Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Санкт-Петербургский национальный исследовательский
Академический университет Российской академии наук»
Центр высшего образования

Кафедра математических и информационных технологий

Бочаров Федор Александрович

Платформа инкрементальных вычислений на базе MapReduce

Магистерская диссертация

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Омельченко А. В.

Научный руководитель:
Юрченко А. А.

Рецензент:
Романенко Ф. С.

Санкт-Петербург
2017

SAINT-PETERSBURG ACADEMIC UNIVERSITY
Higher education centre

Department of Mathematics and Information Technology

Fyodor Bocharov

Incremental data processing over MapReduce

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Alexander Omelchenko

Scientific supervisor:
Alexander Yurchenko

Reviewer:
Fyodor Romanenko

Saint-Petersburg
2017

Оглавление

Введение

Рассмотрим задачу построения индекса WEB-а [15], который поисковая система могла бы использовать для ответов на запросы пользователей. Для ее решения необходимо обработать большое количество данных, и очевидно, что эффективно решить такую задачу на одном вычислительном узле не представляется возможным, т.к. объемы обрабатываемых данных исчисляются десятками петабайт, поэтому эту и другие задачи обработки больших объемов данных принято решать распределенно.

Для распределенной обработки больших объемов данных в компании Google была разработана модель вычислений под названием MapReduce [8], которая позволяет описать процесс обработки данных как цепочку простых примитивов функциональной парадигмы – функций *map* и *reduce*. Эта модель до сих пор используется в крупнейших компаниях, таких как Google, Яндекс, Microsoft и др. для обработки данных, и, в частности, для построения поискового индекса, но она не лишена недостатков.

Одним из таких недостатков является необходимость повторной обработки всего объема данных при изменении лишь небольшой их части. Ввиду другого недостатка – латентности, MapReduce не позволяет быстро обновлять результат подсчета, что в контексте задачи построения поискового индекса, может негативно отразиться на результатах работы поисковой системы в целом.

Проблема быстрого обновления результатов подсчета при изменении небольшой части данных была менее актуальна 15 лет назад, когда MapReduce только разрабатывался, т. к. данных было значительно меньше, чем сейчас, но со временем их объем увеличивался, и проблема приобретала все бóльшую актуальность.

Одним из решений данной проблемы может быть запуск процесса обработки только на изменившихся данных, и эта идея описана в статье про систему Percolator [5]. При разработке этой системы авторы не использовали модель MapReduce, а предложили способ по-другому представить процесс обработки данных, что позволило решить проблему.

В компании Яндекс по историческим причинам большинство данных хранится и обрабатывается на MapReduce-кластерах, что не позволяет использовать модель Percolator для работы с ними так, как описано в статье. В рамках данной работы эта модель была реализована в модели MapReduce.

1. Обзор предметной области

1.1. MapReduce

В 2004 году компания Google опубликовала статью про парадигму вычислений MapReduce[8] и систему, реализующую эту парадигму.

1.1.1. Описание

Идея этой парадигмы состоит в том, чтобы упростить реализацию процессов распределенной обработки данных за счет описания этих процессов как последовательности вызовов двух стадий: Map и Reduce. Всю работу, связанную с отслеживанием выполнения этих стадий на кластере и перезапуске вычислений при сбоях отдельных вычислительных узлов берет на себя система, реализующая эту парадигму. Пользователю достаточно реализовать функции, которые будут вызываться на каждой из стадий:

- **map**$(k1, v1) \rightarrow$ `list`$(\langle k2, v2 \rangle)$, вызывается на стадии Map, принимает на вход ключ и значение (например, имя документа и его содержимое) и отображающет их в список, возможно, новых пар $\langle$`ключ, значение`$\rangle$
- **reduce**$(k2,$ `list`$(v2)) \rightarrow$ `list`$(v2)$, вызывается на стадии Reduce, принимающет на вход ключ и список значений пар с этим ключом (обычно берутся пары, полученные на стадии Map), и возвращающет список, возможно, других значений

Процесс вычислений устроен таким образом, что система назначает каждому вычислительному узлу часть данных, которые он будет обрабатывать, и, если это стадия Map, узел разбивает эти данные на пары $\langle$`ключ, значение`$\rangle$ и последовательно передает их на вход пользовательской функции *map*. Если узел выполняет стадию Reduce, его задача сгруппировать значения пар с одинаковым ключом, и передать их пользовательской функции *reduce*. Если часть данных с ключом, который обрабатывает узел, находится на другом узле, система организует копирование данных с одного узла на другой. Таким образом на каждой стадии обрабатывается весь объем имеющихся данных. Все вычисления на узлах производятся независимо, и узлы взаимодействуют только при пересылке частей данных в Reduce стадии. Пользовательский код не может обмениваться информацией между узлами.

В данной парадигме действительно легко описать многие процессы обработки данных. Например, процесс подсчета частот слов в корпусе документов может быть описан как цепочка из двух стадий: сначала на стадии Map для каждого слова w в документе будут сгенерированы пары $\langle w, 1 \rangle$, а затем на стадии Reduce для каждого слова будут посчитано, сколько раз данное слово встречалось суммарно во всей коллекции. Эта информация может быть использована в дальнейшем для построения

поискового индекса. В частности, в статье [8] сказано, что реализация построения индекса поисковой системы Google на MapReduce позволила значительно упростить этот процесс.

1.1.2. MapReduce в Яндекс

В Яндекс существует система распределенных вычислений **YT** (Yandex Tables) [7], которая реализует парадигму MapReduce, и помимо этого предоставляет дополнительные операции для работы с данными, такие как сортировка и join. YT состоит из нескольких компонент.

Одной из этих компонент является распределенная файловая система Кипарис, которая обеспечивает отказоустойчивое и надежное хранение и доступ к данным. В отличии от классических распределенных ФС, таких как GFS [6] и ей подобные, Кипарис хранит данные в таблицах. Таблица – это набор строк, состоящий из колонок с данными и, возможно, схема, которая описывает, в какой колонке какой тип данных находится. Дополнительно Кипарис обеспечивает транзакционную работу с таблицами, позволяя читать или изменять несколько таблиц одновременно, но не позволяя одновременно изменять разные строки одной таблицы в разных транзакциях.

