как инфраструктура, конфигурации и так далее). Инженеры данных могут сосредоточиться на разработке распределённых систем с помощью простых в использовании API Spark, в то время как саентисты могут использовать масштаб и скорость ядра Spark. Не менее важно и то, что Spark MLlib - это библиотека общего назначения, предоставляющая алгоритмы для большинства сценариев использования, в то же время позволяя сообществу развивать и расширять ее для специализированных сценариев. Чтобы ознакомиться с ключевыми терминами машинного обучения, пожалуйста, ознакомьтесь с Machine Learning Key Terms, Explained Мэтью Майо.

2.6. ML Pipelines

Обычно запуск алгоритмов машинного обучения включает в себя последовательность задач, включая предварительную обработку данных, извлечение признаков, обучение модели и этапы валидации. Например, классификация текстовых документов может включать в себя сегментацию и очистку текста, извлечение признаков и обучение классификационной модели с кросс-валидацией. Хотя есть много библиотек, которые мы можем использовать для каждого этапа, соединить все этапы воедино не так просто, как может показаться, особенно с огромными наборами данных. Большинство библиотек машинного обучения не предназначено для распределённых вычислений или не обеспечивают возможность создания и поддержки пайплайна.

ML Pipelines - это высокоуровневое API для MLlib, которое находится в пакете «spark.ml». Пайплайн состоит из последовательных этапов.

Существует два основных этапа пайплайна: трансформация и оценивание. Трансформация заключается в сборе данных и выдаче подготовленного набора данных. Например, токенизатор - это трансформатор, который преобразует набор данных с текстом в набор данных с токенизированными словами. Оценивание должно сначала произойти на входном наборе данных, чтобы создать модель, которая является трансформатором, преобразующим входной набор данных. Например, логистическая регрессия - это алгоритм, который обучается на наборе данных с метками и признаками, а затем создает логистическую регрессивную модель.

2.7. GraphX

GraphX является компонентом Apache Spark для графов и параллельных графовых вычислений. На высоком уровне GraphX расширяет Spark RDD с помощью абстракции Graph: ориентированного мультиграфа со свойствами, закрепленными на каждой вершине и ребре. Для обеспечения вычислений графов GraphX предоставляет набор фундаментальных операторов (например, subgraph, joinVerticles и aggregateMessages), а также оптимизированный вариант Pregel API. Кроме того, GraphX включает в себя растущую коллекцию алгоритмов и конструкторов графов для упрощения задач аналитики графов.

2.8. Spark Streaming

Spark Streaming - это расширение ядра Spark Api, которое позволяет инженерам данных и саентистам работать с данными в режиме реального времени из различных источников данных, включая (но не ограничиваясь) Kafka, Flume и Amazon Kinesis. Эти обработанные данные могут быть отправлены в файловые системы, базы данных и дашборды. Его ключевой абстракцией является является дискретный поток, или короче говоря, DStream, который представляет собой поток данных, разделенный на небольшие части. DStream построен на RDD, основной абстракции Spark. Это позволяет Spark Streaming легко интегрироваться с любыми другими компонентами Spark, такими как MLlib и Spark SQL.

2.9. Structured Streaming

Представленный как часть Apache Spark 2.0, Structured Streaming - это высокоуровневая надстройка поверх движка Spark SQL. Это декларативный API, который расширяет DataFrame и DataSet, поддерживающий пакетные, интерактивные и потоковые запросы. Преимущество этого подхода заключается в том, что он позволяет разработчикам применять свой опыт работы со статическими наборами данных (т.е. пакетными) и легко применять его к бесконечным наборам данных (т.е. Потоковой передаче).

2.10. spark-packages.org

spark-packages.org - это своего рода хранилище документаций для сообщества с целью отслеживания растущее число пакетов с открытым исходным кодом и библиотек, которые работают с Apache Spark. Пакеты Spark облегчают пользователям поиск, обсуждение, оценку и установку пакетов для любой версии Spark и позволяют разработчикам легко интегрировать пакеты для своих целей.

Пакеты Spark включают в себя интеграцию с различными источниками данных, инструментами управления, библиотеками более высокого уровня, специфичными доменными библиотеками, алгоритмами машинного обучения, примерами кода и другим контентом Spark. Примеры пакетов включают в себя Spark-CSV (который теперь включен в Spark 2.0) и пакеты интеграции Spark ML, включая GraphFrames и TensorFrames.

2.11. Catalyst Optimizer

Spark SQL является одним из наиболее технически задействованным компонентом Apache Spark. Он поддерживает как SQL-запросы, так и DataFrame API. В основе Spark SQL лежит оптимизатор Catalyst, который использует расширенные функции языка программирования (например, соответсвует шаблонам Scala и квазицитаты), является новым способом создания расширяемого оптимизатора запросов.

Catalyst основан на конструкциях функционального программирования Scala и разработан с учетом этих двух целей:

- Легкость добавления новых методов и функций оптимизации Spark SQL.
- Предоставление возможности внешним разработчикам расширять оптимизатор (например, добавление правил, новых источников данных, поддержка новых типов данных и т.д.)

Кроме того, Catalyst поддерживает оптимизацию как в целях скорости работы, так и в целях использования ресурсов.

3. SPARK. РАБОТА С DATAFRAME

3.1 Отложенные вычисления

В основе множества других систем, хранящих данные в оперативной памяти, лежат «мелкозернистые» обновления изменяемых объектов, то есть обращения к конкретной ячейке таблицы для сохранения промежуточных результатов. Напротив, набор RDD вычисляется полностью отложенным образом. Spark приступает к вычислению секций лишь при вызове действия. Действие — операция Spark, возвращающая что-либо внешней по отношению к Spark (к исполнителям Spark) системе. Примером может послужить возврат данных драйверу (такими операциями, как count или collect) или запись данных во внешнюю систему хранения (например, операцией copyToHadoop). Действия вызывают срабатывание планировщика, который строит ориентированный ациклический граф (directed acyclic graph, DAG) на основе зависимостей между преобразованиями наборов RDD. Другими словами, Spark вычисляет действие в обратном порядке путем описания последовательности шагов, необходимых для формирования каждого из объектов итогового распределенного набора данных (каждой секции). А затем согласно этой последовательности, называемой планом выполнения, планировщик находит недостающие секции для каждого этапа до тех пор, пока не вычислит весь результат.

3.2 Преимущества отложенного вычисления с точки зрения производительности и удобства использования

Отложенное вычисление позволяет Spark объединять операции, не требующие взаимодействия с драйвером (их называют преобразованиями с взаимно однозначными зависимостями), чтобы избежать многократных проходов по данным. Например, пусть Spark-программа вызывает для одного и того же набора RDD функции map и filter. Spark может отправить каждому из исполнителей инструкции по выполнению как map, так и filter. Далее фреймворк может выполнить обе эти функции для каждой из секций, что потребует лишь однократного об - ращения к данным вместо отправки двойного набора инструкций и обращения к каждой секции два раза. Теоретически это должно вдвое снизить вычислительную сложность. Парадигма отложенных вычислений не только более эффективна, но и позволяет легче реализовать в Spark (по сравнению с другими фреймворками, напри - мер MapReduce) логику, требующую от разработчика объединения операций отображения. Разумная стратегия отложенных вычислений фреймворка Spark обеспечивает возможность выражения той же логики намного меньшим количеством строк кода. Для этого достаточно выстроить в цепочки операции с узкими зависимостями, после чего механизм вычисления Spark выполнит всю работу по их объединению.

