



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

---

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**  
**К НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ**  
**НА ТЕМУ:**

*«Анализ методов распределенных вычислений в  
распределенных системах хранения информации»*

Студент

ИУ7-75Б

Романов С. К.

Руководитель НИР

Бекасов Д. Е.

*Рекомендуемая руководителем НИР оценка* \_\_\_\_\_

2024 г.

## **РЕФЕРАТ**

Расчетно-пояснительная записка 64 с., 17 рис., 5 табл., 32 ист.

В работе представлен анализ методов распределенных вычислений в распределенных системах хранения информации.

Рассмотрены существующие системы распределенных вычислений. Составлены критерии сравнения распределенных вычислений, рассмотрена классификация систем распределенных вычислений. Составлен вывод на основе полученных результатов.

### **КЛЮЧЕВЫЕ СЛОВА**

Распределенные системы, Распределенные вычисления

# СОДЕРЖАНИЕ

<b>ВВЕДЕНИЕ . . . . .</b>	<b>4</b>
<b>1 Анализ предметной области . . . . .</b>	<b>6</b>
1.1 Распределенные системы . . . . .	6
1.2 Базовые понятия . . . . .	10
1.2.1 Пакетная обработка . . . . .	10
1.2.2 Потоковая обработка . . . . .	11
1.2.3 Алгоритм выбора лидера . . . . .	14
<b>2 Существующие решения . . . . .</b>	<b>15</b>
2.1 Hadoop MapReduce . . . . .	15
2.1.1 Программная модель . . . . .	15
2.1.2 Реализация . . . . .	16
2.1.3 Экосистема Hadoop . . . . .	18
2.2 Apache Spark и RDD . . . . .	20
2.2.1 Программная модель . . . . .	20
2.2.2 Реализация . . . . .	24
2.3 Pregel . . . . .	26
2.3.1 Программная модель . . . . .	26
2.3.2 Реализация . . . . .	28
2.4 DryadLINQ . . . . .	31
2.4.1 Программная модель . . . . .	31
2.4.2 Реализация . . . . .	34
2.5 Google Dataflow . . . . .	36
2.5.1 Программная модель . . . . .	36
2.5.2 Реализация . . . . .	38
2.6 Apache Flink . . . . .	40
2.6.1 Программная модель . . . . .	40

2.6.2	Реализация . . . . .	42
2.7	Apache Samza . . . . .	45
2.7.1	Програмная модель . . . . .	45
2.7.2	Реализация . . . . .	47
<b>ЗАКЛЮЧЕНИЕ . . . . .</b>		<b>51</b>
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ . . . . .</b>		<b>63</b>
<b>ПРИЛОЖЕНИЕ А . . . . .</b>		<b>64</b>

## ВВЕДЕНИЕ

Компьютерная индустрия изменила курс в 2005 году, когда Intel, последовав примеру IBM Power 4 [1] и процессору Niagara [2] от Sun Microsystems, объявили, что их высокопроизводительные микропроцессоры отныне будут опираться на несколько процессоров или ядер [3]. Новое в отрасли слово «многоядерный» отражает план удвоения количества стандартных ядер на матрицу с каждым поколением полупроводниковых процессоров. Многоядерный процессор поможет многопрограммным рабочим нагрузкам, которые содержат набор независимых последовательных задач [4], однако вопросом остается как отдельные задачи станут быстрее. Переход от последовательных вычислений к умеренно параллельным значительно усложняет разработку систем, не вознаграждая эти большие усилия значительно лучшим соотношением производительности к энергопотреблению [5].

Следовательно, многоядерные процессоры едва ли являются идеальным решением. Поскольку решить проблему параллелизма с помощью многоядерных решений, скорее всего, не удастся, то отсюда вытекает нужда в конкретном решении для параллельного аппаратного и программного обеспечения. Однако главная гипотеза заключается не в том, что традиционные научные вычисления и строгие математические модели — это будущее параллельных вычислений; она заключается в том, что совокупность знаний, полученных при создании программ, которые хорошо работают на массово параллельных компьютерах, может оказаться полезной при распараллеливании будущих приложений [6]. Тем более, что многие приложения сегодня требуют обработку больших объемов данных, а не проведение сложных вычислений. Необработанная мощность процессора редко является ограничивающим фактором для подобного рода приложений — более серьезными проблемами обычно являются объем данных, их сложность и скорость, с которой они изменяются.

**Цель работы** — анализ методов распределенных вычислений в распре-

деленных системах хранения информации.

Для достижения поставленной цели требуется решить следующие задачи:

- Провести обзор существующих систем распределенных вычислений;
- Провести анализ подходов к проектированию распределенных вычислений;
- Сформулировать критерии сравнения методов распределенных вычислений;
- Классифицировать существующие методы распределенных вычислений.

# 1 Анализ предметной области

## 1.1 Распределенные системы

За прошедшие годы было представлено несколько определений распределенных систем, но ни одно из них не является удовлетворительными и не согласуются друг с другом. Среди некоторых определений Кулуриса [7] есть одно, которое определяет распределенную систему как «систему, в которой аппаратные или программные компоненты, расположенные на подключенных к сети компьютерах, взаимодействуют и координируют свои действия только путем передачи сообщений».

Еще одно определение, дающее общее представление о распределенных системах звучит следующим образом:

Распределенная система — это вычислительная среда, в которой многочисленные компоненты расположены на нескольких вычислительных устройствах в сети. Эти устройства разделяют работу, обмениваясь данными и координируя свои усилия, чтобы конечному пользователю казаться единой сплошной системой. Распределенная система может представлять собой совокупность различных конфигураций, таких как мейнфреймы, компьютеры, рабочие станции и миникомпьютеры [8].

Любое централизованное решение основано на том, что один узел назначается ответственным за все вычисления проводимые приложением, обрабатывает их локально, и данный узел является общим для всех пользователей системы. Следовательно, существует единая точка контроля и единая точка отказа. Возможно такого рода решение и является оптимальным для небольших задач, но при увеличении нагрузки на систему, возникает, например, необходимость в более отказоустойчивой системе, которая будет обеспечивать доступность данных на протяжении всего времени работы системы.

На рис. 1 показаны различные архитектурные подходы к построению вычислительных систем.

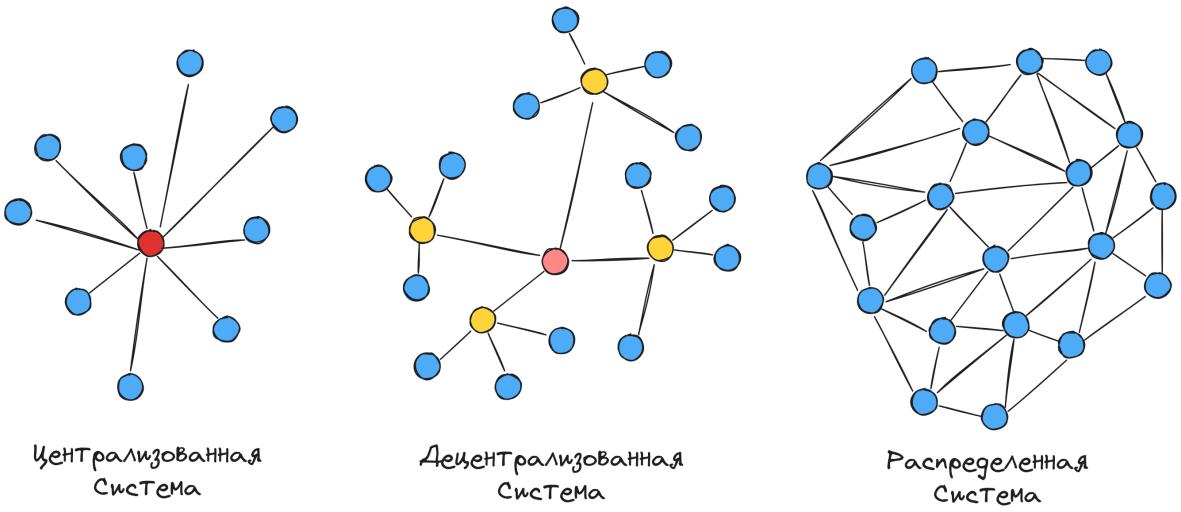


Рисунок 1 – Обзор архитектур вычислительных систем

Мотивацией роста децентрализованных вычислений является доступность недорогих, высокопроизводительных компьютеров и сетевых инструментов. Такая система может обладать более высокой производительностью, чем один конкретный суперкомпьютер. Целью таких систем является минимизация затрат на связь и вычисления. В распределенных системах этапы обработки приложения распределены между участвующими в ней узлами. Основным шагом во всех архитектурах распределенных вычислений является понятие связи между узлами системы.

На рис. 2 представлено сравнение нескольких наиболее важных распределенных систем с тремя суперкомпьютерами,ключенными в список TOP500 [9]. Что касается общей производительности, распределенные системы часто ненамного отстают от технологий, используемых в настоящее время NASA или военными, для функционирования которых требуется значительный финансовый капитал. Для создания диаграммы для суперкомпьютеров были использованы результаты производительности LINPACK [10], которые измеряют скорость решения сложной системы линейных уравнений. С другой стороны, для распределенных систем была собрана информация, предоставленная системными администраторами. Чаще всего они основаны на количестве загруженных результатов за определенный промежуток времени.

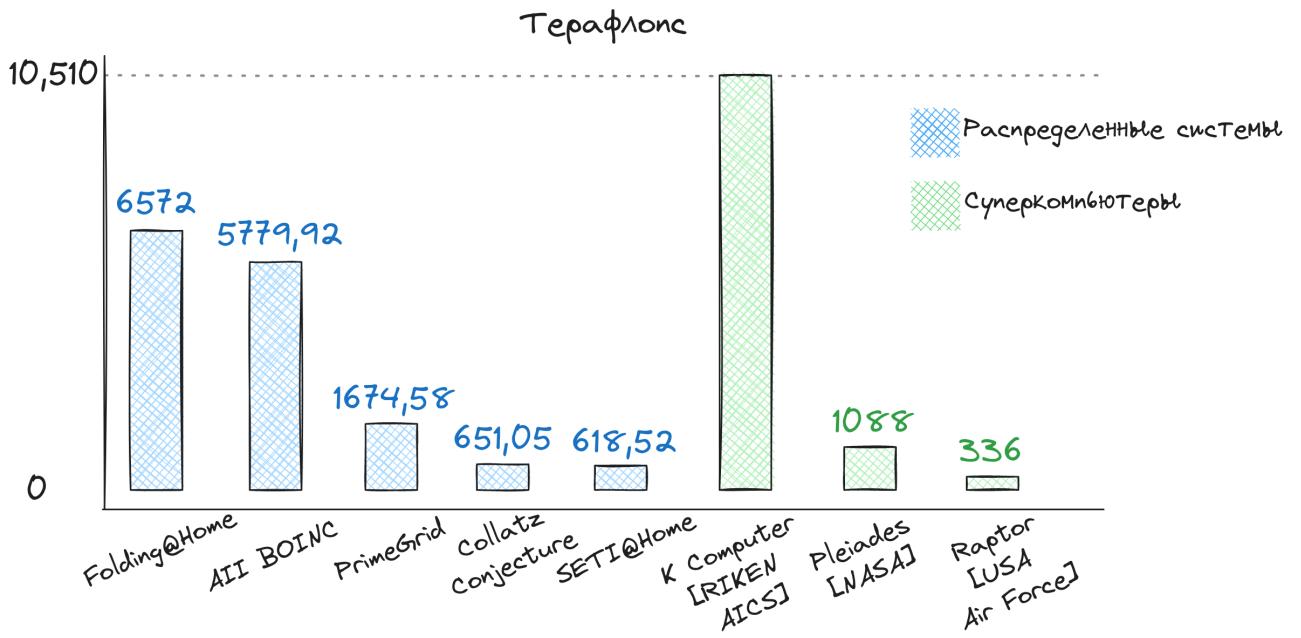


Рисунок 2 – Сравнение производительности различных вычислительных систем.

Приложение, утилизирующее распределенные вычисления, обычно строится из стандартных «блоков», которые предоставляют необходимую функциональность. Так например, приложение может требовать реализации следующих функций:

- хранение данных для дальнейшего к ним доступа (базы данных);
- хранение результата дорогостоящей операции для ускорения чтения (кэширование);
- возможность для пользователей искать данные по ключевому слову или фильтровать их различными способами (поисковые индексы);
- отправление сообщений другому процессу, которые будут обрабатываться асинхронно (потоковая обработка);
- периодическая обработка большого объема накопленных данных (пакетная обработка).

На рис. 3 представлена общая организация абстрактной распределенной системы.

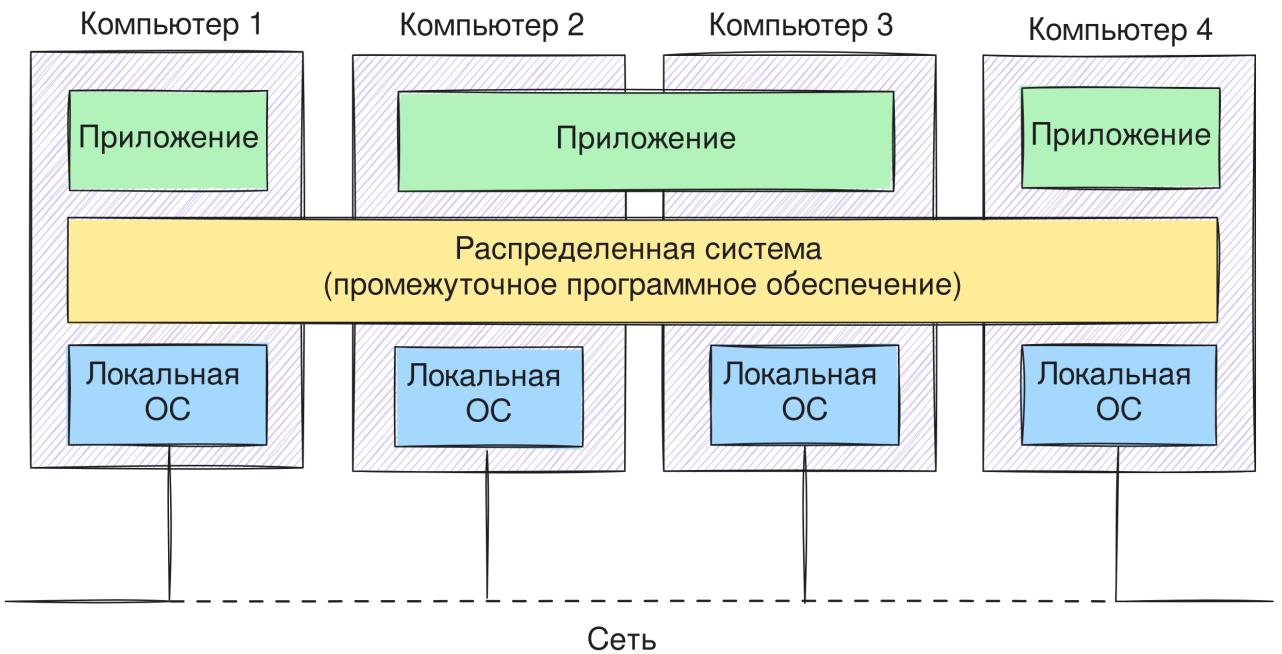


Рисунок 3 – Организация распределенной системы.

