ML BD

Katya Koshchenko

Spring 2020

1 Лекция 1. Introduction, GFS, HDFS, Map Reduce

1.1 Google File System (GFS)

1.1.1 Требования к системе

1. Производительность
2. Масштабируемость (легко добавлять новые машины)
3. Надежность
4. Доступность

Мотивация к требованиям: распределенная файловая системя (РФС) состоит из какого-то количества машин, которые постоянно ломаются. И поломки становятся нормой жизни, а не какими-то редкими ситуациями. Поэтому важна надежность. Еще выяснилось, что гораздо легче поддерживать большие файлы, чем много мелких, а при работе с большими данными большинство операций — запись в конец или потоковое чтение. И возникает проблема, что несколько клиентов могут работать с одним файлом. Отсюда вытекают доступность, масштабируемость и производительность.

1.1.2 Система

- *GFS-client* — много клиентов
- *GFS-master* — одна машина
- *Chunk-серверы* — на них хранятся данные, их много

Файлы большие, хранятся следующим образом. Файл разбивается на куски (chunks) фиксированного размера (64 MB). Мастер дает каждому куску глобальный идентификатор. Затем эти куски реплицируются и раскидываются по chunk-серверам, которые выбирает мастер

Master:

1. Хранит метаданные: что-то в духе индекса, для файла знает идентификаторы chunk'а, а для chunk'а знает, к какому файлу он принадлежит
2. Создает / реплицирует chunk'и (для надежности, по умолчанию минимум 2 копии каждого файла)
3. Сборка мусора: файл удален, надо освободить место, почистить метаданные
4. HeartBeat сообщения для общения с chunk-серверами: проверка того, жив ли сервер
5. Обрабатывает запросы связанные с метаданными, сами данные через него не проходят, иначе bottleneck
6. Логгирует все операции (для надежности)

Chunk-сервер:

1. Хранит данные на локальном диске
2. Обменивается данными с пользователями и другими серверами
3. Обменивается запросами и метаданными с мастером

1.1.3 Как это все работает, и кто что делает

Несколько chunk серверов могут быть в одной подсети, и при ее отказе откажут все машины сразу. Если так получилось, что все копии какого-то файла были именно на этих машинах, то потеряем файл. Поэтому для надежности надо реплики раскидывать и между подсетями тоже, за это ответственный мастер.

Что влияет на то, на каком chunk сервере будет создан кусок:

- Утилизация диска, сети
- Как давно создан последний кусок (наподобие кеша, если недавно создавали, то ожидаем, что его еще будут читать в ближайшее время)
- Местоположение сервера

Когда машина падает и число реплик какого-то куска оказывается ниже заданного уровня, то происходит репликация. Master через HeartBeat понял, что сервер вышел из строя, и начинает в порядке приоритета реплецировать потерянные куски. Приоритет — у кого больше копий потеряно (в простом варианте).

Мастер также занимается баансировкой нагрузки, перераспределяя chunk'и между серверами. Это нужно для выравнимания нагрузки между машинами и лучшей утилизации дисков и сетей.

Сборка мусора: файл сразу не удаляется, а по-особому переименовывается (вроде отметки об удаленности) и в течение заданного времени его можно восстановить. Если никто не восстановил, то, когда мастер видит удаленный файл, он освобождает его память на chunk-серверах и чистит свои метаданные

На chunk-сервере хранятся еще метаданные о том, какие chunk-id и их версии на нем лежат

Мастер может восстанавливаться по логам и делает периодические снепшоты (когда лог стал слишком длинным).

1.2 Hadoop

Hadoop Distributed File System (HDFS) — open-source реализация GFS

- Master — NameNode
- Chunk-сервер — DataNode

Глобальное отличие: В GFS клиент сам связывается с несколькими chunk-серверами, а в HDFS связывается только с одной DataNode, а она уже пересылает данные всем остальным.

Также тут резализван MapReduce.