3.3 Отложенное вычисление и отказоустойчивость

Spark отличается отказоустойчивостью. Это значит, что он не подведет, не потеряет данные и не вернет неправильные результаты даже в случае отказа машины, на которой он работает, или сбоя сети. Уникальность обеспечения отказоустойчивости фреймворком Spark состоит в следующем: каждая секция данных содержит информацию о зависимостях, достаточную для повторного ее вычисления. Большинство парадигм распределенных вычислений, при которых пользователи могут работать с изменяемыми объектами, обеспечивают отказоустойчивость за счет журналирования обновлений или дублирования данных на других машинах. В отличие от них

Spark не требует поддержания журнала обновлений каждого объекта RDD или журнала промежуточных шагов, поскольку сам объект содержит всю информацию о зависимостях, необходимую для репликации всех его секций. Следовательно, при потере секции в RDD есть достаточно информации относительно ее происхождения, чтобы вычислить ее заново, причем этот расчет можно распараллелить для ускорения восстановления.

Последствия отложенного вычисления для отладки очень существенны, ведь оно означает возможность сбоя Spark-программы только в момент выполнения действия. Допустим, вы работали с примером подсчета количества слов, после чего аккумулировали результаты на драйвере. Если переданное значение для стоп-слов было неопределенным (например, из-за того, что оно представляло собой результат выполнения программы на языке Java), конечно, произойдет сбой обработки кода с исключением «указатель, содержащий неопределенное значение» в функции проверки contains. Однако подобный сбой проявится, только когда программа начнет выполнять этап сбора результатов. Даже отслеживание стека будет показывать: сбой произошел на этапе сбора, наводя на мысль о том, что источник его — в операторе сбора. Поэтому лучше всего вести разработку в среде, предоставляющей доступ к полной отладочной информации.

Из-за отложенного вычисления отслеживание вызовов сбойных заданий Spark в стеке (особенно при внедрении в большие системы) будет зачастую неизменно указывать на сбой в момент действия, даже если проблема в логике возникла в преобразовании, происходящем намного раньше по ходу программы.

3.4 Хранение данных в памяти и управление памятью

Превосходство Spark над MapReduce в производительности особенно ощутимо в сценариях использования, включающих повторяемые вычисления. В значительной степени фреймворк обязан этим повышением производительности хранению данных в оперативной памяти. Вместо записи данных на диск в промежутке между проходами Spark имеет возможность

хранить данные в исполнителях, загруженных в оперативную память. При этом данные каждой секции доступны в оперативной памяти всякий раз, когда они нужны.

Spark предлагает три варианта управления памятью: в оперативной памяти в виде сериализованных данных, в оперативной памяти в виде десериализованных данных и на жестком диске. У каждого из них есть свои преимущества относительно времени выполнения и занимаемого пространства.

- В оперативной памяти в виде десериализованных Java-объектов. Наиболее очевидный способ хранения объектов в наборе RDD — в виде исходных десериализованных Java-объектов, определяемых драйверной программой. Это самая быстродействующая форма хранения данных в оперативной памяти, поскольку экономит требуемое на сериализацию время. Однако она, возможно, не самая выгодная относительно используемой памяти, так как данные приходится хранить в виде объектов.
- В виде сериализованных данных. При перемещении по сети объекты Spark преобразуются в потоки байтов с помощью стандартной библиотеки сериализации языка Java. Это, вероятно, более медленно работающий подход, поскольку чтение сериализованных данных требует от CPU большего объема действий, чем чтение десериализованных. Однако это зачастую выгоднее в смысле используемой памяти, поскольку пользователь получает возможность выбирать наиболее эффективное представление. В то время как Java-сериализация более эффективна, чем использование целых объектов, Kryo-сериализация, возможно, еще эффективнее в смысле расхода памяти.
- На диске. Наборы RDD, секции которых слишком велики для хранения в оперативной памяти каждого из исполнителей, можно записать на диск. Это явно более медленный метод в случае повторяемых вычислений, но, вероятно, и более отказоустойчивый при длинных последовательностях преобразований и, наверное, единственный разумный вариант при очень

больших объемах вычислений. Функция persist() классов RDD позволяет пользователю контролировать метод хранения набора RDD. По умолчанию persist() сохраняет набор RDD в виде десериализованных объектов в оперативной памяти, но пользователь может управлять способом хранения RDD путем передачи в persist() в виде параметра одной из множества опций хранения. Мы рассмотрим различные варианты повторного использования набора RDD в подразделе «Виды повторного использования: кэш, сохранение, контрольная точка, перетасовочные файлы». При сохранении наборов RDD их реализация по умолчанию вытесняет наиболее давно применяемую секцию (так называемое LRU-кэширование), если место, которое она занимает, необходимо для вычисления или кэширования новой секции. Однако это поведение можно изменить, управляя назначением приоритетов с помощью функции persistencePriority() классов RDD

3.5 API DataFrame

API DataFrame модуля Spark SQL позволяет работать с наборами DataFrame, не прибегая к регистрации временных таблиц или генерации SQL-выражений. API DataFrame включает как преобразования, так и действия. Выполняемые над наборами DataFrame преобразования по своей природе более реляционные, в то время как API Dataset (рассматриваемый далее) скорее функциональный.

Преобразования

Выполняемые над наборами DataFrame преобразования аналогичны по своему характеру преобразованиям RDD, но с более реляционным оттенком. Вместо произвольных функций, недоступных для анализа оптимизатора, используется ограниченный синтаксис выражений, благодаря которому оптимизатор получает больше информации. Как и в случае с наборами RDD, у нас есть широкие возможности разбиения преобразований на простые с одним набором DataFrame, несколькими наборами DataFrame, данными типа «ключ — значение», а также оконные/группирующие преобразования.

Преобразования Spark SQL являются отложенными только частично, схема вычисляется немедленно.

Простые преобразования наборов DataFrame и SQL выражения

Простые преобразования наборов DataFrame позволяют делать практически все, для чего достаточно построчной обработки данных1. С их помощью можно совершать большинство выполняемых над наборами RDD операций, за исключением использования выражения Spark SQL вместо произвольных функций. Для иллюстрации этого мы начнем с рассмотрения различных видов операций фильтрации, доступных для наборов DataFrame.

Функции DataFrame, такие как filter, принимают на входе выражения Spark SQL вместо лямбда-выражений. Благодаря им оптимизатор понимает, что представляет собой условие, и в случае filter часто может избежать чтения ненужных записей.

API DataFrame модуля Spark SQL содержит огромное множество доступных операторов. Можно использовать все стандартные математические операторы с плавающей точкой, а также стандартные логические и побитовые операторы (в начале названия которых стоит bitwise, чтобы отличать их от логических). При операциях над столбцами для равенства/неравенства применяются операторы === и !== с целью избежать конфликта с внутренними операторами Scala. Для столбцов строк доступны функции startsWith/endsWith, substr, like и isNull. Полный список операций приведен на http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column

Помимо простой фильтрации данных, можно также создать объект DataFrame с новыми столбцами или обновленными значениями в старых. Фреймворк Spark задействует для этого синтаксис выражений, который мы обсуждали для filter, за исключением того, что вместо необходимости применять условие (например, проверку на равенство) результаты используются в качестве значений нового объекта DataFrame.