Другой важной компонентой является *планировщик*. Его задача состоит в том, чтобы следить за нагрузкой на кластере, обеспечивать честное распределение ресурсов и надежное выполнение операций. Если во время выполнения какой-либо операции часть вычислительных узлов оказывается недоступна, планировщик гарантирует перезапуск вычислений на работающих узлах.

Так же хотелось бы отметить, что в YT отсутствуют единственные точки отказа как в Кипарисе, так и в планировщике, что гарантирует действительно надежное хранение и обработку данных в этой системе.

1.2. Percolator

В 2010 году компанией Google была представлена статья про систему Percolator[5], которая реализует отличную от MapReduce модель вычислений для обработки данных и позволяет обрабатывать данные инкрементально, т. е. запускать вычисления только для изменившихся данных.

1.2.1. Описание системы

Система предоставляет пользователю произвольный доступ к данным, которые хранятся в большой таблице. Это позволяет работать только с измененными данными, не обрабатывая весь объем, как это делается в MapReduce. Для достижения высокой пропускной способности система постоянно сканирует таблицу большим количеством

потоков, и, если обнаруживает новые данные в пользовательских колонках, запускает пользовательские функции-обработчики, которые «подписаны» на эти колонки. Поскольку запуск пользовательского кода происходит многопоточно, Percolator предоставляет ACID-транзакции [9] с уровнем изоляции snapshot [17]. Таким образом, Percolator предоставляет пользователю две абстракции: ACID-транзакции и модель подписок на изменения данных.

Типичное приложение в такой модели состоит из нескольких функций обработчиков, которые связаны между собой следующим образом: какое-то внешние изменение таблицы (например, была скачана web-страница, и ее содержимое сохранилось в какую-то из колонок) инициирует запуск первого обработчика. После его работы изменяется содержимое в каких-то других колонках таблицы, что вызывает запуск следующих функций-обработчиков, и т. д. В результате только новые данные проходят через каскад обработчиков, не вызывая пересчет состояния той части данных, которая не менялась.

1.2.2. Дизайн системы

В качестве таблицы с данными Percolator использует другую разработку Google – NoSQL-СУБД [18] BigTable [3]. Это распределенная СУБД, которая представляет из себя KV-хранилище и позволяет надежно хранить данные и обращаться к ним по составному ключу вида ⟨`ключ, колонка, время`⟩. BigTable позволяет атомарно читать и обновлять строки в базе (строка – все ключи с одинаковой первой компонентой), но не предоставляет возможность транзакционно читать и обновлять несколько строк одновременно.

Percolator состоит из трех приложений, запущенных на каждом узле кластера: исполняемый файл Percolator, с которым слинкованы пользовательские функции-обработчики, исполняемый файл сервера таблетов BigTable [3] и исполняемый файл сервера данных GFS [6]. Во время работы Percolator вызывает функции-обработчики, которые делают RPC-запросы для работы с данными к серверу таблетов BigTable, который, в свою очередь, обращается к серверу данных GFS. Так же в системе присутствует сервис блокировок и компонента, предоставляющая монотонно возрастающую последовательность чисел, которые используются как временны́е метки в транзакциях.

Для пользователя данные выглядят как набор из нескольких небольших таблиц, данные в которых адресуются парой ⟨`ключ, колонка`⟩. Данные представляют из себя нетипизированный набор байт, и ответственность за корректную с точки зрения типов работу с ними ложится на пользователя.

1.2.3. Реализация транзакций

Percolator предоставляет многострочные ACID транзакции с уровнем изоляции snapshot. Недостатком snapshot изоляции является то, что такой подход не предоставляет строгую сериализуемость транзакций, и подвержен такой аномалии, как *write skew* [17], когда прочитанные данные одной транзакции параллельно изменяются другой транзакцией, и обе транзакции завершаются успешно. Из достоинств можно отметить, что для чтения данных при этом уровне изоляции не требуются блокировки, т. к. транзакция всегда читает неизменяемый снимок данных. Так же snapshot изоляция не допускает *write-write* конфликтов, т. е. если две транзакции одновременно пытаются изменить одни и те же данные, хотя бы одна из транзакций завершится неуспешно.

Для реализации транзакции используются транзакционные возможности BigTable и дополнительные блокировки, которые Percolator поддерживает явно, в отличии от других СУБД, где блокировки обычно реализуются в компоненте, отвечающей за доступ к данным на диске, поскольку Percolator обращается к BigTable через ее программный интерфейс. Информация об этих блокировках хранится в BigTable в специальных колонках, наравне с пользовательскими данными.

Сами транзакции устроены следующим образом: каждой транзакции присваивается временна́я метка t – время ее начала. Далее транзакция совершает чтения и записи данных, и, если не произошло конфликтов, транзакция завершается успешно, а ее данные сохраняются в базу. Записи дополнительно буферизируются и сохраняются одновременно в конце транзакции.

Для атомарного сохранения всех изменений, которые совершает транзакция, в Percolator используется алгоритм *two-phase commit* [12], который координируется узлом, инициирующем изменение. На первой фазе происходит проверка отсутствия конфликтов и блокировка данных, которые будут изменены. На второй фазе происходит изменение данных и снятие блокировок, что позволит другим транзакциям увидеть и прочитать обновления.

При чтении выбирается снимок данных с временно́й меткой t' такой, что $t' \leq t$. Если в служебной колонке есть информация о том, что эти данные заблокированы, и блокировка совершена в момент времени t'' такой, что $t'' \leq t$, то это значит, что конкурентная транзакция в данный момент осуществляет запись данных, и текущая транзакция должна дождаться окончания этой записи.

1.2.4. Механизм подписок

В Percolator пользователь реализует функции-обработчики, которые «подписываются» на различные колонки в базе, и вызываются, когда данные в колонках, на которые они подписаны, обновляются. Эта идея схожа с идеей триггеров в реляци-

онных СУБД [13], но различие состоит в том, что в Percolator обновление данных и запуск обработчика происходит в разных транзакциях. В Percolator механизм подписок используется для удобства работы с системой, тогда как в реляционных СУБД триггеры являются инструментом обеспечения целостности данных.

