Параллельные и распределенные вычисления. Задание 6 “Индекс Википедии”.

Старченко Владимир

Основная часть

Поставленная задача

Построить инвертированные индексы для русской и английской Википедий.

Индекс должен состоять из строк вида word t docid1:tfidf1 t docid2:tfidf2 . . . , где word - слово, docid - идентификатор статьи, tfidf - значение меры TD-IDF для пары слово-статья, t - символ табуляции (пробелов вокруг него быть не должно).

Для каждого слова должно быть включено 20 наиболее релевантных статей в порядке убывания их TF-IDF. При равных значениях TF-IDF статьи должны располагаться в порядке возрастания docid.

При построении индекса следует игнорировать 20 наиболее часто встречающихся в текстах статей слов, используя предварительно вычисленный список данных слов.

Требования к парсингу слов минимальные. Обязательно только отбрасывание знаков пунктуации и приведение к нижнему регистру. Для простоты в тексте русской Википедии можно вообще оставлять только кириллические символы, а в английской - латинские. Приведение слов к нормальной форме не требуется.

Дополнительно требуется минимизировать время построения индекса и обеспечить масштабируемость реализации, то есть возможность эффективно обрабатывать данные большего объема и использовать большее число машин.

Для решения следует использовать Hadoop MapReduce. Программы могут быть реализованы на Python (Hadoop Streaming) или Java. Полученные индексы должны быть сохранены в HDFS в отдельных директориях в описанном формате.

Обоснование решения

В данной задаче у меня было всего 3 идейно различных решения.

Тривиальный вариант (A): Маппер обсчитывает часть статей, для каждой статьи считает tf для слов в ней и печатает на выход значения [word, tf, docid]. Редьюсер получает на вход слова, отсортированные по алфавиту. Считывае все строки [word, tf, docid], полученные от мапперов, сортирует по tf, оставляет 20, умножает на tf на idf, печатает ответ для данного слова. Здесь есть небольшая модификация. Так как не нужно сортировать все строки для данного слова, а только 20 первых, можно сделать оптимизацию. Сделать PriorityQueue на 20 элементов и вместо сортировки всех (за n log n), сортировать только первые 20 (n log 20). Идеальным вариантом было бы находить за линию k-ю порядковую статистику. Однако k-я порядковая за линию — довольно утомительная вещь, да и прирост был бы не большим по сравнению с PriorityQueue. Так же хочется заметить, что в решении с PriorityQueue требуется хранить всего 20 элементов, что дает улучшение по памяти. Такое решение дает счет 21K CoreSeconds.

Улучшение (B): Можно попробовать свалить всю работу по сортировке из редьюсера на встроенный sort, выполняемый между map и reduce. Раз уж сортировка и так выполняется, почему бы и нет. Тогда редьюсер получал бы уже отсортированные слова. Ему нужно было бы всего лишь прочитать и запомнить первые 20, а остальные пропустить (увеличивая счетчик для вычисления idf потом). Это делается с помощью Options для хадупа и работает примерно 18K CoreSeconds.

Мега-оптимайз (C): Довольно большое время занимает передача данных из маппера в редьюсер. Попробуем это исправить. Предположим, что данные, получаемые одним маппером помещаются в его память. Сейчас маппер у нас пишет на выход слово, даже если оно встречалось во всех его статьях. Если таких статей много, а в редьюсере все кроме (возможно) двадцати будут выкинуты. Значит они напрасно передавались. Тогда в маппере сделаем то же, что и в редьюсере: собрав все тройки [word, tf, docid] в список и

отсортировав его, создадим PriorityQueue на 20 элементов. В итоге получим для каждого слова список из не более чем 20-ти статей и индексов tf, а так же число – в скольких статьях участвовало слово для конкретного маппера. В редьюсере же нужно всего лишь найти сумму этих чисел и все пары [tf, docid] прогнать через PriorityQueue. Это решение дало CoreSeconds 9500. В нем тоже есть некоторые моменты, которые можно улучшать. Про это будет сказано ниже.

Описание реализации

Выше было довольно подробно указаны реализации, однако некоторые моменты стоит отметить.