Специализированные преобразования DataFrame для отсутствующих и зашумленных данных

Модуль Spark SQL также предоставляет специальные инструменты для работы с отсутствующими, неопределенными и некорректными данными. Благодаря использованию функций isNan и isNull совместно с фильтрами можно формировать условия для строк, которые надо оставить. Например, при наличии некоего количества различных столбцов, вероятно, с различными уровнями точности (причем часть из них могут быть неопределенными) можно задействовать coalesce(c1, c2, ..), которая вернет первый, не равный null столбец. Аналогично для числовых данных nanvl возвращает первое не равное NaN значение (например, nanvl(0/0, sqrt(2), 3) возвращает 3). Чтобы упростить взаимодействие с отсутствующими данными, функция na для объектов DataFrame позволяет обращаться к некоторым утилитам из класса DataFrameNaFunctions (http://spark.apache.org/docs/latest/api/scala/index.html# org.apache.spark.sql.DataFrameNaFunctions), часто используемым для работы с отсутствующими данными.

Не только построчные преобразования

Иногда принятия решения построчно, доступного с помощью filter, недостаточно. Модуль Spark SQL позволяет сделать выборку неповторяющихся строк путем вызова метода dropDuplicates, но, как и в случае с аналогичной операцией над наборами RDD (distinct), это требует перетасовки, поэтому зачастую выполняется намного медленнее, чем filter. В отличие от RDD, dropDuplicates может дополнительно отбрасывать строки на основе лишь подмножества столбцов, таких как поля ID.

Агрегирующие функции и функция groupBy

В модуле Spark SQL имеется множество агрегирующих функций, обладающих широкими возможностями, а его оптимизатор позволяет легко объединять несколько таких функций в одно действие/один запрос. Так же как в объектах DataFrame библиотеки Pandas, функция groupBy возвращает

специальные объекты, над которыми можно осуществлять отдельные виды агрегирования. В версиях фреймворка Spark, предшествующих 2.0, для этого использовался универсальный класс GroupedData (http://spark.apache.org/docs/1.6.2/api/scala/index.html#org.apache.spark.sql.GroupedData), но в версии 2.0 и более поздних преобразование groupBy для наборов DataFrame такое же, как и для Dataset

Возможности агрегирующих функций для наборов Dataset расширены, они воз- вращают GroupedDataset (в предшествующих 2.0 версиях фреймворка Spark) или KeyValueGroupedDataset при группировке с произвольной функци- ей и RelationalGroupedDataset при группировке с реляционным/Dataset DSL-выражением.

Агрегирующие функции min, max, avg и sum реализованы как удобные методы непосредственно класса GroupedData, причем можно описать и другие путем передачи выражений методу agg.

Для вычисления нескольких различных сводных показателей или более сложных сводных показателей следует задействовать API agg класса GroupedData вместо непосредственного вызова методов count, mean и тому подобных функций. При использовании API agg необходимо указать или список выражений агрегирования, описывающих сводные показатели строки, или карту соответствий имен столбцов именам агрегирующих функций. После вызова agg с требуемыми сводными показателями будет возвращен обычный набор DataFrame с результатами агрегирования.

Оконные функции

В Spark SQL 1.4.0 появились оконные функции, упрощающие работу с диапазонами или окнами строк. При создании окна необходимо указать, на какие строки оно распространяется, порядок строк в пределах каждой секции/группы и размер окна (допустим, K строк до и J строк после ИЛИ-диапазона между значениями).

Оконные функции очень удобны для вычисления, скажем, средней скорости в случае зашумленных данных, относительных продаж и т. п.

4. РАБОТА С ДАННЫМИ ТИПА «КЛЮЧ — ЗНАЧЕНИЕ» (RDD)

RDD (Resilient Distributed Dataset) – это простая, неизменяемая, распределенная коллекция объектов во фреймворке Apache Spark. RDD представляет собой распределенный набор данных, который делится на множество частей, обрабатывающихся различными узлами в кластере. Наборы РДД могут содержать объекты с любыми типами данных на языках Python, JAVA или Scala

Как устроен RDD: свойства и структура

RDD – это разновидность датасета (простого набора данных), который разделен на множество машин, работающих в кластере.

RDD имеет следующие свойства:

1. Неизменяемость и секционирование – РДД состоит из набора записей, которые делятся на разделы. Раздел – это единица параллелизма в РДД. Каждый раздел является логическим подразделением данных, которое является неизменяемым (immutable) и хранится на отдельном узле в кластере.

2. Применение общих операций (coarse-grained operations), которые способны манипулировать всеми данными одновременно (например, фильтр или группировка).

3. Отказоустойчивость: все преобразования над наборами РДД ведутся в распределенной среде с поддержкой репликации (копированием данных между узлами), и каждое преобразование регистрируется каждым отдельным узлом в кластере. Следовательно, при выходе из строя одного узла, данные можно будет восстановить с помощью любого другого рабочего узла.

4. Ленивые вычисления: Apache Spark проводит необходимые преобразования над РДД только один раз в момент их создания. Это значительно сокращает общее время выполнения всех операций и ускоряет работу над данными.

5. Сохраняемость: пользователи могут выбирать удобный для себя формат хранения РДД (например, в памяти или на диске в файле).

RDD можно создавать вручную, а можно загружать из внешних источников. Источниками хранения РДД могут служить следующие источники:

- текстовый файл;
- CSV-файл;
- файл со структурой JSON-документа;
- база данных (через драйвер JDBC)

Какие операции с RDD существуют в Spark: основные виды

Существует масса методов для работы с распределенными наборами данных (RDD, Resilient Distributed Dataset): фильтрация, удаление дубликатов, случайная выборка элементов, применение функций к каждому элементу и пр. Все эти операции над распределенными наборами данных в Spark можно отнести к одному из следующих двух видов:

- действия;
- преобразования.

Действия

Действия – это такой тип операций с RDD, который возвращает конкретное значение. Действия применяются в том случае, когда необходимо вывести конкретное значение в консоль. Самыми известными действиями служат методы `collect()`, `take(n)` и `count()`. Метод `collect()` применяется, когда необходимо получить элементы датасета в виде массива или списка.

Важно отметить, что в момент вызова нового действия происходит вычисление всего набора RDD с самого начала. Это может занимать огромное количество времени и ресурсов, поэтому методами действий следует пользоваться очень осторожно.

Преобразования

Преобразования – это операции над коллекциями данных RDD, результатом которых служат новые RDD. Вычисление преобразованных

RDD откладывается до того момента, когда к ним будут применены действия. Преобразования в основном выполняются поэлементно, то есть ведутся вычисления над каждым элементом в датасете по отдельности. Основными преобразованиями считаются методы `map()`, `filter()`, `distinct()`.

Метод `map()` принимает в качестве параметра функцию и применяет ее к каждому элементу RDD, а затем получает новый набор.

Иногда нам нужно получить набор, содержащий определенные элементы, которые соответствуют определенному условию. В этом нам поможет метод `filter()`.

Порой мы получаем на обработку огромную коллекцию с данными. Работать с такими данными бывает очень проблематично, поскольку это требует большого количества времени и ресурсов. Причиной такого размера могут быть дубликаты, которые затмили наши данные. Чтобы их удалить, можно использовать очень простой метод `distinct()`

Преобразования могут выполняться также с двумя наборами данных одновременно, например, в случае объединения или пересечения датасетов, Декартова произведения и вычитания.

Бывают случаи, когда необходимо комбинировать преобразования и действия. Например, следующий код показывает, как вычислить квадраты чисел коллекции RDD и вывести их на экран

Перетасовка (Shuffle)

Итак, мы разобрались в том, что у нас есть N блоков данных и P потоков (работников), которые эти блоки данных могут перерабатывать параллельно.

