

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Ю.А. Васильев, С.В. Герасимов, М.А. Казачук

СОВРЕМЕННЫЕ МЕТОДЫ РАСПРЕДЕЛЕННОГО ХРАНЕНИЯ И ОБРАБОТКИ ДАННЫХ

Учебно-методическое пособие



МОСКВА – 2025

УДК 004.75(075.8)
ББК 16.262я73
В19



<https://elibrary.ru/ulzffq>

Рецензенты:

М.И. Петровский – кандидат физико-математических наук, доцент факультета ВМК МГУ;
А.В. Чернов – кандидат физико-математических наук, доцент факультета ВМК МГУ

Васильев, Юлий Алексеевич.

В19 **Современные методы распределенного хранения и обработки данных** : учебно-методическое пособие / Ю.А. Васильев, С.В. Герасимов, М.А. Казачук. – Москва : МАКС Пресс, 2025. – 336 с.

ISBN 978-5-317-07462-3

<https://doi.org/10.29003/m4759.978-5-317-07462-3>

Учебно-методическое пособие содержит расширенный конспект лекций по курсу «Современные методы распределенного хранения и обработки данных», читаемому на кафедре Интеллектуальных информационных технологий факультета ВМК МГУ.

Ключевые слова: распределенные вычисления, потоковая обработка данных, оптимизация вычислений, планирование ресурсов.

УДК 004.75(075.8)

ББК 16.262я73

Авторы выражают благодарность за неоценимую помощь в подготовке методического пособия студентам и выпускникам кафедры Интеллектуальных информационных технологий факультета ВМК МГУ: Коновальчуку Дмитрию, Купцову Евгению, Лешонкову Артемию, Лисовому Владимиру, Малышеву Кириллу, Степочкину Даниле, Трофимову Денису, Фатееву Олегу, Харунову Ильясу, Чернышеву Вадиму, а также Васильевой Наталии за помощь в оформлении и редактировании иллюстраций.

*Публикация учебного пособия поддержана Некоммерческим фондом
развития науки и образования «Интеллект»*

ISBN 978-5-317-07462-3

© Ю.А. Васильев, С.В. Герасимов, М.А. Казачук, 2025
© Оформление. ООО «МАКС Пресс», 2025

Оглавление

Глава 1: Введение в большие данные

1.1	Большие данные	11
1.2	Какие данные бывают?	13
1.3	“Правило четырех V”	14
1.4	Большие данные в широком смысле	15
1.5	Ввод-вывод и обработка запросов	16
1.6	Статистический анализ больших данных	23
1.7	Машинное обучение на больших данных	28
1.8	Контрольные вопросы	39

Глава 2: Практические сценарии использования больших данных

2.1	Сценарий №1. Реклама	40
2.2	Сценарий №2. Кредитный скоринг и другие банковские задачи	44
2.3	Сценарий №3. Астрофизика	57
2.4	Контрольные вопросы	62

Глава 3: Google File System

3.1	Проблема чтения больших данных	63
3.2	Масштабирование	67
3.3	Технологии организации параллельной обработки данных	68
3.4	The Google File System	69
3.5	GFS: архитектура	70
3.6	Концептуальная архитектура	71
3.7	Контрольные вопросы	79

Глава 4: Модель вычислений MapReduce

4.1	Принцип работы MapReduce	80
4.2	Пример распределенных вычислений	81
4.3	Применение MapReduce в обработке большого текста	83
4.4	Роль стадий в модели вычислений	85
4.5	Технические детали работы фреймворка	87
4.6	Контрольные вопросы	89

Глава 5: Apache Hadoop

5.1	Появление и развитие Apache Hadoop	90
5.2	Установка Hadoop	92
5.3	Принципы работы HDFS	93
5.4	Контрольные вопросы	104

Глава 6: Hadoop MapReduce: Java API

6.1	Java как основной интерфейс для Hadoop	105
6.2	Тасовка (Shuffle)	112
6.3	Слияние (Reduce)	113
6.4	Терминология MapReduce в Hadoop	114
6.5	Передача параметров	115
6.6	Формат входных данных	120
6.7	Разбиение на порции (split)	121
6.8	Реализация TextInputFormat	124
6.9	Выходные данные заданий	125
6.10	Разбиение на части (Partitioner)	126
6.11	Контрольные вопросы	127

Глава 7: Hadoop MapReduce: детали

7.1	Файл последовательности (SequenceFile)	130
7.2	Мотивация сжатия данных	131
7.3	Обработка сжатых входных данных	132
7.4	Обработка ошибок и сбоев	133
7.5	Время выполнения задачи	138
7.6	Счетчики (Counters)	139
7.7	Сортировка данных	140
7.8	Примеры больших данных	145
7.9	Контрольные вопросы	149

Глава 8: Планирование вычислений и отладка приложений

8.1	Планировщик в Hadoop 1.0	151
8.2	Apache YARN (Hadoop 2.0)	153
8.3	Планирование исполнения программ в многопользовательском кластере	160
8.4	Основные типы планировщиков	171

8.5	Mesos	173
8.6	Отладка MapReduce-приложений	180
8.7	Контрольные вопросы	182
Глава 9: Теория сложности алгоритмов MapReduce		
9.1	Основные понятия	184
9.2	Примеры алгоритмов	185
9.3	Модель коммуникационной стоимости	192
9.4	Теория сложности MapReduce	195
9.5	Примеры алгоритмов с учетом теории сложности MapReduce	196
9.6	Графовая модель задач в MapReduce	201
9.7	Получение нижних оценок $r(q)$	205
9.8	Контрольные вопросы	206
Глава 10: Облачные вычисления		
10.1	Развертывание облачных вычислений	207
10.2	Платформа Azure	209
10.3	Кластер HDInsight	216
10.4	Контрольные вопросы	219
Глава 11: Введение в Apache Spark		
11.1	Ограничение возможностей Hadoop	220
11.2	Возможности Apache Spark	228
11.3	Запуск на YARN	247
11.4	Контрольные вопросы	265
Глава 12: Apache Spark SQL		
12.1	Основные компоненты Spark SQL	266
12.2	Catalyst Optimizer	272
12.3	Project Tungsten	277
12.4	Контрольные вопросы	282
Глава 13: Apache Hive. Форматы хранения таблиц в HDFS		
13.1	Форматы хранения таблиц	283
13.2	Apache Hive	287
13.3	Фильтр Блума	292
13.4	Optimized Row Columnar File	293
13.5	Дополнительные материалы	295

13.6 Контрольные вопросы	296
Глава 14: Apache Pig	
14.1 Мотивация использования	297
14.2 Функциональные возможности	298
14.3 Операторы диагностики	299
14.4 Режимы запуска Pig	300
14.5 Язык Pig Latin	301
14.6 Контрольные вопросы	310
Глава 15: HBase, BigTable: основы и модель данных	
15.1 Введение в HBase и Bigtable	311
15.2 Особенности организации Column Families	315
15.3 Версионность данных	316
15.4 Хранение и управление данными	318
15.5 Запросы данных	320
15.6 Контрольные вопросы	321
Глава 16: HBase, BigTable: архитектура	
16.1 Архитектура HBase	322
16.2 HBase Region	324
16.3 Условия хранения данных в HBase	327
16.4 Компоненты HBase	328
16.5 Доступ к HBase	329
16.6 Контрольные вопросы	331
Заключение	
Список литературы	

ГЛАВА 1

Введение в большие данные

Данная глава посвящена понятию **больших данных**, их происхождению и влиянию на развитие вычислительной техники. Далее будут рассмотрены различные типы данных, причины роста их объемов и ключевые характеристики. Особое внимание будет уделено значимости обработки больших данных для современных технологий и бизнеса.

За последние несколько десятков лет развитие вычислительной техники испытывает взрывной рост. Всего 40 лет назад бытовой компьютер с 32 КБ оперативной памяти (см. Рисунок 1), внешней памятью — кассетным магнитофоном и устройством отображения — телевизором (часто черно-белым) был мечтой программиста (см. Рисунок 2).



Рис. 1: БК 0010-01 производился в 1983–1993 г. Процессор: частота 3 или 4 МГц, разрядность 16 бит, 32 КБ ОЗУ (из них 16 КБ — видеопамять). “Цены на БК-0010-01 были сопоставимы с ценой цветного телевизора или хорошего музыкального центра и в 2–3 раза выше заработной платы инженера...”¹

¹[http://wiki-org.ru/wiki/БК_\(семейство_компьютеров\)](http://wiki-org.ru/wiki/БК_(семейство_компьютеров))

(a) Кассетный магнитофон ²(b) Черно-белый телевизор ³

Рис. 2: Техника 1980-х годов

Уже в 2010-х годах были доступны смартфоны с высокопроизводительными графическими ускорителями, в десятки миллионов раз мощнее БК 0010-01. Эволюция виртуальных машин, предоставляемых облачными сервисами Microsoft Azure, представлена в Таблице 1. Здесь vCPU — количество виртуальных процессоров, ОЗУ — объем оперативной памяти (в Гибибайтах), Хранилище — объем временного хранилища (SSD, в Гибибайтах), Сеть — максимальная пропускная способность сетевого интерфейса (в Гигабитах в секунду).

В 2017 году самый мощный в мире китайский суперкомпьютер, изображенный на Рисунке 3, имел 10 миллионов ядер и занимал внушительную площадь в центре обработки данных.

В начале 2025 года наступила эпоха эксафлопсных суперкомпьютеров. Лидеры мирового рейтинга Top500 преодолели рубеж в один эксафлопс про-

²http://www.shizaudio.ru/audio/./data/media/24/_302-1.jpg

³http://rw6ase.narod.ru/index1/tw/tw_cw_stand1/spektr_c380d.html

⁴<https://azure.microsoft.com/en-us/blog/largest-vm-in-the-cloud/>

⁵<https://azure.microsoft.com/en-us/blog/announcing-the-lv2-series-vms-powered-by-the-amd-epyc-processor/>

⁶<https://learn.microsoft.com/en-us/azure/virtual-machines/ecasv5-ecadsv5-series>

⁷<https://learn.microsoft.com/en-us/azure/virtual-machines/ecesv5-ecedsv5-series#ecesv6-series-specifications>

⁸<https://www.newsweek.com/china-plans-super-supercomputer-544441>

Таблица 1: Эволюция виртуальных машин Microsoft Azure (2015–2024)

Серия	vCPU	ОЗУ, ГиБ	Хранилище, ГиБ	Сеть, Гбит/с	Особенности
G-Series (2015) ⁴	до 32	до 448	до 1023	до 32	Intel Xeon E5, большие объемы памяти и локального SSD
Lv2-Series (2017) ⁵	до 80	до 640	до 2000	до 32	AMD EPYC, оптимизирована для памяти
Dv5-Series (2021) ⁶	до 96	до 672	до 2400	до 32	Новые процессоры Intel Xeon, улучшенная производительность
Dsv6-Series (2024) ⁷	до 192	до 1800	до 10560	до 200	Поддержка NVMe, Azure Boost, улучшенная сеть



Рис. 3: Sunway TaihuLight ⁸, Национальный суперкомпьютерный центр. 93 петафлопса, 40960 RISC процессоров, 4+256 ядер в каждом, частота — 1,45 ГГц, 10,6496 млн ядер, Оперативная память 1,31 ПБ, Пространство 605 м²

изводительности. Самым мощным суперкомпьютером по состоянию на 2025 год является El Capitan (см. Рисунок 4), установленный в США и использующийся для моделирования ядерных процессов, а также для передовых научных исследований в области материаловедения, исследования космоса, климата и развития искусственного интеллекта. Его производительность составляет около 1,74 эксафлопса. В системе более 11 миллионов ядер: используется комбинация серверных процессоров AMD EPYC 4-го поколения (24 ядра на чипе) и новейших графических ускорителей AMD Instinct MI300A, объединенных высокоскоростной сетью Cray Slingshot-11.



Рис. 4: El Capitan ⁹, Национальная лаборатория им. Лоуренса

1.1 Большие данные

Большую роль в развитии вычислительной техники и программного обеспечения сыграли технологии хранения данных. Новые технологии хранения

⁹<https://cq.ru/articles/tech/luchshie-superkompiutery-mira-i-zachem-oni-nuzhny>

данных мотивируют собирать и обрабатывать все большие объемы данных. А это стимулирует развитие других направлений, связанных с вычислениями: например, разработку процессоров и программного обеспечения. Начало цифровой эпохи, ознаменованное превышением объема цифровых данных над аналоговыми, представляет собой важную веху в истории информационных технологий (см. Рисунок 5). Этот переход является следствием развития технологий хранения данных.

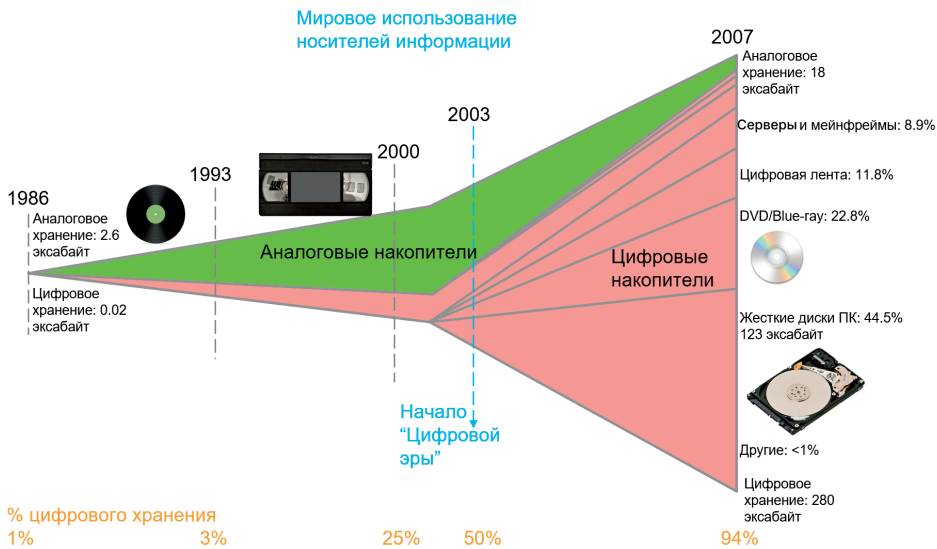


Рис. 5: Использование носителей информации ¹⁰

Объемы генерируемых данных стремительно растут: за период 2016–2017 годов было создано больше данных, чем за всю предшествующую историю человечества. В ближайшее время совокупный объем глобального цифрового пространства достигнет приблизительно 44 зеттабайт, что эквивалентно 44 триллионам гигабайт.

¹⁰<https://www.martinhilbert.net/worldinfocapacity.html/>

1.2 Какие данные бывают?



Рис. 6: Разновидности данных

Рост объемов данных важно рассматривать также с точки зрения их разнообразия (см. Рисунок 6). Первые информационные системы были ориентированы на автоматизацию бухгалтерского учета и оперировали документами. В современном контексте основное внимание уделяется поведенческим данным, которые представляют наибольший интерес для банков, страховых компаний и розничных сетей. Анализ поведения клиентов является важным инструментом для принятия управленческих решений. Например, такими данными являются показатели датчиков мобильных устройств, транзакции по банковским картам и активность пользователей в социальных сетях. Еще одним широким классом данных является контент: электронные сообщения, мультимедийные данные (аудио, изображения, видео). Также при анализе данных и принятии решений в бизнесе часто полезно принимать во внимание не только поведенческие данные и контент, но и данные о социальных связях клиентов. Например, в задачах кредитного скоринга и антифрода (борьбы с

мошенничеством) полезно анализировать друзей клиента в соцсетях, данные созаемщиков, поскольку их поведение частично отражает поведение самого клиента.

1.3 “Правило четырех V”

В контексте больших данных часто возникает широко известное “**правило четырех V**”, схема которого приведена на Рисунке 7. Это четыре недостатка, присущих большим данным.

Первый недостаток очевиден — это большой объем (**Volume**). Большой адронный коллайдер собирает 15 ПБ данных в год. Объем данных Large Synoptic Survey Telescope превышает 6 ПБ в год (около 20 ТБ за ночь).

Velocity говорит о высокой интенсивности появления данных и необходимости их обрабатывать зачастую в реальном времени. Например, в финансовой сфере компаниям интересно знать, что происходит с клиентом в данный момент. Где он сейчас территориально находится, что он делал в Интернете минуту назад, заходил ли он на страницу другого банка (возможно, ему уже отказали в выдаче кредита), какие поисковые запросы выполнял (возможно, это был запрос: “Где купить справку с подтверждением зарплаты”).

Variety говорит о многообразии типов больших данных. Большие данные могут иметь вид структурированных таблиц, однако чаще они не структурированы (текст, мультимедийные данные) либо слабо структурированы (смесь структурированных и неструктурированных данных, например расшифровки телефонных разговоров с временными метками произнесения слов и разметкой каналов клиент-оператор).

Пожалуй, самый большой недостаток больших данных — их низкое качество (**Veracity**). Они могут быть недостоверными, противоречивыми, неполными. В бизнесе практикуется покупка данных о поведении своих клиентов у поставщиков данных: Интернет-компаний, мобильных операторов. Эта сфера имеет много нюансов в правовом регулировании. Важно отметить, что

каждый такой источник данных имеет свою степень покрытия клиентов данными. Часто поставщик данных может не обладать данными по значительной части клиентов, данные которых запрашивались покупателем. Данные о клиенте, полученные от альтернативных поставщиков, могут быть противоречивыми. Некоторые поставщики могут предоставлять “грязные” данные, обусловленные неточной идентификацией клиента по ключевым полям. Бытует мнение, что качественных больших данных не бывает.

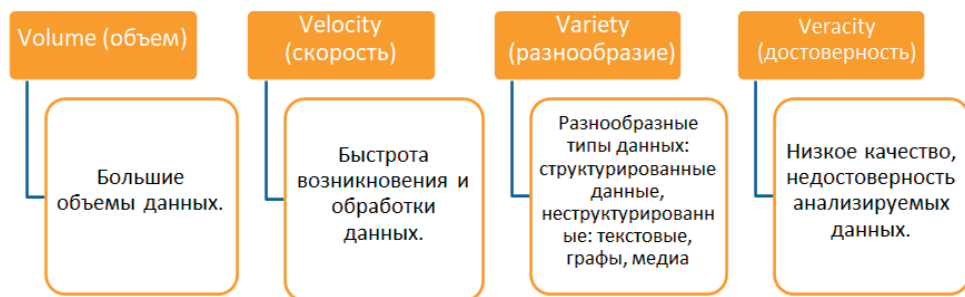


Рис. 7: “Правило четырех V”

1.4 Большие данные в широком смысле

Не существует точного определения термина большие данные. В бизнесе большими данными часто называют не обязательно большие данные, а просто данные о клиентах, поставляемые компании внешними поставщиками данных. С точки зрения обработки и анализа можно выделить три направления больших данных (см. Рисунок 8):

- Сбор и первичная обработка;
- Статистический анализ;
- Машинное обучение.



Рис. 8: Направления больших данных

1.5 Ввод-вывод и обработка запросов

Сбор и первичная обработка больших данных — предмет исследований в области разработки высоконагруженных систем. Проблематика разработки высоконагруженных систем весьма широка, рассмотрим в качестве примера одну из проблем, иллюстрирующую техническую сложность построения высоконагруженных систем: обработку большого числа запросов в модели клиент-сервер. Бэкенды высоконагруженных систем должны обеспечивать обслуживание сотен тысяч одновременных запросов, поступающих от клиентов через соединения в рамках механизма сокетов. Естественными вариантами обработки запросов являются модели **“нить на запрос”** либо **“процесс на запрос”**, которые схематически изображены на Рисунке 9. Однако эти модели неэффективны при большом числе запросов: порождаются сотни тысяч параллельных нитей или процессов операционной системы.

Высокие издержки на обработку такого числа параллельных сущностей операционной системой делают данные модели нежизнеспособными. Одним из возможных решений является использование так называемых **легковесных задач (co-routine)** (Рисунок 10), которые псевдопараллельно исполняются в рамках одной нити операционной системы. Очередность исполнения легковесных задач определяется фактом прихода данных через асинхронные сокеты (подробности можно посмотреть в документации по системным вызовам `select`, `poll`, `epoll` или в документации по асинхронным функциям в

Python, C++, библиотеке Twisted Framework и др.). Часть легковесных задач, требующих процессорного времени, т.е. подразумевающих обработку данных, может быть исполнена в рамках пула нитей. Такая архитектура хорошо зарекомендовала себя в бэкендах высоконагруженных систем.

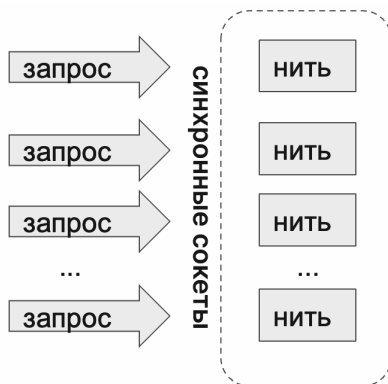


Рис. 9: Модель “нить на запрос”

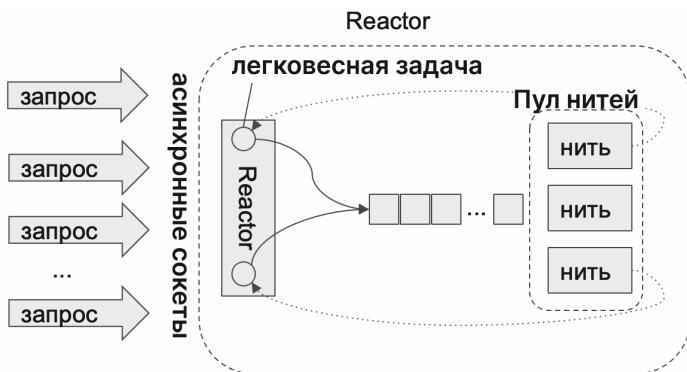


Рис. 10: Использование легковесных задач

1.5.1 Реляционные системы управления базами данных (РСУБД)

Сбор, обработку и анализ данных трудно себе представить без **реляционных систем управления базами данных**. Эта технология, проверенная десятилетиями, основана на следующих концепциях:

- Фиксированная схема базы данных (БД);
- Нормализованные таблицы;
- Использование языка SQL (Structured query language);
- Поддержка транзакций, индексирования;
- Развитые возможности по оптимизации, производительности, хранению данных, планов исполнения запросов.

Следует отметить, что РСУБД долгое время являлись стандартным способом интеграции приложений. Однако к недостаткам РСУБД следует отнести:

- Проблему несоответствия реляционной модели и объектно-ориентированной, используемой в бизнес-логике приложений (бизнес-логика разрабатывается в терминах классов, методов, наследования, а в РСУБД есть только таблицы);
- Плохую горизонтальную масштабируемость;
- Коммерческое лицензирование многих “продвинутых” реализаций РСУБД, используемых в индустрии (например, в бизнесе долгое время был очень популярен дорогостоящий Oracle).

Важно отметить, что тенденциями развития информационных систем являются:

- Рост объемов данных;
- Горизонтальное масштабирование, использование кластеров;
- Уход от баз данных как средства интеграции систем (SOA);
- Развитие веб-приложений, частая модификация их логики и структуры хранения данных.

Бурное распространение информационных систем, использование гибких методологий разработки, подразумевающих частую модификацию логики и структур хранения данных, определили новые требования к системам хранения данных:

- Горизонтальная масштабируемость;
- Отсутствие схемы данных — разработчикам неудобно жить в условиях жесткой типизации структур данных, вносить изменения и доработки и в программный код, и в скрипты создания таблиц, а также обновлять их структуру. Возникло желание управлять данными процессами только в коде на языке программирования (ЯП).

1.5.2 NoSQL

Плохое соответствие РСУБД обозначенным требованиям привело к появлению **Not Only SQL (NoSQL)** баз данных. Существует несколько типов СУБД NoSQL. Их разновидностями и примерами программных реализаций являются:

- Ключ-значение (Berkley DB, MemcacheDB, AeroSpike);
- Документные (MongoDB);
- Семейство столбцов (HBase);
- Графовые.

Далее рассмотрим классический тип — БД **ключ-значение**.

1.5.3 NoSQL БД ключ-значение

В БД ключ-значение используется аналог таблиц РСУБД. Однако в данном случае таблицы имеют всего два столбца: ключ и значение. Каждая таблица обычно (но не всегда) соответствует сущности предметной области, ключом является идентификатор сущности, значением — значения сразу всех или

многих атрибутов сущности, представленные байтовым массивом. Структура этого байтового массива не фиксируется в БД. Разбор значений осуществляется в программном коде. Основными операциями являются CRUD-операции (**create**, **read**, **update**, **delete**) для заданного ключа. БД ключ-значение как правило не поддерживают соединение таблиц. При необходимости соединения создается новая таблица, содержащая поля соединяемых таблиц. Таким образом, для БД ключ-значение характерно дублирование данных (см. Таблицы 2 и 3).

Таблица 2: Таблица покупателей

Ключ	Значение
1	{Name: Иванов, Age: 20}
2	{Name: Петров, Age: 25}
3	{Name: Сидоров, Age: 40}

Таблица 3: Таблица заказов

Ключ	Значение
1	{CustomerId:1, Date:2014-06-16, OrderLines: [{Name:футболка, Price:500, Count:1}, {Name:шорты, Price:600, Count:1}]}
2	{CustomerId:1, Date:2014-06-17, OrderLines: [{Name:футболка, Price:500, Count:2}, {Name:шорты, Price:600, Count:1}]}
3	{CustomerId:2, Date:2014-06-18, OrderLines: [{Name:футболка, Price:500, Count:3}]}

Таким образом, основные концепции БД ключ-значение:

- “Значение” — массив байт. Запросы по значениям отсутствуют;

- Агрегатно-ориентированность. Структуры данных в значительной степени основываются на запросах к ним. Скорость важнее нормализованного хранения.

1.5.4 Масштабирование: горизонтальный sharding

Основным механизмом горизонтальной масштабируемости БД ключ-значение является **горизонтальный sharding** (см. Рисунок 11). В данном случае содержимое таблицы размещается на нескольких серверах. Каждый сервер содержит записи, соответствующие некоторому диапазону ключей или значению хеш-функции от ключа. Клиентское приложение знает, к какому серверу ему следует обращаться по данному ключу, или же запрос клиента маршрутизируется отдельной компонентой БД. Такая схема позволяет неограниченно масштабировать запросы при росте объемов данных в таблице.

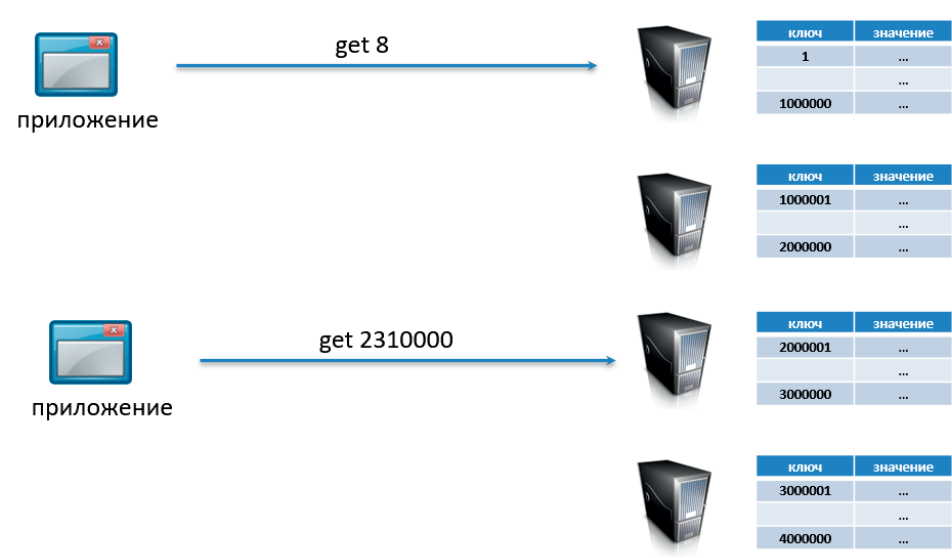


Рис. 11: Горизонтальный sharding

1.5.5 Масштабирование: репликация master-slave

Другим вариантом масштабирования, который используется в том числе в некоторых РСУБД, является **репликация** — дублирование таблицы на несколько серверов. В репликации “**master-slave**” (см. Рисунок 12) один из серверов (master) принимает запросы на модификацию данных. Остальные серверы (типа slave) обрабатывают читающие запросы (поскольку читающих запросов часто больше пишущих). Master распространяет произведенные им изменения таблицы на slave-серверы.



Рис. 12: Репликация master-slave

1.5.6 Масштабирование: одноранговая репликация

В **одноранговой репликации** (см. Рисунок 13) каждый из серверов способен читать и модифицировать свою копию таблицы. После модификации таблицы на одном из серверов изменения распространяются на остальные

серверы. При любом варианте репликации возникает проблема консистентности данных и транзакционной обработки. Это объясняется тем, что клиентское приложение может прочесть данные с сервера, на который еще не были распространены изменения данных, осуществленные на другом сервере.



Рис. 13: Одноранговая репликация

1.6 Статистический анализ больших данных

Перейдем к проблеме статистического анализа больших данных (см. Рисунок 14). Новые данные, собранные OLTP (онлайн транзакционными) БД, с некоторой периодичностью передаются в так называемое **аналитическое хранилище** (Reporting Database или база данных отчетности). С аналитическим хранилищем работают аналитики и исследователи данных (data scientists). Целью первых является поиск в данных полезных закономерностей с помощью статистических запросов. Data scientists заинтересованы в построении моделей машинного обучения на основе собранных в хранилище данных. Также на основе таблиц аналитического хранилища создаются регулярные аналитические отчеты, необходимые “бизнесу” в рамках своей операционной

деятельности. Структура таблиц аналитического хранилища отличается от структуры таблиц OLTP БД: его таблицы удобны для анализа, формирования признаков моделей машинного обучения и построения отчетов. Базовыми технологиями организации аналитического хранилища являются:

- РСУБД (Аналитические SQL-запросы);
- OLAP (Online Analytical Processing) — технология многомерного анализа данных, позволяющая быстро получать значения агрегационных функций (в том числе сложных) в разрезе нескольких измерений;
- Пакетная обработка структурированных/неструктурированных/плохо структурированных данных (например, журналов);
- Многие аналитические хранилища для обработки как табличных, так и неструктурированных данных строятся на базе стека технологий Apache Hadoop и Apache Spark. Соответствующие технологии: Hadoop-стек (Hive, Pig, Impala), Spark (SparkSQL). Интерфейсы: программный код (Python, Java, Scala), SQL, специализированные языки (Pig Latin, HiveQL).

1.6.1 Поколоночное хранение

Интересной особенностью современных аналитических хранилищ является **поколоночное** хранение содержимого таблиц в оперативной памяти и на жестком диске. При выполнении статистических запросов колончатый формат хранения оказывается предпочтительнее традиционного построчного (см. Рисунок 15). В частности, требуется чтение значений лишь тех столбцов, по которым считается статистика.

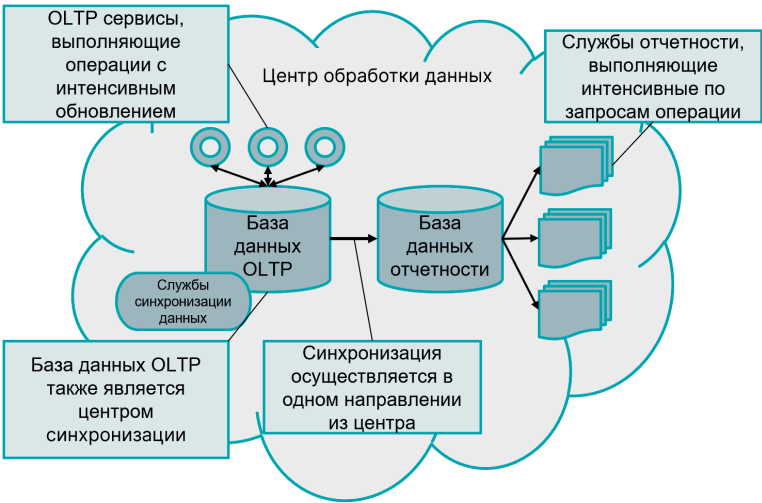


Рис. 14: Статистический анализ

Клиент	Продукт	Дата	Сумма		Клиент	Продукт	Дата	Сумма
Иванов	TV	01.02.2025	8000	↔	Иванов	TV	01.02.2025	8000
Петров	Навигатор	01.02.2025	1000		Петров	Навигатор	01.02.2025	1000
Сергеева	Смартфон	02.02.2025	7000		Сергеева	Смартфон	02.02.2025	7000
Матвеева	Смартфон	03.02.2025	9000		Матвеева	Смартфон	03.02.2025	9000

Построчное хранение

Поколоночное хранение

Рис. 15: Построчное и поколоночное хранение

1.6.2 Лямбда-архитектура

Рассмотрим современные архитектуры информационных систем, предназначенных для обработки больших данных. Наиболее популярными архитектурами являются **Лямбда** и **Каппа-архитектуры**. Обработку данных можно разделить на пакетную — обработку уже собранных данных и потоковую — обработку событий, которые приходят в систему в данный момент. Лямбда-архитектура (см. Рисунок 16) подразумевает одновременное журналирование всех событий (так называемый **Batch Layer**) и формирование реакции на событие в реальном времени (**Speed Layer**). Однако для формирования

реакции на событие в реальном времени могут потребоваться исторические данные из журнала, сбор которых затруднен (поскольку это журналируемая информация).

Рассмотрим **пример** работы Интернет-банка. Допустим, пришло событие “клиент зашел в Интернет-банк”. Как необходимо обработать это событие? Показать или не показать клиенту спецпредложение? Показывать его при каждом заходе не нужно, это может вызвать раздражение клиента. Банк решает показывать спецпредложение на каждый 10-й вход. В Batch Layer сохранится сам факт захода (например, идентификатор клиента, время). Как Speed Layer определит, следует ли показать спецпредложение? Долгое решение: обращаться к Batch Layer, находить там данного клиента и подсчитывать число заходов в Интернет-банк. Вместо этого за подсчет и хранение числа заходов на сайт каждого клиента отвечает **Serving Layer**. Теперь Speed Layer имеет возможность формировать решение на основе входного события и данных, полученных из Serving Layer. Важно, что Serving Layer всегда можно заполнить с нуля на основе журнала, которым является Batch Layer.

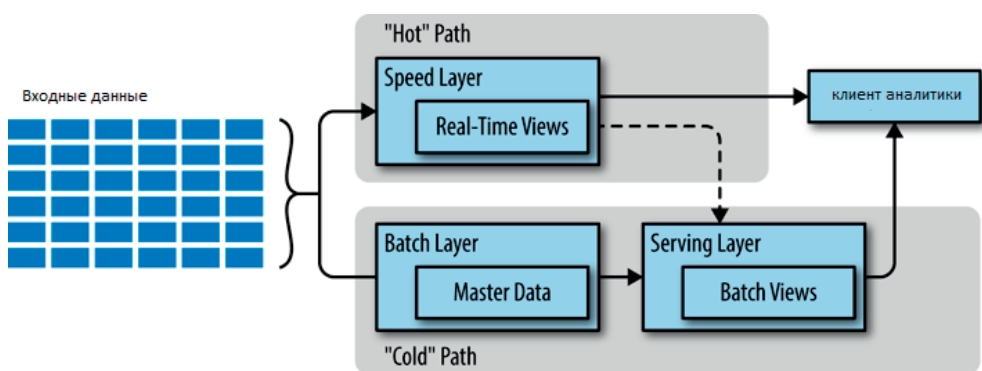


Рис. 16: Схема Лямбда-архитектуры [1]

Особенности Лямбда-архитектуры:

- Неизменчивость Master Data;
- Приближенные облегченные запросы к данным в “hot” path;

- Частичное дублирование логики в “cold” и “hot” paths;
- Устойчивость к сбоям (fault tolerance).

1.6.3 Каппа-архитектура

Каппа-архитектура представляет собой Лямбда-архитектуру без Serving Layer. Рассмотрим **пример** с банком. Предположим, что показ спецпредложения определяется тем, является ли заход клиента на сайт повторным за последнюю минуту. В такой ситуации обработку событий в Speed Layer и **Long Term Store** (так в Каппа-архитектуре называется Batch Layer) можно проводить полностью единообразно, не привлекая дополнительную базу для хранения статистики. Каппа-архитектура имеет ограниченное применение: она отлично подходит для предметных областей вроде алгоритмического трейдинга и плохо применима, когда Speed Layer принимает решение на основе глубокой истории событий (такое чаще всего бывает в случае взаимодействия с клиентом в банках, страховых и других компаниях). Особенностью Каппа-архитектуры является единая логика пакетной и потоковой обработки.

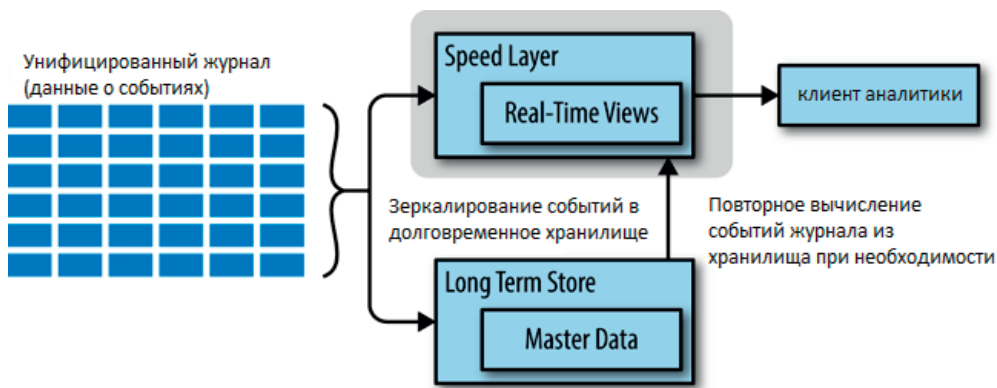


Рис. 17: Схема Каппа-архитектуры [1]

1.7 Машинное обучение на больших данных

Целью данного раздела является объяснение и демонстрация проблем построения моделей машинного обучения на больших данных. Однако для начала необходимо кратко рассказать о самом машинном обучении.

Известная фраза банкира Натана Майера Ротшильда “Кто владеет информацией — тот владеет миром” на самом деле больше относится не просто к информации, а к знаниям, извлеченным из нее. Именно получение полезных знаний и является главной целью интеллектуального анализа данных (Data Mining).

1.7.1 Data Mining

Data Mining — это совокупность методов обнаружения в данных ранее неизвестных, нетривиальных, практически полезных и доступных интерпретаций знаний, необходимых для принятия решений в различных сферах человеческой деятельности.

Data Mining основывается на таких областях прикладной математики, как теория вероятностей, математическая статистика, методы оптимизации, линейная алгебра (см. Рисунок 18). Сегодня Data Mining требует от специалиста познаний и в области инфраструктур больших данных [2], [3].

В Data Mining выделяют два тесно связанных направления: **Knowledge discovery** и **Decision Support** (см. Рисунок 19). Целью **Knowledge Discovery in Databases (KDD)** является обнаружение новых и ценных знаний в данных, которые помогают решать реальные бизнес-задачи. **Машинное обучение (Machine learning)** используется как инструмент построения точных математических моделей, позволяющих в автоматическом режиме делать прогнозы, необходимые в соответствующей предметной области. Таким образом, целью **систем поддержки принятия решений (Decision Support)** является автоматизированное построение точной модели машинного обучения для ее последующего использования.



Рис. 18: Основы Data Mining

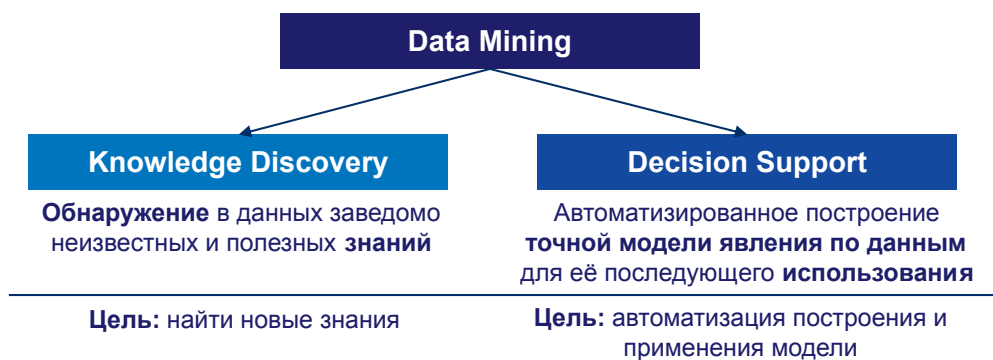


Рис. 19: Разновидности Data Mining

1.7.2 Базовая терминология Data Mining

Рассмотрим базовую терминологию и задачи Data Mining на примере радиолокационной станции (РЛС), собирающей такие параметры о летательных аппаратах, как скорость, ускорение, вертикальная скорость, высота (см. Таблицу 4). Перечисленные характеристики называются **атрибутами**. Атрибут

может быть **непрерывным**: тогда между его значениями можно определить расстояние, либо **категориальным** — без расстояния. Объект предметной области, по которому собираются атрибуты, является **прецедентом**. Несколько объектов с измеренными значениями атрибутов называются **набором данных**. Все атрибуты из примера — **непрерывные**. Примеры **категориальных** атрибутов — погодные условия (нормальные, осадки, шторм).

Таблица 4: Параметры РЛС

Скорость, м/сек	max ускорение, м/сек ²	max верт. скорость, м/сек	min высота, м
710	20	110	2000
1100	100	300	500
820	25	150	2500
720	20	115	2400
740	22	120	2000

1.7.3 Классификация

Одна из самых популярных задач машинного обучения — **классификация**. Данная задача заключается в построении модели, способной прогнозировать значение некоторого категориального атрибута объекта, называемого **целевой переменной**, по известным значениям остальных атрибутов (их называют **входными атрибутами**). Модель строится на **обучающей выборке** — наборе данных, где известны значения как входных атрибутов, так и целевой переменной. Модель применяется на выборке, в которой на момент применения значение целевой переменной неизвестно. В нашем примере целевой переменной может быть класс летательного средства. Если число возможных значений целевой переменной равно двум, такая задача называется **бинарной классификацией**. Если число значений более двух — **многоклассовой классификацией**.

Таблица 5: Параметры РЛС для задач классификации

Скорость, м/сек	max ускорение, м/сек ²	max верт. скорость, м/сек	min высота, м	Тип летательного средства
710	20	110	2000	C2
1100	100	300	500	C2
820	25	150	2500	C1
720	20	115	2400	
740	22	120	2000	

Например, мы можем взять первые три записи из Таблицы 5 в качестве обучающей выборки, на их основе построить модель и предсказать классы двух последних записей таблицы.

С точки зрения математики, классификация сводится к построению поверхностей, которые наилучшим образом разделяют точки в пространстве значений входных атрибутов по целевым классам. Эти поверхности могут быть как простыми — плоскими или линейными, так и иметь более сложную форму (см. Рисунок 20).

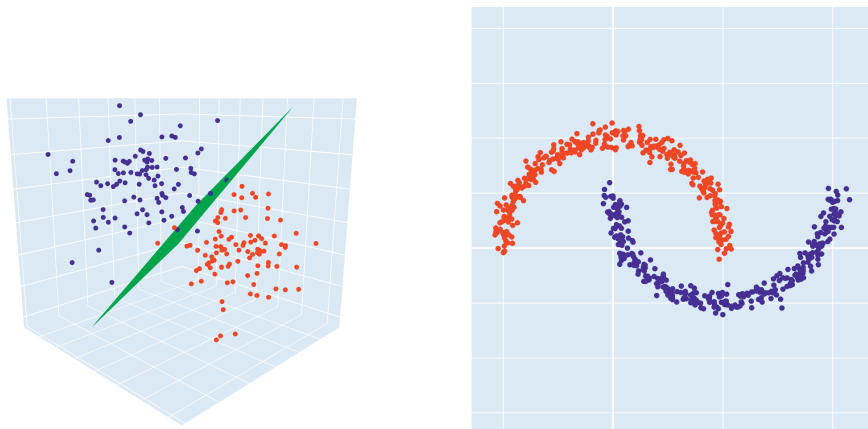


Рис. 20: Разделяющие поверхности

1.7.4 Логистическая регрессия

Классическим алгоритмом бинарной классификации является **логистическая регрессия**. Она предполагает линейную разделимость классов и подбирает оптимальные коэффициенты разделяющей гиперплоскости с помощью метода максимального правдоподобия на обучающей выборке. Для моделирования вероятности класса целевой переменной используется **логистическая кривая (сигмоида)** (см. Рисунок 21).

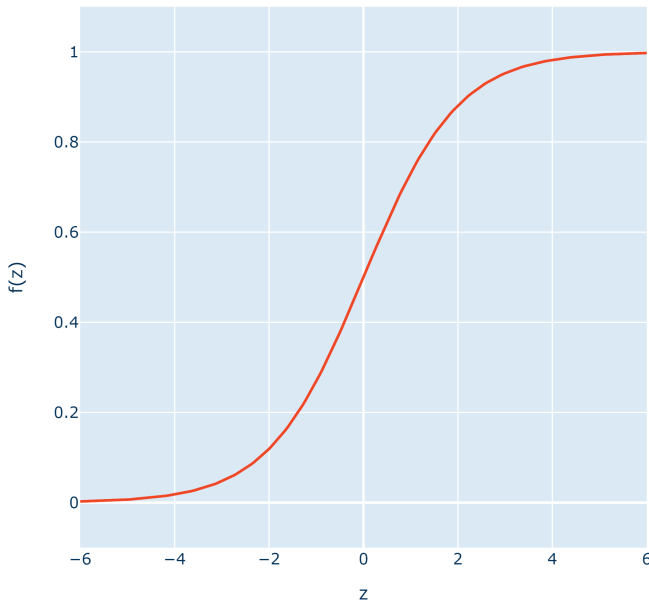


Рис. 21: График сигмоиды

$$\mathbb{P}\{y = 1|x\} = f(z), f(z) = \frac{1}{1 + e^{-z}},$$

$$z = \theta^T x = \theta_1 x_1 + \dots + \theta_n x_n,$$

$$\mathbb{P}\{y = 0|x\} = 1 - f(z) = 1 - f(\theta^T x),$$

$$\mathbb{P}\{y|x\} = f(\theta^T x)^y (1 - f(\theta^T x))^{1-y}, y \in \{0, 1\},$$

$$\begin{aligned}\hat{\theta} &= \operatorname{argmax}_{\theta} L(\theta) = \operatorname{argmax}_{\theta} \prod_{i=1}^m \mathbb{P}\{y = y^{(i)} | x = x^{(i)}\}, \\ \ln L(\theta) &= \sum_{i=1}^m \log \mathbb{P}\{y = y^{(i)} | x = x^{(i)}\}, \\ \ln L(\theta) &= \sum_{i=1}^m y^{(i)} \ln f(\theta^T x^{(i)}) + (1 - y^{(i)}) \ln(1 - f(\theta^T x^{(i)})).\end{aligned}$$

1.7.5 Прогнозирование

Задача прогнозирования или **задача восстановления регрессии** отличается от классификации непрерывностью целевой переменной. Например, в задаче с параметрами РЛС, самолеты могут иметь параметр “время нахождения в зоне контроля”: по обучающей выборке, аналогично классификации, восстанавливаются пропущенные значения (см. Таблицу 6).

Таблица 6: Параметры РЛС для задач прогнозирования

Скорость, м/сек	max уско- рение, м/сек ²	max верт. ско- рость, м/сек	min высо- та, м	Время нахож- дения в зоне контроля, сек
710	20	110	2000	2.1
1100	100	300	500	3.4
820	25	150	2500	8
720	20	115	2400	
740	22	120	2000	

И классификация, и прогнозирование сводятся к построению функции $f(x)$ оптимальным образом на основе обучающей выборки: входные признаки — $X = x_1 x_2 \dots x_n$, целевая переменная — y , обучающая выборка — $X_1, y_1; X_2, y_2; \dots; X_n, y_n$.

Задачи классификации и прогнозирования активно используются в ключевых сценариях банков, страховых компаний, сетей супермаркетов. Они ста-

ли незаменимым инструментом и за пределами бизнеса: в медицине и биоинформатике (для разработки лекарств) (см. Таблицу 7).

Таблица 7: Классификация и прогнозирование

Задача	X	y
Персонификация контента пользователя соцсети	“Лайки” и контент	Рекомендованный контент
Рекомендации товаров и услуг	История покупок пользователя, “лайки”	Набор рекомендованных товаров и услуг
Карточный антифрод	Последние транзакции	Решение: фрод/ норма
Кредитный скоринг с использованием данных соцсетей	Анкетные данные, поведение и контент, связанный с пользователем в Интернете, соцсетях	Решение по кредиту
Поиск взаимосвязей между физической активностью, питанием и заболеваемостью	Общая информация о пациенте, данные диетологических дневников, показатели датчиков	Набор вероятных диагнозов

1.7.6 Кластеризация

Еще одной задачей машинного обучения является **кластеризация**. Основной целью является поиск структуры в данных, а именно поиск групп похожих друг на друга прецедентов с точки зрения значений атрибутов. Кластерный анализ часто применяют именно с целью KDD.

Например, в задаче с параметрами РЛС можно поставить цель — найти группы “похожих” летательных аппаратов, пролетающих над РЛС (см. Таблицу 4 и Рисунок 22).

В кластерном анализе возникает необходимость в подсчете расстояния между объектами. Поэтому большинство алгоритмов кластеризации вычислительно сложны: квадратичны по числу объектов в наборе данных. Кластеризация может быть полезна финансовым компаниям для лучшего понимания структуры своих клиентов и создания продуктов, соответствующих найденной структуре.

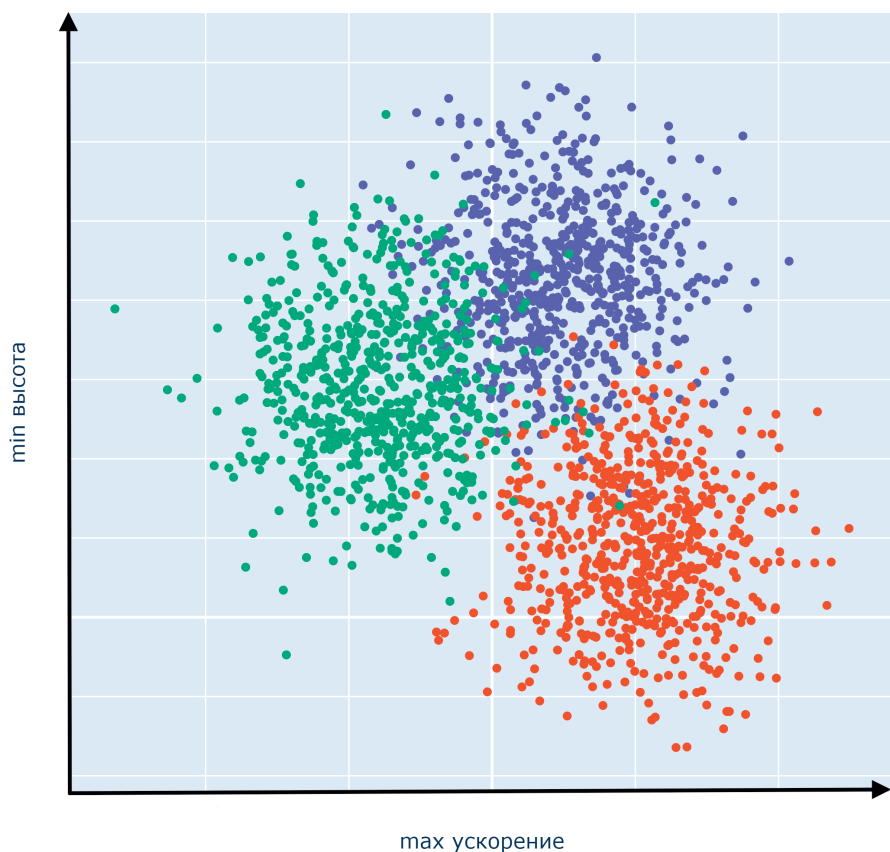


Рис. 22: Результат кластерного анализа ¹¹

¹¹<https://kamalmeet.com/machine-learning/k-means-clustering/>

1.7.7 Обнаружение аномалий

Задача **обнаружения аномалий** похожа на задачу кластеризации. Однако ее целью является поиск **аномалий** — прецедентов, которые не вошли в **кластеры** — группы похожих друг на друга объектов (см. Рисунок 23). Поиск аномалий может быть полезен в обнаружении мошенничества в транзакциях пластиковых карт, а также при прогнозировании поломок оборудования.

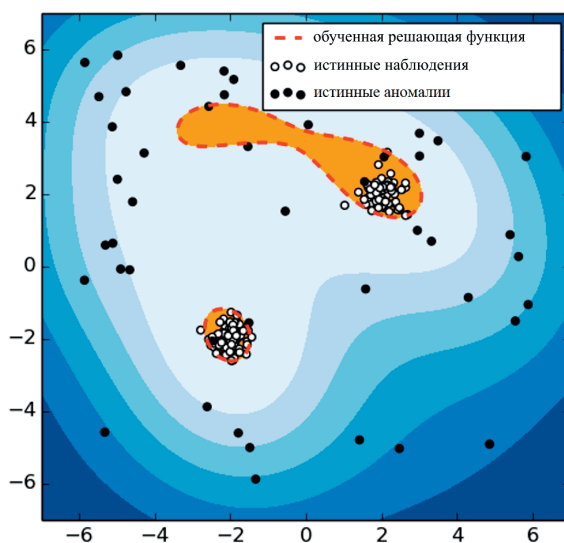


Рис. 23: Поиск в наборе данных “нетипичных” прецедентов ¹²

1.7.8 Применение методов машинного обучения на больших данных

Рассмотрим сложности использования методов Data Mining и, в частности, машинного обучения на больших данных:

- Полиномиальная сложность алгоритмов по числу атрибутов;

¹²https://scikit-learn.org/0.15/auto_examples/covariance/plot_outlier_detection.html

- Низкое качество данных;
- Для обучения глубоких нейронных сетей характерны интенсивные вычисления;
- В задачах обучения с подкреплением (Reinforcement Learning, RL) обучение моделей производится на очень больших выборках.

Даже на современном уровне развития вычислительной техники обработка и анализ больших объемов данных являются проблемами из-за дисбаланса скоростей компонентов вычислительных систем. В 2015 году ситуация выглядела следующим образом (см. Рисунок 24). Жесткий диск, условно, в 500 раз медленнее оперативной памяти. Сеть — в 50 раз медленнее. А собранные ретроспективные большие данные лежат во внешней памяти жесткого диска.

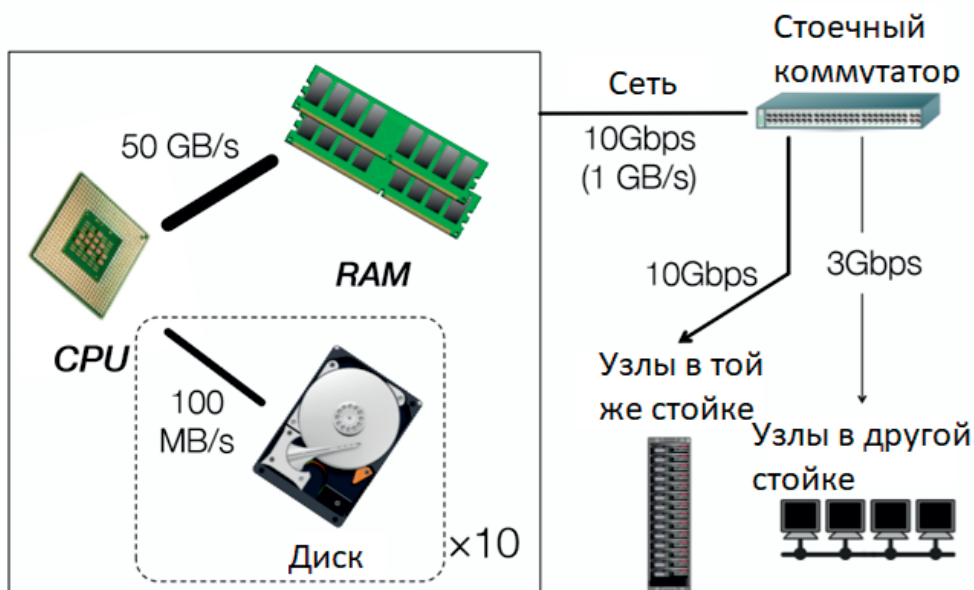


Рис. 24: Скорости компонентов вычислительных систем в 2015 году

С распространением твердотельных накопителей (SSD) дисбаланс сгладился, но разница по-прежнему велика (см. Таблицу 8). SSD не заменил RAM,

расположившись между RAM и HDD. Также SSD имеет быстрый Random Access.

Таблица 8: Скорости компонентов в 2017 году

	Послед. чтение, ГБ/сек	Послед. запись, ГБ/сек	Чтение со случайным доступом, IOPS/сек	Запись со случайным доступом, IOPS/сек	Цена, USD/GB
HDD	0,15-0,2	0,15-0,2	80-200	80-200	0,02-0,05
SSD	1,8-3,5	0,56-2,1	150'000- 440'000	128'000- 360'000	0,40-0,60
RAM	25-150				6-15

Также появился GPU с высокоскоростной памятью. Однако дисбаланс скоростей всегда является актуальной причиной, затрудняющей обработку и анализ больших массивов данных (см. Рисунок 25).

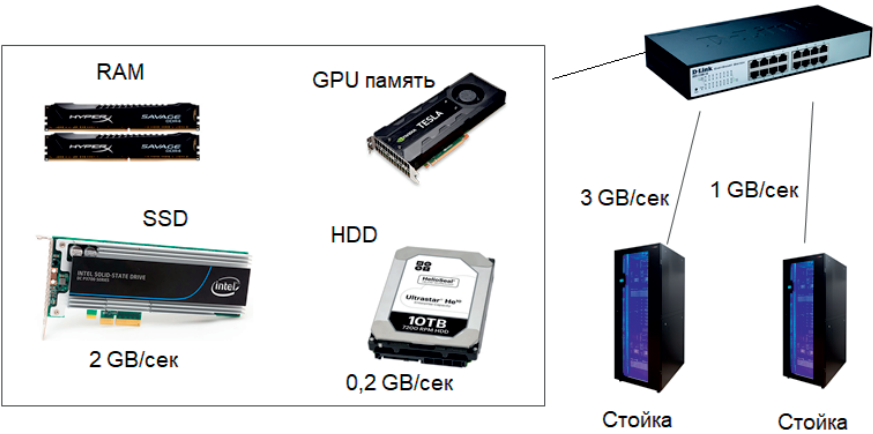


Рис. 25: Скорости компонентов вычислительных систем с появлением GPU

1.8 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

- Что такое большие данные?
- Что является причинами появления больших данных?
- В чем состоят проблемы больших данных и каких типов обработки данных они касаются?
- В чем состоят отличия NoSQL баз данных от реляционных?
- Что такое Каппа и Лямбда-архитектуры?

ГЛАВА 2

Практические сценарии использования больших данных

В данной главе рассматриваются ключевые сценарии применения больших данных и машинного обучения в различных прикладных областях науки и бизнеса. Например, как предсказательные модели позволяют повысить эффективность персонализированного показа баннеров. Рассматриваются особенности построения моделей при наличии дисбаланса классов, разницы в распределениях между обучающей и тестовой выборками, а также необходимость интерпретируемости решений. Также глава содержит обзор современных методов машинного обучения, применяемых для анализа сложных и разнородных данных.

2.1 Сценарий №1. Реклама

Вероятно, одним из первых и наиболее значимых направлений применения больших данных является реклама. Результативность рекламной кампании в значительной степени зависит от того, насколько точно реклама адресована целевой аудитории — это свойство называют таргетированием.

2.1.1 Простой пример

Допустим, с каждой продажи товара компания получает прибыль в 100 рублей. На рекламную кампанию затрачено 5000 рублей, и она охватывает 10 000 человек. Если **конверсия** — то есть доля откликнувшихся на рекламу

— составляет 1,5%, то прибыль составит:

$$10000 \times 0,015 \times 100 \text{ р.} = 15000 \text{ р.}$$

Чем выше конверсия, тем выше итоговая прибыль. Возникает естественный вопрос: как увеличить конверсию? Даже простейшие методы таргетирования могут существенно повысить эффективность кампании.

2.1.2 Интернет-реклама

Реклама столкнулась с большими данными, попав в Интернет. Конверсия от различных действий клиентов в Интернете показана на Рисунке 26.

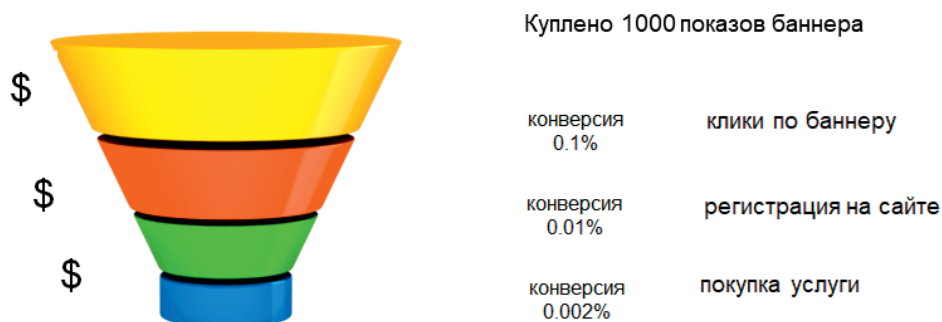


Рис. 26: Конверсия Интернет-рекламы

Персонализированный показ рекламных баннеров в Интернете работает по следующей схеме. Когда пользователь заходит на сайт, на котором размещается реклама (так называемый сайт-издатель), необходимо мгновенно определить, какой именно рекламный баннер ему показать. Для этого сайт формирует запрос на биржу Интернет-рекламы с предложением показа баннера пользователю Y на сайте X .

Рекламодатели получают этот запрос и в автоматическом режиме анализируют информацию: кто этот пользователь, на каком сайте он находится,

насколько для них интересна эта возможность показа. Чтобы принять решение, они обращаются к данным, собранным специализированными DMP-компаниями (Data Management Platform). Эти компании собирают так называемые клик-стримы — последовательности посещенных сайтов и действий пользователя в Интернете, обычно с помощью браузерных cookies.

На основе накопленных клик-стримов и истории реакций пользователей на рекламу (например, фактов кликов) строится модель машинного обучения. Эта модель позволяет оценить вероятность того, что конкретный пользователь кликнет по конкретной рекламе. Именно такую оценку рекламодатель использует во время участия в торгах на бирже, чтобы принять решение: стоит ли бороться за этот показ и какую ставку предложить. Вся процедура — от получения запроса до оценки и участия в торгах — занимает всего несколько миллисекунд. Схема такого взаимодействия представлена на Рисунке 27.