Как и к любому другому ПО, к распределенным системам применимы часто рассматриваемые инженерные критерии [11], а именно:

1. **Reliability** — Надежность — Система должна продолжать корректно работать (выдавать корректный результат на желаемом уровне производительности) даже перед лицом неблагоприятных факторов (аппаратных или программных сбоев и возможной человеческой ошибки);
2. **Scalability** — Масштабируемость — По мере роста системы (при увеличении объема данных, трафика или сложности) должны существовать разумные способы и инструменты управления этим ростом;
3. **Maintainability** — Сопровождаемость — Со временем, набор разработчиков, работающих над системой, может сильно изменяться, и все они должны быть в состоянии продуктивно работать над продуктом (проектирование и эксплуатация, как поддержание текущего поведения, так и адаптация системы к новым требованиям).

## **1.2 Базовые понятия**

### **1.2.1 Пакетная обработка**

В производственных условиях пакетная обработка определяется как одновременная обработка нескольких обращений [12]. То же самое может относиться к процессам обслуживания, например, некоему информационному сеансу, который организуется для группы клиентов, а не индивидуально. Однако Батист [13] утверждает, что продолжительность обработки пакета также может быть определена суммой времени обработки всех отдельных обращений. Это намекает на форму пакетной обработки, при которой обращения обрабатываются последовательно.

Из предыдущего следует, что можно различать различные типы пакетной обработки. В целом, пакетная обработка определяется как одновременное, (квази-)последовательное или параллельное выполнение действия в различных случаях одним и тем же ресурсом [14]. Следовательно, рассматриваются три типа пакетной обработки, которые проиллюстрированы на примере на рис. 4, где действие всегда выполняется одним и тем же ресурсом в обоих случаях.

- Одновременная пакетная обработка.

Объекты действия находятся в одном пакете, когда они выполняются одним и тем же ресурсом для разных случаев в одно и то же время. Например: несколько деталей автомобиля, которые необходимо покрасить в один и тот же цвет, можно собрать вместе в покрасочной камере. На рисунке 4 два объекта типа В представляют собой одновременный пакет, поскольку время запуска и завершения в разных случаях соответствует.

- Последовательная пакетная обработка.

Объекты действия находятся в последовательном пакете, когда они выполняются одним и тем же ресурсом для разных случаев сразу или почти сразу друг за другом. Например: сотрудники обрабатывают свои электронные письма только два раза в день, когда они обрабатываются после-

довательно, где может наблюдаться задержка в несколько секунд между обработкой двух разных электронных писем. Два объекта типа А на рисунке 4 образуют последовательный пакет, поскольку начальная временная метка для второго случая соответствует конечной временной метке для первого случая.

- Параллельная пакетная обработка.

Объекты действия находятся в параллельном пакете, когда они выполняются одним и тем же ресурсом для отдельных случаев, частично перекрывающихся по времени. Например: клерк уже может начать бронирование второго счета, когда требуется дополнительная информация для завершения первого. На рисунке 4 объекты С, D, E, F, G представляют типы параллельных пакетов.

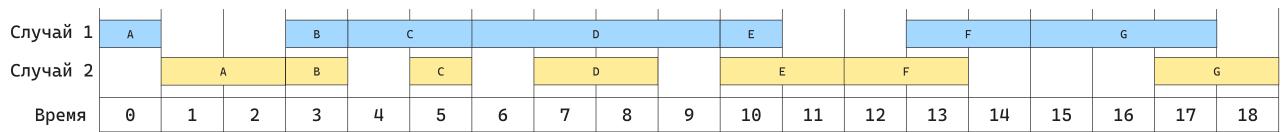


Рисунок 4 – Схематическое представление двух случаев, в которых действие всегда выполняется одним и тем же ресурсом для обоих случаев.

### 1.2.2 Потоковая обработка

Поток данных, используемый при потоковой обработке данных, представляет собой счетную бесконечную последовательность элементов и используется для представления элементов данных, которые становятся доступными с течением времени. Примерами являются показания датчиков в приложении для мониторинга окружающей среды, котировки акций в финансовых приложениях или сетевые данные в приложениях для компьютерного мониторинга. Система с потоковой обработкой данных анализирует элементы из потоков по мере их поступления, чтобы своевременно выдавать новые результаты и при необходимости быстро реагировать на эти данные.

## Модели

Потоки могут быть структурированными или неструктурными. В структурированном потоке элементы соответствуют определенному формату или схеме, что позволяет дополнительно моделировать поток.

Напротив, неструктурные потоки могут иметь произвольное содержимое, часто являющееся результатом объединения потоков из многих источников [15].

Существуют три основные модели структурированных потоков, которые различаются по тому, как элементы потока связаны друг с другом и влияют друг на друга [16]:

1. модель турникета;
2. модель кассового аппарата;
3. модель временных рядов.

Наиболее общей моделью является модель турникета. В этой модели поток моделируется как вектор элементов, и каждый элемент  $S_i$  в потоке представляет собой модификацию (увеличение или уменьшение) элемента базового вектора. Размер вектора в этой модели является всей областью элементов потока. Эта модель также является моделью, обычно используемой в традиционных системах баз данных, для которой определены CRUD операции.

В модели кассового аппарата элементы в потоке являются только дополнениями к базовому вектору, и никогда больше не смогут покинуть этот вектор. Это похоже на базы данных, записывающие историю отношений.

Наконец, модель временных рядов рассматривает каждый элемент в потоке  $S_i$  как новую независимую векторную запись. В результате базовая модель представляет собой постоянно увеличивающийся вектор и, как правило, неограниченный. Поскольку в этой модели каждый элемент может обрабатываться индивидуально, он часто используется в современных механизмах потоковой обработки.

## **Время**

Время является центральным понятием во многих системах для обработки потоков, либо потому, что они заинтересованы в обновлении своего взгляда на мир с учетом последних данных, полученных из потоков, либо потому, что они нацелены на выявление временных тенденций во входных потоках [16].

По этой причине в большинстве моделей потоковой обработки элементы данных связаны с некоторой временной меткой из заданной временной области. Общая семантика времени включает время события, которое является временем создания элемента, и время обработки, которое является временем, когда система потоковой обработки начинает обрабатывать элементы [17].

Различная семантика времени создает различные проблемы с точки зрения порядка и синхронизации. Интуитивно понятно, что, хотя время события и время обработки в идеале должны быть равны, на практике несинхронизированные часы систем, а также переменные задержки связи и обработки приводят к перекосу между временем события и временем обработки, который не только отличен от нуля, но и сильно варьируется [17].

## **Окна**

Окна являются одной из основных частей практически всех систем потоковой обработки. Они определяют ограниченное количество элементов в неограниченном потоке данных. Они используются для выполнения вычислений, которые были бы невозможны (бесконечны) в случае неограниченных данных, таких как вычисление среднего значения всех элементов в потоке чисел. Наиболее распространенными типами окон являются окна, основанные на подсчете и времени. Первые определяют свой размер в терминах количества элементов, которые они включают, в то время как вторые определяют свой размер в терминах временного интервала и включают все элементы с временной меткой, включенные в этот временной интервал.

В обоих случаях мы различаем плавающие окна, которые непрерывно перемещаются с появлением новых элементов, таким образом, всегда захватывая

новые элементы, и сменяющиеся окна, которые могут накапливать несколько элементов перед перемещением [18].

Совсем недавно были определены новые типы окон, чтобы лучше отражать потребности приложений. Они включают в себя окна, определяемые сеансом и данными, которые изменяются по размеру и определяют свои границы на основе элементов данных: например, в приложении для мониторинга программного обеспечения окно может включать все и только те элементы, которые относятся к сеансу, открытому конкретным пользователем этого программного обеспечения [17].

### **1.2.3 Алгоритм выбора лидера**

В распределенных алгоритмах выборы лидера — это процесс назначения одного процесса организатором, координатором, инициатором или последовательностью выполнения некоторой задачи, распределенной между несколькими компьютерами (узлами). Другими словами, выбор лидера — это процесс определения того, что процесс будет управлять некоторой задачей, распределенной между несколькими процессами или узлами [19].

Ниже приведены причины выбора лидера:

1. Централизованное управление упрощает синхронизацию процессов, но в таком случае это приведет к появлению единой точки отказа системы.
2. Предоставить решение для выбора нового управляющего узла (лидера) при отказе существующего лидера.
3. Существует множество алгоритмов выбора лидера, и один из них может быть выбран на основе конкретных требований. Примерами таких алгоритмов являются алгоритм забияки или алгоритм Чанга и Робертса

Существует два критерия отбора:

- Поиск экстремумов: основан на глобальном приоритете. Каждый процесс характеризуется фиксированным значением оценки.
- Основанный на предпочтениях: Процессы в группе могут голосовать за лидера на основе личных предпочтений.

## **Вывод**

В данном разделе были рассмотрены основные понятия, связанные с распределенными вычислениями, и представлены основные алгоритмы, используемые в таких системах. Также были рассмотрены причины использования распределенных систем и сравнение их производительности с суперкомпьютерами. Таким образом, в данном разделе была представлена общая картина распределенных вычислений, которая позволяет понять проблематику, связанную с этой областью, и рассматривать их в контексте реальных систем.

## **2 Существующие решения**

### **2.1 Hadoop MapReduce**

MapReduce — это программная модель для обработки генерации больших наборов данных и связанная с ней реализация в виде системы Hadoop. Пользователи задают функцию сопоставления (Map), которая обрабатывает некоторую пару ключ/значения для генерации набора промежуточных пар ключ/значение, и функцию уменьшения (Reduce), которая объединяет все промежуточные значения, связанные с одним и тем же промежуточным ключом. Программы, написанные в этом функциональном стиле, автоматически распараллеливаются и выполняются на большом кластере обычных машин. Система выполнения заботится о разделении входных данных, планирования выполнения программы на множестве машин, обработки сбоев машин и управления необходимой межмашинной связью [20].

#### **2.1.1 Программная модель**

Вычисления в MapReduce используют набор входных пар ключ/значение и создает набор выходных пар ключ/значение. Пользователь библиотеки на основе MapReduce выражает вычисления в виде двух функций: Map и Reduce.

Функция Map, написанная пользователем, принимает входную пару и создает набор промежуточных пар ключ/значение. Алгоритм MapReduce группи-

рует все промежуточные значения, связанные с одним и тем же промежуточным ключом  $I$ , и передает их функции Reduce.

Функция Reduce, также написанная пользователем, принимает промежуточный ключ  $I$  и набор значений для этого ключа. Оно объединяет эти значения для формирования возможно меньшего набора значений. Обычно при каждом вызове Reduce создается только нулевое или одно выходное значение. Промежуточные значения передаются в пользовательскую функцию Reduce через итератор. Это позволяет нам обрабатывать списки значений, которые слишком велики, чтобы поместиться в памяти.

### 2.1.2 Реализация

В качестве примера реализации алгоритма MapReduce была взята имплементация системы Hadoop, разработанная в рамках Apache Foundation, основанная на публикации сотрудников компании Google [20]. На рис. 5 изображена общая схема алгоритма.

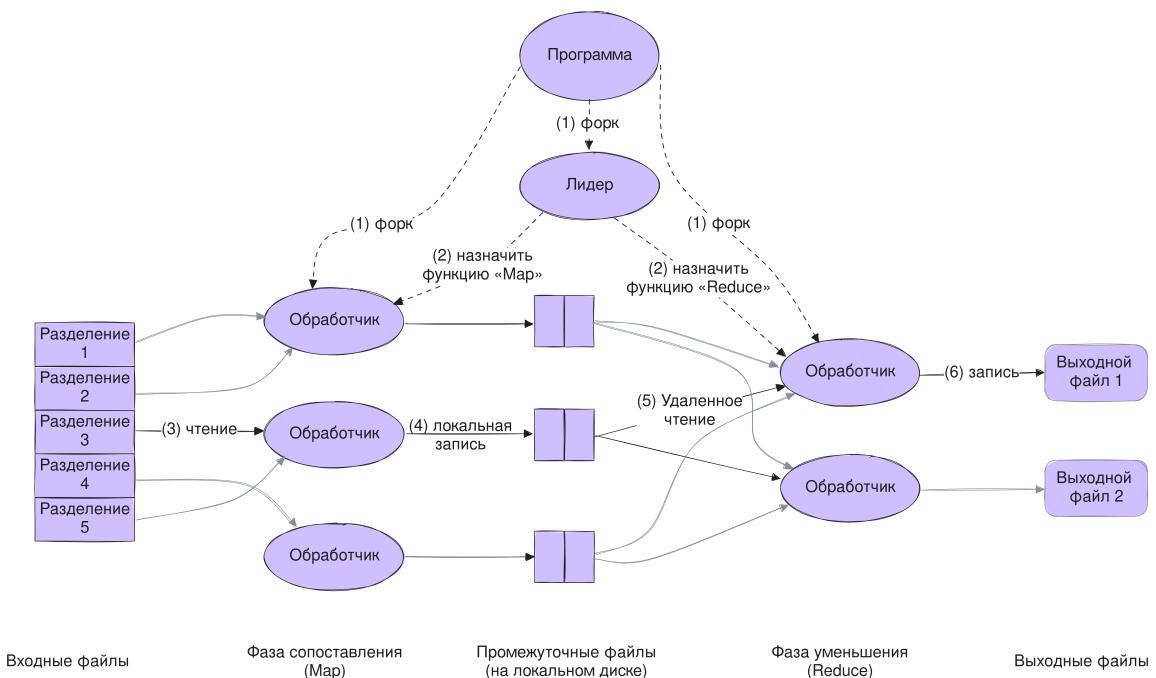


Рисунок 5 – Схема работы алгоритма MapReduce

Когда пользовательская программа вызывает функцию MapReduce, выполняется следующая последовательность действий (нумерованные метки на

рис. 5 соответствуют номерам в списке ниже):

1. Алгоритм разбивает входные файлы на  $M$  фрагментов, которые затем запускает множество копий программы на кластере компьютеров.
2. Одна из копий программы является специальной — `master`. Остальные — это процессы, которым `master` назначает работу. В общем итоге назначается  $M$  задач типа `Map` и  $R$  задач типа `Reduce`.
3. Процесс, которому назначена задача типа `Map`, считывает содержимое соответствующего фрагмента данных. Происходит анализ пары ключ/значение из входных данных и передает каждую пару в определенную пользователем функцию `Reduce`. Промежуточные пары ключ/значение, созданные функцией `Reduce` буферизуются в памяти.
4. Буферизованные пары записываются на локальный диск, разбитые на  $R$  фрагментов с помощью функции секционирования. Расположение этих буферизованных пар на локальном диске передается обратно ведущему процессу, который отвечает за пересылку этих местоположений процессам с назначенными функциями `Reduce`.
5. Когда ведущий процесс уведомляет рабочий процесс с назначенной функцией `Reduce` об этих местоположениях, он использует удаленные вызовы процедур для считывания буферизованных данных с локальных дисков рабочих процессов типа `Map`. Когда рабочий процесс типа `Reduce` прочитал все промежуточные данные, он сортирует их по промежуточным ключам, чтобы все вхождения одного и того же ключа были сгруппированы вместе. Сортировка необходима, потому что обычно много разных ключей сопоставляются с одной и той же задачей `Reduce`.
6. Рабочий процесс типа `Reduce` выполняет итерацию по отсортированным промежуточным данным и для каждого встреченного уникального промежуточного ключа передает ключ и соответствующий набор промежуточных значений пользовательской функции `Reduce`. Выходные данные функции `Reduce` добавляются в конечный выходной файл для этого раз-

деля Reduce.