Для эффективно поиска обновленных данных Percolator поддерживает специальную колонку с уведомлениями об изменениях. Когда какая-либо часть данных обновляется, информация о том, какая часть именно обновилась попадает в колонку с уведомлениями. Это позволяет сократить время поиска измений и запуска обработчиков. После того, как транзакция с обработчиком завершилась успешно, запись из колонки с уведомлениями удаляется.

Percolator гарантирует, что при обновлении данных обработчики будут запущены не более одного раза. Это обеспечивается за счет того, что для каждой колонки C в базе хранится дополнительная служебная колонка C_write, в которой хранится временна́я метка транзакции, в которой был запущен обработчик, подписанный на изменения колонки C. Если происходит ситуация, при которой несколько транзакций запустились одновременно после обновления, успешно завершится только одна, а результаты остальных будут отменены в силу *write-write* конфликта при записи в колонку C_write.

2. Постановка задачи

Поскольку Percolator позволяет удобно строить процессы инкрементальной обработки данных, представляется интересной идея его реализации на MapReduce, потому что:

- Большинство данных в компании Яндекс хранится на MapReduce-кластерах, поэтому работать с ними логичнее и проще на этих же кластерах
- Результаты вычислений могут быть использованы в дальнейшем для обработки другими MapReduce-процессами
- Использование общего MapReduce-кластера и отлаженной работающей платформы распределенных вычислений позволит сэкономить большое количество ресурсов

Поэтому целью данной работы является разработка платформы, реализующей модель вычислений Percolator в модели MapReduce.

Для достижения поставленной цели были выделены следующие задачи:

- Проанализировать, какие ограничения накладывает модель MapReduce на реализацию Percolator в этой модели
- Реализовать и протестировать платформу инкрементальных вычислений на MapReduce
- Обеспечить внедрение и поддержку разработанной платформы

3. Анализ Percolator в модели MapReduce

Поскольку главными идеями в Percolator являются ACID-транзакции и механизм подписок, в первую очередь необходимо понять, эти идеи можно реализовать на MapReduce.

Рассмотрим еще раз, как реализованы транзакции в Percolator. Каждая транзакция читает и пишет строки в BigTable, причем делает это с транзакционными гарантиями, предоставляемыми СУБД. Это позволяет обеспечить упорядоченный доступ к отдельным строкам в базе, а при дополнительной работе, которую проделывает Percolator, позволяет упорядочить доступ и ко множеству строк.

Еще одной отличительной чертой транзакций в Percolator является то, что функции-обработчики, запущенные в них, пишут и читают данные по произвольным ключам. Быстрый доступ к данным по ключу так же обеспечивается BigTable.

Механизм подписок и гарантии, которые он предоставляет, так же основаны на возможности BigTable транзакционно читать и обновлять строки базы.

Таким образом, можно выделить следующие возможности BigTable, на которые опирается описанная реализация Percolator:

- Возможность быстро читать и обновлять произвольные строки (а значит и ключи) в базе
- Транзакционное чтение и обновление строк в базе

Теперь рассмотрим особенности модели MapReduce на примере системы YT. Операции Map и Reduce представляют из себя последовательное чтение и одновременную обработку данных. Новые данные обычно дописываются в конец к существующим таблицам, либо порождают новые таблицы. В связи с этим для хранения данных используется специальная файловая система, оптимизированная для таких шаблонов использования.

Другой особенностью этой модели является то, что транзакции работают с целыми таблицами, а не с отельными их строками, поэтому невозможно одновременно изменить разные строки одной таблицы.

Таким образом можно выделить следующие особенности реализации MapReduce в YT:

- Оптимизация хранилища данных для последовательного доступа
- Транзакционность на уровне целых таблиц, но не их частей

Хотелось бы отметить, что это не особенности конкретной реализации, а особенности модели в целом. Например, GFS так же оптимизирована для последовательного доступа к файлам.

Поскольку Percolator основывается на BigTable, используя ее в качестве хранилища данных, замена этой технологии на файловую систему, используемую для MapReduce, которая не предоставляет требуемую функциональность, делает невозможной реализацию Percolator так, как описано в статье. Поэтому для реализации Percolator на MapReduce требуется решить следующие проблемы:

- Обеспечение возможности чтения произвольных ключей
- Обеспечение транзакционной работы с подмножеством ключей

Первая проблема возникает из-за того, что запуск пользовательских функций-обработчиков реализуется с помощью стадий Map, или Reduce, или их комбинаций. Поскольку на каждой стадии в один момент времени вычислительный узел работает с данными только одного ключа, требуется каким-то образом предоставить ему данные других ключей.

Вторая проблема возникает из-за того, что Кипарис не поддерживает построчные транзакции, тогда как это очень важная часть в модели Percolator, потому что она позволяет пользователю не думать о консистентности данных – констистентность обеспечивается системой за счет того, что все изменения происходят в транзакциях.

4. Описание платформы

В данном разделе приведено описание платформы на уровне идей, без описание деталей реализации – они описаны в следующих разделах.

4.1. Модель вычислений

Разработанная платформа полностью реализует модель вычислений Percolator, добавляя некоторые дополнительные возможности.

Логически, данные, так же как и в Percolator, представляют из себя таблицу, в ячейках которой по ключу ⟨`ключ, колонка`⟩ находятся пользовательские данные. Там может находиться как обычное значение, такое как число или строка, так и словарь. Во втором случае для обращения к значениям используется ключ вида ⟨`ключ, колонка, подключ`⟩, где `подключ` – это ключ в словаре лежащем в ячейке ⟨`ключ, подключ`⟩. В отличии от Percolator, который хранит предоставляет доступ к данным как к массиву байт, в разработанной платформе есть возможность типизированной работы с данными, что уменьшает вероятность ошибки работы с типами.

Основные вычисления происходят в *триггерах* – пользовательских функциях-обработчиках, которые вызываются, когда обновляется какая-то часть данных. Для работы триггеру передается ключ, данные которого изменились, и *контекст* – объект, который позволяет читать и обновлять данные в таблице.