В маппере можно не хранить весь список слов, только лишь словарь word: PriorityQueue, чтобы поддерживать топ 20 самых высоких статей по tf. Так же нужно хранить словарь word: counter, где counter - счетчик в сколькох статьях участвовало данное слово. С помощью этого счетчика потом будет вычисляться idf.

Стоит отметить, что вариант с PriorityQueue не проходит по памяти. Видимо, сам PriorityQueue занимает довольно много места, поэтому на каждое уникальное слово в маппере хранить по PriorityQueue не получится. Поэтому я отказался от PriorityQueue в пользу heapq. Оно позволяет сделать, по сути, ту же PriorityQueue, только на массивах. Это занимает гораздо меньше памяти.

Отдельно скажу скажем про нахождение топ 20 слов. Задача очень простая:

В маппере для всех слов по всем текстам этого маппера составляем Counter и передаем редьюсеру на вход пары [word, count] где count - количество вхождений этого слова в тексты данного маппера

В редьюсере акумулируем значения от разных мапперов (суммой) и вставляем в PriorityQueue на 20 элементов. В конце печатаем топ 20 слов из очереди.

Результаты

Результаты в CoreSeconds разнятся от запуска к запуску, но в среднем получаются такие результаты:

En (CoreSeconds 9586): https://everest.distcomp.org/jobs/5a3e2ca61500001700eb5f43
Ru (CoreSeconds 3010): https://everest.distcomp.org/jobs/5a3e2be21500001500eb5f36
Проверка валидности: https://everest.distcomp.org/jobs/5a3e2d8c150000d505eb5f58
Топ 20 (en): https://everest.distcomp.org/jobs/5a3eb3cb150000543aeb6bdb
Топ 20 (ru): https://everest.distcomp.org/jobs/5a3eb3ad150000543aeb6bd1

Анализ решения

Решения довольно простые и в основном, корректность следует из алгоритма. Но есть кое что, на чем стоит остановиться отдельно.

tf-idf считаются кореектно, так как мы воспользовались тем, что сортировка модет происходить без знаний о idf. Только по tf. Значит в каждый момент значения по tf будут отсортированы правильно.

Решение (C): Может понадобиться сброс памяти с агрегированными данными в маппере, про это будет сказано ниже.

Контрольные вопросы

Вопрос 1.

Опишите свое решение. Какие MapReduce-программы запускаются, что каждая из них делает, как реализованы map и reduce? (вес 0.2)

В моем решении запускается всего 1 map-reduce программа, в которой, на основании аргументов командной строки, запускаются функции для map и reduce. Выше были описаны подробные реализации. Если на уровне идеи, маппер делает первичную обработку: парсит тексты, получает данные, с которыми можно работать дальше, частично их агрегирует. Редьюсер агрегирует данные из маппера, окончательно обсчитывает некоторые параметры (топ 20 по всем мапперам, idf), записывая в выходные файлы.

Вопрос 2.

Какие из рассмотренных на занятиях приемов, оптимизаций и стратегий для MapReduce-программ вы использовали в своем решении? Почему? (вес 0.2)

В программе (A) мы начали использовать PrioryQueue. Это просто оптимизация, которая улучшает произподительность и по времени и по памяти.