- Когда все задачи Map и Reduce завершены, ведущий процесс запускает пользовательскую программу. На этом этапе вызов MapReduce в пользовательской программе возвращается обратно к пользовательскому коду.

После успешного завершения MapReduce выходные данные помещаются в  $R$  выходных файлах (по одному для каждой задачи Reduce, с именами файлов, указанными пользователем). Как правило, пользователям не нужно объединять эти выходные файлы  $R$  в один файл — зачастую эти файлы передаются в качестве входных данных другому вызову MapReduce или используют их в другом распределенном приложении, способного обрабатывать входные данные, разделенные на несколько файлов.

### 2.1.3 Экосистема Hadoop

Различные проекты экосистемы Apache Hadoop взаимосвязаны, как показано на рисунке 6.

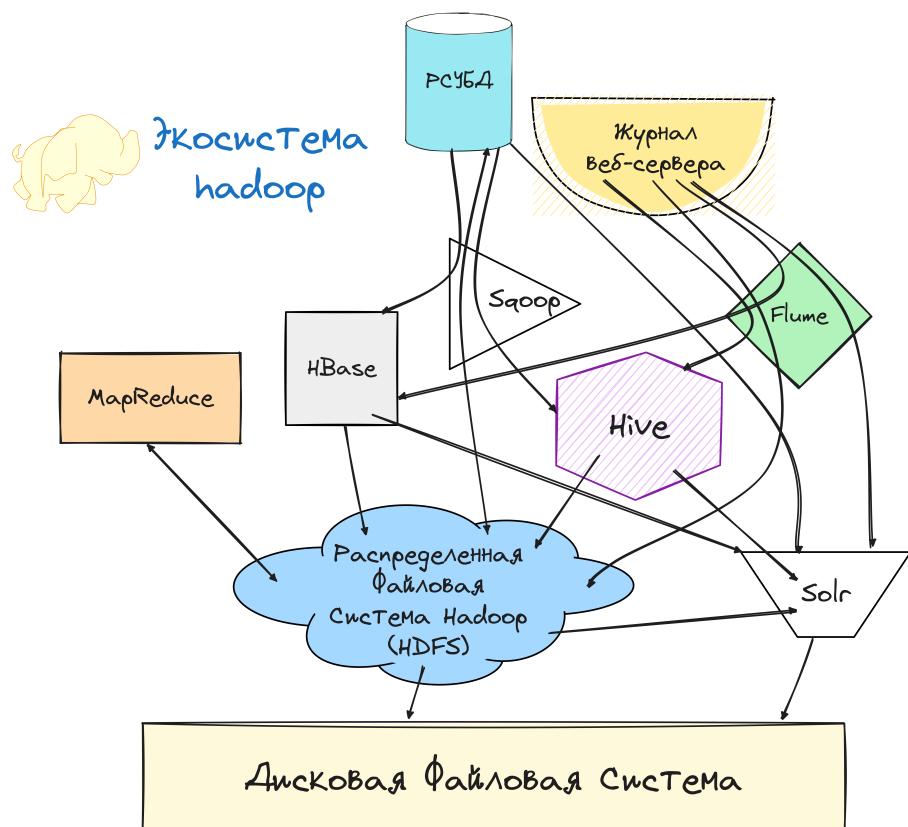


Рисунок 6 – Экосистема Apache Hadoop

MapReduce обрабатывает данные, хранящиеся в HDFS. HBase — это NoSQL база данных, которая поддерживает все виды данных и, следовательно, способна обрабатывать все, что есть в базе данных Hadoop. Hive — система управления базами данных с SQL-подобным языком запросов, позволяет выполнять запросы, агрегировать и анализировать данные системы. По умолчанию HBase и Hive хранят данные в HDFS. Sqoop используется для массовой передачи данных из реляционной системы управления базами данных (РСУБД) в HDFS, Hive и HBase. Sqoop также поддерживает массовую передачу данных из HDFS в реляционную базу данных. Flume используется для потоковой передачи данных журнала, приемником которой может быть HDFS, Hive, HBase или Solr. Flume, основанный на источниках и приемниках, поддерживает несколько видов источников и приемников с акцентом на потоковую передачу данных в режиме реального времени в отличие от одноразовой массовой передачи с помощью Sqoop. Solr используется для индексации данных из HDFS, Hive, HBase и СУБД. HDFS хранит данные в дисковой файловой системе и фактически является абстрактной файловой системой поверх дисковой файловой системы. Solr также хранит данные в дисковой файловой системе по умолчанию, но также может хранить индексированные данные в HDFS [21].

Помимо вышеперечисленных компонентов, есть и некоторые другие, которые выполняют задачу сделать Hadoop способным обрабатывать большие наборы данных:

- YARN — подсистема, помогающая управлять ресурсами в кластерах, состоящая из 3 компонентов: менеджер ресурсов, менеджер узлов и менеджер приложений [22]. YARN выполняет планирование и распределение ресурсов для системы Hadoop. Менеджер ресурсов имеет привилегию выделять ресурсы для приложений в системе, в то время как менеджер узлов работает над распределением ресурсов, таких как процессорное, память, пропускная способность, а затем подтверждают это менеджеру ресурсов. Менеджер приложений работает как интерфейс между менеджером ре-

сурсов и менеджером узлов и выполняет согласование в соответствии с требованиями обоих.

- Apache ZooKeeper — открытая программная служба для координации распределённых систем, организованная на основе резидентной базы данных категорий «ключ — значение». Изначально входила в экосистему Hadoop, впоследствии стала проектом верхнего уровня Apache Software Foundation
- Pig — платформа для структурирования потока данных, обработки и анализа огромных массивов данных. Pig выполняет работу по выполнению команд, где в фоновом режиме выполняются все действия MapReduce. После обработки Pig сохраняет результат в HDFS. Язык Pig Latin специально разработан для этого фреймворка, который работает в собственной среде выполнения Pig Runtime.

## 2.2 Apache Spark и RDD

### 2.2.1 Программная модель

Apache Spark — это унифицированный аналитический движок для крупномасштабной обработки данных [23]. Основной абстракцией в Spark является устойчивый распределенный набор данных (Resilient Distribute Dataset – RDD), который представляет собой доступную только для чтения коллекцию объектов, распределенных по набору машин, которые могут быть восстановлены в случае потери раздела.

Чтобы использовать Spark, разработчики пишут программу-драйвер, которая реализует высокоуровневый поток управления их приложением и параллельно запускает различные операции. Spark предоставляет две основные абстракции для распределения вычислений: устойчивые распределенные наборы данных и параллельные операции с этими наборами данных (вызываемые путем передачи функции для применения к набору данных). Кроме того, Spark поддерживает два ограниченных типа общих переменных, которые могут использоваться в функциях, запущенных в кластере [24].

## RDD

Формально RDD — это доступная только для чтения секционированная коллекция записей. RDD могут быть созданы только с помощью детерминированных операций либо с данными в некотором стабильном хранилище, либо с другими RDD. Данные операции называются преобразованиями, чтобы отличать их от других операций с RDD. Примеры преобразований включают такие операции как отображение (*map*), фильтрацию (*filter*) и объединение (*join*).

Элементы RDD необязательно должны существовать в физическом хранилище; вместо этого дескриптор RDD содержит достаточно информации для вычисления RDD, начиная с данных в хранилище. Это означает, что RDD всегда можно восстановить в случае сбоя узлов.

В Spark каждый RDD представлен объектом Scala. Spark позволяет программистам создавать RDD четырьмя способами:

- Из файла в общей файловой системе.
- Путем «распараллеливания» коллекции Scala (например, массива) в программе драйвера, что означает разделение ее на несколько фрагментов, которые будут отправлены нескольким узлам.
- Путем преобразования существующего RDD. Набор данных с элементами типа *A* может быть преобразован в набор данных с элементами типа *B* с помощью операции, называемой *flatMap*, которая передает каждый элемент через предоставленный пользователем функция типа  $A \rightarrow List[B]$ . Другие преобразования могут быть выражены с помощью *flatMap*, включая *Map* (передача элементов через функцию типа  $A \rightarrow B$ ) и *Filter* (выбор элементов, соответствующих предикату).

Стоит отметить, что *flatMap* имеет ту же семантику, что *Map* в MapReduce.

- Путем изменения времени жизни существующего RDD. По умолчанию RDD являются ленивыми и эфемерными. То есть разделы набора данных материализуются по требованию, когда они используются в параллельной операции (например, путем передачи блока файла через функцию *Map*),

и удаляются из памяти после использования. Однако пользователь может изменить время жизни RDD с помощью двух действий:

1. Кэширование. Данное действие оставляет RDD ленивым, но намекает на то, что его следует сохранить в памяти после первого вычисления, потому что он будет использован повторно.
2. Сохранить. Данное действие оценивает набор данных и записывает его в распределенную файловую систему. Сохраненная версия используется в будущих операциях.

## Параллельные операции

Ниже перечислены основные преобразования RDD и действия, доступные в Spark. Следует отметить, что преобразования — это отложенные (ленивые) операции, которые определяют новый RDD, в то время как действия запускают вычисление для возврата значения программе или записи данных во внешнее хранилище.

Трансформации:

- $map(f : T \rightarrow U) : RDD[T] \rightarrow RDD[U]$
- $filter(f : T \rightarrow Bool) : RDD[T] \rightarrow RDD[T]$
- $flatMap(f : T \rightarrow Seq[U]) : RDD[T] \rightarrow RDD[U]$
- $sample(fraction : Float) : RDD[T] \rightarrow RDD[T]$  (Детерминированная выборка)
- $groupByKey() : RDD[(K, V)] \rightarrow RDD[(K, Seq[V])]$
- $reduceByKey(f : (V, V) \rightarrow V) : RDD[(K, V)] \rightarrow RDD[(K, V)]$
- $union() : (RDD[T], RDD[T]) \rightarrow RDD[T]$
- $join() : (RDD[(K, V)], RDD[(K, W)]) \rightarrow RDD[(K, (V, W))]$
- $cogroup() : (RDD[(K, V)], RDD[(K, I)]) \rightarrow RDD[(K, (Seq[V], Seq[I]))]$
- $crossProduct() : (RDD[T], RDD[U]) \rightarrow RDD[(T, U)]$
- $mapValues(f : V \rightarrow W) : RDD[(K, V)] \rightarrow RDD[(K, W)]$  (Сохраняет разделения)
- $sort(c : Comparator[K]) : RDD[(K, V)] \rightarrow RDD[(K, V)]$

- $\text{partitionBy}(p : \text{Partitioner}[K]) : \text{RDD}[(K, V)] \rightarrow \text{RDD}[(K, V)]$

Действия:

- $\text{count}() : \text{RDD}[T] \rightarrow \text{Long}$
- $\text{collect}() : \text{RDD}[T] \rightarrow \text{Seq}[T]$
- $\text{reduce}(f : (T, T) \rightarrow T) : \text{RDD}[T] \rightarrow T$
- $\text{lookup}(k : K) : \text{RDD}[(K, V)] \rightarrow \text{Seq}[V]$  (Воспроизводится над хэш/набор данных разделенном RDD)
- $\text{save(path : String)}$ : Сохраняет RDD в файловой системе

### Разделяемые переменные

Такие операции, как Map, Filter и Reduce, вызываются путем передачи функций в Spark. Обычно, эти функции могут ссылаться на переменные в области, в которой они созданы. Когда Spark запускает функции на рабочем узле, эти переменные в него копируются. Однако Spark также позволяет программистам создавать два ограниченных типа общих переменных для поддержки двух простых, но распространенных шаблонов использования:

- Широковещательные переменные: Если большой фрагмент данных, доступный только для чтения (например, таблица поиска), используется в нескольких параллельных операциях, предпочтительнее распространять его среди рабочих только один раз, вместо того чтобы копировать его при каждом вызове функции. Spark позволяет программисту создать такой объект, который переносит значение и гарантирует, что оно копируется каждому процессу только один раз.
- Накопители: Это переменные, к которым рабочие процессы могут добавлять данные только с помощью ассоциативной операции и которые может считывать только драйвер. Их можно использовать для реализации счетчиков, как в MapReduce, и для обеспечения более императивного синтаксиса для параллельных сумм. Аккумуляторы могут быть определены для любого типа, который имеет операцию «добавить» и значение «ноль».

## 2.2.2 Реализация

Spark построен поверх Mesos, «кластерной операционной системы», которая позволяет нескольким параллельным приложениям совместно использовать кластер компьютеров и предоставляет API для приложений для запуска задач в этом кластере [25]. Это позволяет Spark работать совместно с существующими кластерными вычислительными платформами, такими как Mesos-адаптациями для Hadoop и MPI, и обмениваться с ними данными. Кроме того, использование Mesos значительно сократило затраты на программирование, которые приходилось затрачивать на Spark.

На рисунке 7 показано верхнеуровневое представление системы Spark.

