

### **Аннотация**

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

# Содержание

<b>1</b>	<b>Введение</b>	<b>5</b>
1.1	Постановка задачи . . . . .	8
1.2	Цель работы . . . . .	9
1.3	План работы . . . . .	10
<b>2</b>	<b>Дизайн библиотеки Roren</b>	<b>11</b>
2.1	Interface graph . . . . .	12
2.2	Понятие executor-a . . . . .	14
2.3	YT executor . . . . .	15
<b>3</b>	<b>Реализация executor-a</b>	<b>17</b>
3.1	Построение Roren graph-a . . . . .	18
3.2	Трансляция в граф YT операций . . . . .	20
3.3	Алгоритм оптимизации . . . . .	21
3.3.1	Подходы к оптимизации . . . . .	21
3.3.2	Обход и правила . . . . .	22
3.4	Оптимизатор . . . . .	27
3.4.1	Стратегии оптимизации . . . . .	27
3.4.2	Предобход для Flatten . . . . .	28
3.5	Дополнения алгоритма . . . . .	29
3.5.1	Сохранение сортированности . . . . .	29
3.5.2	Когда не стоит оптимизировать . . . . .	29
<b>4</b>	<b>Запуск pipeline-ов</b>	<b>31</b>
4.1	Уровни оптимизации . . . . .	31
4.2	Анализ исполнений . . . . .	32
4.3	Логирование . . . . .	34
<b>5</b>	<b>Заключение</b>	<b>44</b>

5.1	Результаты . . . . .	44
5.2	Планы на будущее . . . . .	45

# 1. Введение

Распределенные системы, основанные на принципах параллельной обработки и разделения задач, значительно повышают эффективность и масштабируемость при работе с большими объемами данных. Одним из ведущих подходов в архитектуре таких систем является технология MapReduce [1], первоначально разработанная Google для индексации веб-страниц и обработки больших массивов информации, распределенных по множеству серверов.

Технология MapReduce организована вокруг двух основных функций: Map и Reduce. Функция Map принимает входные данные и преобразует их в промежуточные пары ключ-значение, которые затем агрегируются по ключам и передаются функции Reduce. Задача функции Reduce заключается в суммировании или иной форме обработки всех промежуточных значений, связанных с каждым ключом, для генерации конечного результата. Этот подход позволяет распределять обработку данных между множеством узлов, тем самым резко увеличивая производительность и обеспечивая горизонтальное масштабирование.

С другой стороны, существует задача описания бизнес-логики для обработки данных. Для решения подобных задач часто применяется методика разработки специализированных процессов - pipeline-ов. В рамках этой методики используется код, который определяет граф обработки данных. Эта структура определяет путь, по которому данные перетекают из одной системы в другую. В процессе такого перемещения данные могут подвергаться различным изменениям: обогащение путем интеграции дополнительной информации, или фильтрации, когда из потока удаляются нерелевантные или избыточные данные. Эти процессы помогают оптимизировать качество и структуру данных, делая их более подходящими для анализа и отчетности.

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

Batch-обработка подразумевает работу с большими объемами данных,

которые были предварительно сохранены в некотором хранилище. Это означает, что все данные уже подготовлены и доступны для обработки до начала выполнения pipeline-a. При таком подходе основной акцент делается на пропускную способность системы, так как инфраструктура должна быть рассчитана на выполнение сложных и трудоемких вычислений. Примером batch-обработки может служить анализ и построение выжимки из поисковых сессий пользователя за определенный день.

Важно отметить, что в процессе batch-обработки особое внимание уделяется не только скорости обработки больших данных, но и их последующей оптимизации для хранения и анализа. Например, после обработки данных о поисковых сессиях могут быть выделены ключевые метрики, которые затем используются для улучшения пользовательского опыта или оптимизации поисковых алгоритмов. Таким образом, качество обработки данных напрямую влияет на эффективность последующих бизнес-процессов.

Streaming-обработка данных означает, что данные поступают в систему порциями, известными как micro-batch, при этом полный объем данных неизвестен на момент запуска pipeline-a. В таких системах особенно важно уделять внимание времени, необходимому для обработки каждой такой порции. Это требует от инфраструктуры возможности быстро реагировать на потоковые данные и эффективно их обрабатывать, чтобы минимизировать задержки и обеспечить актуальность обрабатываемой информации. Примером использования streaming-обработки может служить агрегация данных о последних поисковых запросах пользователя, что позволяет предоставлять релевантные и своевременные результаты.

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 [1].

Благодаря использованию набора простых и понятных примитивов, появляется возможность разработки универсальной библиотеки, которая может предложить общий API для создания разнообразных процессов обработки данных. В результате такого подхода в компании Яндекс была разработана и реализована библиотека под названием Roren [2].

## 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 [3]. На машинах кластера при этом запускается некоторое количество job-ов. Job [4] - 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 YТ операций.

В третьей главе рассмотрим реализацию компонента, производящего трансляцию. Мы изложим этапы перевода roren графа в граф YТ 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 [5].

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 [6]) 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.

## 2.1. Interface graph

Библиотека Rogen 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 таблица, 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.

Rogen 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 (рис 1).

## 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.

Аpi 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 [7] формата во внутреннее представление пары ключ-значение.

