Аннотация

Работа посвящена разработке инструмента для эффективного запуска графов обработки данных на MapReduce кластерах. Итоговый компонент - executor для библиотеки описания pipeline-ов Roren, исполняющий код поверх кластера YT.

Содержание

1. Введение

Обработка данных является важной задачей для крупных компаний. Такого рода задача решается с помощью написания процессингов данных. Код с помощью базы данных описывает некоторый граф или, иначе, pipeline. По нему данные перемещаются из одной системы в другую, попутно происходит их обогащение или, наоборот, фильтрация.

Существует два вида обработки больших объемов данных: batch и streaming.

Batch подразумевает обработку больших массивов данных, предварительно сохраненных в некотором хранилище, то есть все данные готовы к процессингу перед запуском pipeline-а. В таком типе обработки важна пропускная способность (throughput), инфраструктура должна справляться с тяжелыми вычислениями. Примером такого процессинга может быть построение выжимки из поисковый сессий пользователя за день.

Streaming предполагает, что мы обрабатываем данные, которые поступают к нам некоторыми порциями (micro-batch), причем они не известны в полном объеме на момент запуска pipeline-а. В таком роде pipeline-ов часто оказывается важно максимальное время, затраченное на обработку порции данных. Примером такого процессинга может являться некоторая агрегация самой актуальной информации о поисковых запросах человека.

Помимо прочего два рассмотренных подхода объединяет то, что логика процессинга данных не зависит от способа обработки. Действительно, в случае batch процессингов данные могут быть обогащены более сложно вычислимыми статистиками. Однако с точки зрения, например, библиотеки для выражения логики на некотором языке программирования логика может быть представлена композицией функций многих входов и выходов. Более точная классификация такова: произвольные функции, модифицирующие, фильтрующие или мультиплицирующие данные (Map) и функции группировки (Reduce) по выделенному из данных ключу [ссылка].

Благодаря тому, что логика выражается с помощью набора примитивов, есть возможность написать библиотеку, которая предоставит общее API для написания различных процессингов. Такая библиотека под названием “Roren” была написана в Яндексе [ссылка].

1.1. Постановка задачи

Пользователь библиотеки Roren задает в коде некоторый граф из сущностей API, согласно которому данные перетекают из источников в выходные таблицы, попутно преобразуясь.

Данный граф обработки можно запускать с помощью разработанного в Яндексе MapReduce фреймворка YT [ссылка]. На машинах кластера при этом запускается некоторое количество job-ов. Job - это код обработки, вычитывающий фрагмент данных. Отдельно запущенный job требует ресурсы сети, диска и оперативной памяти.

Что такое эффективное исполнение графа обработки данных? Это применение набора эвристик для трансляции API pipeline-а в минимальное количество job-ов. Например, случай запуска графа, являющегося композицией функций, в виде отдельный job на отдельную функцию является крайне неэффективным. Важно отметить, что напрямую влиять на количество job-ов - это неестественный способ оптимизации. API MapReduce кластера позволяет запускать операции, который некоторым оптимальным образом разбиваются на job-ы. Поэтому метрика для оптимизации переформулируется в минимизацию количества запущенных операций.

Наконец, сформулируем задачу - написать компонент выполняющий эффективную трансляцию сущностей библиотеки Roren в операции MapReduce кластера системы YT.

1.2. Цель работы

Цель работы состоит в реализации части библиотеки Roren, отвечающей за перевод и выполнение pipeline-ов поверх YT. Компонент должен позволять запускать произвольные графы обработки данных, выражаемые операциями Map/Reduce.

Получившаяся реализация в сравнении с текущей должна:

- Переводить pipeline-ы в графы с меньшим количеством YT операций
- Иметь расширяемый на более сложные YT операции алгоритм трансляции

1.3. План работы

Во второй главе мы рассмотрим дизайн библиотеки Roren: от API pipeline-ов до запуска YT операций.