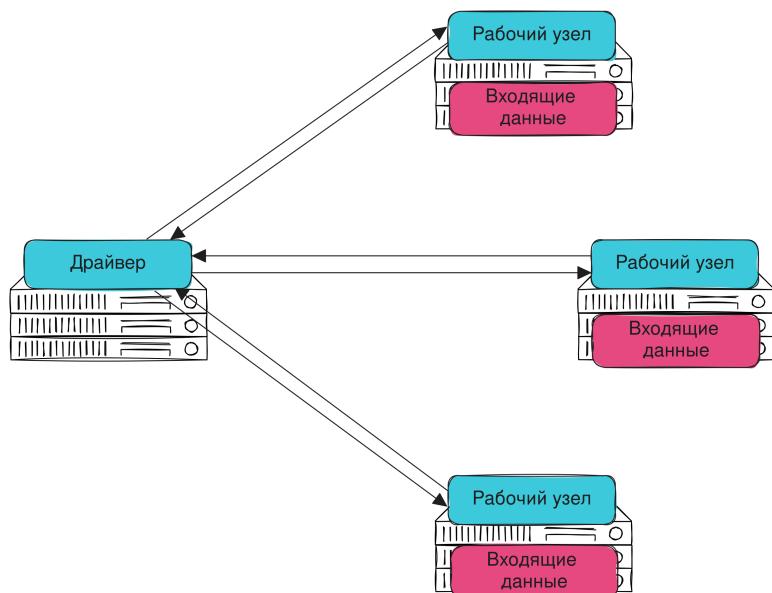


Рисунок 7 – Верхнеуровневое разделение Spark

Важным моментом при разработке интерфейса для взаимодействия RDD заключается в том, как представить зависимости между этими RDD. Классифицировать такого рода зависимости можно на два типа: узкие зависимости, где каждый раздел родительского RDD используется не более чем одним разделом дочернего RDD, и широкие зависимости, где от RDD может зависеть несколько дочерних разделов. Рисунок 8 демонстрирует примеры зависимостей RDD для некоторых операций.

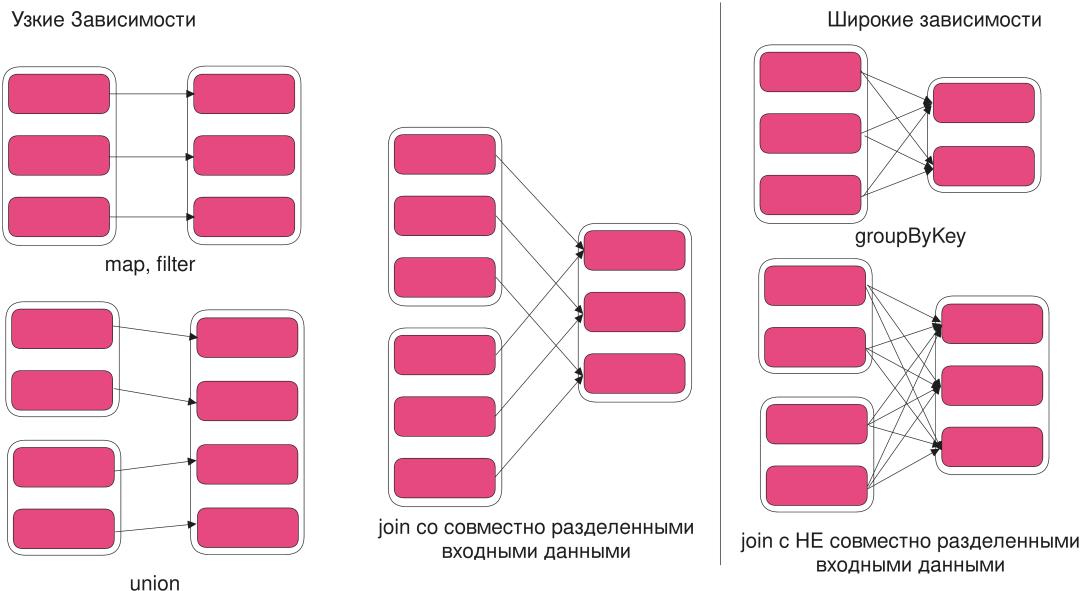


Рисунок 8 – Узкие и широкие зависимости

Всякий раз, когда пользователь запускает действие (например, *count* или *save*) в RDD, планировщик проверяет линейный график этого RDD, чтобы построить список этапов для выполнения, как показано на рисунке 9. Каждый этап как правило содержит наибольшее возможное количество конвейерных преобразований с узкими зависимостями. Границами этапов являются операции перетасовки, необходимые для широких зависимостей, или любые уже вычисленные разделы, хранящиеся в памяти, которые могут прервать вычисление родительского RDD. Затем планировщик запускает задачи для вычисления недостающих разделов на каждом этапе, пока не будет вычислен целевой RDD.

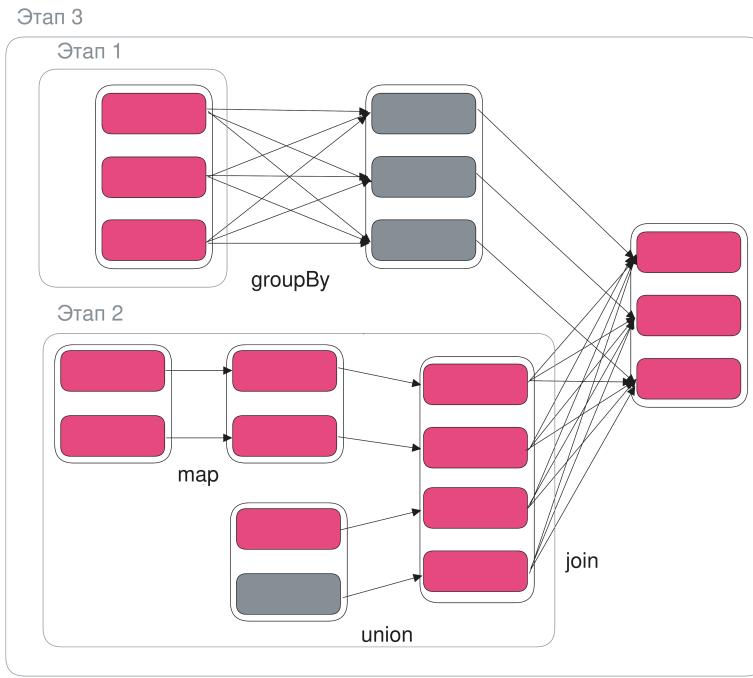


Рисунок 9 – Формирование этапов в Spark. Прямоугольники со сплошными контурами — это RDD. Разделы — это заштрихованные прямоугольники, при этом некоторые из них выделены серым цветом, если они уже есть в памяти. Чтобы выполнить действие в RDD, создаются этапы сборки с широкими зависимостями и проводятся узкие преобразования внутри каждого этапа. В этом случае выходной RDD этапа 1 уже находится в оперативной памяти, поэтому есть возможность сразу запустить этап 2, а затем этап 3.

## 2.3 Pregel

### 2.3.1 Программная модель

Pregel является системой для обработки большеразмерных графов, разработанной компанией Google [26]. Архитектура Pregel-программ основана на объемной синхронно-параллельной модели объемной синхронно - параллельной модели Валианта [27].

Входными данными для вычислений в Pregel является ориентированный граф, в котором каждая вершина однозначно идентифицируется строковым идентификатором вершины. Каждая вершина связана с изменяемым значением, определяемым пользователем. Направленные ребра связаны со своими ис-

ходными вершинами, и каждое ребро состоит из изменяемого, определенного пользователем значения и идентификатора целевой вершины.

Вычисления в Pregel состоят из последовательности итераций, называемых супершагами, разделенных точками глобальной синхронизации. Во время супершага фреймворк вызывает определенную пользователем функцию для каждой вершины, практически параллельно. Данная функция определяет поведение в одной вершине  $V$  и на одном супершаге  $S$ . Она может считывать сообщения, отправленные в  $V$  на супершаге  $S-1$ , отправлять сообщения другим вершинам, которые будут получены на супершаге  $S + 1$ , и изменять состояние  $V$  и его исходящих ребер. Сообщения обычно отправляются по исходящим ребрам, но оно может быть отправлено в любую вершину, идентификатор которой известен.

Завершение алгоритма основано на том, что каждая вершина графа голосует за остановку. На супершаге 0 каждая вершина находится в активном состоянии; все активные вершины участвуют в вычислении любого данного супершага. Вершина деактивирует себя, проголосовав за остановку алгоритма. Это означает, что вершине больше не нужно выполнять никакой работы, если она не запущена извне, и Pregel не будет выполнять эту вершину на последующих суперэтапах, пока не получит сообщение. Если вершина повторно активирована сообщением, она должна явно деактивировать себя снова. Алгоритм завершается, когда все вершины одновременно неактивны и нет передаваемых сообщений. Этот конечный автомат проиллюстрирован на рисунке 10

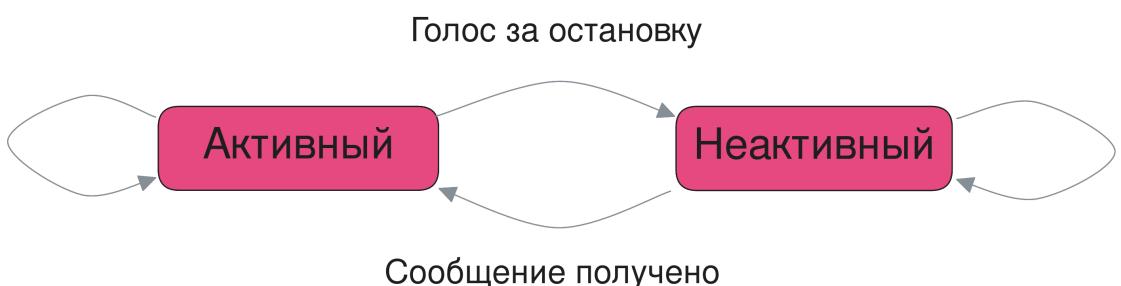


Рисунок 10 – Вершинный конечный автомат.

Выходные данные Pregel — это набор значений, явно выводимых вершинами. Зачастую это ориентированный граф, изоморфный входным данным, но это не является необходимым свойством системы, поскольку вершины и ребра могут добавляться и удаляться во время вычислений. Так например, алгоритм кластеризации может генерировать небольшой набор несвязанных вершин, выбранных из большого графа. Алгоритм интеллектуального анализа графа может просто выводить агрегированную статистику, полученную из графа.

### 2.3.2 Реализация

Pregel был разработан для кластерной архитектуры Google. Каждый кластер состоит из тысяч компьютеров, организованных в стойки с высокой пропускной способностью внутри стойки. Кластеры взаимосвязаны, но распределены географически.

Pregel делит граф на разделы, каждый из которых состоит из набора вершин и всех исходящих ребер этих вершин. Присвоение вершины разделу зависит исключительно от идентификатора вершины, что подразумевает возможность узнать, к какому разделу принадлежит данная вершина, даже если вершина принадлежит другому компьютеру или даже если вершина еще не существует. Функция разбиения на разделы по умолчанию — это  $\text{hash}(ID) \bmod N$ , где  $N$  — количество разделов, которое пользователи могут изменить.

В приведенной на рисунке 11 последовательности показаны четыре суперэтапа, необходимые для завершения вычисления максимального значения для графа с четырьмя узлами. На каждом шаге каждая вершина считывает все входящие сообщения и устанавливает максимальное значение для своего текущего значения и тех, которые были ей отправлены. Затем он отправляет это максимальное значение по всем своим ребрам. Если максимальное значение в узле не изменяется во время супершага, узел затем голосует за остановку.

└ - Входные данные

● - Активный

○ - Остановленный

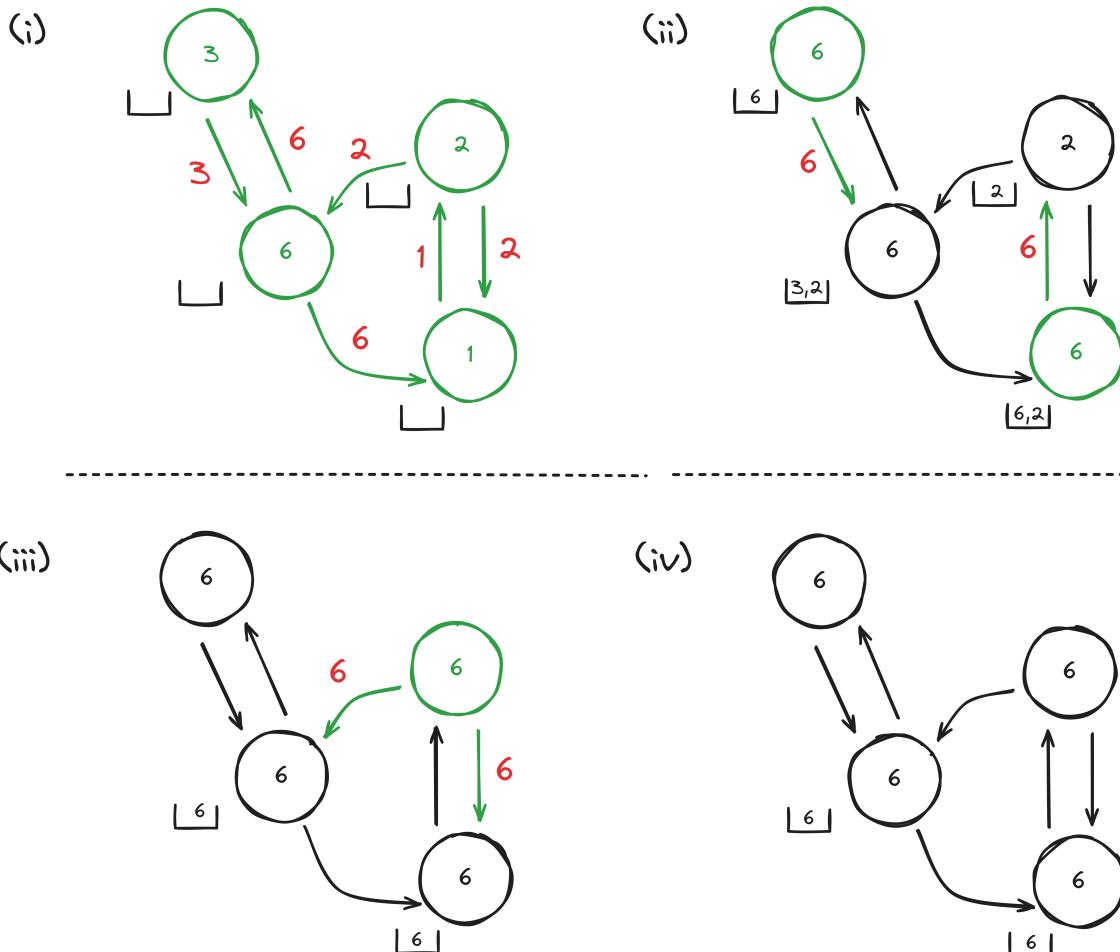


Рисунок 11 – Пример работы Pregel

При отсутствии сбоев выполнение программы Pregel состоит из нескольких этапов:

1. Множество копий пользовательской программы начинают выполняться на кластере машин. Одна из этих копий действует как ведущая, (далее «мастер»). «Мастеру» не назначается какая-либо часть графа, но он отвечает за координацию действий рабочих узлов. Рабочие машины используют систему управления кластером, чтобы определить местоположение «мастер» и отправить ей регистрационные сообщения.
2. «Мастер» определяет, сколько разделов будет иметь граф, и назначает

один или несколько разделов каждой рабочей машине. Количество может контролироваться пользователем. Наличие более одного раздела на рабочую машину обеспечивает параллелизм между разделами и лучшую балансировку нагрузки и, как правило, повышает производительность. Каждый рабочий процесс отвечает за поддержание состояния своего участка графа, выполнение пользовательского метода `Compute()` в его вершинах и управление сообщениями для других рабочих узлов и от них. Каждому рабочему узлу предоставляется полный набор назначений для всех узлов.