Существует важное замечание, что MapReduce - это операция Map, рещардирование 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 [8] форматах. YtWrite имеет возможность указания схемы, в том числе сортированной, для записи выходных значений [9]. Так как рассматриваемая в данной работе реализация является proof-of-concept, мы опускаем реализации I/O в формате Protobuf и сортированных данных.

```

32     auto pipeline = MakeYtPipeline("freud", "///tmp");
33
34     pipeline
35         | YtRead<TNode>("///home/ermolovd/yt-tutorial/staff_unsorted")
36         | ParDo([] (const TNode& node) -> TUserLoginInfo {
37             TUserLoginInfo info;
38             info.Name = node["name"].AsString();
39             info.Login = node["login"].AsString();
40             return info;
41         })
42         | ParDo([] (const TUserLoginInfo& info) {
43             TUserEmailInfo result;
44             result.Name = info.Name;
45             result.Email = info.Login + "@yandex-team.ru";
46             return result;
47         })
48         | ParDo([] (const TUserEmailInfo& info) {
49             TNode result;
50             result["name"] = info.Name;
51             result["email"] = info.Email;
52             return result;
53         })
54         | YtWrite(
55             "///tmp/ermolovd-tt",
56             TTableSchema()
57                 .AddColumn("name", NTi::String())
58                 .AddColumn("email", NTi::String())
59         );
60
61     pipeline.Run();

```

Рис. 1: Типичный pipeline с YT executor-ом

### 3. Реализация executor-a

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-a, входные и выходные collection-ы.

Из wrapper-ов построена иерархия. Существует единый интерфейс обертки, который частично реализуют гранулярный wrapper и графовый wrapper (рис 2). 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.

```
92  class ITransformWrapper
93      : public virtual TThrRefBase
94  {
95  public:
96      virtual ~ITransformWrapper() = default;
97
98      virtual void LinkTable(TTableId tableId, bool isSource) = 0;
99
100     virtual std::vector<TPartPtr>& GetParts() = 0;
101     virtual const std::vector<TPartPtr>& GetParts() const = 0;
102
103     virtual size_t GetPartCount() const = 0;
104
105     virtual TWrapperId GetId() const = 0;
106
107     virtual TLevel& AccessLevel() = 0;
108     virtual TLevel GetLevel() const = 0;
109 };
```

Рис. 2: Интерфейс обертки для конденсации

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 функцией 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, ParDo Wrapper 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 RogenGraph-a 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-a на Map/Reduce кластере.

### 3.3. Алгоритм оптимизации

#### 3.3.1. Подходы к оптимизации

Стандартным подходом в оптимизации деревьев является pattern-matching [10]. 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 (рис 3). Это нарушает его корректность, данные для операции не готовы, т.к. чтобы их приготовить нужно запустить эту же операцию.

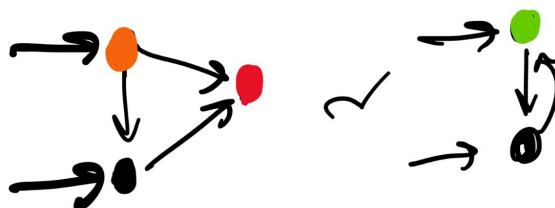


Рис. 3: Случай появления петли в графе

Предлагается обходить граф, сохраняя зависимости по данным при обходе и в процессе оптимизаций. Это можно сделать с помощью обхода в топологическом порядке [11]. 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

$\$Transform^*$  - оригинальный Transform из *roren*, имеющий один вход

Алгоритм будет опираться на правила (рис. 4), которые позволяют объединять Transform-ы в одну операцию для дальнейшего запуска YT операций.

$$\begin{aligned}
 PD^{\pm} &\rightarrow PD^{\pm} \sim PD^{\pm} \\
 PD^{+} &\rightarrow PD^{+} \sim PD^{+} \\
 PD^{-} &\rightarrow Gbk \quad * \\
 PD^{+} &\rightarrow Gbk \sim Gbk \rightarrow PD^{+}
 \end{aligned}
 \qquad
 \begin{aligned}
 \pm Gbk &\rightarrow PD^{\pm} \quad *
 \end{aligned}$$

Рис. 4: Правила  $roren \rightarrow roren$

**Алгоритм 1.** Сначала перестроим граф  $G$  по правилу  $F \rightarrow \$Transform^* \sim \$Transform$ , где  $Transform$  - это некоторое преобразование, которое после перестройки может иметь несколько входов (рис. 5).

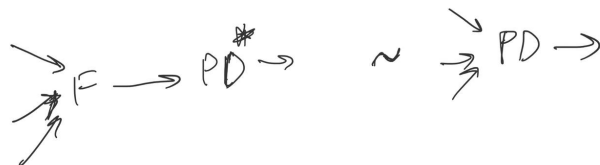


Рис. 5: Преобразование с Flatten

Построим слоистую сеть  $R$  по  $G$  (рис. 6).

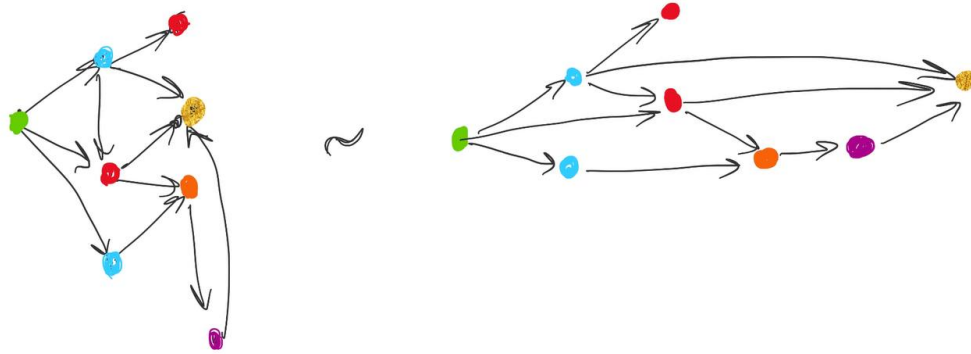


Рис. 6: Построение слоистой сети

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 участвовать 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. 7), которые можно пытаться разрешать через pattern-matching или оставлять как есть

