ML BD

Katya Koshchenko

Spring 2020

1 Лекция 2. Apache Spark

1.1 Ограничения MapReduce

Недостатки MapReduce:

- MR плохо подходит для интерактивных алгоритмов ML из-за постоянного взаимодействия с HDFS. Пример: хранятся точки, сделали итерацию логистической регрессии, обновили веса. Для следующей итерации опять будем читать датасет с HDFS. Для внутренней безопасности все всегда хранится на HDFS
- Если в таблице хранятся какие-то данные и хотим провести аналитику, то только парадигмами MR это сделать сложно.

Частичные решения были предложены для проблем: Hive, Pig. Они для решения задач аналитики. Hive — предоставляет юзеру язык запросов на подобие SQL (HiveQL). То есть юзер пишет запрос, который планировщиком превращается в набор MR. Как? В Hive есть Meta Store, какое-то хранилище метаданных, которое для файла хранит инфу, что он состоит из таких колонок, типов, и тд. Compiler из запроса строит план, который из себя представляет MR джобы, затем входит в метастор, чтобы понять типы колонок зарпоса и тд. То есть прогеру теперь не надо самому писать MR таски, которые что-то будут смотреть в нужном порядке

Pig — похож на хайв, только язык запросов другой, Perl подобный, и там нет Meta Store. То есть юзер должен сам указать, что в этом файле есть такие колонки с такими типами. То есть указывать больше данных.

1.2 Spark RDD

Ввели определение RDD — Resilent (отказоустойчивый), Distributed (разбитый на партиции), Dataset. Из них можно только читать (immutable), распределенно они хранятся на кластерах, причем по дефолту не в HDFS, а в локальной памяти. Это основная абстракция в Spark. И есть операции, которыми можем из одной RDD получить другую.

Для отказоустойчивости надо запоминать граф вычислений: как из набора RDD получаем интересующую. И если есть такой граф, и часть данных была потерена, то для восстановления нужно просто пройти по графу зависимостей.

Как можно построить RDD. Можно из данных на HDFS или из оперативной памяти, а можно из другого RDD выполнив над ним либо transformation, либо action. Примеры трансформаций: map, filter, join, reduceByKey и тд. Action основные — count (скольк элементов в датасете), save (сохранение рдд на хдфс, например), collect (элементы распределенного по машинам датасета собираем на драйвер, то есть машину, организующую выполнение задач на кластере, типо мастера). Эти action и transformation дают декларативное описание, что хотим сделать с данными. И все отражается в графе вычислений.

Юзер может управлять RDD partitioning. Вот она как-то распределена, и разбита на партиции. Юзер может решать, сколько партиций хотим иметь. Если он достаточно маленький, то зачем разбивать на 100 кусков и хранить на разных мшаинах, если можно на одной вполне. И еще юзер может контролировать persistence. Во он описал вычисления и знает, что какой-то RDD будет часто переиспользоваться. Здесь юзер может сказать, что тк для каждой итерации алгоритма нужно будет прочитать все точки в датасете, то этот рдд хочу хранить в оперативной памяти.

(На слайде был пример кода. textFile — чтение с хдфс, persist — хранить в оперативной памяти.)

Конкретный RDD знает, из каких он был получен, и в случае необходимости его поэтому можно пересчитать легко. При этом все вычисления ленивые. Когда описываем транформации, то ничего не происходит. Они триггерятся только когда появляется action в программе. Еще когда пишем persist, то храним в оперативке. Но если у нас мало опертивки или кластеры нагружены, то можно передать аргумент, чтобы хранить на диске. Скорость упадет, но все еще лучше, чем в хдфс.

1.3 Spark programs

Драйвер свзывается с работниками. И когда решает, на какой машине хочет запустить какую-то задачу, то, как и в случае с MR, он принимает во внимание инфу о местоположении данных. Ведь хотим минимизировать пересылку данных по сети. Короче если можем запустить на той же машине, где данные, то так и сделаем, а иначе запустим на ближайшей.

(На слайде описание трансформаций и действий и их изменений по типам)

(Несколько примеров гафиков, показывающих, что Spark сильно быстрее Hadoop и HadoopBinMem)

Пример с PageRank. Пусть надо поджоинить датасеты по ключу, но их куски на разных кластерах. Чтобы не гонять данные между машинами, надо делать свою функцию партицирования. Копартицитные датасеты — разбивались одной функций партицирования. И тогда это гарантирует, что в рамках одной партиции будут одни и те же ключи. В таком случае join происходит у нас только в рамках одной партиции, что тоже сильно ускоряет.

Партиции в RDD — какая-то атомарная часть датасета, над одной партицией всегда работает один работник. А еще RDD знают о своих зависимостях, из чего было построены (для надежности).

Зависимости бывают узкие и широкие. Узкая, если дочерний партишн зависит от одного или более в родительском, при условии, что в графе вычислений можем точно узнать, от каких

будет зависеть. Это дает выигрыш по скорости. Если знаем, что для кого-то будут выполнены последовательно узкие зависимости, то не нужно несоколько раз сканировать датасет и применять по очереди все трансофрмации. Можно за один проход сделать все, что нужно. Широкие зависимости обычно появляются в ситациях, когда есть shuffle датасета, то есть на этапе построения графа не можем гарантированно ответить на вопрос, от каких родительских будет зависеть дочерний. Например, если не co-partitioned, то один ключ может быть разбросан по нескольким партициям, значит точно восстановить не можем.

Узкие после падения легко восстановить, а в широких для восстановления одной партиции придется пройтись по всем родительским.

1.4 Implementation

В рамках одного работника может быть несколько независящих другот друга Spark executors. И каждый обрабатывает свои партиции (двое не могут одну).

Спарк контекст связывает пользовательскую прогу с кластером, где решаются задачи. Контекст хранит инфу о кол-ве ресурсов, которые хотим выделить на задачу (кол-во executors, кол-во ядер для них, оперативки и тд). Еще можем управлять способом выделения ресурсов: статически или динамически. Статически — хотим столько executors, памяти и тд, вне зависимости от того, запущены какие-то задачи или нет. Просто эти ресурсы будут за нами на кластере зарезервированы и никто больше юзать их не будет. Динамически — чем больше потребность, тем больше ресурсов будет нам предоставляться.

Пайплайн. Scheduler использует графы зависимости и рабивает на этапы. Как только видит широкую зависимость (шафл данных), то говорит, что между этими шафлами — один stage. Так как на одном стейдже все узкие, то можем посчитать за один проход. Ну и чтобы это посчитать, запускаются таски.

Итого. Есть спарк приложение, в рамках которого на каждый action есть одна работа, они разбиваются на stages (узкие обрабатываем параллельно), которые запускаются последоватлельно, в их рамках запускаются таски. Память не бесконечная, так что периодически партиции будут вытесняться.