В третьей главе рассмотрим реализацию компонента, производящего трансляцию. Мы изложим этапы перевода roren графа в граф YT операций, алгоритм трансляции, и детально остановимся на компоненте оптимизатора.

В четвертой главе посмотрим на примеры графов и их трансляций, обсудим тестирование.

В последней главе мы поговорим о результатах работы и обсудим пути развития компонента.

2. Дизайн библиотеки Roren

Дизайн библиотеки Roren построен аналогично open-source инструменту Apache Beam [ссылка].

Apache Beam предлагает универсальный API для создания сложных pipeline-ов обработки данных, которые можно запускать на различных движках. В случае Roren движки называют executor-ами. Основные принципы этого уровня основаны на модели Beam (ранее известной как модель Dataflow) и реализованы в различной степени в каждом из executor-ов Roren.

Такой дизайн позволяет разработчикам проектировать масштабируемые приложения для обработки данных, которые могут адаптироваться к разным технологическим платформам и инфраструктурам без необходимости изменения кода. Использование Roren упрощает интеграцию с различными источниками данных и позволяет эффективно управлять потоками данных и их обработкой в реальном времени.

2.1. Interface graph

Библиотека Roren включает в себя несколько абстракций, которые упрощают процесс обработки данных в условиях больших распределенных систем. Эти абстракции применимы как к batch, так и к streaming источникам данных. При создании pipeline-а есть возможность структурировать задачи обработки данных, используя абстракции преобразований данных и результатов их применений - transform-ов и collection-ов соответственно.

Pipeline описывает всю задачу обработки данных от начала до конца. Он включает в себя чтение входных данных, их преобразование и запись выходных таблиц. При его создании есть возможность указать параметры выполнения, специфичные для конкретного паттерна выполнения: настройки streaming или MapReduce инфраструктуры.

Collection представляет собой распределенный набор данных, с которым работает pipeline. Набор данных может быть ограниченным, то есть происходить из фиксированного источника, как файл, или неограниченным, то есть поступать из постоянно обновляющегося streaming источника. Типичный pipeline создает начальный collection, читая данные из внешнего источника данных. Далее, коллекции служат входными и выходными данными для каждого преобразования в pipeline.

Transform представляет операцию обработки данных. Каждый transform принимает один или несколько объектов collection в качестве входных данных, выполняет функцию обработки, которую пользователь предоставляет, над элементами этой коллекции данных, а затем производит ноль (в случае стока графа - записи во внешнее хранилище) или более выходных объектов collection.

Roren включает множество I/O преобразований - библиотечных transforms, которые читают или записывают данные в различные внешние системы хранения данных. Чтения и записи таблиц в случае MapReduce или топиков

брокеров сообщений осуществляются через общие интерфейсы IRawRead и IRawWrite.

Трансформации могут изменять, фильтровать, группировать, анализировать или иным образом обрабатывать элементы collection-а. Преобразование создает новую выходную коллекцию данных, не модифицируя входную коллекцию. Типичный pipeline применяет последующие трансформации к каждой новой выходной collection по очереди, пока обработка не будет завершена. Однако стоит отметить, что пайплайн не обязательно должен быть одной прямой линией трансформаций, применяемых одна за другой. Можно рассматривать коллекции как переменные, а преобразования как функции, применяемые к этим переменным, что позволяет создать сложный граф обработки.

После того, как в коде описаны все transform-ы, pipeline запускается с использованием назначенного executor-а [картинка].

2.2. Понятие executor-а

Executor - это движок, отвечающий за то, каким именно образом данные поступают и обрабатываются графом transform-ов.

В простейшем случае streaming процессинга пользователь указывает входные топики брокера логов или очереди сообщений, ожидая, что данные относительно небольшими порциями будут вычитаны из источников, к ним применяются transform-ы и они будут записаны в выходные топики, очереди или таблицы.