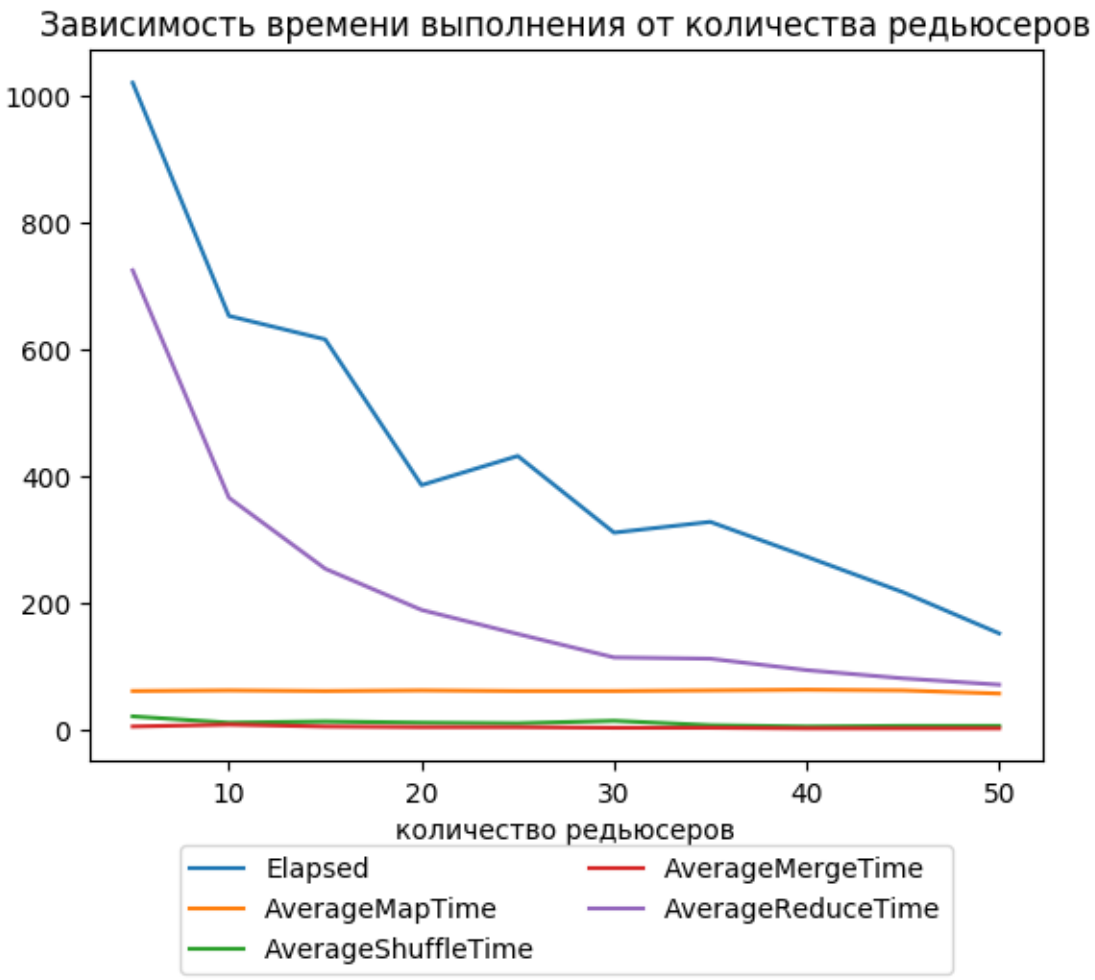
В программе (B) мы отдали часть работы (всю работу) по сортировке встроенной сортировке между маппером и редьюсером. Это дало прирост по скорости, хоть и не большой.