3. «Мастер» назначает часть пользовательских данных каждому рабочему узлу. Входные данные обрабатываются как набор записей, каждая из которых содержит произвольное количество вершин и ребер. Разделение входных данных ортогонально разбиению самого графа и обычно основано на границах файла. Если рабочий узел загружает вершину, принадлежащую этому рабочему участку графа, соответствующие структуры данных немедленно обновляются. В противном случае данный узел помещает сообщение в очередь удаленному узлу, которому принадлежит вершина. После завершения загрузки входных данных все вершины помечаются как активные.
4. «Мастер» инструктирует каждый узел выполнить супершаг. Рабочие узлы выполняет цикл по своим активным вершинам, используя по одному потоку для каждого раздела. Узлы вызывают `Compute()` для каждой активной вершины, доставляя сообщения, которые были отправлены на предыдущем супершаге. Сообщения отправляются асинхронно для обеспечения наложения вычислений, обмена данными и пакетной обработки, но доставляются сообщения до окончания супершага. Когда рабочий процесс завершен, он сообщает «мастеру», сколько вершин будет активно на следующем суперэтапе. Этот шаг повторяется до тех пор, пока активны какие-либо вершины или пока передаются какие-либо сообщения.

5. После остановки вычислений «мастер» может дать указание каждому узлу сохранить свою часть графа.

На рисунке 12 изображена общая архитектура Google Pregel.

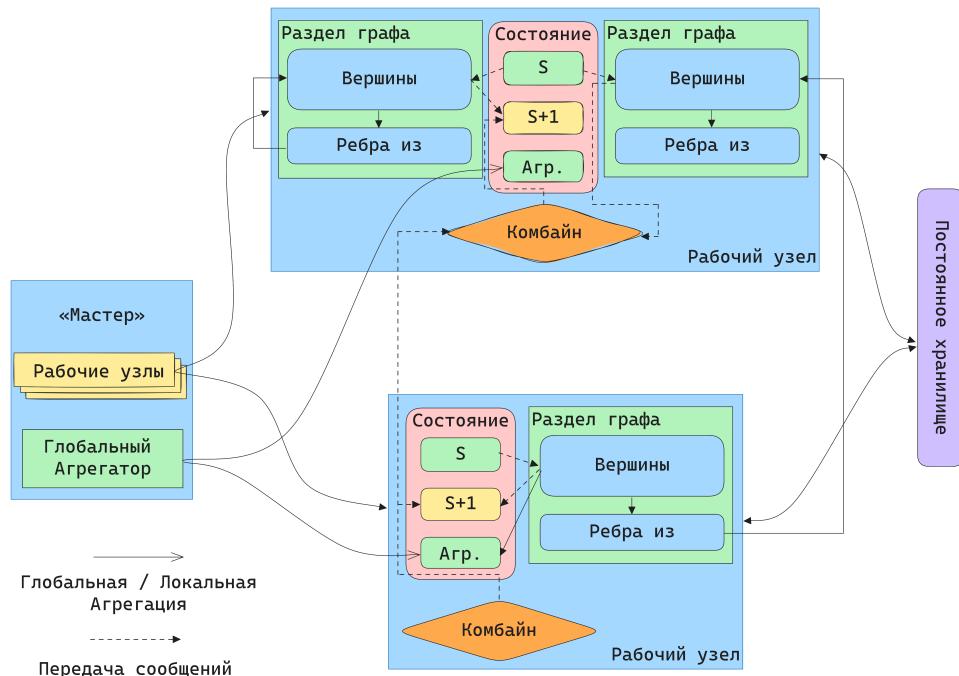


Рисунок 12 – Архитектура Google Pregel

## 2.4 DryadLINQ

### 2.4.1 Программная модель

DryadLINQ описывает систему для распределения вычислений .NET LINQ выражений на кластере Dryad. Цель DryadLINQ — сделать распределенные вычисления на большом вычислительном кластере достаточно простыми для каждого программиста. DryadLINQ сочетает в себе две важные части технологии Microsoft: механизм распределенного выполнения Dryad и интегрированные запросы на языке .NET (LINQ) [28].

Термин LINQ относится к набору конструкций .NET для манипулирования наборами и последовательностями элементов данных. Базовым типом для LINQ коллекции является тип `IEnumerable<T>`. Это абстрактный набор данных объектов типа  $T$ , доступ к которому осуществляется с помощью интерфейса итератора. LINQ также определяет интерфейс `IQueryable<T>`, который являет-

ся подтипом `IEnumerable<T>` и представляет собой (неоцененное) выражение, построенное путем объединения наборов данных LINQ с использованием операторов LINQ [29].

DryadLINQ сохраняет модель программирования LINQ и расширяет ее для параллельного программирования, определяя небольшой набор новых операторов и типов данных. Входные и выходные данные DryadLINQ вычислений представлены объектами типа `DryadTable<T>` – подтипом `IQueryable<T>`. Подтипы `DryadTable<T>` поддерживают внутренних поставщиков хранилищ, которые включают в себя распределенные файловые системы, коллекции файлов NTFS и наборы таблиц SQL.

Основное ограничение, налагаемое системой DryadLINQ для обеспечения распределенного выполнения, заключается в том, что все функции, вызываемые в выражениях DryadLINQ, должны быть свободны от побочных эффектов. На общие объекты можно ссылаться и читать их свободно, и они будут автоматически сериализованы и распространены там, где это необходимо. Однако, если какой-либо общий объект изменен, результат вычисления не определен. DryadLINQ не проверяет и не обеспечивает отсутствие побочных эффектов.

DryadLINQ предлагает два оператора для повторного разбиения данных: `HashPartition<T,K>` и `RangePartition<T,K>`. Эти операторы необходимы для принудительного разбиения выходного набора данных, и они также могут использоваться для переопределения выбранного оптимизатором плана выполнения. Однако с точки зрения LINQ они не являются операционными задачами, поскольку они просто реорганизуют коллекцию без изменения ее содержимого.

Оставшимися новыми операторами являются `Apply` и `Fork`, которые можно рассматривать как «аварийный выход», который программист может использовать, когда требуется вычисление, которое не может быть выражено с помощью любого из встроенных операторов LINQ. `Apply` принимает функцию  $f$  и передает ей итератор по всей коллекции входных данных, позволяя выполнять произвольные потоковые вычисления. Оператор `Fork` очень похож на `Apply`,

за исключением того, что он принимает один входной элемент и генерирует несколько выходных наборов данных.

Большинство программ могут быть написаны непосредственно с использованием DryadLINQ примитивов. Так например, модель MapReduce может быть компактно изложена следующим образом в листинге 1 (для ясности исключены точные сигнатуры типов):

### Листинг 1 – MapReduce с использованием DryadLINQ

```
public static MapReduce<T> MapReduce<T>(  
    source,  
    mapper, // func(T) → Ms  
    keySelector, // func(M) → K  
    reducer // func(K, Ms) → Rs  
) {  
  
    var mapped = source.SelectMany(mapper);  
    var groups = mapped.GroupBy(keySelector);  
    return groups.SelectMany(reducer);  
}
```

На рисунке 13 изображен план выполнения DryadLINQ задач.

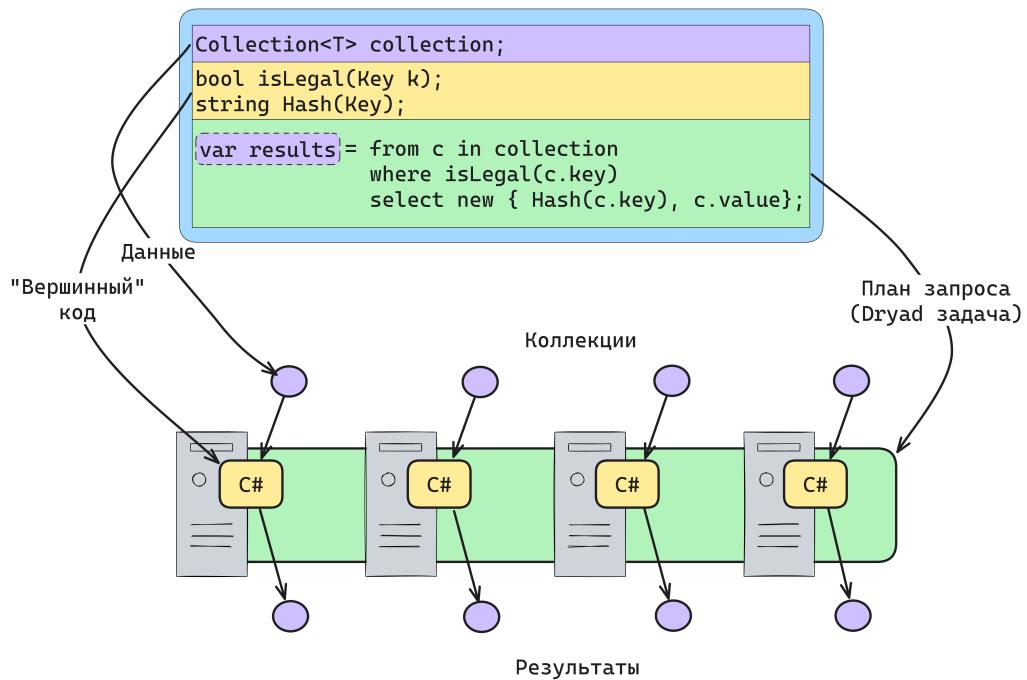


Рисунок 13 – План выполнения DryadLINQ задач.

#### 2.4.2 Реализация

Когда DryadLINQ получает управление, система начинает с преобразования необработанного выражения LINQ в граф плана выполнения (EPG), где каждый узел является оператором, а ребра представляют его входы и выходы. EPG тесно связан с традиционным планом запросов к базе данных, но используется более общая терминология плана выполнения, чтобы охватить вычисления, которые не просто сформулированы как «запросы». EPG представляет собой ориентированный ациклический граф — существование общих подвыражений и операторов, таких как Fork, означает, что EPG не всегда могут быть описаны деревьями. Затем DryadLINQ применяет оптимизацию для перезаписи терминов в EPG. EPG — это «скелет» графа потока данных Dryad, который будет выполнен, и каждый узел EPG реплицируется во время выполнения для создания «этапа» Dryad (набора вершин, выполняющих одно и то же вычисление в разных разделах набора данных).

Статическая оптимизация DryadLINQ представляет собой правила условной перезаписи графа, запускаемые предикатом свойств узла EPG. Большинство статических оптимизаций сосредоточены на минимизации дискового и сетевого ввода-вывода. Наиболее важными являются:

1. Конвейеризация: В одном процессе может быть выполнено несколько операторов. Конвейерные процессы сами по себе являются выражениями LINQ и могут выполняться существующей реализацией LINQ на одном компьютере.
2. Устранение избыточности: DryadLINQ удаляет ненужные шаги разделения хэша или диапазона.
3. Быстрая агрегация: Поскольку повторное разбиение наборов данных обходится дорого, последующие агрегации перемещаются перед операторами разбиения там, где это возможно.
4. Сокращение ввода-вывода: Там, где это возможно, DryadLINQ использует

TCP-канал Dryad и каналы FIFO в памяти вместо сохранения временных данных в файлах. Система по умолчанию сжимает данные перед выполнением разбиения на разделы, чтобы уменьшить сетевой трафик. Пользователи могут вручную переопределить настройки сжатия, чтобы сбалансировать загрузку процессора с нагрузкой на сеть, если оптимизатор примет неверное решение.

В качестве динамической оптимизации, DryadLINQ использует Dryad API для динамического изменения графика выполнения по мере того, как информация из запущенного задания становится доступной. Агрегирование дает возможность для большого сокращения времени ввода-вывода, поскольку оно может быть оптимизировано в виде дерева в соответствии с местоположением, агрегируя данные сначала на уровне компьютера, затем на уровне ряда компьютеров и, наконец, на уровне всего кластера.

EPG используется для получения плана выполнения Dryad после фазы статической оптимизации. Хотя EPG кодирует всю необходимую информацию, это не запускаемая программа. DryadLINQ использует динамическую генерацию кода для автоматического синтеза кода LINQ для запуска в вершинах Dryad. Сгенерированный код компилируется в сборку .NET, которая отправляется на компьютеры кластера во время выполнения. Для каждого этапа планирования выполнения сборка содержит два фрагмента кода:

1. Код для под выражения LINQ, выполняемого каждым узлом.
2. Код сериализации данных канала. Этот код намного эффективнее стандартных методов сериализации .NET, поскольку он может полагаться на контракт между считывателем и записывающим устройством канала для доступа к одному и тому же статически известному типу данных.

Под выражение в вершине строится из фрагментов общего EPG, передаваемого в DryadLINQ. EPG создается в контексте выполнения исходного клиентского компьютера и может зависеть от этого контекста двумя способами:

1. Выражение может ссылаться на переменные в локальном контексте. Эти

ссылки удаляются путем частичного вычисления подвыражения во время генерации кода. Для примитивных значений ссылки в выражениях заменяются фактическими значениями. Значения объектов сериализуются в файл ресурсов, который отправляется на компьютеры в кластере во время выполнения.

2. Выражение может ссылаться .NET. .NET Reflection используется для поиска транзитивного закрытия всех несистемных библиотек, на которые ссылается исполняемый файл, и они отправляются на компьютеры кластера во время выполнения.