Например, так мог бы выглядеть триггер, который подсчитывает ранк веб-страницы, основываясь на рангах страниц, ссылающихся на нее:

```
1  def count_rank(url, ctx):
2    new_rank = 0
3    for link, rank in ctx.iter_float(url, "incomming_link_rank"):
4      new_rank += rank
5
6    old_rank = ctx.get_float(url, "rank")
7    if new_rank != old_rank:
8      ctx.set_float(url, "rank", new_rank)
9      for link in extract_links(ctx.get_str(url, "content")):
10       ctx.set_float(link, "incomming_link_rank", url, new_rank)
```

Листинг 1: пример триггера

В данном случае `url` – это ключ, данные которого изменились, а `ctx` – *контекст*. На строках 3-4 происходит итерация по входящим ссылкам и суммируются их ранги. Далее, в строках 6-10, происходит обновление ранга, если новый ранг не равен старому. Функция `extract_links` извлекает ссылки из веб-страницы и предоставляет итератор по ним.

Для запуска триггеров так же, как и в Percolator, применяется модель подписок и уведомлений, т. е. как только в какой-то колонке обновляются данные, платформа запускает триггеры, «подписанные» на нее, причем вычисления запускаются не на всем объеме данных, а только на изменившихся.

Обновления данных может быть следствием того, что новые значения были импортированны из каких-либо таблиц Кипариса, либо какой-то триггер произвел запись в базу.

Таким образом типичное приложение в разработанной платформе выглядит как набор триггеров, которые каскадно вызываются и обрабатывают данные.

4.2. Транзакции

Разработанная платформа, так же, как и Percolator, обеспечивает ACID-транзакции с уровнем изоляции snapshot, в которых работают пользовательские триггеры. В Percolator для фиксации изменений используется алгоритм *two-phase commit*, и, поскольку Percolator использует BigTable, которая является OLTP-системой [14], этот алгоритм имеет хорошую производительность. Важной особенностью является то, что при блокировке изменяемых данных на первой фазе одной транзакцией, другая сразу же «видит» эти изменения и завершается неудачей. Если использовать этот же алгоритм при реализации транзакций в модели MapReduce, возникает проблема с тем, что пользовательский код не может обмениваться инфомацией между разными узлами. Таким образом, если одна транзакция на одном вычислительном узле осуществляет блокировку изменяемых данных, транзакция на другом узле, работающая с этими же данными, не узнает об этом, что приведет к состоянию гонки. Для того, чтобы исправить эту ситуацию необходимо проводить дополнительную стадию вычислений (дополнительный вызов *map* или *reduce*), что значительно понизит производительность платформы.

Помимо *two-phase commit* есть еще несколько подходов к реализации транзакций в классических транзакционных системах:

1. Timestamp ordering [11]
2. Optimistic locking [10]
3. Deterministic transactions [4]

Все эти подходы поддерживают уровень изоляции *serializable* [16], но поскольку в разработанной платформе гарантии менее строгие, идеи этих подходов приобретают более простой вид. Рассмотрим подробнее каждую из них.

Идея алгоритма *timestamp ordering* состоит в том, что транзакции образуют очередь, и каждой транзакции в очереди присваивается уникальная временна́я метка.

Далее происходит выполнение транзакций из очереди, причем транзакция с бóльшей временнóй меткой не может выполниться позже транзакции с мéньшей временнóй меткой. Понятно, что при таком подходе с уровнем изоляции snapshot не будет конфликтов, поскольку в один момент времени будет исполняться только одна транзакция, а значит никаке две транзакции не будут одновременно менять одни и те же данные.

Идея алгоритма *optimistic locking* состоит в том, что каждая транзакция T выполняется независимо от остальных, читая данные без блокировок, а при фиксации изменений менеджер транзакций проверяет, что прочитанные транзакцией T данные не изменились за время ее работы, и если записи T не конфликтуют с другими транзакциями, то T успешно завершается. При snapshot изоляции транзакции всегда читают данные без блокировок, поэтому данная оптимизация теряет смысл.

Алгоритм *deterministic transactions* появился относительно недавно, и он описан в статьях [4] [1] [2]. В некотором смысле он объединяет два предыдущих подхода. Его идея состоит в том, чтобы разбить все транзакции на небольшие группы и выполнить каждую из них по отдельности. Внутри группы транзакции выполняются, читая данные без блокировок, как в *optimistic locking*. После этого менеджер транзакций получит информацию о том, какие данные транзакция прочитала и какие данные записала. На основании этих данных менеджер транзакций назначает каждой транзакции временнýю метку – момент времени, в который изменения транзакции должны быть зафиксированы, причем пытается сделать это таким образом, чтобы увеличить число успешно завершившихся транзакций. Например, если есть транзакция T_1, которая прочитала данные из ячейки A, и есть транзакция T_2, которая записала данные в эту ячейку, чтобы обе успешно завершились, временнáя метка T_1 должна быть меньше T_2, тогда T_1 прочитает актуальные данные. Понятно, что при snapshot изоляции читаемые данные не меняются, и менеджеру транзакций достаточно назначить различные временны́е метки для транзакций, чтобы между ними не возникало конфликтов.

В разработанной платформе выбран именно подход *deterministic transactions*, поскольку он лишен недостатка подхода *two-phase locking*. Так же с помощью этого подхода возможно реализовать транзакции с уровнем изоляции serializable – достаточно упорядочить доступ к каждой ячейке таблицы, что можно сделать за одну стадию MapReduce, и тогда возможно получить некоторый глобальный порядок для всех транзакций. В первых же двух подходах потребуется передавать данные между узлами, как и в *two-phase locking*, что понизит производительность.

Сами транзакции реализованы следующим образом: каждой транзакции назначается уникальный идентификатор перед ее выполнением. Далее в транзакции запускается триггер, которые читает правильный снимок базы и подготавливает набор записей, которые должны быть сохранены в базу. После этого выполняется операция *медиации* транзакций, где проверяется, что среди всех транзакций нет конфликту-

ющих, и каждой транзакции назначается временна́я метка – момент времени, в который записи этой транзакции должны быть сохранены в базу. Всем транзакциям назначается одинаковая временная метка, чтобы не возникло ситуации, когда триггер в транзакции читал устаревший снимок данных. Если какие-то транзакции конфликтуют, для сохранения из конфликтующих транзакций выбирается транзакция с наибольшим идентификатором, а остальные перезапускаются и проходят повторную медиацию. После медиации выполняется операция сохранения данных в базу. В этой операции создается новая порция данных, которые будут использованы в вычислениях в дальнейшем, а так же для тех триггеров, транзакции которых считаются успешно завершенными, сохраняется информация о том, что эти триггеры выполнились, и обработали новые данные. Для триггеров, чьи транзакции должны быть перезапущены, и триггеров, которые должны быть запущены из-за того, что произошла запись в базу, создаются уведомления.