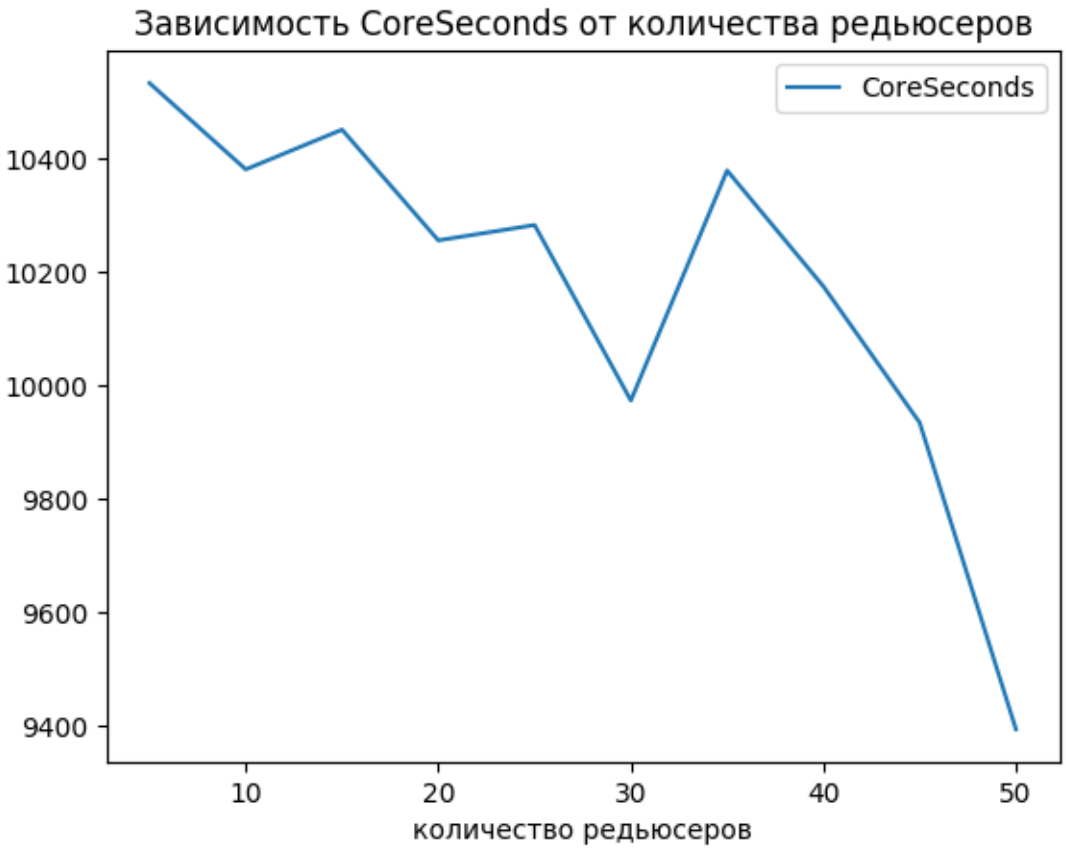
В программе (C) мы отдавали часть работы редьюсера мапперу. Это избавляло от передачи не нужных данных. Хоте это нарушает в чистом виде идею map-reduce, но выигрывает довольно много по времени. (in-mapper combining) Так же при этой стратегии может возникнуть необходимость делать сброс агрегированных данных каждое определенное количество записей.

Вопрос 3.

Какое число редьюсеров используется для каждого из запусков? Почему? (вес 0.3)

Малое количество однозначно плохо, так как каждый редьюсер получит очень много данных и астрономическое время работы будет очень велико. Слишком большое тоже плохо, так как идет не оправданная нагрузка на сервер. В качестве некоторого оптимального значени было выбрано значение 30-35. Увеличение до таких параметров дает прирост по времени, однако дальнейшее увеличение не даст значительного прироста, а лишь увеличит нагрузку на сервис. Для подтверждения были получены графики:





Они подтверждают выбор 30-35 в качестве финального количества редьюсеров.

Вопрос 4.

Насколько хорошо ваше решение будет масштабироваться в случае многократного роста объема входных данных? Не станут ли какие-то этапы узким местом? Нет ли риска переполнения памяти? (вес 0.3)

В качестве примера рассмотрим программу (C), так как она самая оптимальная.

Mapper: Во первых, мы везде действовали из предположения, что каждый текст загружается в память маппера полностью. Если это не так, нужно делать дополнительные ухищрения в маппере. После обработки текста остается collections.Counter, который меньше чем текст. Далее мы сохраняем все слова. Если весь кусок, который обрабатывает маппер, помещается в память, то такое решние масштабируемо. В данном случая получается, что решение масштабируемо, так как память на один маппер фиксирована. Если честно, моё решение имеет риск переполнения памяти и от этого минуса можно избавиться с помошью небольших оптимизаций. Например, использовать более выгодные структуры данных или раз в определенное количество времени частично или полностью сбрасывать полученные агрегированные. Но так как это не критично и не сильно влияет на ответ, я этого делать не стал.

Reducer: В редьюсере в единичный момент времени хранятся не более 40 значений от мапперов (редьюсер последоватеьно просматривает то, что прислали ему маперы) + некоторое константное количество данных на накладные расходы, значит оно полностью масштабируемое.

Аналогично дела обстоят и с масштабируемостью top20. Мы в маппере используем такой же (аналогичный, чуть меньше по значениям) Counter. В редьюсере используем такую же PriorityQueue.