В случае batch процессинга после указания входных и выходных таблиц с помощью примитивов API pipeline-а ожидается запуск некоторого количества операций, а как следствие Map/Reduce job-ов на кластере. Вместе эти операции будут эквивалентны применению к входным данным преобразований, указанных пользователем.

На этом этапе можно задуматься, что ParDo соответствует Map операции запущенной на кластере, а GroupByKey - Reduce.

2.3. YT executor

Далее мы рассмотрим специфику запуска именно на YT Map/Reduce кластере.

Api YT имеет возможность создания операций с многими входами, многими выходами. Причем зачастую, есть возможность писать выходные таблицы с промежуточных стадий составных операций.

Если рассмотреть некоторый подграф, состоящий целиком из ParDo, то теоретически ничто не мешает запустить его как одну Map операцию, которая внутри себя сохраняет структуру вызова пользовательских функций из ParDo. Наличие возможности указать многие входы и выходы операции позволяет запускать подграфы с несколькими истоками и стоками.

В свою очередь, GroupByKey в общем случае может быть представлен в виде MapReduce операции. Это объясняется тем, что перед функцией группировки будут вызвано какое-то количество ParDo преобразующих входные данные из YSON [ссылка] формата во внутреннее представление пары ключ-значение.

Существует важное замечание, что MapReduce - это операция Map, решардирование с помощью хэширования и запуск Reduce. Такого рода операция эффективнее Map, сортировки данных и Reduce.

I/O общение с таблицами кластера осуществляется через реализацию RawRead и RawWrite - YtRead и YtWrite. С помощью YtRead можно чтения таблиц в YSON или Protobuf [ссылка] форматах. YtWrite имеет возможность указания схемы, в том числе сортированной, для записи выходных значений. Так как рассматриваемая в данной работе реализация является proof-of-concept, мы опускаем реализации I/O в формате Protobuf и сортированных данных.

3. Реализация executor-а

Был реализован отдельный drop-in executor с несколькими этапами трансляции.

На первом этапе из pipeline и сущностей API собирается граф, который содержит похожие на roren transform-ы вершины. Разница здесь в том, что это отдельные примитивы, которые легко сливаются. Так же в графе существуют collection-ы, напрямую взятые из графа API.

Далее следует стадии оптимизации, которая может быть тривиальной. Ниже по тексту рассмотрим именно такой случай, а позже поставим оптимизатор на свой этап.

Завершающим этапом является перевод в граф YT операций.

3.1. Построение Roren graph-a

Отдельная вершина, в которую происходит трансляция roren transform-ов - это wrapper или обертка. Рассматриваемая обертка должна хранить в себе информацию о положении в исходном графе, тип transform-а, входные и выходные collection-ы.

Из wrapper-ов построена иерархия. Существует единый интерфейс обертки, который частично реализуют гранулярный wrapper и графовый wrapper [картинка]. Логически гранулярный wrapper - это набор некоторых Part-ов, которые в себе сохраняют информацию о преобразовании, его входных и выходных коллекциях. В свою очередь, графовый wrapper хранит уникальный идентификатор вершины и глубину слоя в графе [картинка].

Далее по иерархии в соответствие с transform-ами из API Roren сопоставлены wrapper: ReadWrapper, WriteWrapper, ParDoWrapper, GroupByKeyWrapper и FlattenWrapper.

У ParDoWrapper-а есть возможность сколлапсировать с другим ParDoWrapper-ом. Это естественная возможность, т.к. логически ParDo является фукнцией над некоторыми входными данными, а коллапсирование - способ выразить композицию функций. Однако помимо объединения в композицию функций из двух функций с разными входами можно сделать одну с многими входными таблицами.

Здесь важно заметить, что практически каждый wrapper может иметь входные или выходные таблицы, кроме GroupByKeyWrapper и FlattenWrapper. Очевидно, что ReadWrapper и WriteWrapper имеют какие-то входные и выходные таблицы соответственно. Ради упрощения конденсации графа, ParDoWrapper в отличие от ParDo может иметь входные и выходные таблицы, как результат слияния с ReadWrapper или WriteWrapper.