И все у нас было бы хорошо, если бы эти блоки жили до конца приложения, но почти в любом приложении будет обработка, которая влечет за собой полную перетасовку наших блоков. Это, например, соединение двух таблиц по ключу (JOIN), группировка по ключу (GROUP BY). В этом случае работает всем хорошо известный паттерн MapReduce, при котором происходит перераспределение данных всего набора по ключу на новые

блоки данных, так, чтобы строки с одним и тем же ключом находились только в одном блоке. Этот процесс в Spark называется Shuffle. Почему я написал его с большой буквы? Потому что это очень сложный и дорогостоящий процесс, при котором увеличивается потребление памяти на исполнителях, потребление дисковой памяти на узлах кластера и сетевой обмен между узлами кластера. Очень напоминает превращение гусеницы в бабочку – все разваливается на куски, чтобы потом собраться в новом обличии, и так же энергозатратно.

Задание делится на этапы

В Spark обработка блоков от одной перетасовки (Shuffle) до другой называется этапом (Stage). Заметим, что до перетасовки все блоки обрабатываются параллельно, после перетасовки они тоже обрабатываются параллельно, но новый этап не начнется пока этот процесс не пройдут все блоки в конце предыдущего этапа. Таким образом, граница между этапами – это место ожидания при параллельной обработке блоков. Заметим также, что в рамках одного этапа все задачи (task) над одним блоком происходят последовательно в рамках одного потока. То есть блок никуда не передается по сети, но все блоки обрабатываются параллельно. Получается, что количество блоков в границах этапа неизменно.

Мы пришли к следующей картине: все задания делятся на этапы, а в рамках каждого этапа количество блоков неизменно и равно …. И вот здесь начинается самое интересное. Количество работников у нас известно (P = executors*cores), а вот сколько блоков будет на каждом этапе – это вопрос, от которого напрямую зависит производительность нашего приложения. Ведь если блоков много, а исполнителей мало, то каждый исполнитель будет обрабатывать последовательно несколько блоков и наоборот: если блоков мало, а исполнителей больше, то некоторые исполнители будут простаивать, когда остальные трудятся не покладая рук. Самое интересное здесь – то, что, когда приложение работает медленно, пытаются выдать ему больше исполнителей, но производительность в этом случае не увеличивается.

Давайте начнем выяснять объем работы по этапам. Здесь и далее для простоты картины будем рассматривать блоки только одного набора данных. В каждый момент времени исполнители могут обрабатывать несколько несвязанных друг с другом этапов. Например, перед JOIN два набора данных будут обрабатываться независимо друг от друга и, таким образом, делить исполнителей между собой. В этом случае количество блоков обработки будет их суммой. Но для наших целей необходимо понять – что происходит с одним набором данных. На первом этапе все будет зависеть от того, откуда произошел ваш набор данных. Например, если вы читаете директорию файлов parquet из HDFS, то количество блоков на первом этапе в общем случае будет равно (количеству блоков HDFS, из которых состоят все файлы .parquet из загружаемой директории). То есть в этом случае каждый блок HDFS будет представлять собой отдельный блок данных для обработки. Но не забываем, что такое распределение блоков будет сохраняться до конца этапа.

5. SPARK. ОБРАБОТКА ДАННЫХ И ML

У фреймворка Spark есть две библиотеки машинного обучения: Spark MLlib и Spark ML — с совершенно разными API, но очень похожими алгоритмами. Эти библиотеки унаследовали многие из относящихся к производительности нюансы API наборов RDD и Dataset, на которых они основаны, но есть у них и свои особенности. MLlib — исторически первая из этих библиотек, перешедшая в фазу лишь поддержки/исправления ошибок. При обычных обстоятельствах мы бы на ней не останавливались, а сосредоточились на новом API, однако не все возможности существующих алгоритмов были перенесены в новый API Spark ML. Spark ML — более новая (вдохновленная примером scikit-learn) библиотека, находящаяся в фазе активной разработки.

Выбор между библиотеками Spark MLlib и Spark ML

На первый взгляд, самое очевидное различие между библиотеками MLlib и ML — используемые типы данных: первая поддерживает наборы RDD, а вторая — наборы DataFrame и Dataset. Различие в форматах данных не столь существенно, ведь обе библиотеки работают с наборами RDD/Dataset векторов, которые легко можно преобразовывать из формата RDD в Dataset и наоборот.

С точки зрения подхода к архитектуре основная цель Spark MLlib — предоставить базовый набор алгоритмов, оставляя пользователю решение львиной доли задач конвейеризации данных, их очистки, подготовки и выбора признаков. Spark ML же (вдохновленная примером библиотеки scikit-learn) старается предоставить пользователю API для всего, начиная от подготовки данных и заканчивая обучением модели.

В настоящее время единственный доступный вариант при необходимости вы - полнить потоковую обработку или динамическое обучение (online training) — использовать API библиотеки MLlib. Ряд алгоритмов из библиотеки Spark MLlib поддерживают обучение на потоковых данных с применением API DStream пакета Spark Streaming

Выбирать между MLlib и ML желательно с прицелом на будущее. В библиотеке Spark ML и далее станут появляться новые возможности, которые не будут переноситься в библиотеку MLlib, находящуюся на этапе только исправления ошибок.

Интегрированный API конвейеров библиотеки Spark ML упрощает реализацию метаалгоритмов, таких как поиск по значениям параметров по различным компонентам. Оба API поддерживают алгоритмы регрессии, классификации и кластеризации. Если вы еще не выбрали библиотеку для вашего проекта, то имеет смысл по умолчанию выбирать Spark ML, ведь это основная активно развивающаяся библиотека машинного обучения для Spark.

Работаем с библиотекой MLlib

Многие факторы, касающиеся производительности при работе с Spark Core, применимы также и к MLlib. Один из самых очевидных — повторное использование наборов RDD, ведь во многих алгоритмах машинного обучения применяются итеративные вычисления или оптимизации, поэтому очень важно сохранять входные данные в нужных точках.

Алгоритмы обучения с учителем в API Spark MLlib ориентируются на наборы RDD маркированных точек, а алгоритмы без учителя используют наборы RDD векторов. Эти маркированные точки и векторы являются уникальными для библиотеки MLlib, они отличаются от классов векторов языка Scala и эквивалентных классов библиотеки Spark ML

Для выбора признаков и масштабирования необходимо нахождение данных во внутреннем формате Spark, так что, прежде чем говорить об этом, взглянем на кодирование данных в требуемый формат.

После кодирования во многих конвейерах данных машинного обучения про - исходит фильтрация данных, чтобы исключить некорректные или искаженные записи, которые могут привести к проблемам в любой итоговой модели. Пакет MLlib фреймворка Spark предполагает выполнение фильтрации с помощью преобразований наборов RDD.

После выполнения первоначальной фильтрации можно воспользоваться удобными инструментами для выбора признаков и масштабирования, предоставляемыми библиотекой MLlib. Эти преобразователи признаков подходят для выбора и кодирования признаков и масштабирования.

Работа с векторами фреймворка Spark

Внутренний формат вектора Spark отличается от аналогичного формата языка Scala, и существуют отдельные векторные библиотеки для MLlib и ML. Вместо непосредственного формирования векторов фреймворк предоставляет объект-фабрику org.apache.spark.mllib.linalg.Vector, способную формировать как плотные, так и разреженные векторы. При наличии массива

признаков можно непосредственно сформировать плотный вектор Spark с помощью метода Vector.dense

Если вы хотите представить имеющийся у вас плотный вектор в виде разреженного, то можно вызвать для него метод toSparse или непосредственно создать разреженный вектор, передав последовательность ненулевых кортежей (индекс, значение) в метод Vector.sparse.