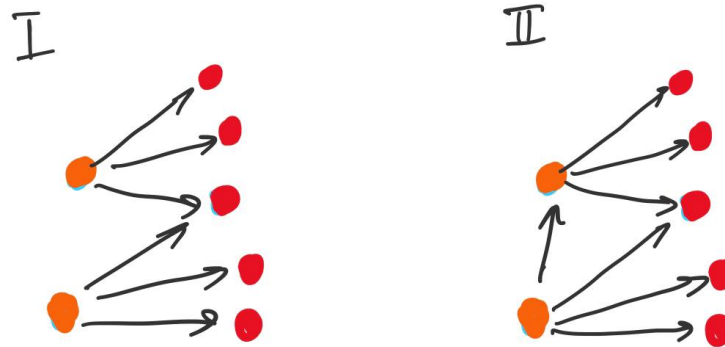


Рис. 7: Типы коллизий

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

После начинается стадия  $\text{expand}$ , 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. 8.

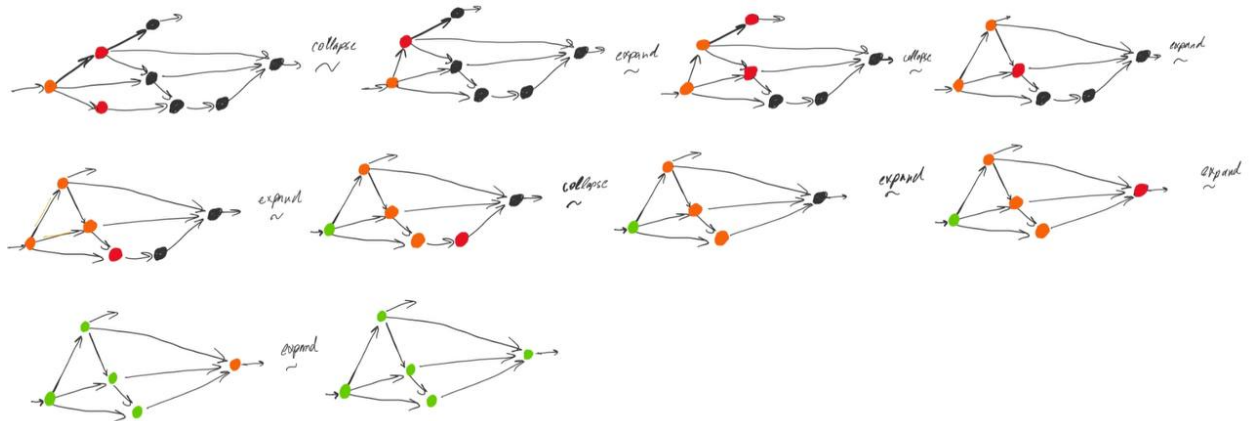


Рис. 8: Пример обхода

После того, как завершился обход по сети  $R$ . Можно перевести граф в операции  $\text{Map}$ ,  $\text{Reduce}$  и  $\text{MapReduce}$  по правилам  $\text{roren} \rightarrow \text{YT}$  (рис. 9).

На самом деле, в алгоритме есть несколько точек кастомизации поведе-

$$\begin{array}{lll}
- PD \sim M & \pm PD^+ \rightarrow PD^+ \sim M & \pm Gbk \Rightarrow Gbk^+ \sim MR \\
- PD^+ \sim M & + PD^+ \rightarrow PD^+ \sim M^+ & + Gbk^+ \rightarrow Gbk \sim R \\
+ PD \sim M & \pm PD \rightarrow Gbk \sim MR & + Gbk^+ \rightarrow Gbk^+ \sim R^+ \\
+ PD^+ \sim M^+ & + PD^+ \rightarrow Gbk \sim R & - Gbk \rightarrow PD \sim MR \\
- Gbk \sim MR & \pm PD \rightarrow Gbk^+ \sim MR/S+R^+ & + Gbk^+ \rightarrow PD \sim R \\
- Gbk^+ \sim MR/S+R^+ & + PD^+ \rightarrow Gbk^+ \sim R^+ & + Gbk^+ \Rightarrow PD^+ \sim R^+ \\
+ Gbk \sim R & & \\
+ Gbk^+ \sim R^+ & & 
\end{array}$$

Рис. 9: Правила roren  $\rightarrow$  YT

ния.

- Мы можем пытаться разрешать коллизии (рис. 7) первого типа без pattern-matching-a - сливать две оранжевые вершины
- 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
- Combine распадается в два Reduce; Flatten, после которого не шло никаких Transform, станет YT Map-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 (рис 10), позволяющий коллапсировать оранжевую и красную вершину связанные ребром. Важно, что этот случай подразумевает, что нет другой оранжевой вершины, ребра которой ведут в рассматриваемую красную.

```
1  class IStrategy
2  {
3  public:
4      virtual ~IStrategy() = default;
5
6      // Optimize orange -> red
7      virtual bool TryCollapsing(TWrapperId from, TWrapperId to) = 0;
8
9      // Resolve conflict of branches
10     virtual std::vector<TWrapperId> TryResolvingConflict(TWrapperId to, std::set<TWrapperId>& prevForTo) = 0;
11 };
```

Рис. 10: Интерфейс стратегии конденсации вершин

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.