3.2. Трансляция в граф YT операций

Отдельный компонент отвечает за перевод RorenGraph-а в граф YT операций. Это включает в себя преобразование узлов, отвечающих за фильтрацию, агрегацию и трансформацию данных, в задачи Map и Reduce в YT. Основной вызов здесь — обеспечить, чтобы все зависимости и последовательности операций были правильно интерпретированы и перенесены, сохраняя при этом оптимальную производительность и масштабируемость в новой среде.

Граф из полученных операций может быть запущенной под общей транзакцией. Операции будут запущены с учетом причинности, конкретнее в топологическом порядке по слоям.

На этом этапе сложность составляет сопоставление входных и выходных таблиц файловым дескрипторам, через которые происходит запись в YT job. Помимо этого, информация о конденсированных подграфах должна быть сериализуема для запуска в виде job-а на Map/Reduce кластере.

3.3. Алгоритм оптимизации

3.3.1. Подходы к оптимизации

Стандартным подходом в оптимизации деревьев является pattern-matching [ссылка]. В простейшем случае это означает поддержку стека вершин. В ходе оптимизации вершины извлекаются из стека, локальная конфигурация вокруг вершины пробегает варианты оптимизаций, в случае успеха некоторые вершины сливаются и новые помещаются в стек.

Проблема такого подхода в графах обработки данных состоит в следующем. Если извлекать вершины из стека в произвольном порядке, можно получить ситуацию, при которой после оптимизаций в графе возникает петля [картинка]. Это нарушает его корректность, данные для операции не готовы, т.к. чтобы их приготовить нужно запустить эту же операцию.

Предлагается обходить граф, сохраняя зависимости по данным при обходе и в процессе оптимизаций. Это можно сделать с помощью обхода в топологическом порядке. Для предложенного в данной работе алгоритма требуется более сильное упорядочивание вершин.

3.3.2. Обход и правила

На входе имеем $roren$ граф G. Обозначения:

PD - ParDo, F - Flatten, Gbk - GroupByKey

$\$Transform^{\pm}$ - Transform с сохранением сортировки или без

$\pm$ - вход, сортированный или нет

$\$Transform^{*}$ - оригиальный Transform из $roren$, имеющий один вход

Алгоритм будет опираться на правила (рис. 1), которые позволяют объединять Transform-ы в одну операцию для дальнешего запуска YT операций.

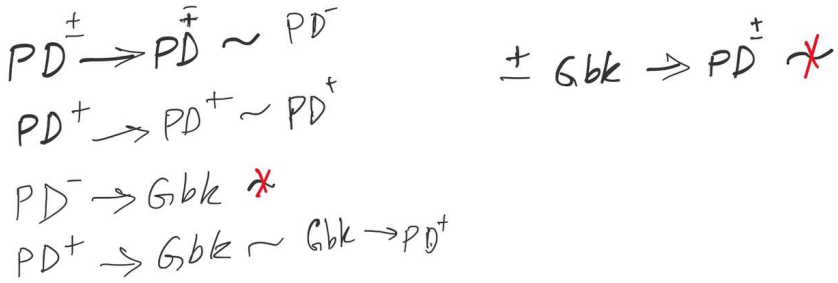

Рис. 1: Правила roren → roren

Алгоритм 1. Сначала перестроим граф G по правилу $F \to \$Transform* \sim \$Transform$, где $Transform$ - это некоторое преобразование, которое после перестройки может иметь несколько входов (рис. 2).

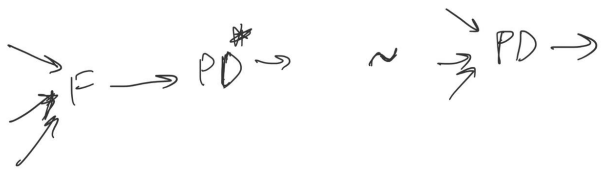

Рис. 2: Преобразование с Flatten

Построим слоистую сеть R по G (рис. 3).