5. Реализация платформы

5.1. Дизайн платформы

Разработанная платформа состоит из нескольких компонент:

- Основная библиотека для работы с YT
- Движок, запускающий вычисления
- Дополнительные библиотеки для написания триггеров

Основная библиотека определяет формат, в котором хранятся данные, а так же в ней реализованы основные операции, такие как обработка новых данных и запуск триггеров. Она может быть использована другими приложениями, которые так же хранят данные в Кипарисе в определяемом библиотекой формате и используют YT для вычислений.

Движок представляет из себя многопоточное приложение, которое запускает по расписанию задачи. Задачи бывают разных типов, например импорт данных из внешних по отношению к платформе таблиц Кипариса, запуск обработки новых данных, запуск триггеров и т. д. Так же в движке есть специальный тип задач, который позволяет запускать произвольный пользовательский исполняемый файл с заданной периодичностью. Это бывает полезно, когда данные должны быть дополнительно обработаны перед импортированием во внутренний формат платформы.

В дополнительных библиотеках описан программный интерфейс, который должны реализовывать пользовательские триггеры. В данный момент платформа предоставляет писать триггеры на языках Python и C++, но этот список может быть достаточно просто расширен. Подробнее запуск триггеров описан в следующих разделах.

Для того, чтобы запустить вычисления на платформе, пользователь должен создать 3 файла:

- Файл со схемой таблицы, в которой хранятся пользовательские данные. В этом файле описан набор колонок, которые есть в таблице. Каждая колонка описывается именем и типом данных, который в ней лежит. Так же в этом файле присутствует описание триггеров. Каждый триггер описывается названием, списком колонок, на которые он «подписан» и *хостом*, в котором он будет запускаться
- Файл с конфигурацией движка. В этом файле описываются задачи и интервал их запуска. Так же там может быть описана конфигурация для отдельных задач
- Библиотека с триггерами

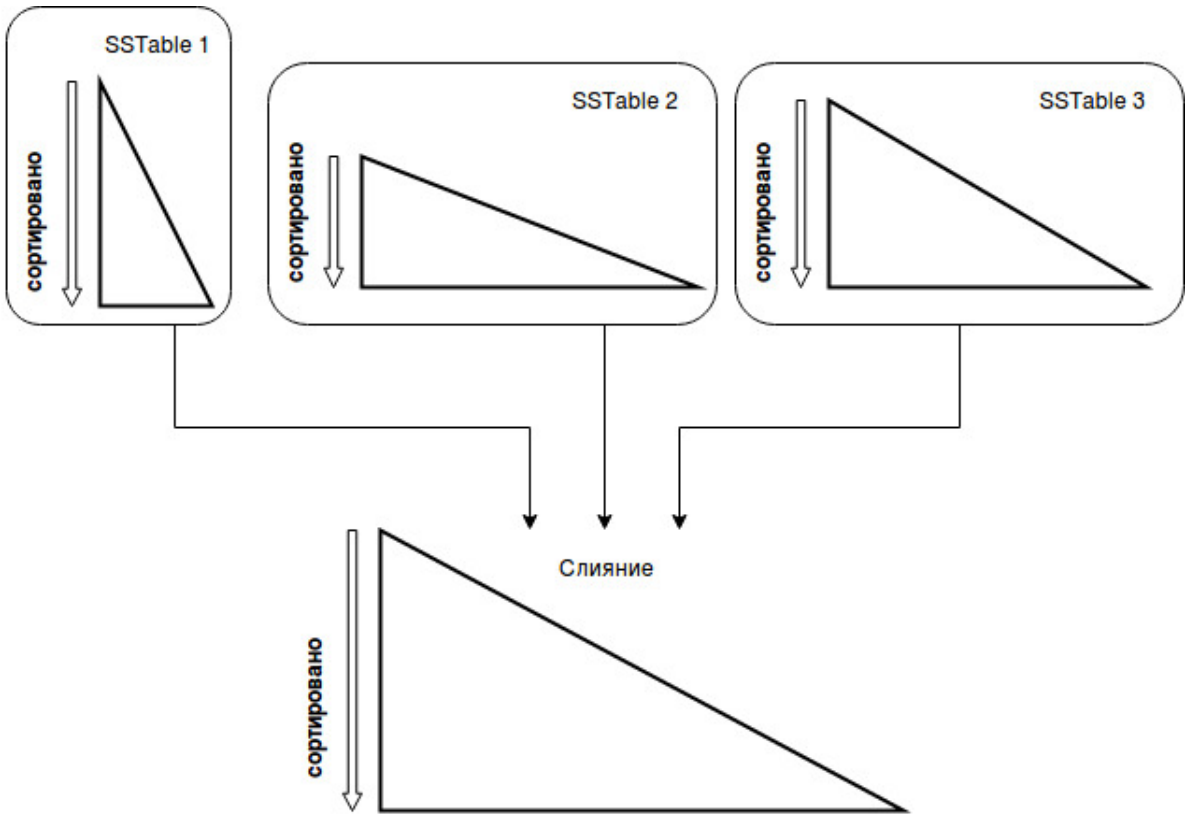

Рис. 1: LSM-дерево

5.2. Хранение данных

Данные базы хранятся в таблицах Кипариса, строки которых имеют формат:

⟨`Key, Column, Subkey, Value, Timestamp`⟩, где

Key – ключ, который пользователь указывает при обращении к данным

Column – колонка, которую пользователь указывает при обращении к данным

Subkey – подключ, который пользователь указывает при обращении к данным. Пустая строка, если значение в ячейке ⟨`Key, Column`⟩ не является словарем

Value – пользовательские данные, хранятся как массив байт

Timestamp – временна́я метка транзакции, которая записала эти данные в базу

Данный формат выбран потому, что он позволяет описать базу с произвольным набором колонок, и при его использовании можно достаточно просто получить данные всех колонок со всеми подключами для одного ключа – достаточно выполнить стадию Reduce и в качестве ключа указать системе поле `Key`. Это используется в операции запуска триггеров.