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.

### 3.4.2. Предобход для Flatten

В дизайн Rogen 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. 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 удалит из графа Flatten, соединяя предыдущие и последующие transform-ы напрямую.

## **3.5. Дополнения алгоритма**

### **3.5.1. Сохранение сортированности**

Сохранение сортированности может быть легко встроено в текущий алгоритм. Правила, по которым будет происходить оптимизация помечены знаком ”+” (рис. 4)

### **3.5.2. Когда не стоит оптимизировать**

В процессе преобразования данных размер данных, ассоциированный с каждым ключом, может значительно измениться. Если трансформация включает фильтрацию данных, то объем данных по ключу уменьшается. Напротив, если трансформация предполагает декомпозицию или увеличение детализации данных, объем увеличивается. Поэтому от того, в каком порядке и какие именно преобразования будут объединены может зависеть количество пропускаемых через систему промежуточных данных [11].

Слияние независимых веток графа данных может быть эффективным решением для упрощения структуры, однако такой подход имеет свои ограничения. Важно учитывать, что при таком слиянии может пропасть параллельность исполнения операций, что может повлиять на общую производительность процессинга данных.

При проектировании операций в рамках системы возможно установление лимитов на использование ресурсов для каждой операции. Эти лимиты можно учитывать при слиянии операций, что поможет оптимизировать расход ресурсов и повысить эффективность выполнения задач.

Применение правил слияния с учетом сортированности данных может значительно повысить эффективность обработки. Однако, перед тем как использовать эти правила повсеместно, рекомендуется выполнить предварительный обход графа и определить, до какого уровня сортированность данных сохраняется на разных этапах обработки.

В случае, если в графе встречаются преобразования отличные от покрытых Wrapper-ами, их можно не оптимизировать в ходе обхода, сохраняя корректность.

## 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 построения RorenGraph-а или вне него.

## 4.2. Анализ исполнений

На картинке приведен пример исходного Rogen графа (рис. 11) и полученный из него граф операций YT (рис. 12).

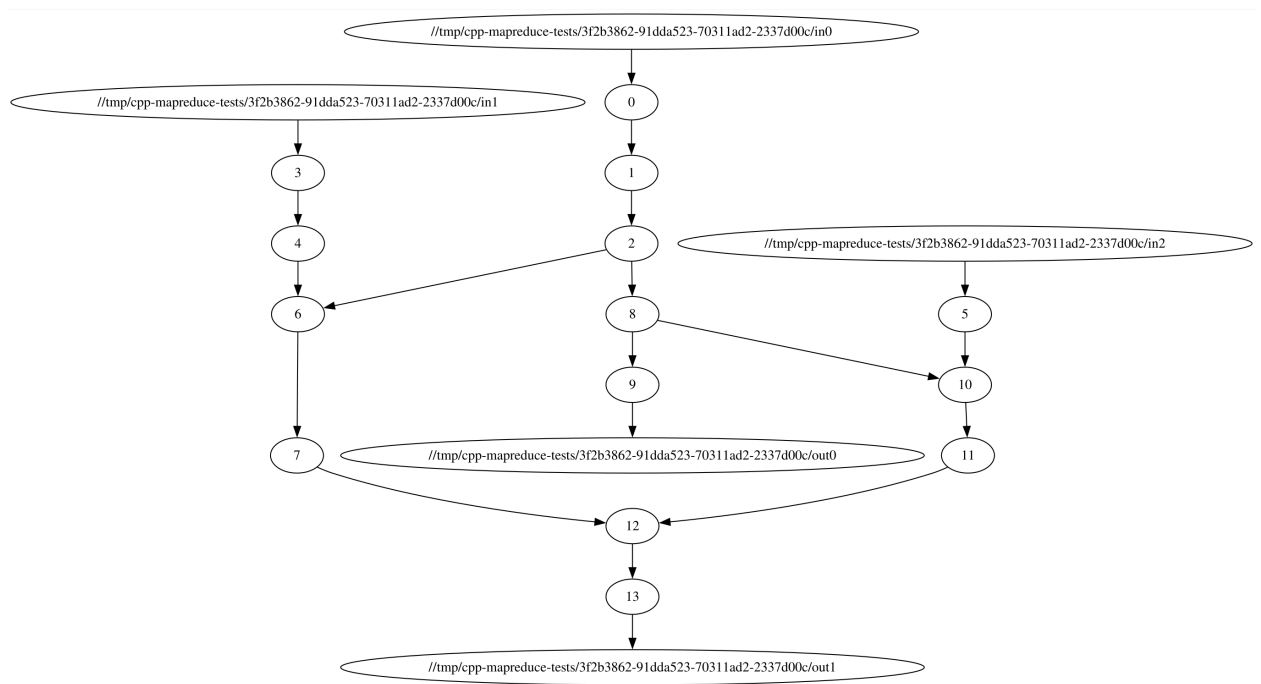


Рис. 11: Неоптимизированный rogen граф из ParDo

Также есть возможность изучить Rogen графы полученные на промежуточных этапах: после удаления Flatten (рис. 13), после оптимизации (рис. 14) и подграф сконденсированный в единственную вершину (рис. 15)

Также приведем пример для голен графа (рис. 16), который результирует в единственную операцию MapReduce (рис. 17).

Ниже прикреплены скриншоты соответствующие Map (рис. 18) и MapReduce (рис. 19) job-ам из двух предыдущих примеров.