Подготовка текстовых данных

Не все признаки можно кодировать непосредственно, в случае текстовых данных необходимо преобразовать их в числовой формат. Для кодирования таких данных существует утилита Word2Vec и класс HashingTF. Последний реализует одну из простейших операций над признаками, не требующую никакого обучения и непосредственно применимую к данным. HashingTF оперирует с наборами RDD из Iterable[String], поэтому вы можете разбить свои данные на лексемы любым удобным образом.

Несмотря на простоту вышеописанного подхода, при нем возвращается объект типа SparkVector и отбрасывается все, кроме результата преобразования HashingTF. Использовать это напрямую нежелательно, ведь, скорее всего, нужно будет со - хранить смесь признаков и информации о метках. Вместо того чтобы применять функцию transform непосредственно к наборам RDD во время подготовки данных, можно задействовать ее для кодирования строковых записей в специальном ассоциативном массиве

Для некоторых из преобразователей признаков, скажем Word2Vec, необходимо обучение, подобно тому, как происходит в «традиционных» моделях машинного обучения. Основное отличие в том, что их результаты, скорее всего, нет смысла использовать непосредственно. Для обучения модели Word2Vec нужно разбить входные данные на лексемы, после чего просто вызвать метод fit экземпляра Word2Vec, который вернет объект Word2VecModel.

Подготовка данных для машинного обучения с учителем

Перед использованием алгоритмов для маркированных данных сначала нужно создать объект LabeledPoint с метками и вектор признаков. Для LabeledPoint необходимо, чтобы метки были числами типа double аналогично тому, что элементы вектора тоже должны быть такими числами. Как и с кодированием признаков, если метки числовые, то преобразовать типы будет несложно, однако для других типов придется задействовать пользовательскую функцию или аналогичную методику (допустим, StringIndexer).

Обучение моделей библиотеки MLlib

После выбора и масштабирования признаков наступает время обучения модели. У большинства алгоритмов библиотеки MLlib есть метод run, который принимает на входе набор RDD объектов LabeledPoint (алгоритмы машинного обучения с учителем) или объектов Vector (алгоритмы машинного обучения без учителя) и возвращает модель.

У каждого из алгоритмов есть свои параметры, способные существенно влиять на производительность, так что не жалейте времени на просмотр документации по используемому алгоритму. Несмотря на множество преимуществ настройки, большинство алгоритмов будет работать без каких-либо особых установок.

Предсказание

Теперь, когда уже есть модель, время задействовать ее для предсказания значений. Простейший способ — использовать в том же кластере Spark, где она обучалась, так как при этом модель не нужно будет экспортировать, а потом загружать. Однако во многих сценариях применения требуется меньшая задержка, чем может обеспечить механизм пакетной обработки Spark. Обычно не имеет смысла использовать кластер Spark для динамического предсказания, разве что в вашем случае микропакетный подход работает. Spark обеспечивает ограниченную поддержку предсказания по одной записи за раз, но для этого все равно нужен локальный кластер.

У большинства моделей MLlib есть функции predict, работающие с наборами RDD объектов типа SparkVector, а у некоторых — отдельная функция predict для обработки по одному вектору за раз. Это справедливо не для всех моделей; так, у LDAModel вместо функции predict есть метод topicDistribution, поэтому обязательно загляните в документацию API по используемой модели.

Выдача и сохранение

Сохранение модели и выдача тесно связаны, так как при многих схемах развертывания приходится задействовать для выдачи результатов набор машин, отличный от применяемого для обучения. Даже если вы используете модель только для пакетного предсказания по записям в заданиях Spark, все равно вы, вероятно, захотите сохранить модель, чтобы обратиться к ней в различных заданиях Spark.

Библиотека MLlib поддерживает экспорт в два формата: внутренний формат Spark и PMML (Predictive Model Markup Language, язык разметки для прогнозного моделирования (https://en.wikipedia.org/wiki/Predictive_Model_Markup_Language)). В библиотеке MLlib сохранение модели реализовано с помощью типажей Saveable и PMMLExportable. Первый предоставляет экспорт во внутренний формат Spark, легко читаемый в Spark и реализованный для множества моделей. Зачастую Saveable также оказывается более экономным в смысле расходуемого места. Объем выводимых данных при использовании PMMLExportable больше, чем при Saveable, и, как ни странно, модели, экспортированные с помощью экспорта PMML, нельзя загрузить обратно в Spark. Однако преимуществом экспорта PMML является возможность чтения во внешних системах.

6. ОЧЕРЕДИ И БРОКЕРЫ

Брокер сообщений представляет собой тип построения архитектуры, при котором элементы системы «общаются» друг с другом с помощью

посредника. Благодаря его работе происходит снятие нагрузки с веб-сервисов, так как им не приходится заниматься пересылкой сообщений: всю сопутствующую этому процессу работу он берёт на себя.

Можно сказать, что в работе любого брокера сообщений используются две основные сущности: producer (издатель сообщений) и consumer (потребитель/подписчик).

Одна сущность занимается созданием сообщений и отправкой их другой сущности-потребителю. В процессе отправки есть ещё серединная точка, которая представляет собой папку файловой системы, где хранятся сообщения, полученные от продюсера.

Здесь возможны несколько вариантов:

- Сообщение отправляется напрямую от отправителя к получателю
- Схема публикации/подписки.

В рамках этой схемы обмена сообщениями отправитель не знает своих получателей и просто публикует сообщения в определённую тему. Потребители, которые подписаны на эту тему, получают сообщение. Далее на базе этой системы может быть построена работа с распределением задач между подписчиками. То есть выстроена логика работы, когда в одну и ту же тему публикуются сообщения для разных потребителей. Каждый «видит» уникальный маркер своего сообщения и забирает его для исполнения.

Для чего нужны брокеры сообщений?

- Для организации связи между отдельными службами, даже если какая-то из них не работает в данный момент. То есть продюсер может отправлять сообщения, несмотря на то, проявляет ли активность потребитель в настоящее время.
- За счёт асинхронной обработки задач можно увеличить производительность системы в целом.
- Для обеспечения надёжности доставки сообщений: как правило, брокеры обеспечивают механизмы многократной отправки сообщений в тот

же момент или через определённое время. Кроме того, обеспечивается соответствующая маршрутизация сообщений, которые не были доставлены.

Недостатки брокеров сообщений:

- Усложнение системы в целом как таковой, так как в ней появляется ещё один элемент. Кроме того, возникает зависимость от надёжности распределённой сети, а также потенциальная возможность возникновения проблем из-за потребности в непротиворечивости данных, так как некоторые элементы системы могут обладать неактуальными данными.
- Из-за асинхронной работы всей системы, а также её распределённого характера могут возникать ошибки, выяснение сути которых может стать непростой задачей.
- Освоение подобных систем является не самым простым вопросом и может занять существенное время.