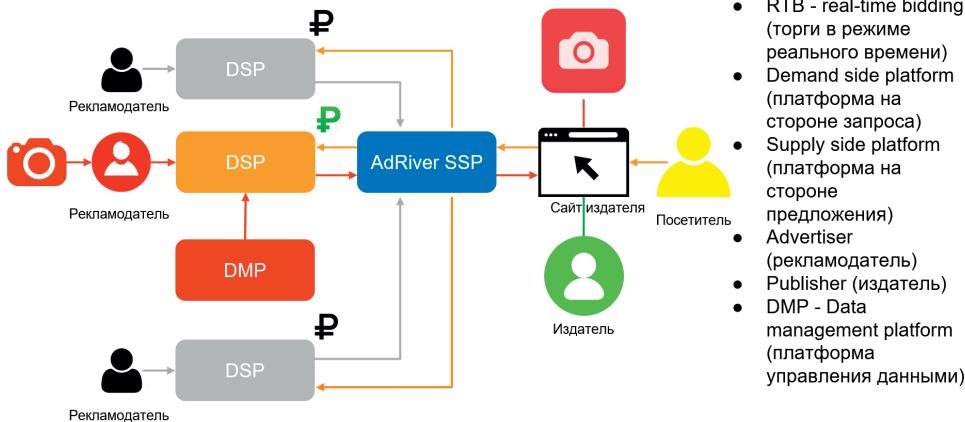


Рис. 27: Real-time bidding ¹³

С каждым годом все больше людей по всему миру пользуются Интернетом, и 2025 год не стал исключением. Как видно на Рисунке 28, доля

¹³<https://www.adriver.ru/doc/rtb/>

Интернет-пользователей продолжает расти. При этом большинство людей используют не одно, а сразу несколько устройств — например, смартфон, планшет и ноутбук.

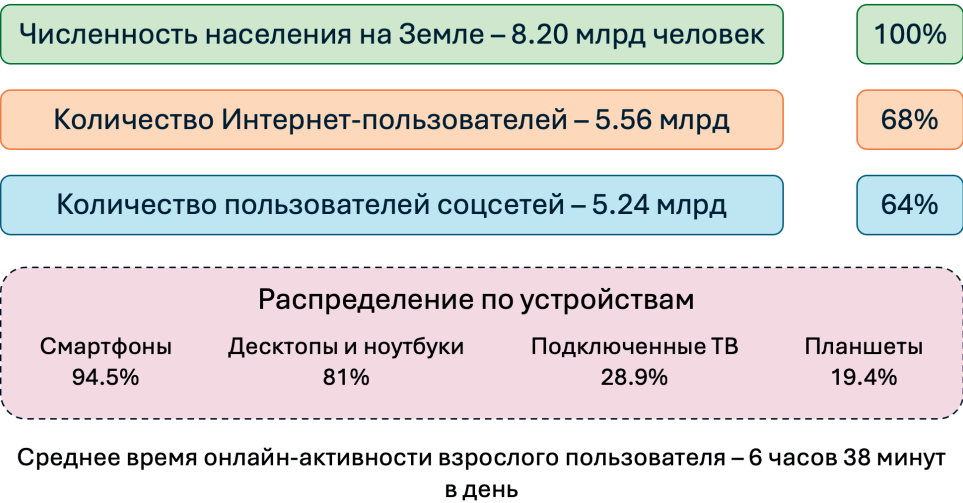


Рис. 28: Статистика Digital 2025 по использованию Интернета

Для моделей таргетирования особенно важны события, произошедшие недавно — то есть данные из последней части клик-стрима пользователя. Это позволяет делать более точные прогнозы о его поведении. Однако такой подход приводит к росту объема обрабатываемой информации. Необходимо учитывать множество взаимодействий пользователей с сайтами и приложениями, поступающих с разных устройств и в режиме реального времени. Именно поэтому задача таргетирования напрямую связана с проблемами больших данных: требуется хранить, обрабатывать и анализировать огромные объемы информации с минимальными задержками.

2.2 Сценарий №2. Кредитный скоринг и другие банковские задачи

Рассмотрим, как методы машинного обучения и технологии работы с большими данными применяются в банковской сфере, особенно в розничном банкинге. Один из ключевых примеров — задача кредитного скоринга.

2.2.1 Задача кредитного скоринга

Задача: автоматически ранжировать заявки на получение кредита по вероятности невозврата.

На первый взгляд, задача кредитного скоринга — это классическая бинарная классификация: клиент либо вернет кредит, либо не вернет. Однако на практике есть важные особенности:

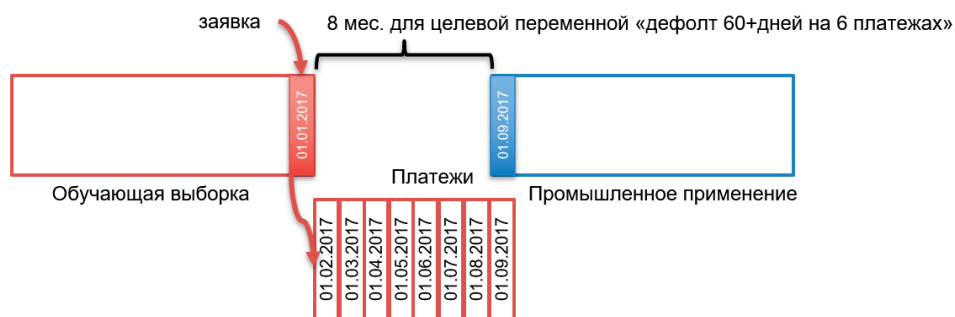


Рис. 29: Задача кредитного скоринга

- Задача относится к бинарной классификации, но с **дисбалансом классов**: большинство клиентов возвращают кредиты, и только небольшой процент допускает просрочку;
- Чтобы понять, вернул клиент деньги или нет, нужно подождать несколько месяцев (обычно 6–12). Поэтому обучающие данные всегда немного устаревшие (см. Рисунок 29);

- Распределение признаков и целевой переменной во времени может меняться: $p(X_{train}, Y_{train}) \neq p(X_{prod}, Y_{prod})$. Это означает, что поведение клиентов и внешняя среда со временем изменяются, и модель может со временем терять актуальность.

Кроме особенностей постановки задачи, следует учитывать специфику входных данных. Современные системы скоринга используют поведенческие данные клиента из множества источников — например, банковских транзакций, данных о кредитной истории, информации из внешних агрегаторов и т.д. Это приводит к появлению:

- Большого количества признаков, часто сильно коррелированных между собой;
- Неполных или неточных данных (не все источники имеют информацию о каждом клиенте, и не вся информация достоверна).

Еще одной особенностью является то, что в кредитном скоринге распределения данных на обучающей выборке и выборке для промышленного применения существенно отличаются. Модель строится на выданных кредитах (заключенных договорах), а применяется к общей популяции клиентов банка. Поэтому некоторые диапазоны значений входных признаков плохо представлены в обучающей выборке (например, высокие значения максимальной продолжительности наличия просроченной задолженности по кредиту), поскольку предыдущая модель не позволила выдать кредит потенциально неблагонадежным клиентам. По этой же причине затруднена оценка качества модели на ретроспективных данных. Значения метрик AUC и lift на множестве выданных кредитов ничего не говорят о точности модели на входящем потоке клиентов. Для объективной оценки модели приходится ждать результатов ее реального применения — так называемый bad rate — % невозвратов кредитов (см. Рисунок 30).

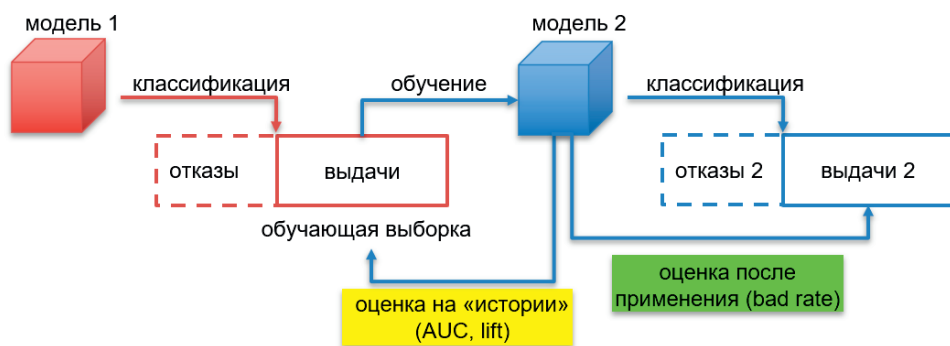


Рис. 30: Схема модели кредитного скоринга

2.2.2 Классический кредитный скоринг

Классическим подходом к решению задачи кредитного скоринга является применение логистической регрессии с предварительной группировкой переменных. В результате этого преобразования сырое значение атрибута заменяется на некоторую статистику по группе (bad rate либо weight of evidence (WoE)) (см. Рисунок 31):

$$WoE = \left[\ln \left(\frac{\text{Относительная частота хороших}}{\text{Относительная частота плохих}} \right) \right] \cdot 100.$$

Группировка полезна в случае наличия нелинейных или даже немонотонных зависимостей между значением признака и вероятностью невозврата кредита. Признаки в обязательном порядке отфильтровываются по степени коррелированности и уровню стабильности во времени. Далее применяется обычная логистическая регрессия с одним из автоматических способов отбора значимых переменных (forward, backward, step-wise).

Для решения проблемы различающихся распределений на обучающей выборке и выборке для применения ($p(X_{train}, Y_{train}) \neq p(X_{prod}, Y_{prod})$) в кредитном скоринге используются методы **reject inference**. На Рисунке 32 показаны значения bad rate для групп некоторого атрибута. С июня 2016 стала работать скоринговая карта, использующая этот атрибут. Видно, что часть

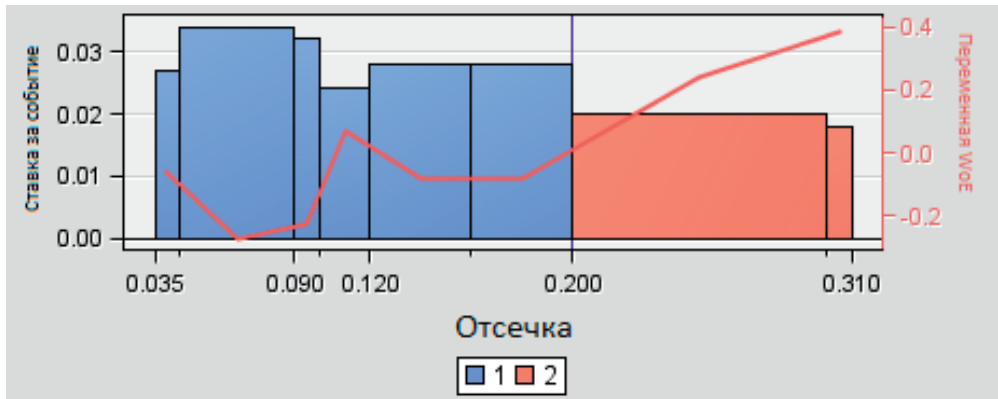


Рис. 31: Результат применения группировки

групп перестали работать: уровни bad rate для них практически равны. Методы reject inference сводятся к тому, что отказанным заявкам на кредит назначаются значения целевой переменной согласно эвристическому правилу. Значение “не вернет” может проставляться при превышении вероятности невозврата некоторой отсечки, другая эвристика добавляет отказанную заявку в обучающую выборку с обоими значениями целевой переменной и с весами, соответствующими вероятностям возврата и невозврата кредита. Обогащенная отказанными заявками выборка позволяет более объективно оценить точность модели, а иногда и повысить точность за счет использования отказов в обучении.

2.2.3 Современные тенденции кредитного скоринга

Современными тенденциями кредитного скоринга являются:

- Поведенческие данные;
- Большие выборки (а также малые и средние);
- Признаковое пространство высокой размерности, скоррелированность признаков;

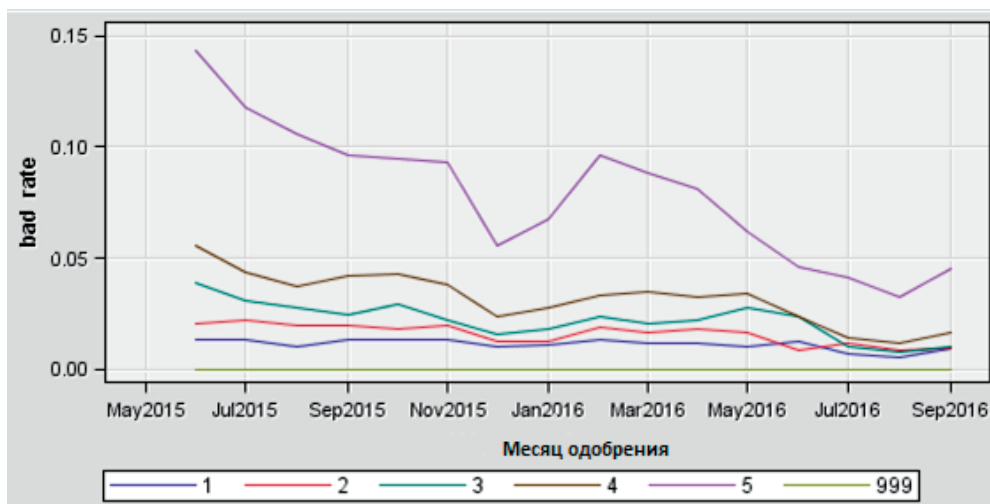


Рис. 32: Значения bad rate для групп некоторого атрибута

- Неструктурированные данные (текст, журналы);
- Данные со связями.

Для современного кредитного скоринга логистическая регрессия с группировкой плохо применима. Это происходит из-за проблем больших данных: большое число коррелированных атрибутов, большие выборки, неструктурированные данные, данные с социальными связями, пропуски в данных.

2.2.4 Современные алгоритмы машинного обучения

На помощь приходят более эффективные алгоритмы машинного обучения: линейные методы с регуляризацией, случайный лес, градиентный бустинг и глубокое обучение. На сегодня золотым стандартом построения моделей машинного обучения на незашумленных табличных данных является градиентный бустинг. Существует несколько реализаций градиентного бустинга (XGBoost, LightGBM, CatBoost), обеспечивающих высокую производительность обучения, распределенное обучение, встроенную обработку категориальных атрибутов и пропущенных значений. В связке с методами интерпре-

тации сложных моделей SHAP (SHapley Additive exPlanations позволяет объяснять вклад каждого признака в предсказание) и LIME (Local Interpretable Model-agnostic Explanations обеспечивает интерпретацию локальных решений модели) градиентный бустинг является универсальным инструментом моделирования и успешно применяется в финансовых технологиях.

Интерпретируемость моделей необходима для экспертного отбора предикторов, стабильных во времени и не зависящих от временных воздействий на клиента (промоакций, рассылок и т.д.), а также является обязательным требованием регулятивных органов (например, Центробанка).

В то же время метод случайного леса чаще применяется при работе с небольшими или зашумленными выборками. Глубокое обучение подходит для сценариев анализа текстовых и мультимедийных данных: например, записей телефонных разговоров с клиентами — анализа уровня стресса по необработанному звуку. Рекуррентные нейронные сети являются хорошим выбором для анализа поведенческих данных — последовательностей: данных бюро кредитных историй, карточных транзакций. Таким образом, можно определить особенности применения каждого типа алгоритмов.

Регуляризованные линейные методы (L1/L2 LR, SVM):

- Подходят при наличии скоррелированных признаков;
- Эффективны при разреженных входных данных (например, тексты, файлы журналов).

Случайный лес:

- Хорошо справляется с зашумленными данными;
- Подходит для задач с высокой размерностью;
- Работает при наличии коррелированных признаков.

Градиентный бустинг:

- Эффективен при больших объемах данных;

- Поддерживает высокую размерность;
- Устойчив к мультиколлинеарности признаков.

Глубокое обучение:

- Применимо к данным с локальной структурой (например, текст, поведенческие паттерны, звук, изображения);
- Подходит для анализа последовательностей (транзакции, история платежей);
- Актуально для работы с сырыми мультимедийными данными.

2.2.5 Разное покрытие источниками данных

Для эффективного создания точной предсказательной модели имеет смысл создавать модели для каждого типа данных и затем объединять их в единую модель, используя технику **blending'a** (схема техники приведена на Рисунке 33) — обучения объединяющей модели на отступах базовых моделей. В качестве объединяющей модели обычно используется простой линейный алгоритм.

Blending:

- Независимая разработка базовых моделей (различные исполнители, разные обучающие выборки);
- Удобен в операционной деятельности (источники могут быть оперативно отключены в метамоделах);
- Обработка пропусков в данных;
- Не учитываются взаимосвязи между атрибутами из разных источников.

“Left Join” и одна модель:

- Учитывают связи между атрибутами из разных источников;

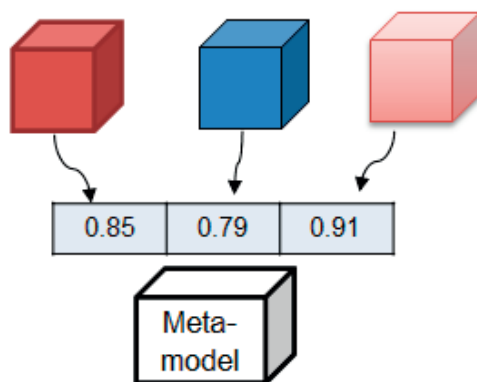


Рис. 33: Blending

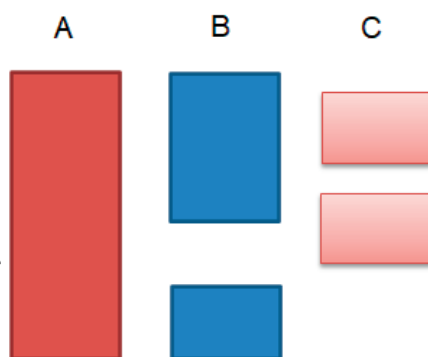


Рис. 34: “Left Join” и одна модель

- Неудобна в операционной деятельности (ситуация технического сбоя, нестабильности источника);
- Обработка пропусков в данных.

Отдельные модели:

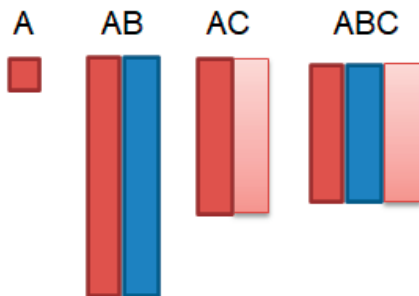


Рис. 35: Отдельные модели

- Учитывают связи между атрибутами из разных источников;
- Неудобны в операционной деятельности (множество моделей, ситуация технического сбоя, нестабильности источника);
- Обработка пропусков в данных.

2.2.6 Дополнительные вызовы в задачах кредитного скоринга

Современные задачи кредитного скоринга выходят далеко за рамки простой бинарной классификации. Помимо технических и бизнес-аспектов, связанных с качеством данных и архитектурой моделей, существует ряд теоретических и практических вызовов, с которыми сталкиваются специалисты при разработке и эксплуатации скоринговых систем:

- Решение проблемы различных распределений обучающей и целевой выборок:

- **Domain adaptation** — адаптация модели под новое распределение;
 - **Transfer learning** — перенос знаний из одной задачи/домена в другой;
 - **Active learning** — активный отбор наиболее информативных примеров для дообучения.
- Анализ социальных сетей (и сетевых баз данных). Заемщики могут быть связаны друг с другом через различные сети: семейные, рабочие, территориальные. Анализ таких сетей позволяет выявлять риски, связанные, например, с влиянием социального окружения;
 - Интеграция экспертного знания в сложные модели (например, через ручной отбор признаков, задание ограничений на поведение модели, использование гибридных моделей);
 - Учет изменений в закономерностях во время использования модели (например, через мониторинг качества предсказаний, регулярного дообучения модели, внедрение онлайн-обучения).

Таким образом, современный кредитный скоринг гораздо сложнее, чем построение бинарного классификатора для некоторой фиксированной выборки. Современные модели кредитного скоринга решают проблему reject inference, способны учитывать социальные связи заемщиков, предоставляют возможность адаптации существующей модели под изменение зависимостей во времени.

2.2.7 Неэффективное использование ресурсов в “односерверных” решениях

Когда финансовая организация сталкивается с задачами, требующими обработки больших данных, появляется потребность в более гибкой и масштабируемой инфраструктуре.

Исторически в банке существует несколько подразделений, отвечающих за свои бизнес-сценарии: департамент по управлению рисками, отдел по работе с клиентами (customer relationship management, CRM), контактный центр и другие. Каждой из таких команд выделяется собственный сервер, на котором установлен инструментарий для анализа данных, например, JupyterHub или его аналоги.

Однако такой подход имеет недостатки:

- Неравномерная нагрузка: в разные моменты времени одни серверы могут быть перегружены, а другие — простаивать. Если сотрудники временно не работают (перерыв, совещание, конец рабочего дня), то ресурсы их сервера остаются неиспользованными.
- Ограниченные возможности масштабирования: ресурсы “односерверного” решения ограничены, и при необходимости запуска тяжелых задач может просто не хватить вычислительной мощности.

Ресурсов отдельного сервера может быть недостаточно, например, когда в память загружается большая выборка для обучения модели (см. Рисунок 36). При этом может не хватить оперативной памяти или же ядер для обучения модели за разумное время.

2.2.8 “Стоимость ядер”

Один из вариантов увеличения ресурсов сервера — добавить в существующий сервер больше памяти или более мощные процессоры. Однако это не всегда возможно: слоты памяти могут быть уже заняты, более мощный процессор может быть несовместим с материнской платой. Также следует отметить, что стоимость сервера не зависит линейно от его мощности. За вдвое более мощный сервер придется заплатить в 3–4 раза большую сумму (см. Рисунок 37).

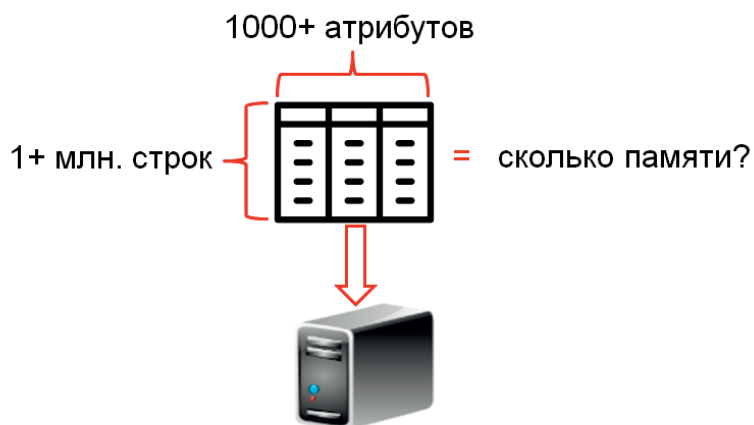


Рис. 36: Большая выборка



Рис. 37: Увеличение стоимости ресурсов

2.2.9 Регулярное применение моделей к “базовым клиентам”

Обучение моделей на больших обучающих выборках — не единственный пример проблемы больших данных при применении методов машинного обучения в бизнес-задачах.

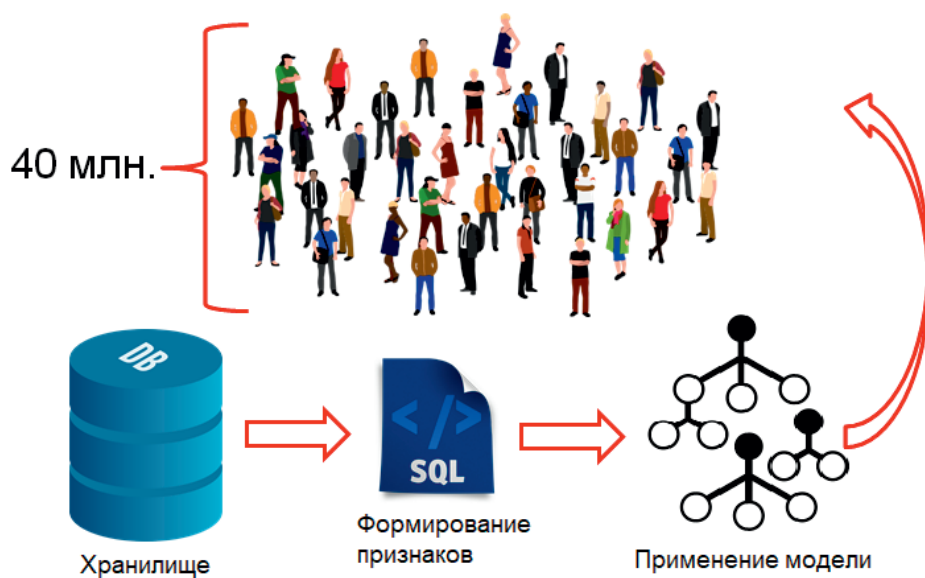


Рис. 38: Применение моделей к “базовым клиентам”

Подразделение CRM строит множество моделей, прогнозирующих приобретение клиентом различных продуктов. Эти модели на регулярной основе (например, ежедневно) должны применяться ко всей клиентской базе. С клиентами, обладающими наибольшей склонностью к приобретению продукта, осуществляется коммуникация. Следовательно, на ежедневной основе необходимо производить пересчет признаков и применение моделей CRM для всей клиентской базы организации, которая может содержать десятки миллионов людей (схематично это изображено на Рисунке 38).

2.2.10 Анализ журналов мобильных приложений и онлайн-площадок банка

Еще один пример актуальности больших данных — построение моделей на основе журналов онлайн-площадок и мобильных приложений организации. Финансовым компаниям выгодно подробно журналировать поведение клиентов на своих Интернет-ресурсах. Даже небольшие детали в поведении могут значительно улучшить точность прогнозов. По этой причине журналы могут быть достаточно тяжеловесными, накапливая несколько ТБ данных в месяц. На основе журналов необходимо формировать признаки и обучать модели машинного обучения, а также применять модели в режиме реального времени, когда клиент совершает какое-то новое действие на онлайн-ресурсе (схема приведена на Рисунке 39).

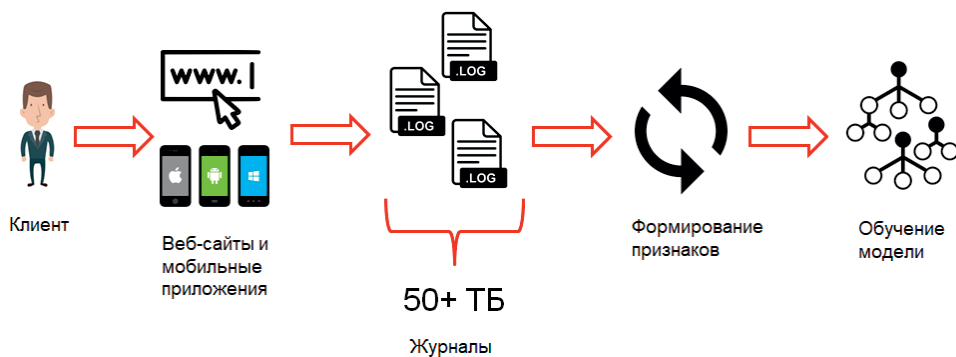


Рис. 39: Применение модели в режиме реального времени

2.3 Сценарий №3. Астрофизика

Очередной важный пример применения технологий больших данных — наблюдательная астрофизика. 400 лет назад Галилео Галилей наблюдал за звездами с помощью телескопа и записывал положение небесных объектов.

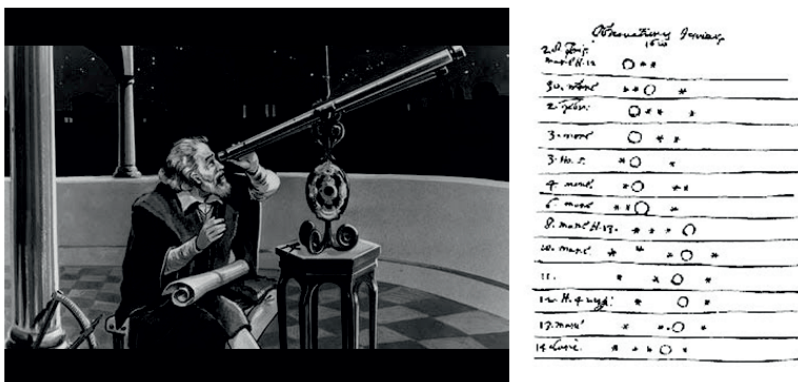
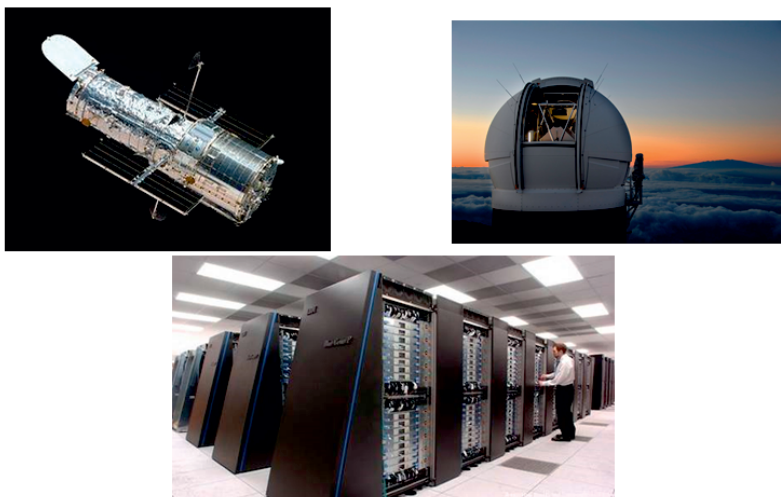
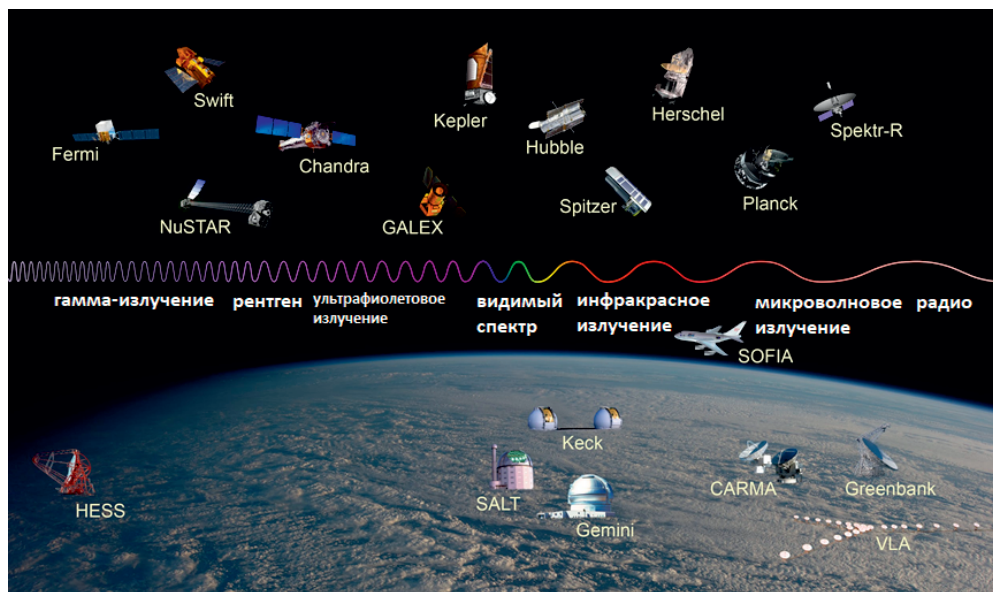


Рис. 40: Работа Галилео Галилея 400 лет назад ¹⁴

Рис. 41: Современная астрофизика ¹⁵Рис. 42: Телескопы, работающие в разных диапазонах волн ¹⁶

В результате формируются **каталоги небесных объектов** — таблицы, в которых по каждому обнаруженному на снимках объекту содержатся значения тысяч атрибутов.

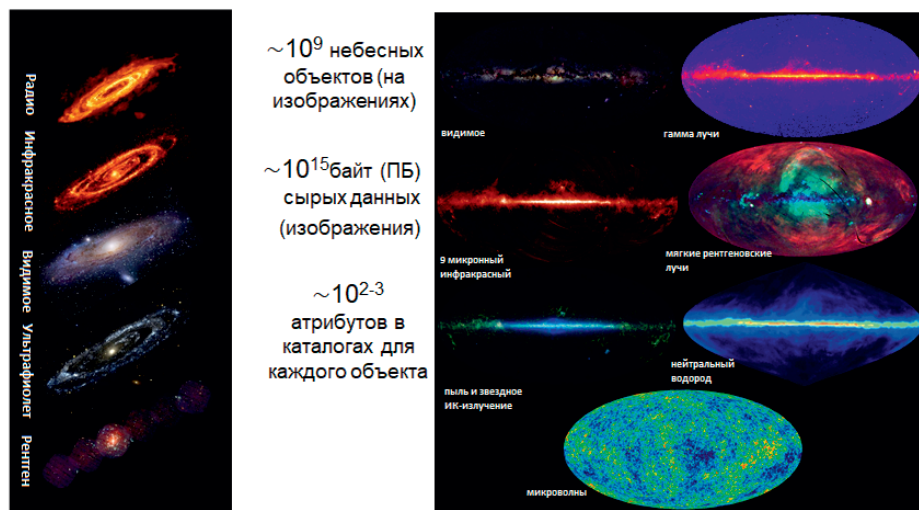


Рис. 43: Цифровые обзоры неба ¹⁷

Широкоугольный оптический телескоп фиксирует интенсивность свечения небесных объектов в нескольких интервалах длин волн — фильтрах.

2.3.2 Спектры

По небольшой выборке объектов (несколько миллионов) телескоп может собрать детальную спектральную информацию. Спектры объектов позволяют астрофизику точно определить многие характеристики небесного объекта: расстояние до него, тип объекта (квазар — Рисунок 45, звезда — Рисунок 46, галактика — Рисунок 47), массу. Спектральные выборки играют роль обучающих выборок, позволяющих тренировать модели, прогнозирующие значения перечисленных свойств. Высокоточные прогнозы по всем объектам небесной

¹⁷https://www.nasa.gov/mission_pages/GLAST/news/gammaray_best.html

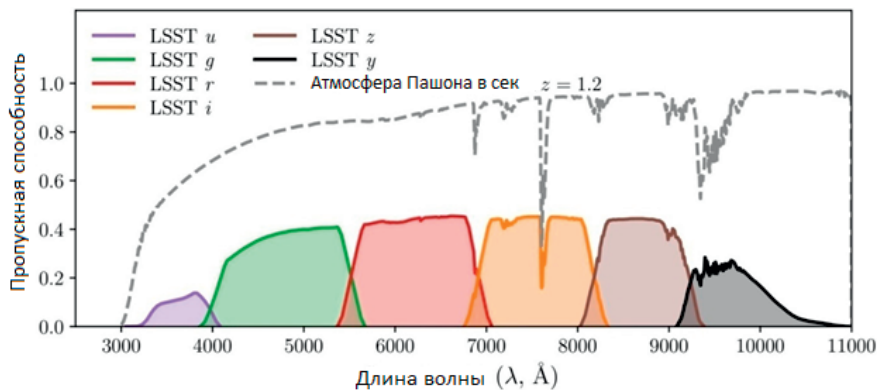
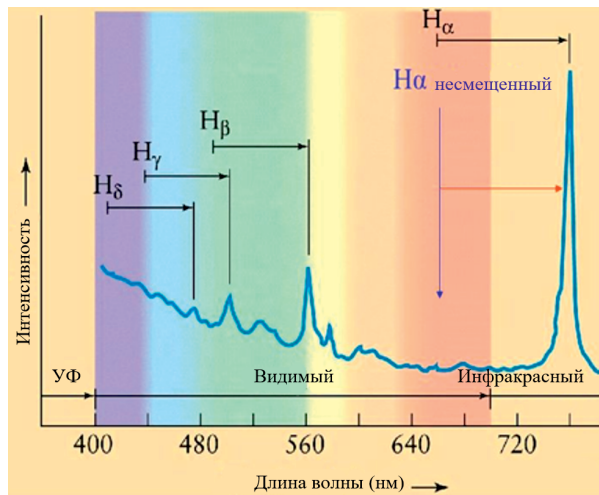


Рис. 44: Данные в фильтрах

сферы крайне важны астрофизикам для проверки теорий происхождения Вселенной, природы темной материи и других. Для $\sim 0.5\%$ (5 млн) всех известных небесных объектов собраны спектры.

Рис. 45: Квазар¹⁸

¹⁸<https://selenas.ru/space-objects/black-holes/kvazary-dlja-chajnikov-cto-takoe-kvazar/>

ГЛАВА 3

Google File System

В данной главе рассматриваются основы распределенной файловой системы Google File System (GFS), которая была специально разработана для хранения и обработки огромных объемов данных.

3.1 Проблема чтения больших данных

Представьте, что при решении прикладной задачи необходимо прочитать содержимое одного жесткого диска объемом 1 ТБ. При скорости последовательного чтения 100 МБ/сек на это потребуется:

$$\frac{1024 \cdot 1024 \text{ МБ}}{100 \text{ МБ/сек}} \approx 10000 \text{ сек} \approx 3 \text{ часа.}$$

Важно отметить, что время покрывает только чтение данных — без анализа, обработки или передачи. Такая задержка — одна из ключевых проблем при работе с большими данными.

Способы решения проблемы

Один из самых простых способов ускорения — **параллелизация чтения данных**. Для этого требуется: (1) разделить большой файл на части; (2) хранить эти части на разных дисках; (3) читать их параллельно. При таком подходе общая скорость чтения теоретически должна увеличиваться пропорционально количеству используемых дисков (см. Рисунок 48).

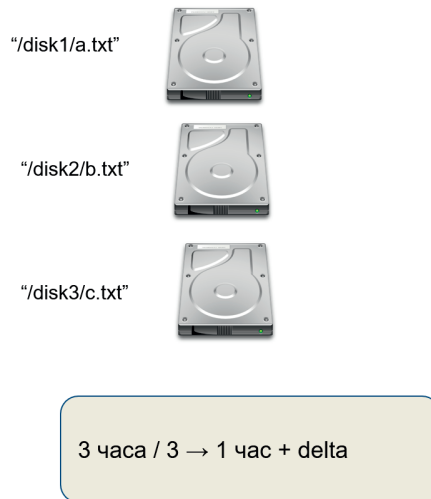


Рис. 48: Пример распределенной обработки файлов

Соответствующая программа будет выглядеть следующим образом:

```
#define BUF_SIZE 1024

double
find_min(const char *filename)
{
    double arr[BUF_SIZE];
    int f = open(filename, O_RDONLY);
    size_t n;
    double min = DBL_MAX;
    int tmp;
    while((n = read(f, arr, BUF_SIZE * sizeof(double))) > 0) {
        tmp = n / sizeof(double);
        for(int i = 0; i < tmp; i++){
            if(arr[i] < min){
```

```
        min = arr[i];
    }
}
return min;
}

int
main(void)
{
    char *filename[] = {"/disk1/a.bin", "/disk2/b.bin",
                        "/disk3/c.bin"};

    int fd[2];
    pipe(fd);
    double min = DBL_MAX;
    int i;
    for(i = 0; i < 3; i++){
        if(fork() == 0){
            break;
        }
    }
    if(i == 3){
        close(fd[1]);
        double total_min = min;
        for(int i = 0; i < 3; i++){
            read(fd[0], &min, sizeof(double));
            if(min < total_min) {
                total_min = min;
            }
        }
    }
}
```

```

    printf("%f\n", total_min);
    return 0;
}
close(fd[0]);
min = find_min(filename[i]);
write(fd[1], &min, sizeof(double));
return 0;
}

```

Помимо параллельного чтения из разных файлов с разных дисков, можно применить специальную технологию хранения данных — **RAID 0**. В этой конфигурации данные автоматически разбиваются на блоки, которые чередуются и записываются на несколько жестких дисков одновременно (см. Рисунок 49). Такой подход позволяет получить аналогичное ускорение, как и при ручной реализации параллельного чтения. При этом не требуется программировать обработку данных на нескольких устройствах — система делает это автоматически.

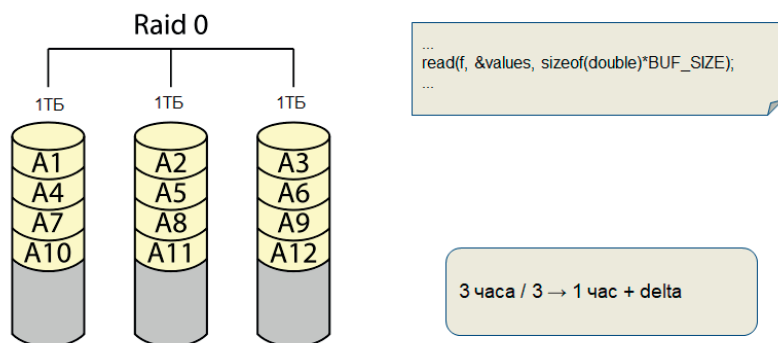


Рис. 49: Применение RAID 0 ¹⁹

¹⁹https://wiki2.org/en/File:RAID_10_6Drives_svg

Перечисленные решения имеют ряд недостатков:

- Ограниченное масштабирование: количество дисков в одном сервере физически ограничено;
- Надежность: для защиты от потери данных часто используют дублирование (резервные копии или RAID с избыточностью), но это снижает эффективность использования ресурсов. Чем больше дублируем — тем меньше данных можем хранить;
- Отсутствие гибкости: RAID работает в пределах одного сервера. В сценариях с действительно большими данными этого может быть недостаточно.

3.2 Масштабирование

Масштабирование — это свойство системы эффективно справляться с увеличением рабочей нагрузки при добавлении ресурсов. Другими словами, при увеличении объема данных или числа запросов система должна сохранять приемлемую скорость отклика и обработки, если ей добавить дополнительные аппаратные мощности.

Существует два основных типа масштабирования (см. Рисунок 50). При **вертикальном масштабировании** увеличиваются мощности одного сервера (например, установка большего объема оперативной памяти или более мощного процессора). При **горизонтальном масштабировании** добавляются новые серверы (или узлы) в систему, чтобы выполнять задачи параллельно.

Для работы с большими объемами данных важно использовать именно горизонтальное масштабирование. В таком случае можно параллельно выполнять операции чтения: когда данные хранятся на множестве дисков, расположенных на разных узлах кластера, их можно читать одновременно с



Вертикальное масштабирование



Горизонтальное масштабирование

Рис. 50: Масштабирование

нескольких машин. Это позволяет значительно ускорить обработку данных без перегрузки отдельных серверов.

3.3 Технологии организации параллельной обработки данных

Чтобы реализовать параллельную обработку данных, можно использовать низкоуровневые механизмы (например, POSIX-сокеты) или высокоуровневые решения (например, MPI, Message Passing Interface). Однако эти подходы требуют значительных усилий по разработке.

На практике хочется иметь горизонтально масштабируемый фреймворк для решения пакетных вычислительных задач широкого класса на больших массивах входных данных, обладающий устойчивостью к аппаратным сбоям. Для его работы необходимо иметь готовую горизонтально масштабируемую технологию параллельного чтения файлов, обеспечивающую их надежное хранение.

Именно такую систему разработала компания Google в 2003 году — это распределенная файловая система **Google File System (GFS)**. GFS позволила Google эффективно обрабатывать колоссальные объемы данных, автоматически распределяя хранение и обработку между тысячами серверов. Позднее на базе GFS была построена и модель параллельных вычислений MapReduce, о которой мы подробно поговорим в последующих главах.

GFS и MapReduce были представлены в работах:

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung “The Google File System” [5]²⁰.
- Jeffrey Dean and Sanjay Ghemawat “MapReduce: Simplified Data Processing on Large Clusters”, 2004 [6]²¹.

3.4 The Google File System

Концептуальные требования, которым отвечает GFS:

- Возможность надежной работы файловой системы на очень ненадежном дешевом оборудовании:
 - “Component failures are the norm rather than the exception.” (Отказы компонентов являются скорее нормой, чем исключением);
 - “We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies.” (Мы сталкивались с проблемами, вызванными ошибками приложений, ошибками операционной системы, человеческими ошибками, отказами дисков, памяти, разъемов, сетей и источников питания);
- Эффективная работа с небольшим числом больших файлов:

²⁰<http://research.google.com/archive/gfs.html>

²¹<http://research.google.com/archive/mapreduce.html>

- “Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it.” (Часто встречаются файлы размером несколько гигабайт. Каждый файл обычно содержит множество объектов приложения (например, веб-документы). Когда мы регулярно работаем с быстро растущими наборами данных объемом много ТБ, содержащими миллиарды объектов, управлять миллиардами файлов размером примерно в КБ становится непосильной задачей, даже если файловая система может это поддерживать);
- Основные операции: параллельная запись в конец файла (append) и последовательное чтение (producer-consumer). Чтение и запись могут выполняться параллельно. Запись и чтение по произвольному смещению выполняются редко;
- Высокая пропускная способность вместо быстрого отклика.

3.5 GFS: архитектура

Компоненты GFS находятся на **узлах кластера** — серверах, соединенных локальной сетью. Компонентами являются: **клиенты** — части программной библиотеки, предоставляющие приложениям API GFS, **master** — компонента, хранящая метаинформацию о каждом файле (права, места расположения блоков), иерархию каталогов, **chunkserver’ы** — хранилища блоков файла. Блок файла в оригинальной статье называется **chunk**. Размер chunk’а велик по сравнению с размером блока в обычной файловой системе (ФС) и составляет 64 МБ. Chunk’и хранятся на **chunk-серверах** в виде обычных файлов ФС Linux. При обращении к файлу на чтение или запись клиент

сначала получает от master'а информацию о chunkserver'ах, на которых находятся нужные блоки файла, после чего напрямую читает данные с соответствующих chunk-серверов. **Chunk handle** — это идентификатор chunk'а. Блоки файла не кэшируются на клиенте (однако кэшируется информация о местоположении блоков). При чтении chunk'ов неявно используется механизм буферов ФС Linux, поскольку chunk'и — это обычные файлы в локальной ФС. Взаимодействие компонент GFS осуществляется в режиме **heartBeat-сообщений**, пересылаемых между master'ом и chunk-серверами с заданной периодичностью. Схема архитектуры приведена на Рисунке 51.

Зафиксируем основные **компоненты**: Client (1..*); Master (1); Chunkservers (1..*). А также **ключевые термины**: Chunk (обычно 64 МБ); Chunk handle.

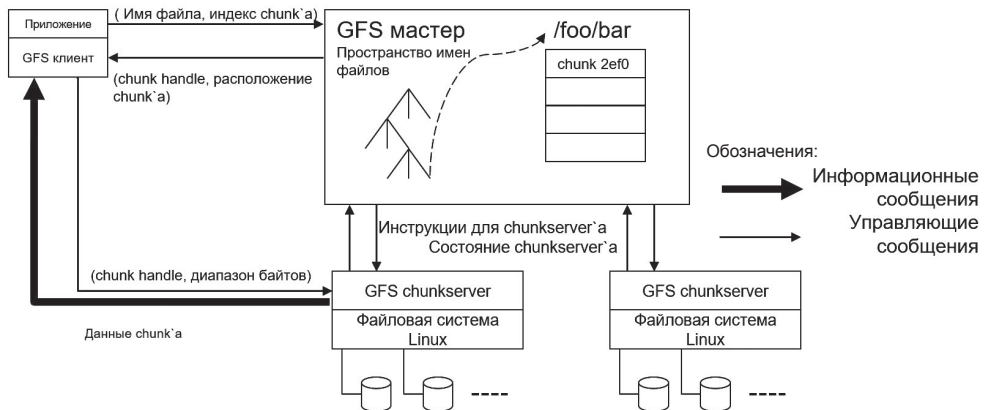


Рис. 51: Архитектура Google File System [5]

3.6 Концептуальная архитектура

Для понимания принципов работы GFS важно рассмотреть ее ключевые архитектурные концепции. Далее будут приведены основные компоненты и особенности этой распределенной файловой системы:

- Организация хранения блоков;
- Организация структур данных на сервере управления (master)
- Дополнительные операции (например, Snapshot)
- “Слабая” модель консистентности данных;
- Механизмы восстановления после сбоев.

3.6.1 Блоки (chunks)

- Размер блока — большой (64 МБ):
 - Клиент способен заэшировать местоположение chunk’ов файла (поскольку их немного), снижается нагрузка на master;
 - Уменьшается размер метаданных на master, ее можно хранить в оперативной памяти;
- Блоки хранятся на chunkservers как файлы ФС Linux:
 - Данные chunk на chunkserver могут занимать меньше блока (оптимизация внутренней фрагментации);
 - Целостность данных контролируется контрольными суммами (проверяются при передаче блока chunkserver’ом);
 - Неявно используется буферизация ФС Linux на chunkserver;
- Блоки реплицируются для обеспечения надежного хранения. По умолчанию — три реплики каждого блока.

3.6.2 Master

- Хранятся в **оперативной памяти**: (1) Имена файлов; (2) Файл→блоки (соответствие файлов и блоков); (3) Права доступа; (4) Текущее местоположение блоков (<64 бит);

- Логируются в **журнал операций** (operation log, write-ahead-log; периодически для журнала создаются контрольные точки — **checkpoints**):
(1) Имена файлов; (2) Файл→блоки (соответствие файлов и блоков);
(3) Права доступа;
- Текущее местоположение chunk'ов запрашивается у chunkservers: (1) При подключении chunkserver; (2) Периодически в heartBeat.

Для обеспечения высокой производительности вся необходимая информация для работы master-сервера хранится в оперативной памяти. Это иерархия имен, соответствие файлов и блоков, права доступа, местоположение блоков. Важно отметить: все, кроме текущего местоположения блоков, дополнительно записывается в специальный журнал файловой системы (**лог**). Это делается для того, чтобы можно было восстановить содержимое оперативной памяти master-сервера в случае сбоя.

Когда chunk-сервер запускается, он сообщает master'у, какие блоки у него хранятся. Это позволяет master-серверу получить актуальную картину распределения данных. Кроме того, журнал регулярно “проигрывается” в фоновом режиме — на его основе создаются снимки состояния, которые отражают текущее содержимое структур master-сервера. Благодаря этому, в случае сбоя для восстановления требуется прочитать только небольшой фрагмент последнего журнала, а не весь журнал с момента запуска системы. Это значительно ускоряет перезапуск и восстановление работы.

Также, в фоновом режиме master осуществляет ряд важных активностей: сборку мусора — удаление блоков файлов, помеченных на master как удаленные, ререпликацию блоков chunk-сервера при выходе его из строя и миграцию реплик для обеспечения равномерного распределения копий блоков по chunk-серверам.

3.6.3 Модель консистентности

GFS поддерживает две операции модификации данных: добавление в конец файла (append) и запись по произвольному смещению. При осуществлении этих операций содержимое реплик файла может находиться в одном из трех состояний: **консистентен** (содержимое всех реплик файла одинаково), **определен** (консистентность + содержимое файла соответствует последней модификации (нет чередования фрагментов двух модификаций)), **неконсистентен**. Примеры представлены на Рисунке 52. Ситуации, которые приводят к этим состояниям, указаны в Таблице 9.

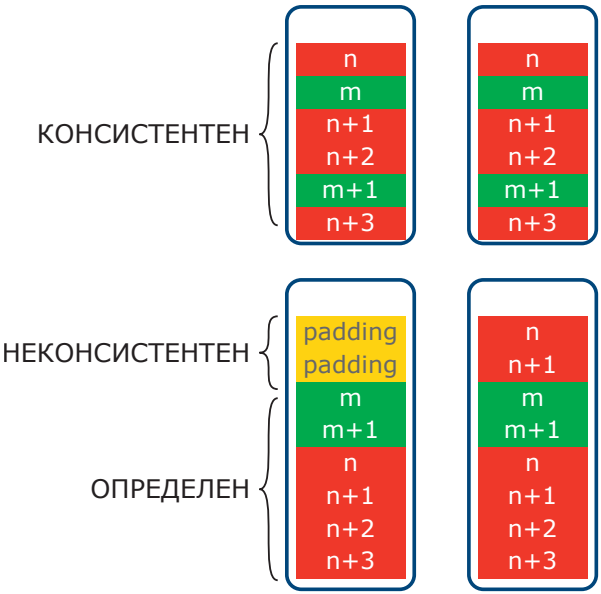


Рис. 52: Консистентность, определенность, неконсистентность

3.6.4 Аренда и порядок модификаций

В распределенной системе важно, чтобы все копии (**реплики**) файла оставались согласованными при одновременной записи. Для этого используется механизм аренды (lease). **Аренда** — это временное разрешение, выда-

Таблица 9: Состояние модифицируемого участка файла

	Write	Record append
Последовательно и успешно	определен	определен или неконсистентен
Параллельно и успешно	консистентен, но не определен	
Неуспешно	неконсистентен	

ваемое узлу (арендатору), которое дает права на доступ к ресурсу (например, к блоку данных) на определенное время. Если клиент хочет изменить данные, в дело вступает **primary-реплика** — основная копия блока. Именно она определяет порядок модификаций, координирует процесс записи среди всех остальных реплик и подтверждает клиенту успешное завершение операции. Таким образом, primary-реплика действует как менеджер записи, обеспечивая целостность и согласованность всех копий данных в системе.

Порядок модификаций:

1. Клиент узнает у master, какая реплика обладает арендой данного блока. Если аренда свободна, master выбирает реплику, которой передается аренда (primary-реплика);
2. Master передает клиенту идентификаторы и местоположения primary и **secondary-реплик**. Клиент запоминает их и больше не обращается к master'у;
3. Клиент пересылает данные модификации блока всем репликам (в режиме конвейера). Клиенты не применяют модификации, а запоминают их в LRU (least recently used)-буфере;
4. После получения подтверждений записи модификаций в буферы от всех chunkserver'ов клиент отправляет запрос на применение модификаций primary-реплике. Primary-реплика запоминает порядок запросов на

модификацию, пришедших возможно от нескольких клиентов. Локально выполняет модификации в соответствии с этим порядком;

5. Primary-реплика перенаправляет secondaries запрос на модификацию и порядок модификации;
6. Secondaries выполняют модификации и отсылают подтверждение primary;
7. Primary-реплика оповещает клиента о завершении записи. Клиент оповещается об ошибке записи любой реплики. В этом случае клиент повторяет попытку записи блока.

Этот алгоритм проиллюстрирован на Рисунках 53 и 54. Важно упомянуть несколько замечаний. Во-первых, при создании snapshot аренда может быть отозвана. Во-вторых, если владелец аренды недоступен, через 60 секунд можно делегировать аренду другому chunkserver.

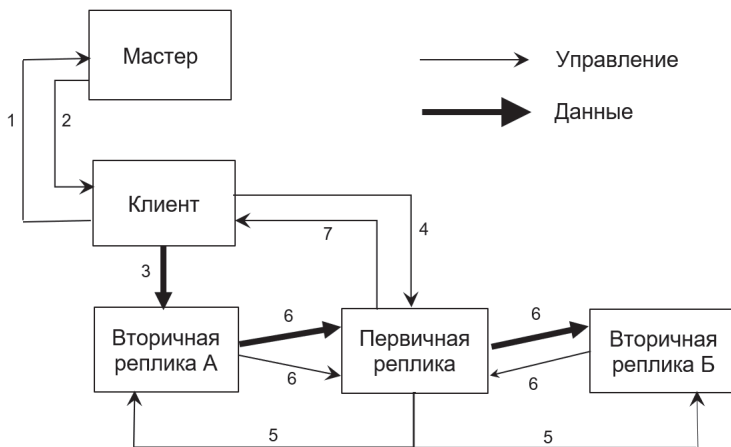


Рис. 53: Схематичный порядок модификаций [5]

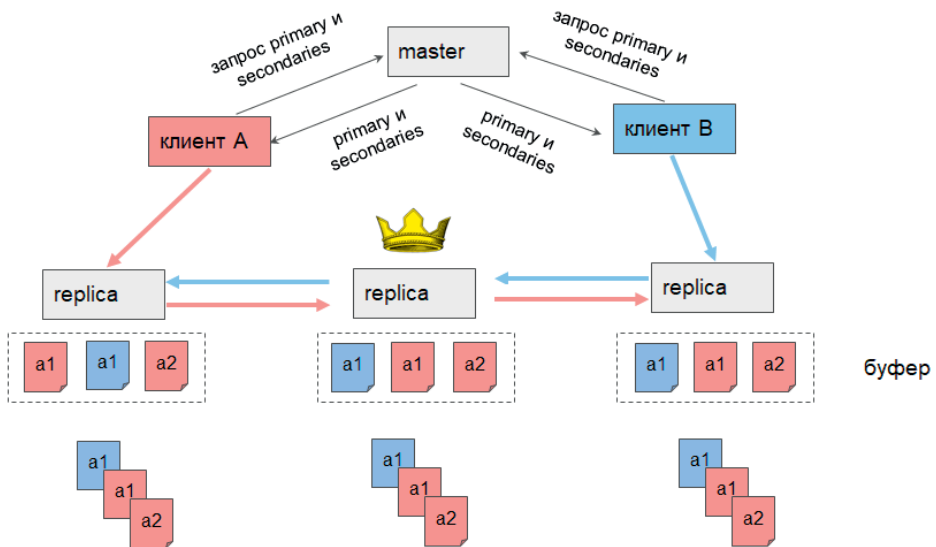


Рис. 54: Иллюстрация алгоритма порядка модификаций

3.6.5 Параллельный record append

Рассмотрим особенности параллельного добавления данных (record append) в GFS. Эта операция часто используется при ведении логов или когда несколько клиентов одновременно добавляют данные в один файл.

Основные особенности:

- При выполнении append клиент не указывает конкретную позицию в файле (система выбирает самостоятельно);
- Максимальный размер данных для одной операции append — не более четверти блока;
- GFS гарантирует, что данные будут добавлены атомарно хотя бы один раз (формально — at least once atomically).

Дополнительные детали:

- Primary-реплика сначала проверяет, уместятся ли новые данные в текущий блок. Если нет, то она дописывает в конец блока “заполнитель” (**padding**) и сообщает вторичным репликам (secondary) сделать то же самое. В результате в файле могут появиться пропуски. А запись фрагмента данных произойдет уже в следующем блоке.
- Если в процессе записи происходит ошибка хотя бы на одной из реплик, клиент повторяет операцию `append`. Это может привести к появлению дублей и неконсистентным фрагментам (система гарантирует по крайней мере одну успешную запись).

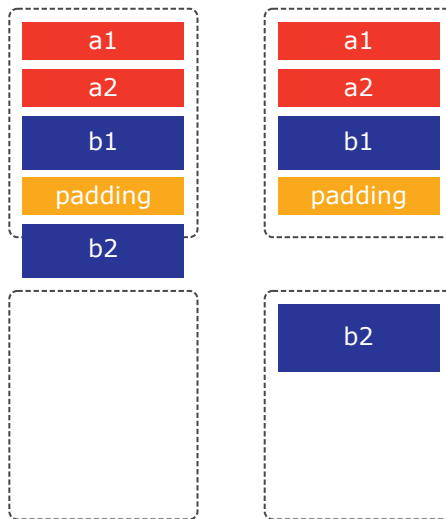


Рис. 55: Padding

Важно отметить, что из-за слабой модели консистентности на приложения ложатся обязанности по использованию идентификаторов и контрольных сумм записей, а также обеспечению `record append` вместо `write`.

3.6.6 Снимки файловой системы (Snapshots)

Еще одной функцией GFS является создание копии фрагмента ФС. **Snapshot** обеспечивает возможность создания точной копии содержимого

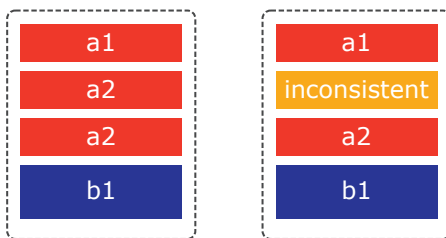


Рис. 56: Возможные дубли и неконсистентные фрагменты

любой папки. При копировании блокируется модифицирование папки. Копирование производится по принципу *copy-on-write*, то есть реальное дублирование блока происходит лишь в момент его модификации.

Пример:

Snapshot : */home/user* $\longrightarrow$ */save/user*

Чтобы гарантировать целостность копии во время создания, GFS использует блокировку на чтение (*read-lock*) (накладывается на родительские каталоги: */home* и */save*) и блокировку на запись (*write-lock*) (на сами копируемые каталоги: */home/user* и */save/user*). Такой подход предотвращает ситуацию, если кто-то попытается создать файл */home/user/foo* в момент создания *snapshot*.

3.7 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

- Каковы изначальные требования к GFS?
- Какие компоненты входят в архитектуру GFS?
- Какова модель консистентности при параллельных записи и добавлении в файл?

ГЛАВА 4

Модель вычислений MapReduce

В данной главе рассматривается модель вычислений MapReduce, предназначенная для эффективной обработки больших объемов данных в распределенной среде. Фреймворк MapReduce реализует масштабируемую обработку с автоматическим управлением вычислительными ресурсами, восстановлением после сбоев и учетом локальности данных.

4.1 Принцип работы MapReduce

В 2004 году Google представила свою модель распределенных вычислений **MapReduce**. Модель была распространена Google в виде фреймворка, в рамках которого программисту необходимо лишь реализовать функции `map` и `reduce` со следующими прототипами:

$$\text{map} : (k1, v1) \rightarrow \text{list}(k2, v2);$$

$$\text{reduce} : (k2, \text{list}(v2)) \rightarrow \text{list}(v3).$$

Программы, написанные в таком функциональном стиле, могут быть автоматически распараллелены и исполнены на больших кластерах. Фреймворк автоматически выполняет распределенную на кластере обработку исходных данных, находящихся в Google File System, согласно предложенным реализациям функций `map` и `reduce`.

4.2 Пример распределенных вычислений

Представим, что начальник тюрьмы приказал своим охранникам подсчитать встречаемость слов в книгах своей домашней библиотеки. Охранники решили привлечь к этой задаче заключенных, находящихся на первом этаже тюрьмы.

После завтрака каждый заключенный увидел у себя в камере стопку книг и тетрадку для записи статистики встречаемости слов. Таким образом, вся библиотека была разделена на части, и каждый заключенный занялся своей частью библиотеки. Заключенный просматривал текст книги и формировал в тетрадке запись “слово, 1”. Это стадия **Map** модели **MapReduce**, а каждый заключенный — так называемый **mapper** (см. Рисунок 57).

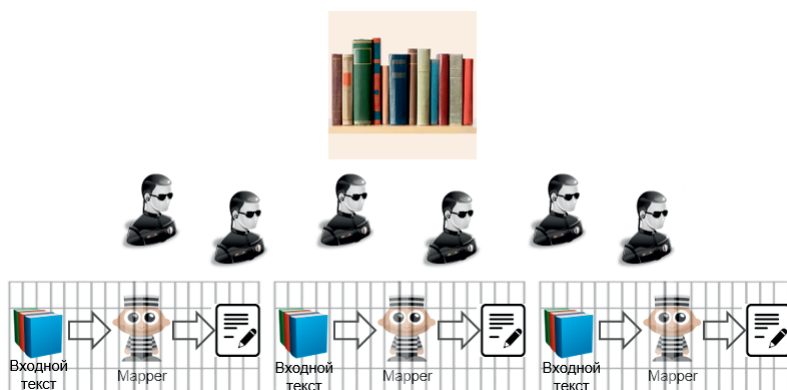


Рис. 57: Визуализация этапа сбора статистики слов в тексте

Охранники забирают тетрадь каждого заключенного, сортируют ее слова по алфавиту и делят каждую тетрадь с отсортированными данными на три тетради по диапазонам первых букв в словах (см. Рисунок 58).