## 4.3. Логирование

Есть два независимых способа логирования: логирование Roren графа оберток и логирование YT графа операций.

Класс `TDumpYtGraphToDOTVisitor` в коде демонстрирует реализацию паттерна `visitor` в контексте визуализации структуры данных графа операций `TYtGraphV3` в формате DOT. Формат DOT как раз используется для визуализации графов.

Этот паттерн позволяет организовать обход графа таким образом, чтобы можно было применять специфические операции к каждому элементу графа без изменения их классов. Основной метод, который инициирует процесс визуализации. Он вызывает методы `Prologue`, `Traverse`, и `Epilogue`, которые собственно и управляют процессом генерации DOT-описания. `Prologue` и `Epilogue` служат для подготовки и завершения документа DOT, в то время как `Traverse` обеспечивает обход всех узлов графа.

В методе `Traverse` происходит обход узлов графа, где для каждой операции и связанной с ней таблицы вызываются специфичные методы `OnOperation` и `OnTableInfo`. Это демонстрирует гибкость паттерна в обработке различных типов узлов.

Методы `OnOperation` и `OnTableInfo` реализуют действия, которые нужно выполнить для каждого типа узлов в графе. `OnOperation` обрабатывает узлы, представляющие операции, генерируя строки для DOT файла, которые описывают эти операции и их связи. Аналогично `OnTableInfo` и `OnTable` обрабатывают узлы, представляющие таблицы, добавляя информацию о таблицах и их связях с операциями.

`Visitor` использует внешний объект `Stream` для формирования результатов своей работы и объект `Graph`, который содержит данные графа, над которым выполняется обработка. Это позволяет легко модифицировать логику визуализации или добавлять новые типы визуализаций, не изменяя основную

структуру класса или классы данных.

Класс `TPrinter` предназначен для визуализации структуры графа `TRorenGraph` в формате DOT, который используется для описания направленных графов. Конструктор класса инициализирует экземпляр с ссылкой на граф, а метод `DumpDOT` генерирует строковое представление графа. В этом методе используются лямбда-функции для форматирования идентификаторов таблиц и вершин, а также для обработки путей и меток. В процессе генерации DOT-описания, метод перебирает все таблицы и вершины графа, выписывая связи между ними и формируя строку, которая затем выводится в выходной поток `TStringOutput`. Такой подход позволяет динамично строить визуальное представление структуры данных, что особенно полезно для анализа сложных зависимостей и иерархий внутри графа.

Отрисовка графов производится с помощью инструмента `Graphviz` [12].