Когда брокеры сообщений могут быть полезны:

- Если в рамках вашей системы есть действия, которые требуют для своего выполнения много времени и потребляют много ресурсов, при этом они не требуют немедленного результата.
- Микросервисы: если ваша система достаточно сложна и состоит из отдельных сервисов, то для их координации можно использовать брокер сообщений, который в этом случае будет выступать в роли как бы центрального роутера. Каждый сервис подписывается только на свой тип сообщений, выстраивается определённая логика их обработки.
- Мобильные приложения: здесь возможен вариант с задействованием push-уведомлений, когда множество смартфонов с установленным приложением подписаны на определённую тему. Если в ней публикуется какая-либо новость, то подписанный смартфон выводит уведомление.
- Транзакционные системы: если какое-то действие системы состоит из множества отдельных этапов, каждый из которых выполняется отдельным элементом системы, то в этом случае брокер сообщений может

выступить в роли своеобразной «доски уведомлений». Каждый сервис отписывается после того, как его этап общей задачи был выполнен. После этого в работу вступает следующий сервис, обрабатывающий, соответственно, следующий этап общей задачи.

Какие есть брокеры сообщений?

Следует сразу оговориться, что брокеров сообщений существует великое множество, и приведённый по ссылке список прямо говорит об этом.

Каждый из них обладает определёнными преимуществами и хорошо решает возложенные на него задачи. Например, если одни предназначены для создания инфраструктуры связи между распределёнными частями приложения, другие предназначены для достаточно специфических задач, например «интернета вещей», и функционируют на основе легковесного протокола MQTT. Подобные брокеры служат для сбора статистики, температуры и других показателей с распределённых датчиков, установленных на определённых машинах, элементах конструкций, географически разных точках территории.

Малое время задержки для прохода сообщения по сети, а также возможность двунаправленной связи в реальном времени (так как MQTT является надстройкой над дуплексным протоколом websockets) позволяют реализовывать достаточно интересные применения. Среди них может быть даже управление робототехническими устройствами в реальном времени. При таком управлении задержка не превышает десятков миллисекунд, что вполне допустимо для низкоскоростных устройств. Например, для управления роботами, движущимися в промышленных цехах и использующимися для перевозки деталей от станков или сырья к производственным станкам.

Также информационные системы на базе протокола MQTT используются в автомобильной сфере и в ряде других промышленных областей (HiveMQ).

Apache Kafka и RabbitMQ

В корпоративных инфраструктурных системах популярностью пользуются Apache Kafka и RabbitMQ. В связи с чем часто возникает вопрос, какую из них выбрать в конкретном случае?

Говоря о RabbitMQ, можно сказать, что он представляет собой классический брокер, в котором присутствуют две описанные выше сущности – продюсер (система, генерирующая сообщения о разнообразных событиях) и подписчик, являющийся получателем этих сообщений.

Обе эти сущности в процессе работы взаимодействуют с очередью сообщений, которая представляет собой хранилище, где накапливаются отправляемые сообщения.

Система устроена таким образом, что поддерживает обоюдное уведомление об успешности доставки с двух сторон: после того как продюсером было отправлено целевое сообщение и оно получено, система отправляет продюсеру уведомление об успешном приёме. В свою очередь потребитель, если сообщение им успешно получено, также отправляет уведомление в систему. Если же получение прошло неуспешно, отправляется информационное сообщение, а сообщение от продюсера остаётся в очереди, пока не будет получено подписчиком.

Основной особенностью этого брокера является возможность настройки гибкого роутинга: при отправке сообщение необязательно должно проходить только прямолинейный путь от продюсера к подписчику. В процессе оно может проходить через ряд промежуточных узлов обмена, которые могут перенаправлять его в различные очереди.

В рамках этого брокера инициатором информационного обмена является продюсер, только он отправляет сообщение в сеть, в то время как подписчик не может запросить его сам (так называемая «push-доставка сообщений»).

Apache Kafka представляет собой брокер, который, в отличие от RabbitMQ, хранит все сообщения в виде распределённого лога, причём гарантируется, что порядок сообщений отражает последовательность их

поступления в систему. Сообщение в этом логе хранится в течение определённого времени, и работа построена таким образом, что продюсеры пишут новые сообщения в систему, а подписчики сами их запрашивают. При надобности организуется хранение сообщений в рамках тем. То есть можно сказать, что происходит определённого рода группировка сообщений в рамках одной темы.

Если попытаться некоторым образом обосновать выбор в пользу той или иной системы, то следует учесть, что RabbitMQ позволяет сконфигурировать даже весьма сложные сценарии доставки сообщений, что даёт разработчикам гибкость в построении нужного сценария информирования о событиях. При этом следует учитывать, что порядок доставки сообщений не гарантируется.

В свою очередь, брокер Apache Kafka больше предназначен для построения высоконагруженных систем сферы bigdata, так как сама его парадигма параллельной обработки, репликации позволяет создавать достаточно надёжные системы и обеспечивать неограниченные возможности по масштабированию. Высокая пропускная способность, а также возможности извлечения сообщений из очереди за определённый период времени (так как они хранятся в очереди, как мы сказали ранее, именно в том порядке, в каком были отправлены) являются мощным инструментом для анализа происходящего в историческом разрезе.

7. ПОТОКОВАЯ ОБРАБОТКА С УЧЕТОМ СОСТОЯНИЙ И ОСНОВЫ ПОТОКОВОЙ ОБРАБОТКИ

7.1. Источники данных

Под источниками данных в контексте потоковой обработки подразумеваются источники, из которых непрерывно поступают информационные потоки для дальнейшей обработки. Эти источники могут быть разнообразными и включать в себя:

- Сенсоры и устройства IoT: В мире Интернета вещей (IoT) сенсоры на устройствах непрерывно собирают данные о состоянии окружающей среды, например, температуре, влажности, движении и других параметрах. Эти данные могут быть важными для мониторинга и управления процессами.
- Логи приложений: Приложения и сервисы веб-приложений, серверов и других систем генерируют логи, которые содержат ценную информацию о работе системы, ошибках, событиях и запросах. Анализ логов в реальном времени может помочь выявлять проблемы и аномалии.
- Социальные сети: Социальные платформы генерируют огромное количество данных о поведении пользователей, их интересах и взаимодействиях. Анализ потоков данных из социальных сетей может помочь компаниям понимать мнения пользователей и адаптировать стратегии маркетинга.
- Системы мониторинга и управления: В области инфраструктуры и промышленности множество систем мониторинга и управления генерируют данные о состоянии оборудования, производственных процессах и энергопотреблении. Эти данные могут быть важными для обеспечения эффективной работы и предотвращения аварийных ситуаций.

Гарантированная доставка и управление задержками

Один из ключевых аспектов при работе с источниками данных в потоковой обработке - это гарантированная доставка и управление задержками. Потоковая обработка требует точности и актуальности данных, поэтому важно, чтобы данные достигали обработки немедленно и без потерь.

В этом контексте, принципы обеспечения гарантированной доставки включают в себя:

- Устойчивость к отказам: Системы потоковой обработки должны быть спроектированы с учетом возможных отказов и сбоев при передаче данных. Механизмы повторной отправки и механизмы обнаружения ошибок могут быть использованы для обеспечения надежности доставки.

- Управление задержками: Задержки при передаче и обработке данных могут повлиять на актуальность анализа. Поэтому важно иметь механизмы, позволяющие управлять задержками и оптимизировать время доставки данных.

- Буферизация и масштабируемость: Для обработки больших объемов данных и обеспечения устойчивости к временным нагрузкам может использоваться буферизация. Буферы могут сглаживать временные пики и позволять системе эффективно работать даже в условиях повышенной нагрузки.

Важно понимать, что разнообразие источников данных требует гибких и адаптивных решений. Например, при работе с данными с датчиков IoT, может потребоваться оптимизация для обработки высокой частоты данных, а при работе с социальными сетями - обработка и фильтрация больших объемов информации. Оптимизация и настройка источников данных - важная часть успешной реализации архитектуры потоковой обработки данных.