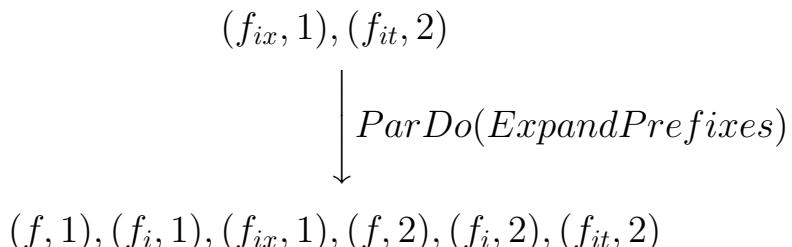
## 2.5 Google Dataflow

### 2.5.1 Программная модель

Термин «Dataflow» используется для описания модели обработки данных «Google Cloud Dataflow» [30], которая основана на технологии «FlumeJava» [31] и «MillWheel» [32]. Одной из реализаций модели «Dataflow» является Apache Beam.

Модель Dataflow содержит две основных операции, которые работают с парами ключ-значение, проходящими через систему [33]:

- ParDo для универсальной параллельной обработки. Каждый обрабатываемый входной элемент (который сам по себе может быть конечной коллекцией) предоставляется определяемой пользователем функции (называемой DoFn в Dataflow), которая может выдавать ноль или более выходных элементов на вход. Например, рассмотрим операцию, которая расширяет все префиксы ключа ввода, дублируя значение между ними:



- GroupByKey для группируемых пар ключ-значение по ключу.

$$(f, 1), (f_i, 1), (f_{ix}, 1), (f, 2), (f_i, 2), (f_{it}, 2)$$

$\downarrow$   
*GroupByKey*

$$(f, [1, 2]), (f_i, [1, 2]), (f_{ix}, [1]), (f_{it}, [2])$$

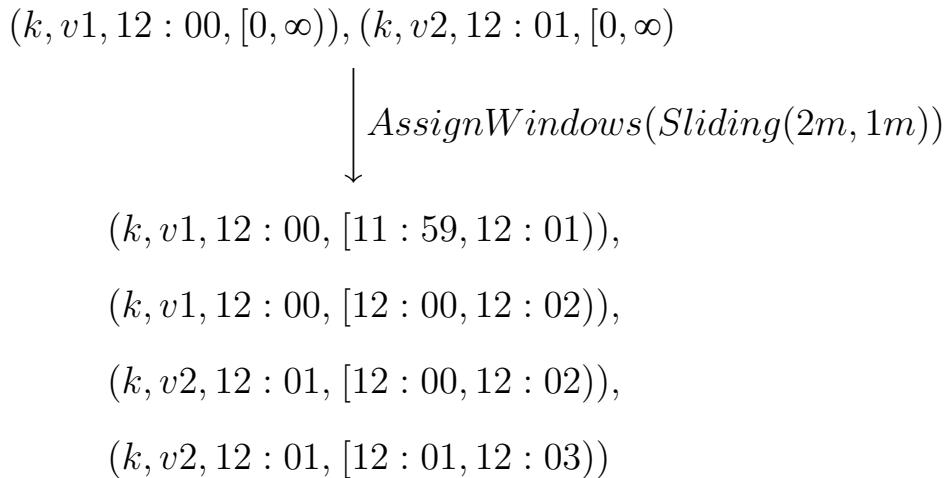
Операция ParDo работает поэлементно с каждым входным элементом и, таким образом, естественным образом преобразуется в работу с потоковыми данными. Операция GroupByKey, с другой стороны, собирает все данные для данного ключа перед отправкой их вниз по потоку. Если источник входных данных неограничен, у системы нет возможности узнать, когда он закончится. Распространенным решением этой проблемы является обработка данных окнами.

Системы, поддерживающие группировку данных, обычно переопределяют свою операцию GroupByKey, по сути, превращая ее в GroupByKeyAndWindow. Особенность Dataflow модели заключается в поддержке невыровненных окон, за которыми закреплены 2 основные идеи. Первая заключается в том, что проще рассматривать все стратегии управления окнами как невыровненные с точки зрения модели и позволять базовым реализациям применять оптимизации, относящиеся к выровненным случаям, где это применимо. Вторая заключается в том, что управление окнами можно разбить на две взаимосвязанные операции:

- Set<Window> AssignWindows(T datum), который присваивает элементу ноль или более окон.
- Set<Window> MergeWindows(Set<Window> windows), который объединяет окна во время группировки. Это позволяет создавать окна по мере поступления данных и группировать их вместе.

Следует обратить внимание, что для поддержки оконного управления временем события изначально, вместо передачи пар (ключ, значение) через систему, теперь передается (ключ, значение, время события, окно) 4-значный кортеж. Элементы представляются системе с временными метками времени события

(которые также могут быть изменены в любой точке конвейера) и первоначально назначаются глобальному окну по умолчанию, охватывающему все время события, предоставляя семантику, соответствующую значениям по умолчанию в стандартной пакетной модели.



### 2.5.2 Реализация

Реализация Google Cloud Dataflow во многом основана на других проектах Google – FlumeJava [31], в качестве модели для пакетной обработки данных, и MillWheel, в качестве модели для потоковой обработки данных.

Центральный класс библиотеки FlumeJava — PCollection<T>, зачастую большой неизменяемый набор элементов типа Т. Коллекция может иметь либо четко определенный порядок (называемый последовательностью), либо элементы могут быть неупорядоченными (называемыми коллекцией). Поскольку они менее ограничены, коллекции более эффективны для создания и обработки, чем последовательности. PCollection<T> может быть создана из находящейся в памяти Java PCollection<T>. PCollection<T> также может быть создана путем чтения файла в одном из нескольких возможных форматов. Например, текстовый файл может быть прочитан как PCollection<String>, а файл с двоичным представлением, может быть прочитан как PCollection<T>, учитывая спецификацию того, как декодировать каждую двоичную запись в объект Java типа Т. Могут быть прочитаны наборы данных, представленные несколькими фрагментами файла в виде единой логической коллекции.

Вторым основным классом FlumeJava является PTable<K, V>, который представляет собой неизменяемую мультикарту с ключами типа K и значениями типа V.

PTable<K, V> является подклассом PCollection<Pair<K, V>> и действительно представляет собой просто неупорядоченный набор пар. Некоторые операции FlumeJava применяются только к коллекциям пар, и в контексте Java было сделано решение определить подкласс, чтобы охватить эту абстракцию; на другом языке PTable<K, V> лучше было бы определить как синоним типа PCollection<Pair<K, V>>.

Чтобы обеспечить оптимизацию запросов FlumeJava, параллельные операции выполняются лениво с использованием отложенного вычисления. Каждый объект коллекции представлен внутри либо в отложенном (еще не вычисленном), либо в материализованном (вычисленном) состоянии. Отложенная коллекция содержит указатель на отложенную операцию, которая ее вычисляет. Отложенная операция, в свою очередь, содержит ссылки на коллекции, которые являются ее аргументами (которые сами по себе могут быть отложены или материализованы), и отложенные коллекции, которые являются ее результатами. Когда вызывается операция FlumeJava, она просто создает объект отложенной операции и возвращает новую отложенную коллекцию, которая указывает на нее. Таким образом, результатом выполнения серии операций FlumeJava является направленный ациклический граф отложенных коллекций и операций — граф плана выполнения

На высоком уровне MillWheel представляет собой граф определяемых пользователем преобразований входных данных, которые генерируют выходные данные. Каждое из этих преобразований может быть распараллелено на произвольном количестве машин, так что пользователю не нужно беспокоиться о балансировке нагрузки на детальном уровне. Абстрактно входные и выходные данные в MillWheel представлены тройками (ключ, значение, временная метка). В то время как ключ является полем метаданных, имеющим семантическое

значение в системе, значением может быть произвольная строка байтов, соответствующая всей записи. Контекст, в котором выполняется пользовательский код, ограничен определенным ключом, и каждое вычисление может определять ключ для каждого источника ввода в зависимости от его логических потребностей. Временным меткам в этих тройках пользователь MillWheel может присвоить произвольное значение (но обычно они близки к времени системных часов, когда произошло событие). Если бы пользователь агрегировал количество поисковых запросов в секунду, то он хотел бы присвоить значение временной метки, соответствующее времени, в которое был выполнен поиск.

В совокупности конвейер пользовательских вычислений формирует граф потока данных, поскольку выходные данные одного из вычислений становятся входными данными для другого и так далее. Отложенная операция, в свою очередь, содержит ссылки на коллекции, которые являются ее аргументами (которые сами по себе могут быть отложены или материализованы), и отложенные коллекции, которые являются ее результатами. Когда вызывается операция FlumeJava, она просто создает объект отложенной операции и возвращает новую отложенную коллекцию, которая указывает на нее. Таким образом, результатом выполнения серии операций FlumeJava является направленный ациклический граф отложенных коллекций и операций — граф плана выполнения

## 2.6 Apache Flink

### 2.6.1 Програмная модель

Apache Flink — это система с открытым исходным кодом для обработки потоковых и пакетных данных.

Apache Flink следует парадигме, которая включает обработку потоков данных в качестве объединяющей модели для анализа в реальном времени, непрерывных потоков и пакетной обработки как в модели программирования, так и в механизме выполнения. В сочетании с устойчивыми очередями сообщений, которые позволяют квази-произвольное воспроизведение потоков данных (как в Apache Kafka или Amazon Kinesis), программы потоковой обработки

не делают различий между обработкой последних событий в режиме реального времени, непрерывной периодической агрегацией данных в больших окнах или обработкой терабайт информации. Вместо этого эти различные типы вычислений просто начинают свою обработку в разных точках долговременного потока и поддерживают разные формы состояния во время вычислений. Благодаря очень гибкому оконному механизму программы Flink могут вычислять как ранние, так и приблизительные, а также отложенные и точные результаты за одну и ту же операцию, устранив необходимость комбинировать различные системы для двух вариантов использования. Flink поддерживает различные понятия времени (время события, время приема, время обработки), чтобы предоставить программистам высокую гибкость в определении того, как события должны быть соотнесены.

На рисунке 14 изображена програмная модель Apache Flink.

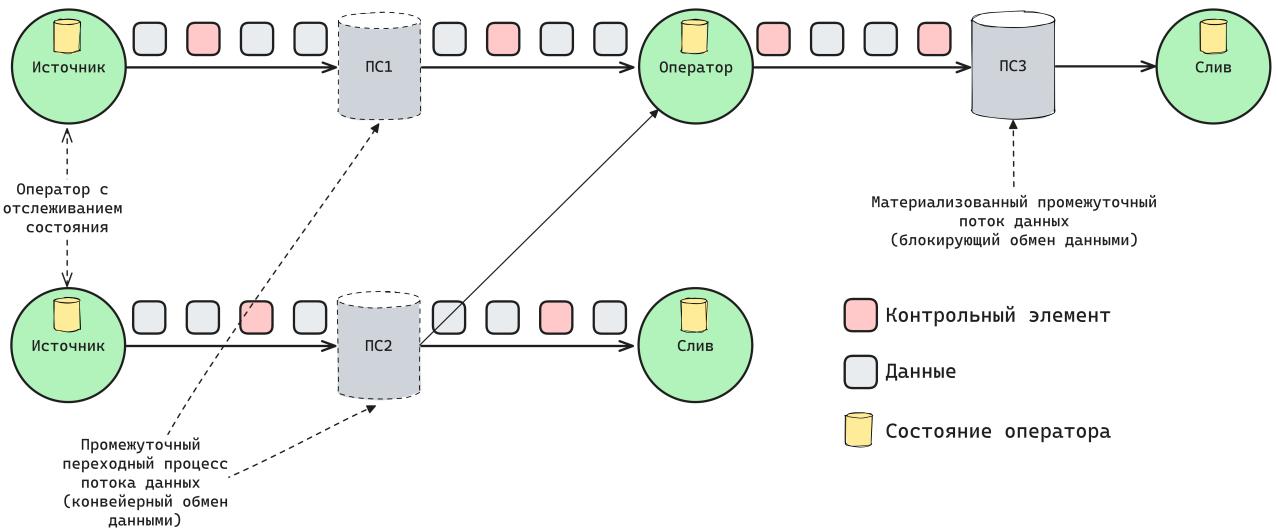


Рисунок 14 – Программная модель Flink

Промежуточные потоки данных Flink являются основной абстракцией для обмена данными между операторами. Промежуточный поток данных представляет собой логический дескриптор данных, которые создаются оператором и могут использоваться одним или несколькими операторами. Промежуточные потоки логичны в том смысле, что данные, на которые они указывают, могут

материализоваться на диске, а могут и не материализоваться. Конкретное поведение потока данных параметризуется более высокими уровнями Flink (например, программным оптимизатором, используемым API набора данных).

1. Конвейерный и блокирующий обмен данными.

Это позволяет Flink достигать высокой пропускной способности, устанавливая размер буферов на высокое значение (например, несколько килобайт), а также низкой задержки, устанавливая время ожидания буфера на низкое значение (например, несколько миллисекунд).

2. Управляющие события.

Помимо обмена данными, потоки в Flink передают разные типы управляющих событий. Это специальные события, вводимые в поток данных операторами, и доставляются по порядку вместе со всеми другими записями данных и событиями в разделе потока. Принимающие операторы реагируют на эти события, выполняя определенные действия по их прибытии. Flink использует множество специальных типов контрольных событий, включая:

- барьеры контрольных точек, которые координируют контрольные точки, разделяя поток на до-контрольные точки и после-контрольные точки.
- «водяные знаки», сигнализирующие о ходе событий во времени внутри раздела потока;
- итерационные барьеры, сигнализирующие о том, что раздел потока достиг конца суперэтапа.

### 2.6.2 Реализация

Flink предоставляет низкоуровневую операцию потоковой обработки – `ProcessFunction` – которая обеспечивает доступ к основным компоновочным блокам любого потокового приложения [34]:

1. События (отдельные записи в потоке);
2. Состояние (отказоустойчивое, согласованное);

### 3. Таймеры (время события и время обработки).

На рисунке 15 изображена итерационная модель обработки потоков в Apache Flink.