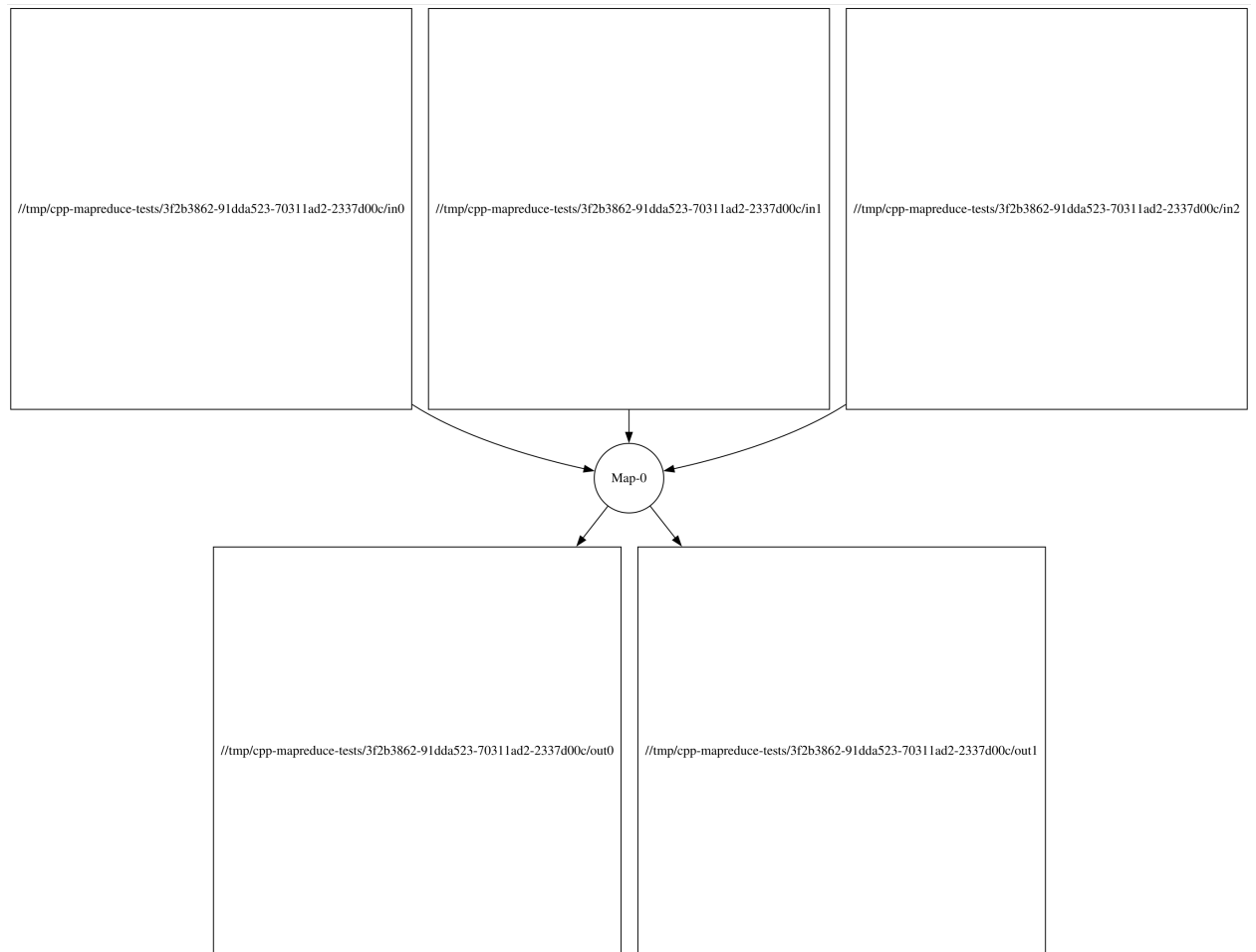


Рис. 12: Итоговый YТ граф

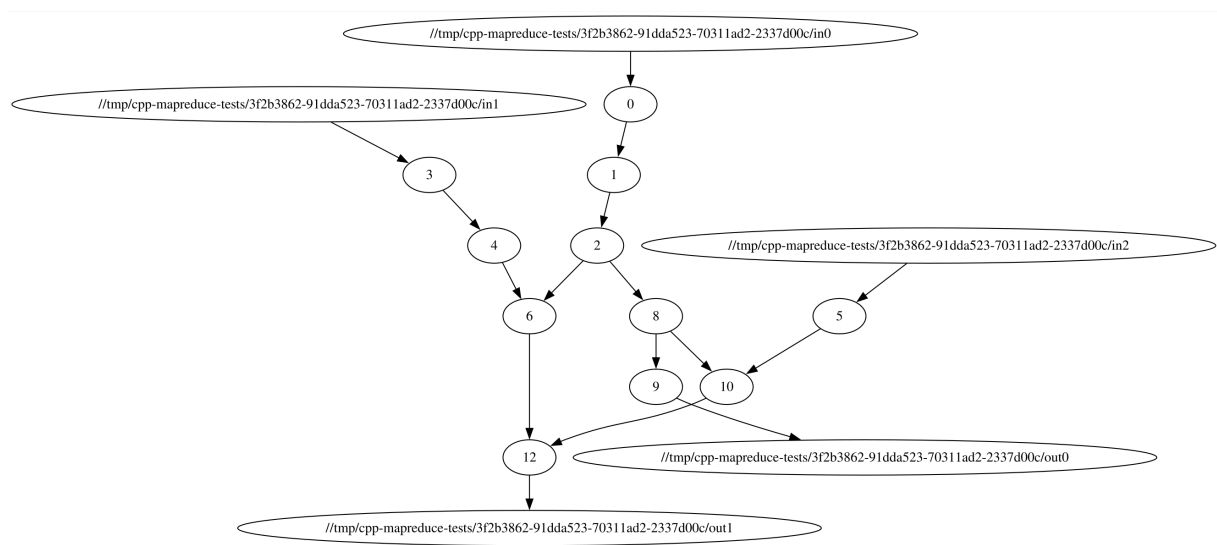


Рис. 13: Roren граф из ParDo после предобхода Flatten

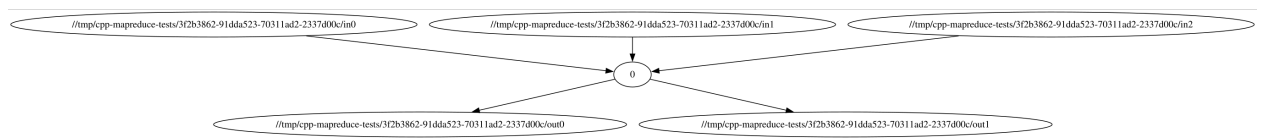


Рис. 14: Rogen граф из ParDo после основного обхода

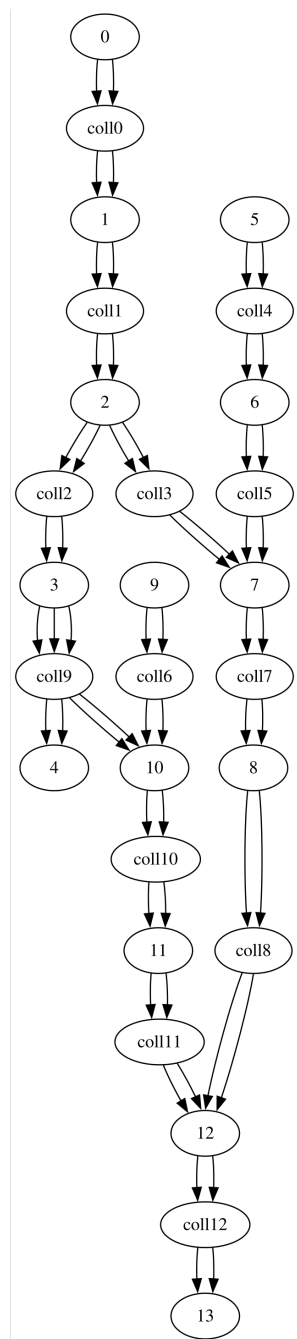


Рис. 15: Подграф полностью оптимизированного rogen графа

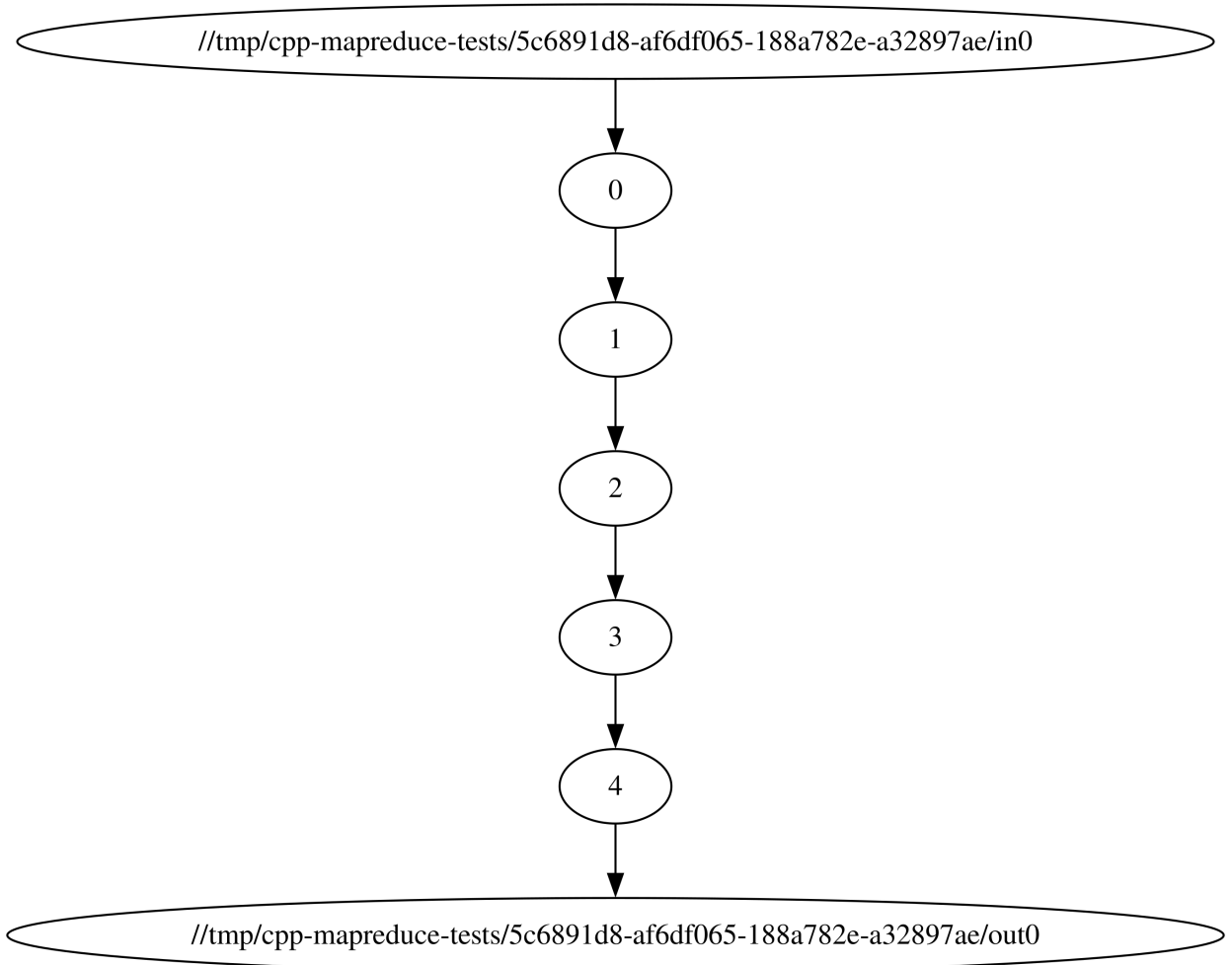


Рис. 16: Rogen граф из ParDo и GroupByKey (цифра 2 на рисунке) до оптимизаций

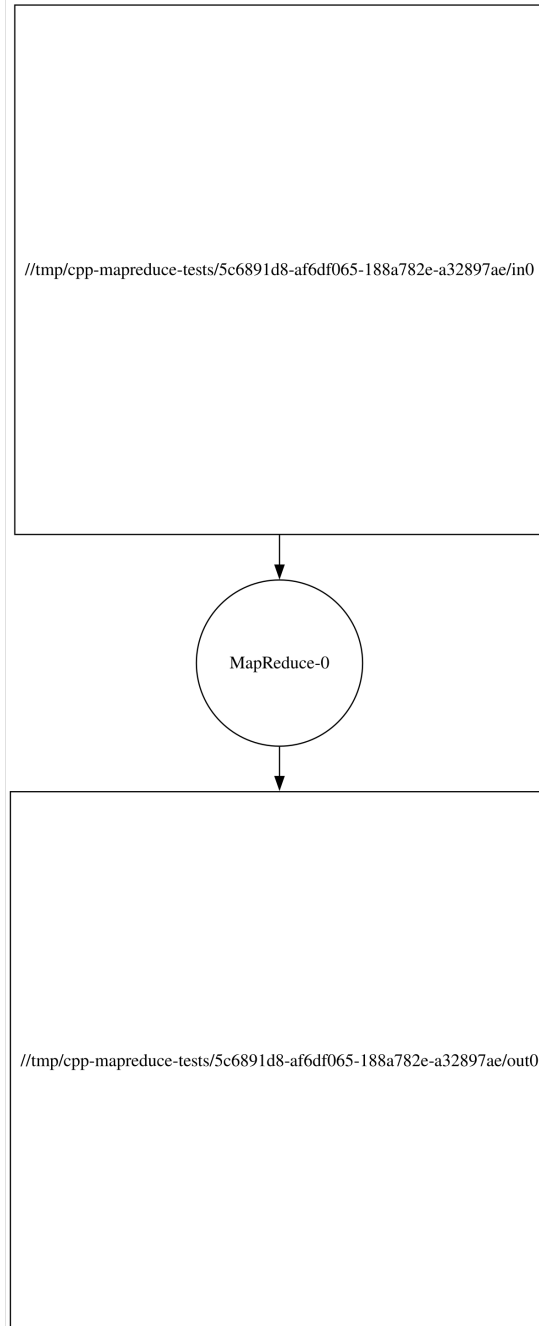








Рис. 17: Итоговый YT граф с одной MapReduce операцией



### Input 3

Name	Filters	Tags
 <a href="#">//tmp/cpp-mapreduce-tests/3f2b3862-91dda523-70311ad...</a>		
 <a href="#">//tmp/cpp-mapreduce-tests/3f2b3862-91dda523-703...</a>	-	-
 <a href="#">//tmp/cpp-mapreduce-tests/3f2b3862-91dda523-703...</a>	-	-
 <a href="#">//tmp/cpp-mapreduce-tests/3f2b3862-91dda523-703...</a>	-	-

### Output 2

Name	Live preview	Tags
 <a href="#">//tmp/cpp-mapreduce-tests/3f2b3862-91dda523-70311ad...</a>		
 <a href="#">//tmp/cpp-mapreduce-tests/3f2b3862-91dda523-703...</a>	-	-
 <a href="#">//tmp/cpp-mapreduce-tests/3f2b3862-91dda523-703...</a>	-	-

### Mapper



Class name	NRoren::NPrivate::TMultiParDoMap
Environment	YT_JOB_ARGUMENTS={"has_state"=%true,"job_name"="NRoren::NPrivate::TMultiParDoMap","output_t able_count"=2}
Files	<a href="#">jobstate</a>  <a href="#">cppbinary%</a>
