ML BD

Katya Koshchenko, Egor Bogomolov

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.