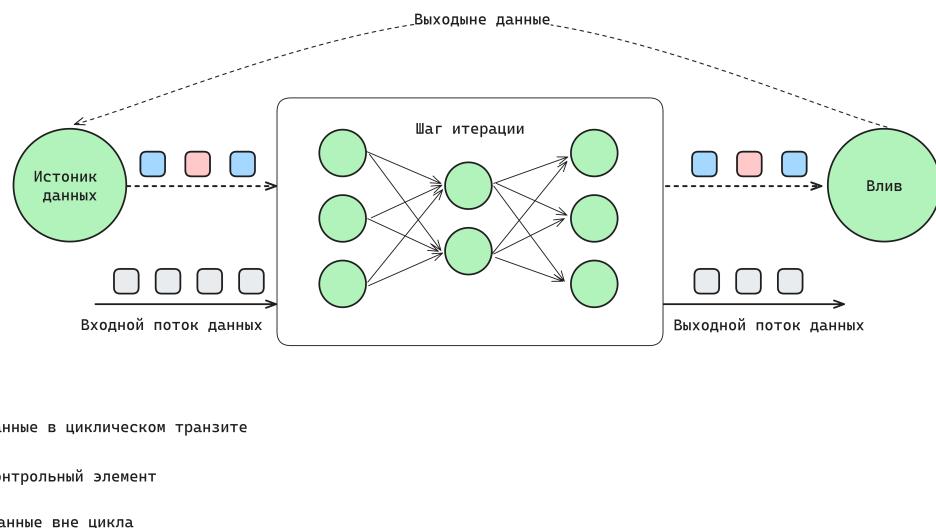


Рисунок 15 – Итерационная модель Flink

Реализация операций над двумя входными потоками обеспечивается с помощью низкоуровневой Flink операции `join`, которая привязана к двум разным входам (при необходимости объединить более двух потоков, можно каскадировать несколько низкоуровневых объединений) и предоставляет индивидуальные методы обработки записей из каждого входного сигнала. Реализация низкоуровневого соединения обычно выполняется по следующей схеме:

1. Создание и поддержка объектов состояния, отражающих текущее состояние выполнения.
2. Обновление состояний при получении элементов с одного (или обоих) входных потока.
3. Использование текущего состояния для преобразования данных и получения результата при получении элементов с одного или обоих входных потока.

Кластер Flink включает в себя три типа процессов: клиент, менеджер работ (`JobManager`) и, по крайней мере, один менеджер задач (`TaskManager`). Кли-

ент берет программный код, преобразует его в граф потока данных и отправляет его в JobManager. На этом этапе преобразования также проверяются типы данных (схемы) данных, которыми обмениваются операторы, и создаются сериализаторы и другой код, специфичный для типа/схемы.

Dataflow программы дополнительно проходят этап оптимизации запросов на основе затрат, аналогичный физической оптимизации, выполняемой оптимизаторами реляционных запросов. JobManager координирует распределенное выполнение потока данных. Он отслеживает состояние и прогресс каждого оператора и потока, планирует новых операторов и координирует контрольные точки и восстановление. При настройке высокой доступности JobManager сохраняет минимальный набор метаданных на каждой контрольной точке в отказоустойчивом хранилище, так что резервный JobManager может восстановить контрольную точку и восстановить выполнение потока данных оттуда. Фактическая обработка данных происходит в TaskManager. TaskManager выполняет один или несколько операторов, которые создают потоки, и сообщает об их статусе TaskManager. TaskManager поддерживают буферные пулы для буферизации или материализации потоков и сетевые подключения для обмена потоками данных между операторами. Он отслеживает состояние и прогресс каждого оператора и потока, планирует новых операторов и координирует контрольные точки и восстановление. При настройке высокой доступности JobManager сохраняет минимальный набор метаданных на каждой контрольной точке в отказоустойчивом хранилище, так что резервный JobManager может восстановить контрольную точку и восстановить выполнение потока данных оттуда. Фактическая обработка данных происходит в TaskManager. TaskManager выполняет один или несколько операторов, которые создают потоки, и сообщает об их статусе TaskManager. TaskManager поддерживают буферные пулы для буферизации или материализации потоков и сетевые подключения для обмена потоками данных между операторами.

## 2.7 Apache Samza

### 2.7.1 Програмная модель

Apache Samza — это распределенная система для отказоустойчивой потоковой обработки с отслеживанием состояния.

Samza использует секционированное локальное состояние наряду с механизмом фонового журнала изменений с низкими накладными расходами, что позволяет масштабировать его до сотен терабайт для каждого приложения. Восстановление после сбоев ускоряется за счет перепланирования на основе привязки к хосту. В дополнение к обработке бесконечных потоков событий, Samza поддерживает обработку конечного набора данных в виде потока либо из источника потоковой передачи, таких как Kafka, моментального снимка базы данных, либо из файловой системы (например HDFS), без необходимости изменять код приложения. Этим он отличается от архитектур на основе лямбда, которые требуют обслуживания отдельных баз кода для пакетной и потоковой обработки [35].

Задача в Samza состоит из набора экземпляров виртуальной машины Java (JVM), каждый из которых обрабатывает подмножество входных данных. Код, выполняемый в каждой JVM, включает в себя платформу Samza и пользовательский код, реализующий требуемую функциональность для конкретного приложения. Основным интерфейсом для пользовательского кода является Java-интерфейс `StreamTask`, который определяет метод `process()`. Как только Samza-задача развернута и инициализирована, фреймворк вызывает метод `process()` один раз для каждого сообщения в любом из входных потоков. Выполнение этого метода может иметь различные эффекты, включая запрос или обновление локального состояния и отправку сообщений в выходные потоки. Эта модель вычислений очень похожа на функцию Map модели MapReduce с той разницей, что входные данные Samza-задача обычно бесконечны (неограничены). Аналогично MapReduce, каждая задача Samza представляет собой однопоточ-

ный процесс, который выполняет итерацию по последовательности входных записей. Входные данные для задания *Samza* разбиваются на непересекающиеся подмножества, и каждая входная секция назначается ровно одной задаче обработки. Одной и той же задаче обработки может быть назначено более одного раздела, и в этом случае обработка этих разделов чередуется в потоке задачи. Однако количество разделов во входных данных определяет максимальную степень параллелизма задач [36].

Интерфейс журнала предполагает, что каждый раздел входных данных представляет собой полностью упорядоченную последовательность записей и что каждая запись связана с монотонно увеличивающимся порядковым номером или идентификатором (известным как смещение). Поскольку записи в каждом разделечитываются последовательно, задача может отслеживать ход выполнения, периодически записывая смещение последней прочитанной записи в долговременное хранилище. Если задача потоковой обработки перезапускается, она возобновляет использование входных данных из последнего записанного смещения. Чаще всего *Samza* используется в сочетании с *Apache Kafka*.

*Kafka* предоставляет секционированный отказоустойчивый журнал, который позволяет издателям добавлять сообщения в раздел журнала, а потребителям (подписчикам) для последовательного чтения сообщений в разделе журнала. *Kafka* также позволяет заданиям потоковой обработки повторно обрабатывать ранее просмотренные записи, сбрасывая смещение потребителя на более раннюю позицию, что полезно при восстановлении после сбоев.

В то время как каждый раздел входного потока назначен одной конкретной задаче задания *Samza*, выходные разделы не привязаны к задачам. То есть, когда система отправляет выходные сообщения, он может назначить их любому разделу выходного потока. Этот факт может быть использован для группировки связанных элементов данных в тот же раздел, как показано на рисунке 16.

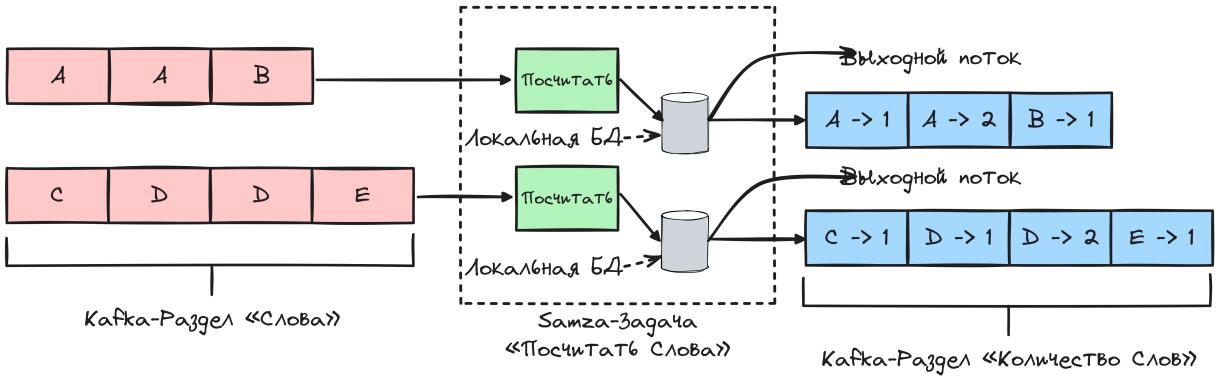


Рисунок 16 – Пример исполнения задачи в системе Samza

Когда потоковые задачи объединяются в многоступенчатые конвейеры обработки, выходные данные одной задачи становятся входными данными для другой задачи. В отличие от многих других фреймворков потоковой обработки, Samza не реализует свой собственный транспортный уровень сообщений для доставки сообщений между потоковыми операторами. Вместо этого для этой цели используется Kafka; поскольку Kafka записывает все сообщения на диск, он обеспечивает большой буфер между этапами конвейера обработки, ограниченный только доступным дисковым пространством на брокерах Kafka.

### 2.7.2 Реализация

Samza-задача — это неделимый этап вычислений: в задачу подается один или несколько входных потоков; на входных данных выполняются различные обработки — от простых операций (например, фильтрации, объединения и агрегации) до сложных алгоритмов машинного обучения; и генерируется один или несколько новых выходных потоков.

Samza представляет задачи в виде ориентированного графа операторов (вершин), соединенных потоками данных (ребрами).

Оператор — это преобразование одного или многих потоков в другой поток (потоки). В зависимости от количества входных и выходных потоков Samza поддерживает три типа операторов.

1. Операторы один-к-одному:

- *map* — применение определенной функции к каждому сообщению;

- *filter* – фильтрация сообщений на основе функции;
- *window* – разбивает поток на окна и агрегирует;
- *partition* – перераспределение потока по другому ключу.

## 2. Операторы множество-к-одному:

- *join* – объединение  $\geq 2$  потоков в один поток на основе заданной функции;
- *merge* – слияние  $\geq 2$  двух потоков в один поток.

3. Определяемые пользователем – разделение или репликация потока на  $\geq 2$  потока. Это достигается за счет того, что несколько операторов могут использовать один и тот же поток.

Внутренне, как показано на рисунке 17, задание делится на множество параллельных, независимых и идентичных задач, а входной поток делится на разделы (например, P1, ..., P<sub>p</sub>).

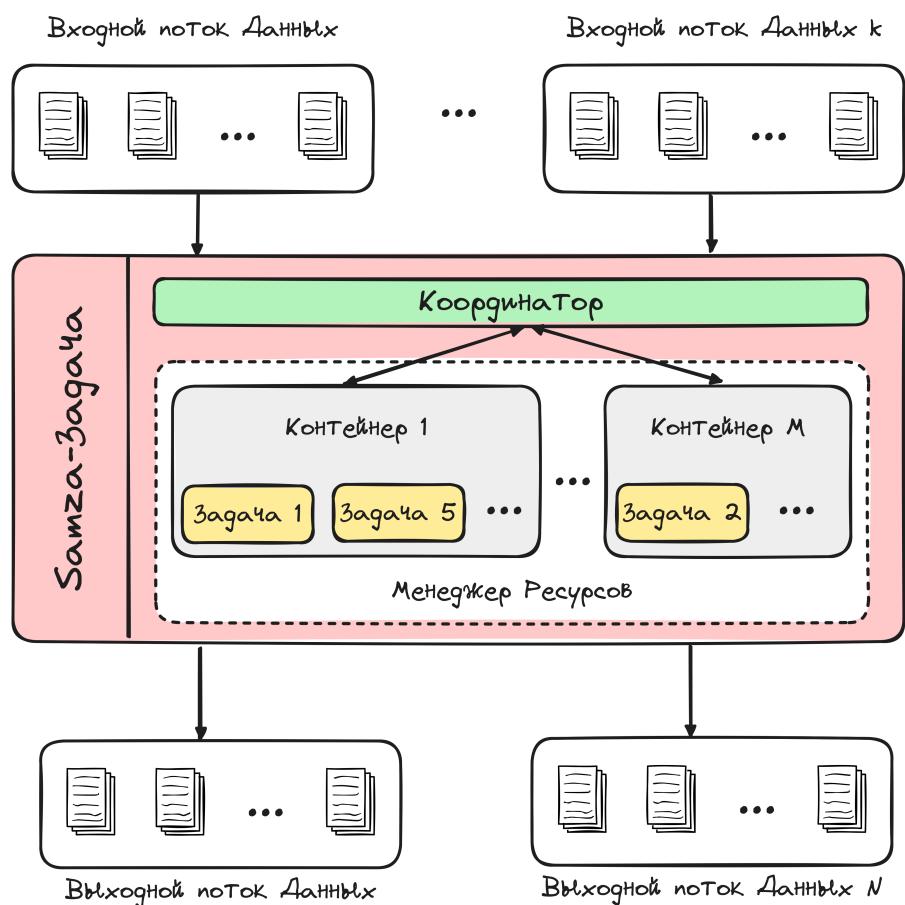


Рисунок 17 – Внутренняя архитектура Samza-задачи

Каждая задача выполняет идентичную логику, но на своем собственном входном разделе (подход параллелизма данных). Каждая задача запускает весь граф операторов. Для каждого входящего сообщения задача пропускает сообщение через граф (выполняя операторы над сообщением) до тех пор, пока не будет достигнут оператор без выходных данных или конечный поток выходных данных. Большинство ребер промежуточного потока остаются локальными для задачи, т. е. они не пересекают границу задачи. Это сохраняет большинство

коммуникаций локальными и минимизирует сетевой ввод-вывод. Единственным исключением является оператор разделения, где сообщения перераспределяются между всеми задачами на основе логики разделения. Для нелокальных потоков и входных и выходных потоков задач Samza использует отказоустойчивый (без потери сообщений) и воспроизводимый (с большими возможностями буферизации) механизм связи.

## **Вывод**

В данном разделе были рассмотрены основные подходы к обработке данных. Были рассмотрены основные принципы работы пакетных и потоковых систем, а также рассмотрены основные реализации таких систем. Были рассмотрены системы семейства Hadoop, Spark, Pregel, DryadLINQ, Google Dataflow, Apache Flink и Apache Samza. Все рассмотренные системы обладают своими преимуществами и недостатками, и могут быть применены в зависимости от поставленной задачи.