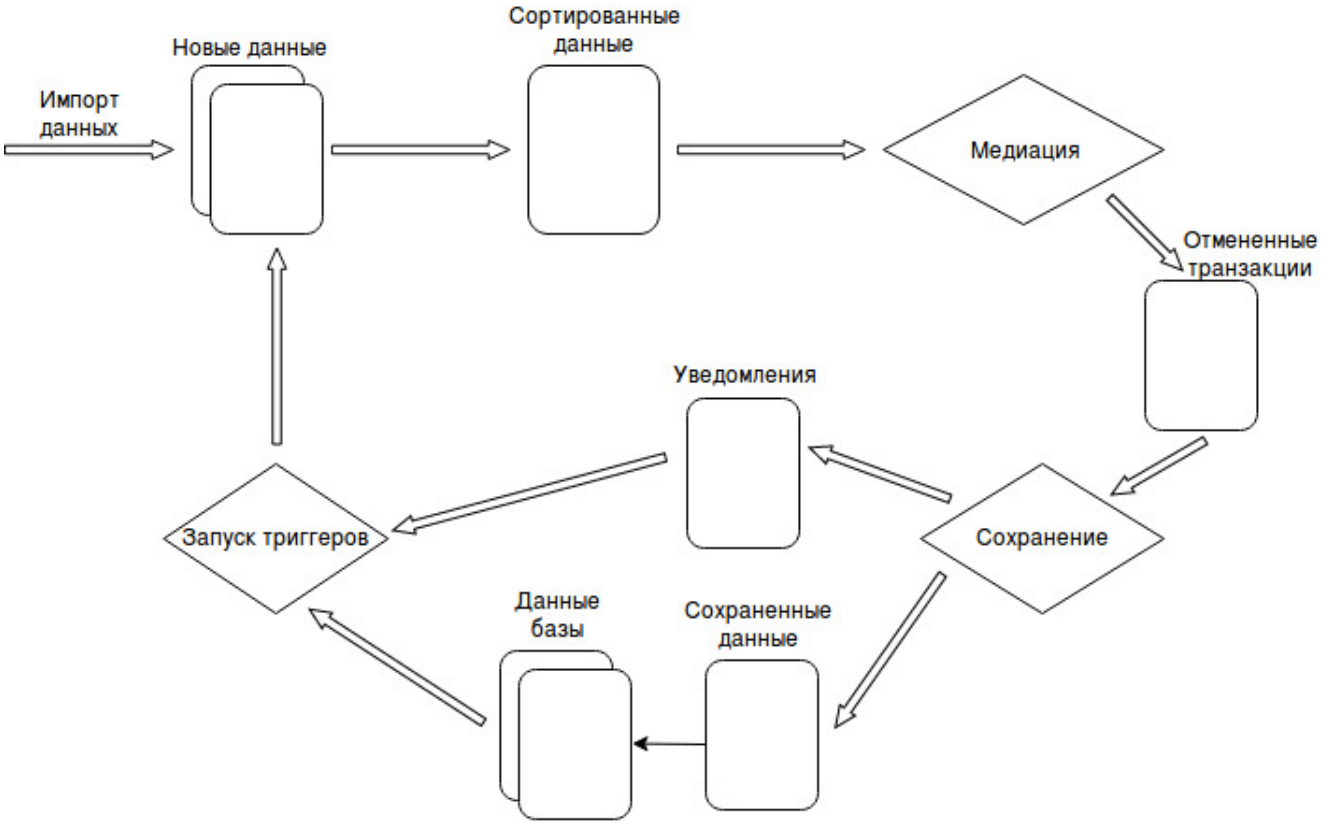

Рис. 2: процесс обработки данных

Каждая таблица является SSTable и сортирована по ⟨`Key, Column, Subkey`⟩ по возрастанию, а все таблицы вместе образуют структуру данных LSM-дерево [19]. Такой подход позволяет при записи новых данных не изменять уже существующие, а только добавлять новую таблицу в список таблиц базы, дополнительно сортируя ее, если она не сортирована. Далее, при выполнении операций с базой, например при запуске триггеров, YT автоматически выполнит слияние этих таблиц (см. Рис. 1) и предоставит список записей с заданными ключей.

Отдельно от таблиц с данными хранится информация о том, какие триггеры должны быть выполнены в связи с изменением данных в колонках, на которые они «подписаны». Эти данные так же лежат в сортированных таблицах с тем же форматом, и используются при запуске триггеров. Строки в этих таблицах имеют специальное значение колонки `Column`, что позволяет отличать их от пользовательских записей, когда происходит сохранение данных в базу или запуск триггеров.

Новые данные хранятся отдельно в несортированных таблицах, но с тем же форматом. Перед работой с ними, они дополнительно сортируются, после чего их можно использовать в Reduce.

5.3. Обработка новых данных

Процесс обработки данных реализован в основной библиотеке в виде нескольких операций, запускаемых на YT, каждая из этих операций является *reduce*-операцией. Сам процесс изображен на Рис. 2.

Все новые данные, которые появляются либо после имортирования таблиц Кипариса, либо после того, как триггеры запишут что-то в базу, сохраняются во внутреннем формате платформы. После этого, с периодичностью, описанной в конфигурации, движок запускает операцию обработки новых данных. Она состоит из двух частей: медиация транзакций и сохранение результатов успешных транзакций в базу.

Медиация транзакций представляет из себя Reduce стадию с ключем ⟨`Key, Column`⟩ таблицы с сортированными новыми данными. Это позвляет проверить, что не существует больше одной транзакции, которая пишет в ячейку базы ⟨`Key, Column`⟩, но, если все-таки существует несколько таких транзакций, выбрать ту, результаты которой будут сохранены, а результаты остальных отменить. Для упрощения реализации в качестве временно́й метки транзакций берется время начала операции.