7.2. Платформа потоковой обработки

Основные функции платформы потоковой обработки:

- Управление данными: Платформы предоставляют средства для сбора, передачи и обработки данных из различных источников. Они позволяют настроить правила для фильтрации, трансформации и агрегации данных в реальном времени.

- Обработка данных: Одним из ключевых компонентов платформ потоковой обработки является система обработки данных. Она включает в себя алгоритмы для агрегации, анализа, классификации и других операций над потоками данных.

- Управление задержками: Многие платформы позволяют управлять задержками при обработке данных. Это важно для соблюдения требований к актуальности анализа в различных сценариях.

- Масштабируемость: Платформы обеспечивают масштабируемость для обработки больших объемов данных. Они могут

автоматически адаптироваться к изменяющейся нагрузке и обеспечивать стабильную производительность.

- Управление состоянием: В некоторых случаях потоковая обработка требует учета состояния процесса. Платформы предоставляют средства для управления состоянием данных и процессов.
- Интеграция: Платформы потоковой обработки обычно поддерживают интеграцию с другими системами и сервисами. Это позволяет включать потоковую обработку в более широкие архитектурные решения.

Примеры популярных платформ:

- Apache Kafka: Это распределенная платформа потоковой обработки и сообщений. Она спроектирована для высокопроизводительного сбора, передачи и хранения данных в реальном времени. Kafka обеспечивает масштабируемость и устойчивость к отказам, что делает ее популярным выбором для обработки больших объемов данных.
- Apache Flink: Это распределенная платформа для обработки данных в реальном времени и батчевом режиме. Flink обеспечивает поддержку сложных аналитических операций и высокую производительность. Он также предоставляет возможность управления задержками и обработки состояния.
- Apache Storm: Это система обработки данных в реальном времени, предназначенная для анализа потоков данных с низкой задержкой. Storm позволяет создавать сложные топологии обработки и обеспечивает надежность и устойчивость к сбоям.
- Amazon Kinesis: Это управляемая платформа потоковой обработки данных от Amazon Web Services (AWS). Kinesis предоставляет инструменты для сбора, анализа и визуализации данных в реальном времени.

Выбор подходящей платформы потоковой обработки зависит от требований проекта, масштаба задачи и экосистемы инструментов, с которой приходится работать. Например, если требуется обработка огромных объемов данных, Apache Kafka может быть предпочтительным вариантом, а

если необходимы сложные аналитические операции, то Apache Flink может быть более подходящим решением. Каждая платформа имеет свои преимущества и ограничения, и важно правильно подобрать ту, которая наиболее соответствует потребностям конкретного проекта.

7.3. Обработка и преобразование данных

Операции над потоками данных: фильтрация, преобразование, объединение

Обработка и преобразование данных являются центральной частью потоковой обработки. Важно иметь набор операций, которые позволяют анализировать данные, выделять важную информацию и адаптировать её для дальнейшего использования. Среди наиболее распространенных операций над потоками данных выделяются:

- Фильтрация: Эта операция позволяет отбирать данные на основе заданных условий. Например, можно фильтровать поток событий, чтобы выбрать только те, которые соответствуют определенному критерию.
- Преобразование: Преобразование данных позволяет изменять их формат или структуру. Это может включать в себя переименование полей, преобразование типов данных и другие манипуляции.
- Объединение: Объединение данных из разных источников позволяет получать более полную картину. Например, данные из разных сенсоров могут быть объединены для более точного анализа события.

Обработка оконными функциями

В некоторых случаях анализ данных требует учета временных интервалов, например, агрегация данных за определенный период времени. Для этого используются оконные функции, которые позволяют группировать данные в определенные временные окна и применять к ним агрегирующие операции.

Существует несколько видов оконных функций:

- Временные окна: Данные группируются по временным интервалам, например, по часам, дням или неделям. Это позволяет агрегировать данные за определенный период.
- Счетчиковые окна: Данные группируются по количеству событий. Например, можно анализировать данные за каждые 1000 событий.
- Сессионные окна: Данные группируются по сессиям, которые могут определяться по временным интервалам неактивности между событиями.

Обеспечение низкой задержки и высокой производительности

Обработка данных в реальном времени требует минимизации задержек и обеспечения высокой производительности системы. Для этого используются различные методы и подходы:

- Параллелизм и распределение: Использование параллельных вычислений и распределенных систем позволяет обрабатывать большие объемы данных эффективно.
- Компактное представление данных: Оптимизация представления данных может снизить нагрузку на сеть и память, что уменьшит задержки.
- Использование кэширования: Кэширование результатов предыдущих операций может сократить вычисления и ускорить обработку.
- Управление памятью: Эффективное использование памяти может снизить накладные расходы и улучшить производительность.

Обработка и преобразование данных в потоковой обработке — это искусство нахождения баланса между актуальностью, точностью и производительностью. Разработчики должны учитывать особенности данных, архитектуры системы и требования к конечным результатам. Эффективные методы обработки данных в потоке могут сделать решение более отзывчивым, релевантным и значимым для бизнеса или пользователей.

7.4. Хранение состояния

Проблемы хранения состояния в потоковой обработке

Хранение состояния в контексте потоковой обработки является важной и сложной задачей. Состояние представляет собой информацию, которую система должна запоминать между различными событиями для обеспечения целостности и актуальности анализа данных. Однако хранение состояния в потоковой обработке сталкивается с рядом проблем:

- Масштабируемость: При обработке больших объемов данных требуется эффективное масштабирование хранилища состояния. Как обеспечить быстрое доступ к состоянию при росте нагрузки и объема данных?
- Управление состоянием: Как управлять состоянием в распределенных системах? Как обеспечить согласованность данных и избежать конфликтов при одновременном доступе?
- Надежность: Состояние должно быть устойчивым к сбоям и отказам. Как обеспечить сохранность данных даже при сбоях в системе?
- Задержки: Некоторые операции над состоянием могут занимать время, что может повлиять на общую задержку при обработке данных. Как минимизировать влияние задержек при работе с состоянием?

Вот несколько подходов и решений:

1. Инмемори хранилища: Использование инмемори (памяти) хранилищ данных может существенно ускорить доступ к состоянию. Такие хранилища предоставляют быстрый доступ к данным за счет хранения их в оперативной памяти, что позволяет снизить задержки при обработке. Однако такой подход требует управления памятью и может быть ограничен объемом доступной памяти на устройстве.

2. Распределенные базы данных: Использование распределенных баз данных, таких как Cassandra, Apache HBase или Amazon DynamoDB, может обеспечить масштабируемость и надежность хранения состояния. Такие базы данных могут автоматически реплицировать данные для обеспечения отказоустойчивости.

3. Кэш-системы: Кэш-системы, такие как Redis или Memcached, могут использоваться для хранения часто используемых данных состояния. Они предоставляют быстрый доступ к данным за счет хранения их в памяти, но требуют внимания к вопросам надежности и управления жизненным циклом данных.

4. Системы управления состоянием: Существуют специализированные системы управления состоянием, которые предоставляют механизмы для сохранения, обновления и запроса состояния в потоковой обработке. Эти системы могут обеспечивать согласованность данных и устойчивость к сбоям.

5. Паттерн "Легковесное состояние": Этот паттерн предполагает, что состояние не хранится непосредственно в системе потоковой обработки, а вынесено во внешние системы. Система потоковой обработки хранит только ссылки на состояние. Этот подход может помочь снизить нагрузку на систему обработки и упростить управление состоянием.