Command	"YT_USE_CLIENT_PROTOBUF=0 ./cppbinary"

Рис. 18: Спецификация Map операции запущенной на кластере YT

### Input 1

Name	Filters	Tags
<a href="#">//tmp/cpp-mapreduce-tests/5c6891d8-af6df065-188a782e...</a>		
<a href="#">//tmp/cpp-mapreduce-tests/5c6891d8-af6df065-188a...</a>	-	-

### Output 1

Name	Live preview	Tags
<a href="#">//tmp/cpp-mapreduce-tests/5c6891d8-af6df065-188a782e...</a>		
<a href="#">//tmp/cpp-mapreduce-tests/5c6891d8-af6df065-188a...</a>	-	-

### Mapper

Class name	NRoren::NPrivate::TMultiParDoMap
Environment	YT_JOB_ARGUMENTS={"has_state"=%true;"job_name"="NRoren::NPrivate::TMultiParDoMap";"output_able_count"=1}
Files	<a href="#">jobstate</a> <a href="#">cppbinary</a>
Command	"YT_USE_CLIENT_PROTOBUF=0 ./cppbinary"

### Reducer

Class name	NRoren::NPrivate::TImpulseRawJob
Environment	YT_JOB_ARGUMENTS={"has_state"=%true;"job_name"="NRoren::NPrivate::TImpulseRawJob";"output_able_count"=1}
Files	<a href="#">jobstate</a> <a href="#">cppbinary</a>
Command	"YT_USE_CLIENT_PROTOBUF=0 ./cppbinary"

Рис. 19: Спецификация MapReduce операции запущенной на кластере YT

## 5. Заключение

### 5.1. Результаты

Реализован компонент для библиотеки Roren - drop-in замена текущему executor-у [13].

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 Transforms, 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-a:

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