Операция сохранения данных так же является Reduce стадией с тем же ключом, но помимо таблицы с данными в нее поступают уведомления для триггеров, которые должны были быть запущены на предыдущей итерации, а так же информация о том, результаты каких транзакций были отменены. Операция проверяет, что триггер отработал успешно и его транзакция завершилась успешно, и, если так, то создает новую порцию данных, которые будут храниться в базе. Если же триггер отработал неуспешно или его транзакция завершилась неуспешно (записи транзакции были отменены), то для него повторно создается уведомление. Помимо сохранения в базу, записи успешно завершившихся транзакций копируются в *кэш ключа*, при изменении которого триггер пытался прочитать текущие данные. Если таких ключей несколько, данные копируются в кэши всех ключей. Данные о том, какие ключи читали текущую ячейку хранятся отдельно от базы, и при выполнении операции сохранения учавствуют в Reduce. Это нужно для того, чтобы обеспечить корректное чтение чужих ключей. При таком подходе данные в кэшах синхронно обновляются с данными в базе, поэтому когда триггер попытается прочитать данные чужого ключа, он всегда увидит актуальную копию данных. Кэш ключа хранится с данными в специальной колонке, чтобы его можно было отличить от данных и корректно восстановить при запуске триггеров.

5.4. Интерфейс триггеров

В данный момент платформа обеспечивает возможность написания триггеров на двух языках: C++ и Python.

Триггеры на языке C++ собираются в разделяемую библиотеку. Для того, чтобы

написать триггер на C++, пользователь должен отнаследоваться от специального интерфейса, который определен в дополнительных библиотеках платформы: и воспользоваться специальным макросом для того, чтобы этот триггер можно было найти в библиотеке при попытке его запустить.

```
class ITrigger {
public:
    virtual void Execute(
        const std::string& key,
        TTriggerContext& ctx) = 0;
};
```

Листинг 2: интерфейс C++ триггера

Триггеры на языке Python собираются в специальный исполняемый файл. Пример триггера на Python уже был продемонстрирован в Листинге 1. Для того, чтобы создать исполняемый файл с триггерами на Python, пользователю достаточно написать триггеры и передать их в специальный класс, который будет их вызывать. Этот класс реализует RPC-протокол, через который передаются разные команды для запуска триггеров и возвращается результат их работы.

5.5. Запуск триггеров

После того, как данные обновились, происходит запуск триггеров на обновившихся данных. Это так же, как и обработка новых данных, происходит на Reduce стадии с ключом ⟨Key⟩. За счет этого триггеры запускаются только на тех ключах базы, которые действительно обновились. Эта операция работает с таблицами, хранящими данные базы и таблицой, в которой находится информация о триггерах, которые должны быть запущены. Кроме этого в операцию передаются файлы с триггерами, которые пользователь указал в конфигурации.

После запуска операция читает все записи, которые ей передает система и аггрегирует их, определяет, какие записи указывают на то, какие триггеры запускать, и запоминает это. Записи с данными базы и записи, хранящие кэш ключа аггрегируются по временно́й метке так, что остается только запись с самой большой временно́й меткой, и сохраняются в большой словарь – контекст.

После того, как операция подготовила контекст, триггеру происходит запуск триггеров. Для каждого триггера известно, какой *хост* должен его запускать. Хост – это некоторый процесс, который может подготовить среду для запуска триггеров, и затем производить их запуск. Хост для триггеров на C++ загружает файл с триггерами на C++, который указал пользователь, как разделяемую библиотеку, и находит там триггеры, которые надо запустить. Хост для триггеров разделен на две части: одна

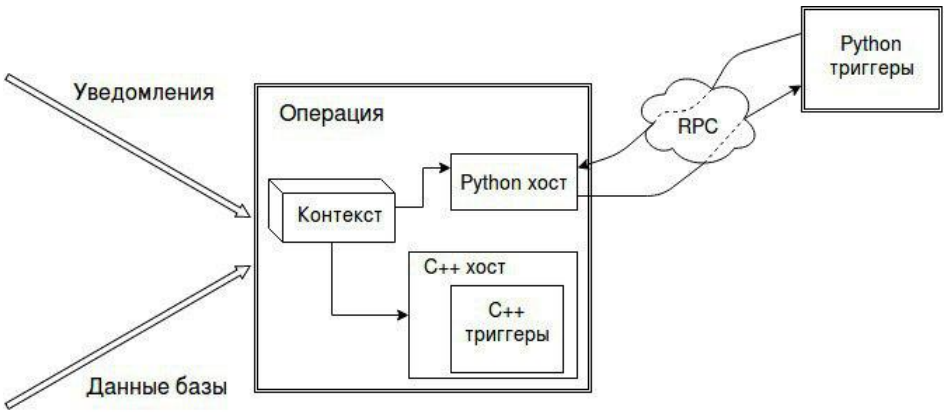

Рис. 3: запуск триггеров

из них находится в контексте процесса, в котором запущена операция, а другая – исполняемый файл с триггерами, который запускается в дочернем процессе. Эти части обмениваются сообщениями по RPC-протоколу.

Во время работы триггеров специальный поток в процессе операции контролирует использование памяти в хостах, и, если хост превысил определенный предел, его аварийно завершают и запускают снова, но ключ, который привел к повышенному потреблению памяти больше не используется для запуска триггеров.

Триггер считается успешно отработавшим, если во время его работы хост не был аварийно завершен, и триггер не вернул ошибку или бросил исключение. В этом случае его записям присваивается уникальный идентификатор транзакции – число, которое получается комбинацией временно́й метки начала операции (она присваивается перед началом операции) и номера ключа в таблице с уведомлениями, и они сохраняются в платформе для дальнейшей обработки. Помимо этого, при успешном завершении триггера, в специальную колонку вместе с записями сохраняется информация о том, что триггер завершился успешно, чтобы в дальнейшем можно было понять, что этот триггер больше не нужно запускать.

В случае, когда триггер завершается неуспешно, информация об этом записывается в лог, чтобы пользователь мог увидеть это и предпринять меры. Однако триггер может завершиться неудачно, если попытается прочитать ключ, которого нет в кэше. Эту ситуацию нельзя считать ошибочной, поскольку она возникла не из-за ошибок в коде, а по вине платформы. В этом случае в специальных таблицах Кипариса сохраняется информация о ключе, при работе с которым не удалось прочитать данные, и ячейках, которые не удалось прочитать. Перед запуском триггеров эта информация используется для того, чтобы пополнить кэши ключей. Так же эта информация дописывается к ячейкам, чтобы при изменении их значений, их состояние было скопировано в кэш текущего ключа.