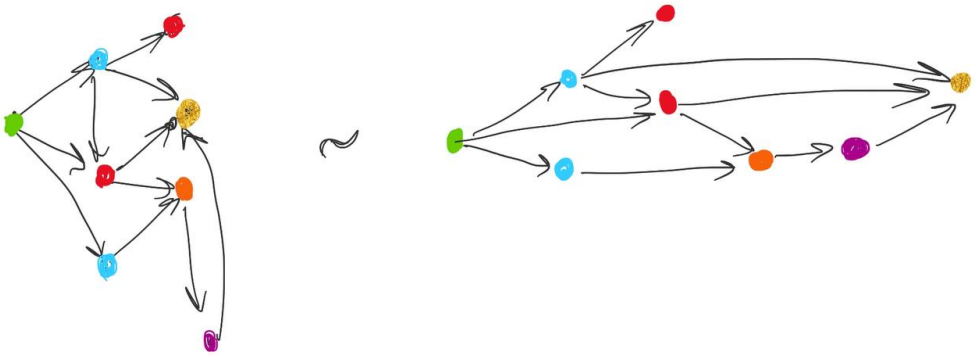

Рис. 3: Построение слоистой сети

Теперь будем проходить по R. Шаг обхода будет состоять из двух стадий: collapse и expand. На стадии collapse некоторые вершины сливаются, на стадии expand новые вершины добавляются в рассмотрение. Обход будем делать по слоям, рассматривая текущий и следующий, по ходу присваивая вершинам цвета:

- черные - ещё не посещенные вершины, на слоях отличных от текущего и следующего,
- оранжевые - посещенные вершины, на текущем слое или на предыдущих слоях, если есть ребро в черную вершину,
- красные - не посещенные вершины, находящиеся на следующем слое,
- зеленые - посещенные вершины, которые больше не будут участовать в слиянии.

При старте обхода помечаем первый слой оранжевым цветом, второй слой красным.

На произвольном шаге алгоритма рассматриваем каждую из оранжевых вершин отдельно. Сначала идет стадия collapse. На данном этапе может возникнуть 2 типа коллизий (рис. 4), которые можно пытаться разрешать, но мы не будем...

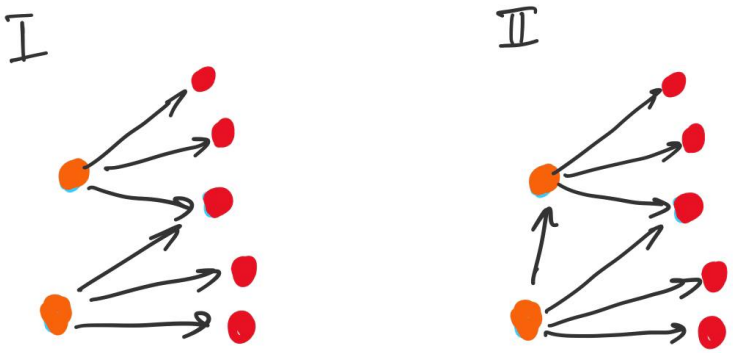

Рис. 4: Типы коллизий

Для вершин, в которых не возникает коллизий, применяя правила roren $\rightarrow$ roren, можно влить красную вершину в оранжевую.

После начинается стадия expand, оставшиеся красные вершины окрашиваются в оранжевый, текущий слой продвигается вперед. Сейчас оранжевые вершины с прошлого слоя могут стать зелеными, в случае если все ребра входят в оранжевые вершины.

Пример обхода приведен на рис. 5.