## Список литературы

1. *Dean J., Ghemawat S.* MapReduce: Simplified Data Processing on Large Clusters. — 2004.
2. *Dmitry Ermolov.* Roren library. — URL: <https://cppconf.ru/archive/2023/talks/36a6a02a1c444021970215ebb7a5e1c6/>.
3. *Max Babenko.* YTsaurus: An open source big data platform for distributed storage and processing. — URL: <https://ytsaurus.tech/en>.
4. Job: Data processing. — URL: <https://ytsaurus.tech/docs/ru/user-guide/data-processing/overview>.
5. Challenges and experiences in building an efficient apache beam runner for IBM streams / S. Li [и др.] // Proceedings of the VLDB Endowment. — 2018. — Август. — Т. 11. — С. 1742—1754. — DOI: 10.14778/3229863.3229864.
6. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing / T. Akidau [и др.] // Proceedings of the VLDB Endowment. — 2015. — Т. 8. — С. 1792—1803.
7. YSON, a JSON-like data format. — URL: <https://ytsaurus.tech/docs/en/user-guide/storage/yson>.
8. Protocol Buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. — URL: <https://protobuf.dev/overview/>.
9. *Pominov D.* — 2024.
10. *Valiente G., Martínez C.* An Algorithm for Graph Pattern-Matching. — 1997. — Янв.

11. FlumeJava: Easy, Efficient Data-Parallel Pipelines / C. Chambers [и др.]. — 2010. — URL: <http://dl.acm.org/citation.cfm?id=1806638>.
12. Graphviz. — URL: <http://magjac.com/graphviz-visual-editor/>.
13. Roren YT executor. — URL: <https://github.com/ytsaurus/ytsaurus/tree/main/yt/cpp/roren/yt>.