Все тетради со словами из одного диапазона букв объединяются охранниками в одну тетрадь с отсортированными словами, которая отправляется отдельному заключенному на втором этаже, отвечающему за соответствующий диапазон букв (см. Рисунок 59). Его задачей является суммирование единиц

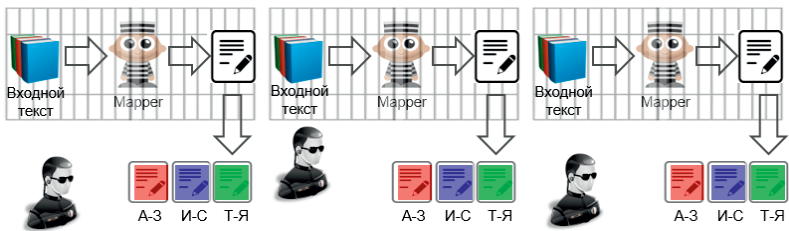


Рис. 58: Визуализация этапа распределения слов между заключенными 2-го этажа

для каждой группы повторяющихся слов. Выполнив свою работу, заключенные со второго этажа оставили свои тетради с финальной статистикой в камерах и ушли на прогулку. Заключенные со второго этажа — это **reducer**’ы, а та работа, которую они проделали — реализация функции **reduce**. Действия охранников соответствуют фазе **Тасовка (shuffle)**, которую выполняет фреймворк MapReduce. Камера заключенного на первом этаже — это узел кластера, на котором находится блок файла (файлом в данном случае является библиотека). А блок файла — это стопка книг, которая досталась каждому заключенному на первом этаже.

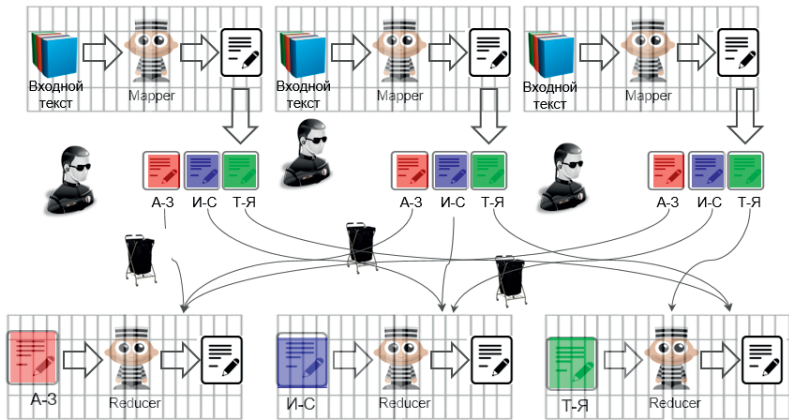


Рис. 59: Визуализация этапа вычисления частот повторения слов

4.3 Применение MapReduce в обработке большого текста

Рассмотрим более подробно процесс подсчета статистики встречаемости слов в тексте. В следующем примере `reducer`'ы получают слова, не исходя из диапазона первых букв слова, а на основе четности первой буквы.

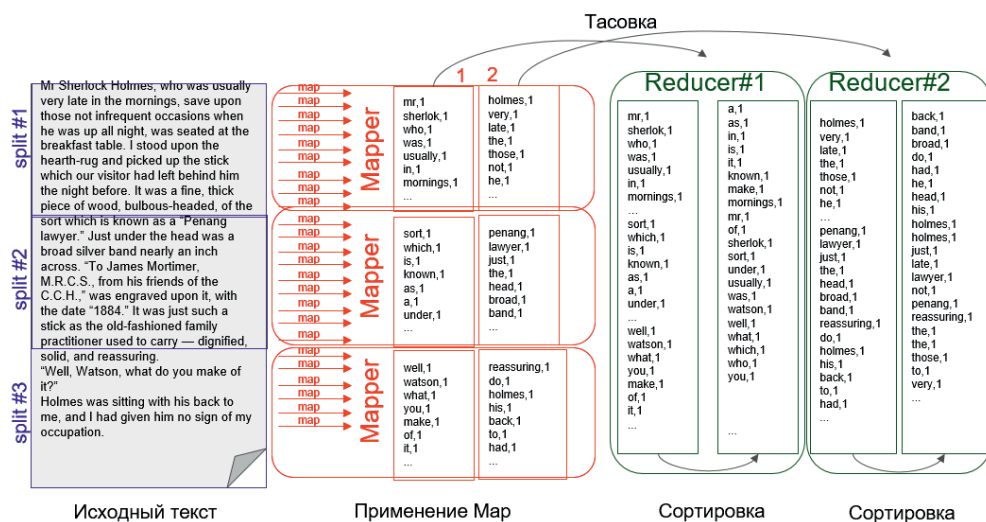


Рис. 60: Разбиение слов на два кластера с использованием функции `map` на примере анализа текста

Данный текст находится в распределенной файловой системе и состоит из блоков, также называемых **chunks** или **сплитами (splits)** (см. Рисунок 60, слева). На узлах кластера создаются компоненты **mapper** (см. Рисунок 60, по центру). Эти компоненты создаются именно на тех узлах кластера, на которых лежат блоки файла. Компоненты `mapper` вызывают функцию `map` для каждой строки файла. Функция `map` разбирает строку файла на слова и формирует пары “слово, 1”.

Каждый `mapper` записывает результат своей работы в два отдельных ло-

кальных файла. В первом файле окажутся слова, начинающиеся на четную букву, а во втором — на нечетную букву. Эти две группы файлов соответствуют двум **reducer’ам**, которые будут обрабатывать данные mapper’ов на стадии reduce. Reducer’ы, получая данные mapper’ов по локальной сети, выполняют сортировку полученных файлов (см. Рисунок 60, справа).

Далее reducer’ы вызывают функцию **reduce** для каждой пары “ключ, значение”, соответствующей одному ключу (см. Рисунок 61). Данная функция производит суммирование единиц для каждой повторяющейся группы слов (для каждого ключа). В результате в распределенной файловой системе образуется по одному локально отсортированному файлу от каждого reducer’а.

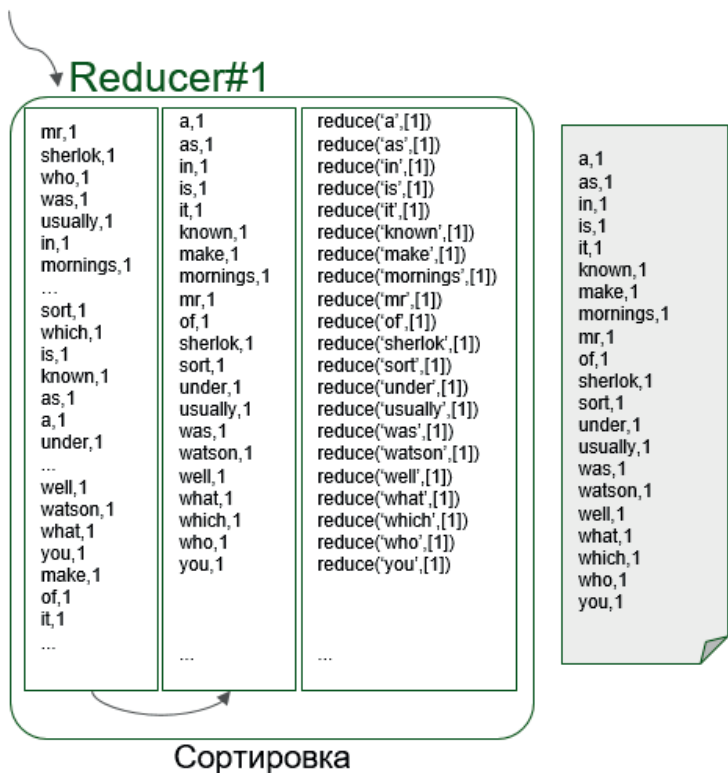


Рис. 61: Вычисление частот повторений различных слов с использованием функции **reduce** на примере анализа текста

4.4 Роль стадий в модели вычислений

4.4.1 Map

Рассмотрим более формально стадии Map и Reduce. Стадия Map соответствует применению некоторого действия (реализованного в функции `map`) к каждому элементу исходной последовательности значений (или последовательности пар “ключ, значение”, “ключ” в которых опционален). Результатом действия является или одна пара “ключ, значение” произвольного типа, или несколько пар, или ни одной пары. Возможность возвращать пустой список выходных пар “ключ, значение” позволяет осуществлять фильтрацию элементов входной последовательности по условию. Важное преимущество состоит в том, что функция `map` может вызываться параллельно для разных частей входной последовательности.

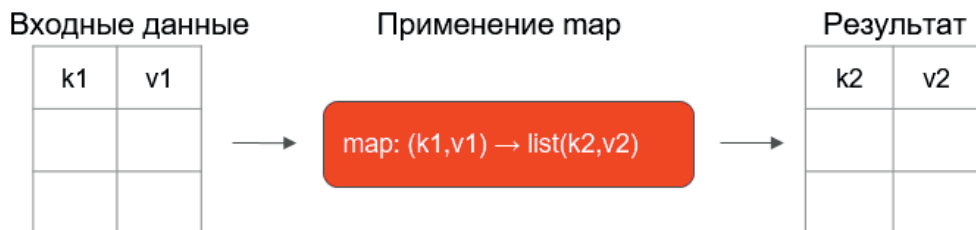


Рис. 62: Применение функции `map`

Стадия Map модели MapReduce подходит для выполнения следующих прикладных задач:

- Поиск строк файла, удовлетворяющих регулярному выражению. Например, использование команды `grep`;
- Проекция и фильтрация реляционных отношений. Например, выполнение запроса `select category from Order where sum>1000`;
- Поиск оптимальных гиперпараметров модели машинного обучения по некоторой сетке. Например, при применении кросс-валидации.

Возможны дополнительные возможности за счет использования прототипа `map`: $(k1, v1) \rightarrow list(k2, v2)$ вместо прототипа $(k1, v1) \rightarrow (k2, v2)$.

4.4.2 Reduce

На стадии `Reduce` фреймворк без участия программиста сортирует последовательность пар “ключ, значение”, полученных от стадии `Map`. Сортировка выполняется параллельно и распределенно. Далее фреймворк вызывает функцию `reduce`, реализованную программистом, для каждого уникального значения ключа и соответствующих ему значений.

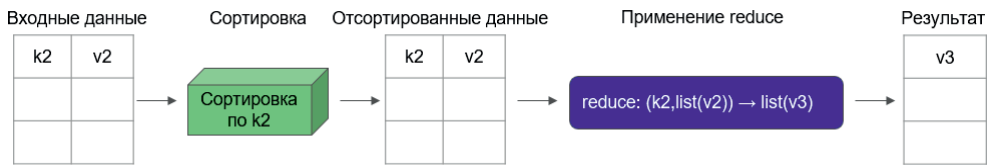


Рис. 63: Применение функции `reduce`

Таким образом, стадия `Reduce` может пригодиться хотя бы для сортировки больших массивов данных и при выполнении агрегационных SQL-запросов. Например, с помощью `Reduce` может быть реализована операция соединения отношений.

4.4.3 Применение MapReduce в обработке табличных данных

Пусть необходимо соединить значения из таблиц `a` и `b` по ключу. Сначала в значение каждой из таблиц добавим “тег” (см. Рисунок 64) — название таблицы, к которой принадлежит пара “ключ, значение”. Далее фреймворк выполнит сортировку данных по ключу (см. Рисунок 65, слева), вызовет пользовательскую функцию `reduce` (см. Рисунок 65, справа). Рассмотрим реализацию соединения `INNER JOIN` через вложенные циклы. Внешний цикл проходит по элементам с одним из тегов, внутренний — по элементам со вторым тегом. В теле внутреннего цикла генерируется пара “ключ, значение”, состоящая из текущих элементов внутреннего и внешнего циклов.

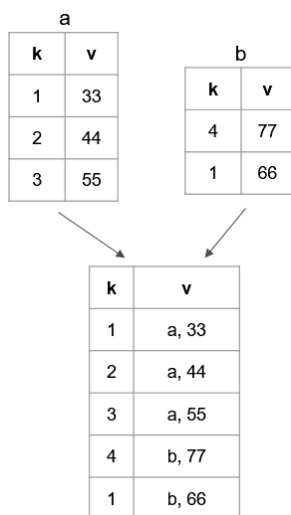


Рис. 64: Пример слияния двух таблиц в одну

Аналогичным образом можно реализовать другие разновидности соединения.

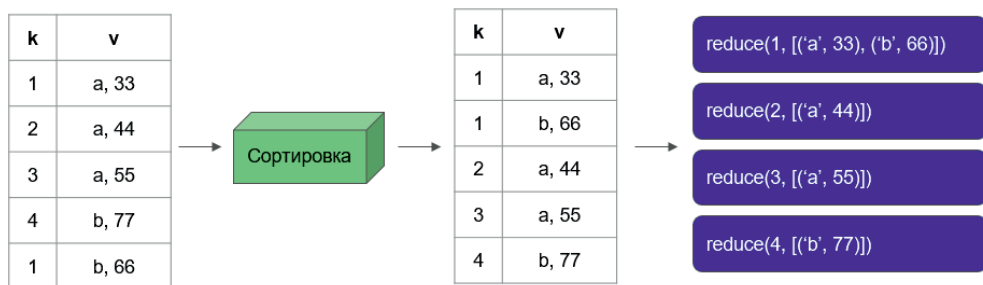


Рис. 65: Пример сортировки данных и вызова функции reduce

4.5 Технические детали работы фреймворка

Фреймворк разворачивается на том же кластере, что и Google File System. В фреймворке определена компонента **master**, которая получает от пользователя детали задания для исполнения (реализации функций **map** и **reduce**,

имена входного и выходного файлов GFS).

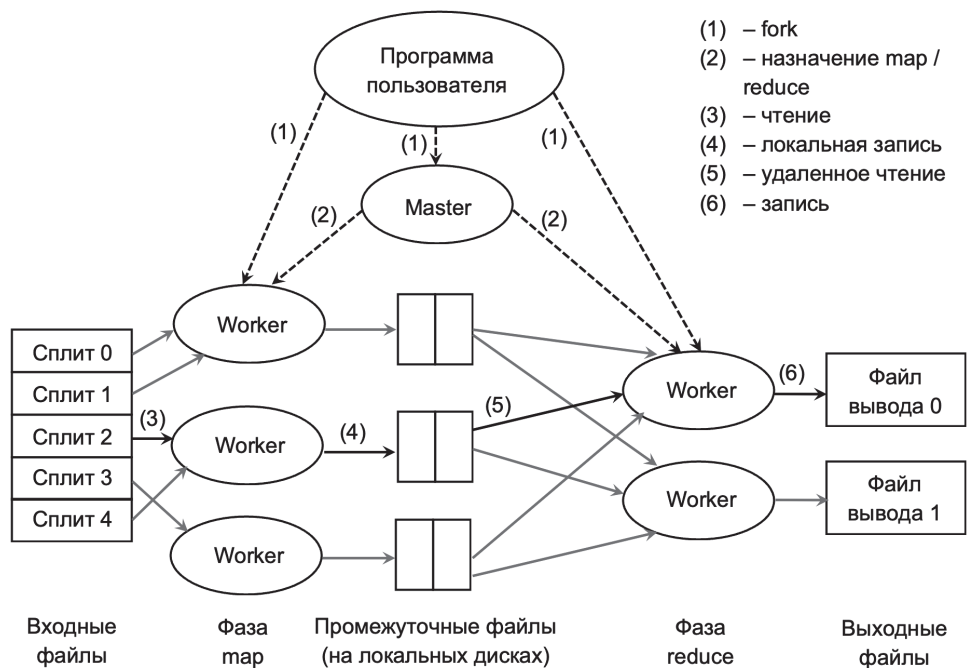


Рис. 66: Компоненты фреймворка MapReduce [6]

На остальных узлах располагаются компоненты **worker** (см. Рисунок 66). На части worker’ов инициируется расчет стадии Map, и worker соответствует mapper’у. Mapper’ы запускаются на тех узлах, где находятся блоки входного файла. Это называется **принципом локальности данных**. Каждый mapper должен обработать свой блок файла — вызвать функцию **map** и передать в нее по очереди каждую строку своего блока. Далее происходит распределенная сортировка сгенерированных mapper’ами пар “ключ-значение”. Отсортированные пары распределяются между несколькими worker’ами типа reducer, на которых и происходит выполнение функции **reduce**. Итоговые пары “ключ, значение” записываются в GFS каждым reducer’ом в отдельный файл.

Таким образом, использование фреймворка MapReduce в связке с файловой системой GFS предоставляет разработчику мощные инструменты для обработки больших данных. В частности, эта связка обеспечивает:

1. Распределенную файловую систему, позволяющую надежно хранить и обрабатывать большие объемы данных;
2. Распределенное и параллельное выполнение функции `map`, преобразующей входные пары вида $(k1, v1)$ в список промежуточных пар $list(k2, v2)$;
3. Распределенную сортировку промежуточных данных по ключам для последующей агрегации;
4. Распределенное и параллельное выполнение функции `reduce`, которая обрабатывает пары $(k2, list(v2))$ и возвращает агрегированные результаты;
5. Учет локальности данных при планировании задач `map`, что минимизирует передачу данных по сети и повышает производительность;
6. Автоматическое восстановление после сбоев, что важно в кластерах с множеством узлов;
7. Горизонтальную масштабируемость — возможность обрабатывать больше данных за счет добавления новых узлов в кластер.

4.6 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

1. Какие компоненты входят в реализацию Google MapReduce?
2. В чем суть стадий Map, сортировки и Reduce?

ГЛАВА 5

Apache Hadoop

Данная глава начинает рассказ об Apache Hadoop [7] — открытой реализации идей Google File System и Google MapReduce. В первой части рассказа будут описаны принципы работы, компоненты и особенности распределенной файловой системы HDFS.

5.1 Появление и развитие Apache Hadoop

Работа над поисковой системой Nutch и системой полнотекстового поиска Lucene привела Дуга Каттинга и Майкла Кафареллу к идее создания собственной распределенной файловой системы и фреймворка для обработки больших данных. Основой для вдохновения стали статьи компании Google, в которых описывались технологии GFS (Google File System) и MapReduce. Созданный ими проект получил название **Hadoop** — в честь плюшевого слона, любимой игрушки сына Дуга Каттинга.

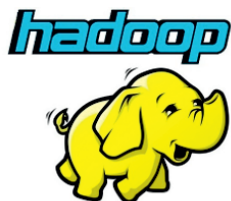


Рис. 67: Логотип Hadoop ²²

Система быстро обрела популярность: сначала внутри компании Yahoo, куда позже перешел Дуг Каттинг, а затем и в других организациях, включая

²²<https://ru.wikipedia.org/wiki/Hadoop>

Facebook, Amazon и The New York Times. Со временем Hadoop стал развиваться в рамках Apache Software Foundation — открытого сообщества, занимающегося развитием программного обеспечения с открытым исходным кодом. В дальнейшем Google передал Apache права на использование своей модели вычислений MapReduce. Через несколько лет основатели проекта перешли в компанию Cloudera, специализирующуюся на разработке собственного дистрибутива Hadoop и коммерческих решениях на его основе.



Рис. 68: Логотип The Apache Software Foundation ²³

Современный Apache Hadoop включает как минимум три ключевых компонента:

1. **HDFS** (Hadoop Distributed File System) — распределенная файловая система;
2. **MapReduce** — модель обработки данных, рассмотренная в Главе 4;
3. **YARN** (Yet Another Resource Negotiator) — система управления ресурсами и планировщик заданий в кластере, поддерживающий запуск не только MapReduce-приложений.

В версии Apache Hadoop 1.0 (2011 год) архитектура была основана на жесткой связке MapReduce и HDFS, а управление ресурсами осуществлялось внутри MapReduce. В версии 2.0 (2013 год) была введена система YARN, которая отделила управление ресурсами от MapReduce и позволила запускать на кластере другие вычислительные фреймворки. Наконец, в Hadoop 3.0 (2017 год) появилась поддержка кодирования стиранием (Erasure Coding) для экономии дискового пространства, контейнеризация приложений с помощью Docker, распределение GPU-ресурсов и улучшенная отказоустойчивость.

²³<https://www.apache.org/>

Иногда отдельным проектом также считают **Hadoop Common** — вспомогательную библиотеку для использования функционала Hadoop.

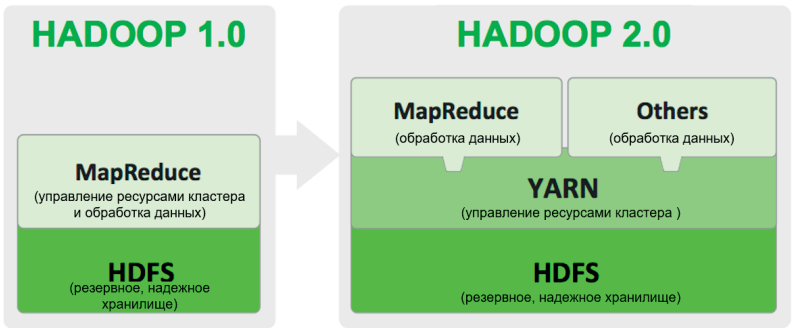


Рис. 69: Эволюция Hadoop ²⁴

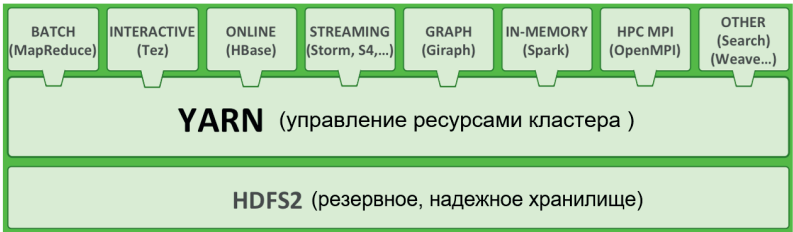


Рис. 70: Визуализация работы YARN ²⁵

5.2 Установка Hadoop

Hadoop написан на языке программирования Java. Основной интерфейс для написания программ на MapReduce — **Java API**. Также возможна реализация логики MapReduce-приложений на произвольном языке программирования (более распространены скриптовые языки) и интеграция логики во

²⁴<https://www.rittmanmead.com/blog/2015/09/managing-impala-and-other-mixed-workloads-on-the-oracle-big-data-appliance/>

²⁵<https://opensource.com/life/14/8/intro-apache-hadoop-big-data>

фреймворк с помощью потоков ввода-вывода (**Streaming**). Для написания программ на C++ Hadoop предоставляет библиотеку **Pipes**.

Hadoop можно скачать в виде оригинального дистрибутива с официального сайта hadoop.apache.org и вручную установить на узлы кластера. Другим вариантом является использование адаптированных вариантов дистрибутивов Hadoop от Cloudera, Hortonworks и MAPR. Эти дистрибутивы включают в себя сам Hadoop и разнообразное программное обеспечение из стека Hadoop, доступное через менеджер пакетов. Инсталляция производится в удобном интерактивном режиме. Разработчики данных дистрибутивов также проводят оптимизацию производительности некоторых компонентов стека Hadoop.

Однако самый простой способ работы с Hadoop — использование облачных сервисов Amazon Elastic MapReduce или HDInsight в рамках облака Microsoft Azure. Облачный кластер Hadoop можно развернуть в интерактивном режиме или через конфигурационный файл в облаке всего за несколько минут. Пользователю этих облачных сервисов доступна возможность выбрать производительность каждого узла кластера и число узлов. Тарифицируется время использования каждого узла. Стоимость кластера зависит от типов узлов и их числа.



Рис. 71: Логотипы компаний, распространяющих адаптированные варианты Hadoop

5.3 Принципы работы HDFS

Hadoop Distributed File System (HDFS) является преемником Google File System (GFS) и во многом наследует ее архитектурные принципы. Однако при разработке HDFS было решено упростить реализацию, отказавшись

от ряда возможностей, которые при обработке больших данных не были необходимы.

В частности, в HDFS нет механизма параллельного добавления данных (append) в один и тот же файл. Если несколько процессов ведут запись логов, каждый из них может создавать и заполнять свой собственный файл, а не пытаться писать в один общий. Также HDFS не поддерживает операции чтения или записи по произвольному смещению в файле. Подобные операции крайне редко используются при пакетной (batch) обработке больших данных.

HDFS основан на следующих компонентах:

- **Клиенты** — приложения, использующие методы программной библиотеки Hadoop;
- **DataNodes** — компоненты, хранящие блоки файлов (аналог chunkserver'ов GFS);
- **NameNode** (аналог master в GFS) — компонента, содержащая метаинформацию о файлах, места расположения блоков файлов, иерархию папок.

5.3.1 Сценарии чтения, записи и восстановления

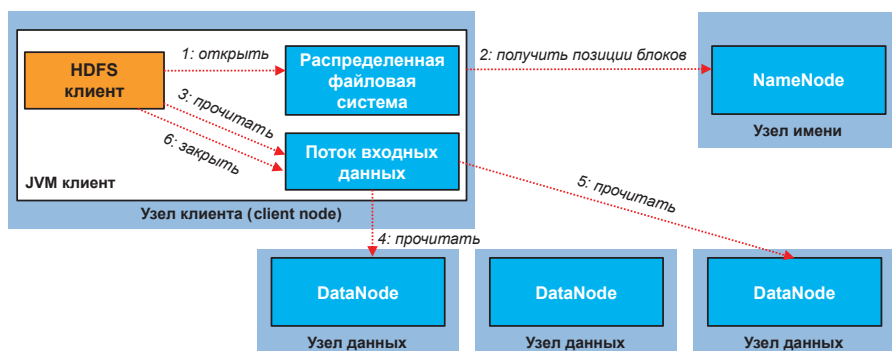


Рис. 72: Операция чтения в HDFS [1]

Сценарий чтения выглядит следующим образом: клиент получает информацию о местоположении блоков файла у NameNode, и далее напрямую читает содержимое необходимых ему блоков с соответствующих узлов кластера, где находятся DataNode'ы.

При **записи** данные разбиваются на пакеты, которые находятся на клиенте в очереди на отсылку. Пакет перемещается из очереди на отсылку в очередь на подтверждение и передается на первый DataNode, который записывает его и пересылает в конвейерном режиме оставшимся DataNode. В обратную сторону пересылается подтверждение о записи и пакет удаляется из очереди подтверждений.

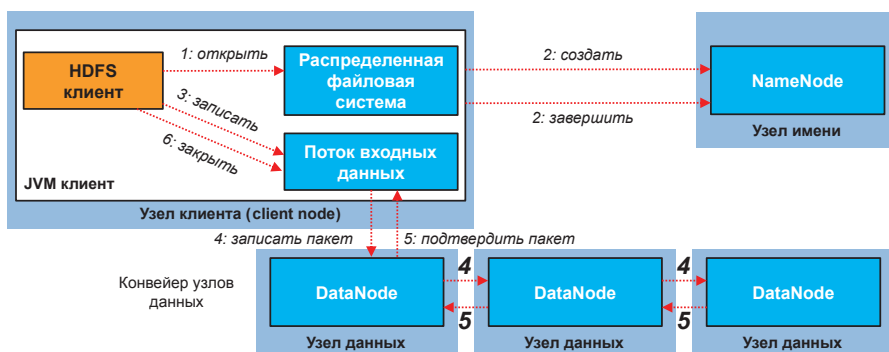


Рис. 73: Операция записи в HDFS [1]

При записи файла в HDFS система сначала синхронно сохраняет минимально допустимое количество копий (реплик), заданное параметром `dfs.replication.min`. Остальные реплики (`dfs.replication - dfs.replication.min`) создаются асинхронно. Это делается для повышения отказоустойчивости: если в процессе асинхронной репликации произойдут сбои, данные уже будут сохранены в минимально допустимом объеме. Важно отметить, что параметр `dfs.replication` задается индивидуально для каждого файла. Данные становятся гарантированно доступными для чтения только после того, как файл будет закрыт.

Компонент NameNode отвечает за управление метаданными в HDFS. Для повышения производительности и надежности данные хранятся одновременно в оперативной памяти и на диске. В оперативной памяти хранится актуальное состояние метаданных файлов (включая права на доступ, время записи и обращения к файлу, идентификаторы блоков файла и другую информацию). Также в оперативной памяти хранится размещение блоков по DataNodes. На жестком диске находится журнал изменений файловой системы (файл **edits**) и снимок состояния файловой системы (файл **fsimage**), сохраняемый периодически.

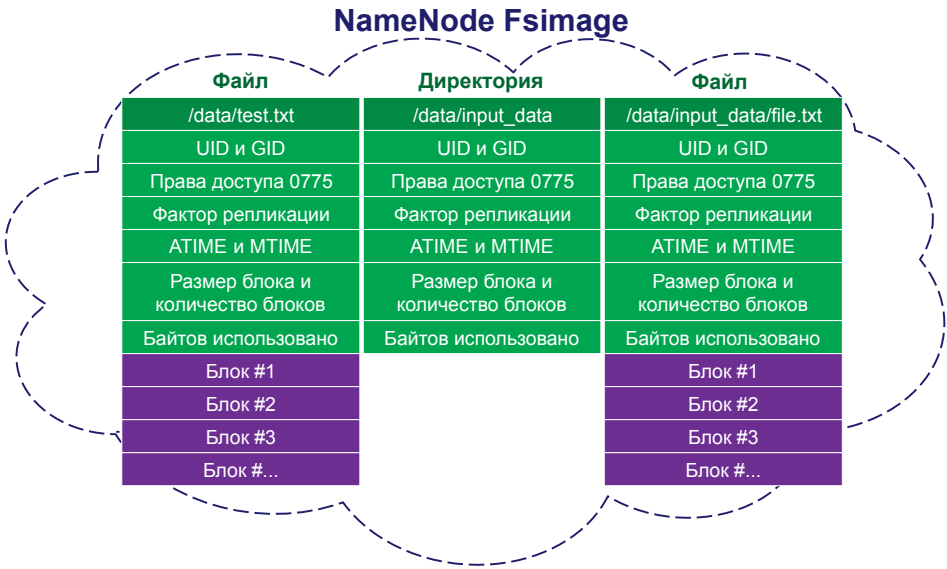


Рис. 74: Структура файла fsimage

При сбое NameNode можно восстановить актуальное состояние метаданных, применив изменения из журнала edits к последнему сохраненному снимку fsimage. В результате получается обновленная версия fsimage, которую можно использовать для загрузки метаданных в оперативную память при старте системы. При создании и перемещении файлов обновляется журнал изменений edits и метаданные, однако fsimage не изменяется.

После перезапуска NameNode не знает, где физически находятся блоки файлов. Чтобы восстановить эту информацию, каждый DataNode автоматически сообщает NameNode, какие блоки он хранит, и на основе этих сообщений восстанавливается полная карта размещения блоков по узлам. **Критически важными** для восстановления системы являются файлы edits и fsimage. Регулярно должна создаваться копия этих файлов (локально или удаленно (NFS)).

С течением времени файл edits может значительно увеличиться, что замедляет процесс восстановления. Чтобы этого избежать, используется механизм контрольных точек (**checkpoints**). Для этого в HDFS предусмотрена вспомогательная служба — **Secondary NameNode**, которая периодически:

- Запрашивает у NameNode файлы fsimage и edits;
- Объединяет их, накатывая изменения из edits на fsimage;
- Возвращает NameNode обновленный fsimage.
- Частота создания контрольных точек настраивается либо по времени (например, каждые 30 минут), либо по количеству операций в журнале edits.



Рис. 75: Конвейер механизма контрольных точек [1]

5.3.2 High Availability

Для сокращения времени восстановления HDFS после сбоя используется механизм **High Availability** (высокой доступности). Он предполагает наличие двух NameNode в кластере: **Active NameNode** — основной узел, который обслуживает запросы клиентов; **Standby NameNode** — резервный узел, готовый в любой момент принять управление. Каждая NameNode ведет собственную копию fsimage. Однако журнал изменений (edits) хранится в распределенной файловой системе и доступен обоим NameNode. Standby NameNode регулярно считывает этот журнал, чтобы поддерживать свою копию fsimage в почти актуальном состоянии. Кроме того, все DataNode сообщают информацию о хранимых блоках сразу обоим NameNode, чтобы в случае переключения не было потери информации о размещении данных.

Важно, чтобы в каждый момент времени был только один активный NameNode. Иначе возможна рассинхронизация данных и ошибки. Для координации используется специальная система — **ZooKeeper**, которая реализует механизм выбора лидера в распределенной среде. Это позволяет гарантировать, что активной станет только одна из NameNode, даже если сбой произошел одновременно у нескольких компонентов.

Таким образом, механизм High Availability заключается в быстром переключении на резервный NameNode в случае отказа основного NameNode. Основные особенности этого механизма:

- Разделение на Active/Standby;
- Общее хранилище для edits;
- Непрерывная актуализация метаданных Standby NameNode из edits;
- Пересылка информации о местоположении блоков DataNodes обоим NameNodes;
- Незапланированное/запланированное (например, при обновлении ПО NameNode) преодоление сбоя;

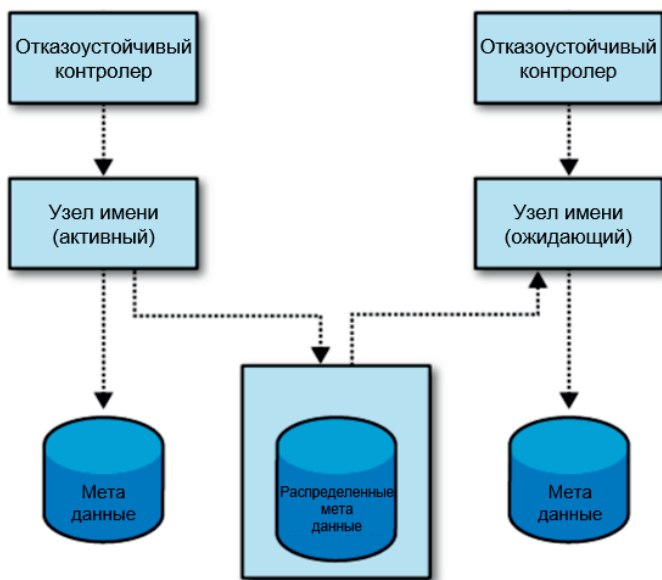


Рис. 76: Конвейер механизма High Availability [1]

- Изоляция при сбоях (обеспечивает единственность Active Node).

5.3.3 HDFS Federation

NameNode может стать бутылочным горлышком при увеличении числа файлов. Решить эту проблему помогает механизм **HDFS Federation**, который позволяет использовать несколько NameNode, привязанных к отдельным папкам в иерархии папок. Соответствия папки и NameNode хранятся в конфигурационном файле и доступны HDFS-клиентам. Привязка DataNode к конкретным NameNode отсутствует.

Таким образом, особенность механизма HDFS Federation заключается в возможности добавления NameNode, управляющих своей частью иерархии каталогов. Механизм обеспечивает независимость иерархии файлов при общем множестве DataNodes.

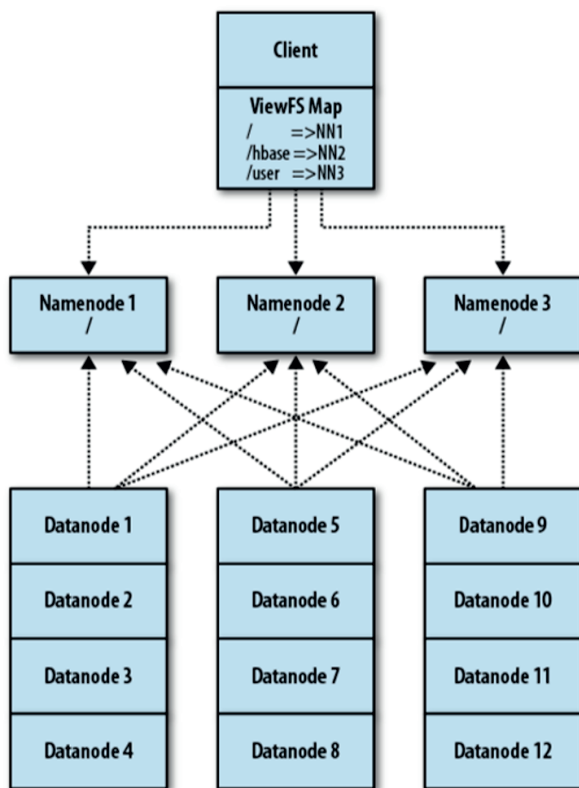


Рис. 77: Конвейер механизма HDFS Federation [1]

5.3.4 Выбор размера блока

При разработке HDFS перед инженерами стояла задача определить, какой размер блока использовать при хранении файлов в распределенной файловой системе. Хотя этот параметр можно настраивать индивидуально для каждого файла, система должна иметь разумное значение по умолчанию. Ключевым предположением стало требование: время получения метаданных от NameNode не должно превышать 1% от времени передачи самого файла клиенту. На момент проектирования среднее время взаимодействия с NameNode составляло около 10 миллисекунд. При этом считалось, что данные могут считываться с диска и передаваться по сети со скоростью примерно 100 МБ

в секунду. Следовательно, размер блока должен быть не менее 100 МБ. Разработчики приняли чуть большее значение (128 МБ), которое стало стандартным размером блока в первых версиях Hadoop.

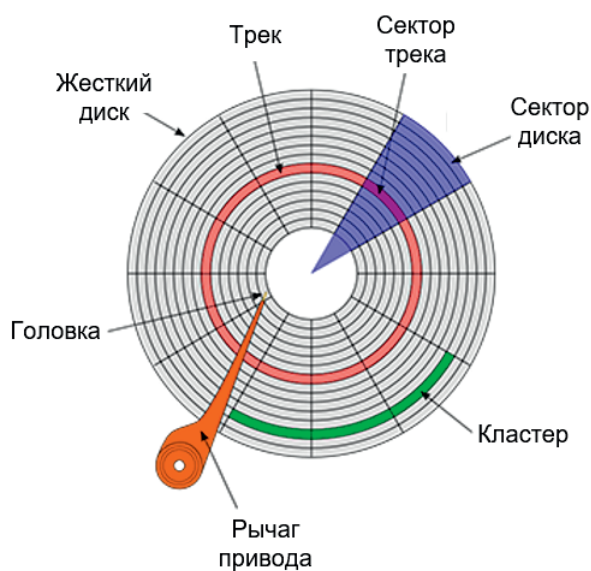


Рис. 78: Чтение данных с дискового носителя ²⁶

Рассмотрим, как размер блока влияет на эффективность работы распределенной файловой системы и производительность MapReduce-заданий. Большой размер блока оптимизирует издержки на общение клиентов с NameNode и экономит память, занимаемую структурами данных на NameNode. Однако с точки зрения MapReduce большой размер блока может снижать степень параллелизма. При обработке файл распределяется между меньшим числом узлов в кластере, ограничивая возможности по распараллеливанию чтения и обработки файла.

²⁶https://en.wikipedia.org/wiki/Disk_sector#Cluster

5.3.5 Задание топологии кластера

В конфигурационных файлах HDFS администратор может описать физическую структуру кластера — указать, какие узлы находятся в одних и тех же стойках или принадлежат разным дата-центрам. Эта информация называется **топологией кластера** и используется HDFS при выборе мест размещения реплик данных.

Алгоритм размещения реплик должен удовлетворять двум **основным требованиям**: обеспечивать высокую надежность хранения данных и хорошую пропускную способность при чтении и записи.

Принцип работы алгоритма следующий. Первая реплика блока размещается на узле, где работает клиент (если это возможно). Вторая реплика записывается на случайный узел в другой стойке, чтобы обеспечить отказоустойчивость в случае выхода из строя всей стойки. Третья реплика создается на другом случайном узле внутри стойки второй реплики. Дополнительные реплики (если они предусмотрены) размещаются на случайных узлах по всему кластеру с учетом равномерного распределения нагрузки между стойками. Такой подход является сбалансированным: он повышает надежность хранения данных и одновременно ограничивает объем межстоечного трафика, что важно при интенсивной работе с большими объемами данных.

Топология задается с помощью скриптов и конфигурационных файлов:

- node1 /dc1/rack1;
- node2 /dc1/rack1;
- node3 /dc1/rack2.

При чтении файла HDFS будет использована ближайшая к клиенту реплика. Расстояние между узлами определяется как расстояние между листьями в иерархии (узел, стойка, центр обработки данных (ЦОД) — корень иерархии).

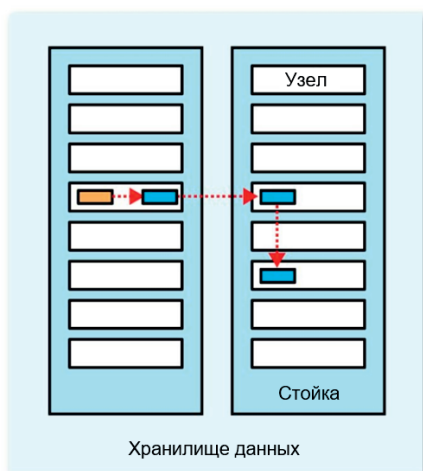


Рис. 79: Размещение реплик [1]

Также HDFS предоставляет возможность работать с файловой системой через командную строку. Большая часть команд похожи на соответствующие команды файловой системы Linux. Например, специальные команды `put`, `copyFromLocal`, `get`, `copyToLocal` предназначены для переноса файлов между локальной файловой системой и HDFS.

```
hdfs dfs -cmd <args>
hdfs dfs -ls /
hdfs dfs -cat /hdfsDirectory/hdfsFile
hdfs dfs -mkdir /hdfsDirectory
hdfs dfs -put localfsFile /hdfsDirectory
hdfs dfs -copyFromLocal localfsFile /hdfsDirectory
hdfs dfs -get hdfsFile /localfsDirectory
hdfs dfs -copyToLocal hdfsFile /localfsDirectory
hdfs dfs -rm /hdfsDirectory/hdfsFile
```

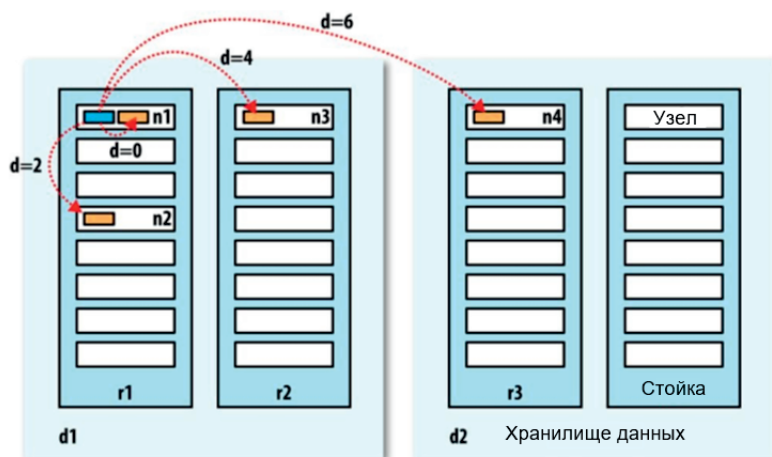


Рис. 80: Расстояние между листьями в иерархии [1]

5.4 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

- Какие компоненты входят в архитектуру HDFS?
- Какие данные хранятся в ОЗУ и на диске NameNode?
- В чем суть сценария чтения/записи данных в HDFS?
- Что такое Primary/Secondary и Active/Standby NameNode?
- Какие достоинства/недостатки при небольшом/большом размере блока в HDFS?
- В чем состоит алгоритм размещения реплик?

ГЛАВА 6

Hadoop MapReduce: Java API

В данной главе пойдет речь о принципах и подходах написания программ для Hadoop MapReduce.

6.1 Java как основной интерфейс для Hadoop

Для разработки MapReduce-программ в Hadoop используется язык Java. Фреймворк предоставляет программисту набор классов и интерфейсов, позволяющих реализовывать необходимые этапы обработки данных.

Ключи и значения, используемые в MapReduce-программах, должны реализовывать интерфейс `Writable`. Это обеспечивает возможность преобразования объектов в поток байтов и обратно — сериализацию и десериализацию. Кроме того, ключи должны реализовывать интерфейс `Comparable`, так как Hadoop выполняет сортировку данных по ключам между этапами Map и Reduce.

Чтобы упростить работу, Hadoop предоставляет готовые классы-обертки для основных атомарных типов Java, уже реализующих оба интерфейса `Writable` и `Comparable`. Это позволяет быстро начать работу без необходимости писать собственные реализации с нуля. Таким образом, должны быть реализованы следующие интерфейсы:

- Для ключей: `WritableComparable<T>`;
- Для значений: `Writable`.

```
package org.apache.hadoop.io
```

```
interface Writable {
    void readFields(DataInput in) throws IOException;
    void write(DataOutput out) throws IOException;
}

interface java.lang.Comparable<T> {
    int compareTo(T o);
}

// interface WritableComparable<T> extends Writable,
//     java.lang.Comparable<T>.
```

Существуют различные реализации `WritableComparable<T>`:

- `BooleanWritable` — логические значения (`true/false`);
- `ByteWritable` — целое число, представляемое как байт;
- `DoubleWritable` — число с плавающей запятой двойной точности (`double`);
- `FloatWritable` — число с плавающей запятой одинарной точности (`float`);
- `IntWritable` — целое число (`int`);
- `LongWritable` — длинное целое число (`long`);
- `Text` — строки символов, аналог класса `String` в Java, но оптимизированный для MapReduce;
- `NullWritable` — специальный тип, представляющий пропущенные значения.

6.1.1 Интерфейс `RawComparator`

Рассмотрим интерфейс `RawComparator`, который позволяет сравнивать фрагменты байтовых массивов, каждый из которых описывается смещением и длиной. Это позволяет сравнивать значения напрямую, не преобразуя их в объекты Java.

При обычной сортировке Hadoop десериализует каждый ключ, чтобы вызвать метод `compareTo`. Однако десериализация — это дополнительная вычислительная нагрузка, особенно при обработке больших объемов данных. Реализация `RawComparator` позволяет обойти этот шаг и сравнивать байтовые представления ключей напрямую, ускоряя процесс сортировки.

Класс, реализующий данный интерфейс, может быть задан методом `setSortComparatorClass` класса `Job`, который описывает текущее MapReduce-задание.

```
package org.apache.hadoop.io;
import java.util.Comparator;
public interface RawComparator<T> extends Comparator<T>
{
    public int compare(byte[] b1, int s1, int l1,
                      byte[] b2, int s2, int l2);
}
```

6.1.2 Распределитель (mapper)

Функция `map` в Hadoop задается путем создания собственного класса, унаследованного от шаблонного класса `Mapper`. Для этого необходимо переопределить метод `map`, в котором реализуется основная логика преобразования входных пар “ключ-значение” во множество промежуточных пар. Для возврата промежуточных результатов используется метод `write` объекта `Mapper.Context`, передаваемого в качестве аргумента в метод `map`.

Дополнительно может быть переопределен метод `setup`, который вызывается один раз при создании экземпляра `Mapper`. Этот метод часто используется для предварительной инициализации, в частности — для получения параметров конфигурации задания. Доступ к этим параметрам осуществляется через метод `getConfiguration` объекта `Mapper.Context`.

```
package org.apache.hadoop.mapreduce;

public class Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
    extends Object {
    protected void setup(
        org.apache.hadoop.mapreduce.Mapper.Context context)
        throws IOException, InterruptedException;

    protected void map(KEYIN key, VALUEIN value,
        org.apache.hadoop.mapreduce.Mapper.Context context)
        throws IOException, InterruptedException;
    ...
}

class Mapper.Context
{
    void write(KEYOUT key,VALUEOUT value)
        throws IOException, InterruptedException;
    ...
}
```

Существует ряд готовых реализаций, унаследованных от базового класса `Mapper`. Особого внимания заслуживает класс `MultithreadedMapper`, который позволяет выполнять вызовы функции `map` в нескольких потоках с использованием пула нитей. В отличие от стандартного `Mapper`, где метод

`map` вызывается строго последовательно (каждый новый вызов начинается только после завершения предыдущего), `MultithreadedMapper` обеспечивает параллельную обработку входных данных. Краткое описание наиболее часто используемых реализаций `Mapper` представлено в Таблице 10.

6.1.3 Редуктор (Reducer)

Класс `Reducer` в Hadoop аналогичен классу `Mapper` по структуре и принципу использования, но выполняет иную функцию в процессе обработки данных. Если задача `Mapper` — преобразовать входные данные в пары ключ-значение и распределить их по ключам, то задача `Reducer` — агрегировать все значения, соответствующие одному и тому же ключу, и, при необходимости, произвести дополнительную обработку.

```
package org.apache.hadoop.mapreduce;
```

Таблица 10: Описание реализаций распределителей

Распределитель	Описание
<code>FieldSelectionMapper<K,V></code>	<code>unix cut Mapper<K,V,Text,Text></code>
<code>InverseMapper<K,V></code>	<code>Mapper<K,V,V,K></code> , меняющий местами ключ и значение
<code>RegexMapper<K></code>	<code>Mapper<K,Text,Text,LongWritable></code> Образец: <code>mapreduce.mapper.regex</code> Группа: <code>regexmapper.group</code> . Возвращает: (совпадение, 1), если совпало
<code>TokenCountMapper<K></code>	Разбивает строку на токены с помощью <code>StringTokenizer</code> <code>Mapper<K,Text,Text,LongWritable></code> (токен, 1)
<code>MultithreadedMapper<K,V></code>	Многонитевый распределитель. Число нитей в пуле: <code>multithreadedmapper.threads</code>

```
public class Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
    extends Object {
        protected void setup(
            mapreduce.Reducer.Context context)
            throws IOException, InterruptedException;
        protected void reduce(KEYIN key,
            Iterable<VALUEIN> values,
            mapreduce.Reducer.Context context)
            throws IOException, InterruptedException;
        ...
    }

class Reducer.Context
{
    void write(KEYOUT key,VALUEOUT value)
        throws IOException, InterruptedException;
    ...
}
```

6.1.4 Комбайнер (Combiner)

Комбайнер — это вспомогательный этап, который выполняется на стороне Map-стадии до пересылки промежуточных данных на Reduce-узлы. По сути, это локальное применение логики редуктора к частичным результатам с целью их предварительного агрегирования. Это уменьшает объем данных, передаваемых по сети, что особенно важно при работе с большими объемами информации.

Важно понимать, что не все функции могут быть корректно вычислены с помощью комбайнера. Комбайнер применим только к тем агрегатным функциям, которые являются дистрибутивными (например, сумма, максимум) или алгебраическими (например, среднее значение, вычисляемое через

Таблица 11: Описание и примеры различных типов агрегатных функций

Типы агрегатных функций	Примеры	Описание
Дистрибутивные	max, min, sum, count	Функцию можно вычислить рекурсивно на подмножествах данных
Алгебраические	avg, var	Функцию можно вычислить на подмножествах данных с помощью нескольких дистрибутивных функций
Холистические	median, quantiles	Функцию нельзя вычислить рекурсивно на подмножествах данных

сумму и количество). Для холистических функций (например, медианы или перцентилей) комбайнер использовать нельзя, так как частичные результаты не позволяют восстановить итоговое значение. Примеры различных типов функций и их пригодность к использованию в комбайнере представлены в Таблице 11.

В Hadoop MapReduce комбайнер реализуется аналогично Reduce: программист указывает класс с помощью метода `job.setCombinerClass(...)`. При этом не требуется отдельный базовый класс `Combiner`, подойдет созданный класс для Reduce. Его можно указать в MapReduce-задании в качестве класса, используемого и для Reduce, и для Combine.

```
public static class MyReducer extends
    Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();
    public void reduce(Text key,
        Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
```

```

    int sum = 0;
    for (IntWritable val : values) {
        sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
}
}
job.setReducerClass(MyReducer.class);
job.setCombinerClass(MyReducer.class);

```

6.2 Тасовка (Shuffle)

Один из ключевых этапов выполнения MapReduce-программы — это передача промежуточных данных от mapper'ов к reducer'ам. Этот процесс называется **тасовкой (shuffle)**, общая схема которой представлена на Рисунке 81 [1].

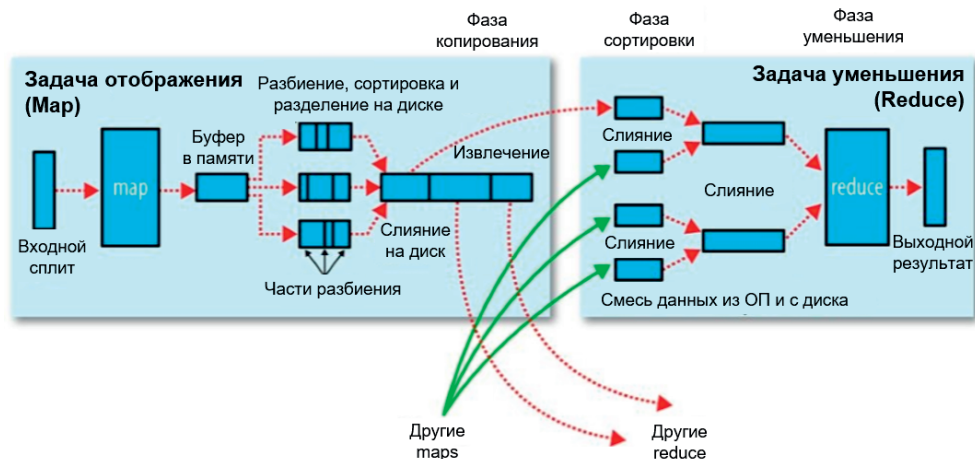


Рис. 81: Операция тасовки

После того как функция `map` сформировала пары “ключ-значение”, они помещаются в специальный циклический буфер в оперативной памяти. Отдельная фоновая нить периодически считывает накопленные данные, сортирует их по ключам и делит на части (секции) — по числу `reducer`’ов в задании. Каждая секция содержит данные, предназначенные для конкретного `reducer`’а. Полученные промежуточные файлы сохраняются на диск.

Сформированные отсортированные файлы, разбитые на секции, объединяются в общий файл с помощью сортировки слиянием (`merge sort`). Содержимое секций итогового файла отправляется на `Reducer`. Каждый `reducer` в несколько этапов выполняет сортировку слиянием данных, полученных от `Mapper`.

Таким образом, тасовка заключается в следующем:

- Запись в циклический буфер в памяти;
- Фоновое чтение из буфера, сортировка по ключу и запись файла на диск;
- Объединение отсортированных файлов.

6.3 Слияние (Reduce)

Сортировка слиянием на стороне `Reduce` проводится в несколько этапов. В настройках фиксируется число файлов, которые объединяются на каждом этапе. Результаты промежуточного слияния хранятся на жестком диске в виде файлов. Слияние организуется таким образом, чтобы минимизировать использование диска. Визуализация операции слияния представлена на Рисунке 82.

Таким образом, использование `Combiner` позволяет:

- Уменьшить объем трафика при передаче данных по сети данных;

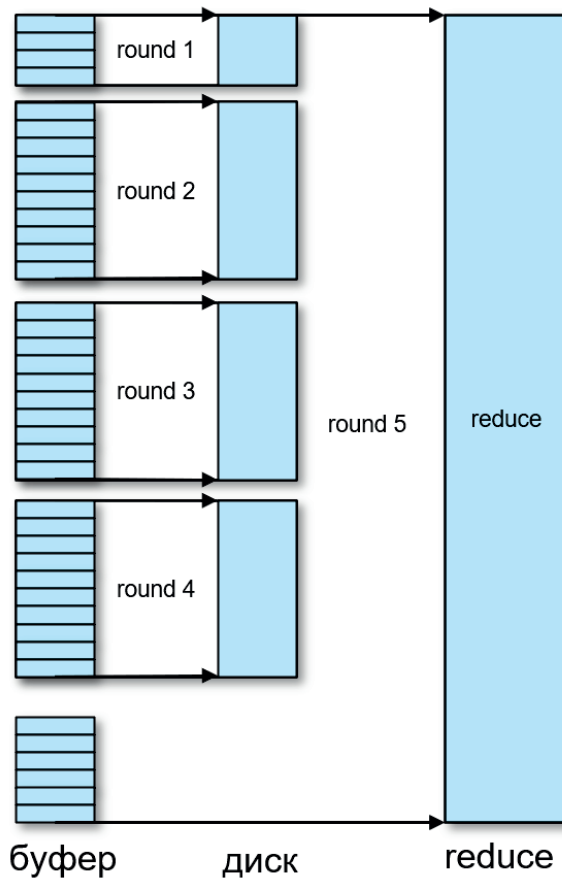


Рис. 82: Операция слияния

- Уменьшить объем данных, записываемых на диск в процессе объединения отсортированных последовательностей на стороне Reduce.

6.4 Терминология MapReduce в Hadoop

При реализации MapReduce-программ используется следующая терминология. Большинство терминов имеют соответствующие классы в Java API.

- **Driver** — программа, реализующая алгоритм для Hadoop MapReduce;

- **Job** — задание, соответствующее данной программе, которое должно быть вычислено на кластере;
- **Configuration** — набор значений параметров фреймворка MapReduce для исполнения данного задания и параметров, относящихся к логике задания;
- **Task** — задача, часть задания, которая решается экземпляром класса Mapper или Reducer;
- **Combiner** — reducer, предназначенный для исполнения на mapper'е;
- **Partitioner** — задает логику распределения ключей по reducer'ам;
- **InputFormat** — предназначен для разбора входного формата данных (обычно находящихся в HDFS), формирования последовательности пар “ключ-значение” и интеграции с mapper;
- **OutputFormat** — предназначен для сохранения результатов работы MapReduce-задания в некотором формате хранения.

6.5 Передача параметров

Рассмотрим пример классической программы MapReduce:

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
```

```
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class WordCount {
    public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable>{
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object k, Text v, Context context)
            throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(v.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class IntSumReducer
        extends Reducer<Text,IntWritable,Text,IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values,
            Context context)
            throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }
}
```

```
        }
        result.set(sum);
        context.write(key, result);
    }
}

public static void main(String[] args)
throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf,
                                                    args).getRemainingArgs();

    Job job = Job.getInstance(conf, "WordCount");
    FileInputFormat.addInputPath(job,
                                   new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job,
                                      new Path(otherArgs[1]));
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    job.setNumReduceTasks(2);

    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

Пример, приведенный выше, не предусматривает возможность передачи параметров конфигурации через командную строку при запуске программы. Однако на практике такая возможность полезна для задания путей к вход-


```
    job.setJarByClass(ToolWordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    return job.waitForCompletion(true) ? 0 : 1;
}
}
```

Теперь возможно передавать параметры через командную строку следующим образом:

```
$ hadoop jar ToolWordCount.jar ToolWordCount input output
$ hadoop jar ToolWordCount.jar ToolWordCount input output
    -D mapred.reduce.tasks=5
$ hadoop jar ToolWordCount.jar ToolWordCount input output
    -conf config.xml
$ hadoop jar ToolWordCount.jar ToolWordCount input output
    -fs hdfs://namenode -jt localhost:8021
```

Рассмотрим значения опций:

- **-D** позволяет задать значение параметров фреймворка или пользовательского приложения;
- **-conf** позволяет передать параметры конфигурации в виде xml-файла;
- **-fs** и **-jt** задают NameNode HDFS и точку входа на кластер для запуска задания — планировщик (об этом мы будем говорить позднее);
- **-files** указывает имена файлов, разделенные запятыми, которые будут скопированы в кластер MapReduce;

- `-libjars` указывает имена файлов jar, разделенные запятыми, для включения в путь к классу;
- `-archives` указывает имена архивов, разделенные запятыми, которые будут разархивированы на вычислительных машинах.

6.6 Формат входных данных

Java API предоставляет ряд абстрактных и конкретных классов, реализующих входные форматы данных. Большинство форматов основаны на файлах HDFS. Часть из них позволяет читать текстовые файлы, а формат файлов последовательностей способен хранить пары “ключ-значение” в компактном бинарном виде. Классы входных форматов данных представлены на Рисунке 83.

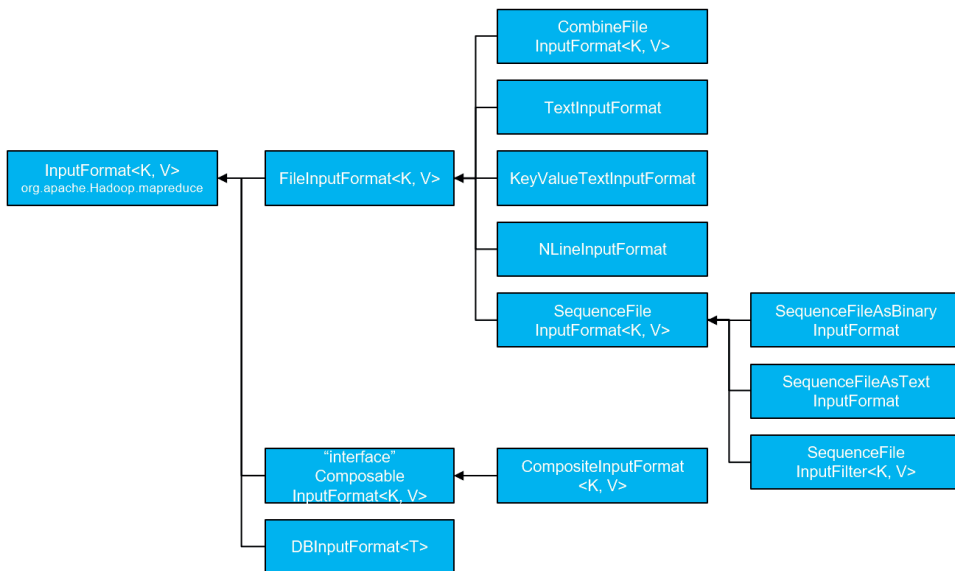


Рис. 83: Входные форматы данных [1]

Рассмотрим подробнее несколько входных форматов данных:

- В `TextInputFormat` каждая строка файла порождает пару “ключ-значение”, значением является сама строка, ключом — смещение относительно начала файла;
- `KeyValueTextInputFormat` позволяет с помощью настроек задать положение ключа и значения в строке текстового файла;
- `SequenceFileInputFormat` позволяет работать с бинарными ключами и значениями и компактно хранить данные в бинарном виде;
- В `NLineInputFormat` каждые N строк текстового файла формируют одно значение.

Hadoop MapReduce допускает перечисление в задании нескольких путей до входных файлов и использование шаблонов — имен файлов с символом `*`.

6.7 Разбиение на порции (split)

Важной функцией входного формата является разбиение файла на порции (сплит или `split`), для каждой из которых создается отдельный маркер и выделяется пары “ключ-значение”. Для реализации входного формата существуют следующие абстрактные типы данных в MapReduce Java API:

- `InputSplit`, определяющий порцию данных;
- `RecordReader`, разбивающий порцию на пары “ключ-значение”;
- `InputFormat`, инкапсулирующий логику разбиения на порции и конкретного `RecordReader`’а.

```
public abstract class InputSplit {  
    public abstract long getLength()  
        throws IOException, InterruptedException;
```

```
    public abstract String[] getLocations()
        throws IOException, InterruptedException;
    // host names
}

public abstract class InputFormat<K, V> {
    public abstract List<InputSplit>
        getSplits(JobContext context)
        throws IOException, InterruptedException;
    public abstract RecordReader<K, V>
        createRecordReader(InputSplit split,
                           TaskAttemptContext context)
        throws IOException, InterruptedException;
}

public interface RecordReader<K,V> {
    boolean next(K key, V value) throws IOException;
    K createKey();
    V createValue();
    long getPos() throws IOException;
    public void close() throws IOException;
    float getProgress() throws IOException;
}
```

Для обработки каждой порции файла создается отдельный маррер. Визуализация использования маррер'ов представлена на Рисунке 84. Число маррер'ов равно числу сплитов.

В некоторых ситуациях данных не очень много, а вычислительная нагрузка велика. Здесь могут оказаться полезны порции небольшого размера. Размер порции для `FileInputFormat` можно регулировать с помощью параметров, указанных в Таблицах 12 и 13.