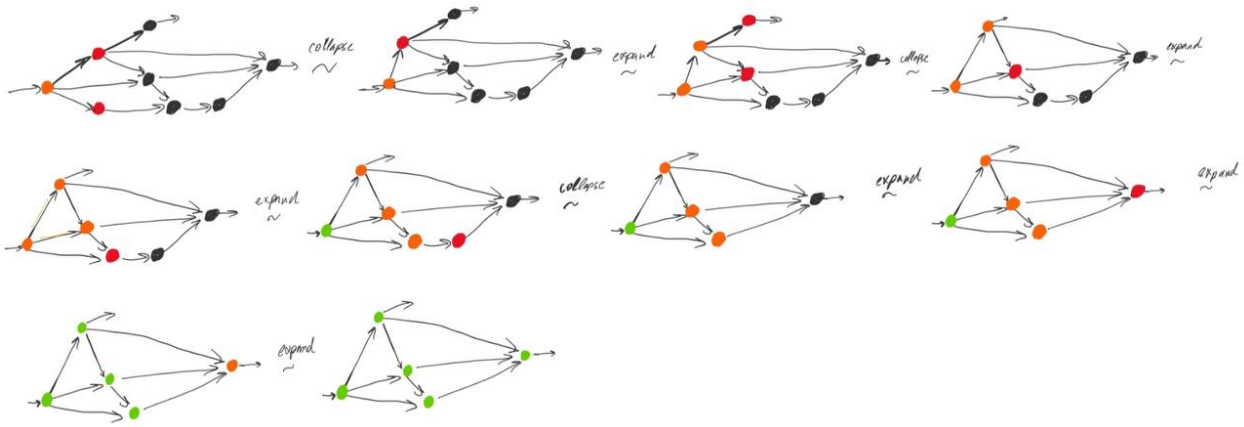

Рис. 5: Пример обхода

После того, как завершился обход по сети R. Можно перевести граф в операции Map, Reduce и MapReduce по правилам roren $\rightarrow$ YT (рис. 6).

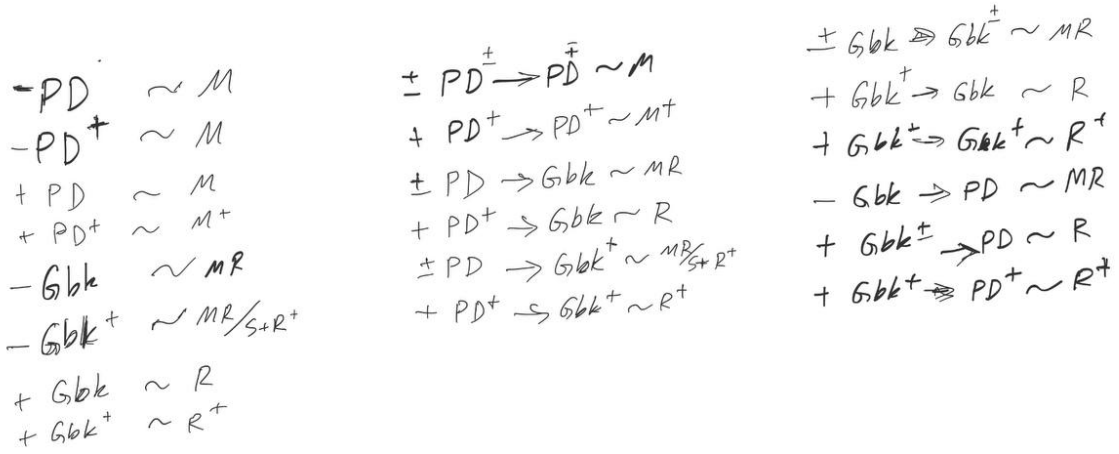

Рис. 6: Правила roren → YT

Комментарий. На самом деле, в алгоритме есть несколько точек кастомизации поведения.