6. Продвинутые алгоритмы и структуры данных: Использование оптимизированных алгоритмов и структур данных может помочь улучшить производительность работы с состоянием. Например, Bloom фильтры могут быть использованы для быстрого определения наличия элемента в наборе данных.

7. Использование внешних хранилищ данных для состояния. Для решения проблем хранения состояния многие системы потоковой обработки используют внешние хранилища данных. Это могут быть распределенные базы данных, кэш-системы или другие хранилища. Важно правильно выбирать хранилище, учитывая требования проекта и характеристики данных.

Преимущества использования внешних хранилищ данных:

- Масштабируемость: Внешние хранилища часто предоставляют механизмы масштабирования, позволяющие управлять ростом объема данных.

- Согласованность и надежность: Некоторые хранилища обеспечивают средства для согласованности и надежности данных, что может упростить управление состоянием.
- Отказоустойчивость: Внешние хранилища могут обеспечивать сохранность данных даже при сбоях в системе, что важно для поддержания целостности состояния.
- Разнообразие хранилищ: Существует множество различных хранилищ данных, позволяющих выбрать наиболее подходящее для конкретных требований. Это могут быть реляционные базы данных, NoSQL-хранилища, кэши и другие.

Хранение состояния - это один из ключевых аспектов потоковой обработки данных, который имеет прямое влияние на качество анализа и реакцию системы на изменения. Выбор подходящего хранилища данных и разработка эффективной стратегии работы с состоянием требует глубокого понимания требований проекта и характеристик данных. Правильное решение в области хранения состояния помогает обеспечить надежность, производительность и актуальность анализа данных в потоковой обработке.

8. ЛАБОРАТОРНЫЕ ЗАНЯТИЯ

Примерный перечень тем

1. Установка и развертывание Apache Spark
2. Применение Apache Spark для считывания, обработки и записи данных
3. Применение data frame API Apache Spark
4. Применение rdd API Apache Spark
5. Применение Apache Spark для решения задачи преобразования данных
6. Применение Apache Spark для решения задачи анализа данных
7. Применение Apache Spark для решения задачи машинного обучения

8. Установка и развертывание Apache Kafka

9. Создания простейшего сервиса Apache Kafka, который «слушает» источник и передает данные на Apache Spark job

10. Установка и развертывание Apache Flink

11. Создание простейшего сервиса Apache Flink поставляющего данные на основе состояний

9. КОНТРОЛЬНАЯ РАБОТА

Примерный перечень тем

1. Распределенное хранение и обработка данных

Примерные задания

1.Опишите модель параллельных вычислений фреймворка Spark:

2.Опишите преимущества и недостатки отложенных вычисление. Отказоустойчивость

3. Как происходит хранение данных в памяти и управление памятью

4. Опишите понятия неизменяемости и интерфейс RDD

5. Какие типы наборов RDD вы знаете

6. Чем отличаются преобразования и действия?

7. Чем отличаются широкие и узкие зависимости?

10. ДОМАШНЯЯ РАБОТА

Примерный перечень тем

1. Особенности применения методов распределенных вычислений

Примерные задания

Цель работы:

Изучить возможности Apache Spark и выделить ключевые особенности использования данного инструмента для чтения, преобразования, записи данных, а также решения задач анализа данных

Задание и требования:

Домашняя работа включает изучение литературы и документации по Apache Spark, а также применение изученных методов и функций для решения практической задачи на выданном наборе данных. Задание будет включать следующие этапы: считать набор данных, выполнить необходимые преобразования и расчеты, сохранить преобразованный набор данных.

Работа должна быть оформлена в виде jupyter notebook или python скрипта и отправлена в форму для приема работы. Задание индивидуально

11. ЭКЗАМЕНАЦИОННЫЕ ВОПРОСЫ

1. Что Такое Spark?
2. Каковы ключевые особенности Spark?
3. Что такое SCC?
4. Что такое RDD?
5. Что такое неизменность (Immutability)?
6. Что такое YARN?
7. Какие самые распространённые языки программирования в Spark?
8. Сколько менеджеров кластера доступны в Spark?
9. Каковы обязанности движка Spark?
10. Что такое ленивые вычисления’?
11. Что такое раздел (Partition)?

12 Для чего нужен Spark Streaming?

13. Нормально ли запускать все ваши процессы на локализованном ноде?

14. Для чего используется SparkCore?
15. Имеет ли применение File System API в Spark?
16. Чем MapReduce отличается от Spark?
17. Что вы понимаете под трансформациями в Spark?
18. Что такое Apache Kafka?
19. Как запустить сервер в Kafka?

20. Что такое традиционные методы передачи данных и чем Kafka лучше?

21. Что такое zookeeper в Kafka и можем ли мы использовать эту программу без него?

22. Почему Kafka является такой важной частью технологии?

23. Объясните, что такое последователь и лидер в Kafka.

24. Что такое потребители и пользователи в Kafka?

25. Как вы используете Kafka в качестве системы хранения данных?

26. Объясните максимальный размер сообщения, которое может принять Kafka.

27. Как разбалансировать кластер в Kafka?

12. УЧЕБНО-МЕТОДИЧЕСКОЕ И ИНФОРМАЦИОННОЕ ОБЕСПЕЧЕНИЕ ДИСЦИПЛИНЫ

Электронные ресурсы (издания)

1. Клеппман М., Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2018. — 640 с.

2. Стренгхольт Питхейн, Масштабируемые данные. Лучшие шаблоны высоконагруженных архитектур. — СПб.: Питер, 2022. — 368 с

3. Бёрнс Б. Распределенные системы. Паттерны проектирования. — СПб.: Питер, 2019. — 224 с

4. Дэвис К. Шаблоны проектирования для облачной среды / пер. с анг. Д. А. Беликова. – М.: ДМК Пресс, 2020. – 388

5. Уэске Ф., Калаври В. Потоковая обработка данных с Apache Flink / пер. с англ. В. С. Яценкова. – М.: ДМК Пресс, 2021. – 298

6. Дилан Скотт, Виктор Гамов, Дейв Клейн, Kafka в действии / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2022. – 310 с

7. Нархид Ния, Шапира Гвен, Палино Тодд, Apache Kafka. Потоковая обработка и анализ данных. — СПб.: Питер, 2019. — 320 с

8. Сэнди Риза, Ури Лезерсон, Шон Оуэн, Джош Уиллс, Spark для профессионалов: современные паттерны обработки больших данных. — СПб.: Питер, 2017. — 272 с

9. Холден Карау, Рейчел Уоррен, Эффективный Spark. Масштабирование и оптимизация. — СПб.: Питер, 2018. — 352 с

Профессиональные базы данных, информационно-справочные системы

1. Единое окно доступа к образовательным ресурсам. Раздел Информатика и информационные технологии http://window.edu.ru/catalog

2. Интернет-Университет Информационных Технологий http://www.intuit.ru/

3. Веб-сервис для хостинга IT-проектов и их совместной разработки Github http://www.github.ru

Материалы для лиц с ОВЗ

Весь контент ЭБС представлен в виде файлов специального формата для воспроизведения синтезатором речи, а также в тестовом виде, пригодном для прочтения с использованием экранной лупы и настройкой контрастности.

Базы данных, информационно-справочные и поисковые системы

1. ЭБС Университетская библиотека онлайн «Директ-Медиа» http://www.biblioclub.ru/

2. eLibrary ООО Научная электронная библиотека http://elibrary.ru