1.3 MapReduce

1.3.1 Что это такое и примеры

Лог событий рекламы в VK за день это 70 GB. Писать под каждую задачу низкоуровневый код — глупо. Заметим, что большое количество задач формулируется в виде последовательных map и reduce. А эти функции уже можно распараллелить.

1. map: $(k_1, v_1) \to list(k_2, v_2)$
2. reduce: $(k_2, list(v_2)) \to list(v2)$

Пример *WordCount*:

1. Map: слово → (word, 1)
2. Reduce: сумма счетчиков для каждого слова

Пример *DistributedGrep*:

1. Map: ищет строчки, содержащие паттерн
2. Reduce: ничего не делает

Пример *InvertedIndex*:

1. Map: документ → (word, docID)
2. Reduce: слово → (word, list(docID)) — чистит список идентификаторов документов

Цели MapReduce: система берет на себя все, касающееся надежности и параллелизма (отказоустойчивость, распределение данных, планирование и распределение ресурсов). Пользователь пишет две функции и отдыхает.

1.3.2 Система

Система состоит из мастера, MapWorker'ов, ReduceWorker'ов. Клиент указывает количество работников каждого типа. Мастер распределяет Chunk'и между работниками. MapWorker выдает разбиение (partition): берем хеш от ключа и отправляем по нему в определенный partition. Reduce должен быть инкрементальным.

Машины, как и раньше, могут падать. Если падает MapWorker, то перезапускается его map задача + все educe задачи, которые зависят от него. Если падает ReduceWorker, то перезапускается только его reduce задача. При падении мастера прерывается весь процесс MapReduce вычислений.

Хотим минимизировать пересылку данных по сети. Мастер знает, где лежат данные, на которых хотим запускать map задачи. Идеальный вариант: запускаем map там же, где лежат данные. Если это невозможно, то мастер выбирает работника, который по топологии сети находится максимально близко к данным.

Мастер время от времени опрашивает работников, на которых запущены задачи, чтобы понять их статус. Задача находится в одном из состояний:

- Idle — еще не запущена, ожидает запуска
- In progress
- Completed

Если Worker не отвечает в течение заданного времени на HeartBeat сообщения, то все его задачи переходят в Idle и распределяются мастером

Straggler — Worker, которому надо сильно больше времени для завершения его задач в сравнении с остальными. Может быть из-за медленного железа, далеких данных и тд. В таком случае мастер запускает дублирующие задачи для задач in progress в тот момент, когда большинство задач уже completed. Задача завершится, когда завершится либо она сама, либо любой дубликат.

Хеш может быть плохим, но можно задать его самому, чтобы нагрузка не упала на какие-то серверы в большем масштабе.

В результате работы map на локальном диске работника окажется R частей данных. При этом данные на вход reduce задачи будут подавать в порядке неубывания ключей. Из-за этого можем

сделать частичную сортировку за просто так: reduce = id function, тогда после завершения всех reduce задач получим файл из R частей, внутри которых данных посортированы. Если ReduceWorker один, то просто получаем сортировку. Или же можно сделать partitioning по ключу, а не по их хэшу.

Можно задавать combiner функции, которые на MapWorker делают предподсчет. Например, это можно использовать в задаче WordCout, где слова естественного языка распределены по Zipf law.

Если в данных есть записи, которые заставляют map/reduce падать с ошибкой, то весь процесс упадет с ошибкой. Мастер определяет записи, которые несколько раз падают и затем помечает их как bad record, которые затем можно пропустить при перезапуске задачи.

Counters — map/reduce могут инкрементировать счетчики, если встречают какие-то события.

2 Apache Spark

2.1 Лекция 2. 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.

2.2 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.

2.3 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.

2.4 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.

3 Лекция 3. SparkSQL

3.1 Background and Goals

Spark проблемы: низкоуровневый процедуральный код и отсутсвие оптимизаций.

Shark — первая попытка сделать резляционный интерфейс Spark, сделал так, чтобы Apache Hive System запускалась на Spark. Но в нем команды можно было писать только именно SQL строкой, оптимайзер был настроен именно на MR, не расширить вообще было.