- Мы можем пытаться разрешать коллизии (рис. 4): для первого типа сливать две оранжевые вершины, для второго придумать правила для ”треугольников”.
- После преобразований размер данных по ключу может сильно меняться: уменьшаться, если трансформ фильтрует данные, увеличиваться, если дробит.
- Можно сливать независимые ветки графа, но стоит думать об отсутствии параллельности исполнения в данном случае.
- У операций в качестве атрибутов могут быть выставлены лимиты на ресурсы, которые есть возможность учитывать при слияниях.
- Прежде чем применять во всех местах правила слияния с учетом сортированности, можно сделать предпроход и понять до каких входом прокидывается сортированность.
- Кажется, что coGbk пока нельзя средствами YT выразить; Combine распадается в два Reduce; Flatten, после которого не шло никаких Transform,

станет YT Merge-ом.

- Если у Transform-а есть side выход и ребро в другой Transform, то можно не делать временную таблицу.

3.4. Оптимизатор

Стадия оптимизации подразумевает несколько обходов по заранее построенному RorenGraph-у.

3.4.1. Стратегии оптимизации

В оптимизаторе отдельно выделен интерфейс IStrategy [картинка], позволяющий коллапсировать оранжевую и красную вершину связанные ребром. Важно, что этот случай подразумевает, что нет другой оранжевой вершины, ребра которой ведут в рассматриваемую красную [картинка].

Помимо этого IStrategy позволяет разрешить конфликтную ситуацию, когда есть множество оранжевых вершин, ребра которых ведут в общую красную вершину.

Реализацией данной стратегии может быть стратегия, которая коллапсирует вершины и разрешает конфликты.

Однако это далеко не единственный вариант. Пользователям может понадобится стратегия, которая не сливает параллельные ветви, т.к. это увеличивает latency готовности конкретных таблиц [картинка].

Также пользователи могут иметь вычислительно тяжелые операции, требующие большого количества RAM. Так, в случае отдельного запуска операций проблем не будет, но при их слиянии памяти, отведенной на один job, может не хватить [картинка].

3.4.2. Предобход для Flatten

В дизайн Roren заложена концепция, которая позволяет легко описывать в коде графы. Она состоит в том, что у transform-а может быть только один вход. В случае графов с Flatten такой дизайн приведет к тому, что требуется рассматривать несколько слоев transform-ов [картинка]. Также для упрощения алгоритма мы рассматриваем граф, состоящий из transform-ов, не обращая внимания на коллекции, но сохраняя их.

Для того, чтобы перейти к упрощенному графу требуется сделать предобход, который удалить из графа Flatten, соединяя предыдущие и последующие transform-ы напрямую.

3.5. Дополнения алгоритма

3.5.1. Сохранение сортированности

3.5.2. Когда не стоит оптимизировать

4. Запуск pipeline-ов

4.1. Уровни оптимизации

Для проверки корректности каждого из этапов оптимизации удобно не писать дополнительные тесты, а отключать некоторые из этапов, проверяя, что корректность итогового графа не ломается. Например, можно не делать предобход Flatten, если представить Flatten - как тривиальный ParDo. В свою очередь как YT операция Flatten представляется в виде тривиального Map-а.

Есть возможность отключать полностью любые оптимизации, проверяя, находится ли ошибка в коде построение RorenGraph-а или вне него.

4.2. Анализ исполнений

На картинках приведены примеры исходных API графов, полученных из них графов операций YT [картинка].

5. Заключение

5.1. Результаты

Реализован компонент для библиотеки Roren - drop-in замена текущему executor-у [ссылка].

Алгоритм, реализованный внутри executor-а, на нетривиальных графах с многими входами и выходами способен объединять ParDo в одну сложную Map операцию. Так же работает перевод с MapReduce операциями.

Полученный алгоритм легко расширяется на произвольные новые Transform-ы, в том числе на новые классы преобразований, например, преобразования сохраняющие сортированность входных данных.

5.2. Планы на будущее

Задача написания подобного рода инструмента не решается идеально. Существует несколько направлений, в которые можно расширять возможности executor-а:

- Перенос алгоритма оптимизации и логики с Flatten в production версию YT executor
- Добавление большего количества transform-ов в текущую версию executor-а
- Добавление в алгоритм техники pattern matching-а