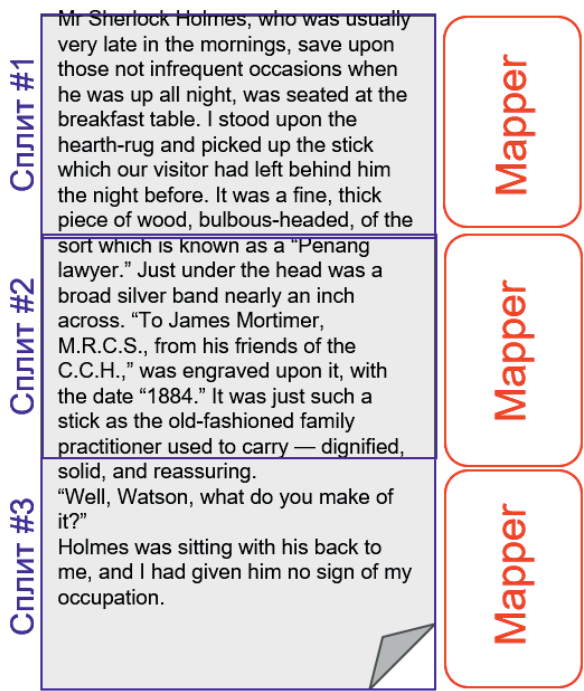


Рис. 84: Визуализация использования mapper’ов. Число mapper’ов равно числу сплитов

Таблица 12: Описание свойств, связанных с размером блока

Название свойства	Тип	По умолчанию	Описание
mapreduce.input. .fileinputformat. .split.minsize	long	1	Минимальный раз- мер сплита в бай- тах
mapreduce.input. .fileinputformat. .split.maxsize	long	MAX_VALUE	Максимальный размер сплита в байтах
dfs.blocksize	long	128 МБ	Размер блока HDFS

Таблица 13: Описание размеров блока

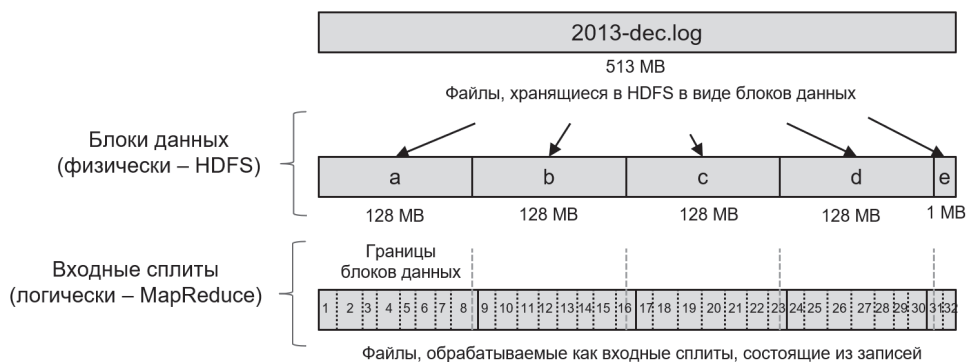
minSize	maxSize	blockSize	splitSize
1	Long.MAX_VALUE	128МБ	128МБ
1	Long.MAX_VALUE	256МБ	256МБ
256МБ	Long.MAX_VALUE	128МБ	256МБ (2 блока!)
1	64МБ	128МБ	64МБ

6.8 Реализация TextInputFormat

Рассмотрим каким образом класс `TextInputFormat` обеспечивает корректное разбиение текстового файла на порции (сплиты), близкие по размеру, с одновременным сохранением целостности строк в каждой порции.

Информация о размере файла хранится в метаданных, доступных на узле `NameNode`. Это позволяет `TextInputFormat` получить точную информацию о размере файла и на ее основе определить границы порций. За обработку содержимого каждой порции отвечает объект `RecordReader`, использующий следующий подход: чтение начинается с заданного смещения (начала порции), продолжается до конца текущей строки, после чего начинается формирование пар “ключ-значение” (где ключом обычно выступает смещение, а значением — содержимое строки) для всех последующих строк, пока не достигнута граница порции.

В большинстве случаев граница порции будет приходиться на середину строки. В этом случае `RecordReader` продолжит чтение до завершения текущей строки, даже если это выходит за пределы порции. Это необходимо для обеспечения целостности строк. Именно по этой причине `RecordReader` для всех порций (кроме первой) должен пропускать содержимое до первого символа перевода строки. Пример соответствия порций и блоков изображен на Рисунке 85.

Рис. 85: Соответствие порций и блоков ²⁷

6.9 Выходные данные заданий

Существует ряд выходных форматов данных. Для многих простых задач используется текстовый формат. Каждый reducer создает свой выходной файл с именем `part-r-XXXXX`. Соответственно число выходных файлов равно числу reducer'ов.

В задании можно отключить Reduce-стадию. Тогда выходные файлы будут создаваться на основе пар “ключ-значение”, порожденных mapper'ами. Число файлов будет равно числу mapper'ов. Их имена будут иметь вид: `part-m-XXXXX`. Например, выходной файл для каждого mapper (если `mapreduce.reduce.tasks = 0`):

- `part-m-00000`;
- `part-m-00001`;
- `part-m-00002`.

Выходной файл для каждого reducer:

- `part-r-00000`;

²⁷https://training-course-material.com/training/Hadoop_Administration

- `part-r-00001`;
- `part-r-00002`;

Следует учесть, что если `mapreduce.reduce.tasks = 1`, то выходной файл вероятно будет глобально отсортирован по ключу. Однако MapReduce не гарантирует отсортированности файлов `reducer` по ключу.

6.10 Разбиение на части (Partitioner)

Рассмотрим **Partitioner** — это компонента Hadoop MapReduce, предназначенная для распределения ключей по `reducer`'ам. Число `reducer`'ов определяет пользователь. Цель `partitioner`'а — отправить данные с одними ключами на один `reducer`, с другими — на другой.

API Hadoop гарантирует, что не возникнет ситуации, при которой часть данных с некоторым ключом отправится на один `reducer`, а другая часть с тем же ключом — на другой.



Рис. 86: Разбиение на части

Реализация Partitioner по умолчанию видится логичной (остаток от деления на число `reducer`'ов значения хеш-функции от ключа). Однако, имеется и возможность переопределения Partitioner, которая может быть полезна в

случае неравномерной частоты встречаемости ключей. Например, если один ключ встречается чаще остальных, то возникнет неравномерное распределение работы по reducer'ам.

```
package org.apache.hadoop.mapreduce
public abstract class Partitioner<K,V> {
    public abstract int getPartition(
        K key, V value, int numPartitions);
}

job.setPartitionerClass(HashPartitioner.class);

public class HashPartitioner<K, V>
    extends Partitioner<K, V> {
    public int getPartition(
        K key, V value, int numReduceTasks) {
        return (key.hashCode() & Integer.MAX_VALUE)
            % numReduceTasks;
    }
}
```

6.11 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

- Какие интерфейсы должны реализовывать классы ключей и значений?
- Какой прототип имеют функции map и reduce?
- Для каких математических функций можно использовать Combiner?

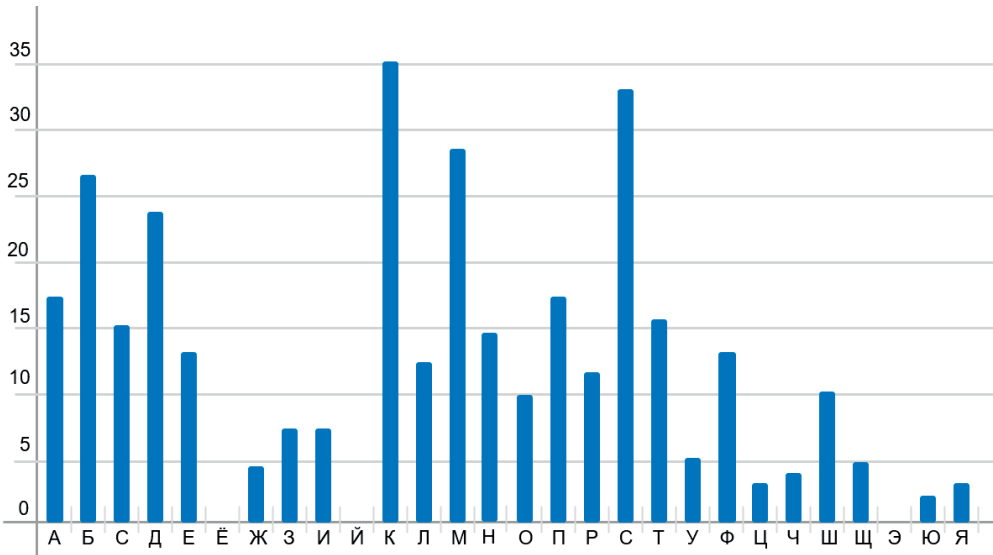


Рис. 87: Пример гистограммы распределения значений ключа. Распределение неравномерно



Рис. 88: Для знакомства с Hadoop рекомендуются книги [1,8,9]

- В чем преимущества использования Combiner?
- В чем суть тасовки?
- В чем состоит основная идея реализации TextInputFormat?
- В каких ситуациях следует использовать свою реализацию Partitioner?

ГЛАВА 7

Hadoop MapReduce: детали

В данной главе продолжается обсуждение принципов написания программ для Hadoop MapReduce. Далее подробно рассматриваются особенности программных реализаций, использующих сжатие данных, счетчики и сортировку, а также подходы контроля за выполнением задач.

7.1 Файл последовательности (SequenceFile)

Для эффективного хранения больших объемов данных, а также в ситуациях, когда требуется работа с бинарной информацией, в Hadoop может использоваться специальный формат **файлов последовательностей (SequenceFile)**. Он представляет собой один из вариантов организации хранения данных в HDFS, ориентированный на работу с парами “ключ-значение”. В последующих главах также будут рассмотрены более продвинутые форматы, такие как RCFile и ORC, обладающие расширенными возможностями хранения и обработки данных.

SequenceFile состоит из записей, каждая из которых содержит ключ, значение и соответствующую информацию о длине этих элементов. С некоторой периодичностью между записями вставляется специальный элемент — синхронизационный маркер. Он позволяет найти начало очередной записи при чтении данных с произвольной позиции в файле. Синхронизационный маркер — это аналог символа конца строки в **TextInputFormat**. В формате файла последовательности со сжатием каждое значение хранится в сжатом виде, ключ не сжимается. В настройках формата можно выбрать кодек, который будет использован при сжатии.

Также существует разновидность файла последовательности с **компрессией по блокам**. В этом случае данные группируются в блоки с заданным в конфигурационном файле числом пар “ключ-значение”. В рамках каждого блока отдельно в сжатом виде хранятся ключи и отдельно значения. Это обеспечивает лучшую эффективность компрессии. Между блоками вставляется синхронизационный маркер.

Вероятность коллизии при использовании 16-байтового MD5 маркера в 1 ПБ файле — 10^{-23} .

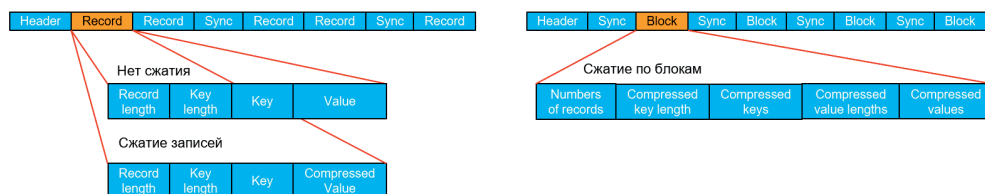


Рис. 89: Варианты сжатия [1]

7.2 Мотивация сжатия данных

Применение сжатия позволяет решать сразу несколько ключевых задач:

- Снижение затрат на хранение данных за счет уменьшения их объема;
- Уменьшение времени, необходимого для передачи данных по сети, а также для операций чтения и записи с диска.

Сжатие может использоваться на различных этапах обработки данных:

- Сжатие входных данных реализуется в случае поступления исходных данных в виде архивов;
- Сжатие выходных данных, формируемых на стадии выполнения функции `map`, позволяет снизить объем пересылаемой информации между узлами;

- Сжатие итоговых выходных данных (после завершения выполнения программы) особенно важно при подготовке результатов к долговременному хранению или передаче во внешние системы.

7.3 Обработка сжатых входных данных

Существует множество алгоритмов сжатия, поддерживаемых в Hadoop MapReduce непосредственно или через внешние библиотеки. Важным требованием к алгоритму сжатия в случае MapReduce-обработки является возможность распаковки архива по частям, поскольку каждый mapper в идеале должен получить на вход данные, соответствующие своему блоку файла.

Такая возможность предусмотрена в некоторых алгоритмах. Например, bzip2 вставляет внутрь архива синхронизирующие маркеры. В процессе работы читателя формата входных данных (например, `TextInputFormat`) MapReduce выполняет распаковку по частям bzip2-архива, в котором находится исходный файл с данными.

Если же исходный файл сжат с помощью gzip, который не поддерживает распаковку по частям, MapReduce не будет пытаться разбить файл на несколько split'ов, каким бы большим он ни был. Gzip может быть полезен для сжатия большого числа маленьких файлов. Работа с такими файлами напрямую неэффективна:

- Чтение большого числа файлов из HDFS потребует интенсивного общения с NameNode;
- Будет создано слишком большое число mapper'ов (на каждый файл будет создаваться отдельный mapper).

Формат сжатия определяется по расширению файла. Основными вопросами, связанными с форматом сжатия, являются:

- Поддержка разбиения;

Таблица 14: Описание форматов сжатия

Формат сжатия	ПО	Алгоритм	Расширение	Распаковка по частям
Deflate	zlib	Deflate	.deflate	Нет
gzip	gzip	Deflate	.gz	Нет
bzip2	bzip2	bzip2	.bz2	Да
LZO	LZO	LZO	.lzo	+ - (предварительное индексирование или hadoop-lzo)
LZ4	LZ4	LZ4	.lz4	+ - (проект hadoop-4mc)
Snappy	Snappy	Snappy	.snappy	Нет

- Эффективность;
- Время сжатия.

Перечисленные свойства регулируют настройки сжатия для выходных файлов задания и в процессе тасовки. При обработке входных данных необходимость распаковки определяется по расширению файла.

При обработке реальных массивов больших данных большую роль играют коэффициент сжатия и время упаковки и распаковки (см. Рисунок 90).

Из приведенных графиков на Рисунке 91 следует, что, например, bzip2 обладает эффективностью сжатия, но работает медленнее, чем gzip.

7.4 Обработка ошибок и сбоев

Важным механизмом Hadoop MapReduce является **обработка ошибок и сбоев** (англ. failure handling). Он обеспечивает надежность распределенных вычислений в условиях работы с большим количеством узлов и нестабиль-

Таблица 15: Описание форматов сжатия

Имя свойства	Тип свойства	Значение по умолчанию	Описание
mapreduce.output. .fileoutputformat. .compress	boolean	false	Сжатие выходных данных
mapreduce.output. .fileoutputformat. .compress.codec	Classname	org.apache. .hadoop.io. .compress. .DefaultCodec	Кодек для сжатия выходных данных
mapreduce.output. .fileoutputformat. .compress.type	String	RECORD	Тип сжатия для выходных SequenceFile (NONE, RECORD, BLOCK)
mapreduce.map. .output.compress	boolean	false	Сжатие выходных данных mapper (всегда в формате SequenceFile)
mapreduce.map. .output.compress. .codec	Classname	org.apache. .hadoop.io. .compress. .DefaultCodec	Кодек для сжатия выходных данных mapper

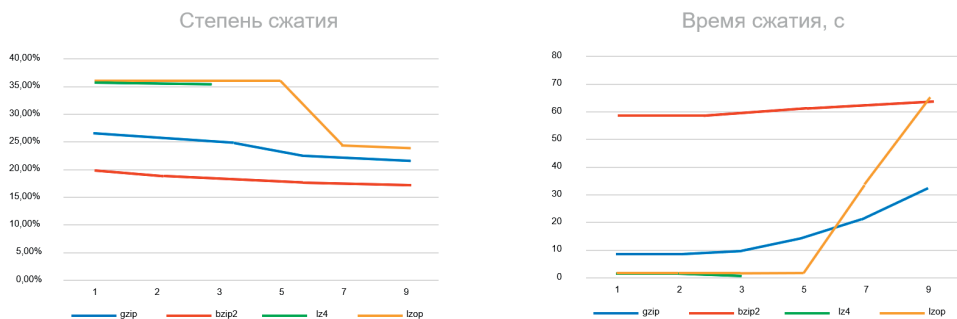


Рис. 90: Эффективность алгоритмов сжатия. Сжимаемый файл 445 МБ, режимы сжатия (от худшего к лучшему по степени сжатия): 1, 2, 3, 5, 7, 9

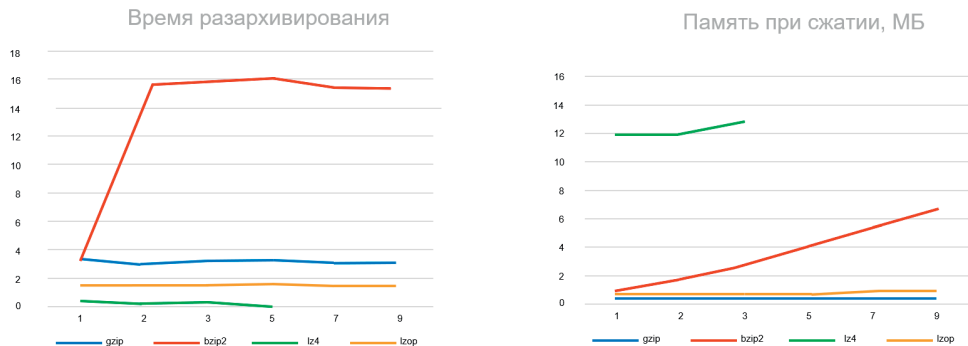


Рис. 91: Сравнение времени компрессии и декомпрессии. Сжимаемый файл 445 МБ, режимы сжатия (от худшего к лучшему по степени сжатия): 1, 2, 3, 5, 7, 9

ной вычислительной среды. Согласно модели MapReduce, сбоями (failures) считаются следующие ситуации:

- Возникновение ошибки (исключения) при выполнении пользовательского кода, реализующего функции `map` или `reduce`
- Наличие некорректных или поврежденных данных, препятствующих их успешной обработке;

- Принудительное завершение задачи планировщиком из-за превышения допустимого объема потребляемой памяти;
- Аварийное отключение или недоступность вычислительного узла по сетевым причинам либо вследствие аппаратного сбоя;
- Снижение производительности узла по причине деградации оборудования (например, замедление работы накопителя или процессора). Хотя такая ситуация формально не считается сбоем, она также учитывается в Hadoop и обрабатывается отдельными механизмами, направленными на снижение влияния “медленных” узлов на общее время выполнения задания.

В реальных вычислительных кластерах выполнение MapReduce-приложений управляется планировщиком YARN, который выделяет каждому заданию ограниченный объем вычислительных ресурсов (например, оперативная память и процессорное время). Нарушение этих ограничений может привести к принудительному завершению выполнения задачи, что также рассматривается как сбой.

При возникновении сбоя в процессе выполнения MapReduce-задания возникает вопрос: необходимо ли полностью перезапускать все задание или достаточно восстановить его отдельные части? Архитектура Hadoop MapReduce предполагает, что в случае сбоя перезапускается только та задача (mapper или reducer), в процессе выполнения которой произошла ошибка. Такой подход существенно снижает избыточные вычисления и повышает эффективность работы системы.

Дополнительный вопрос, связанный с восстановлением: каким образом организуется повторный запуск reducer’а, и как система обрабатывает уже завершенные mapper’ы? Ответ: выходные данные всех успешно завершенных mapper-задач сохраняются на диске до тех пор, пока все задание не будет полностью завершено. Это позволяет избежать повторного исполнения уже завершенных этапов и использовать промежуточные результаты при повтор-

ном запуске reducer'ов.

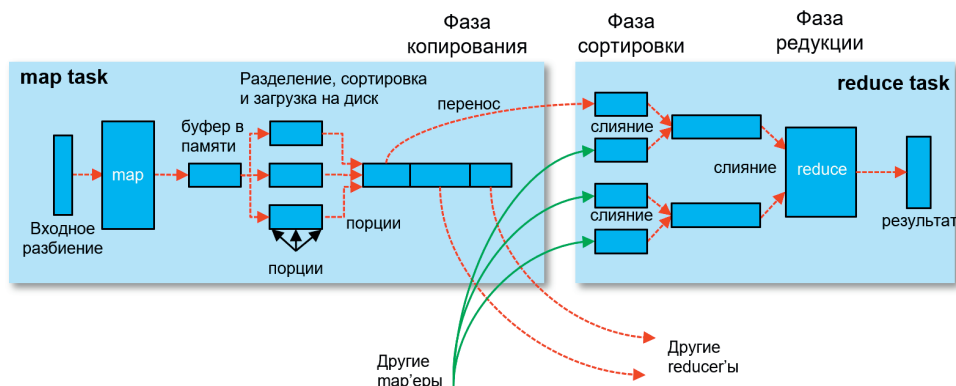


Рис. 92: Схематичное изображение MapReduce-задачи [1]

7.4.1 Идемпотентность

Важным термином в контексте обработки сбоев является идемпотентность. **Идемпотентность** — свойство объекта или операции при повторном применении операции к объекту давать тот же результат, что и при первом (лат. *idem* — “тот же самый”, *potens* — “способный”).

Применительно к MapReduce:

- На одних и тех же входных данных повторный запуск задания должен привести к аналогичным результатам;
- При перезапуске задач задания из-за сбоев узлов результат выполнения задания не должен меняться.

Идемпотентное задание MapReduce будет возвращать один и тот же результат при повторном запуске. В случае сбоя результат исполнения задания после перезапуска будет таким же, как если бы сбоя не было.

Рассмотрим следующий вопрос: будет ли идемпотентным задание, в котором функция **reduce** возвращает первое значение из переданного ей спис-

Таблица 16: Время выполнения задачи

Свойство	По умолча- нию	Описание
mapreduce.task. .timeout	600000	Число миллисек., в течение которых ожидается активность от задачи (чте- ние, запись, изменение счетчиков и т.п.)
mapreduce.map. .maxattempts	4	Сколько раз mapper может быть пере- запущен из-за сбоя до признания всего задания неуспешным
mapreduce.reduce. .maxattempts	4	Сколько раз reducer может быть пере- запущен из-за сбоя до признания всего задания неуспешным

ка значений, соответствующих заданному ключу? Ответ: такая программа неидемпотентна, так как порядок поступления данных от различных таррер-задач к reducer’у может отличаться при каждом запуске. Поскольку функция reduce опирается на первое значение в последовательности, изменение поряд-ка входных данных приведет к различным результатам, нарушая условие идемпотентности.

7.5 Время выполнения задачи

Для обнаружения деградации оборудования или неоптимальной реализа-ции кода функций **map** или **reduce**, в конфигурационном файле задается мак-симальное время выполнения задачи. Если в задаче происходит сбой, задача перезапускается. Число перезапусков до признания всего задания неуспеш-ным также задается в настройках задания (см. Таблицу 16).

7.6 Счетчики (Counters)

При обработке больших данных бывает полезно в реальном времени отслеживать значения статистических показателей, релевантных в конкретном случае: например, число некорректных записей или общую длину обработанных текстов. Hadoop предоставляет такую возможность в виде **механизмов счетчиков**.

Счетчик можно интерпретировать как глобальную в рамках задания целочисленную переменную, для которой определена операция увеличения текущего значения на некоторое другое значение. Программист может сам определить подобный именованный счетчик и вставить код модификации значения счетчика в тела функций `map` и `reduce`. Кроме этого, фреймворк MapReduce автоматически использует предопределенные счетчики, которые подсчитывают важную статистику исполнения любого задания.

Мотивация использования счетчиков:

- Удобное средство мониторинга задания;
- Поддержка корректного подсчета статистических показателей в условиях перезапуска задач. При перезапуске задачи счетчики будут сброшены к значениям, которые они имели до запуска задачи.

```
enum WordType {
    BAD,
    GOOD
}

if(isBad(word))
    context.getCounter(WordType.BAD).increment(1);

// Dynamic counter
context.getCounter("WordStat", "SpacesCount").increment(1);
```

Системные счетчики подразделяются на счетчики задач (**Task Counters**), заданий (**Job Counters**), прочитанных и записанных данных (**File Input/Output Format Counters**). Счетчики можно объединять в группы. Значение счетчика можно увидеть в пользовательском интерфейсе Hadoop или обратиться к счетчику через API по идентификатору задания.

Приведем пример кода по получению значения счетчика **REDUCE_INPUT_GROUPS** (количество групп ключей, обработанных на шаге Reduce, или число вызовов **reduce**):

```
String jobIdStr = "job_202406161234_0001";

Cluster cluster = new Cluster(conf);
Job job = cluster.getJob(JobID.forName(jobId));
Counters counters = job.getCounters();
Counter counter = counters.findCounter(
    "org.apache.hadoop.mapreduce.TaskCounter",
    "REDUCE_INPUT_GROUPS");
System.out.println(counter.getValue());
```

В Таблице 17 приведены некоторые предопределенные счетчики задач.

7.7 Сортировка данных

Как было рассмотрено ранее, Hadoop может использоваться для распределенной обработки и сортировки больших объемов данных (например, строк в текстовых файлах). Рассмотрим детально программу с минимальным функционалом для запуска MapReduce-задания:

```
public class MinimalMapReduce extends Configured
implements Tool {
    @Override
```

Таблица 17: Предопределенные счетчики задач

Счетчик	Описание
MAP_INPUT_RECORDS	Число входных записей, обработанных всеми mapper'ами
MAP_INPUT_BYTES	Суммарный объем входных данных всех вызовов map
MAP_OUTPUT_RECORDS	Число выходных пар, созданных всеми mapper'ами
MAP_OUTPUT_BYTES	Суммарный объем выходных данных всех вызовов map
MAP_OUTPUT_MATERIALIZED_BYTES	Суммарный объем выходных данных всех mapper'ов, записанных на диск (сжатие влияет!)
COMBINE_INPUT_RECORDS	Число входных записей, обработанных всеми combiner'ами
COMBINE_OUTPUT_RECORDS	Число выходных пар, созданных всеми combiner'ами
REDUCE_INPUT_GROUPS	Число входных групп ключей, обработанных всеми reducer'ами
REDUCE_INPUT_RECORDS	Число входных записей, обработанных всеми reducer'ами
GC_TIME_MILL	Суммарное время на сборку мусора (в миллисек.)

```
public int run(String[] args) throws Exception {
    if (args.length != 2) {
        System.err.printf("Usage: %s " +
            "[generic options] <input> <output>\n",
            getClass().getSimpleName());
        ToolRunner.printGenericCommandUsage(System.err);
        return -1;
    }

    Job job = Job.getInstance(getConf());
    job.setJarByClass(getClass());
    FileInputFormat.addInputPath(job,
        new Path(args[0]));
    FileOutputFormat.setOutputPath(job,
        new Path(args[1]));
    return job.waitForCompletion(true) ? 0 : 1;
}

public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new MinimalMapReduce(),
        args);

    System.exit(exitCode);
}
}
```

Она соответствует программе с явно указанными значениями по умолчанию, которая приведена далее:

```
public class MinimalMapReduceWithDefaults extends Configured
    implements Tool {
    @Override
    public int run(String[] args) throws Exception {
```

```
Job job = Job.getInstance(getConf());
FileInputFormat.addInputPath(job,
                               new Path(args[0]));
FileOutputFormat.setOutputPath(job,
                                new Path(args[1]));

if (job == null) {
    return -1;
}
job.setInputFormatClass(TextInputFormat.class);
job.setMapperClass(Mapper.class);

job.setMapOutputKeyClass(LongWritable.class);
job.setMapOutputValueClass(Text.class);

job.setPartitionerClass(HashPartitioner.class);

job.setNumReduceTasks(1);
job.setReducerClass(Reducer.class);

job.setOutputValueClass(Text.class);
job.setOutputFormatClass(TextOutputFormat.class);

return job.waitForCompletion(true) ? 0 : 1;
}

public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(
        new MinimalMapReduceWithDefaults(),
        args);
    System.exit(exitCode);
}
```

```
    }  
}  
  
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {  
    protected void map(KEYIN key, VALUEIN value, Context context)  
        throws IOException, InterruptedException {  
        context.write((KEYOUT) key, (VALUEOUT) value);  
    }  
}  
  
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {  
    protected void reduce(KEYIN key, Iterable<VALUEIN> values,  
        Context context)  
        throws IOException, InterruptedException {  
        for (VALUEIN value: values) {  
            context.write((KEYOUT) key, (VALUEOUT) value);  
        }  
    }  
}  
  
public class HashPartitioner<K, V> extends Partitioner <K, V> {  
    public int getPartition(K key, V value,  
        int numReduceTasks) {  
        return (key.hashCode() & Integer.MAX_VALUE) %  
            numReduceTasks;  
    }  
}
```

Для корректной сортировки строк файла необходимо внести одно важное изменение в реализацию: Mapper должен возвращать пару вида (значение, `NullWritable`). По умолчанию количество reducer равно 1.

Данная реализация действительно выполняет глобальную сортировку строк файла (все данные обрабатываются одним reducer). Однако такой подход имеет существенный недостаток: слияние отсортированных данных выполняется нераспределенно и ограничивает масштабируемость. Если же увеличить количество reducer для распараллеливания обработки, то глобальная сортировка будет нарушена (каждый Reducer будет сортировать только свою часть данных).

Решением проблемы может стать сбор частоты встречаемости строк на подмножестве данных. На основе данных реализуется Partitioner, который распределяет строки по Reducer'ам, ориентируясь на разбиение упорядоченного множества строк на интервалы с примерно равной частотой встречаемости. В Hadoop MapReduce API для этих целей предусмотрены специализированные классы: **TotalOrderPartitioner** обеспечивает глобальный порядок распределения данных между Reducer'ами; **RandomSampler** используется для выборки данных и оценки квантилей.

Пример. Реализация TeraSort в Hadoop MapReduce:

- Расчет строковых “квантилей” на подвыборке данных;
- Построение схемы разбиения (partitioning) данных на основе квантилей;
- Применение классов `RandomSampler(double freq, int numSamples, int maxSplitsSampled)` и `TotalOrderPartitioner`, опция `-files` для клонирования файлов по узлам.

7.8 Примеры больших данных

7.8.1 Биржевые данные

Биржевые данные представляют собой факты выставления и снятия заявок на покупку или продажу финансовых инструментов, а также произошедшие сделки. Данные поступают с высокой скоростью. Для многих торго-

вых стратегий критически важны скорость получения новых данных и скорость формирования реакции торговой стратегии на полученные данные — покупки или продажи активов.

Bid (покупка)	Price (Цена)	Ask (продажа)
	\$35.68	900
	\$35.66	350
	\$35.65	109
	\$35.60	792
	\$35.60	1247
	\$35.59	895
	\$35.57	670
	\$35.56	1480
	\$35.54	1020
	\$35.53	485
320	\$35.50	
621	\$35.49	
1340	\$35.48	
904	\$35.46	
550	\$35.43	
800	\$35.42	
96	\$35.41	
340	\$35.40	
512	\$35.39	
128	\$35.38	

Рис. 93: Биржевой стакан

Рассмотрим задачу вычисления максимальной цены финансовых инструментов за каждую секунду на основе потока сделок. Простейшим решением этой задачи на MapReduce является использование в качестве ключа пары (инструмент, секунда), в качестве значения — цены сделки. В `reduce` вычисляется максимум по всем сделкам данного инструмента за данную секунду.

В Hadoop MapReduce существует и другой способ решения этой задачи.

В качестве ключа можно использовать тройку (инструмент, секунда, цена сделки), в качестве значения — тип `NullWritable`, но при этом самостоятельно реализовать `Partitioner`, который будет распределять данные по reducer'ам, основываясь лишь на инструменте и секунде. Далее, передать в метод `setGroupingComparatorClass` класса `Job` реализацию компаратора, сравнивающего ключи также лишь на основе инструмента и секунды.

В результате в методе `reduce` первым или последним значением (в зависимости от реализации отношения `<` в `Comparator`) будет максимальное значение цены. Этот механизм в Hadoop MapReduce называется **вторичной сортировкой (Secondary Sort)**.

Также, возникает вопрос относительно обработки первого или последнего значения цены, соответствующей ключу, если по прототипу `reduce` значение имеет тип `NullWritable`. Используется следующая реализация: в процессе итерирования по `NullWritable`, переданным в функцию `reduce` в качестве значений, меняется объект, на который ссылается параметр-ключ.

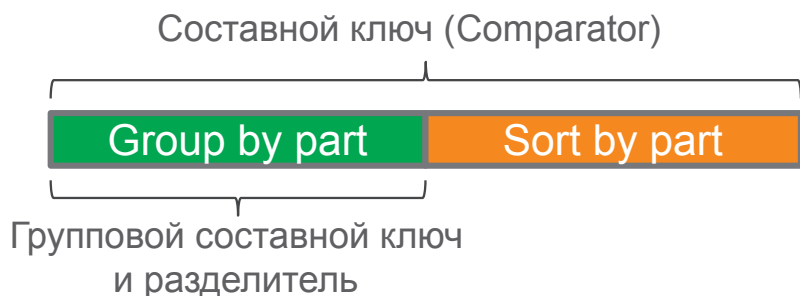


Рис. 94: Структура `Comparator`

На Рисунке 96 одинаковым цветом показаны ключи и значения, которые попадут в вызов `reduce`.

7.8.2 Соединение таблиц

Рассмотрим более прикладной пример применения вторичной сортировки. Пусть в одной таблице находятся сообщения пользователя, в другой —

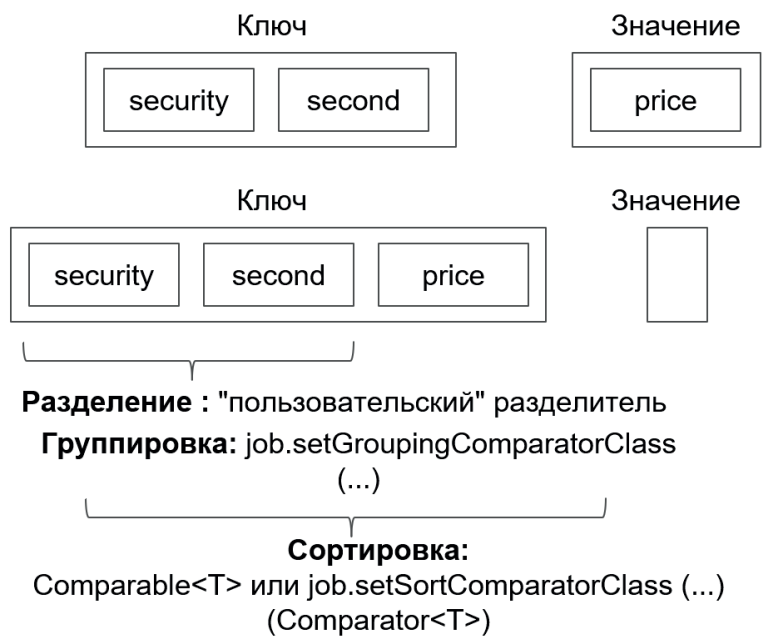


Рис. 95: Пример: максимальная цена каждой акции за каждую секунду

ключ			значение
USD/RUB	11:01:02	65.2	
USD/RUB	11:01:02	65.64	
USD/RUB	11:01:02	65.8	
USD/RUB	11:01:02	65.9	
USD/RUB	11:01:03	65.7	
USD/RUB	11:01:03	65.8	
USD/RUB	11:01:03	65.82	
USD/RUB	11:01:04	65.81	
USD/RUB	11:01:04	65.82	

Рис. 96: Пример: максимальная цена каждой акции за каждую секунду

данные профиля пользователя. Профили и сообщения находятся в отношении один ко многим. Требуется соединить эти таблицы.

В классической реализации соединения на стороне Reduce потребуется загрузить все значения в память, во внешнем цикле итерироваться по значениям из одной таблицы, во вложенном — по второй и на каждой итерации формировать строку-результат. Сообщений у пользователя очень много, и объема памяти может не хватить.

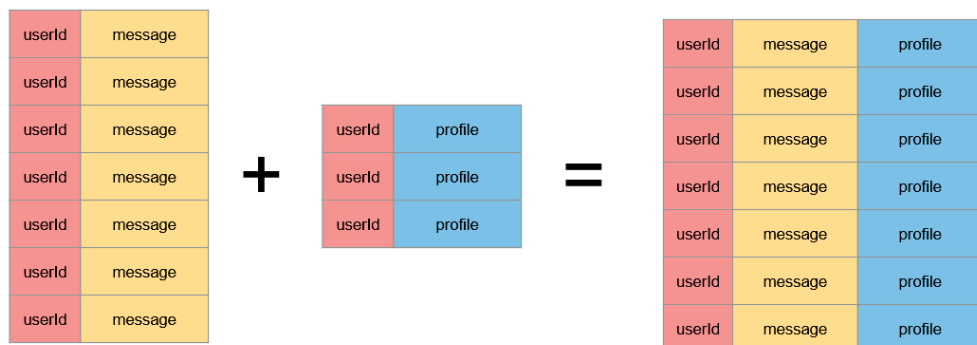


Рис. 97: Соединение сообщения с данными об авторе

Одним из решений является включение тега таблицы (например, со значениями 0 и 1) в ключ и использование вторичной сортировки, чтобы обеспечить попадание строки из таблицы профилей в качестве первого значения в reduce. Аналогичным образом можно использовать вторичную сортировку не только для соединения таблиц, находящихся в отношении “один ко многим”, но и таблиц, в одной из которых ключи повторяются реже, чем в другой.

7.9 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

- Что такое формат файла последовательности и когда он применяется?
- В какие моменты в модели MapReduce можно использовать компрессию данных?

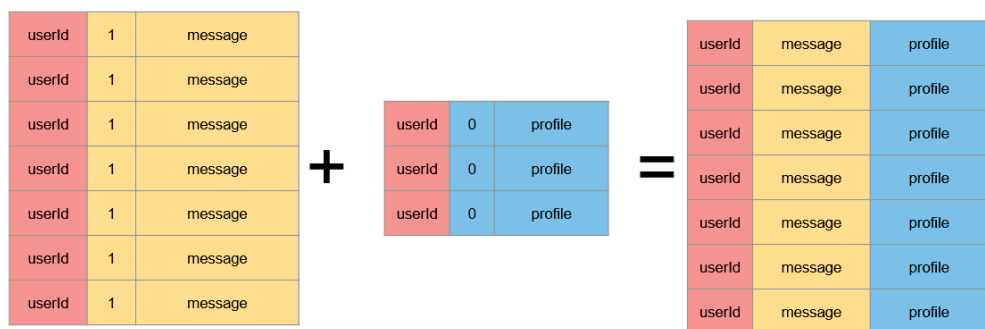


Рис. 98: Модифицированное соединение сообщения с данными об авторе

- Каким требованиям должны удовлетворять алгоритмы сжатия для эффективного использования в MapReduce?
- Что такое сбой в модели MapReduce?
- Что такое идемпотентность алгоритма?
- Что такое механизм счетчиков и для чего он используется?
- Что такое Secondary Sort и когда она может быть полезна?
- В каких ситуациях следует использовать свою реализацию Partitioner?

ГЛАВА 8

Планирование вычислений и отладка приложений

В данной главе рассматриваются вопросы планирования вычислений и отладки MapReduce-приложений в кластере Apache Hadoop.

8.1 Планировщик в Hadoop 1.0

В ранних версиях Apache Hadoop механизм **планирования вычислений** был организован достаточно просто. Планировщик был представлен единственной компонентой **JobTracker** (центральный управляющий элемент кластера) и множеством компонент **TaskTracker** (исполнительная компонента, размещаемая на каждом узле кластера), по одной компоненте TaskTracker на каждый узел. Пользовательские MapReduce-приложения направлялись на выполнение через JobTracker.

Данная компонента управляла распределением ресурсов между различными приложениями, запущенными в кластере, и одновременно координировала выполнение всех задач (map и reduce) для каждого конкретного приложения. Таким образом, JobTracker был ответственен и за планирование выделения ресурсов между приложениями, и за планирование запросов ресурсов в рамках каждого приложения.

Все ресурсы кластера резервировались статически в виде **слотов** — логических единиц вычислительных ресурсов. В то время основным (и фактически единственным) типом управляемого ресурса являлась оперативная

²⁸<https://edshare.gcu.ac.uk/4383/2/index.html>

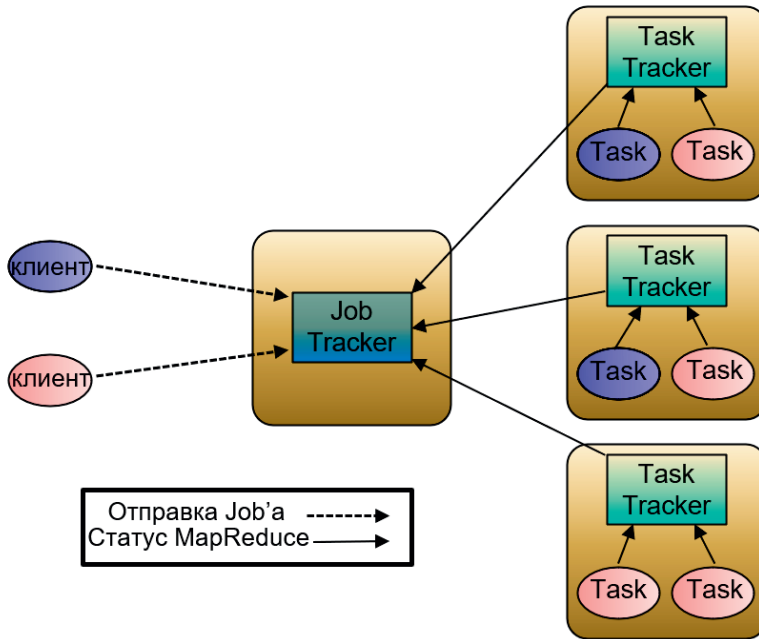


Рис. 99: Схема планировщика Hadoop 1.0 ²⁸. **JobTracker** ответственен за планирование одновременного вычисления нескольких MapReduce-приложений на кластере. Каждый **TaskTracker** управляет mappers и reducers на своем узле. Ресурсы кластера описывались **слотами** под задачи map или reduce

память. Слоты предназначались для запуска задач типов map и reduce.

За контроль использования ресурсов на каждом узле отвечал компонент TaskTracker. При наличии свободного слота TaskTracker сигнализировал об этом JobTracker. Если в системе имелись незапущенные задачи, JobTracker назначал одну из них на соответствующий узел. При этом приоритет отдавался map-задачам, а выбор узла для их выполнения учитывал локальность данных — предпочтение отдавалось узлам, на которых физически хранятся необходимые фрагменты входных данных. Очередность выполнения приложений управлялась с помощью FIFO-очереди (First-In-First-Out) с учетом системы приоритетов.

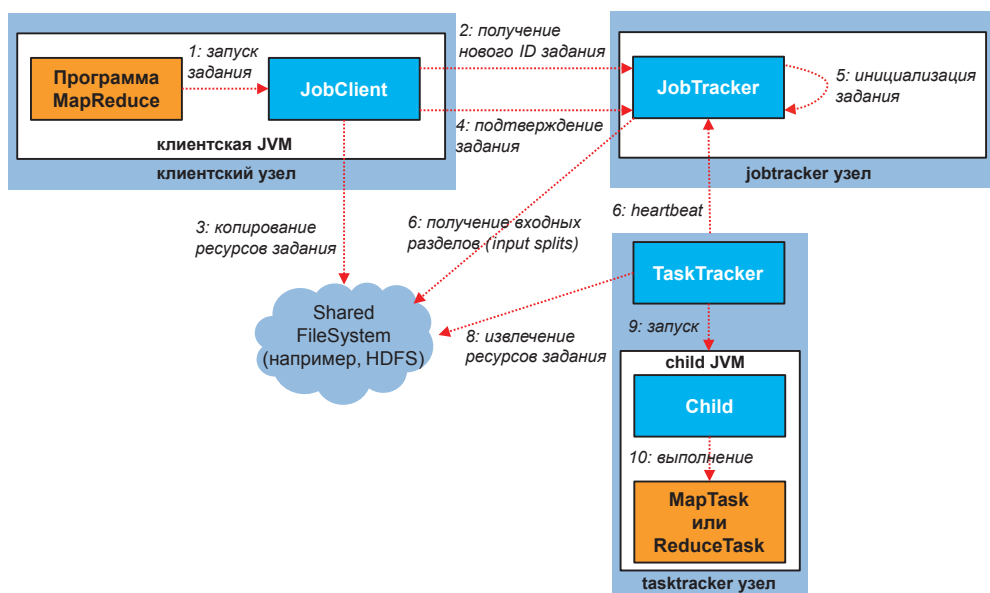


Рис. 100: Задачи MapReduce задания выполняются в рамках отдельных виртуальных машин Java, находящихся на узлах кластера [1]

Основным недостатком такой организации планирования вычислений является совмещение обязанностей по разграничению ресурсов кластера между приложениями и координацией исполнения задач отдельного приложения в единственной компоненте — JobTracker. Также отсутствовала возможность исполнения на том же кластере заданий, использующих модель вычислений, отличную от MapReduce.

8.2 Apache YARN (Hadoop 2.0)

В 2013 году была опубликована работа [10], описывающая новый планировщик Hadoop, который был назван Yet Another Resource Negotiator (**YARN**). Он вошел в состав 2-й версии Hadoop. YARN был призван решить недостатки старого планировщика, в частности, разделить обязанности по распределению ресурсов кластера между заданиями и координации исполне-

ния задач задания по разным компонентам, предоставить возможность исполнения на кластере не только MapReduce-приложений, а также эффективно планировать исполнение приложений с разными потребностями в ресурсах и при этом минимизировать архитектурные различия с Hadoop 1.0.

Основные требования, предъявляемые к системе YARN:

1. **Мультиарендность (multitenancy):** возможность одновременного выполнения множества заданий с различными требованиями к ресурсам. Планировщик должен оставаться масштабируемым при увеличении числа параллельных заданий;
2. **Поддержка различных моделей вычислений:** отказ от привязки исключительно к MapReduce и предоставление возможности запуска других типов приложений (например, графовые вычисления, потоковая обработка);
3. **Осведомленность о размещении данных:** планирование задач с учетом локальности данных, что позволяет минимизировать сетевой трафик и ускорить обработку;
4. **Эффективное использование ресурсов:** гибкое распределение доступных ресурсов между задачами с целью максимальной загрузки вычислительной инфраструктуры;
5. **Гибкая модель ресурсов:** возможность описания ресурсов не только в терминах памяти, но и других метрик (например, количество процессорных ядер);
6. **Высокая доступность и надежность:** устойчивость к сбоям компонентов управления и обеспечение непрерывности выполнения заданий;
7. **Безопасность:** контроль доступа, аудит действий и защита данных в условиях многопользовательского режима;

8. **Минимальные архитектурные изменения:** сохранение общих принципов организации выполнения заданий в кластере, без радикальной переработки всех компонентов системы, унаследованных от JobTracker в Hadoop MapReduce 1.0.

8.2.1 Структура YARN

Рассмотрим основные компоненты архитектуры YARN. Во-первых, центральный компонент **ResourceManager**, ответственный за глобальное управление ресурсами кластера. Он занимается распределением доступных ресурсов между всеми приложениями и не учитывает особенности их вычислительной модели. Во-вторых, компонент **ApplicationMaster**, отвечающий за управление жизненным циклом конкретного приложения. Его задачи: отправка запросов в ResourceManager на выделение ресурсов и координация запуска и выполнения задач. Для каждого запущенного приложения создается экземпляр ApplicationMaster (позволяет изолированно управлять различными типами приложений). Для каждой вычислительной модели используется собственная реализация ApplicationMaster (например, для MapReduce-приложений используется MRApplicationMaster (MRAppMaster)). Наконец, агент **NodeManager**, запускаемый на каждом узле кластера, аналогичен TaskTracker в Hadoop 1.0. NodeManager управляет вычислительными контейнерами, следит за их состоянием, а также проводит мониторинг и формирует отчетность о локальном использовании ресурсов (оперативной памяти, процессорного времени и т.д.).

Компоненты YARN взаимодействуют между собой при помощи **heartBeat-сообщений**: периодических обращений компонентов друг к другу с фиксированным временным периодом. Важно отметить, что этот процесс сказывается на производительности планировщика.

Взаимодействие ApplicationMaster и ResourceManager основывается на запросах ресурсов. ApplicationMaster, реализующий логику новой модели вы-

²⁹<https://edshare.gcu.ac.uk/4383/2/index.html>

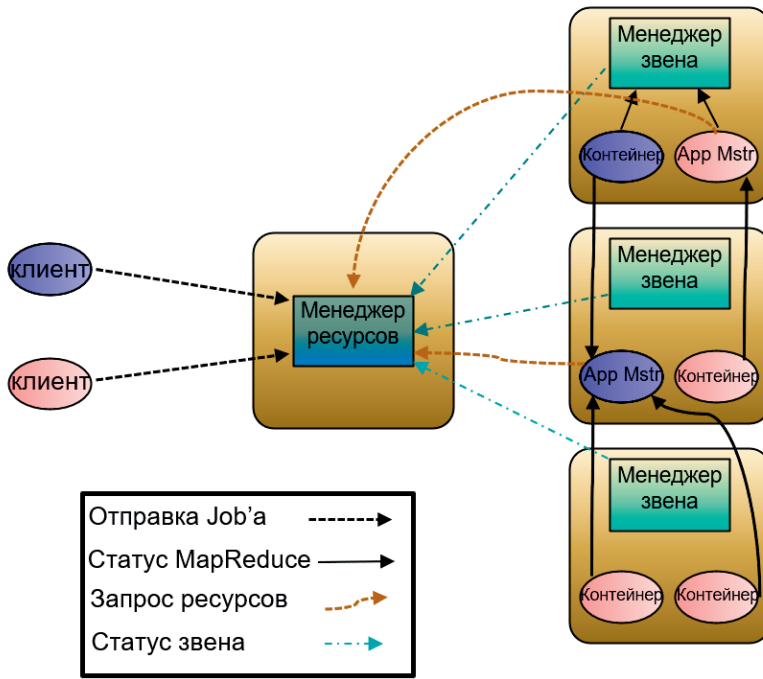


Рис. 101: Управление вычислениями — YARN+MRAppMaster²⁹. **ResourceManager** (Менеджер ресурсов, 1 шт. на кластер) — планировщик, распределяющий ресурсы кластера по запросам AppMaster'ов. **NodeManager** (Менеджер звена, 1 шт. на узел) создает и производит мониторинг “контейнеров” своего узла. **AppMaster** реализует логику распределенных вычислений своего типа (например, **MRAppMaster** для MapReduce). Для каждого запускаемого приложения создается отдельный AppMaster нужного типа. **Container** — контейнер: ресурс (ОЗУ, ядра и др.), выделенный приложению

числений, должен быть написан на языке программирования, исполняемом на Java-машине. Задания, исполняемые приложениями в рамках этой модели вычислений, могут быть любыми исполняемыми модулями или скриптами (не обязательно Java-приложениями).

Ключевые характеристики YARN:

- Модель запросов (request-based, pull-модель). ApplicationMaster самостоятельно отправляет запросы на получение необходимых ресурсов в ResourceManager;
- Изолированное управление выполнением заданий. Каждое задание выполняется отдельным ApplicationMaster с его моделью вычислений (например, MRApplicationMaster);
- Механизм периодического обмена сообщениями (heartBeat).

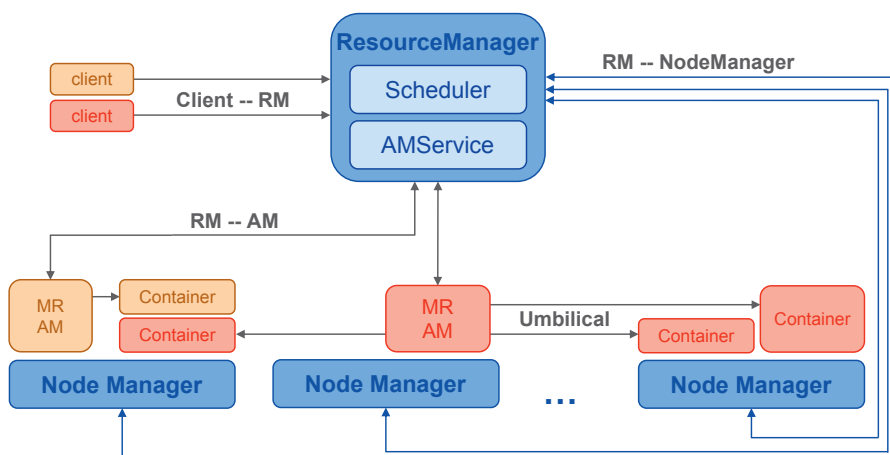


Рис. 102: Архитектура YARN. Темным цветом отмечены системные компоненты, светлым — два запущенных приложения [10]

8.2.2 Запрос на выделение ресурсов

Запрос ресурсов содержит требования к ресурсам и число единиц ресурса. Также в запросе могут быть указаны предпочтения по размещению ресурсов (в виде указания хоста или стойки). Каждая единица ресурса называется **контейнером**. Если ResourceManager принимает решение выделить ресурсы данному заданию, в ответе на запрос он возвращает число переданных

приложению контейнеров (может быть меньше запрошенного) и их местоположение (имена узлов, на которых выделены контейнеры).

Основные параметры запроса:

- **Требования к размещению:** указание предпочтительного хоста, стойки или других топологических параметров;
- **Приоритет внутри приложения:** возможность задания важности различных задач (например, контейнеры для задач map могут иметь более высокий приоритет, чем для reduce);
- **Объем оперативной памяти (RAM):** минимально необходимый объем памяти на один контейнер;
- **Количество процессорных ядер (CPU):** необходимое число ядер для исполнения задач в контейнере;
- **Дополнительные ресурсы (опционально):** доступ к локальному диску, графическим процессорам (GPU) и т.п.;
- **Количество контейнеров:** общее число единиц ресурса, запрашиваемых в текущем запросе.

8.2.3 Настройки памяти JVM

Задачи (mapper и reducer) исполняются виртуальной машиной Java, поэтому основными характеристиками контейнера являются параметры памяти JVM: размеры стека функций и кучи. В Таблице 18 показаны конфигурационные свойства YARN. Они характеризуют общий объем ресурсов кластера и размеры партий, которыми их можно запрашивать. Также отображены требования конкретного MapReduce-приложения к контейнеру для запуска одного mapper или reducer.

Ядра в YARN контролируются с помощью **Linux cgroup** — это механизм виртуализации ресурсов в рамках ОС, позволяющий контролировать распределение ресурсов (память, CPU) в рамках групп процессов.

Таблица 18: Конфигурационные свойства YARN

Название параметра	По умолчанию	Описание
mapreduce.framework.name	local	Модель управления вычислениями (local, yarn, classic)
yarn.resourcemanager. .address	host : 8032	Хост:порт менеджера ресурсов YARN
yarn.scheduler.minimum- allocation-mb	1024	Минимальный объем памяти для контейнера
yarn.scheduler.maximum- allocation-mb	8192	Максимальный объем памяти для контейнера
yarn.scheduler.minimum- allocation-vcores	1	Минимальное число ядер для контейнера
yarn.scheduler.maximum- allocation-vcores	32	Максимальное число ядер для контейнера
yarn.nodemanager.resource. .memory-mb	8192	Объем памяти NodeManager
yarn.nodemanager.resource. .cpu-vcores	32	Число ядер NodeManager
mapreduce.map.memory.mb	1024	Объем памяти для mapper
mapreduce.map.cpu.vcores	1	Число ядер для mapper
mapreduce.reduce. .memory.mb	1024	Объем памяти для reducer
mapreduce.reduce. .cpu.vcores	1	Число ядер для reducer

8.2.4 Спекулятивное исполнение

Одна из возможностей YARN — **спекулятивное исполнение**. При активации этой функции долгое выполнение задачи запускает ее дубль. По завершению выполнения задачи или ее дубля оставшаяся запущенная задача

уничтожается. Спекулятивное исполнение позволяет оптимизировать время исполнения заданий на ненадежном оборудовании, склонном к деградации. Стоит отметить, что запуск дублирующей задачи в случае медленного выполнения исходной задачи не повышает надежность, но оптимизирует время исполнения в условиях ненадежного оборудования.

Таблица 19: Конфигурационные свойства спекулятивного исполнения

Название параметра	По умолчанию	Описание
mapred.map.tasks. .speculative.execution	true	Спекулятивное исполнение для map
mapred.reduce.tasks. .speculative.execution	true	Спекулятивное исполнение для reduce
yarn.app.mapreduce. .am.job.speculator.class	DefaultSpeculator	Стратегия спекулятивного исполнения
yarn.app.mapreduce. .am.job.estimator.class	LegacyTask RuntimeEstimator	“Оцениватель” времени исполнения

8.3 Планирование исполнения программ в многопользовательском кластере

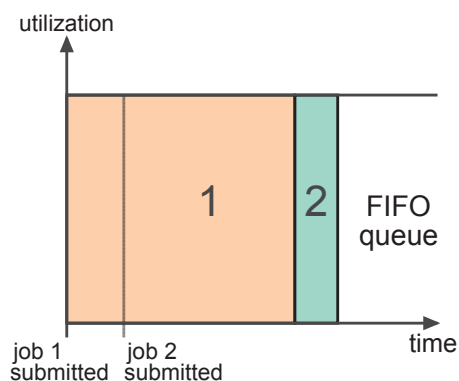
8.3.1 Алгоритмы планирования

Рассмотрим возможные алгоритмы планирования исполнения приложений в YARN:

- Очередь программ (FIFO — First-In-First-Out);
- Очередь с приоритетами (mapred.job.priority);
- Capacity Scheduler;
- Fair Scheduler.

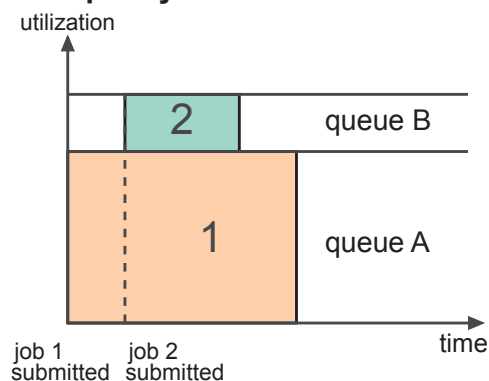
YARN унаследовал от планировщика Hadoop 1.0 алгоритмы планирования типа **очередь** и **очередь с приоритетами**. При использовании очередей задания выполняются на кластере последовательно. Однако, даже при наличии приоритетов, возможна ситуация, при которой выполнение небольшого задания значительно задерживается из-за занятости кластера ресурсоемким процессом, начатым ранее.

i. FIFO Scheduler



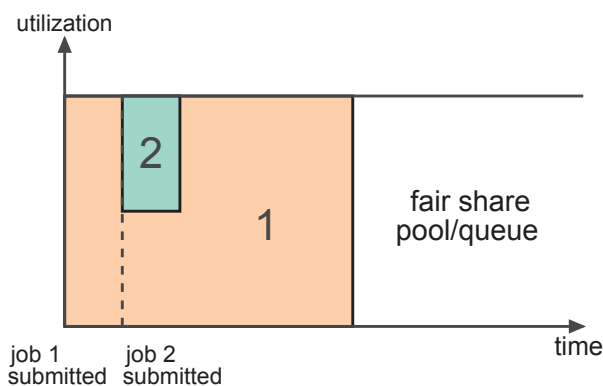
(a) FIFO Scheduler

ii. Capacity Scheduler



(b) Capacity Scheduler

iii. Fair Scheduler



(c) Fair Scheduler

Рис. 103: Алгоритмы планирования исполнения приложений в YARN [1]

Новыми алгоритмами планирования в YARN стали **Capacity Scheduler** и **Fair Scheduler**. Оба используют модель очередей заданий, в которой часто

отдельным очередям сопоставляются конкретные пользователи. **Capacity Scheduler** распределяет ресурсы кластера пропорционально заранее заданным квотам (вместимостям), а **Fair Scheduler** стремится обеспечить равномерное распределение ресурсов между активными очередями.

8.3.2 Capacity Scheduler

В Capacity Scheduler очереди образуют иерархию, соответствующую группам пользователей кластера, проектам или системам, инициирующим запуск приложений. Каждой очереди статически назначается `capacity`. Очередь может получить больше ресурсов, чем ее `capacity`, если ресурсы в данный момент свободны. Насколько больше — определяет свойство `max capacity`. Очередь может быть приписана отдельному пользователю кластера или нескольким пользователям. Внутри очереди задания выполняются согласно FIFO, но существует и Fair-стратегия — возможность параллельного исполнения заданий внутри очереди.

Свойства Capacity Scheduler:

- Иерархия очередей (`queue`);
- Статическое распределение `capacity` между очередями. Очередь может получить больше ресурсов, чем ее `capacity`, если они свободны (насколько много — определяет свойство `max capacity`);
- Ordering policies заданий внутри очереди: FIFO (с приоритетами) или Fair;
- Вытеснение (`preemption`) контейнеров между очередями или между заданиями внутри очереди (основываясь на приоритетах или равномерном распределении ресурсов между заданиями);
- Резервирование ресурсов (`resource reservation`) — возможность гарантировать приложению фиксированный объем ресурсов (в единицах выделения ресурсов);

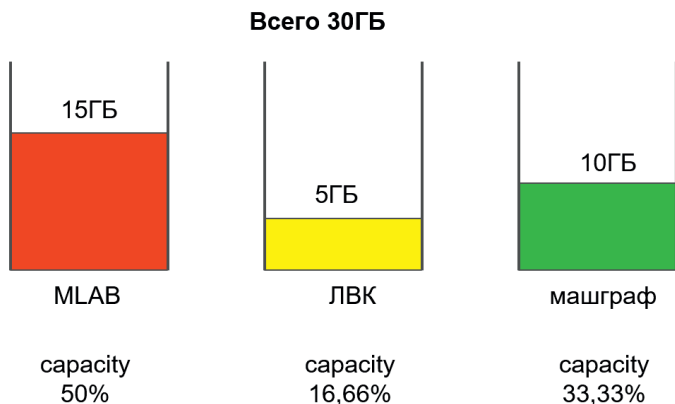


Рис. 104: Пример распределения вместимости (*capacity*)

- Разграничение прав на управление заданиями на уровне очередей, отображение пользователей ОС на очереди.

Рассмотрим пример, иллюстрирующий понятия вместимости (*capacity*) и максимальной вместимости (*max capacity*) очередей (см. Рисунок 104). Предположим, что очередям MLAB, LBK и машграф выделены следующие доли *capacity* кластера: 50%, 16,66% и 33,33% соответственно. В пересчете на объем оперативной памяти это составляет 15 ГБ, 5 ГБ и 10 ГБ.

Если *max capacity* очередей — 70%, 20%, 90%, то очередь машграф может занять 20 ГБ ОЗУ вместо указанной *capacity* 10 ГБ, если в других очередях есть 10 ГБ свободной памяти.