Заметили, что большинство пайплайнов — комбинации реляционных и процедуральных алгоритмов. И сделали SparkSQL — новый модуль в Apache Spark. В нем DataFrame — коллекции структурированных записей, которыми можно манипулировать, пользуясь Spark APIs и процедурацльный, и реляционные. Их можно создать напрямую из RDD.

3.2 Programming interface

DataFrame — распределнная коллекция строк с одинаковой схемой. Он эквивалентен таблице в реляционной базе данных. Можно им пользоваться как RDD.

Строить фреймы можно из внешнего источника (HDFS, Hive) или существующей RDD. Вообще его можно рассматривать как RDD объектов-строк, так что над ним всякие процедуральные операции как map можно делать. Реляционные операции выполнять можно, используя DSL (domain-specific language), похожий на Pandas в питоне.

Фреймы тоже ленивые, так что оптимизировать их в радость. Spark строит перед их запуском логический план, а потом физичекский план. В отличае от оригинального Spark, этот строит AST выражения, которое передается в Catalyst для оптимизации.

Всякие штуки. Cache() <-> persist(), кэширование может быть полезно для интерактивных запросов и итеративных алгоритмов мл. UDF — функция, определяемая юзером, которая выполнится над фреймом.

3.3 Catalyst

В Каталисте лежат какие-то базовые библиотеки для представления и правила манипулирования с деревьями. Самый популярный подход к составлению правил: найти поддерево определенной структуры pattern matching функциями и заменить их на что-то. Каталист группирует правила в батчи и запускает каждый, пока он не достигнет точки фиксации, то есть пока дерево не перестанут меняться.

4 Лекция 4. Distributed ML Introduction.

5 Лекция 5. Categorical Features.

Фичи бывают: бинарные, вещественные, категориальные (айди, категория, город и тд), порядковые. Кросс-фича $X_{ij} \in C_i \times C_j$ построена из двух категориальных фичей. Она может иметь и sparse (one-hot encoding), и dense (target mean encoding) представление.

Даже на искусственном примере с полом юзера и доменом объявления кросс-фичи вроде могут быть полезными, так как несут обычно для модели очень мощный сигнал. Но с ними есть проблемы:

- Ручной поиск таких правил очень трудоемкий. Надо разбираться в предметной области, чтобы искать такие сочетания. В примере были кросс-фичи второго порядка (две категориальные фичи в представлении). А если порядок увеличивать, то сложность еще сильнее возрастает.
- При их использовании размерность признакового пространства быстро растет. Тк даже если у нас есть 3 варианта пола, 20000 объявлений, это уже 60000. А если размерность категориальных фичей миллионы, то безумно много. С такими большими пространствами сложно работать

Сегодня говорим про то ,как люди на практике с ними борятся.

5.1 Field-aware Factorization Machines

5.1.1 Poly2

Самая простая — Poly2. Как перейти от ручного поиска кроссфичей — добавить сразу все фичи. То есть рассмотреть все попарные умножения с некоторым весом:

$$Poly2(W, X) = LM(W, X) + \sum_{j1=1}^{n} \sum_{j2=j1+1}^{n} W_{h(j1,j2)} \cdot X_{j1} \cdot X_{j2}$$

где h кодирует j1, j2 в натуральное число. Проблемы подхода:

- Вычислительная сложность. Если ненулевых слагаемых n', то будет $O(n' * *2)$ действий.
- Получаем только кроссфичи второго порядка.
- С этой моделью легко переобучиться. Если пара X_j1, X_j2 в обучающем множестве встречается редко, то вес в слагаемом не сможем достаточно хорошо обучить.