5.6. Дополнительные возможности платформы

Помимо основных операций с данными платформа предоставляет дополнительные возможности, полезные для ее эксплуатации.

Одной из таких возможностей является возможность *компактификации* данных. Эта функция позволяет контролировать размер базы, и регулярно удалять из базы слишком старые снимки данных. Пользователь может настроить процесс компактификации как для всей базы, так и для отдельных ее колонок. Для этого ему достаточно указать в конфигурации «максимальный возраст» данных в базе, и автоматическая задача, запускаемая движком, будет следить за тем, чтобы возраст данных в базе не превышал это значение. Под возрастом данных понимается разница между значением времени на момент запуска операции и времени сохранения данных в базу.

Другой полезной возможностью является *сборка мусора*. Операции работы с базой не удаляют данные, с которыми они работают. Например, операция сохранения новых данных в базу не удаляет таблицы с новыми данными, и они хранятся в системе. Это нужно, чтобы в случае каких-либо проблем, возникших в ходе работе платформы или триггеров, можно было отследить на каких данных они произошли, и устранить их. Если все работает без ошибок, то таблицы просто занимают место. Сборка мусора позволяет удалять слишком старые таблицы, что позволяет значительно сэкономить используемое место. Пользователь так же может настроить максимальный возраст таблиц.

Помимо этих возможностей платформа предоставляет различные мониторинги, которые позволяют пользователю в реальном времени наблюдать за состоянием платформы и замечать аномалии. Так, например, можно посмотреть, как долго работают триггеры, и как много памяти они используют. Кроме мониторингов самой платформы есть возможность сделать пользовательские мониторинги. Для этого пользователь должен создать в схеме специальную колонку, по данным из которой платформа будет строить графики. Например, в задаче построения индекса с помощью этого можно увидеть, сколько документов в данный момент проиндексировала система.

6. Заключение

В рамках данной работы был проведен анализ модели Percolator в рамках модели MapReduce, и выявлены проблемы, не позволяющие реализовать систему так, как описано в статье. Эти проблемы были успешно решены, и была разработана платформа инкрементальных вычислений на MapReduce.

Данная платформа уже используется в нескольких проектах компании Яндекс. Несколько из них сразу были реализованы с ее использованием, а один изначально был написан на MapReduce, а впоследствии переписан на инкрементальную обработку. В силу того, что он обрабатывал не слишком большие объемы данных, значительного ускорения добиться не удалось, но удалось значительно повысить надежность и удобство разработки, за счет предоставляемых платформой абстракций, что важно для развития проекта.

Таким образом, все поставленные задачи выполнены и цель достигнута. В будущем планируется развитие платформы, реализация более строгого уровня изоляции транзакций, развитие пользовательского интерфейса и работа над отказоустойчивостью платформы.

Список литературы

[1] Alexander Thomson, J. Abadi Daniel. The Case for Determinism in Database Systems, VLDB. — 2010. — URL: http://db.cs.yale.edu/determinism-vldb10.pdf.

[2] Alexander Thomson, J. Abadi Daniel. Rethinking serializable multiversion concurrency control (Extended Version), Yale. — 2015. — URL: https://arxiv.org/pdf/1412.2324.pdf.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat et al. Bigtable: A Distributed Storage System for Structured Data, Google, Inc. — 2006. — URL: https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf.

[4] Alexander Thomson, Thaddeus Diamond, Shu-chun Weng et al. Calvin: Fast Distributed Transactions for Partitioned Database Systems, SIGMOD. — 2012. — URL: http://cs.yale.edu/homes/thomson/publications/calvin-sigmod12.pdf.

[5] Daniel Peng, Frank Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications, Google, Inc. — 2010. — URL: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/36726.pdf.

[6] Ghemawat Sanjay, Gobioff Howard, Leung Shun-Tak. The Google File System. — 2003. — URL: https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf.

[7] Habrahabr. YT: зачем Яндексу своя MapReduce-система и как она устроена. — 2016. — URL: https://habrahabr.ru/company/yandex/blog/311104/.

[8] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters, Google, Inc. — 2004. — URL: https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf.

[9] Wikipedia. ACID // Википедия, свободная энциклопедия. — 2002. — URL: https://en.wikipedia.org/wiki/ACID.

[10] Wikipedia. Optimistic concurrency control // Википедия, свободная энциклопедия. — 2003. — URL: https://en.wikipedia.org/wiki/Optimistic_concurrency_control.

[11] Wikipedia. Timestamp-based concurrency control // Википедия, свободная энциклопедия. — 2003. — URL: https://en.wikipedia.org/wiki/Timestamp-based_concurrency_control.

[12] Wikipedia. Two-phase commit protocol // Википедия, свободная энциклопедия. — 2004. — URL: https://en.wikipedia.org/wiki/Two-phase_commit_protocol.

[13] Wikipedia. Database trigger // Википедия, свободная энциклопедия. — 2005. — URL: https://en.wikipedia.org/wiki/Database_trigger.

[14] Wikipedia. Online transaction processing // Википедия, свободная энциклопедия. — 2005. — URL: https://en.wikipedia.org/wiki/Online_transaction_processing.

[15] Wikipedia. Search engine indexing // Википедия, свободная энциклопедия. — 2006. — URL: https://en.wikipedia.org/wiki/Search_engine_indexing.

[16] Wikipedia. Serializability // Википедия, свободная энциклопедия. — 2006. — URL: https://en.wikipedia.org/wiki/Serializability.

[17] Wikipedia. Snapshot isolation // Википедия, свободная энциклопедия. — 2006. — URL: https://en.wikipedia.org/wiki/Snapshot_isolation.

[18] Wikipedia. NoSQL // Википедия, свободная энциклопедия. — 2009. — URL: https://en.wikipedia.org/wiki/NoSQL.

[19] Wikipedia. Log-structured merge-tree // Википедия, свободная энциклопедия. — 2013. — URL: https://en.wikipedia.org/wiki/Log-structured_merge-tree.

[20] Wikipedia. K-Way Merge Algorithms // Википедия, свободная энциклопедия. — 2015. — URL: https://en.wikipedia.org/wiki/K-Way_Merge_Algorithms.