Иерархия очередей, вместимости и максимальные вместимости очередей определяются в конфигурационном файле.

```
<configuration>
  <property>
    <name>yarn.scheduler.capacity.root.queues</name>
    <value>lvk,mlab</value>
  </property>
  <property>
```

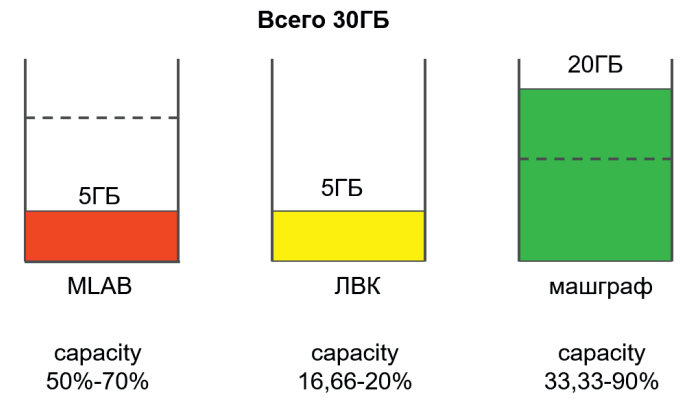


Рис. 105: Пример вместимости (capacity) и максимальной вместимости (max capacity) очереди

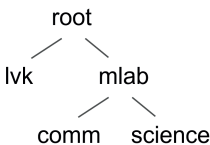


Рис. 106: Пример иерархии очередей в Capacity Scheduler

```
<name>yarn.scheduler.capacity.root.mlab.queues</name>
<value>comm,science</value>
</property>
<property>
  <name>yarn.scheduler.capacity.root.lvk.capacity</name>
  <value>40</value>
</property>
<property>
  <name>yarn.scheduler.capacity.root.mlab.capacity</name>
  <value>60</value>
</property>
```

```
<property>
  <name>yarn.scheduler.capacity.root.lvk.maximum-capacity
</name>
  <value>75</value>
</property>
<property>
  <name>yarn.scheduler.capacity.root.mlabs.comms.capacity
</name>
  <value>50</value>
</property>
<property>
  <name>yarn.scheduler.capacity.root.mlabs.science.capacity
</name>
  <value>50</value>
</property>
</configuration>
```

Определение задачи к очереди осуществляется с помощью `mapreduce.job.queueName`.

8.3.3 Fair Scheduler

Развитием Capacity Scheduler стал Fair Scheduler. Его основной концепцией является равномерное разделение одного ресурса (например, ОЗУ) между очередями, в которых в данный момент запущены задания.

Dominant Resource Fairness (DRF) — это принцип справедливости, который распространен на ресурсы нескольких типов. Справедливость в DRF может соблюдаться при распределении ресурсов как между очередями, так и внутри очередей. Внутри очередей также допустимо FIFO-планирование. Очередям можно назначать веса, пропорционально которым между ними будут разделяться ресурсы.

К свойствам Fair Scheduler относятся равномерное распределение ресурсов между активными очередями в соответствии с весами (нет жестких capacity) и гарантия ресурсов по минимальной вместимости очереди.

Рассмотрим **пример равномерного распределения ресурсов между очередями**. Пусть есть две очереди: MLAB и ЛВК. При запросе ресурсов заданиями из этих очередей контейнер Fair Scheduler будет выделять ресурсы очередям равномерно.

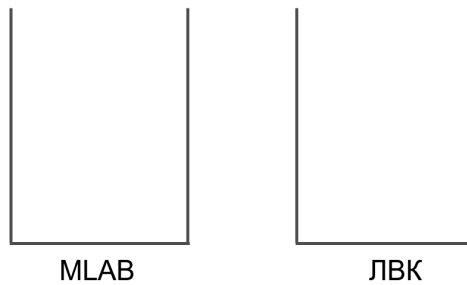


Рис. 107: Пример выделения ресурсов для двух очередей (часть 1)

Случайным образом выбираем очередь MLAB. Сначала ресурсы будут выделены ей.

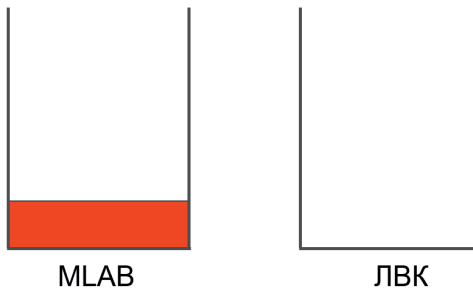


Рис. 108: Пример выделения ресурсов для двух очередей (часть 2)

Затем ЛВК.

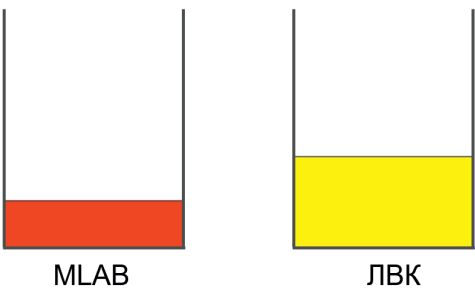


Рис. 109: Пример выделения ресурсов для двух очередей (часть 3)

Планировщик обеспечивает равномерное распределение ресурсов между очередями.

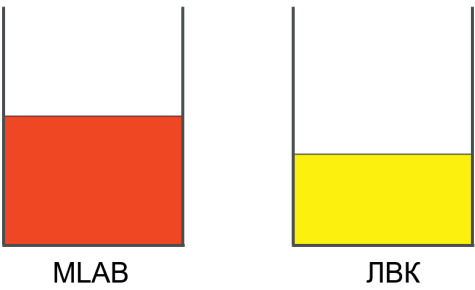


Рис. 110: Пример выделения ресурсов для двух очередей (часть 4)

Продолжение распределения.

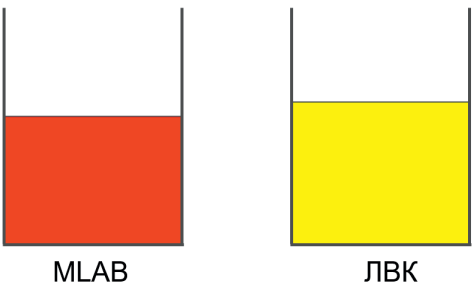


Рис. 111: Пример выделения ресурсов для двух очередей (часть 5)

Для очереди можно определить **минимальную вместимость**. Если в данный момент очередь MLAB получила меньше `min capacity` ресурсов, а остальные очереди уже используют свои минимальные объемы, планировщик будет выделять ресурсы MLAB до достижения `min capacity`.

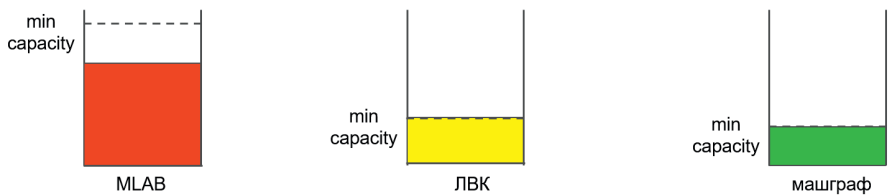


Рис. 112: Пример первоначальных ресурсов для трех очередей

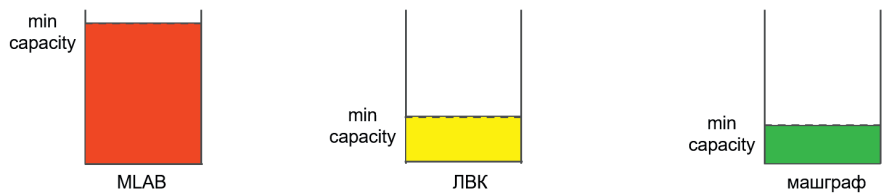


Рис. 113: Пример ресурсов перед равномерной раздачей

Только после этого будет реализовываться *fairness* (справедливость) для всех очередей.

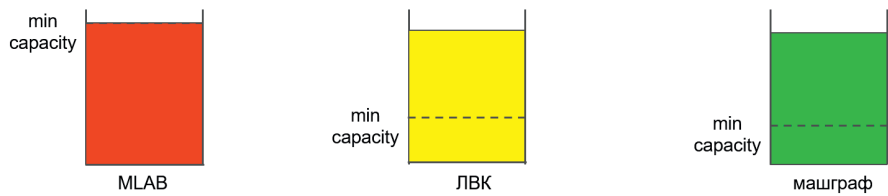


Рис. 114: Пример ресурсов очередей после равномерной раздачи

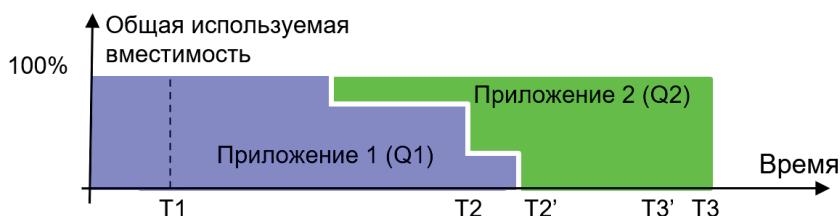
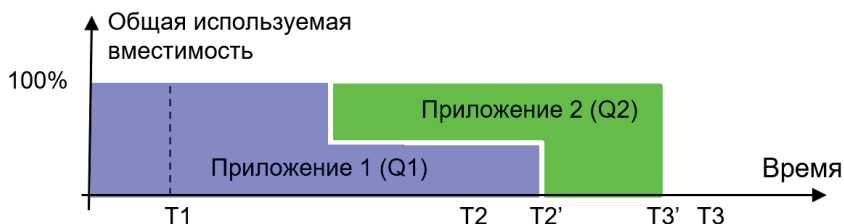
Когда вытеснение не включено**Когда вытеснение включено**

Рис. 115: Пример работы с включенным/выключенным механизмом вытеснения

8.3.4 Вытеснение

Вытеснение (preemption) — это механизм принудительного освобождения ресурсов (контейнеров), выделенных другим очередям, с целью достижения справедливого распределения ресурсов в кластере. Данный механизм может быть отдельно настроен для обеспечения минимальной гарантированной доли ресурсов (`minShare`) и для соблюдения справедливой доли (`fairShare`).

Несмотря на преимущества справедливого планировщика, новое приложение не может сразу получить ресурсы и вынуждено ожидать освобождения хотя бы части контейнеров текущих заданий. В таких ситуациях применяется механизм вытеснения: если за заданное время очередь не получает положенную ей долю ресурсов, планировщик может инициировать освобождение контейнеров других очередей, чтобы перераспределить ресурсы в пользу нуждающейся.

Для активации механизма вытеснения в планировщике Fair Scheduler необходимо задать соответствующие параметры конфигурации:

- `yarn.scheduler.fair.preemption=true` — включает механизм вытеснения в планировщике. По умолчанию значение параметра — `false`;
- Тайм-ауты вытеснения (в секундах) определяют, через какое время можно инициировать вытеснение ресурсов, если доля очереди не удовлетворяет заданным условиям. Например: `minSharePreemptionTimeout` — минимальное время для ожидания гарантированной доли ресурсов (`minShare`). Аналогично, `fairSharePreemptionTimeout` — параметр для справедливой доли ресурсов (`fairShare`).

8.3.5 Отложенное планирование

Отложенное планирование (delay scheduling) — механизм, позволяющий отложить принятие решения по запросу ресурсов (с учетом требований к местоположению контейнеров) до освобождения ресурсов с нужной локальностью.

В модели MapReduce выполнение задачи зачастую требует, чтобы контейнеры, выделенные для таргет, находились ближе к целевым данным. Поэтому в запросе ресурсов могут быть указаны пожелания по размещению контейнеров на конкретных хостах или стойках (task-aware). Если в момент запроса такие ресурсы недоступны, YARN может предложить доступные контейнеры без учета локальности. Однако передача данных по сети приводит к задержкам. Для минимизации накладных расходов на передачу данных и повышения эффективности используется механизм отложенного планирования.

Суть подхода: если в течение заданного периода времени в кластере освободятся ресурсы, удовлетворяющие требованиям по локальности, они будут предоставлены приложению. В противном случае, по истечении тайм-аута,

YARN выделит доступные контейнеры, даже если они не соответствуют предпочтениям по размещению.

Таблица 20: Параметры отложенного планирования

Свойство	Описание	Пример значения
yarn.scheduler.fair. locality.threshold. node	Ожидание ресурса с определенным местоположением пока не пройдут yarn.scheduler.fair. \ locality.threshold.node · <число узлов> heartBeat'ов	0.5

8.4 Основные типы планировщиков

Рассмотрим основные архитектурные подходы к организации планировщиков вычислений в распределенных системах [11]:

- **Монолитный планировщик** — централизованный компонент, отвечающий за планирование задач одного типа. Такой планировщик обладает полной информацией о ресурсах кластера и принимает решения глобально. Применим в системах с однородной моделью вычислений и ограниченным числом пользователей.
- **Двухуровневый планировщик (two-level scheduling)** основан на планировщиках разных моделей вычислений, которые принимают решения без знания текущих ресурсов кластера. На первом уровне происходит распределение ресурсов между приложениями разных типов, а на втором уровне планируются вычисления внутри приложения. Второй уровень не обладает полной картиной доступных ресурсов кластера. Уровни общаются друг с другом с помощью предложений ресурсов

(планировщик ресурсов 1-го уровня последовательно предлагает приложениям некоторые подмножества ресурсов);

- **Планировщик с общим состоянием (shared-state scheduling)** — архитектура, в которой несколько планировщиков (каждый — для своей модели вычислений) работают параллельно, обладая доступом к общей и полной информации о текущем состоянии ресурсов кластера. Конфликтующие запросы обрабатываются централизованным механизмом арбитража, который принимает решения в соответствии с политикой планирования (например, на основе приоритетов, долей или fairness-модели).

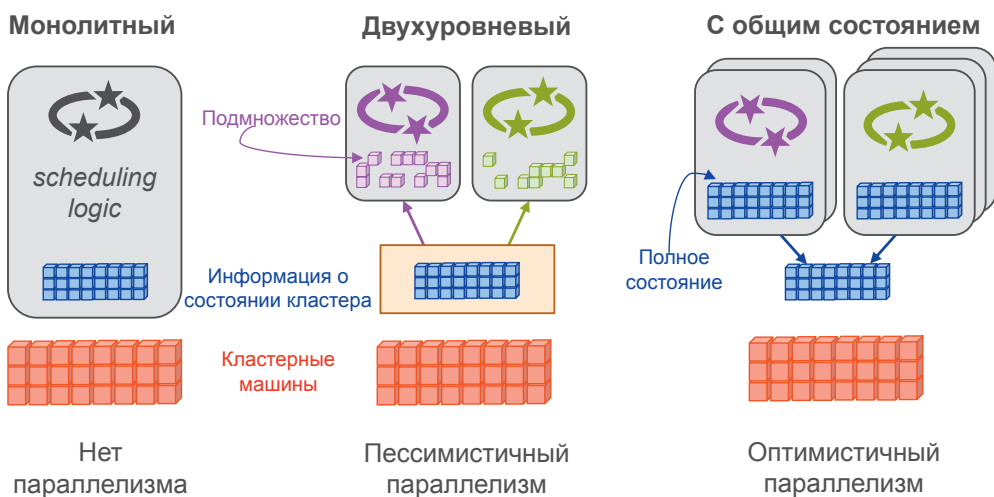


Рис. 116: Схемы работы разных видов планировщиков (часть 1) ³⁰

YARN, вероятно, находится ближе к двухуровневому планировщику, однако в нем используется не push-модель предложений ресурсов, а pull, основанная на запросах. JobTracker из Hadoop 1.0 является монолитным планировщиком.

³⁰<https://www.oreilly.com/library/view/programming-kubernetes/9781492047094/ch01.html>

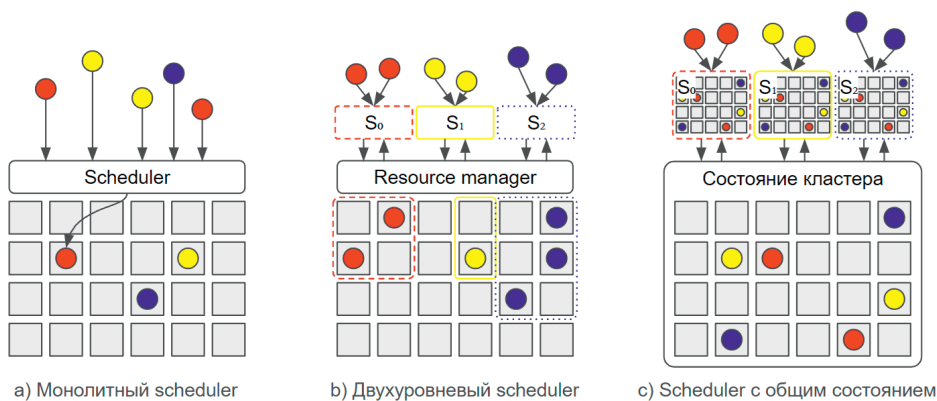


Рис. 117: Схемы работы разных видов планировщиков (часть 2) [11]

8.5 Mesos

Альтернативой YARN'у является планировщик **Mesos** [12]. Он является двухуровневым планировщиком, основанным на предложениях ресурсов. Компонента **master** формирует предложения ресурсов (**resource offers**) приложениям последовательно, в порядке, определяемом алгоритмом планирования. Приложение может принять предложение или отвергнуть его. Компоненты **slave** отслеживают освобождение ресурсов на узлах кластера.

На Рисунке 118 представлена структура планировщика Mesos. **Master** отслеживает освободившиеся ресурсы кластера и предлагает (**resource offer**) их приложениям. **Приложение (framework)** — быстрая, пакетная или фоновая задача некоторого типа (например, MPI, Hadoop). Каждый тип приложений обладает своим планировщиком (**scheduler**) и исполнителем (**executor**). **Slave** — координирует (запускает и отслеживает) ресурсы, занятые задачами на своем узле. **Задача (task)** — единица работы приложения. **Standby master** — резервные узлы master.

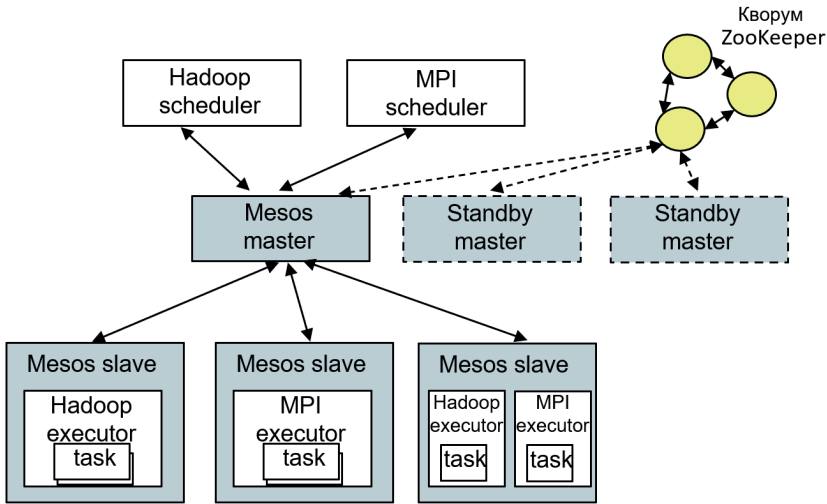


Рис. 118: Структура планировщика Mesos [12]

8.5.1 Resource offer

Предложение (Resource offer) содержит спецификацию ресурсов, которые предлагаются приложению. Приложение может задать требования к интересующим его ресурсам и получать только те предложения, которые удовлетворяют этим требованиям.

Основные концепции предложений:

- Происходят последовательно (pessimistic concurrency);
- Содержат информацию о свободных ресурсах и предлагаемое приложению количество ресурсов;
- Приложение решает, какие ресурсы занять, или отказывается от предложения;
- Master динамически определяет последовательность, в которой ресурсы предлагаются приложениям, и сами предлагаемые ресурсы;

- Фильтры позволяют приложению задать требования на offer, которые ему интересны.

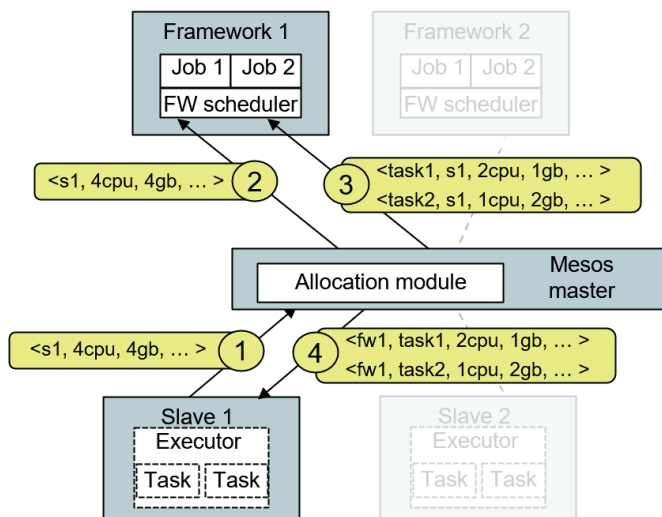


Рис. 119: Работа механизма предложения (Resource offer) в Mesos [12]

8.5.2 Dominant Resource Fairness

При создании Mesos был разработан алгоритм справедливого планирования выделения разнотипных ресурсов³¹ [13]. На многих промышленных Надоор-кластерах приложения запрашивают контейнеры с разной пропорцией ядер и оперативной памяти. В таком случае нужно определить условия справедливого планирования.

К слову, планирование предоставления контейнеров с несколькими типами ресурсов отдаленно напоминает выбор способа заполнения полки коробками нескольких разных размеров. В такой задаче необходимо решить, сколько широких, но низких коробок поставить на полку, а также сколько узких, но высоких. Одно из решений данной задачи реализовано в **Dominant**

³¹<https://people.eecs.berkeley.edu/~lig/papers/df.pdf>

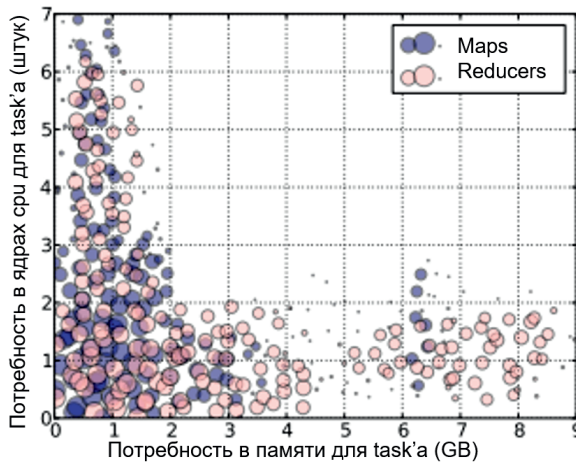


Рис. 120: Пример распределения потребностей для maps и reducers [13]

Resource Fairness. Рассмотрим принцип его работы на следующих примерах.

Пример 1

Пусть приложения А и Б отправили запрос на получение 30 контейнеров со своими требованиями к памяти и ядрам (см. Рисунок 121). Рассмотрим, сколько контейнеров получит каждое из них согласно DRF.

Для каждого приложения вычисляется **доминантный ресурс** — ресурс, доля которого в контейнере относительно общего объема ресурса в кластере максимальна. У приложения А доминантный ресурс контейнера — ОЗУ. Он составляет 3% от всего ОЗУ кластера. У приложения Б доминантный ресурс контейнера — ядра, они составляют 6% от всех ядер кластера. Согласно DRF, приложение Б получит в $6\%/3\%$ (в 2) раза меньше контейнеров, чем А. То есть, 10.

Пример 2

Рассмотрим еще один пример. В кластере имеется 4 CPU и около 15 ГБ памяти. Появилось приложение F1, оно сразу получает предложение всех имеющихся ресурсов. Приложение F1 занимает 2 CPU и 128 МБ памяти.

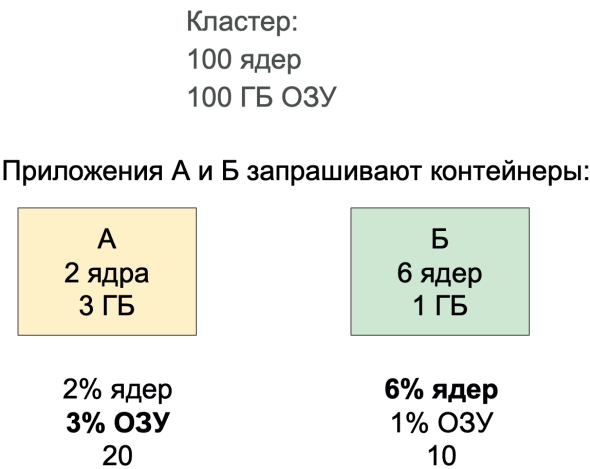


Рис. 121: Пример распределения ресурсов (Dominant Resource Fairness) [13]

Доминантный ресурс приложения F1 занимает 50% кластера. Появляется приложение F2, у него еще нет ресурсов, поэтому предложение оставшихся ресурсов кластера приходит ему. F2 соглашается на 1.5 CPU и 400 МБ памяти. Его доминантный ресурс — CPU занимает 37,5% данного ресурса в кластере.

Предложение повторно отсылается F1, но F1 его отвергает. Далее, предложение приходит в F2, F2 также его отвергает. Появляется F3. Только появившись, оно отнимает 0% ресурсов кластера.

Таблица 21: Пример работы DRF (4 CPUs и 14922 МБ доступно)

Framework	Arrive time	Requested CPU	Requested memory	Duration
F1	t0	2 CPUs	128 MB	25 secs
F2	t0+5	1.5 CPUs	400 MB	20 secs
F3	t0+11	1.8 CPUs	300 MB	20 secs

Теперь в первую очередь F3 получит это предложение, но отвергнет его. Вторым по доле использования ресурсов кластера (37,5%) является F2, оно

получает предложение и также его отвергает. После чего предложение отсылается приложению F1 и т.д.

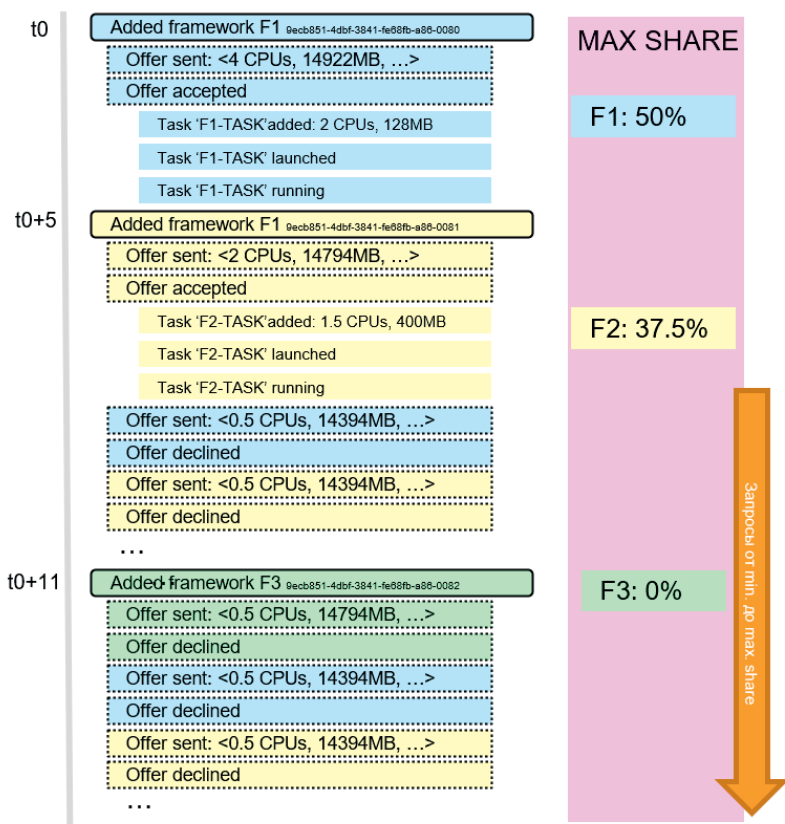


Рис. 122: Пример работы DRF для трех приложений F1, F2, F3 (часть 1)

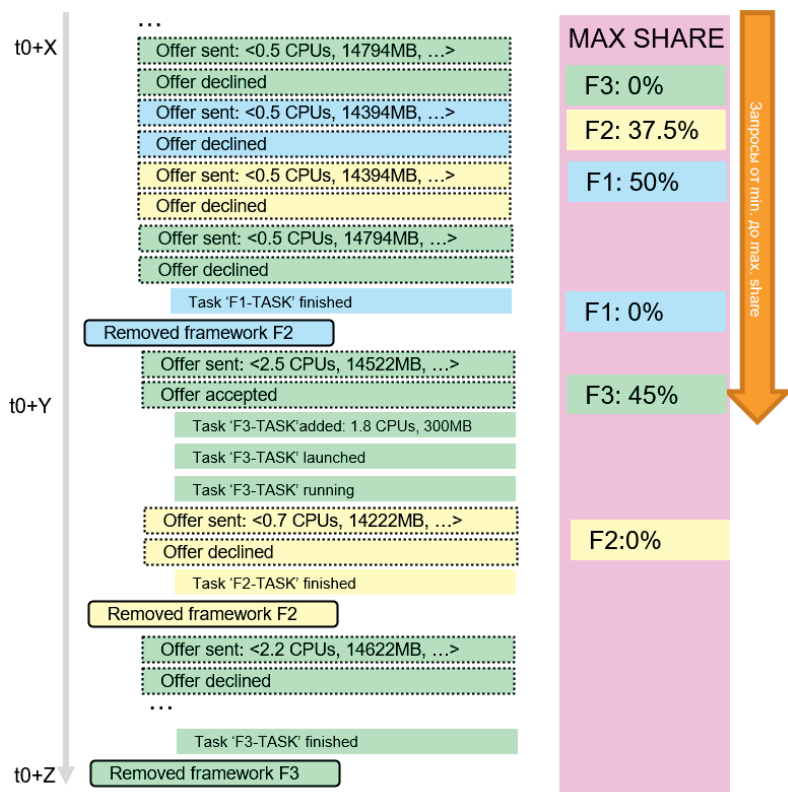


Рис. 123: Пример работы DRF для трех приложений F1, F2, F3 (часть 2)

Основная идея состоит в том, что предложения приходят приложениям в порядке, соответствующем долям, которые занимают доминантные ресурсы приложения в кластере. Приложение с наименьшей на данный момент долей своего доминантного ресурса в кластере получит очередное предложение.

8.5.3 YARN vs Mesos

В Таблице 22 представлено сравнение планировщиков YARN и Mesos. Оба планировщика являются двухуровневыми. Mesos появился до YARN и изначально разрабатывался как планировщик, не привязанный к какой-либо модели вычислений. Планировщики отличаются моделью взаимодействия с

приложениями: **pull** в YARN и **push** в Mesos, способом коммуникации: в YARN это небыстрый по своей сути heartBeat, в Mesos — remote procedure call. Mesos написан на C++, YARN — на языке реализации Hadoop — Java. Таким образом, Mesos быстрее и обладает более широкими возможностями. YARN больше ориентирован на Hadoop MapReduce, и именно он входит в большинство дистрибутивов и облачные реализации Hadoop.

Таблица 22: Сравнение планировщиков YARN и Mesos

Планировщик	YARN	Mesos
Концепция	Масштабируемая замена планировщику из MR 1.0, обеспечивающая планирование не только MR-заданий	Планировщик выполнения на кластере приложений разных типов (разные модели вычислений, короткие, фоновые, пакетные и т.д.)
Тип	двухуровневый	двухуровневый
Язык реализации	Java	C++
Год создания	2013	2009
Модель получения ресурсов	resource request (pull)	resource offer (push)
Способ коммуникации	heartBeat	RPC

8.6 Отладка MapReduce-приложений

Основным средством отладки и автоматического тестирования MapReduce-приложений является просмотр файлов журналов приложения: в нем содержатся события фреймворка MapReduce и сообщения, сформированные самим приложением в потоках стандартного вывода и ошибок. Отслеживать прогресс исполнения приложения (отслеживать занятость

узлов кластера, текущую стадию вычислений) можно в пользовательском интерфейсе (UI) YARN.

Перед запуском на кластере можно провести проверку корректности работы программы на локально развернутом Hadoop или на кластере, но на небольшом объеме данных.

При желании можно использовать unit-тесты, реализовать интеграционное и системное тестирование с помощью классов `MiniMRCluster`, `MiniYARNCluster`, `MiniDFSCluster`, позволяющих из кода создавать временный локальный Hadoop-кластер в виде объекта в оперативной памяти.

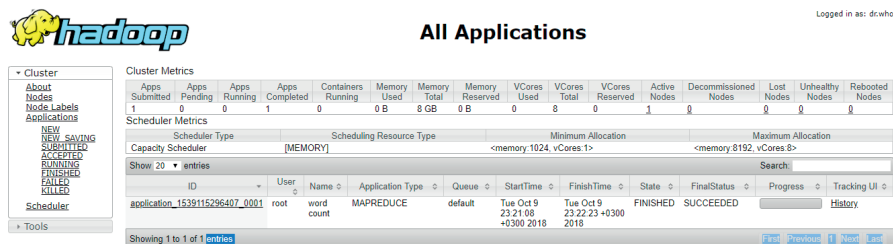


Рис. 124: Web YARN UI

В YARN UI можно увидеть прогресс исполнения своих заданий, посмотреть на содержимое журналов приложения. Также в UI визуализируется текущая занятость кластера другими пользователями.

8.6.1 Журналы

Во время работы приложения его журналы (а именно, журналы контейнеров, созданных приложением) находятся в локальных файловых системах соответствующих узлов кластера. После завершения работы приложения механизм агрегации журналов переносит журналы в HDFS в формате файла последовательности. Журналы доступны через команду `yarn logs` с указанием идентификатора приложения либо в UI History Server. Имеются следующие типы журналов: Stdout, Stderr и Syslog (log4j).

Журналы хранятся в папке локальной ФС узлов:

```
userlogs/  
userlogs/application_1389385968629_0025  
  
// In userlogs/application_1389385968629_0025  
container_1389385968629_0025_01_000001  
container_1389385968629_0025_01_000001/stdout  
container_1389385968629_0025_01_000001/stderr  
container_1389385968629_0025_01_000001/syslog  
container_1389385968629_0025_01_000002  
container_1389385968629_0025_01_000002/stdout  
container_1389385968629_0025_01_000002/stderr  
container_1389385968629_0025_01_000002/syslog
```

Журналы агрегируются в HDFS в бинарном формате после завершения работы задания:

```
/app-logs/<user>/logs/<applicationId>  
yarn logs -applicationId application_1446656107130_0006
```

Идентификация

В журналах отдельно идентифицируются: задание (`job`); задача, относящаяся к заданию (`task`); попытка запуска задачи (`attempt`).

```
job_201711011000_0001  
task_201711011000_0001_m_000000  
attempt_201711011000_0001_m_000000_0
```

В данном примере 201711011000 — момент запуска YARN.

8.7 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

- Как организовано планирование в Hadoop 1.0?
- Из каких компонент состоит архитектура YARN?
- В чем недостатки алгоритма планирования FIFO?
- В чем суть алгоритма планирования Capacity Scheduler?
- В чем суть алгоритма планирования Fair Scheduler?
- Что такое вытеснение?
- В чем состоят отличия YARN от Mesos?
- В чем суть Dominant Resource Fairness?
- Как отлаживаются программы в Hadoop?

ГЛАВА 9

Теория сложности алгоритмов

MapReduce

В данной главе рассматриваются примеры реализации алгоритмов с использованием парадигмы MapReduce, а также описываются подходы для анализа сложности таких алгоритмов.

9.1 Основные понятия

Для удобства описания будем использовать следующие определения. Вызов функции **map** далее будем называть **распределением** или **отображением**. Вызов функции **reduce** будем называть **редукцией** или **сверткой**.

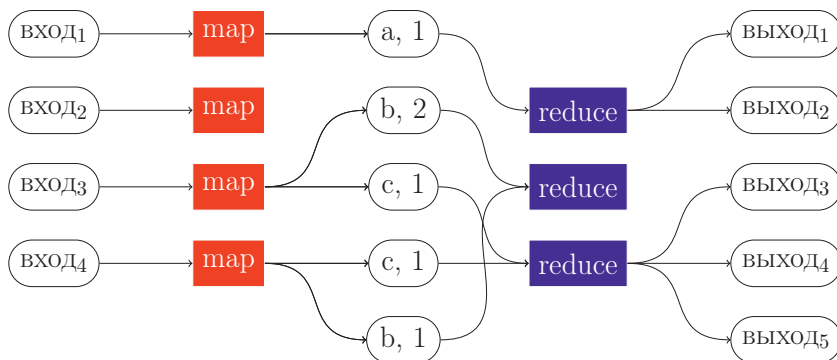


Рис. 125: Пример графа выполнения MapReduce [14]

Для упрощения описания алгоритмов в данной главе будем предполагать следующее: функция **map** принимает на вход отдельное значение (а не пару “ключ-значение”); функция **reduce** возвращает одно или несколько значений

без формирования ключей.

Таким образом, примем следующие обозначения:

- **map** (распределение = отображение): $x \rightarrow [(k, v)]$;
- **reduce** (редукция = свертка): $(k, [v]) \rightarrow [y]$.

9.2 Примеры алгоритмов

9.2.1 Пример 1. Инвертированный индекс

Одним из наиболее простых и наглядных примеров применения модели MapReduce является построение инвертированного индекса для ориентированного графа.

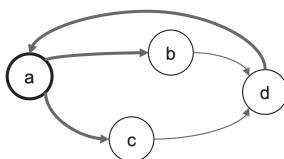


Рис. 126: Пример исходного графа [14]

Пусть входные данные представляют собой ориентированный граф, заданный в виде списка дуг. Каждая дуга представлена парой вершин: (**from**, **to**), где **from** — начальная вершина, а **to** — конечная. Необходимо получить граф, в котором все дуги направлены в противоположную сторону, то есть построить граф, обратный исходному.

Для решения задачи достаточно одной стадии — отображения (**map**), так как каждая дуга обрабатывается независимо, и группировка по ключу не требуется. На стадии **map** направление каждой дуги инвертируется.

Формат данных

from, **to**

Алгоритм

map: $(\text{from}, \text{to}) \rightarrow [(\text{to}, \text{from})]$

from	to
a	b
a	c
b	d
c	d
d	a

from	to
a	d
b	a
c	a
d	b
d	c

Рис. 127: Пример вычисления инвертированного индекса [14]

9.2.2 Пример 2. Умножение матрицы на вектор (вектор небольшого размера)

Рассмотрим задачу умножения большой матрицы на вектор, который целиком помещается в оперативную память. Матрица задана в текстовом файле в построчном формате: каждая строка содержит тройку (i, j, A_{ij}) , где i — номер строки, j — номер столбца, а A_{ij} — соответствующий элемент матрицы.

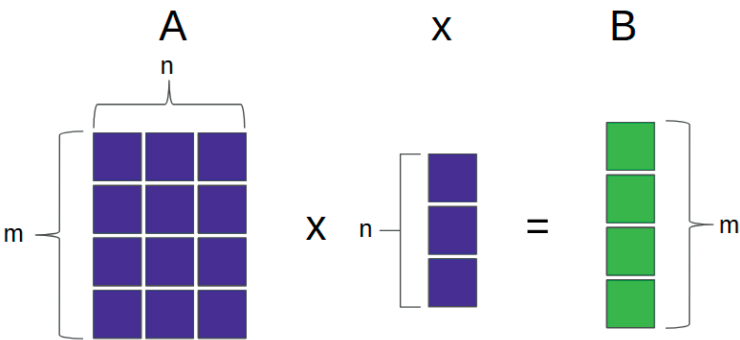


Рис. 128: Умножение матрицы на вектор x , который помещается в ОЗУ [14]

Поскольку вектор мал, для реализации алгоритма достаточно распространить его по узлам кластера. Это можно сделать с помощью класса `DistributedCache` (копирование данных на узлы кластера) или передачи че-

рез параметр `-files` при запуске задания. Таким образом, каждый экземпляр Маррега будет иметь доступ к вектору локально в своей памяти. Функция `map` получит i, j -й элемент матрицы, умножит его на j -й элемент вектора и вернет с ключом i . Каждый вызов `reduce` сформирует значение i -й позиции вектора результата путем суммирования значений, относящихся к ключу i .

Формат данных

i, j, A_{ij}

Алгоритм

1. Распространить вектор x по всем узлам кластера;
2. `map`: $i, j, A_{ij} \rightarrow [(i, A_{ij} \cdot x_j)]$;
3. `reduce`: $i, [A_{i1} \cdot x_1, \dots, A_{in} \cdot x_n] \rightarrow i, \sum_{j=1}^n A_{ij} \cdot x_j$.

9.2.3 Пример 3. Умножение матрицы на вектор (вектор большого размера)

Рассмотрим также задачу умножения матрицы на вектор, который целиком не помещается в оперативную память. В таком случае его можно разбить на части, каждая из которых будет помещаться в памяти, а матрицу разбить на вертикальные полосы с шириной, равной длине части вектора. Далее необходимо умножить каждую полосу матрицы на соответствующую ей часть вектора с помощью предыдущего алгоритма (Пример 2) и просуммировать результирующие векторы.

9.2.4 Пример 4. Операции реляционной алгебры

Рассмотрим реализацию базовых операций реляционной алгебры с использованием MapReduce: выборку (`where`), проекцию (`select`), объединение (`union`), пересечение (`intersect`), разность (`minus`), естественное соединение (`natural join`), а также группировку с агрегированием (`group by`).

В данном примере рассмотрим именно отношения реляционной алгебры — множества кортежей (в SQL — мультимножества), а не таблицы реляционной

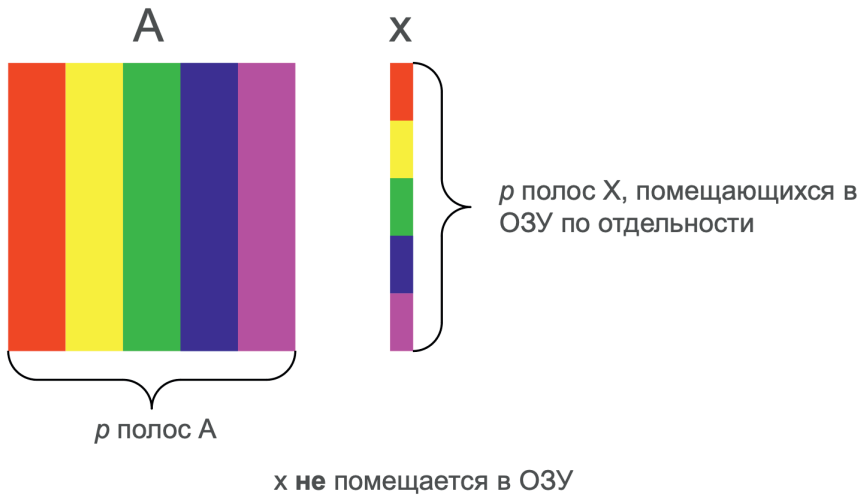


Рис. 129: Пример умножения матрицы на вектор большого размера [14]

БД, которые могут содержать одинаковые строки. Будем считать, что кортежи каждого отношения хранятся в виде файла в распределенной файловой системе, где каждая строка соответствует отдельному кортежу.

- Выборка:
`... WHERE a>b AND NOT (c>-5.1 OR 1=1);`
- Проекция:
`SELECT a, c, d+3 as d1 ...;`
- Объединение, пересечение, разность:
`UNION, INTERSECT, MINUS;`
- Естественное соединение:
`A LEFT JOIN B on x INNER JOIN C on B.y>C.z RIGHT JOIN D on z
 FULL JOIN E on q CROSS JOIN F on w;`
- Группировка и агрегирование:
`SELECT x, y, min(a), max(b), avg(c) from X GROUP BY x, y.`

Рассмотрим псевдокод реализации каждой операции с помощью простой схемы MapReduce. Важно отметить, что многие из них выполняются эффективно за один проход (одну фазу map и reduce).

- **Выборка:**

C — условие

map: $t \rightarrow [(t, t)]$, если $C(t)$, иначе — $[]$

reduce: $(t, [t]) \rightarrow [t]$

- **Проекция:**

S — схема проекции

map: $t \rightarrow [(S(t), S(t))]$

reduce: $(t', [t', t', \dots, t']) \rightarrow [t']$

- **Объединение:**

map: $t \rightarrow [(t, t)]$

reduce: $(t, [t]) \rightarrow [t]$

$(t, [t, t]) \rightarrow [t]$

- **Пересечение:**

map: $t \rightarrow [(t, t)]$

reduce: $(t, [t]) \rightarrow []$

$(t, [t, t]) \rightarrow [t]$

- **Разность:**

$R \setminus S$

map: $t \rightarrow [(t, 'R')]$, если $t \in R$

$t \rightarrow [(t, 'S')]$, если $t \in S$

reduce: $(t, \text{tags}) \rightarrow [t]$, если $\text{tags} == ['R']$, иначе — $[]$

- **Естественное соединение:**

$R(A, B) \bowtie S(B, C)$

map: $(a, b) \rightarrow [(b, ('R', a))]$

$(b, c) \rightarrow [(b, ('S', c))]$

reduce: $(b, \text{values}) \rightarrow$ список кортежей (a, b, c) из всевозможных пар значений из values: одна пара с тегом 'R', вторая — с тегом 'S'

- **Группировка и агрегирование:**

$R(A, B, C)$, группировочный атрибут A, агрегация по B

map: $(a, b, c) \rightarrow [(a, b)]$

reduce: $(a, [b_1, \dots, b_n]) \rightarrow [(a, \text{aggr}(b_1, \dots, b_n))]$

9.2.5 Пример 5. Умножение двух матриц

Рассмотрим общий пример умножения двух матриц. Пусть обе матрицы большие и не помещаются в оперативной памяти. Предлагаемая реализация похожа на соединение таблиц и использует модель MapReduce дважды.

На 1-м шаге функция **map** формирует выходное значение таким образом, чтобы ключом стал номер столбца для элемента матрицы A и номер строки для элемента матрицы B. Reduce получит все элементы с ключом j и сформирует всевозможные произведения значения из матрицы A на значение из матрицы B.

На 2-м шаге используется только стадия reduce, в которой необходимо лишь просуммировать значения, отвечающие ключу — позиции i,k.

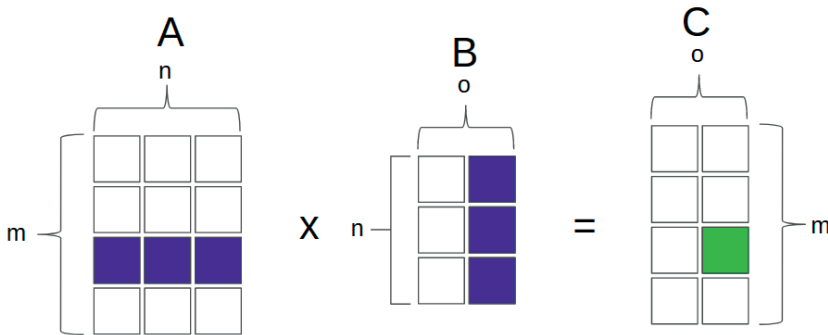


Рис. 130: Умножение двух матриц [14]

Умножение матриц похоже на операцию естественного соединения $A(I, J, A_{ij}) \bowtie B(J, K, B_{jk})$.

Алгоритм

1-й шаг

map: $(A', i, j, A_{ij}) \rightarrow [(j, (A', i, A_{ij}))]$

$(B', j, k, B_{jk}) \rightarrow [(j, (B', k, B_{jk}))]$

reduce: для пар (A', i, A_{ij}) и (B', k, B_{jk}) породить $((i, k), A_{ij}B_{jk})$

2-й шаг

map: вход $\rightarrow$ выход (identity mapper)

reduce: просуммировать значения

9.2.6 Пример 6. Умножение двух матриц за один шаг

Рассмотрим другой вариант реализации перемножения матриц за один шаг. Каждый элемент входных матриц участвует в формировании некоторых элементов матрицы C . На шаге **map** каждый элемент исходных матриц дублируется с ключами (позициями матрицы C), в вычисление которых он войдет. **Reduce** формирует значение матрицы C путем суммирования произведений соответствующих элементов матриц A и B .

Формат хранения матрицы

$A: i, j, A_{ij}$

$B: j, k, B_{jk}$

Алгоритм

map: $(A', i, j, A_{ij}) \rightarrow [((i, k), (A', j, A_{ij})) \text{ для } k=1, \dots, o]$

$(B', j, k, B_{jk}) \rightarrow [((i, k), (B', j, B_{jk})) \text{ для } i=1, \dots, m]$

reduce: отсортировать по j A_{ij} и B_{jk}

return $i, k, \sum_{j=1}^n A_{ij}B_{jk}$

9.3 Модель коммуникационной стоимости

Рассмотрим одну из моделей анализа сложности алгоритмов MapReduce — модель коммуникационной стоимости.

Коммуникационная стоимость стадии (map или reduce) определяется как общий объем входных данных, поступающих на соответствующую стадию, в байтах или в числе объектов.

Коммуникационная стоимость задания — сумма коммуникационных стоимостей его стадий.

В мире больших данных алгоритмы часто просты и обладают линейной вычислительной сложностью, а основным ограничивающим фактором становится передача и считывание больших массивов данных. Коммуникационная стоимость учитывает только входные данные стадий, поскольку выполнение задания в MapReduce организовано по принципу конвейера, где выход одной стадии становится входом следующей. Избыточно учитывать одновременно и вход, и выход соседних стадий. Кроме того, выходные данные задания часто значительно меньше исходного объема, поэтому основное внимание уделяется именно входным данным.

Почему актуальна коммуникационная стоимость?

- Алгоритмы обработки данных часто просты → нагрузка на процессор минимальна;
- Входные данные — результат передачи данных по сети и/или чтения с жесткого диска (скорости невелики).

Почему учитываются входные данные, а не выходные?

- Вход одной задачи является выходом другой;
- Выходные данные, как правило, меньше входных.

9.3.1 Коммуникационная стоимость соединения

Оценим коммуникационную стоимость соединения двух таблиц. Стоимость **map** в точности равна общему числу элементов соединяемых таблиц. Стадия **reduce** с учетом передачи тега таблицы имеет стоимость $O(r + s)$.

$$R(A, B) \bowtie S(B, C)$$

$$r = \text{len}(R)$$

$$s = \text{len}(S)$$

$$\text{cost}(\text{map}) = r + s$$

$$\text{cost}(\text{reduce}) = O(r + s)$$

$$\text{cost}(R(A, B) \bowtie S(B, C)) = O(r + s)$$

9.3.2 Коммуникационная стоимость двух последовательных соединений

Рассмотрим ситуацию двух последовательных соединений таблиц. Добавим параметр p — вероятность соответствия ключей у таблиц R и S и таблиц S и T . Тогда соединение таблиц R и S породит rsp элементов:

$$R(A, B) \bowtie S(B, C) \bowtie T(C, D)$$

$$r, s, t — \text{размеры } R, S, T.$$

Для R и S , S и T вероятности соответствия ключей одинаковы и равны p :

$$\text{cost}(R(A, B) \bowtie S(B, C)) = O(r+s)$$

$$\text{cost}(\langle \text{результат соединения } R \text{ и } S \rangle \bowtie T(C, D)) = O(rsp+t)$$

$$\text{cost}(R(A, B) \bowtie S(B, C) \bowtie T(C, D)) = O(r+s+t+rsp)$$

9.3.3 Коммуникационная стоимость двойного соединения в одном reduce

Другой вариант соединения трех отношений — соединение в одном **reduce**. Используем хеш-функцию с b значениями для хеширования значений атрибута B и хеш-функцию с c значениями для хеширования значений атрибута C . Алгоритм отправляет элементы таблиц в $b \cdot c$ редукций. Каждый элемент

Для выбора b и c используем метод множителей Лагранжа. Продифференцируем выражение для стоимости по b и c и приравняем к нулю. Выразим λ и вычислим оптимальные b и c . Таким образом, получаем значение формулы коммуникационной стоимости двойного соединения в одном `reduce`.

$s + cr + bt \rightarrow \min$, при условии $bc = k$

Метод множителей Лагранжа:

$$s + cr + bt - \lambda(bc - k),$$

$$r - \lambda b = 0,$$

$$t - \lambda c = 0,$$

$$rt = \lambda^2 bc,$$

$$\lambda = \sqrt{\frac{rt}{k}}.$$

Итого: $O(r + 2s + t + 2\sqrt{rtk})$

9.4 Теория сложности MapReduce

Существует другой подход к оценке сложности алгоритмов MapReduce. Назовем **коэффициентом репликации r** — среднее число пар “ключ-значение”, порождаемых вызовами функции `map` на одну входную запись. Значение r можно интерпретировать как средний объем сетевого трафика при обращении к функции `map` на каждую входную запись. Назовем **размером редукции q** — максимальное число значений, соответствующих ключу при вызове функции `reduce`. Значение q можно интерпретировать как пиковую сложность каждой редукции. Для любого MapReduce-алгоритма можно вычислить r и q .

Отметим, что значения r и q связаны между собой. Малое q соответствует большому числу reducers, а они потребуют больше сетевого трафика, чем значение r . При малом значении q функция `reduce` будет иметь возможность

загрузить небольшие входные данные в оперативную память $\rightarrow$ повышается степень параллелизма при выполнении редукций. В таком случае увеличивается сетевой трафик, поскольку отображения вынуждены дублировать данные.

9.5 Примеры алгоритмов с учетом теории сложности MapReduce

9.5.1 Сложность однопроходного умножения матриц

Рассмотрим сложность алгоритма однопроходного умножения матриц на примере квадратных матриц. Каждый вызов `map` породит n пар “ключ-значение”, каждый вызов `reduce` получит $2 \cdot n$ значений.

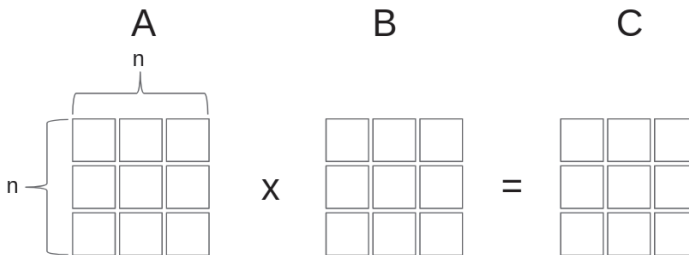


Рис. 135: Пример умножения квадратных матриц [14]

Формат хранения матрицы

A, i, j, A_{ij}

B, j, k, B_{jk}

Алгоритм

`map`: $(A', i, j, A_{ij}) \rightarrow [(i, k), (A', j, A_{ij})]$ для $k=1, \dots, n]$

$(B', j, k, B_{jk}) \rightarrow [(i, k), (B', j, B_{jk})]$ для $i=1, \dots, n]$

`reduce`: отсортировать по j A_{ij} и B_{jk}

return $i, k, \sum_{j=1}^n A_{ij} B_{jk}$

Коэффициент репликации: $r = n$.

Размер редукции: $q = 2n$.

9.5.2 Сложность попарного сравнения изображений

Рассмотрим пример попарного сравнения миллиона изображений, каждое из которых занимает 1 МБ. Тривиальное решение основано на сравнении в каждом **reduce** ровно двух изображений. Для этого каждый **map** $(10^6 - 1)$ раз продублирует изображение. В обозначениях будем использовать фигурные скобки для указания упорядоченной пары изображений.

Проанализируем вычислительную сложность данной реализации. С точки зрения стадии редукции решение является корректным: каждая функция обрабатывает только две записи, что обеспечивает малую нагрузку на отдельные **reduce**-задачи.

Критическим ограничением становится коэффициент репликации, поскольку каждый входной объект должен быть продублирован почти миллион раз. Это приводит к взрывному росту объема данных, передаваемых по сети, и делает данное решение неэффективным и практически неприменимым на больших объемах данных.

Дано:

Число изображений p : 10^6

Размер каждого изображения: 1 МБ

Мера сходства: $s(x, y)$

Требуется: посчитать $s(x, y)$ для всех пар изображений

Наивное решение

map: $(i, P_i) \rightarrow [(\{i, j\}, P_i) \text{ для } j=1, \dots, p \text{ и } j \neq i]$

reduce: $(\{i, j\}, (P_i, P_j)) \rightarrow [s(P_i, P_j)]$

Оценка сложности алгоритма

$r = 10^6 - 1$ (объем трафика: $(10^6 - 1) \cdot 10^6 \cdot 1 \text{ МБ} = 10^{18} \text{ Б}$)

$q = 2$ (объем данных каждой редукции: 2 МБ)

9.5.3 Оптимизированное решение

Для оптимизации решения предлагается добавить группировку изображений в g групп. Тогда основной идеей алгоритма станет сравнение в каждом reduce изображений из двух разных групп, а также сравнение изображений внутри групп.

Алгоритм

map: $(i, P_i) \rightarrow [(\{group(i), j\}, (i, P_i)) \text{ для } j = 1, \dots, g \text{ и } j \neq group(i)]$

reduce:

- Для каждого ключа $\{u, v\}$ имеем $2 \cdot 10^6/g$ значений (i, P_i) , где i принадлежит u или v ;
- Для (i, P_i) и (j, P_j) , где i и j из разных групп, вычислим $s(P_i, P_j)$;
- Если ключ имеет вид $\{u, u \oplus 1\}$, вычисляем $s(P_i, P_j)$ для i, j из группы u .

9.5.4 Пример решения задачи и его сложность

Рассмотрим пример для 6 изображений и 3 групп. Оценим сложность оптимизированного решения.

Сложность

$r = g - 1$ (объем трафика: $10^6 \cdot (g-1) \cdot 1 \text{ МБ} = 10^{15} \text{ Б} = 1 \text{ ПБ}$ для $g=1000$).

$q = 2 \cdot 10^6/g$ (объем входа каждой редукции: $2 \cdot 10^6/g \cdot 1 \text{ МБ} = 2 \cdot 10^9 \text{ Б} = 2 \text{ ГБ}$ для $g=1000$).

Для 1000 групп трафик составит 1 ПБ, размер редукции — 2 ГБ, что реалистично по сравнению с тривиальным решением. Однако число групп можно уменьшить.

9.5.5 Умножение двух матриц за один шаг

Рассмотрим способ оптимизации алгоритма умножения матриц за один шаг с помощью групп.

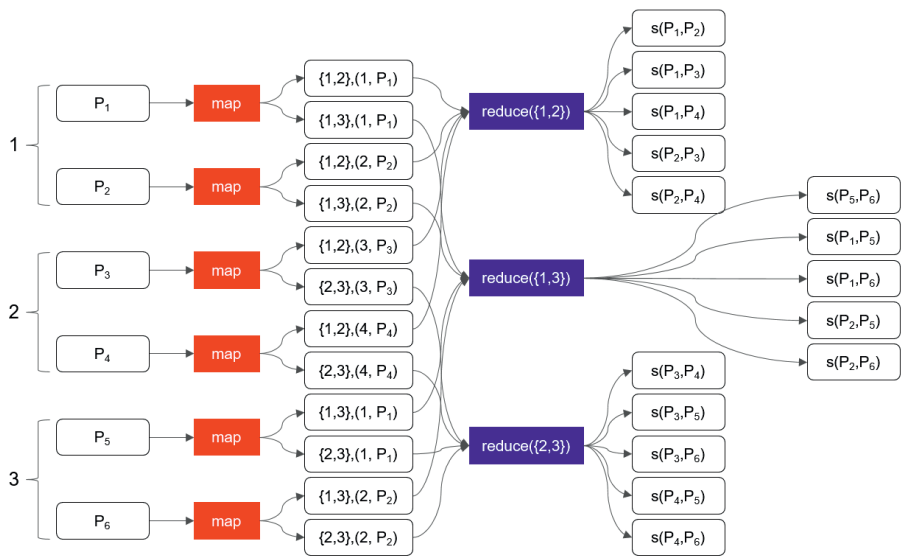


Рис. 136: Пример решения задачи для 6 изображений и 3 групп [14]

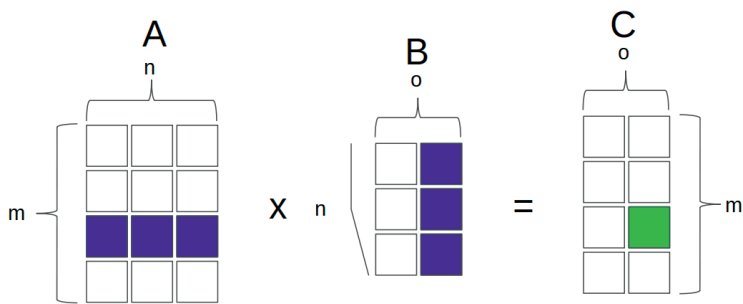


Рис. 137: Пример вычисления одного значения при матричном умножении [14]

Формат хранения матрицыA: i, j, A_{ij} B: j, k, B_{jk} **Алгоритм****map:** ('A', i, j, A_{ij}) $\rightarrow$ [(i, k), ('A', j, A_{ij})] для $k=1, \dots, o$ ('B', j, k, B_{jk}) $\rightarrow$ [(i, k), ('B', j, B_{jk})] для $i=1, \dots, m$ **reduce:** отсортировать по j A_{ij} и B_{jk} return i, k, $\sum_{j=1}^n A_{ij} B_{jk}$ **9.5.6 Оптимизированное умножение матриц за один шаг**

Рассмотрим оптимизацию алгоритма путем разбиения исходных матриц на g полос. Map будет дублировать элемент матрицы уже не n , а g раз. Подсчет коэффициента репликации и размера редукции предоставляется читателю в качестве упражнения.

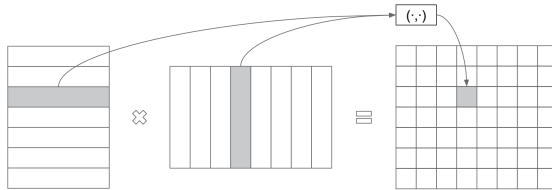


Рис. 138: Пример получения одного значения путем умножения строки на столбец [14]

Дано:Матрицы A, B размера $n \times n$ g — число групп (полос)**Алгоритм****map:** ('A', i, j, A_{ij}) $\rightarrow$ [(group(i), g'), ('A', i, j, A_{ij})] для $g'=1 \dots g$ ('B', j, k, B_{jk}) $\rightarrow$ [(g' , group(j)), ('B', j, k, B_{jk})] для $g'=1 \dots g$ **reduce:** сформировать блок матрицы C

9.5.7 Оптимизация умножения матриц за два шага

Рассмотрим также другой вариант оптимизации с разбиением исходных матриц на одинаковые квадраты и использованием двухпроходного алгоритма. Подсчет сложности алгоритма предоставляется читателю в качестве упражнения.

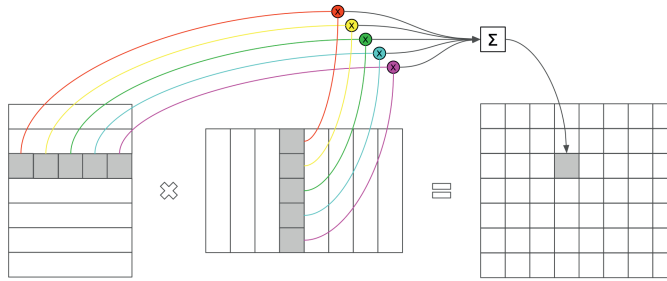


Рис. 139: Получение значения через скалярное произведение [14]

1-й шаг:

map: $(A', i, j, A_{ij}) \rightarrow [(I, J, K), (A', i, j, A_{ij})]$ для $K=1\dots g$

$(B', j, k, B_{jk}) \rightarrow [(I, J, K), (B', j, k, B_{jk})]$ для $I=1\dots g$

reduce: $x_{iJk} = \sum_{j \in J} A_{ij} B_{jk}$

return $((i, k), x_{iJk})$

2-й шаг:

Суммирование в reducer

9.6 Графовая модель задач в MapReduce

Одним из подходов к представлению задачи, решаемой в модели MapReduce, является использование графовой модели. В этом случае каждая входная величина (объект) связывается с выходными результатами, в вычислении которых она участвует. Таким образом формализуется структура задачи, хотя сама стратегия решения при этом еще не задается.