5.1.2 FM

Factorization machines — следуюий шаг в развиитии poly2. Отличие тв том, что в каждой паре был отдельный вес, а теперь у каждой фичи будем обучать вектор веса размерностью k:

$$FM(W, X) = LM(W, X) + \sum_{j1=1}^{n} \sum_{j2=j1+1}^{n} (W_{j1} \cdot W_{j2}) \cdot X_{j1} \cdot X_{j2}$$

Обычно это k небольшое. Почему это вообще должно работать. Потому что это эмбеддинги, ура. Можно любую матрицу положительноопределенную разложить на WWT, а это оно и есть. Плюс подхода: переобучиться сложнее, тк для каждой фичи обучаем отдельный вектор. Сложность почти такая же: $O(n' * *2 * k)$. Но можно немного модицифровать подсчет, и тогда будет O(n' * k):

$$FM(W, X) = \frac{1}{2} \cdot \sum_{j=1}^{n} [(\sum_{k=1}^{n} W_k \cdot X_k) - W_j \cdot X_j] \cdot W_j \cdot X_j$$

(На слайде график как меняется RMSE на датасете нетфликса. Даже если на каждом признаке обучать маленький вектор, то по сравнению с SVM уже растет качество.)

5.1.3 FFM

Field Aware Factorization Machines. Есть запись, что юзер кликнул по объявлению на такой площадке, с таким рекламодателем, такой-то пол. В обычных FM у все попарные произвдеения признаков — три слагаемых. Хотим дать больше степеней свободы модельке. Все фичи можно разбить на категории (fields), и для каждой фичи обучать не один, а F векторов. И теперь при разбиении на пары брать именно тот вектор, который соответствует напарнику фичи:

$$FFM(W, X) = LM(W, X) + \sum_{j1=1}^{n} \sum_{j2=j1+1}^{n} (W_{j1,f2} \cdot W_{j2,f1}) \cdot X_{j1} \cdot X_{j2}$$

На практике размерность категорий обычно сильно меньше размерности фичей. Время работы O(n'**2 * k).

5.1.4 Оптимизация

Как обучать. Можно, конечно, SGD, но обычно гораздо лучше работает Adagrad. На слайде сравнения всякие. Тк ушли от того, что считали по отдельному весу для каждой пары, а вместо этого обучали вектора, то сходиться стало быстрее, и время обучения уменьшилось. FFM обучается помедленнее, тк для каждой фичи больше векторов, но зато размерность векторов сильно меньше можно сделать. Ну и loglos сильно меньше у него.

Оставшиеся минусы. Все еще только фичи второго порядка. И все еще долго обучается.

5.2 Deep Crossing

Теперь про глубокое обучение. Умеем обучать кроссфичи второго порядка. Хотим находить более сложные, но чтобы при этом не переобучались, были устойчивы и тд. Для этого есть Deep Crossing. Что это за модель. Взяли one-hot, заэмбеддили. Затем их все вектора сложили в один большой вектор. В него же суем вектор с некатегориальными (веществennозначными) фичами, можно даже не эмбеддить. Суем это все дело в пять слоев с relu и residual connections (архитектура на слайде). В явном виде веса для фичей мы не получим, то есть не проинтерпретируешь. Но вроде эта вся радость обучается.

5.3 Deep 'n Cross

Сделали опять вектор вещественных с вектором эмбеддингов категориальных. Засовываем эту штуку в Deep network и Cross Network. Deep — обычный DNN. Cross network — пытаются в явном виде обучать взаимодействия определенного порядка. Сколько уровней, вплоть до такого порядка и обучим.

На слайдах таблички с evaluation. Если посмотреть на logloss, то разница с FFM в четвертом знаке, но утверждается, что это уже приносит много денег. На слайде таблица со сравнением числа параметров DCN и DNN, нужных, чтобы добиться какого-то logloss. У DCN это число сильно меньше из-за Cross network подсетки, тк она дает высшие фичи.

Есть ли смысл в итерациях больше второго порядка? На слайде опять график. Как только в cross network появляется хоть 2й слой, то logloss сильно падает, то есть фичи второго порядка сильно помогают. 3й слой уже не так сильно помогает, так что 2го порядка кажется, что уже хватает (но авторы это все привели только на одном датасете).

5.4 Deep FM

Просто взяли FM сложили в сетку. FM компонента: считаем попарные произведение между эмбеддингами. Deep компонента: все эмбеддинги отправляются в полносвязный слой, и с ними что-то происходит.