## ЗАКЛЮЧЕНИЕ

Сравнение рассмотренных методов будет проводиться по следующим критериям:

- Пакетная обработка — характеристика способа обработки данных в виде пакетов (при присутствии оной);
- Потоковая обработка — характеристика способа обработки данных в виде потоков (при присутствии оной);
- Механизм коммуникации — характеристика способа передачи данных между узлами в системе;
- Модель вычисления — оценка программной модели, отвечающей за обработку данных;
- Планирование задач и балансировка нагрузки — характеристика алгоритмов планирования и балансировки нагрузки;
- Масштабируемость — качественная оценка возможностей системы масштабировать вычисления на весь кластер;
- Отказоустойчивость — качественная оценка возможностей системы восстанавливаться после сбоев или неисправностей;
- Сценарий использования — оценка вариантов использования, для которых лучше всего подходит данная система;

На таблице 1 приведена классификация моделей по способам пакетной и потоковой обработке.

Таблица 1 – Классификация программных моделей по способам пакетной и потоковой обработке

<b>Модель</b>	<b>Пакетная обработка</b>	<b>Потоковая обработка</b>
Hadoop MapReduce	Пакеты фиксированного размера.	Серия малых пакетов.
Apache Spark	Обработка посредством RDD	Серия малых пакетов.
Pregel	Итеративный алгоритм для обработки графов.	Отсутствует.
DryadLinq	Формирует пакеты исходя из LINQ выражений.	Потоковая обработка ранее сформированных пакетов.
Google Dataflow	Оконная обработка пакетов.	Обработка в реальном времени.
Apache Flink	Оконная обработка пакетов.	Обработка в реальном времени.
Apache Samza	Оконная обработка пакетов.	Обработка в реальном времени.

На таблице 2 приведена классификация моделей по механизмам коммуникации и планированию задач.

Таблица 2 – Классификация программных моделей по механизмам коммуникации и планированию задач

<b>Модель</b>	<b>Механизм коммуникации</b>	<b>Планирование задач и балансировка нагрузки</b>
Hadoop MapReduce	Мастер-рабочий архитектура	Мастер отвечает за балансировку
Apache Spark	Мастер-рабочий архитектура	Направленный ациклический граф (DAG) из задач
Pregel	Передача сообщений между вершинами графа	Разбиение графа мастер узлом
DryadLinq	Мастер-рабочий архитектура	Направленный ациклический граф (DAG) из задач
Google Dataflow	Коммуникация через конвейеры	Оптимизация плана выполнения на основе графа Dataflow
Apache Flink	Поточная передача данных	Распределение буфера между потребителями
Apache Samza	Поточная передача данных при помощи Apache Kafka	Контейнеризация на основе системы YARN

На таблице 3 приведена классификация моделей по модели вычисления.

Таблица 3 – Классификация программных моделей по модели вычисления

<b>Модель</b>	<b>Модель вычисления</b>
Hadoop MapReduce	Двухэтапная вычислительная модель
Apache Spark	Манипуляция устойчивыми распределенными наборами данных
Pregel	Вершинно-центричная модель
DryadLinq	Декларативная модель
Google Dataflow	Унифицированная модель для пакетной и потоковой обработки
Apache Flink	Обработка потоков
Apache Samza	Событийная обработка потоков

На таблице 4 приведена классификация моделей по масштабируемости и устойчивости к сбоям.

Таблица 4 – Классификация программных моделей по масштабируемости и устойчивости к сбоям

<b>Модель</b>	<b>Масштабируемость</b>	<b>Устойчивость к сбоям</b>
Hadoop MapReduce	Горизонтально масштабируема	Повторное выполнение . проваленных задач
Apache Spark	Горизонтально масштабируема . с использованием оперативной памяти	Репликация данных в рамках RDD
Pregel	Масштабируется в рамках обработки графов	Точки контроля и восстановление состояния
DryadLinq	Горизонтально масштабируема	Точки контроля и восстановление состояния
Google Dataflow	Динамическая масштабируемость под размер данных	Точки контроля и повторное выполнение
Apache Flink	Масштабируется путем добавления менеджеров задач	Точки контроля и распределенние снимков состояния
Apache Samza	Масштабируется путем добавления менеджеров задач	Репликация состояния между узлами

На таблице 5 приведена классификация моделей по сценариям использования.

Таблица 5 – Классификация моделей программного обеспечения по сценариям использования

Модель	Сценарий использования
Hadoop MapReduce	Пакетная обработка больших наборов данных
Apache Spark	Пакетная обработка данных в реальном времени и итеративные алгоритмы
Pregel	Обработка графов большого масштаба
DryadLinq	Распределенная обработка данных общего назначения, научные вычисления
Google Dataflow	Непрерывная обработка данных в реальном времени, приложения, управляемые событиями
Apache Flink	Обработка потоков данных, приложения, управляемые событиями, аналитика в реальном времени
Apache Samza	Обработка потоков данных, приложения, управляемые сообщениями

Результаты, полученные в результате анализа упомянутых систем, приводят к следующим выводам:

- Хотя основополагающая концепция горизонтальной масштабируемости

остается неизменной во всех изученных моделях, подходы к ее достижению демонстрируют адаптацию к современным вызовам.

- Отказоустойчивость является критическим компонентом при проектировании этих систем, отражая присущей распределенным средам непредсказуемости. Помимо традиционных механизмов восстановления после сбоев, автоматическое определение контрольных точек и репликация данных демонстрируют приверженность обеспечению целостности данных и последовательной обработки.
- Разнообразие моделей вычисления отражает понимание разнообразных требований, предъявляемых различными сценариями обработки данных. Apache Flink и Google Dataflow выделяются тем, что предлагают унифицированные модели программирования как для пакетной, так и для потоковой обработки, облегчая когнитивную нагрузку на разработчиков.
- Коммуникационные модели раскрывают архитектурные тонкости, предназначенные для оптимизации потока данных в этих системах. От модели «мастер-рабочий» в MapReduce до подхода, основанного на потоке данных в Google Dataflow, коммуникационная парадигма каждой системы соответствует предполагаемым вариантам использования, демонстрируя нюансы решений, принимаемых для повышения эффективности и быстродействия.
- Возможности систем для решения специализированных задач подчеркивают их преимущества в своей нише. Сосредоточенность Pregel на обработке графов и специализация Apache Flink в обработке потоков с низкой задержкой подчеркивают индивидуальные решения, предлагаемые этими системами. Адаптивность Google Dataflow для обработки ограниченных и неограниченных наборов данных подчеркивает его универсальность, способствуя его актуальности в широком спектре сценариев обработки данных.

В заключение следует отметить, что эти распределенные системы обработки данных служат не только инструментами для управления крупномасштабными данными, но и артефактами, отражающими нюансы решений, принятых их разработчиками. Они демонстрируют динамичный отклик на меняющийся ландшафт распределенных вычислений, где адаптивность, отказоустойчивость и продуманные парадигмы программирования объединяются, чтобы дать разработчикам возможность решать разнообразные задачи обработки данных.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. POWER4 system microarchitecture / J. Tendler [и др.] // IBM Journal of Research and Development. — 2002. — Янв. — Т. 46. — С. 5—25. — DOI: 10.1147/rd.461.0005.
2. Uddin I. Advances in computer architecture. — 2013. — arXiv: 1309 . 5459 [cs.AR].
3. Dual Core Era Begins, PC Makers Start Selling Intel-Based PCs [Электронный ресурс]. — (дата обращения: 22.10.2023). Режим доступа: <https://www.intel.com/pressroom/archive/releases/2005/20050418comp.htm>.
4. The Landscape of Parallel Computing Research: A View from Berkeley : тех. отч. / K. Asanović [и др.] ; EECS Dept., University of California, Berkeley. — 2006. — UCB/EECS-2006—183. — URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>.
5. McKenney P. E. Is Parallel Programming Hard, And, If So, What Can You Do About It? (Release v2023.06.11a). — 2023. — arXiv: 1701 . 00854 [cs.DC].
6. Lynch N. Distributed Algorithms. — Morgan Kaufmann, 1996. — (The Morgan Kaufmann Data Manag). — ISBN 9781558603486. — URL: <https://books.google.at/books?id=7C7oIV48RQQC>.
7. Distributed Systems: Concepts and Design, 4/e. — Pearson Education, 2009. — ISBN 9788131718407. — URL: <https://books.google.ru/books?id=0-FyIBTjVOYC>.
8. A Compendium on Distributed Systems / A. Khole [и др.]. — 2023. — arXiv: 2302.03990 [cs.DC].
9. TOP500 [Электронный ресурс]. — (дата обращения: 15.11.2023). Режим доступа: <https://www.top500.org/lists/top500/>.

10. TOP500 | The Linpack Benchmark [Электронный ресурс]. — (дата обращения: 15.11.2023). Режим доступа: <https://www.top500.org/project/linpack/>.
11. Kleppmann M. Designing Data-intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. — O'Reilly Media, 2017. — ISBN 9781449373320. — URL: <https://books.google.at/books?id=BM7woQEACAAJ>.
12. Pinedo M. Scheduling: Theory, Algorithms, and Systems. — Springer New York, 2012. — (SpringerLink : Bücher). — ISBN 9781461423614. — URL: <https://books.google.com.ec/books?id=QRiDnuXSvVwC>.
13. Baptiste P. Batching Identical Jobs // Math Methods of Operations Research. — 2000. — Т. 52, № 3. — С. 355—367. — ISSN 1432-5217. — DOI: 10.1007/s001860000088. — URL: <https://doi.org/10.1007/s001860000088>.
14. Batch Processing: Definition and Event Log Identification / N. Martin [и др.] // International Symposium on Data-Driven Process Discovery and Analysis. — 2015. — URL: <https://api.semanticscholar.org/CorpusID:18387691>.
15. Doblander C., Rabl T., Jacobsen H.-A. Processing Big Events with Showers and Streams // Specifying Big Data Benchmarks / под ред. T. Rabl [и др.]. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2014. — С. 60—71. — ISBN 978-3-642-53974-9.
16. Margara A., Rabl T. Definition of Data Streams // Encyclopedia of Big Data Technologies. — 2018. — Июль. — С. 1—4. — DOI: 10.1007/978-3-319-63962-8\_188-1.
17. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, out-of-Order Data Processing / R. Lax [и др.] // Proc. VLDB Endow. — 2015. — Авг. — Т. 8, № 12. — С. 1792—

1803. — ISSN 2150-8097. — DOI: 10.14778/2824032.2824076. — URL: <https://doi.org/10.14778/2824032.2824076>.
18. Secret: A Model for Analysis of the Execution Semantics of Stream Processing Systems / I. Botan [и др.] // Proc. VLDB Endow. — 2010. — Сент. — Т. 3, № 1/2. — С. 232—243. — ISSN 2150-8097. — DOI: 10.14778/1920841.1920874. — URL: <https://doi.org/10.14778/1920841.1920874>.
19. Chowdhary K. Distributed Algorithms (Lecture Notes) (Tech) Leader election Algorithms. — 2016. — (дата обращения: 9.10.2023). Режим доступа:<https://www.krchowdhary.com/dist-algo/ledrelct.pdf>.
20. Dean J. MapReduce: Simplified Data Processing on Large Clusters // OSDI'04: Sixth Symposium on Operating System Design and Implementation. — San Francisco, CA, 2004. — С. 137—150.
21. Vohra D. Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools. — Apress, 2016. — (For professionals by professionals). — ISBN 9781484221990. — URL: <https://books.google.ru/books?id=Hz4sDQAAQBAJ>.
22. An Empirical Exploration of the Yarn in Big Data / D. Y. Perwej [и др.] // International Journal of Applied Information Systems. — 2017. — Т. 12. — С. 19—29. — URL: <https://api.semanticscholar.org/CorpusID:54841597>.
23. Apache Spark [Электронный ресурс]. — (дата обращения: 17.11.2023). Режим доступа: <https://spark.apache.org>.
24. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing / M. Zaharia [и др.] // 9 USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). — San Jose, CA : USENIX Association, 2012. — С. 15—28. — ISBN 978-931971-92-8. — URL: <https://api.semanticscholar.org/CorpusID:54841597>.

- //www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf.
25. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center / B. Hindman [и др.] // 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11). — USENIX Association, 03.2011. — URL: <https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center>.
  26. Quick L., Wilkinson P., Hardcastle D. Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis // 2012 IEEE / ACM Conference on Advances in Social Networks Analysis and Mining. — 2012. — C. 457—463. — DOI: 10.1109/ASONAM.2012.254.
  27. Valiant L. G. A bridging model for parallel computation // Commun. ACM. — 1990. — T. 33. — C. 103—111. — URL: <https://api.semanticscholar.org/CorpusID:15655597>.
  28. Microsoft Research | DryadLINQ [Электронный ресурс]. — (дата обращения: 19.11.2023). Режим доступа: <https://www.microsoft.com/en-us/research/project/dryadlinq/>.
  29. DryadLINQ: A System for General-Purpose Distributed Computing Using a High-Level Language / Y. Ylleyu [и др.] // 8 USENIX Symposium on Operating Systems Design and Implementation (OSDI 08). — San Diego, CA : USENIX Association, 12.2008. — URL: <https://www.usenix.org/conference/osdi-08/dryadlinq-system-general-purpose-distributed-data-parallel-computing-using-high>.
  30. Google. Google Cloud Dataflow. [Электронный ресурс]. — (дата обращения: 27.10.2023). Режим доступа: <https://cloud.google.com/dataflow/>,
  31. FlumeJava: Easy, Efficient Data-Parallel Pipelines / C. Chambers [и др.] // ACM SIGPLAN Conference on Programming Language Design (PLDI). —

- 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010. — С. 363—375. — URL: <http://dl.acm.org/citation.cfm?id=1806638>.
32. MillWheel: Fault-Tolerant Stream Processing at Internet Scale / T. Akidau [и др.] // Very Large Data Bases. — 2013. — С. 734—746.
  33. 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.
  34. Apache Flink Implementation. [Электронный ресурс]. — (дата обращения: 11.12.2023). Режим доступа: <https://www.oreilly.com/library/view/serving-machine-learning/9781492024095/ch04.html>,
  35. Samza: stateful scalable stream processing at LinkedIn / S. A. Noghabi [и др.] // Proc. VLDB Endow. — 2017. — Авг. — Т. 10, № 12. — С. 1634—1645. — ISSN 2150-8097. — DOI: 10.14778/3137765.3137770. — URL: <https://doi.org/10.14778/3137765.3137770>.
  36. Kleppmann M. Apache Samza // Encyclopedia of Big Data Technologies / под ред. S. Sakr, A. Zomaya. — Cham : Springer International Publishing, 2018. — С. 1—8. — ISBN 978-3-319-63962-8. — DOI: 10.1007/978-3-319-63962-8\_197-2. — URL: [https://doi.org/10.1007/978-3-319-63962-8\\_197-2](https://doi.org/10.1007/978-3-319-63962-8_197-2).

## **ПРИЛОЖЕНИЕ А**

### **Презентация к курсовой работе**

Презентация содержит 15 слайдов.