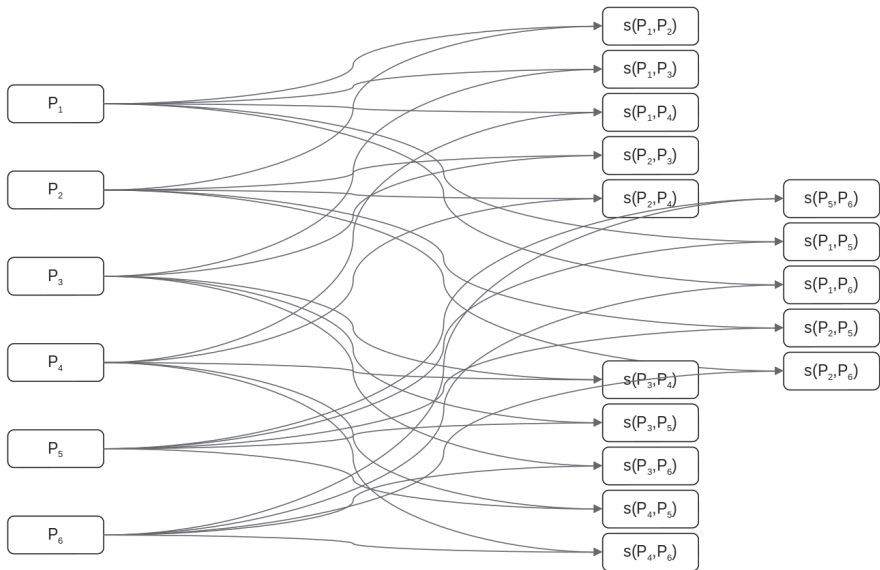


Рис. 140: Пример графовой модели задач в MapReduce [14]

Данный способ визуализации называется **графовой моделью** задач MapReduce. В ряде случаев структура такого графа может зависеть от особенностей входных данных. Например, при выполнении естественного соединения таблиц граф будет определяться набором кортежей, подаваемых на вход.

Элементами графа являются:

- **Входные вершины** — входные значения;
- **Выходные вершины** — выходные значения;
- **Связи (ребра)** — соединяют вход и выход, если соответствующее входное значение участвует в получении данного результата.

9.6.1 Пример задачи: сравнение изображений

В качестве примера рассмотрим задачу попарного сравнения изображений. В этом случае выходными вершинами будут результаты сравнений пар

изображений, а входными — сами изображения. Связи в графе показывают, какие изображения участвуют в формировании каждого результата.

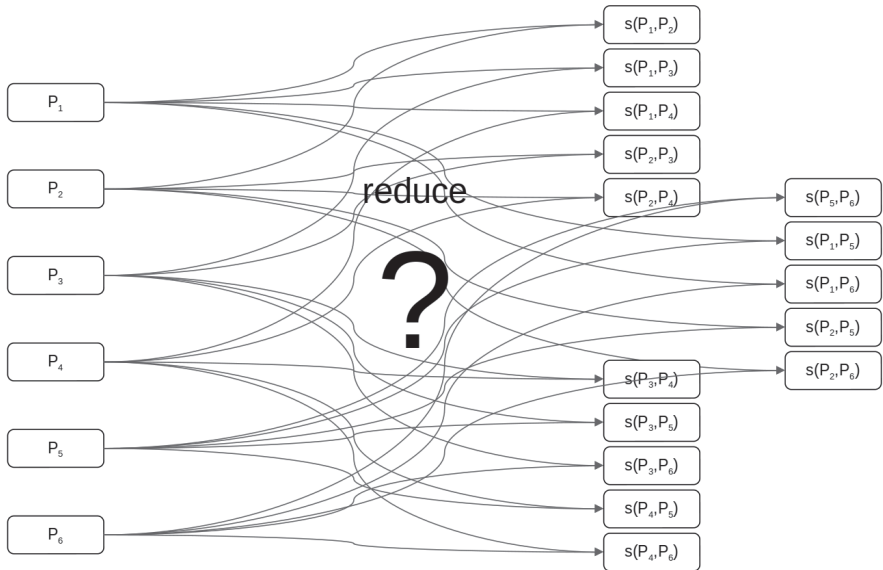


Рис. 141: Пример графовой модели задач в MapReduce (место reduce) [14]

9.6.2 Пример решения задачи

На Рисунке 142 представлен граф решения задачи сравнения изображений с помощью группировки.

9.6.3 Схема сопоставления

Графовое представление решения называется схемой сопоставления. **Схема сопоставления** — распределение входов задачи по операциям редукции с размером редукции q . Схема должна удовлетворять следующим требованиям:

- Каждая операция редукции обрабатывает не более q входов;

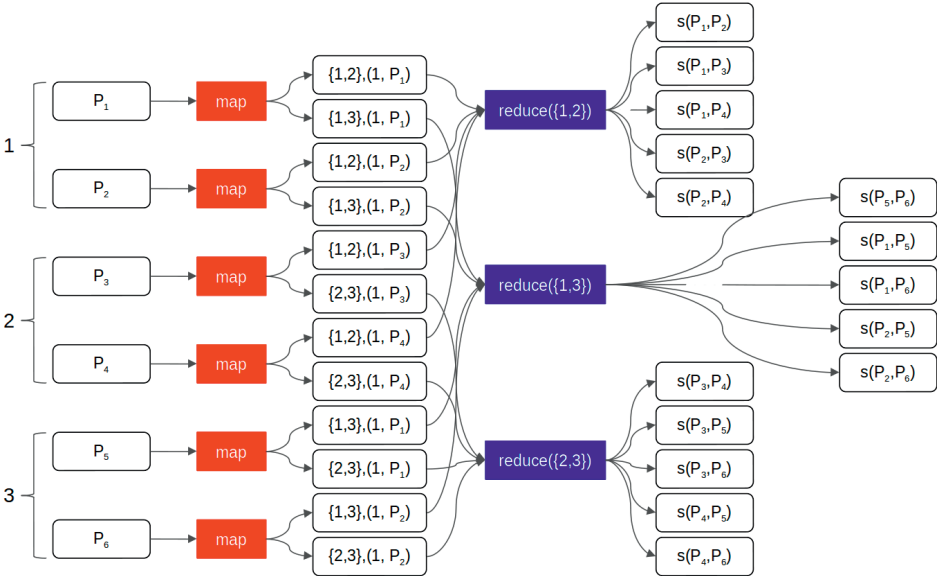


Рис. 142: Пример графовой модели задач в MapReduce (добавление map и reduce) [14]

- Для каждого выхода существует как минимум одна операция редукции, которой назначены все входы, связанные с данным выходом. Эта редукция покрывает выход.

9.7 Получение нижних оценок $r(q)$

Для заданного решения задачи (для заданной схемы сопоставления) можно получить оценку лучшего коэффициента репликации для заданного размера редукции.

Пример: Оптимизированное сопоставление изображений

Число изображений p , число групп g

Число входов: p

Число выходов: $p(p-1) / 2$

$r = g - 1$

$q = 2p / g$

$r \sim 2p / q$

Оценим существует ли алгоритм с лучшим $r(q)$. Верхняя граница числа выходов, которые может покрыть операция редукции с q входами: $\sim q^2/2$, а число выходов $\sim p^2/2$. Пусть число редукций равно k . Следовательно:

$$\sum_{i=1}^k q_i^2 / 2 \geq p^2 / 2,$$

$$\sum_{i=1}^k q_i^2 \geq p^2,$$

$$q \sum_{i=1}^k q_i \geq p^2,$$

$$\sum_{i=1}^k q_i \geq p^2 / q,$$

$$\frac{\sum_{i=1}^k q_i}{p} \geq p / q,$$

$$r \geq p/q.$$

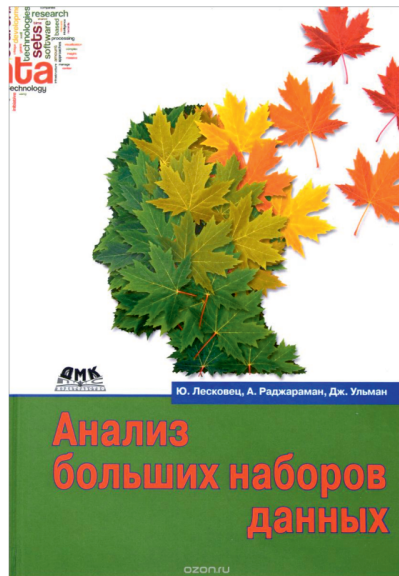


Рис. 143: Книга “Анализ больших наборов данных” [14]

В качестве литературы по данной теме рекомендуется книга Юре Лесковца на английском языке [14].

9.8 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

- Как в модели MapReduce можно реализовать умножение матриц?
- Что такое модель коммуникационной стоимости?
- Что такое коэффициент репликации и размер редукции?
- Что такое графовая модель и схема сопоставления?

ГЛАВА 10

Облачные вычисления

Облачные вычисления (cloud computing) — это модель предоставления вычислительных ресурсов по сети. К таким ресурсам относятся серверы, хранилища данных, базы данных, сетевое оборудование, программное обеспечение, аналитические средства и другие услуги. Облачные вычисления позволяют пользователям получать доступ к вычислительным мощностям по мере необходимости, с возможностью масштабирования и гибкой оплаты.

В данной главе будут рассмотрены основные модели развертывания облачных систем и виды предоставляемых сервисов.

10.1 Развертывание облачных вычислений

Существует несколько моделей развертывания облачных инфраструктур, отличающихся уровнем доступности и контроля:

- **Публичное облако (Public Cloud)** — ресурсы предоставляются внешним провайдером и используются множеством арендаторов (организаций или пользователей);
- **Частное облако (Private Cloud)** — облачная инфраструктура принадлежит и используется одной организацией. Все ресурсы, включая оборудование, находятся под полным контролем;
- **Гибридное облако (Hybrid Cloud)** — объединяет элементы публичной и частной моделей, обеспечивая гибкость и баланс между безопасностью и масштабируемостью. Например, конфиденциальные данные

могут обрабатываться в частной части облака, а менее чувствительная информация — в публичной;

- **Облачное сообщество (Community Cloud)** — инфраструктура совместно используется определенным сообществом организаций с общими задачами и интересами (например, университеты, исследовательские центры, отраслевые консорциумы).



Рис. 144: Примеры поставщиков облачных вычислений

По уровню абстракции и предоставляемых возможностей облачные услуги подразделяются на три основные категории:

- Программное обеспечение как услуга (**Software-as-a-Service, SaaS**) — пользователю предоставляется доступ к прикладному программному обеспечению через Интернет. Примеры: Google Workspace, Microsoft 365;
- Платформа как услуга (**Platform-as-a-Service, PaaS**) — пользователю предоставляется среда для разработки, тестирования и развертывания приложений, включая ОС, системы управления базами данных, веб-серверы и т. д. Примеры: Google App Engine, Heroku;

- Инфраструктура как услуга (**Infrastructure-as-a-Service, IaaS**) — пользователю предоставляется доступ к вычислительным ресурсам: виртуальным машинам, сетям, хранилищам и т. д., с возможностью самостоятельной настройки. Примеры: Amazon EC2, Microsoft Azure.

Примерами поставщиков облачных услуг являются Microsoft Azure, Amazon Web Services, Google Cloud Platform и сервер для научных вычислений SciServer (пример community-модели облака).

10.2 Платформа Azure

Рассмотрим функционирование облачных сервисов на примере платформы Azure от компании Microsoft.



Рис. 145: Логотип Microsoft Azure

10.2.1 Подписка

Единицей выставления счетов Azure является **подписка**. Она выдается пользователю — физическому лицу либо организации, имеет фиксированный период действия и квоты на использование ресурсов. В подписке создается единственный пользователь с правами администратора. Он имеет возможность создавать пользователей и управлять счетами.

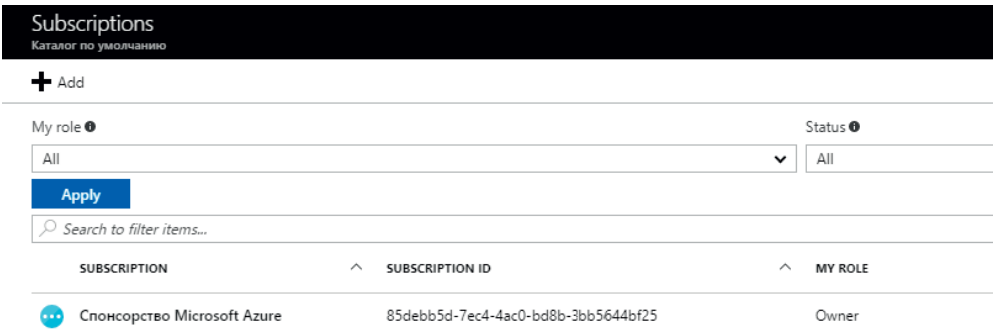


Рис. 146: Окно предоставления подписки на Microsoft Azure

10.2.2 Права доступа к ресурсам

Права пользователя определяются его ролью по отношению к ресурсу. На практике наиболее распространены следующие роли:

- **Owner** — может изменять ресурс и управлять правами на доступ к ресурсу;
- **Contributor** — может изменять ресурс;
- **Reader** — может просматривать ресурс.

Роли, заданные для подписки, наследуются по иерархии ресурсов (подписка → группа ресурсов → ресурс). Их можно модифицировать в узлах иерархии.

10.2.3 Модели развертывания ресурсов

В современных облачных платформах (как и в Microsoft Azure) реализованы различные подходы к развертыванию и организации ресурсов. Один из ключевых механизмов — группировка взаимосвязанных ресурсов в **группы ресурсов** (resource groups). Это позволяет упростить управление, мониторинг и применение политик на уровне проекта.

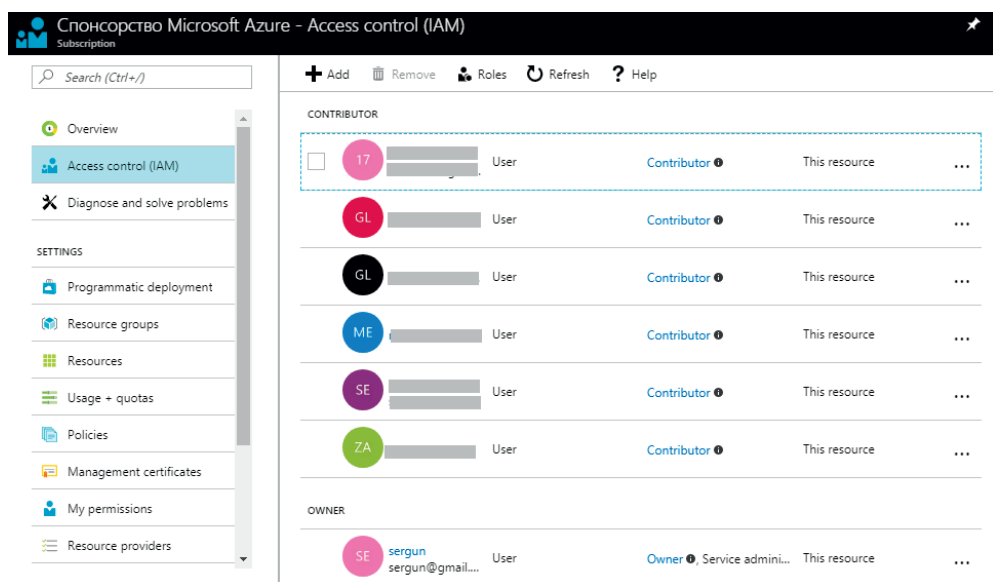


Рис. 147: Пример списка пользователей подписки и их ролей

Классическая модель (Classic Deployment Model) представляет собой устаревший подход, при котором отсутствует возможность логической группировки ресурсов. Каждое созданное облачное средство (виртуальная машина, хранилище, база данных, сетевая инфраструктура и др.) управляется отдельно, что усложняет сопровождение комплексных решений.

Модель с использованием диспетчера ресурсов (Azure Resource Manager, ARM) обеспечивает логическую группировку ресурсов, упрощает управление зависимостями, доступом, шаблонами развертывания (ARM templates) и автоматизацией.

Характеристики группы ресурсов:

- Объединяет ресурсы, относящиеся к одному проекту;
- Каждый ресурс принадлежит единственной группе;
- Ресурсы группы могут находиться в центрах обработки данных из разных регионов мира;

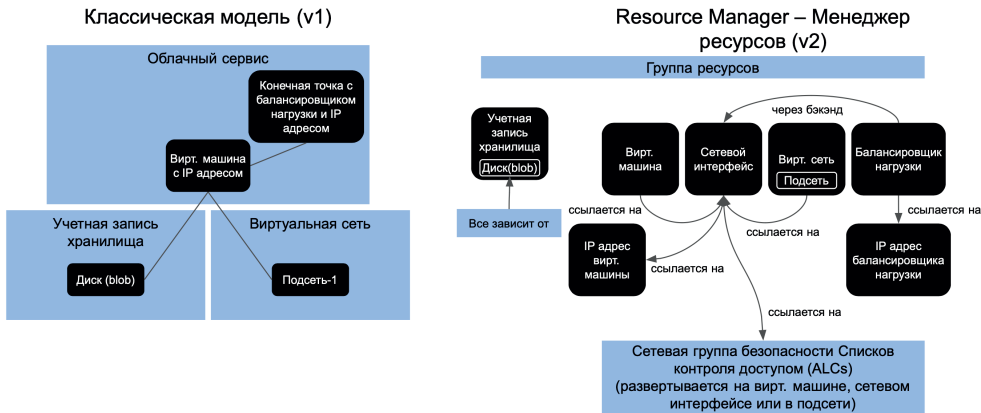


Рис. 148: Пример моделей развертывания ресурсов

- Группа ресурсов имеет назначенный регион, используемый исключительно для хранения метаданных;
- Поддерживается перемещение ресурсов между группами и подписками;
- Для группы могут быть заданы индивидуальные лимиты и политики, аналогичные ограничениям, применяемым на уровне подписки.

10.2.4 Учетная запись хранения BLOB-объектов

Для хранения больших объемов данных в облачной платформе Microsoft Azure применяется концепция **учетной записи хранения объектов BLOB (Binary Large Objects)**. Такой тип хранилища предназначен для эффективной работы с неструктурированными данными — такими как изображения, видео, резервные копии, журналы и другие крупные бинарные объекты.

В рамках одной учетной записи хранения может быть создано неограниченное количество **контейнеров**, которые логически организуют данные и аналогичны папкам верхнего уровня в файловой системе. Контейнеры слу-

vm2017
Resource group

Search (Ctrl+/)

Overview

Activity log

Access control (IAM)

Tags

SETTINGS

Quickstart

Resource costs

Deployments

Policies

Properties

Locks

Automation script

+ Add Columns Delete Refresh Move

Essentials ^

Subscription name (change)
Спонсорство Microsoft Azure

Deployments
2 Succeeded

Subscription ID
85debb5d-7ec4-4ac0-bd8b-3afef644bf25

Filter by name... All types All locations

8 items

NAME	TYPE	LOCATION
disk1	Disk	East US
hourse	Virtual machine	East US
hourse_OsDisk_1_24a8cb21ce8944c793d220cef7...	Disk	East US
hourseNSG	Network security group	East US
hoursePublicIP	Public IP address	East US
hourseVMNic	Network interface	East US
hourseVNET	Virtual network	East US
vm2017loqs	Storage account	East US

Рис. 149: Пример перечня виртуальных ресурсов, относящихся к одной группе: виртуальная машина, диски, сетевые устройства, хранилища файлов журналов

жат для группировки BLOB-объектов, представляющих собой единицы хранения данных.

Для каждой учетной записи хранения задаются следующие параметры:

- **Политики безопасности** — включают контроль доступа, использование ключей, ограничения IP-адресов и протоколов;
- **Тип хранилища** — *горячее* (hot) хранилище предназначено для часто запрашиваемых данных, в то время как *холодное* (cool) хранилище предназначено для редкого доступа (например, архивных данных);
- **Механизм репликации** — определяет устойчивость к сбоям и территориальную избыточность хранения.

Доступ к учетной записи хранения может осуществляться с использованием одного из методов аутентификации, например, **доступа по ключу** (полный доступ к данным) или с использованием **подписей доступа** (ограниченные права доступа по времени, операциям и адресам).

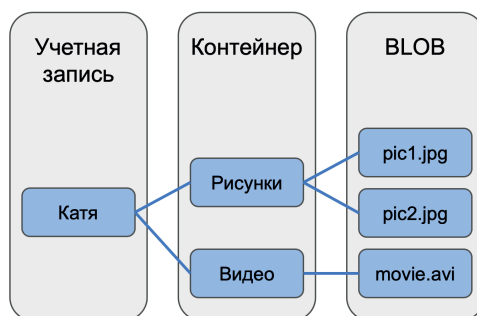


Рис. 150: Схема организации хранилища

10.2.5 Контейнер

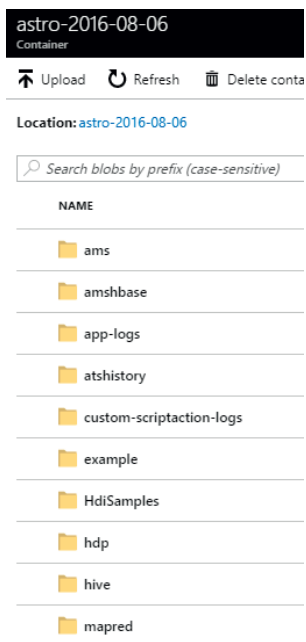


Рис. 151: Интерфейс файловой системы

Для контейнера можно назначить один из уровней доступа: **private** (доступ только по ключу), **blob** (доступ на чтение файлов (конкретных BLOB-объектов) в контейнере по имени), либо **container** (доступ на чтение файлов с возможностью просмотра списка файлов в контейнере).

10.2.6 Типы репликации учетной записи хранения

Учетные записи хранения поддерживают 4 типа репликации:

1. **Локально избыточное хранилище (LRS)**

Данные реплицируются трижды в одном центре обработки данных.

2. **Хранилище, избыточное в пределах зоны (ZRS)**

Данные трижды реплицируются в > 1 центрах обработки данных одно-

го или двух регионов. Первый регион выбирается при создании учетной записи, 2-й — автоматически по соседству.

3. Геоизбыточное хранилище (GRS)

Данные синхронно реплицируются 3 раза в первом регионе и 3 раза асинхронно во втором регионе.

4. Геоизбыточное хранилище с доступом для чтения (RA-GRS)

Аналогично GRS, но во втором регионе данные реплицируются с правами только на чтение и создаются две конечные точки для доступа к данным (чтение и чтение + запись):

`myaccount.blob.core.windows.net`

`myaccount-secondary.blob.core.windows.net`

10.3 Кластер HDInsight

Одним из облачных сервисов, предоставляемых Microsoft Azure, является **кластер HDInsight** — управляемое решение для развертывания и работы с распределенными вычислительными фреймворками (например, Apache Hadoop, Spark, Hive).

При создании кластера HDInsight пользователь может выбрать:

- Количество и тип вычислительных узлов;
- Характеристики головных узлов (на которых размещаются master-компоненты, такие как NameNode и YARN ResourceManager);
- Характеристики рабочих узлов (исполняющих map/reduce-задачи).

10.3.1 Взаимодействие с кластером

Интерфейс Azure Portal предоставляет интуитивно понятные средства выбора конфигурации оборудования для различных ролей узлов. Кластер

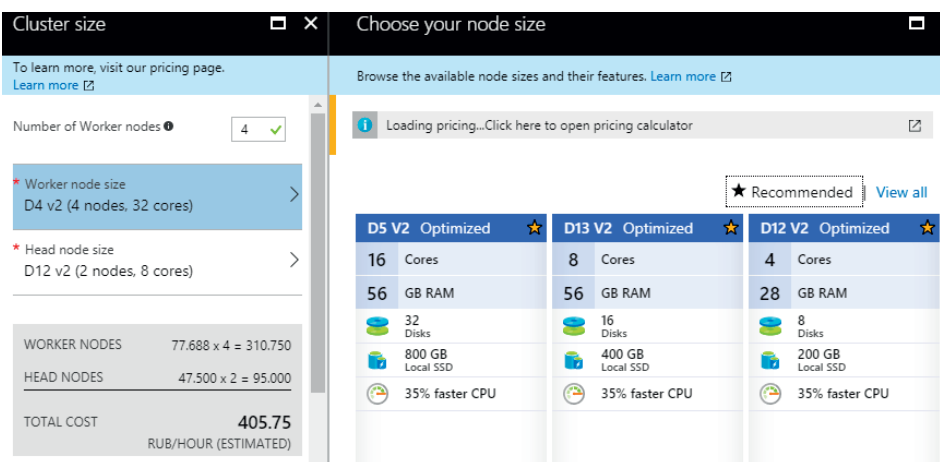


Рис. 152: Интерфейс подбора конфигурации кластера

HDInsight поддерживает различные интерфейсы для управления, настройки и выполнения вычислений:

- Портал Azure (portal.azure.com);
- PowerShell;
- Azure CLI (на Python).

Настройка компонентов кластера осуществляется через Apache Ambari, а отправка заданий — через SSH (например, для запуска Hadoop-заданий вручную) и Ambari Hive View (для выполнения SQL-запросов на Hive).

Мониторинг и отладка осуществляются через:

- Пользовательский интерфейс YARN ResourceManager;
- Команду `yarn logs -applicationId` для получения логов выполнения задач.

10.3.2 Особенности хранения данных в HDInsight

Кластеры HDInsight используют модифицированную архитектуру хранения, отличающуюся от классического Hadoop. Вместо локального хранения

данных на вычислительных узлах используется внешнее сетевое хранилище (Azure Storage Vault). Доступ к этим данным осуществляется с высокой пропускной способностью, близкой к скорости доступа к локальному диску.

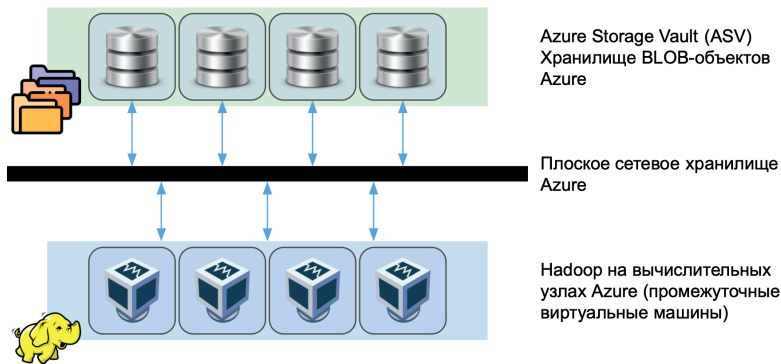


Рис. 153: HDInsight ³²

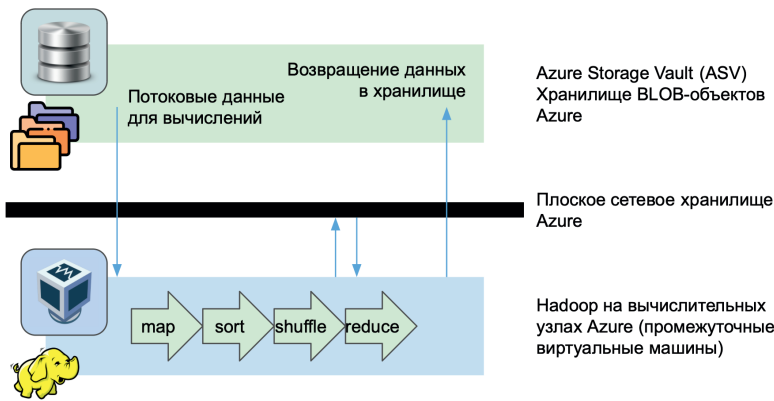


Рис. 154: Структура работы MapReduce в HDInsight ³³

Таким образом, в HDInsight отсутствует принцип **локальности данных**. Данные не хранятся на узлах кластера, а загружаются и сохраняются через

³²<https://dennyglee.com/2013/03/12/using-avro-with-hdinsight-on-azure-at-343-industries/>

³³<https://dennyglee.com/2013/03/18/why-use-blob-storage-with-hdinsight-on-azure/>

сетевое подключение. Mapper'ы читают их из сетевого хранилища, в процессе shuffle данные пересылаются между узлами кластера, выходные данные Reducer записываются в сетевое хранилище.

10.4 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

- В чем состоят отличия public от private-облаков?
- Чем различаются SaaS, PaaS, IaaS-решения?
- В чем состоит ключевое отличие архитектуры Hadoop HDInsight от оригинального Hadoop?

ГЛАВА 11

Введение в Apache Spark

В данной главе рассматривается популярная технология интерактивной обработки больших данных Apache Spark. В отличие от Hadoop, Spark позволяет обрабатывать данные в оперативной памяти, что значительно ускоряет выполнение итеративных алгоритмов и делает его интерактивным. Далее будут рассмотрены основные концепции Spark, его архитектура и ключевые компоненты, а также примеры использования языков Scala и Python для работы с данными в Spark.

11.1 Ограничение возможностей Hadoop

Как было сказано ранее, одно задание Hadoop Job MapReduce состоит из двух стадий: map и reduce. В подавляющем большинстве прикладных задач обработка больших данных подразумевает более сложные графы вычислений. Рассмотренный нами ранее алгоритм умножения матриц требует применения модели MapReduce дважды.

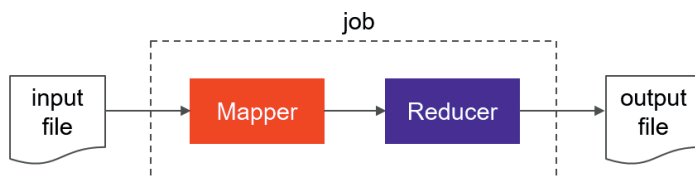


Рис. 155: Пример вычисления задания MapReduce

11.1.1 Сложные сценарии вычислений

В ряде случаев возникает необходимость организовать выполнение нескольких MapReduce-заданий в рамках единого вычислительного процесса. Рассмотрим основные подходы к построению таких сложных вычислительных сценариев:

1. **Последовательный запуск заданий в драйвере (Driver):**

Наиболее простой способ организации нескольких шагов вычислений — запуск заданий последовательно в рамках одной программы-драйвера. Следующее задание запускается после успешного завершения предыдущего.

2. **Использование класса JobControl:**

Библиотека Hadoop предоставляет вспомогательный класс `JobControl`, позволяющий определить зависимости между заданиями. С помощью метода `addDependingJob()` можно явно указать, какие задания должны завершиться до начала выполнения текущего.

3. **Использование Apache Oozie:**

Для организации более сложных вычислительных графов (в том числе с ветвлениями и параллельными путями) рекомендуется использовать инструмент Apache Oozie. Oozie позволяет описывать сценарии выполнения в виде графов зависимостей между заданиями, задаваемых с помощью конфигурационных файлов. Поддерживаются не только MapReduce-задания, но и другие компоненты Hadoop-экосистемы.

Объектная модель MapReduce в Hadoop предусматривает возможность последовательного выполнения нескольких **Mapper**-ов и **Reducer**-ов в рамках одного задания без создания промежуточных файлов на диске. Для этого используются специализированные классы `ChainMapper` и `ChainReducer`, позволяющие организовать цепочку вызовов обработчиков данных.

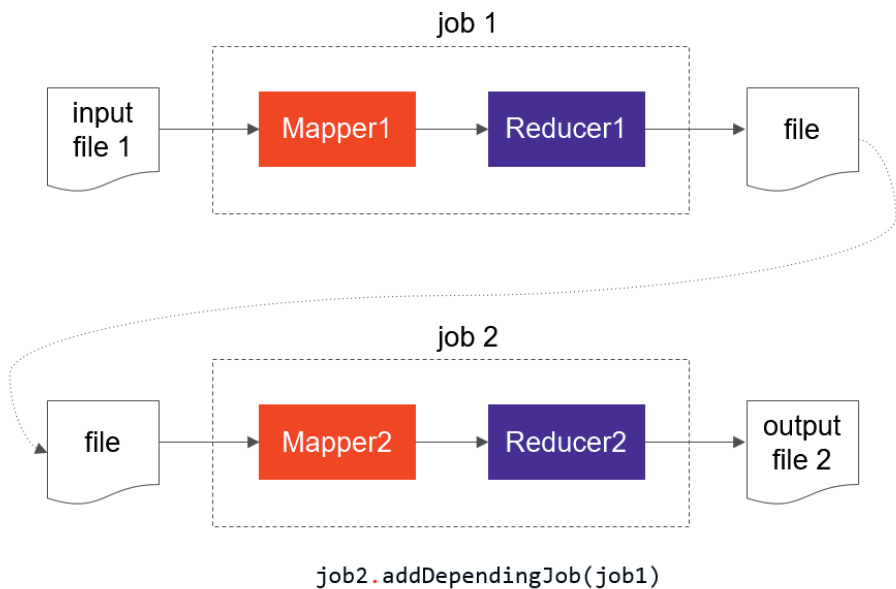


Рис. 156: Пример установки зависимостей между заданиями



Рис. 157: Apache Oozie

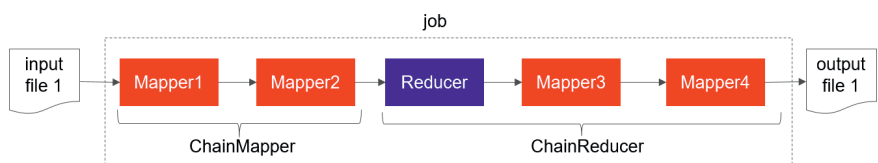


Рис. 158: Пример внедрения цепочек Map и Reduce в модель вычисления

Также широко используется класс `MultipleInputs`, который позволяет брать входные данные из файлов с разными форматами хранения и обрабатывать их разными mapper'ами (потенциально с помощью разных алгоритмов), передавая в общий reduce. Аналогично используется класс `MultipleOutputs`, позволяющий настраивать запись выходных данных в несколько файлов согласно заданной логике.

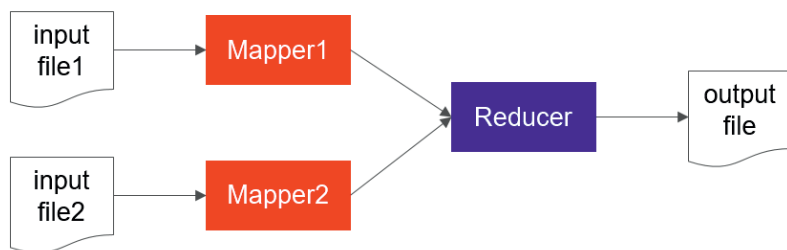


Рис. 159: Пример задания с `MultipleOutputs`

11.1.2 Итеративные алгоритмы

Однако, конструировать произвольные графы вычислений и отлаживать их работу в Hadoop MapReduce не всегда удобно. В частности, проблемой Hadoop являются **итеративные алгоритмы**. Они подразумевают проход по данным в несколько итераций. В случае больших данных и MapReduce это приводит к повторному чтению данных из HDFS. Итеративные алгоритмы характерны для машинного обучения.

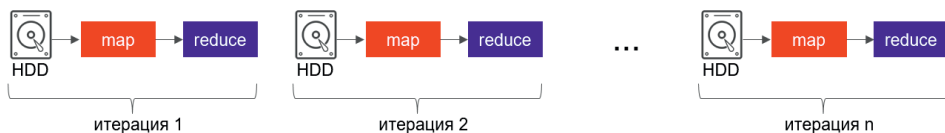


Рис. 161: Пример выполнения итеративных алгоритмов

К **однопроходным алгоритмам** относятся:

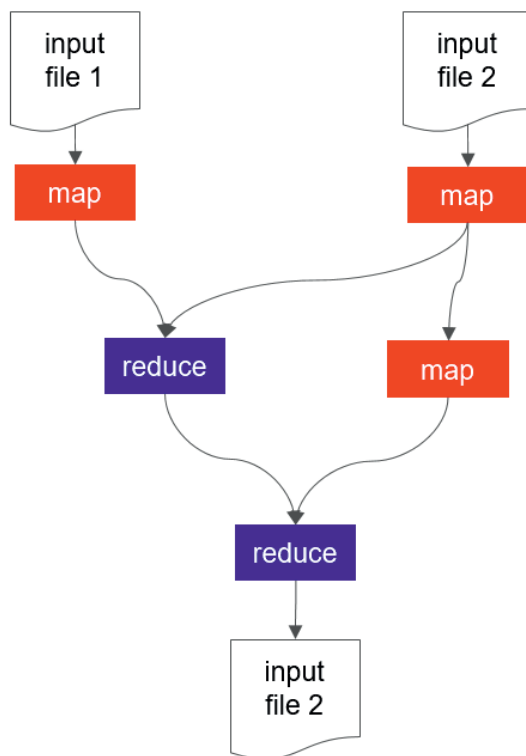


Рис. 160: Пример графа вычислений

- Операции реляционной алгебры;
- Grep;
- Трансформация данных в другой формат;
- Расчет простейших статистик.

К **итеративным алгоритмам** относятся алгоритмы линейной алгебры, методов оптимизации и машинного обучения:

- Кластеризация (K-means, EM и т.д.);

- Градиентный спуск (в том числе стохастический);
- Поиск частых предметных наборов (Apriori, FP-Growth и т.д.);
- PageRank и другие графовые алгоритмы;
- Матричные разложения (UV, SVD и т.д.).

11.1.3 Интерактивность

Apache Hadoop не интерактивен. Реализация алгоритма подразумевает написание и компиляцию программы на Java с перегрузкой функций и реализацией интерфейсов. На текущий момент времени аналитики привыкли работать с динамическими языками программирования (такими как Python) в Jupyter-notebooks для интерактивной обработки данных.

11.1.4 Недостатки Hadoop

Таким образом, можно выделить следующие недостатки Hadoop:

- Hadoop ориентирован на решение одного задания в рамках одной программы MapReduce;
- Hadoop не эффективен в задачах, где требуется повторное чтение данных (итеративные алгоритмы занимают значимую долю алгоритмов машинного обучения). Однако, существует API для объединения нескольких MapReduce-программ (`Job.addDependingJob(Job)`, `JobControl`, `ControlledJob`, `ChainMapper`, `ChainReducer`);
- Hadoop MapReduce не интерактивен и подразумевает написание программы на Java.

```

package org.myorg;
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
public class WordCount {
    public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job(conf, "wordcount");
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.waitForCompletion(true);
    }
}

```

Рис. 162: Код на Java [1]

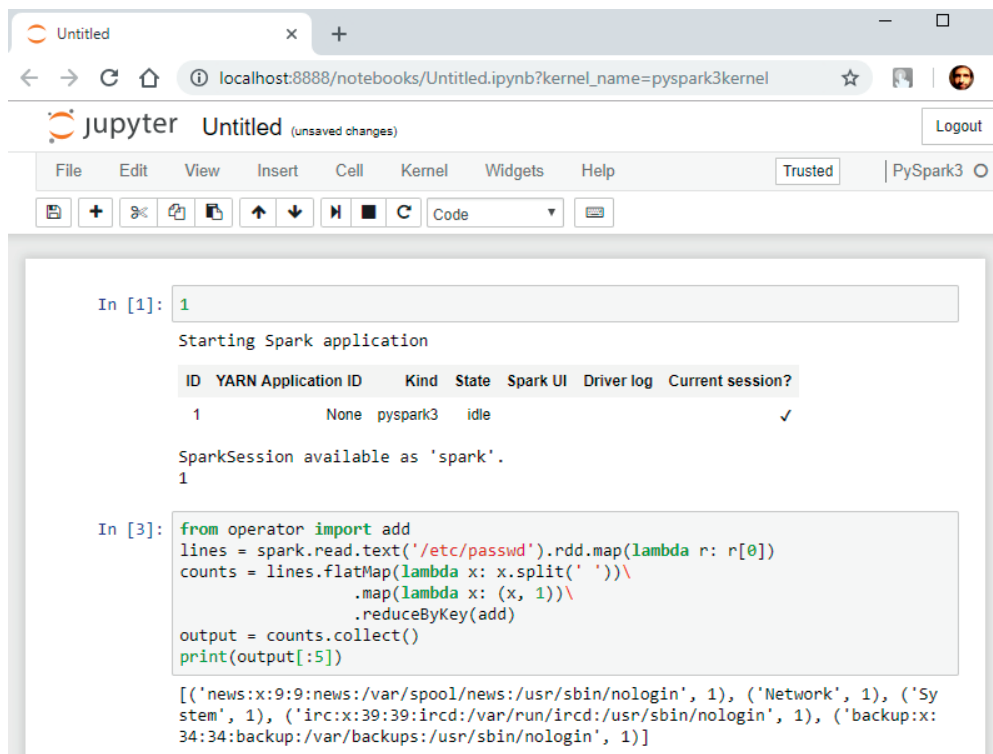


Рис. 163: Spark в Jupyter-notebook

11.2 Возможности Apache Spark



Рис. 164: Apache Spark³⁴



Рис. 165: Матей Захария³⁵



Рис. 166: Amplab³⁶

Проект **Apache Spark** был представлен научной лабораторией университета Беркли в 2012 г. Ключевой фигурой в его создании является Матей Захария — ученый Стэнфордского университета, создавший компанию Databricks, которая занимается системной интеграцией на основе Apache Spark и его дальнейшим развитием.

Одной из **ключевых идей Spark** является кэширование прочитанных из HDFS данных, а также результатов их промежуточной обработки в оперативной памяти. Этот подход позволил в разы ускорить работу итеративных алгоритмов по сравнению с Hadoop.

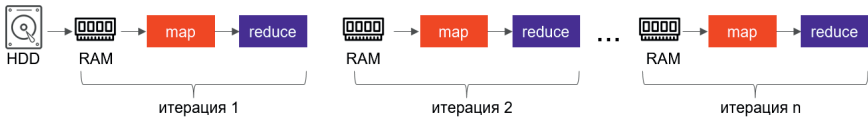


Рис. 167: Пример итеративных алгоритмов с использованием оперативной памяти

³⁴<https://spark.apache.org/>

³⁵<https://www.cidrdb.org/cidr2022/keynotespeakers.html>

³⁶<http://amplab.ru/>

В качестве литературы по Spark рекомендуется оригинальная статья Матея Захария и курса Algorithms, Machines and People Lab³⁷ [15].

Также сегодня существует множество книг про Spark:

- Matei Zaharia at al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing [15];
- Data Science and Engineering with Apache Spark³⁸;
- Spark Programming Guide³⁹.

11.2.1 Функциональное программирование

Spark во многом опирается на парадигму функционального программирования (ФП). В языке Python существуют реализованные в стандартной библиотеке функции `map` и `reduce`. Далее будут приведены их реальные реализации, понятные опытным пользователям Python, а также их упрощенный вид.

Функция `map` применяет к каждому элементу последовательности (например, списка) функцию и возвращает новую последовательность. Функция и последовательность являются аргументами `map`. Функция `map` близка к исполнению стадии Map в MapReduce, в которой к каждому элементу последовательности применялось некоторое действие и образовывалась новая последовательность. Функция `reduce` сворачивает последовательность в скалярное значение с помощью переданной в нее двуместной агрегационной функции. Переданная функция последовательно применяется к элементам последовательности и накапливает результат агрегации.

Важно отметить, что в обеих функциях отсутствует модификация данных.

³⁷<https://amplab.cs.berkeley.edu/>

³⁸<https://github.com/XIEQ/DataScienceEngineeringApacheSpark?ysclid=md8yemqftw501798477>

³⁹<http://spark.apache.org/docs/latest/programming-guide.html>



Рис. 168: Книга «Изучаем Spark. Молниеносный анализ данных» (2015). Авторы: Захария М., Венделл П., Конвински Э. [16]



Рис. 169: Книга «Эффективный Spark. Масштабирование и оптимизация» (2018). Авторы: Карау Х., Уоррен Р. [17]

Реальная реализация:

```
def map(function, iterable):
    for element in iterable:
        yield function(element)
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

Упрощенное представление:

```
def map(function, l):
    l2 = []
    for element in l:
        l2.append(function(element))
    return l2
def reduce(function, l, initializer=None):
    if initializer is None:
        value = l[0]
        start = 1
    else:
        value = initializer
        start = 0
    for element in l[start:]:
        value = function(value, element)
    return value
```

Рис. 170: Мартин Одерски ⁴⁰

Пример работы:

```
a = [1, 2, 3]
b = list(map(lambda x: x*2, a))
c = reduce(lambda x, y: x+y, b)
print(b, c)
>>> [2, 4, 6] 12
```

Функции `map` и `reduce` являются иллюстрациями того, как функциональное программирование реализовано в языке Python. Функциональное программирование исключает модификацию данных, основывается на применении функций первого порядка (способны получать другие функции как аргументы), которые применяются к последовательностям объектов. Из-за отсутствия модифицируемого состояния функциональные программы легко подвергаются параллелизации. Идея обработки последовательностей объектов в функциональном стиле является центральной идеей Apache Spark.

11.2.2 Scala

Spark написан на языке Scala, созданным Мартином Одерски — учеником Николаса Вирта в 2003 году. Scala можно назвать статически типизированным Python, исполняемым на виртуальной машине Java с встроенными коллекциями, поддерживающими многопоточную обработку. Scala поддерживает

⁴⁰<https://insait.ai/prof-martin-odersky/>

парадигму **expression-oriented programming (EOP)** — выражения могут быть вычислены и возвращены как результат функции. Также, язык предоставляет много интересных механизмов (например, сопоставление с образцом — *pattern matching*), которые позволяют писать лаконичный и выразительный код. Благодаря поддержке JVM, в Scala можно использовать весь спектр Java-библиотек.

К **характеристикам Scala** относятся:

- Объектно-ориентированный ЯП;
- Функциональный подход и *expression-oriented programming (EOP)*;
- Поддержка JVM и Java-библиотек;
- “Статически типизированный Python” (возможность вывода типов, отсутствие необходимости указывать типы данных);
- “Some batteries included” благодаря новым проектам на Scala и существующим на Java (векторы и матрицы, ML, визуализация);
- Наличие коллекций с *multithreading* и параллельными вычислениями.

11.2.3 Примеры программирования на Scala

Определение константы:

```
val pi = 3.1415926
```

Определение и использование переменных:

```
var counter = 0  
counter += 1
```

Указание типа:

```
var text: String = "scala"
```

Указание типа “object”:

```
var text: Any = "scala"
```

Expression-oriented programming (EOP) — блоки являются выражениями:

```
var result = { var x = 1; var y = 2; x + y }  
val result2 = if (result > 1) 5 else 2
```

Использование типа Unit — аналога “void”:

```
var result = { var x = 1; x = 3 } // тип Unit, константа ()
```

Вызов функции в Scala похож на использование выражения: точку и скобки при обращении к функции можно опускать. Это и есть еще одно проявление EOP.

Использование операторов / методов:

```
val a = 1  
val b = 2  
a + b  
a.+(b)
```

Scala минималистична с точки зрения необходимости указывать типы данных. В большинстве ситуаций типы данных выводятся автоматически.

Определение функции:

```
def fac(n : Int) = {  
  var r = 1  
  for (i <- 1 to n) r *= i  
  r // или return r, но с указанием типа результата  
}
```

В рекурсивных функциях требуется указывать тип результата:

```
def fac(n : Int) : Int = if (n <= 0) 1 else n * fac(n - 1)
```


Примеры обработки последовательностей в функциональном стиле, упрощенной формы записи лямбда-выражений, использования механизма сопоставления с образцом:

```
val x = 1 to 10
val evens = x.filter((i: Int) => i % 2 == 0)
val evens_v2 = x.filter(_ % 2 == 0)

val numbers = Array(1, 2, 3, 4, 5)
val prod = numbers.reduce(_*_)

val two = 2
val x = List(10,20,30)
val res = (x : Any) match {
  case 1 => "one"
  case i: Int => "int"
  case l: List[_] => l.head
  case two => "two"
}
print(res)
```

11.2.4 Пример работы с PySpark

Поскольку Spark написан на Scala, основным языком для написания программ в нем является Scala. Однако Spark поддерживает и другие ЯП, в том числе популярные среди аналитиков Python и R. Дальнейшие примеры приведены с использованием библиотеки PySpark, которая предоставляет возможность использовать Spark из Python.

Рассмотрим файл журнала, каждая запись которого содержит идентификатор пользователя, дату захода (клика) на некоторый сайт, URL сайта и время захода, выраженное в секундах относительно даты захода. Требуется найти домены, встречающиеся не менее, чем в 100 кликах. Рассмотрим решение этой задачи с использованием PySpark.

0	20141027	http://www.aif-nn.ru/society	45187
0	20141027	http://gmstar.ru/nn	51765
0	20141027	http://horo.mail.ru/predictionaries/today/	41305
0	20141027	http://www.irecommend.ru/content	52231
0	20141027	http://nnovmed.ru/review	52019

Рис. 171: Пример файла журнала, содержащего идентификатор пользователя, дату клика, URL сайта и время клика

Задача: найти домены, встречающиеся в 100+ кликах

Первоначально необходимо создать объект класса `SparkContext`, который олицетворяет соединение со Spark-кластером. При создании `SparkContext` можно указать конфигурационные параметры. В рамках данной задачи укажем параметр возможности перезаписи существующих в HDFS файлов по ходу обработки.

```
import pyspark
conf = pyspark.SparkConf()

# Возможность перезаписывать выходную директорию:
conf.set("spark.hadoop.validateOutputSpecs", "false")

sc = pyspark.SparkContext(conf=conf)
```

Spark доступен напрямую через Spark-консоль, также есть возможность настроить доступ к Spark через Jupyter-notebook. Во многих случаях создание `SparkContext` вручную не требуется: среда разработки создает его самостоятельно.


```
top_stat = domain_stat.filter(lambda d: d[1] >= 100)
top_stat.saveAsTextFile("top_stat.txt")
```

RDD `lines` формируется методом `textFile(...)` `SparkContext`. С настройками по умолчанию число секций `lines` будет равно числу блоков файла. Секции этой RDD окажутся на тех же узлах кластера, где находятся соответствующие блоки файла.

Преобразование `map` применит переданную функцию к каждому элементу `lines`. Элементом `lines` является строка файла, переданная функция определяет 3-й элемент строки (элементы разделены табуляцией). Результатом является RDD `urls`. Следующее преобразование RDD определит из URL доменное имя и сформирует RDD `domain_count` из пар “домен, единица”. Преобразование `reduceByKey` сгруппирует последовательность пар по первому элементу пары — ключу и посчитает значение переданной в `reduceByKey` агрегационной функции. В данном случае — суммы. Далее происходят фильтрация по частоте и запись в файл.

Следует обратить внимание на то, что функции `map`, `reduceByKey`, `filter` (называются **трансформациями**) не приводят к какой-либо обработке исходных данных при их вызове. Они конструируют направленный ациклический граф вычислений (DAG), а реальная обработка данных происходит при вызове функции **действия** (например, `saveAsTextFile`). Это соответствует концепции **отложенных вычислений**. В документации по Spark описаны все функции-трансформации и действия.

На Рисунке 172 изображена последовательность преобразований данных. Каждый квадрат — это секция RDD (в предположении, что изначально их 4). Трансформация `reduceByKey` требует тасовки. В ее втором параметре можно указать число секций, которые будут созданы в итоговой RDD. Это число соответствует числу reducer’ов, использующихся при тасовке.

```
lines = sc.textFile("path_to_file")
urls = lines.map(lambda l: l.split('\t')[2])
```


памяти либо на диске. Это определяется параметром функции `persist`.

При вытеснении секций RDD из кэша используется стратегия `least recently used` (LRU). Функция `cache` является синонимом обращения к `persist` с параметром `MEMORY_ONLY`.

```
lines = sc.textFile("path_to_file")
urls = lines.map(lambda l: l.split('\t')[2])

from urllib.parse import urlparse
domain_count = urls.map(lambda u: (urlparse(u).hostname, 1))
domain_stat = domain_count.reduceByKey(lambda x, y: x + y,
                                       numPartitions = 3)
top_stat = domain_stat.filter(lambda d: d[1] >= 100)
top_stat.persist(MEMORY_ONLY)
top_stat.count()
top_stat.saveAsTextFile("top_stat.txt")
```

При вытеснении секций из памяти используется LRU:

```
RDD.persist(level=MEMORY_ONLY)
RDD.cache()
RDD.unpersist()
```

Кэширование в Spark позволяет сгладить дисбаланс скоростей вычислительных компонент за счет использования оперативной памяти. Кэширование имеет смысл при повторном использовании RDD, в том числе — в итеративных алгоритмах.

Таблица 23: Таблица значений параметра level функции persist

Уровень персистентности	Описание
MEMORY_ONLY	По умолчанию. Хранение разделов RDD в памяти. Разделы вне памяти пересчитываются
MEMORY_AND_DISK	Хранение разделов RDD в памяти. Разделы вне памяти сохраняются на диск
MEMORY_ONLY_SER	Аналогично MEMORY_ONLY, но используется сериализация (сокращение объема, увеличение процессорной нагрузки)
MEMORY_AND_DISK_SER	Аналогично MEMORY_AND_DISK, но используется сериализация (сокращение объема, увеличение процессорной нагрузки)
DISK_ONLY	Хранение разделов RDD на диске
MEMORY_ONLY_2, MEMORY_DISK_2 и т.д.	“Реплицирование” разделов на исполнителях на других узлах
OFF_HEAP (experimental)	Общий пул памяти Tachyon ⁴¹
Уменьшение GC	Раздел не теряется при сбое узла



Рис. 173: 50ГБ/сек (1x)⁴² Рис. 174: 100МБ/- сек (500x)⁴³ Рис. 175: 500МБ/- сек (100x)⁴⁴



Рис. 176: 1ГБ/сек (50x)⁴⁵ Рис. 177: 0.3ГБ/сек (166x)⁴⁶

11.2.7 Стадии и задачи RDD

При преобразованиях RDD в Spark различают узкие и широкие зависимости. Узкая зависимость возникает при трансформациях RDD типа map.

⁴²https://en.wikipedia.org/wiki/DDR3_SDRAM

⁴³https://en.wikipedia.org/wiki/Hard_disk_drive

⁴⁴<https://intikaliskiri.blogspot.com/2016/11/mengupgrade-notebook-dengan-ssd.html?showComment=1615955801596>

⁴⁵<https://eizovisualsolutions.com/monitors/software/radinet-pro-web-hosting/>

⁴⁶https://www.niap-ccevs.org/MMO/Product/st_vid111278-agd.pdf

Цепочки трансформаций с узкими зависимостями происходят на тех же узлах кластера, где расположены секции исходной RDD.

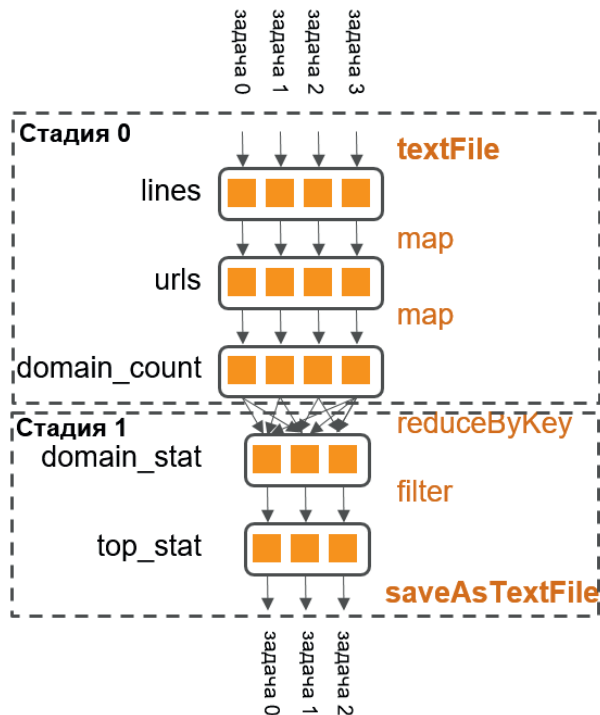


Рис. 178: Пример разметки границ стадий для задачи обработки файлов журнала

Широкие зависимости соответствуют тасовке в MapReduce. При такой трансформации RDD происходит пересылка данных по сети, на узлах кластера появляются секции новой RDD, число секций определяется соответствующим параметром трансформации с широкой зависимостью. В терминологию Spark входит понятие стадии. Это часть DAG, ограниченная тасовкой, уже закэшированной RDD либо действием. В рамках каждой стадии возникает понятие **задач стадии**. Число задач стадии соответствует числу секций, которые подвергаются цепочке преобразований. В примерах стадии имеют 4 и 3 задачи соответственно.

Узкая (narrow) зависимость — секция используется при расчете не более чем одной “дочерней” секции.



Рис. 179: Пример узкой зависимости

Широкая (wide) зависимость — секция может быть использована более чем одной “дочерней” секцией.



Рис. 180: Пример широкой зависимости

`RDD.debugString()` — метод, отображающий зависимости⁴⁷.

11.2.8 Варианты запуска stand-alone программы (-master)

В кластере Spark представлен компонентом **драйвер** (в ней создается `SparkContext`) и множеством компонент типа **исполнитель**, которые выполняют задачи графов вычислений.

⁴⁷<http://localhost:4040/> — Spark UI

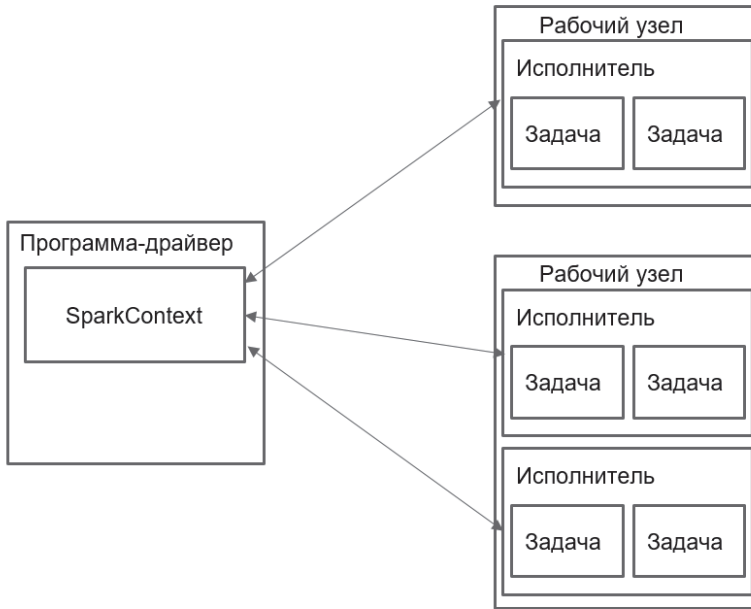


Рис. 181: Пример компонентов Spark: драйвер и исполнители [18]

Программа на Spark может быть отправлена на вычисление с помощью утилиты `spark-submit`. Она позволяет работать с разными вариантами развёртывания Spark:

- Spark Stand-alone — вариант Spark без использования внешнего планировщика;
- Spark на Mesos — вариант Spark, использующий Mesos в качестве планировщика ресурсов;
- Spark на YARN — вариант Spark, использующий YARN в качестве планировщика ресурсов;
- Spark на локальном компьютере.

Таблица 24: Таблица вариантов запуска stand-alone программы

Тип URL	Описание
spark://host:port	Адрес диспетчера кластера Spark Stand-alone (порт обычно 7077)
mesos://host:port	Адрес диспетчера кластера Mesos (порт обычно 5050)
yarn	Адрес ResourceManager YARN берется из конфигурационных файлов, находящихся в HADOOP_CONF_DIR
local	Запуск в локальном режиме на 1 ядре
local[N]	Запуск в локальном режиме на N ядрах
local[*]	Запуск в локальном режиме на всех ядрах

Общий вид:

```
bin/spark-submit [опции]
                  <jar|python файл>
                  [аргументы_командной_строки]
```

Пример:

```
bin/spark-submit --master spark://host:7077
                 --executor-memory 10g
                 my_script.py
```

Наконец, возможно использовать Spark через интерактивный интерпретатор. Ниже представлены команды для запуска интерпретатора на разных языках программирования.

Scala: bin/spark-shell

Python: bin/pyspark (для выбора Python нужной версии используется переменная окружения PYSPARK_PYTHON).

11.3 Запуск на YARN

На YARN Spark может быть запущен в клиентском либо серверном режиме. В клиентском режиме Spark Driver находится за пределами кластера и взаимодействует со Spark Application Master, созданным в контейнере внутри кластера.

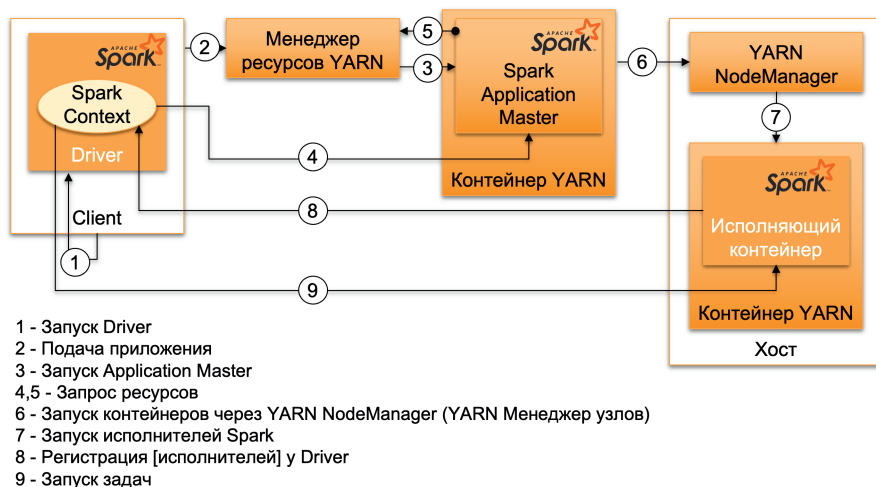


Рис. 182: Запуск на YARN: client mode ⁴⁸

В кластерном режиме Spark Driver и Spark Application Master соседствуют в общем контейнере внутри кластера.

11.3.1 Первичное создание RDD

Кратко рассмотрим основные методы по работе с RDD (см. Таблицу 25). Первичное создание RDD возможно на основе файла с данными либо на основе нераспределенного списка объектов. Spark поддерживает те же форматы, что и Hadoop: они используют общую библиотеку форматов данных.

⁴⁸https://aws.amazon.com/pt/blogs/big-data/submitting-user-applications-with-spark-submit/?nc1=h_ls

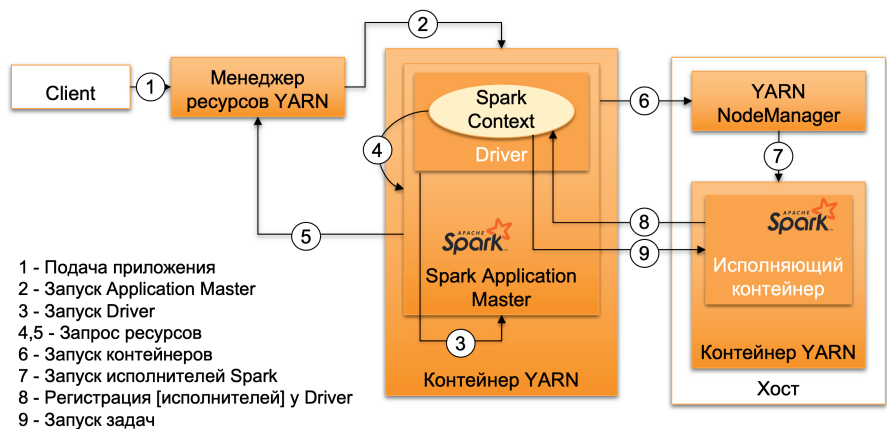


Рис. 183: Запуск на YARN: cluster mode

Таблица 25: Методы SparkContext

Название	Описание
SparkContext.textFile(name, minPartitions=None, use_unicode=True)	Чтение данных из файла с использованием Hadoop TextInputFormat. Число разделов определяется числом сплитов, либо задается явно
SparkContext.parallelize(c, numSlices=None)	Создание RDD на основе обычной коллекции Python

11.3.2 Преобразования одной RDD

Перечисленные в Таблице 26 методы выполняют трансформацию одной RDD. `FlatMap` отличается от `map` тем, что позволяет для каждого объекта входной RDD вернуть ноль, один или несколько объектов, которые войдут в выходную RDD. В частности, `flatMap` может быть полезен для использования фильтрации по условию при обработке RDD.

Таблица 26: Методы преобразования Resilient Distributed Dataset

Название	Описание
<code>RDD.map(action)</code>	Возвращает RDD из результатов применения <code>action: x→y</code> к каждому элементу исходной RDD
<code>RDD.flatMap(action)</code>	Применяет <code>action: x→[y1,...,yn]</code> к каждому элементу исходной RDD, возвращает RDD, состоящую из элементов всех списков
<code>RDD.filter(cond)</code>	Возвращает RDD из элементов исходной RDD, удовлетворяющих условию <code>cond</code>
<code>RDD.distinct()</code>	Возвращает RDD из уникальных элементов исходной RDD
<code>RDD.sample(withReplacement, fraction, seed=None)</code>	Возвращает подмножество RDD (сэмплинг)

В Spark реализованы основные теоретико-множественные операции (преобразования двух RDD):

- `RDD.union(other);`
- `RDD.intersection(other);`
- `RDD.subtract(other);`

- `RDD.cartesian(other)`.

11.3.3 Действия над RDD

Самыми распространенными действиями при работе с RDD являются: сохранение в файл, подсчет числа элементов RDD, `collect` — “материализация” RDD целиком (преобразование RDD в локальный список на драйвере) и `take` — частичная материализация n первых элементов RDD.

Таблица 27: Действия над RDD

Название	Описание
<code>RDD.saveAsTextFile(path, compressionCodecClass=None)</code>	Запись RDD в текстовый файл
<code>RDD.collect()</code>	“Материализация” RDD в Python-список
<code>RDD.count()</code>	Число элементов
<code>RDD.countByValue()</code>	Сколько раз встречается каждое значение (возвращает список пар)
<code>RDD.take(num)</code>	Список из первых num элементов
<code>RDD.top(num)</code>	Список из num элементов в порядке убывания значений
<code>RDD.takeOrdered(num, ordering)</code>	Список из num элементов в порядке убывания значений согласно заданному отношению порядка
<code>RDD.takeSample(withReplacement, num, seed=None)</code>	Аналог <code>sample</code>
<code>RDD.foreach(func)</code>	Выполнить действие над каждым элементом

К действиям также относятся свертки RDD в скалярное значение (см. Таблицу 28). `Fold` позволяет сворачивать пустые RDD, используя нулевое

значение. **Aggregate** разделяет свертку на операции свертки внутри секции и между секциями.

Таблица 28: Методы свертки Resilient Distributed Dataset

Название	Описание
<code>RDD.reduce(func)</code>	“Свертка” с помощью заданной бинарной функции. Не сработает для пустой RDD
<code>RDD.fold(zero, func)</code>	“Свертка” с помощью заданной бинарной функции и нуля
<code>RDD.aggregate(zeroValue, seqOp, combOp)</code>	Аналог <code>fold</code> , но операция разбивается на “свертку” внутри секции (<code>seqOp: U,T→U</code>) и последующую “свертку” между секциями (<code>combOp: U,U→U</code>). Позволяет вычислить несколько агрегаций одновременно. Например, среднее с помощью суммы и количества: <pre>sum_count = rdd.aggregate((0, 0), lambda acc, value: (acc[0] + value, acc[1] + 1), lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1])) sum_count[0] / sum_count[1]</pre>

11.3.4 Механизм контрольных точек (Checkpoint)

Механизм контрольных точек позволяет сохранять RDD в HDFS (либо локальной ФС исполнителей) и автоматически считывать при повторном вычислении. Является альтернативой кэшированию RDD на диске и сохранению RDD в HDFS вручную. Отличается от кэширования тем, что контрольная точка удаляет преобразования, предшествующие формированию RDD. Контрольные точки имеют преимущества над кэшированием в нагруженном

заданиями кластере, в котором часто происходит вытеснение RDD из кэша.

Использование контрольных точек лучше кэширования в случае частых сбоев, а также лучше сохранения в HDFS, так как не нужно явно считывать RDD. Перечислим основные методы для создания контрольных точек и работы с ними:

- `RDD.checkpoint()` — создание контрольной точки;
- `RDD.getCheckpointFile()` — получение пути к контрольной точке;
- `RDD.localCheckpoint()` — создание локальной контрольной точки (в локальной ФС исполнителя, не в HDFS);
- `RDD.isCheckpointed()` — проверка, создана ли контрольная точка для RDD;
- `RDD.isLocalCheckpointed()` — проверка, создана ли локальная контрольная точка для RDD;
- `SparkContext.setCheckpointDir()` — установка директории для контрольных точек.

11.3.5 Сохранение результатов стадии shuffle

При выполнении Spark-приложений можно увидеть визуализированные графы вычислений в интерфейсе Spark UI. При повторном запуске действий над теми же данными можно заметить, что часть стадий отображается серым цветом. Это связано с тем, что Spark автоматически сохраняет выходные данные стадий на границах между этапами shuffle.

В частности, выходные данные задач типа map (до передачи на стадии reduce) сохраняются в виде отсортированных файлов на локальных дисках вычислительных узлов. Такие файлы сохраняются до завершения всего Spark-приложения и могут быть повторно использованы без перерасчета при

повторных действиях над теми же данными. Этот механизм можно рассматривать как внутреннее кэширование, инициируемое самим Spark.

Прямого способа узнать в RDD API, сохранены ли промежуточные результаты, не предусмотрено. Однако такую информацию можно получить через интерфейс Spark UI, где сохраненные стадии выделяются серым цветом.

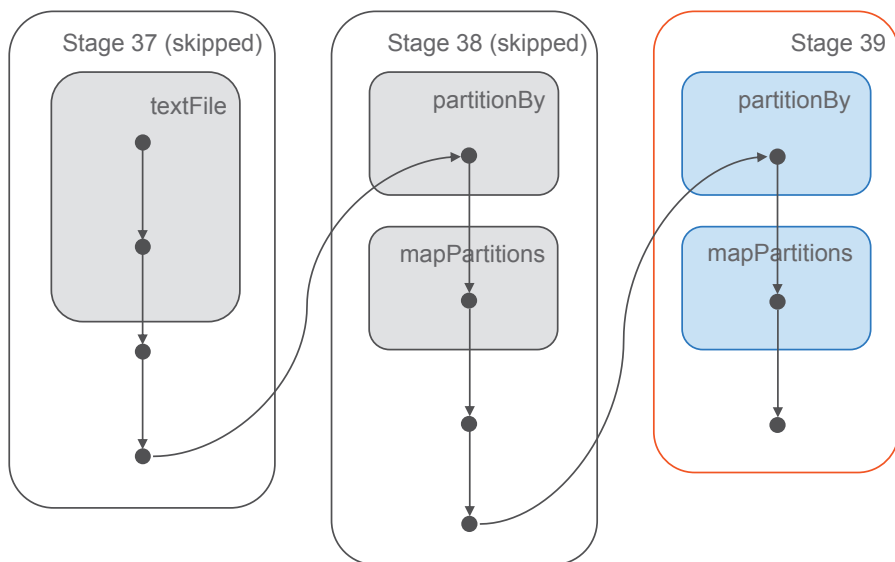


Рис. 184: Визуализированные графы вычислений [17]

11.3.6 Обработка пар “ключ-значение”

Рассмотрим особенности работы с трансформациями RDD в Apache Spark, ориентированными на структуры данных в виде пар “ключ-значение”. Эти трансформации применимы к RDD, элементы которых представляют собой кортежи (tuple), где первый элемент интерпретируется как ключ, а второй — как соответствующее ему значение.

Работа с такими структурами позволяет реализовывать целый ряд типичных операций обработки данных, среди которых:

- Агрегирование и группировка по ключу — объединение всех значений, соответствующих одному ключу, в обобщенную форму (например, в список или агрегированное значение);
- Операции соединения (join) — объединение двух RDD по общим ключам с образованием пар значений (Таблица 29);
- Сортировка по ключу — упорядочивание элементов RDD на основе значений ключей (Таблица 30).

Таблица 29: Методы соединения RDD

Название	Описание
<code>RDD.join(other, numPartitions=None)</code>	Внутреннее соединение
<code>RDD.leftOuterJoin(other, numPartitions=None)</code>	Левое внешнее соединение
<code>RDD.rightOuterJoin(other, numPartitions=None)</code>	Правое внешнее соединение

Таблица 30: Методы сортировки RDD

Название	Описание
<code>RDD.sortBy(keyfunc, ascending=True, numPartitions=None)</code>	Сортировка RDD
<code>RDD.sortByKey(ascending=True, numPartitions=None, keyfunc=None)</code>	Сортировка RDD “ключ-значение” по ключу

Концепция функций агрегации/группировки по ключу близка к глобальным сверткам (`reduce`, `fold`, `aggregate`). Функции принимают на вход определенную пользователем логику секционирования результирующей RDD по

ключам и число секций. `AggregateByKey` и `combineByKey` позволяют разбить вычисление агрегата на стадии `combine` и `reduce`.

Таблица 31: Методы (функции) агрегации/группировки по ключу RDD

Название	Описание
<code>RDD.reduceByKey(func, numPartitions=None, partitionFunc=<function portable_hash>)</code>	“Свертка” значений для каждого ключа (<code>func: T, T → T</code>)
<code>RDD.foldByKey(zeroValue, func, numPartitions=None, partitionFunc=<function portable_hash>)</code>	“Свертка” значений для каждого ключа с указанием элемента-нуля (<code>func: T, T → T</code>)
<code>RDD.aggregateByKey(zeroValue, seqFunc, combFunc, numPartitions=None, partitionFunc=<function portable_hash>)</code>	Аналог <code>foldByKey</code> , но операция разбивается на “свертку” внутри секции (<code>seqOp: U, T → U</code>) и последующую “свертку” между секциями (<code>combOp: U, U → U</code>). Позволяет вычислить несколько агрегаций одновременно (см. <code>RDD.aggregate</code>)
<code>RDD.combineByKey(createCombiner, mergeValue, mergeCombiners, numPartitions=None, partitionFunc=<function portable_hash>)</code>	Объединение элементов для каждого ключа с использованием пользовательского набора функций агрегирования.
<code>RDD.groupByKey(numPartitions=None, partitionFunc=<function portable_hash>)</code>	Группировка значений по ключу. Не использует <code>combine</code> на стороне mappers! Использовать для холистических функций

Отличие проявляется в прототипах. `GroupByKey` стоит применять лишь в той ситуации, когда желаемая агрегация не может быть вычислена по частям и требует пересылки всех данных по сети. В остальных ситуациях более производительной будет реализация на основе любой другой перечисленной функции. Рассмотрим **пример** вычисления среднего на основе `RDD.combineByKey()`:

```
data = sc.parallelize([(0, 2.), (0, 4.), (1, 0.),
                      (1, 10.), (1, 20.)])

sumCount = data.combineByKey(
    lambda value: (value, 1),
    lambda x, value: (x[0] + value, x[1] + 1),
    lambda x, y: (x[0] + y[0], x[1] + y[1]))

averageByKey = sumCount.map(lambda
    (label, (value_sum, count)):
    (label, value_sum / count))
```

11.3.7 Производительность операций с парами “ключ-значение”

При выполнении операций соединения (`join`) двух RDD по ключу, данные обеих RDD, как правило, перераспределяются между узлами кластера. Это связано с необходимостью группировки данных по ключу и их последующего совмещения.

```
yearly_stat = sc.textFile("/yearly_stat.tsv")
daily_stat = sc.textFile("/20181126_stat.tsv")
joined = yearly_stat.join(daily_stat)
...

print(yearly_stat.getNumPartitions())
>>> 100
```

```
print(daily_stat.getNumPartitions())
>>> 5
print(yearly_stat.partitioner)
>>> None
print(daily_stat.partitioner)
>>> None
```

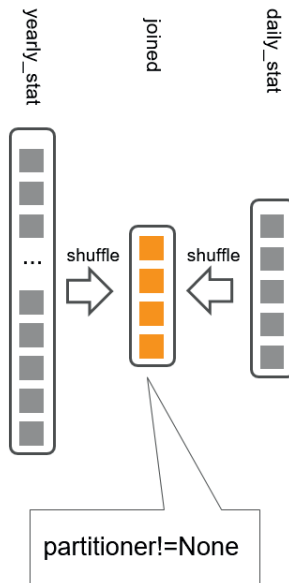


Рис. 185: Пример пересылки значений

Для повышения производительности этой операции необходимо:

1. Применить к каждой из исходных RDD трансформацию `partitionBy(...)`, которая выполняет секционирование данных по ключу с заданным числом секций;
2. Сохранить результат секционирования в памяти с помощью `cache()`;
3. Выполнить операцию `join` в рамках одного действия (`action`).

При соблюдении этих условий обе RDD будут иметь одинаковую схему секционирования, что гарантирует совместное расположение соответствующих данных на узлах кластера. В этом случае операция соединения выполняется локально на каждом узле без дополнительной передачи данных, что обеспечивает значительный прирост производительности.

Секционирование по ключу можно определить по значению атрибута `partitioner` у RDD. Если значение атрибута не `None`, это означает, что RDD секционирована, и `partitioner` содержит функцию, использованную для секционирования.



Рис. 186: Пример секционирования по ключу

```
yearly_stat = sc.textFile("y_stat").map(...).partitionBy(50)
yearly_stat.cache()

daily_stat = sc.textFile("d_stat").map(...).partitionBy(50)
daily_stat.cache()

joined = yearly_stat.join(daily_stat)
...
```


Широко используются следующие понятия, связанные с секционированием RDD. **Co-partitioned RDD (одинаково секционированные RDD)** — RDD используют одинаковый `partitioner` с одинаковым числом секций. **Co-located RDD (совместно расположенные RDD)** — co-partitioned RDD, для которых соответствующие секции находятся на общих узлах.

Атрибут `partitioner` порождается трансформациями, подразумевающими пересылку данных: это агрегации по ключу, соединения и `partitionBy`. После выполнения операций типа `map`, `flatMap` и т.п. `partitioner` обнуляется, т.к. эти функции потенциально могут изменить значения ключей и нарушить логику секционирования по ключу.

Также в Spark есть отдельные трансформации `mapValues`, `flatMapValues` — аналоги `map` и `flatMap`. Их прототип исключает возможность модификации ключей.

11.3.8 Дополнительные возможности

Дополнительными возможностями к интерфейсу RDD являются:

- **Аккумуляторы** — аналог счетчиков Hadoop MapReduce;
- **Широковещательные переменные (broadcast variables)** — read-only объекты, которые копируются в оперативную память каждого исполнителя. Это аналог класса `DistributedCache` в Hadoop.

При создании Spark-приложения программист указывает число исполнителей, требования к памяти и к ядрам каждого исполнителя и драйвера.

11.3.9 Настройки ОЗУ

Рассмотрим какими параметрами, регулирующими использование памяти, обладает исполнитель. Память исполнителя делится на память структур данных Spark (Spark Memory) и память, используемую пользовательскими функциями обработки RDD (User Memory). Также, резервируется 300 МБ

Таблица 32: Параметры spark-submit / pyspark

Параметр	Описание
<code>-master</code>	Вариант запуска (или адрес диспетчера кластера)
<code>-deploy-mode</code>	Где запускать драйвер: на машине, где запущен spark-submit (client, по умолчанию) или на узле кластера (cluster)
<code>-class</code>	Основной класс (для Java-приложений)
<code>-name</code>	Дружественное имя приложения
<code>-jars</code>	Зависимости Java-приложений
<code>-files</code>	Файлы для копирования на узлы
<code>-py-files</code>	Библиотеки и модули Python для копирования на узлы и включения в PYTHONPATH
<code>-conf</code>	Произвольная настройка в формате имя=значение, например, <code>spark.ui.port=36000</code>
<code>-properties-file</code>	Конфигурационный файл. По умолчанию ищется <code>conf/spark-defaults.conf</code>
<code>-executor-memory</code>	Объем памяти каждого исполнителя (По умолчанию 1GB)
<code>-driver-memory</code>	Объем памяти драйвера (По умолчанию 1024MB)
<code>-executor-cores</code>	Число ядер исполнителя (По умолчанию 1)
<code>-driver-cores</code>	Число ядер драйвера (По умолчанию 1)
<code>-num-executors</code>	Число исполнителей (По умолчанию 2)

памяти, чтобы минимизировать исключения `OutOfMemory`. Spark Memory делится на Storage Memory — память, которая используется для кэширования RDD и хранения широковещательных переменных, и Execution Memory, содержащую, в частности, буферы для shuffle. Между ними нет строгой границы, однако свойство `spark.memory.storageFraction` позволяет определить долю Storage Memory, которая гарантированно будет занята только для нужд Storage и не будет использована для Execution.

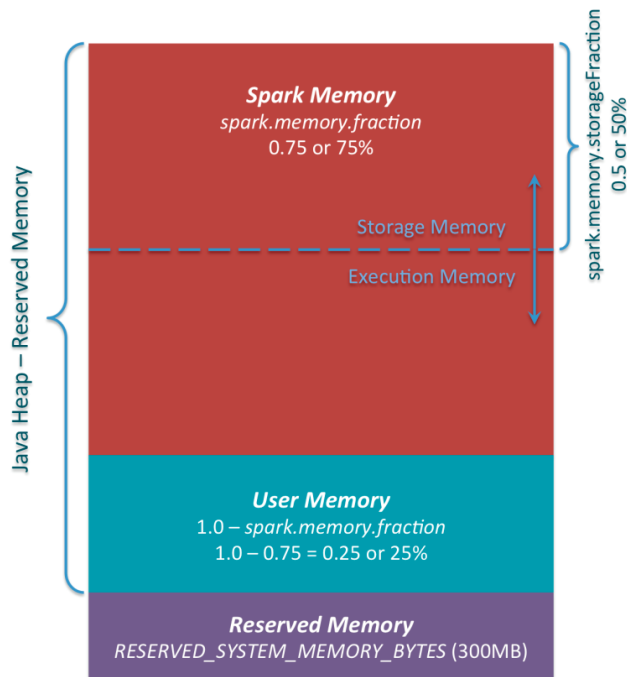


Рис. 187: Структура оперативной памяти: Spark Memory, User Memory, Reserved Memory ⁴⁹

⁴⁹<https://medium.com/@muditnagar/spark-memory-management-v1-6-0-8c9178970652>

11.3.10 Dynamic Resource Allocation

При старте Spark-приложения, например, в виде интерактивного Jupyter-notebook, необходимо определить объем ресурсов: число исполнителей, количество памяти и ядер драйвера и каждого исполнителя. Эти ресурсы будут получены от планировщика кластера. Но интерактивное Spark-приложение работает только при обращении к действию, все остальное время оно простаивает. Тем не менее ресурсы приложения остаются зарезервированными и не могут быть использованы другими приложениями.

В такой ситуации очень полезен **Dynamic Resource Allocation**, реализованный в последних версиях Spark. Это механизм, действующий по следующим правилам: приложение запрашивает у планировщика ресурсы на дополнительных исполнителей, если в течение некоторого промежутка времени очередь задач приложения содержала ожидающие задачи. Если же исполнитель не получал новых задач от приложения в течение некоторого промежутка времени, приложение удаляет исполнителя и возвращает ресурсы в кластер.

Детализация **запроса ресурсов**:

- Приложение запрашивает одного дополнительного исполнителя, если в течение `spark.dynamicAllocation.scheduler BacklogTimeout` секунд в очереди задач приложения находились ожидающие задачи;
- После чего каждые `spark.dynamicAllocation.sustained SchedulerBacklogTimeout` секунд в случае непустой очереди ожидающих задач приложение запрашивает 2-х дополнительных исполнителей.

Детализация **возврата ресурсов**:

- Если исполнитель не получал задач от приложения в течение `spark.dynamicAllocation.executorIdleTimeout` секунд, приложение уничтожает исполнителя;
- Исполнители, использующие кэш RDD, обычно не уничтожаются.

11.3.11 External Shuffle Service (ESS)

Во время выполнения приложения Spark возможна ситуация, когда исполнитель аварийно завершается, и вместе с ним теряются выходные файлы задач `map`, а также кэшированные данные RDD. Чтобы минимизировать подобные потери, Spark по возможности избегает преждевременного удаления исполнителей, которые используют память для кэширования.

Однако для более надежного управления промежуточными результатами `map`-задач может использоваться специальная компонента — **служба внешней тасовки (External Shuffle Service, ESS)**. При использовании ESS данные, генерируемые задачами `map`, сохраняются не в локальную файловую систему исполнителя, а передаются в ESS — фоновый сервис, запущенный на узлах кластера.

Задачи `reduce` запрашивают необходимые данные напрямую из ESS, что позволяет избежать повторного выполнения `map`-задач в случае сбоя исполнителя.

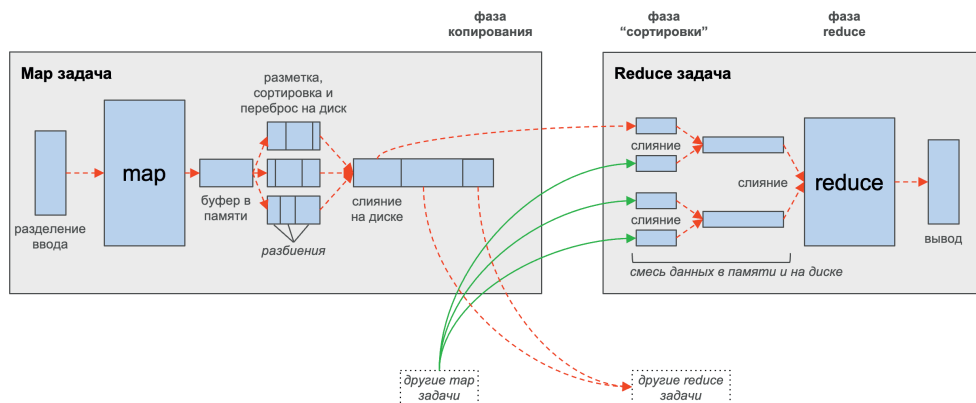


Рис. 188: Служба External Shuffle Service [1]

Ключевые характеристики ESS:

- ESS представляет собой фоновый процесс, запущенный на каждом узле кластера;

- Задачи **map** записывают свои выходные данные в ESS;
- Задачи **reduce** считывают выходные данные **map** из ESS;
- В новых версиях Spark ESS может также использоваться для хранения кэшированных RDD.

11.3.12 Особенности PySpark

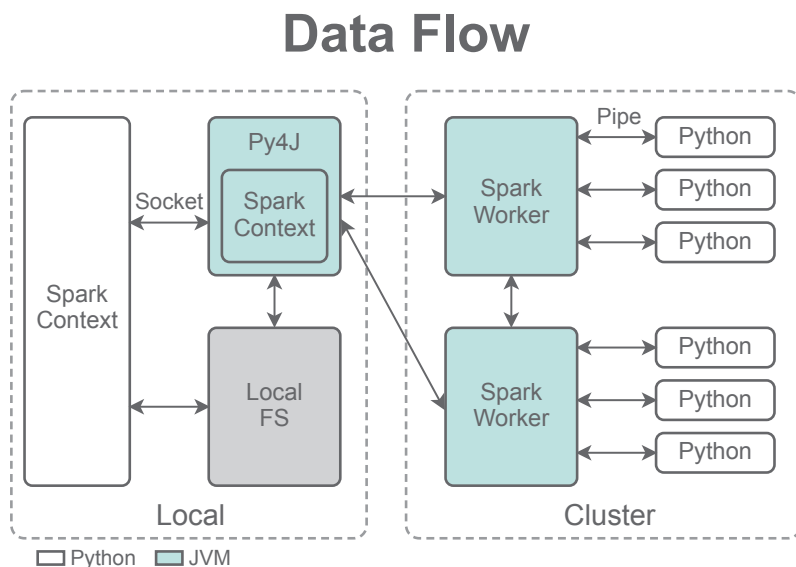


Рис. 189: Пример потока данных для локальных и кластерных вычислений

Коснемся особенностей PySpark. PySpark подразумевает взаимодействие исполнителя с интерпретатором Python. Это обуславливает значимые издержки на сериализацию и десериализацию данных. PySpark старается оптимизировать свою работу: выполнять максимальное число трансформаций внутри Python-интерпретатора, не передавая данные между исполнителем и Python. Другая оптимизация — повторное использование Python-

интерпретатора при выполнении очередной задачи исполнителем ^{50 51}.

Недостатки PySpark для RDD:

- Издержки на (де)сериализацию между JVM и Python;
- multiprocessing-вариант с Python требует дополнительной памяти на broadcast variables по сравнению с multithread JVM-исполнителями.

Оптимизации в PySpark для RDD:

- Исполнение максимума трансформаций внутри интерпретатора Python;
- Повторное использование интерпретатора Python при выполнении очередной задачи исполнителем;
- Использование PyPy (JIT-трансляция в машинный код).

11.4 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

- В чем состоят недостатки Hadoop?
- Что такое RDD, секция, трансформация, действие, стадия, задача?
- В каких ситуациях следует кэшировать RDD?
- Что такое совместно секционированные и совместно расположенные RDD и как их использовать для оптимизации соединения?
- Как можно задать параметры использования памяти для Spark-приложения?
- Что такое Dynamic Resource Allocation и External Shuffle Service?

⁵⁰<https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals>

⁵¹<https://www.slideshare.net/mateuszbuskiewicz/dive-into-pyspark>

ГЛАВА 12

Apache Spark SQL

Данная глава посвящена технологии Spark SQL — API для работы с распределенными датафреймами, позволяющему использовать SQL для обработки больших данных. В данной главе мы рассмотрим основные компоненты Spark SQL, такие как DataFrame, SQL, SparkSession и Catalog, а также оптимизацию запросов с помощью Catalyst Optimizer и Project Tungsten.

12.1 Основные компоненты Spark SQL

Spark SQL — это API Spark, позволяющий работать с **распределенными датафреймами** вместо RDD. Распределенный датафрейм представляет собой таблицу со схемой (именованными и типизированными столбцами). Методы работы с датафреймами аналогичны методам библиотеки Pandas языка Python (см. Рисунок 190). Распределенный датафрейм наследует от RDD основные механизмы:

- Секционирование данных;
- Трансформации и действия;
- Немодифицируемость;
- Ленивые вычисления;
- Кэширование DataFrame в ОЗУ и на диске.

Рассмотрим основные преимущества датафреймов по сравнению с RDD:

⁵²<https://izhangzhihao.github.io/2018/04/01/Spark-The-Definitive-Guide-In-Short/>

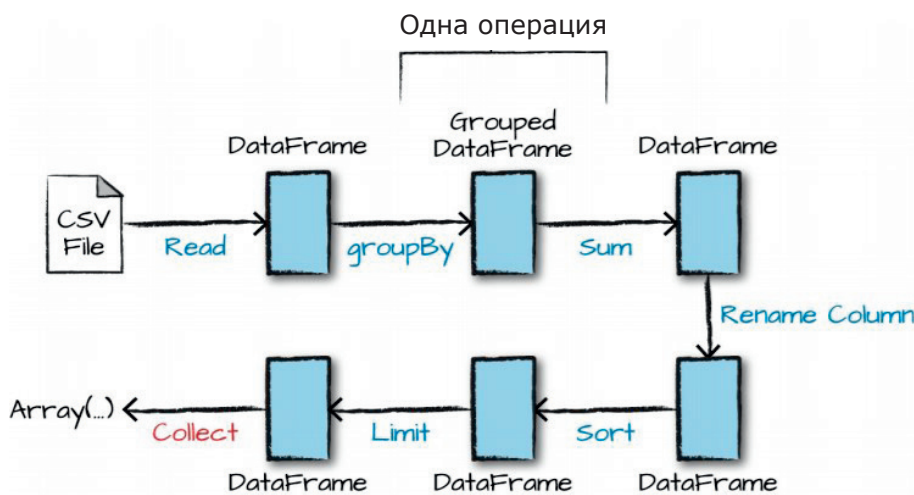


Рис. 190: Операции над датафреймом ⁵²

- Поддержка широкого спектра эффективных форматов хранения и чтения данных, включая как строковые форматы (CSV, JSON, JDBC, Hive, Avro), так и колоночные (Parquet, ORC);
- Возможность использования сложных типов данных в столбцах: структур, массивов, отображений;
- Наличие встроенного SQL-интерпретатора, позволяющего работать с распределенными таблицами с помощью запросов на языке SQL. Такой подход часто рассматривается как распределенная альтернатива библиотеке Pandas;
- Возможность использования пользовательских функций (User Defined Functions, UDF), расширяющих стандартный функционал обработки данных;
- Применение механизмов оптимизации запросов, включая стоимостную оптимизацию (cost-based optimization);

- Повышенная производительность за счет использования механизма генерации исполняемого кода (см. раздел 12.2) вместо интерпретации дерева операций, а также применения компактных представлений данных в управляемой вручную (unmanaged) памяти JVM (см. раздел 12.3).

Для приложений на Python (PySpark) использование датафреймов обеспечивает значительно лучшую производительность по сравнению с RDD. Операции над датафреймами реализованы как обертки над методами на Scala и не требуют дополнительных расходов на сериализацию и десериализацию данных (исключение составляют вызовы UDF).

Распределенный датафрейм Spark представляет собой таблицу со строго заданной схемой. На практике датафреймы часто создаются путем чтения структурированных файлов из HDFS. Пример ниже демонстрирует создание новой сессии Spark в PySpark и чтение CSV-файла с автоматическим определением типов столбцов за счет параметра `inferSchema=True`.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
app = spark.read.csv("kaggle_application_train.csv",
                    header=True, inferSchema=True)

>>> app.printSchema()
root
|-- SK_ID_CURR: integer (nullable = true)
|-- TARGET: integer (nullable = true)
|-- NAME_CONTRACT_TYPE: string (nullable = true)
|-- CODE_GENDER: string (nullable = true)
|-- FLAG_OWN_CAR: string (nullable = true)
|-- FLAG_OWN_REALTY: string (nullable = true)
|-- CNT_CHILDREN: integer (nullable = true)
|-- AMT_INCOME_TOTAL: double (nullable = true)
```

```

|-- AMT_CREDIT: double (nullable = true)
|-- AMT_ANNUITY: double (nullable = true)
|-- AMT_GOODS_PRICE: double (nullable = true)
|-- NAME_TYPE_SUITE: string (nullable = true)
|-- NAME_INCOME_TYPE: string (nullable = true)
|-- NAME_EDUCATION_TYPE: string (nullable = true)
|-- NAME_FAMILY_STATUS: string (nullable = true)
|-- NAME_HOUSING_TYPE: string (nullable = true)

```

Синтаксис, позволяющий работать с распределенными датафреймами, очень похож на Pandas. В пакете `pyspark.sql.functions` находится множество готовых полезных функций для обработки датафреймов. Например, для построения таблицы клиентов с информацией о поле, количестве детей и возрасте можно использовать следующий код:

```

import pyspark.sql.functions as F
app.select(app['CODE_GENDER'], app['CNT_CHILDREN'],
            (-F.round(app['DAYS_BIRTH']/365,
0)).alias('age')).show()

```

```

+-----+-----+-----+
| CODE_GENDER | CNT_CHILDREN | age |
+-----+-----+-----+
|           M |           0 | 26.0 |
|           F |           0 | 46.0 |
|           M |           0 | 52.0 |
|           F |           0 | 52.0 |
+-----+-----+-----+

```

```
>>> app[app['CNT_CHILDREN'] > 5].count()
```

```
42
```

```
>>> app.groupby('CODE_GENDER').mean('CNT_CHILDREN').show()
```

```

+-----+-----+
| CODE_GENDER | avg(CNT_CHILDREN) |
+-----+-----+
|           F |    0.3923970599857 |
|           M |    0.4645484917998 |
+-----+-----+

```

Spark SQL позволяет аналитику (без требования навыков программирования) обрабатывать большие данные, используя SQL. Датафрейм может быть зарегистрирован как временное представление с помощью следующей команды:

```
app.createOrReplaceTempView('app')
```

С зарегистрированными представлениями можно работать с использованием полноценного SQL (включая соединения, оконные функции и т.д.).

```
spark.sql("SELECT CODE_GENDER, CNT_CHILDREN, -
          ROUND(DAYS_BIRTH/365, 0) AS age
          FROM app WHERE NAME_EDUCATION_TYPE=
          'Incomplete higher'")
```

Во время работы с объектами RDD программист в первую очередь создает экземпляр класса `SparkContext`. Этот объект служит основной точкой входа к функциональности Apache Spark. Через `SparkContext` осуществляется подключение к Spark-кластеру, создаются RDD и размещаются переменные на вычислительных узлах.

При работе с датафреймами применяется иной подход — используется объект класса `SparkSession`. `SparkSession` объединяет в себе функциональность `SparkContext`, `SQLContext` и `HiveContext` и представляет собой единый интерфейс для работы с данными в Spark. Кроме того, `SparkSession` управляет каталогом временных представлений (views) и зарегистрированными пользовательскими функциями (User Defined Functions, UDF). Временные

представления создаются на время существования сессии Spark. Они доступны только в рамках текущей сессии и называются TempView. Для создания представлений, доступных между разными сессиями одного и того же приложения, используется механизм GlobalTempView.

Каталог Spark (Spark Catalog) предоставляет интерфейс для доступа к метаданным, включая информацию о таблицах, представлениях и функциях. Spark поддерживает как временный каталог, связанный с текущей `SparkSession`, так и постоянный каталог Hive. Таким образом, Spark SQL частично реализует функциональность Hive, используя в качестве вычислительного движка распределенные датафреймы вместо MapReduce.

Важно отметить, что датафрейм в Spark не обязательно представляет собой плоскую таблицу. Он может быть основан на иерархически структурированных форматах данных, таких как JSON (поддерживается нативно) или XML (обрабатывается через внешние библиотеки). Иерархия в структуре датафрейма реализуется с помощью сложных типов данных — структур, массивов и отображений. Для их обработки в Spark SQL предусмотрено множество функций, доступных в модуле `pyspark.sql.functions`, а в SQL-выражениях может использоваться конструкция `LATERAL VIEW`.

```
prize = spark.read.json("kaggle_prize.json")
```

```
>>> prize.printSchema()
```

```
root
```

```
|-- category: string (nullable = true)
|-- laureates: array (nullable = true)
|   |-- element: struct (containsNull = true)
|       |-- firstname: string (nullable = true)
|       |-- id: string (nullable = true)
|       |-- motivation: string (nullable = true)
|       |-- share: string (nullable = true)
|       |-- surname: string (nullable = true)
```

```
-- overallMotivation: string (nullable = true)
-- year: string (nullable = true)

/* SQL */
SELECT p.*, l.* FROM prize p
LATERAL VIEW EXPLODE(laureates) AS l
```

12.2 Catalyst Optimizer

Важным механизмом Spark SQL является **оптимизатор запросов Catalyst** (см. Рисунок 191). Цепочку трансформаций датафреймов и SQL-запросы Spark преобразует в граф вычислений над RDD. Оптимизатор запросов ответственен за то, чтобы создать оптимальный граф вычислений над RDD.

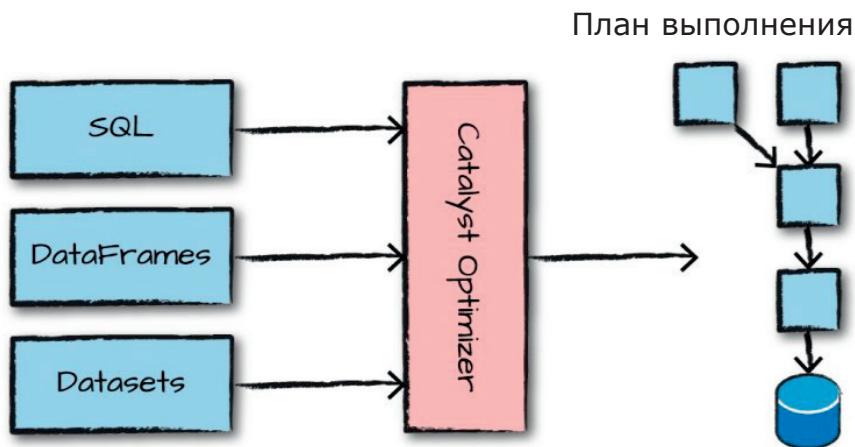
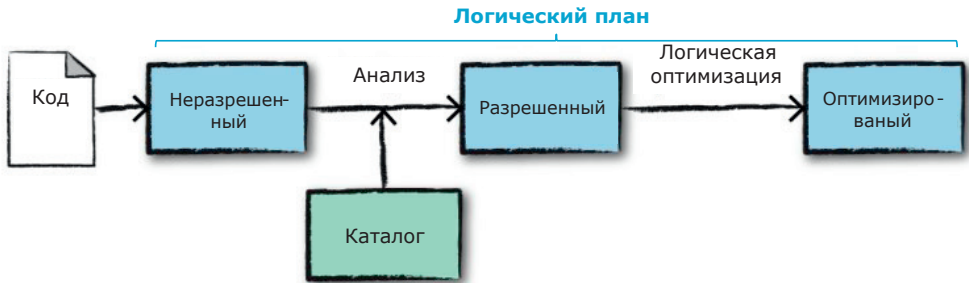


Рис. 191: Общая схема работы Catalyst Optimizer ⁵³

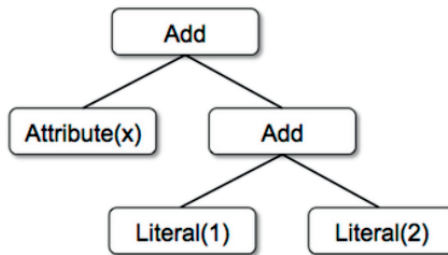
12.2.1 Оптимизации проекции и фильтрации

Оптимизация состоит из двух этапов: **логическая оптимизация** и **физическая оптимизация**. Логическая оптимизация состоит из нескольких

Рис. 192: Схема работы логической оптимизации ⁵³

шагов (см. Рисунок 192). На первом шаге создается **unresolved logical plan** — абстрактное синтаксическое дерево, соответствующее оригинальному запросу. Предположим, есть 3 класса, обозначающих узлы синтаксического дерева: **Literal**, **Attribute** и **Add**. Тогда для выражения $x + (1 + 2)$ можно составить запрос и его синтаксическое дерево (см. Рисунок 193).

```
Add(Attribute(x), Add(Literal(1), Literal(2)))
```

Рис. 193: Пример синтаксического дерева ⁵³

Далее, дерево преобразуется в **resolved logical plan** — на этот момент оптимизатор получает схемы представлений из каталога. Далее происходит

⁵³<https://www.databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html>

непосредственно логическая оптимизация, результатом которой является оптимизированный логический план исполнения запроса.

При создании resolved logical plan могут быть обнаружены такие ошибки, как использование в запросе отсутствующего представления или имени столбца, не существующего в представлениях запроса, а также несоответствие типов в выражениях.

Непосредственно в процессе логической оптимизации происходит применение **эвристических правил оптимизации**, принимающих во внимание только “текст” запроса в привязке к схемам таблиц, например: перемещение условий ближе к чтению источника данных, удаление неиспользуемых столбцов из проекций в подзапросах, предвычисление константных выражений, вычисление повторяющихся выражений один раз, оптимизация логических выражений.

12.2.2 Оптимизации Join

Физическая оптимизация запроса связана с выбором типа реализации операции соединения. Известны три основных разновидности соединения:

- **Sort-Merge Join.** Сортировка отношений по ключу в обеих таблицах и соединение кортежей в процессе сортировки слиянием. Сложность: $O(M \log M + N \log N)$;
- **Hash Join.** Создание хеш-таблицы на основе строк одной из таблиц (build side). Поиск ключей другой таблицы в хеш-таблице (probe side). Сложность: $O(M + N)$;
- **Nested-Loops Join.** Соединение путем прохода по строкам обеих таблиц во вложенном цикле. Этот способ подходит для соединения по произвольному условию (не только EQUI JOIN). Сложность: $O(MN)$.

При выполнении операций соединения Spark может использовать различные алгоритмы, выбор которых зависит от размеров и свойств соединяемых

датафреймов, а также от доступных ресурсов. Наиболее распространенными типами соединений являются следующие:

- **Broadcast Hash Join** — оптимальный вариант, когда один из датафреймов достаточно мал, чтобы быть полностью размещенным в памяти каждого вычислительного узла. На его основе строится хеш-таблица, которая передается (т.е. “широковещательно” рассылается) на все узлы кластера. Второй, потенциально большой датафрейм, обрабатывается распределенно: его строки проверяются на наличие соответствий в локальной хеш-таблице;
- **Broadcast Nested Loop Join** используется в случаях, когда невозможно применить хеш-соединение, например, при отсутствии условия равенства. Один из датафреймов (обычно меньший) копируется на все узлы кластера. Далее для каждой секции второго датафрейма выполняется вложенный цикл, в рамках которого проверяются условия соединения строк из обоих датафреймов. Этот метод может быть менее эффективным, особенно при больших объемах данных;
- **ShuffleHashJoin** — классический JOIN, реализованный в MapReduce. Записи обеих таблиц помечаются тегами, выполняется тасовка (shuffle) по ключу соединения. На стадии reduce строки с одинаковыми ключами поступают на обработку, и применяется соответствующая логика соединения (например, inner, left). ShuffleHashJoin подходит только для EQUI JOIN;
- **SortMergeJoin** предполагает предварительное секционирование обоих датафреймов по одинаковой схеме, а затем сортировку каждой секции по ключу соединения. После этого осуществляется параллельный проход по отсортированным секциям, в ходе которого строки с совпадающими ключами объединяются. Применяется только для EQUI JOIN и особенно эффективен при работе с крупными предварительно отсортированными наборами данных.

Таблица 33: Параметры Join

Опция	По умолчанию	Использование
spark.sql.autoBroadcastJoinThreshold	10MB	Максимальный размер таблицы в байтах для автоматического broadcast
spark.sql.join.preferSortMergeJoin	true	Использовать SortMergeJoin по умолчанию, а не ShuffleHashJoin

Выбор между этими и другими стратегиями соединения осуществляется на этапе планирования запроса с учетом статистики и конфигурации системы.

```
# python
import pyspark.sql.functions as F
a.join(F.broadcast(b), on='id')
a.join(b.hint('broadcast'), on='id')

/* SQL */
SELECT /** MAPJOIN(b) */ ...
SELECT /** BROADCASTJOIN(b) */ ...
SELECT /** BROADCAST(b) */ ...
```

Например, Spark автоматически использует BroadcastJoin, если размер одной из таблиц небольшой. При этом BroadcastHashJoin используется для EQUI JOIN, BroadcastNestedLoopJoin — для произвольных условий соединения. Функция `broadcast` и соответствующие ей директивы SQL позволяют программисту дать Spark'у явное указание использовать BroadcastJoin.

Оптимизатор запросов The Catalyst Optimizer может собирать и использовать статистику таблиц и заданных столбцов таблиц для оптимизации запросов.

Сбор статистик (число строк, min/max, число уникальных значений) выполняется SQL-командой `ANALYZE TABLE`:

```
ANALYZE TABLE <table name>  
COMPUTE STATISTICS [FOR COLUMNS <column1>, <columns2>, ...]
```

Отображение статистики по таблице:

```
DESCRIBE EXTENDED <table name>
```

Отображение использования статистик при оптимизации:

```
EXPLAIN COST SELECT ...
```

Использование статистики при оптимизации исполнения запроса называется **стоимостной оптимизацией** (CBO, **Cost-based optimizations**). В отличие от эвристических методов, стоимостная оптимизация позволяет принимать более эффективные решения: подбора типа JOIN, выбора build side для hash join, определения оптимального порядка при соединении нескольких таблиц.

Статистика, собранная по исходным таблицам, преобразуется через фильтры для получения приближенных статистик, полезных при оптимизации соединения.

12.3 Project Tungsten

Другой важной частью Spark SQL наряду с оптимизатором запросов является **Project Tungsten** (см. Рисунок 194). Данная библиотека является попыткой повысить быстродействие Spark SQL за счет устранения недостатков, присущих JVM-приложениям.

Project Tungsten предписывает использовать unmanaged-память JVM для хранения значений датафреймов. Это обеспечивает более компактное хране-

⁵⁴<https://developer.hpe.com/blog/tips-and-best-practices-to-take-advantage-of-spark-2x/>

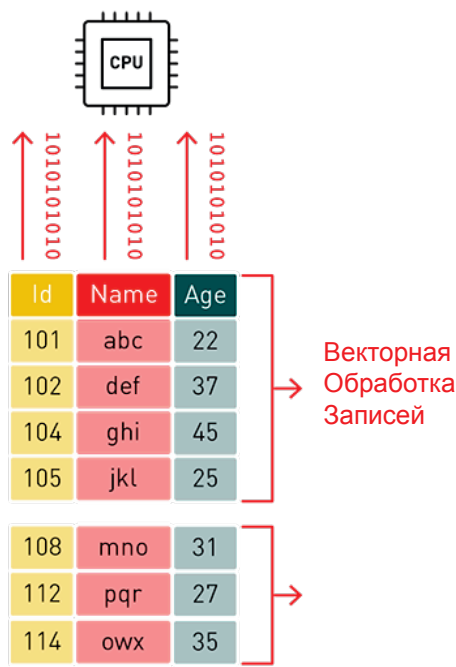


Рис. 194: Особенности Project Tungsten ⁵⁴

ние и минимизирует сборку мусора. В рамках Tungsten предлагается колончатый формат хранения (vectorized record batches) датафрейма в оперативной памяти. Проект представил возможность генерации кода на Java вместо интерпретации абстрактных синтаксических деревьев решений.

12.3.1 Колончатое хранение в памяти, Apache Arrow

Колончатый формат, предложенный в Tungsten, используется при кэшировании датафреймов в оперативной памяти. Колончатое хранение используется в рамках групп строк датафрейма фиксированной длины (группа называется **batch**). Данные каждого столбца внутри batch сжимаются по отдельности.

Данный формат был реализован в рамках проекта **Apache Arrow**, призванного решить проблемы множества форматов данных в современном ПО

Таблица 34: Параметры колончатого хранения в памяти

Опция	По умолчанию	Использование
spark.sql.inMemoryColumnarStorage.batchSize	10000	Число значений столбца в batch
spark.sql.inMemoryColumnarStorage.compressed	true	Автоматический выбор алгоритма сжатия, на основе статистики значений

обработки данных (см. Рисунок 195): высокие издержки на сериализацию и десериализацию, дублирование функционала, обусловленное использованием своего формата в каждом ПО.

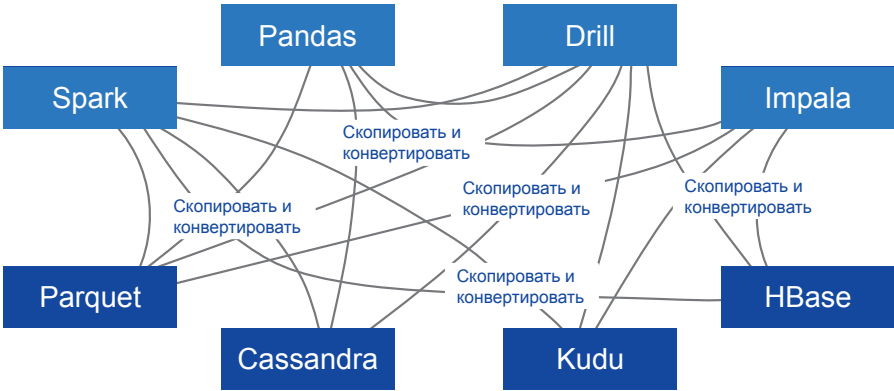
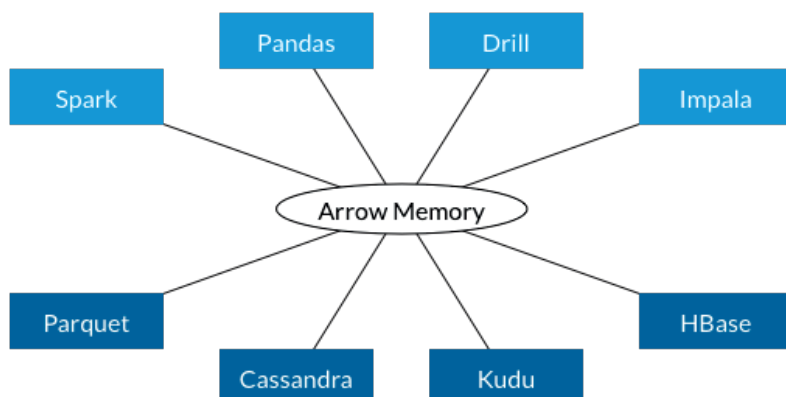


Рис. 195: Взаимодействие форматов данных между собой ⁵⁵

Apache Arrow призван свести к нулю трансформацию данных при взаимодействии приложений (см. Рисунок 196), а также дать толчок повторному использованию кода обработки данных в разных приложениях.

⁵⁵<https://www.datanami.com/2016/02/17/arrow-aims-to-defrag-big-in-memory-analytics/>

Рис. 196: Цель Arrow Memory ⁵⁵

12.3.2 User Defined Function (UDF)

Apache Arrow позволил значительно повысить производительность пользовательских функций в PySpark. Ранее пользовательская функция, написанная на Python, вызывалась для каждой строки датафрейма. Что является неоптимальным как с точки зрения сериализации и десериализации данных при взаимодействии JVM и Python, так и с точки зрения самого Python — небыстрого языка программирования. Arrow позволил использовать векторные UDF, написанные в терминах Pandas series. Apache Arrow используется как в Pandas-датафреймах, так и при in-memory хранении распределенных Spark-датафреймов, поэтому издержки на коммуникацию Spark и Python при работе с UDF-функциями существенно снижаются.

```
from pyspark.sql.functions import pandas_udf, col
```

```
def days2years(days):  
    return (days/365).astype(int)
```

Таблица 35: Параметры использования Apache Arrow

Опция	По умолчанию	Использование
spark.sql.execution.arrow.enabled	false	Использование Arrow
spark.sql.execution.arrow.maxRecordsPerBatch	10000	Размер batch для вызова векторной UDF

```
days2years_func = pandas_udf(days2years, 'integer')
display(app[[days2years_func(col('DAYS_BIRTH'))]])
```

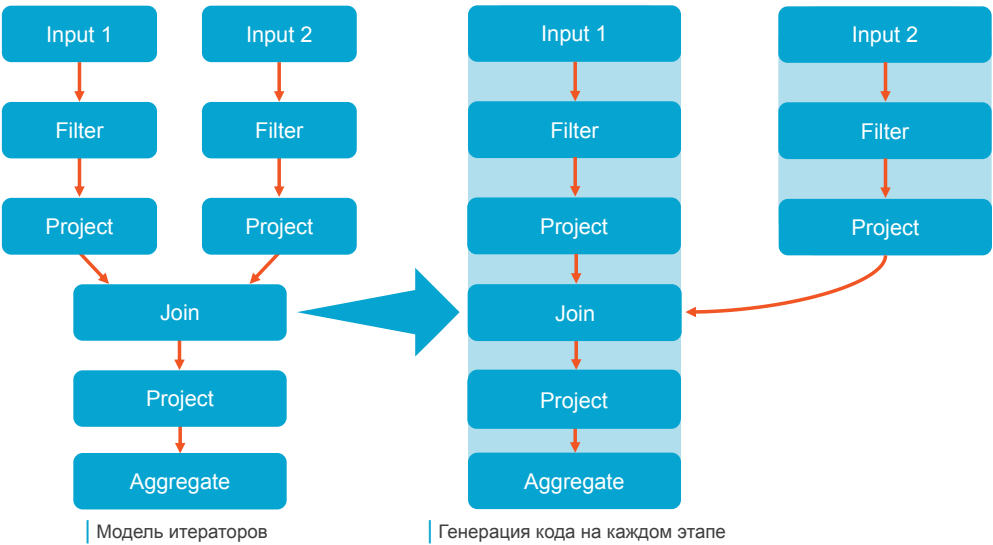


Рис. 197: Применение Whole-Stage Code generation

Whole-Stage code generation на Java позволяет избежать потерь в скорости при многочисленных вызовах виртуальных функций. Рассмотрим простой пример. На Рисунке 197 слева изображена последовательная модель выполнения, где в процессе выполнения будет 9 виртуальных вызовов функций и 8 промежуточных результатов. Справа показана оптимизированная модель

выполнения: на первой стадии производится чтение, фильтрация и проекция данных со второго входа, а на второй стадии — чтение, фильтрация и проекция данных с первого входа, объединение их с результатом первой стадии, проекция и агрегация итогового результата. Кроме этого, генератор создает код, который более эффективно исполняется внутри CPU (лучше используются кэш-память, конвейер и т.д.).

Просмотр сгенерированного для SQL-запроса кода:

```
EXPLAIN CODEGEN ...
```

12.4 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

- В чем преимущества DataFrame по сравнению с RDD?
- Какие стадии оптимизации реализованы в Catalyst Optimizer и в чем суть каждой из них?
- Какие типы соединения встречаются в физическом плане исполнения?
- Что такое Whole-Stage Code Generation?

ГЛАВА 13

Apache Hive. Форматы хранения таблиц в HDFS

Данная глава посвящена эффективным форматам хранения данных в распределенной файловой системе и их обработке с помощью технологии Apache Hive.

13.1 Форматы хранения таблиц

Естественными и важными требованиями к форматам хранения данных в распределенной файловой системе HDFS являются обеспечение высокой скорости записи данных, эффективное использование дискового пространства, а также поддержка быстрого и разнообразного выполнения запросов к хранящимся данным. Кроме того, желательно, чтобы формат позволял гибко адаптироваться к изменениям в структуре данных или к новым типам запросов, которые могут возникнуть в процессе эксплуатации системы.

На текущий момент нами были рассмотрены построчные текстовые форматы (например, CSV) и формат файла последовательности — SequenceFile.

13.1.1 Построчное и поколоночное хранение

Таблицы данных (см. Таблицу 36) можно хранить по строкам (как в csv-файле) или по столбцам. Рассмотренные текстовый формат и SequenceFile (кроме файла последовательности с блочным сжатием) хранят данные построчно (см. Таблицу 37). Построчное хранение является простым естественным форматом хранения данных, обеспечивающим высокую скорость загрузки

Таблица 36: Данные о работниках

Имя	Возраст	Зарплата
Елена	20	100
Вася	22	200
Коля	21	110
Мария	29	210
Юлия	27	300

ки данных в распределенную файловую систему. Недостатком является избыточное чтение (дополнительные затраты на избыточное движение головок жестких дисков при выполнении запросов по подмножеству столбцов). Также неэффективной является компрессия данных: степень сжатия смеси значений из разных столбцов ниже, чем степень компрессии каждого столбца по отдельности.

Таблица 37: Построчное хранение данных

Елена	20	100
Вася	22	200
Коля	21	110
Мария	29	210
Юлия	27	300

В **колончатом формате** (см. Таблицу 38) значения столбцов хранятся отдельно (в отдельных частях файла либо в разных файлах). Недостатком колончатого хранения является наличие издержек на формирование строк результатов запроса (record reconstruction), поскольку значения атрибутов строки находятся в разных частях файла.

Таблица 38: Поколоночное хранение данных

Елена	Вася	Коля	Мария	Юлия
20	22	21	29	27
100	200	110	210	300

13.1.2 RCFile

Частичным решением проблемы восстановления записей (record reconstruction) при колоночном хранении данных является организация хранения не по отдельным столбцам, а по группам столбцов, называемым **семействами столбцов (column families)**. Такие семейства формируются заранее программистом на основе анализа предполагаемых запросов к данным. Столбцы, которые часто используются совместно в запросах, целесообразно объединять в одно семейство. Однако при значительных изменениях логики обращений к данным структура семейств столбцов требует пересмотра и модификации. В целом фиксированная структура column families не способна эффективно поддерживать произвольные типы запросов.

Формат хранения данных **RCFile** (см. Рисунок 198) стал одним из первых прототипов колоночных форматов для хранения реляционных таблиц в распределенных аналитических системах. Он был разработан в компании Facebook и применен при создании системы Apache Hive. Основная идея RCFile — разбить файл на блоки с фиксированным числом строк, внутри которых данные хранятся по колонкам (см. Таблицу 39).

Размер группы можно варьировать для каждого отдельного файла. Каждый столбец в рамках группы архивируется отдельно. Это обеспечивает хороший коэффициент сжатия, а также возможность распаковывать значения столбцов лениво: тогда, когда значения столбца действительно нужны в соответствии с логикой выполнения запроса.

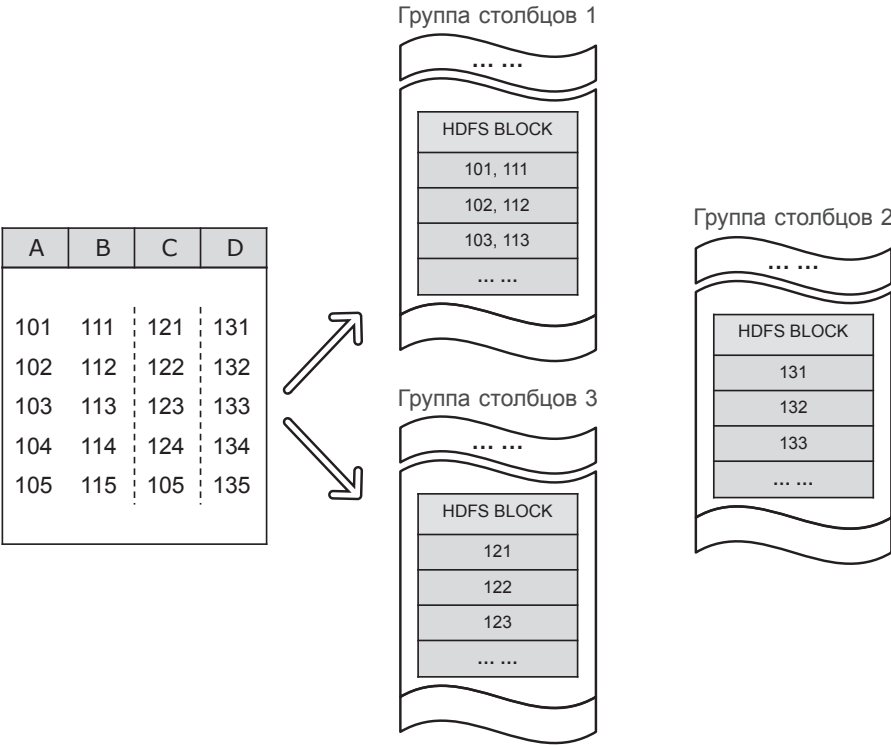


Рис. 198: Формат хранения RCFile [19]

Таблица 39: Хранение, соответствующее RCFile

Елена	Вася	Коля	20	22	21	100	200	110
	Мария	Юлия	29	27	210	300		

Ленивая распаковка

```
tbl(c1,c2,c3,c4)
```

```
SELECT c1 FROM tbl WHERE c4 = 1
```

Будут прочитаны и распакованы только **c1, c4**; **c1** будет прочитана и распакована только в тех группах, где **c4=1**.

Размер группы влияет на производительность исполнения запросов к данным и эффективность их хранения. При маленьком размере группы неэффективно сжатие данных, при большом — уменьшается эффект от ленивой распаковки.

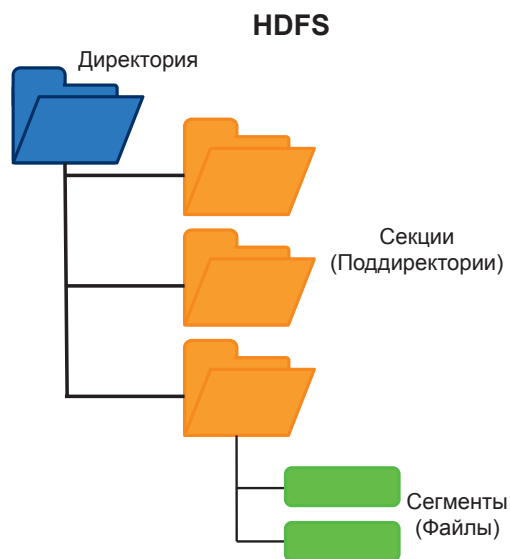
13.2 Apache Hive

Apache Hive был разработан Facebook в 2007 году. Hive не является OLTP-базой (Online Transaction Processing), а предназначен для выполнения аналитических запросов к большим объемам данных. Запросы исполняются путем трансляции их логики в MapReduce-обработку. Hive поддерживает множество форматов хранения таблиц, предоставляет возможность написания user defined-функций на Java, поддерживает индексы. Ограниченно поддерживаются операции **UPDATE** и **DELETE**. Языком написания запросов является **HiveQL (HQL)** — очень близкий к современным стандартам SQL. Интегрируется с HBase.

Во многих сценариях Hive предполагает работу с уже существующими данными в текстовых либо других форматах без их переформатирования и переноса. Метаданные Hive (схема таблиц и представлений) хранятся в реляционной СУБД.

Для эффективного исполнения запросов при создании таблицы Hive использует **секционирование (partitioning)** и **хеширование (clustering)**. Секционирование (см. Рисунок 199) разбивает файл данных по отдельным

⁵⁶<https://www.simplilearn.com/tutorials/hadoop-tutorial/data-file-partitioning>

Рис. 199: Секционирование ⁵⁶

директориям, каждая из которых соответствует своим диапазонам столбцов, по которым происходит секционирование. Секционирование полезно для фильтрующих запросов. Хеширование разбивает данные по значению хеш-функции от ключевых столбцов таблицы (первичного или внешнего ключа) на корзины. Хеширование позволяет значительно ускорить выполнение EQUI JOIN — соединения по равенству ключей.

```
CREATE TABLE students (  
    name string,  
    city string,  
    student_id int )  
PARTITIONED BY (year STRING, month STRING)  
CLUSTERED BY (student_id) INTO 256 BUCKETS
```

Для эффективной работы размер секции и корзины должен быть одинаковым на всех узлах (опция `SKEWED BY`). При этом не имеет смысла использовать разбиение на секции малого размера. Хеширование применяется для

столбцов с высокой кардинальностью (например, id для join). Hive допускает создание таблиц путем вставки данных из файлов, а также использование внешних таблиц — файлов, содержимое которых не переносится.

Контроль типов не производится при изначальной загрузке данных. Файл перемещается в `/user/hive/warehouse/<имя таблицы>` (быстрая операция). Опция `LOCAL` означает загрузку из локальной ФС.

```
LOAD DATA [LOCAL] INPATH "/home/sergun/kv1.txt"
OVERWRITE INTO TABLE students
```

Таким образом, данные не копируются в `/user/hive/warehouse/...`. Связь между таблицей и данными не жесткая: `DROP TABLE` не удаляет файл.

```
CREATE EXTERNAL TABLE students (
    name string,
    city string,
    student_id int )
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION "/user/sergun/students1.csv";
```

При вставке данных в секционированную таблицу используется специальный синтаксис. Например, без использования `partition`:

```
INSERT OVERWRITE TABLE students SELECT * FROM students_old;
```

Указание `partition` вручную:

```
INSERT OVERWRITE TABLE students
PARTITION(year='2018', month='01')
SELECT * FROM students_old WHERE year = '2018' AND month='01';
```

Dynamic partitioning (соответствующие столбцы должны быть последними в `SELECT`):

```
INSERT OVERWRITE TABLE students PARTITION(year, month)
SELECT name, city, student_id, year, month
FROM students_old WHERE year = '2018' AND month='01'
```

Hive поддерживает сложные типы данных:

- Массив `ARRAY<data_type>;`
- Отображение `MAP<primitive_type, data_type>;`
- Структура `STRUCT<col_name : data_type, ...>;`
- Объединение `UNIONTYPE<data_type, data_type, ...>.`

Дополнительно, Hive поддерживает конструкции SQL для работы со сложными типами данных: выражение `LATERAL VIEW` и функцию `explode` для разворачивания значений сложных типов данных в таблицы.

```
CREATE TABLE persons (
    lastname string,
    previous_lastnames ARRAY<int>,
    age int )

SELECT explode(previous_lastnames) as p_lm FROM persons
SELECT lastname, age, p_lm FROM persons
    LATERAL VIEW explode(previous_lastnames) as p_lm
```

Hive поддерживает три типа User Defined Functions: обычные, преобразующие скалярные аргументы в скалярное значение; Aggregated, сворачивающие отношение в скаляр, и Tabular, позволяющие создавать таблицы из скалярных аргументов. Примером Tabular-функции является `explode`.

UDF-функции пишутся пользователем самостоятельно (в том числе, на языке Java) и могут быть использованы в HQL:

- **User Defined Function (UDF)** имеет несколько скалярных аргументов, возвращает скаляр (`round`, `exp`, `lower` и т.д.) 1×1 ;
- **User Defined Aggregated Function (UDAF)**: аргумент — отношение, результат — скаляр (реализуемый интерфейс соответствует вычислению агрегаций по частям) (`min`, `max`, `count` и т.д.) $n \times 1$;
- **User Defined Tabular Function (UDTF)**: несколько скалярных аргументов, результат — отношение, позволяет генерировать таблицы (`explode(array)`, `explode(map)`, ...) $1 \times n$.

HiveQL поддерживает современные аналитические и оконные функции.

```
SELECT a, ROW_NUMBER() OVER (PARTITIONED BY a)
SELECT a, RANK() OVER (PARTITIONED BY a, ORDER BY b)
SELECT a, DENSE_RANK() OVER (PARTITIONED BY a, ORDER BY b)
```

А также HiveQL позволяет хранить данные таблиц в нескольких форматах. При создании таблицы в спецификации `CREATE TABLE` указывается формат хранения.

```
CREATE TABLE students (
    name string,
    city string,
    student_id int )
STORED AS TEXTFILE | SEQUENCEFILE |
    RCFILE | ORCFILE | PARQUET | AVRO
AS SELECT * FROM students_old
```

Наконец, в Hive доступна **индексация данных**. Структура индекса хранится в отдельных таблицах. Hive имеет встроенную поддержку compact и bitmap-индексов (см. Таблицы 40 и 41). Существует возможность реализовывать свои механизмы индексирования. Однако, встроенные возможности

Таблица 40: Compact-индекс. Каждому уникальному значению ставятся в соответствие номера блоков в файле

Пол	_bucketname	_offsets
женский	person/bucket-000	[0,5,6, ...]
мужской	person/bucket-000	[1,2,3,4,7, ...]

индексирования Hive ограничены: compact и bitmap-индексы подходят лишь для категориальных данных с небольшой кардинальностью.

Таблица 41: Bitmap-индекс. Каждому уникальному значению признака соответствует бит в зависимости от значения признака в строке

Пол	_bucketname	_offset (блок)	_bitmaps
женский	person/bucket-000	0	0110001000..
мужской	person/bucket-000	0	1001110111..

13.3 Фильтр Блума

Эффективным методом индексирования категориальных значений является **фильтр Блума** (см. Рисунок 200). Это вероятностная структура данных, не требовательная к памяти, позволяющая проверять значение на принадлежность множеству, по которому построен фильтр Блума. Фильтр Блума может допускать редкие ложноположительные ошибки при проверке принадлежности элемента множеству.

Пусть m — длина битового массива, k — число хеш-функций $h_1(x), \dots, h_k(x)$ длины m . Изначально все биты равны 0. При вставке элемента x биты в позициях $h_1(x), \dots, h_k(x)$ устанавливаются в 1. При проверке принадлежности x множеству возможны 2 ситуации. Если хотя бы один $h_i(x) == 0$,

элемент не принадлежит множеству. Если все $h_i(x) == 1$, элемент принадлежит множеству с вероятностью $(1 - e^{-kn/m})^k$, где n — число элементов во множестве, хеш-функции равномерны и независимы.

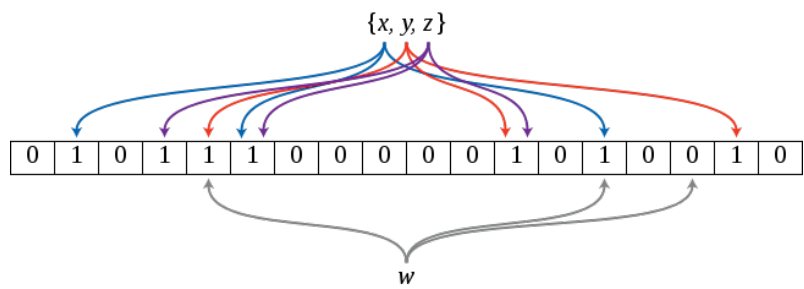


Рис. 200: Фильтр Блума ($m = 18, k = 3$)⁵⁷

13.4 Optimized Row Columnar File

Формат **Optimized Row Columnar file (ORC)** был предложен в Hive и является развитием RCFile. Непосредственно внутри этого формата хранения данных реализован фильтр Блума и min-max фильтр для непрерывных типов данных. Схема таблицы хранится внутри ORC-файла, поэтому ORC является независимым форматом и может быть использован за пределами Hive.

Благодаря формату ORC, Hive получил возможность обновления и удаления строк таблиц. При модификации данных обновления пишутся в отдельные так называемые **дельта-директории** (см. Таблицы 42 и 43).

Таблица 42: Исходные данные

Имя файла	Содержимое
00000_0	Bucket 0
00001_0	Bucket 1

⁵⁷https://en.wikipedia.org/wiki/Bloom_filter
⁵⁸<https://mavink.com/explore/Hive-Data-Types-Parquet>

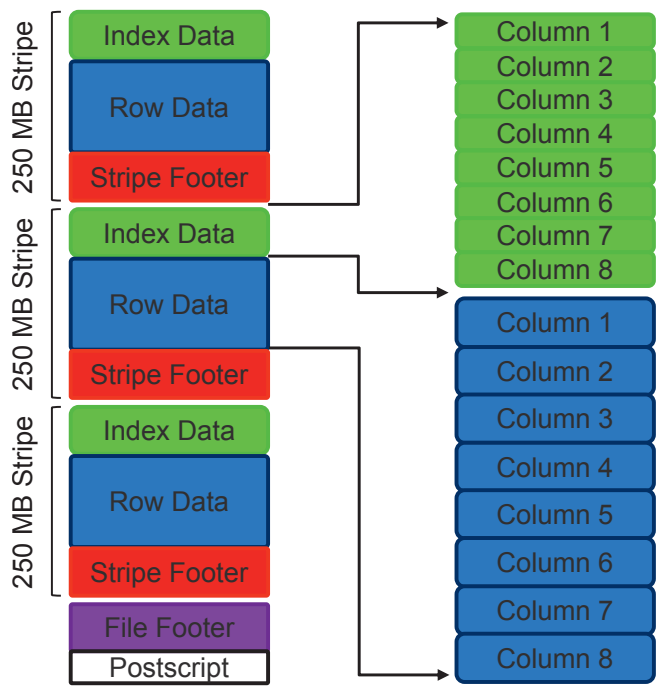


Рис. 201: Хранение в формате ORC ⁵⁸

Таблица 43: Дельта-директория с отдельной транзакцией

Имя файла	Содержимое
00000_0	Bucket 0 base
00001_0	Bucket 1 base
delta_0000005_0000005/ bucket_00000	Transaction 5 to 5, bucket 0 delta
delta_0000005_0000005/ bucket_00001	Transaction 5 to 5, bucket 1 delta

Периодически файлы, соответствующие отдельным транзакциям модификации данных, объединяются в общий дельта-файл. Такая операция называется **minor compaction**. С некоторой периодичностью дельта-файлы обновляют основные файлы с данными (см. Таблицы 44 и 45). Описанный процесс является фоновым. При этом отсутствуют строгие гарантии консистентности при одновременных модификации и чтении.

Таблица 44: Объединенные дельта-файлы (minor compaction)

Имя файла	Содержимое
00000_0	Bucket 0 base
00001_0	Bucket 1 base
delta_0000005_0000010/ bucket_00000	Transaction 5 to 10, bucket 0 delta
delta_0000005_0000010/ bucket_00001	Transaction 5 to 10, bucket 1 delta

Таблица 45: Новые версии “базовых” файлов (major compaction)

Имя файла	Содержимое
base_0000010/bucket_00000	Transaction upto 10, bucket 0 base
base_0000010/bucket_00001	Transaction upto 10, bucket 1 base

13.5 Дополнительные материалы

- Классическая реализация построочного хранения: Google for N-ary storage model (row-storage) [20];
- Поколоночное хранение: A decomposition storage model [21];

- Предшественник RCFile — PAX (Partition Attributes Across) [22];
- Оригинальная статья про RCFile [19];
- Индексы в Hive [23];
- Официальная Wiki Hive: Индексы ⁵⁹ и Bitmap ⁶⁰.

13.6 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

- Какие преимущества и недостатки колончатых форматов?
- Как устроен формат RCFile? Как размер группы влияет на производительность?
- Чем отличается секционирование от кластеризации в Hive?
- Что такое фильтр Блума?
- Какие новые возможности реализованы в ORC по сравнению с RCFile?

⁵⁹<https://cwiki.apache.org/confluence/display/Hive/IndexDev/>

⁶⁰<https://cwiki.apache.org/confluence/display/Hive/IndexDev+Bitmap/>

ГЛАВА 14

Apache Pig

В данной главе рассматривается платформа Apache Pig для обработки и анализа больших данных. В частности, описываются архитектура, язык Pig Latin, основные операторы и примеры использования.

14.1 Мотивация использования

Ранее для анализа больших наборов данных и написания распределенных программ использовалась парадигма MapReduce. Однако программирование в рамках этой модели представляло собой достаточно трудоемкий процесс: необходимо было реализовать две отдельные функции — `map` и `reduce`, что требовало значительного объема кода и глубокого понимания принципов MapReduce. Для упрощения процесса анализа данных были разработаны высокоуровневые инструменты, в частности, платформа **Pig**, созданная компанией Yahoo в 2006 году.

Pig — это платформа для обработки больших данных, включающая язык высокого уровня **Pig Latin** и вычислительную инфраструктуру, которая преобразует написанные сценарии в задачи MapReduce и запускает их на кластере Hadoop. Таким образом, программисту больше не требуется вручную реализовывать MapReduce-программы: вся необходимая логика выполняется автоматически.

Язык **Pig Latin** отличается лаконичностью и простотой синтаксиса. Многие распространенные операции, такие как агрегация, реализуются в одной-двух строках кода. В то время как эквивалентная реализация на MapReduce потребовала бы десятков строк. Дополнительно Pig позволяет пользователям

создавать собственные функции (User Defined Functions, UDF), написанные на Java, Python или JavaScript, и интегрировать их в сценарии на Pig Latin.

Платформа Pig широко используется в трех основных сценариях:

- Исследование и предварительная обработка данных;
- Построение аналитических моделей;
- Генерация отчетов.

Таким образом, Pig предоставляет удобный и эффективный способ работы с большими данными, абстрагируя разработчика от деталей реализации MapReduce-программ.

14.2 Функциональные возможности

Платформа Pig обладает широким набором инструментов для обработки данных:

- Поддержка различных типов данных (**Data Types**);
- Фильтрация записей по заданным условиям (**Filter**);
- Группировка данных по ключу (**Group By**);
- Объединение наборов данных по ключам (**Join Datasets**);
- Сортировка данных по одному или нескольким полям (**Sort Datasets**);
- Выполнение итеративных преобразований над элементами массивов (**Foreach**);
- Загрузка данных из файловой системы (**Load**);
- Разбиение данных на части по условиям (**Split**);

- Возможность подключения пользовательских функций (UDF), реализованных на Java, Python или JavaScript.

Все перечисленные операции существенно упрощают реализацию аналитических задач по сравнению с классическим подходом на MapReduce. Например, чтобы выполнить фильтрацию или группировку данных, достаточно указать соответствующую конструкцию на языке Pig Latin и задать необходимые параметры.

14.3 Операторы диагностики

В среде Pig пользователь может загружать данные из внешних источников с помощью команды **LOAD**, либо создавать данные вручную. Внутри Pig данные интерпретируются как **отношения** — таблицы или массивы, состоящие из строк и столбцов.

Каждое отношение характеризуется **схемой**, представляющей собой набор пар вида <название поля>, <тип данных>. Таким образом, можно рассматривать данные Pig как таблицу, в которой каждый столбец имеет имя и определенный тип значений.

После загрузки данных в Pig над ними можно выполнять различные преобразования и аналитические операции. Однако в процессе разработки и отладки программ может возникнуть необходимость проанализировать структуру или содержимое текущего отношения. Для этого в Pig предусмотрены специальные операторы диагностики.

К основным операторам диагностики относятся:

- **DESCRIBE** — отображает схему отношения, то есть имена столбцов и соответствующие им типы данных;
- **DUMP** — выводит содержимое массива на стандартный поток вывода;
- **EXPLAIN** — показывает внутренний план исполнения программы, включая цепочку задач MapReduce.

Оператор `DESCRIBE` полезен для понимания структуры обрабатываемых данных. Он позволяет убедиться, что схема отношения соответствует ожиданиям и корректно интерпретирована системой Pig.

Если требуется просмотреть фактическое содержимое данных, можно воспользоваться оператором `DUMP`. Он выведет на экран все записи, содержащиеся в указанном отношении. Однако при работе с большими объемами данных полная выгрузка может быть избыточной. В таких случаях удобно использовать оператор `LIMIT`, позволяющий ограничить количество выводимых строк. Например, следующая конструкция отобразит только первые три строки отношения `mytable`:

```
a = LIMIT mytable 3
DUMP a
```

Для анализа того, каким образом программа на языке Pig Latin была трансформирована в цепочку заданий MapReduce, используется оператор `EXPLAIN`. Он позволяет разработчику увидеть, как Pig оптимизировал выполнение задачи и какие стадии MapReduce будут вызваны при запуске программы.

14.4 Режимы запуска Pig

Рассмотрим несколько режимов запуска Pig — локальный интерактивный режим, интерактивный режим на кластере и запуск в виде скрипта.

Локальный интерактивный режим позволяет разработчику тестировать и отлаживать программы на небольших объемах данных, которые могут быть обработаны на одном компьютере. Для запуска Pig на единственной Java-машине необходимо прописать следующую команду:

```
$pig -x local
```

В таком случае Pig будет работать только с локальной файловой системой.

Интерактивный режим на кластере (MapReduce-интерактивный режим) используется для обработки больших наборов данных (напомним, что кластеры рекомендуется использовать, когда данных очень много, и на одной машине поместиться они не могут). Для запуска программы на кластере в интерактивном режиме необходимо использовать команду

```
$pig -x mapreduce
```

Наконец, **запуск в виде скрипта** позволяет разработчику писать и выполнять сценарии на языке Pig Latin, которые могут быть запущены как локально, так и на кластере. Обратим внимание, что в локальном режиме код для MapReduce не генерируется, а происходит его эмуляция.

14.5 Язык Pig Latin

Pig Latin основан на работе с **потоками данных** и не является декларативным языком программирования, в отличие от языка SQL. На вход программе на языке Pig Latin подаются некоторые **массивы** данных, которые подобны таблицам реляционной базы данных и в которых записи аналогичны строкам таблицы и состоят из полей.

Программа на данном языке представляет собой последовательность операторов. **Оператор** — это процедура, принимающая на вход массив, состоящий из набора записей, и формирующая на выходе другой массив. Компилятор Pig преобразует код Pig Latin в сценарии MapReduce.

14.5.1 Сравнение Pig Latin и SQL

Pig Latin является языком программирования потоков данных. На вход он принимает некоторые входные данные. Последовательность операторов представляет собой последовательные операции трансформации данных. SQL является декларативным языком программирования. Программы на SQL представляют собой набор операций, которые выполняются вместе.

Реляционная модель данных, которая используется в SQL, хранит данные в таблицах. Pig имеет собственное внутреннее представление данных (отличное от классических таблиц), которое может интерпретироваться пользователем аналогично представлению табличных данных.

Pig обладает следующими характеристиками:

- Наиболее частый формат ввода данных — текстовый файл с разделителями (табуляцией);
- Pig не предусматривает прямой возможности импорта данных в реляционные базы данных;
- Каждая строка, загруженная из источника, интерпретируется как **tuple (кортеж)** — структура, аналогичная строке в таблице реляционной базы данных;
- Множество кортежей объединяется в структуру **Bag**, которая по своей сути соответствует таблице в базе данных. В отличие от классических таблиц реляционных СУБД, Bag допускает, что разные кортежи могут содержать различное количество полей. Это предоставляет дополнительную гибкость при работе с неструктурированными или частично структурированными данными.

14.5.2 Основы Pig Latin

Pig Latin поддерживает большой набор скалярных типов данных, таких как: `int`, `long`, `chararray`, `bytearray`, `double`, `float`. Единицей данных является **поле**, а упорядоченный набор полей называется **tuple** или кортежем. Обычно кортеж заключается в круглые скобки. Bag — коллекция **tuples**, последовательность строк. Обычно **bag** заключается в фигурные скобки.

Под понятием отношения (**relation**) или упорядоченной таблицы подразумевается внешний **bag (outer bag)**. Графическая иллюстрация основ Pig Latin представлена на Рисунке 202.

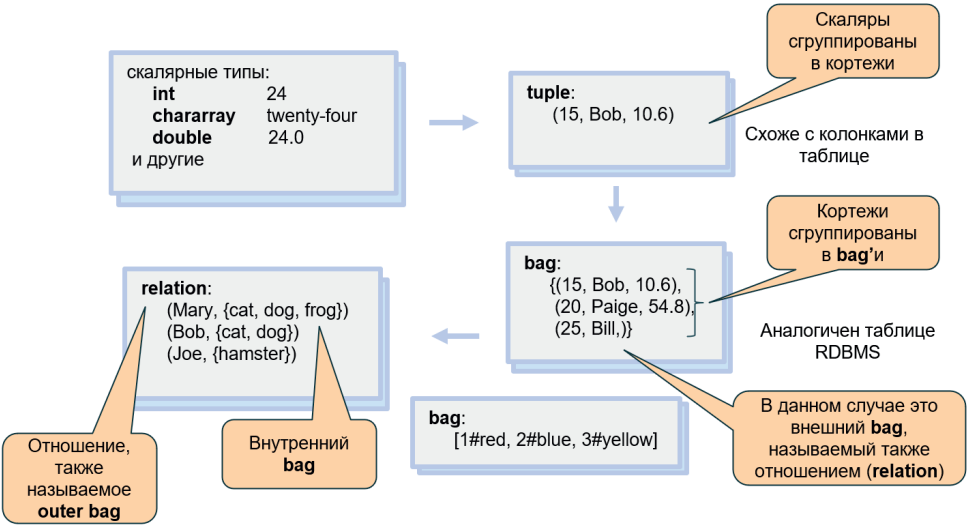


Рис. 202: Графическая иллюстрация основ Pig Latin

14.5.3 Базовые операции и функции

Базовые операции языка Pig Latin можно разделить на три категории: арифметические, логические и операции сравнения. Существующие операции и категории, к которым они относятся, отображены в Таблице 46.

Таблица 46: Основные операции в языке **Pig Latin**

Арифметические	Логические	Сравнения
<code>a + b</code> , <code>"a" + "b"</code>	<code>a and b</code>	<code>a == b</code>
<code>a - b</code> , <code>-a</code>	<code>a or b</code>	<code>a != b</code>
<code>a / b</code>	<code>a and not b</code> , <code>not (a and b)</code>	<code>a < b</code> <code>a <= b</code>
<code>a * b</code>		<code>a > b</code> <code>a >= b</code>
<code>%</code> : вычисление остатка		<code>is null</code>
<code>?</code> : оператор условия		<code>is not null</code>

В частности, в Pig Latin встроены следующие функции:

- **AVG** — вычисление среднего значения;
- **CONCAT** — конкатенация;
- **COUNT** — подсчет количества элементов;
- **DIFF** — вычисление разницы;
- **MAX** — нахождение максимального элемента;
- **MIN** — нахождение минимального элемента;
- **SIZE** — определение размера элемента;
- **SUM** — нахождение суммы;
- **TOKENIZE** — разбиение элемента на подэлементы;
- **IsEmpty** — проверка элемента на пустоту.

14.5.4 Парсинг

Процесс анализа (парсинга) программы, написанной на языке Pig Latin, включает в себя несколько ключевых этапов. На первом этапе осуществляется проверка типов используемых переменных и выражений. Далее выполняется проверка корректности ссылок на переменные и структуры данных. Заключительной стадией является генерация кода, соответствующего логике описанного алгоритма.

Отметим особенности парсинга программы на языке Pig Latin:

- Независимость платформы, на которой выполняется программа;
- В языке предусмотрены специальные команды: **DUMP** — для вывода результата вычислений на экран, и **STORE** — для записи результата во

внешнее хранилище. До тех пор, пока одна из этих команд не встретится в сценарии, фактическое выполнение вычислений не начинается, хотя проверка и анализ программы производятся полностью;

- Язык поддерживает механизмы оптимизации выполнения, позволяющие повышать производительность за счет трансформации и упрощения промежуточного представления программы.

14.5.5 Процедура загрузки данных в Pig

Для загрузки данных существуют 4 основные функции:

- `TextLoader()` — загрузка из текстового формата, каждая строка — tuple;
- `JsonLoader()` — загрузка из JSON-формата, каждая строка — tuple;
- `PigStorage()` — загрузка из текстового файла с разделителем;
- `JsonStorage()` — загрузка из JSON-файла.

Рассмотрим пример:

```
A = load "/datadir/datafile" using PigStorage('\t');
A = load "/datadir/datafile" using PigStorage(',')
    as (f1:int, f2:chararray, f3:float);
A = load "/datadir/" using PigStorage(',')
    as (f1:int, f2:chararray, f3:float);
```

При загрузке данных можно указать используемый тип разделителя. Например, в первой строке примера выше разделителем является символ табуляции. Во второй и последней строках разделителем является запятая.

При загрузке данных также можно указать тип колонок, в которые исходные данные будут считаны. Например, во второй строке данные будут

загружаться в 3 колонки. Первая будет иметь тип данных `int`, вторая имеет формат символьных строк, а последняя содержит вещественные числа. Форматом данных по умолчанию является `bytearray` (бинарная строка).

Стоит отметить, что операторы и команды в Pig не являются регистрозависимыми. Однако, функции и отношения регистрозависимы. Если у полей нет названия, то к полям можно обращаться через `$0`, `$1` и т.д. Нумерация начинается с нуля.

Рассмотрим пример другой программы на Pig Latin:

```
messages = LOAD "messages";  
warns = FILTER messages BY $0 MATCHES '.*WARN+.*';  
STORE warns INTO "warnings";
```

Данная программа является последовательностью операторов, и каждый оператор работает с отношениями. В первой строке происходит считывание тестового набора данных в массив, представляющий собой набор записей. Данный массив имеет название `messages`. Поскольку в программе не указан перечень колонок, все данные будут считаны в одну колонку. Отметим, что загрузка данных в Pig Latin может производиться как из локального файла, так и из лежащего в HDFS.

Во второй строке происходит фильтрация данного массива. Обозначение `$0` определяет обращение к первому столбцу. В массиве `messages` происходит фильтрация данных, и остаются только те данные, которые в первом столбце содержат в себе последовательность символов `WARN`. В частности, в примере используется регулярное выражение. Последовательность символов может встречаться как слева, так и справа. Итоговые данные записываются в массив `warns`. В третьей строке происходит сохранение данного массива (данного отношения) в новый файл с названием `warnings`. Он сохраняется в файловой системе хоста.

14.5.6 Функции и операторы Pig Latin

Рассмотрим функции и операторы языка Pig Latin:

- **TOKENIZE** — разбивает массив на **bag** из слов (разделители — пробелы, запятые и т.д.);
- **GENERATE** — определяет поля нового отношения (используется вместе с **FOREACH**);
- **FOREACH GENERATE** — итерация по каждому элементу в **bag** и его обработка;
- **FLATTEN** — построение прямолинейного набора **tuples** из сложной структуры;
- **GROUP BY** — создание нового **bag** типа $\{(a), \{(a), (a), (a)\}\}$.

14.5.7 Word Count

Рассмотрим пример простейшей программы **Word Count**, написанной на языке Pig Latin.

```
A = LOAD "./input.txt";
B = FOREACH A GENERATE FLATTEN(TOKENIZE((chararray)$0))
    AS word;
C = GROUP B BY word;
D = FOREACH C GENERATE group, COUNT(B);
STORE D INTO "./wordcount";
```

Данная программа состоит из 5 последовательных шагов. В каждом шаге на вход принимается некоторое отношение (массив данных). Результатом выполнения каждого шага также является некоторое отношение. В частности, в результате выполнения первой команды будет прочитан файл `./input.txt` (сохранится в отношении **A**). Допустим, данный файл состоит из двух строчек, в каждой из которых по два символа, перечисленных через запятую. Допустим, в первой строке были символы 'a' и 'b', во второй строке — 'c' и

'a'. Загрузка происходила без уточнения разделителя и наименования полей данных. В таком случае результат имеет вид:

$$A : \{(a, b), (c, a)\}.$$

При выполнении оператора В изначально выполнится **TOKENIZE** первой колонки. Поскольку используемые колонки не указаны, все данные будут определяться в первую колонку. Они разобьются на отдельные токены, и далее, с помощью **FLATTEN**, сгенерируется последовательность **tuples**. Результатом выполнения станет **bag**, в котором каждое из слов именуется как **word**:

$$B : \{\{(a), (b)\}, \{(c), (a)\}\} \rightarrow \{(a), (b), (c), (a)\}.$$

Третья строчка предназначена для группировки полученного результата. Используется группировка по **word** (названию данных кортежей), в результате которой получится группа, состоящая из двух букв 'a' с названием 'a', группа из одного 'b' с названием 'b', группа из одного 'c' с названием 'c':

$$C : \{(a, \{(a), (a)\}), (b, \{(b)\}), (c, \{(c)\})\}.$$

В четвертой команде происходит генерация нового отношения, которое будет состоять из двух полей: **group** (название группы) и счетчик группы (количество элементов):

$$D : \{(a, 2), (b, 1), (c, 1)\}.$$

В последней строке результат сохраняется в файл **./wordcount**.

Аналогичная программа на декларативном языке HiveQL по подсчету слов в документе имеет следующий вид:

```
CREATE TABLE docs (line STRING);
LOAD DATA INPATH "docs" OVERWRITE INTO TABLE docs;
CREATE TABLE word_counts AS
SELECT word, count (1) AS count FROM
    (SELECT explode (split(line, '\s')) AS word FROM docs)
GROUP BY word
ORDER BY word;
```

Изначально создается таблица **docs**, каждая строка **line** которой имеет строковый тип. В нее загружаются данные из файла **docs** с переписыванием

содержимого. Далее создается результирующая таблица `word_counts` с полями `word` и `count`. Для заполнения данной таблицы выполняется оператор `SELECT`. Каждая строка исходной таблицы разделяется по знаку `'\s'` на слова `word`. Благодаря операторам `ORDER BY` и `GROUP BY` данная последовательность слов упорядочивается по алфавиту и группируется по `word`. Значение `count` для каждого `word` считается как количество появлений `word` в каждой группе. Обратите внимание, что `ORDER BY` использует полную сортировку. Если требуется выполнить отдельную сортировку для каждого reducer'a, то необходимо воспользоваться командой `SORT BY`.

14.5.8 Запуск скриптов

Рассмотрим возможности запуска скриптов, написанных на языке Pig Latin. Пусть скрипт `myscript.pig` состоит из одной строчки:

```
a = load "$dir/myfile.txt" as PigStorage('\t');
```

В `myscript.pig` содержимое файла `myfile.txt` загружается в отношение `a` с разделителем (табуляцией). В данной строке используется параметр названия директории `dir`, в которой лежит данный файл. Параметр в скрипте обозначается с помощью знака `$` и названия данного параметра (`dir`). Имеются два подхода для работы с данным параметром:

- Заполнение данного параметра в командной строке при запуске скрипта:

```
> ./pig -param dir="/labfiles" myscript.pig
```

- Использование конфигурационного файла, в котором будет записываться перечень всех используемых параметров. Для данного примера будет записан единственный параметр `dir`. Содержимое файла `myparams.txt`:

```
dir="/labfiles"
```

В командной строке необходимо вызвать:

```
> ./pig -param_file myparams.txt myscript.pig
```

В представленных реализациях после запуска скрипта в командной строке будет заполнено значение параметра `dir`, и в отношении `A` загрузится `myfile.txt` из папки `/labfiles`:

```
a = load "/labfiles/myfile.txt" as PigStorage('\t');
```

14.6 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

- Для каких целей был разработан Pig?
- В каких режимах может запускаться Pig?
- Что такое Pig Latin?
- В чем отличия Pig Latin и SQL?
- Каков алгоритм парсинга в Pig Latin?
- Какие функции и операторы Pig Latin Вы знаете?

ГЛАВА 15

НBase, BigTable: основы и модель данных

В данной главе рассматриваются принципы и особенности работы колоночной базы данных НBase как распределенного хранилища над HDFS. Описываются модель данных, доступ по ключу, работа с ColumnFamilies, версияемость ячеек, а также ограничения и типичные сценарии применения. Приводятся отличия НBase от реляционных СУБД и рассматриваются основные способы запроса данных.

15.1 Введение в НBase и Bigtable

НBase представляет собой колоночную базу данных, разработанную в качестве надстройки над файловой системой HDFS. Она предназначена для хранения и обработки чрезвычайно больших объемов данных (от гигабайтов до петабайтов) и обеспечивает произвольный доступ к этим данным в режиме реального времени (как на чтение, так и на запись).

Модель доступа к данным в НBase имеет следующие ограничения:

- Доступ к строкам возможен только по одному ключу;
- Отсутствует поддержка транзакций по нескольким строкам;
- Поддерживаются исключительно операции, относящиеся к одной строке.

НBase — это распределенная, масштабируемая система (реализована на Java), ориентированная на работу с так называемыми “большими таблица-

ми”, которые могут содержать миллиарды строк и миллионы колонок, а также множество версий значений. В отличие от традиционных реляционных баз данных, не ограничивающих количество колонок на уровне проектирования, HBase изначально создавалась для поддержки такого рода сценариев. HBase позволяет выполнять CRUD операции над данными (создание, чтение, обновление и удаление) в режиме реального времени. Важно отметить, что HBase, как правило, развертывается на серверных вычислительных кластерах и не предназначена для использования на персональных компьютерах.

HBase относится к классу NoSQL баз данных (Not Only SQL). В частности, в HBase отсутствует традиционный SQL-интерфейс, не поддерживаются операции соединения (join), отсутствуют типизация колонок, а также реляционная модель с определенными связями между таблицами. Соединения в HBase могут быть реализованы вручную (с применением технологий MapReduce, Pig или Hive).

Появление HBase и других NoSQL-систем связано с ограничениями классических реляционных баз данных. С ростом объемов данных появилась необходимость не только в вертикальном масштабировании, но и горизонтальном, которое реляционные СУБД изначально не предусматривали. Попытки создания распределенных реляционных систем привели к появлению решений, требующих сложного ручного управления распределением и перераспределением данных между серверами, затрудняющих их эксплуатацию.

Дополнительной причиной для разработки HBase послужили ограничения HDFS: не поддерживается обновление существующих файлов, данные могут только записываться в конец. Кроме того, реляционные базы данных не приспособлены для хранения таблиц с десятками или сотнями тысяч колонок. Это послужило стимулом для создания системы, способной эффективно работать с такими структурами данных в Hadoop.

HBase основана на идеях базы данных **Google BigTable** [24]. Операции чтения и записи строго консистентны в рамках одной строки, то есть операция выполнится либо над всеми данными, либо не выполнится. Свойствами

Таблица 47: Сравнение между файловыми системами HDFS и GFS

	GFS	HDFS
Горизонтально масштабируемая БД	BigTable	HBase
Модель параллельного программирования	MapReduce	Hadoop
Система контроля серверов для распределенных систем	Chubby	ZooKeeper

HBase являются отказоустойчивость, линейная масштабируемость, а также автоматическое разделение (automatic sharding) таблицы на части (splits). HBase отличается прозрачным Java API (наиболее производительное решение для HBase), также существует HBase Shell, предназначенный для взаимодействия пользователя базы данных с HBase. Существует интеграция HBase с MapReduce, где HBase используется в качестве источника данных. В Таблице 47 приведено сравнение между распределенными файловыми системами HDFS и GFS.

Рассмотрим рамки прикладного использования HBase. Во-первых, HBase не является универсальным решением любой задачи, так как по сравнению с реляционными базами данных HBase имеет ограниченный API. HBase подходит для больших объемов данных, которые размещаются на большом количестве рабочих машин, а также при большом числе клиентов и запросов. Если данных мало, то их достаточно разместить на одном или нескольких узлах, что делает невозможным эффективное использование кластера (не получится эффективно выполнять параллельные операции). HBase подходит для выборки значений по заданному ключу, для свободной схемы данных (Schema-less). Схема данных является свободной, когда для разных строк количество колонок и типы данных в колонках могут быть разными (контроль делегируется пользователю). Также, HBase эффективен при работе со схемами с множеством разреженных колонок.

Рассмотрим ситуации, в которых использование HBase нецелесообразно. Прежде всего, HBase не рекомендуется применять в системах, ориентирован-

Таблица 48: Сравнение между реляционными БД и HBase

	Реляционная БД	HBase
Формат хранения данных	Построчный	Поклоночный
Транзакции	По многим строкам	По одной строке
Индексация	По многим рядам	По одному ряду
Размеры данных	Терабайты	Петабайты
Количество операций в секунду	Тысячи	Миллионы

ных на традиционную модель работы с данными, принятую в реляционных базах данных. Например, если приложение предполагает выполнение транзакций, то реализация соответствующего механизма в HBase ложится на плечи разработчика. Кроме того, HBase не поддерживает короткие SQL-запросы и операции, такие как `GROUP BY`, `JOIN`, `WHERE` и другие. Для выполнения подобных действий требуется писать сценарии на MapReduce или использовать вспомогательные инструменты (например, Pig или Hive). HBase также неэффективен, если кластер содержит менее пяти узлов. В Таблице 48 представлено сравнение реляционных СУБД и HBase.

Данные в HBase хранятся в таблицах (**tables**). Каждая таблица состоит из строк (**rows**), доступ к которым осуществляется по уникальному ключу. Ключ представляет собой произвольный байтовый массив, получаемый из бинарного представления любых данных: строк (**String**), чисел (**Long**) или пользовательских структур. При этом ключ уникален в рамках только одной таблицы. Все строки в таблице упорядочены по ключу в лексикографическом порядке, что обеспечивает эффективный доступ к данным.

Ключевой особенностью архитектуры HBase является использование объединений колонок (**ColumnFamilies**). Каждая строка может содержать одну или несколько таких групп, которые необходимо заранее определить при создании таблицы. Данные хранятся в ячейках (**cells**), а доступ к ячейке осу-

ществляется по номеру ряда, номеру объединения колонок и номеру колонки. Данные в HBase хранятся в виде байтового массива (тип отсутствует).

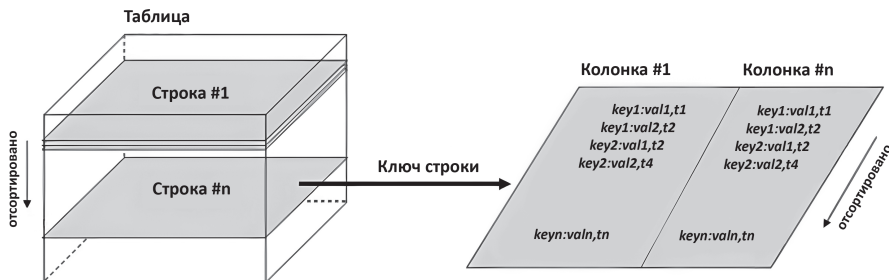


Рис. 203: Схематичное изображение структуры данных в HBase

Еще одной важной характеристикой HBase является поддержка версииности данных. Каждой ячейке может соответствовать несколько значений, отличающихся по времени записи. Такие версии сортируются по временной метке (**timestamp**), что позволяет хранить исторические состояния данных.

Физически строки в таблицах HBase делятся на **регионы (regions)**, что позволяет масштабировать хранение и обработку данных. На Рисунке 204 представлен пример таблицы HBase. Строки отсортированы по ключу (**Row Key**) и включают различные объединения столбцов (**Column Family One** и **Column Family Two**), которые состоят из отдельных колонок.

15.2 Особенности организации Column Families

В HBase колонки группируются в Column Families (CF), при этом каждая колонка идентифицируется как `<ColumnFamily> : <Column>`. Для каждой CF можно задать отдельный алгоритм сжатия, что позволяет оптимизировать хранение под тип данных. CF сохраняются в одном или нескольких HFile, отсортированных лексикографически, что обеспечивает быстрый доступ к данным за счет бинарного поиска.

Конфигурация CF задается статически и может быть изменена только

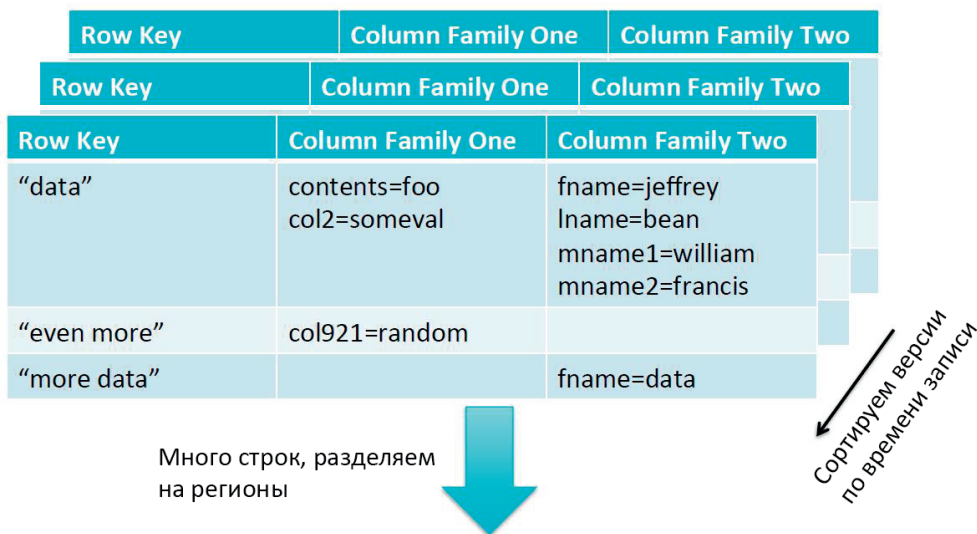


Рис. 204: Пример таблицы в HBase

после перезапуска таблицы. Число CF должно оставаться небольшим — это упрощает архитектуру и повышает производительность. В отличие от CF, количество колонок является динамическим: их можно добавлять во время выполнения, просто указав имя. В одной CF допускается наличие миллионов колонок.

На Рисунке 205 показано, что одинаковые имена колонок в разных CF не конфликтуют. Например, col1 в Family1 и Family2 рассматриваются как независимые.

15.3 Версионность данных

Значения в колонках являются версионными: для одной ячейки хранится несколько версий значений. Количество хранимых версий настраивается в конфигурации CF и по умолчанию равно трем. Версия значения (timestamp) является дополнительным измерением для данных и задается неявно при записи данных (данные записывает **RegionServer**) или явно указывается

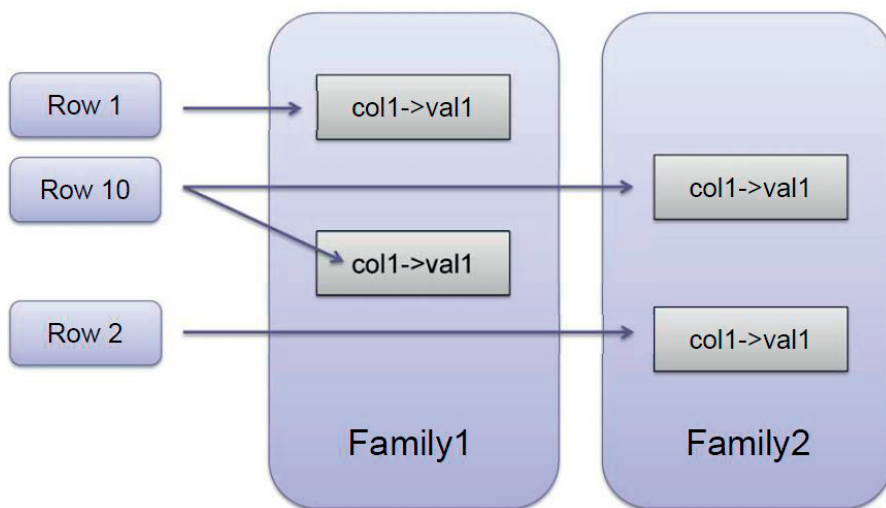


Рис. 205: Пример именования колонок в CF

пользователем. Версии хранятся в убывающем порядке значений timestamp, при этом последнее значение читается первым, что оптимизирует чтение текущего значения. Для получения значения в ячейке необходимо вычислить непосредственно таблицу, ряд, семейство колонок, саму колонку и необходимую версию, то есть момент времени, за который требуется получить данные.

Таким образом, **Value = Table + RowKey + ColumnFamily + Column + Timestamp**.

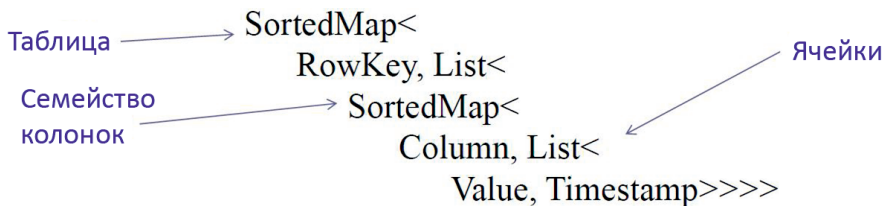


Рис. 206: Структура значения в колонке

На Рисунке 207 изображен пример таблицы в HBase. Данная таблица состоит из двух строк: row1 и row2. Также представлены два объединения колонок — это Name Family и Address Family. В свою очередь каждое из данных

объединений состоит из двух колонок: для первого — это колонки `first_name` и `last_name`; для второго, соответственно, `number` и `address`. Значения, которые хранятся в ячейках, являются версионными. Первый ряд имеет моменты времени: `t1`, `t5`, `t10` и `t15`. При этом значение `Bob` было занесено в момент времени `t1`, которое далее в моменты времени `t5`, `t10` и `t15` не менялось. Если требуется получить, допустим, данные за момент времени `t15`, будет выведено значение `Bob`. Дополнительно хранить `t15` и `first_name`: `Bob` не имеет смысла. Аналогичным образом хранятся значения для всех других ячеек в данной таблице.

Row Key	Time Stamp	Name Family		Address Family	
		first_name	last_name	number	address
row1	t1	<u>Bob</u>	<u>Smith</u>		
	t5			10	First Lane
	t10			30	Other Lane
	t15			<u>7</u>	<u>Last Street</u>
row2	t20	<u>Mary</u>	Tompson		
	t22			<u>77</u>	<u>One Street</u>
	t30		<u>Thompson</u>		

Рис. 207: Пример таблицы в HBase

15.4 Хранение и управление данными

`ColumnFamily` определяет способ разделения данных. Для каждой CF данные хранятся в одном или нескольких файлах в HDFS. При этом, как можно видеть на Рисунке 208, данные физически хранятся по семействам колонок (на картинке в одном файле записывается CF `fam1`, в другом — CF `fam2`). Для каждого row могут быть определены не все значения коло-

нок: ячейки, которые не имеют значения, храниться не будут. Rows хранятся как множество ячеек, и сортировка происходит сначала по RowId, затем по ColumnQualifier.

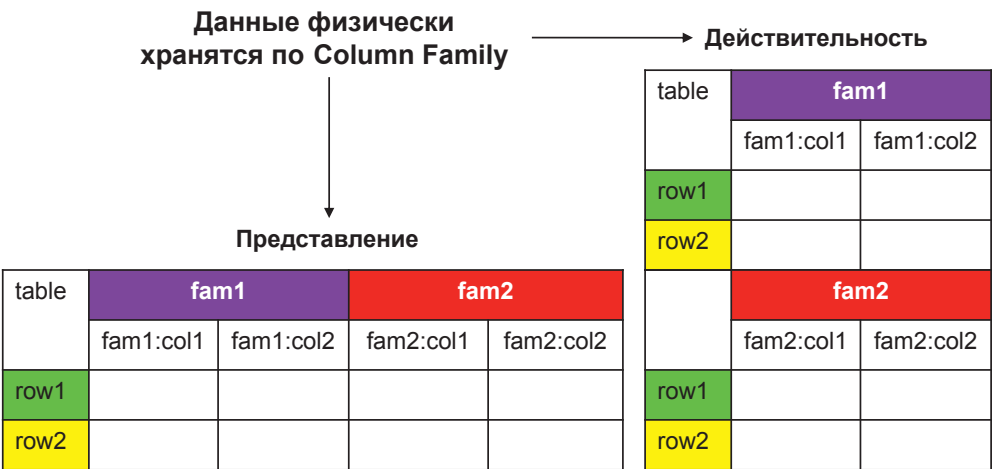


Рис. 208: Пример хранения данных ⁶¹

HBase поддерживает четыре типа операций с данными:

- Put — операция вставки данных. Синтаксис: `put 'table1', 'row1', 'family1:column1', ('version'), 'value'`. Указываются таблица, ряд, семейство колонок и сама колонка (версия опциональна), а также значение ячейки. При дублировании значений строки, столбца и версии данные будут заменены;
- Delete — операция удаления данных. Синтаксис: `delete 'table1', 'row1', 'family1:column1', ('version')`. В случае использования команды в виде `delete 'table1', 'row1'` будут удалены данные всей строки;
- Get — операция выбора одной строки. Синтаксис: `get 'table1', 'row1', {COLUMN => 'family1:column1', VERSIONS => 4}`. Основной

⁶¹<https://www.cs.utexas.edu/~dsb/cs386d/Projects14/HBase.pdf>

синтаксис может быть расширен путем указания столбца и количества версий для выборки. По умолчанию выбирается одна версия столбца;

- **Scan** — операция выбора набора строк. Синтаксис: `scan 'table1', {STARTROW  $\Rightarrow$  'row1', STOPROW  $\Rightarrow$  'row10', VERSIONS  $\Rightarrow$  2, LIMIT  $\Rightarrow$  15}`. Используется редко и обычно расширяется указанием начальной и конечной строки выборки, предельным количеством строк в выборке, а также количеством версий.

15.5 Запросы данных

Существует несколько способов запроса данных из таблиц HBase. Каждый из них ориентирован на определенный тип доступа и имеет особенности производительности:

- **По идентификатору строки (RowId) или набору RowId.** В этом случае извлекаются только строки, соответствующие заданным ключам. Если не указаны дополнительные условия, из каждой строки будут возвращены все ячейки, принадлежащие всем CF таблицы. Такой подход приводит к необходимости объединения большого числа файлов в HDFS, поскольку каждая CF хранится в отдельном HFile;
- **По Column Family.** Запрос, ограниченный конкретным семейством колонок, позволяет значительно сократить объем загружаемых файлов, повышая производительность запроса;
- **По временной метке (Timestamp) или версии (Version).** Запросы такого типа позволяют исключать из обработки HFile-файлы, не содержащие данных в заданном временном диапазоне. Это эффективно снижает нагрузку при анализе версионных данных;
- **По названию колонки (Column Name) или уточнителю (Qualifier).** Такой фильтр позволяет извлекать только определенные

колонки, минимизируя объем передаваемых данных;

- **По значению (Value).** Поиск по значению является наименее эффективным способом, так как требует проверки содержимого каждой ячейки. Однако он поддерживается с применением соответствующих фильтров;
- **С использованием нескольких критериев одновременно.** HBase поддерживает комбинированные запросы, включающие фильтрацию по ряду, семейству колонок, имени колонки, временным меткам и значению.

15.6 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

- Что представляет из себя HBase?
- В чем отличия HBase и BigTable?
- Можно ли назвать HBase реляционной базой данных?
- Можно ли интегрировать HBase с JAVA API?
- Когда выгодно использовать HBase?
- Что такое ColumnFamily?
- Как хранятся данные в HBase?
- Зачем необходимы Timestamps?
- Как можно осуществлять запрос данных из HBase?

ГЛАВА 16

HBase, BigTable: архитектура

В рамках данной главы рассматриваются ключевые компоненты HBase и BigTable, включая структуру таблиц, механизм хранения и доступа к данным, а также взаимодействие с HDFS и ZooKeeper. Особое внимание уделяется принципам масштабирования, обеспечению отказоустойчивости и особенностям работы с Column Families.

16.1 Архитектура HBase

HBase является распределенной базой данных, которая может работать на десятках и сотнях физических серверов, обеспечивая бесперебойную работу даже при выходе из строя некоторых из них. Архитектура HBase является существенно более сложной по сравнению с классическими реляционными системами управления базами данных. Рассмотрим **структуру таблиц** в HBase:

- **Таблица** состоит из одного или нескольких **регионов**;
- **Регион** представляет собой набор строк таблицы, сгруппированных по диапазону ключей:
 - Является основной единицей разбиения данных и служит механизмом масштабирования;
 - При достижении заданного размера регион может автоматически разделиться на два (split), а при необходимости — объединиться с другим регионом. Эти процессы могут быть отключены по конфигурации;

- **Region Server (RS)** — сервер, обслуживающий один или несколько регионов:
 - Развертывается на каждом рабочем узле кластера;
 - Каждый регион обслуживается только одним Region Server;
 - В случае сбоя Region Server соответствующие регионы перераспределяются между оставшимися серверами автоматически;
- **Master Server (HMaster)** — управляет всеми RS и располагается на выделенном узле кластера;
- Хранение данных в HBase осуществляется в **HDFS**;
- Для реализации распределенной координации и управления состоянием кластера HBase использует систему **ZooKeeper** [25].

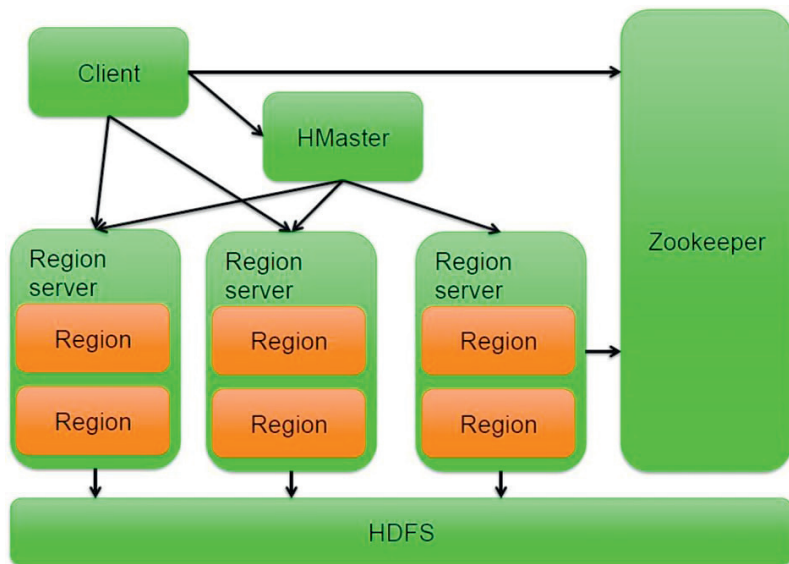


Рис. 209: Архитектура HBase

На Рисунке 209 показана архитектура HBase. Как видно из схемы, единственный экземпляр HMaster управляет множеством Region Servers. Каждый

Region Server может обслуживать несколько регионов, при этом один регион всегда закреплен только за одним сервером. Данные физически хранятся в HDFS, а взаимодействие между компонентами и координация процессов осуществляются с помощью ZooKeeper.

16.2 HBase Region

Поскольку HBase является key-value хранилищем и партиционирование осуществляется по ключу, то крайне важно разделять данные равномерно по всем регионам:

- **Регион — диапазон ключей:**

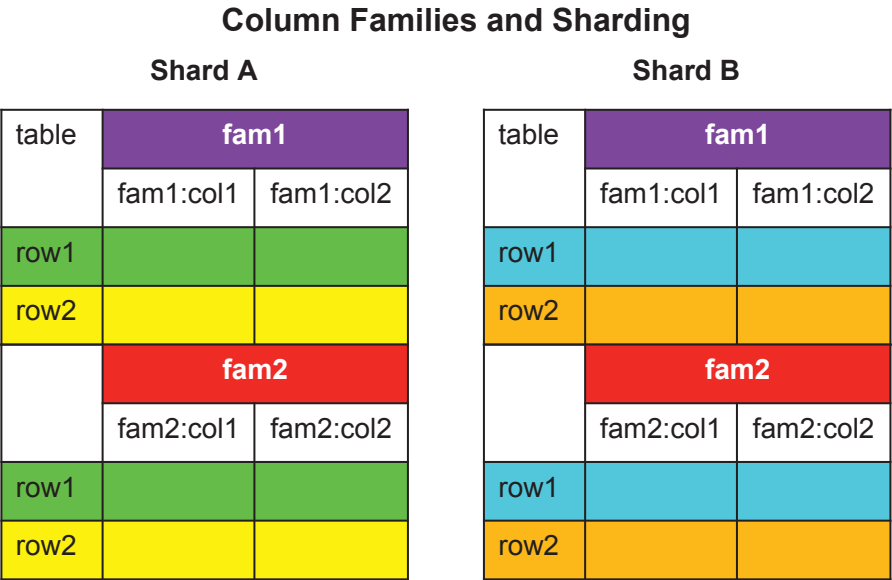
- Start key (включается в регион) → Stop key (не включается в регион);
- От 10 до 1000 регионов на каждом Region Server;
- Размер региона от 1 до 2 GB (использовать регионы маленького размера очень затратно).

- **Добавление данных:**

- Изначально определяется только один регион (можно предварительно задать количество регионов);
- Если размер региона превысит лимит (записывая новые данные), то он разбивается на 2 региона по среднему ключу.

Разбиение на регионы и разбиение каждого региона обеспечивают:

- Быстрое восстановление, если возникли проблемы с регионом (равномерное перераспределение ресурсов);
- Возможность параллельного считывания данных с разных RS;



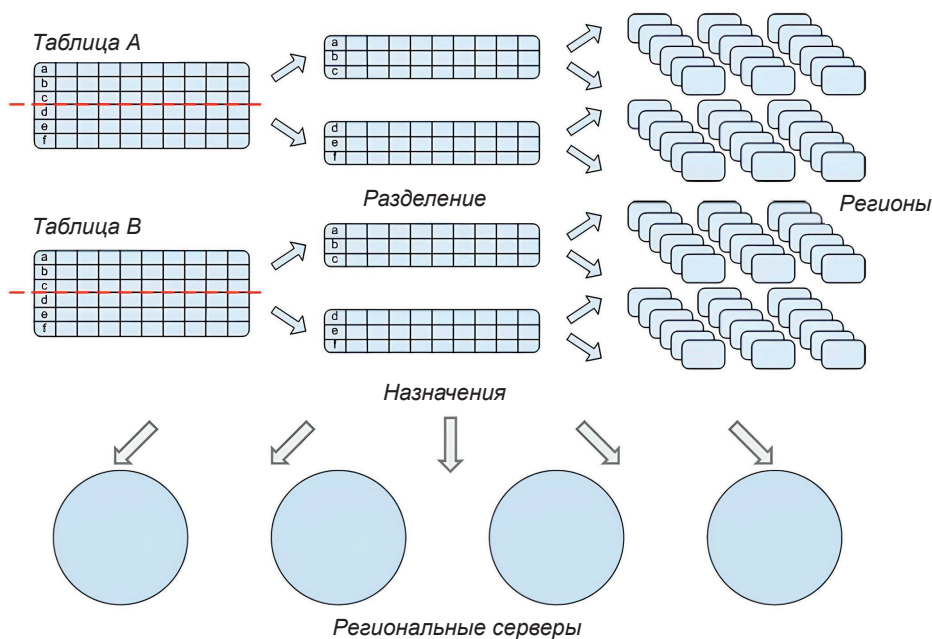


Рис. 211: Разбиение таблиц на части

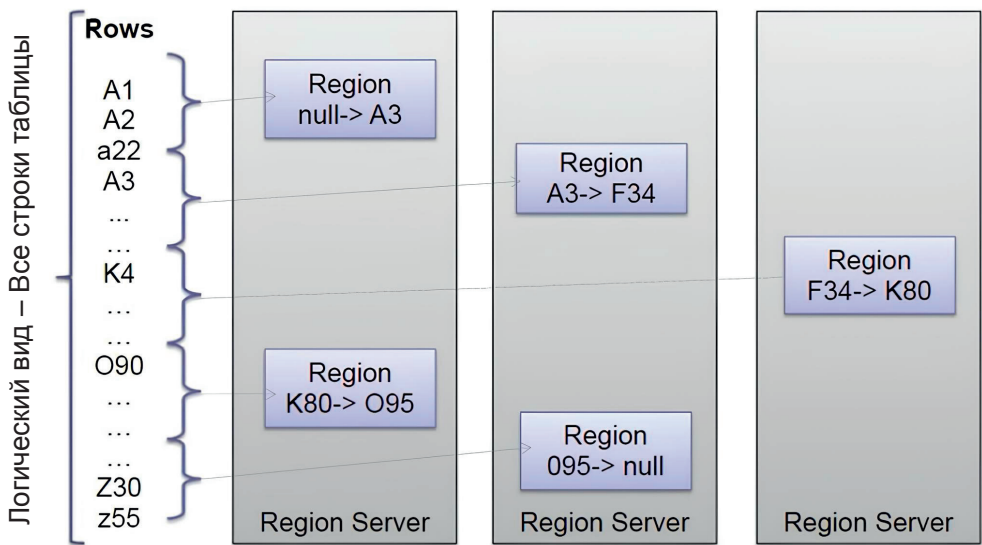


Рис. 212: Распределение Rows между Region Servers

16.3 Условия хранения данных в HBase

Данные в HBase хранятся в специальных файлах **HFiles** (также называемых StoreFiles), которые размещаются в HDFS. Каждый HFile представляет собой структуру хранения вида key-value map, в которой ключи упорядочены в лексикографическом порядке. Порядок сортировки следующий: сначала строки упорядочиваются по ключу (Row Key), затем по имени семейства колонок (ColumnFamily), затем по именам колонок внутри семейства, далее — по временной метке (Timestamp), и, наконец, указывается соответствующее значение ячейки. Данная структура иллюстрируется на Рисунке 213.

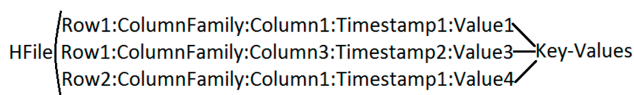


Рис. 213: Формат хранения в HDFS

При добавлении новых данных в HBase они записываются в два хранилища: в журнал **WAL (Write-Ahead Log)** и во временное хранилище в памяти, называемое **memstore** (работает в рамках Region Server). WAL реализован как стандартный файл формата Hadoop SequenceFile и служит для защиты от потери данных в случае сбоя сервера. Если сервер выйдет из строя до того, как данные будут сброшены в постоянное хранилище (HFile), то они смогут быть восстановлены из WAL. Когда объем данных в memstore достигает заданного порогового значения, происходит операция **flush**, в результате которой данные выгружаются в HFile. После этого соответствующие записи могут быть удалены из WAL, поскольку они уже зафиксированы на диске.

Важно отметить, что файловая система HDFS не поддерживает модификацию или удаление содержимого файлов после их записи. Это означает, что невозможно удалить отдельную пару key-value из HFile. Вместо физического удаления создаются специальные пометки — **delete markers**. Маркеры используются для обозначения удаленных записей и могут указывать на уда-

ление строки или колонки. Во время выполнения запросов (runtime) delete markers позволяют фильтровать такие записи.

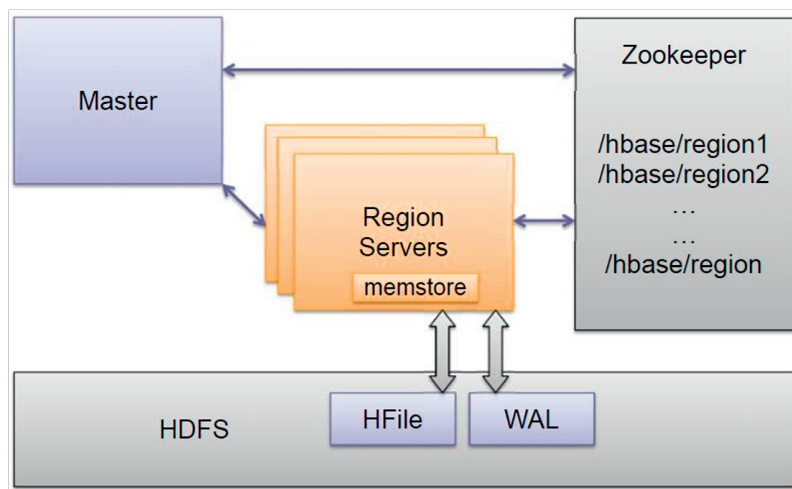
Со временем в кластере накапливается большое количество HFiles, что сказывается на производительности. Для контроля количества файлов и балансировки кластера в HBase предусмотрены **операции компакта** (data compactions). Существуют два вида таких операций — это Minor-compaction и Major-compaction. **Minor-compaction** заключается в объединении несколько небольших HFile в один более крупный. Эта процедура выполняется быстро и не учитывает delete markers. **Major-compaction** заключается в объединении в единый файл всех HFile внутри одного региона с одной ColumnFamily. В процессе Major-compaction delete markers применяются, и удаленные записи не переносятся в результирующий файл.

16.4 Компоненты HBase

Обратим отдельное внимание на компоненты HBase (см. Рисунок 214). HMaster управляет Region Servers и для координации действий поддерживает связь с ZooKeeper. Несмотря на то что RS обладает собственной памятью, все данные хранятся в HDFS (в HFile и WAL).

HMaster отвечает за управление регионами в HBase: он назначает регионы на Region Servers, выполняет их перебалансировку и восстанавливает недоступные регионы. Для координации HMaster использует ZooKeeper, который отслеживает состояние компонентов кластера, сообщает клиентам местоположение строк и гарантирует наличие единственного активного HMaster. При этом HMaster не обрабатывает данные напрямую — клиенты взаимодействуют с Region Servers. Основной задачей HMaster является координация и управление схемой таблиц, включая создание и удаление таблиц и ColumnFamilies.

⁶²<https://vitalflux.com/hbase-architecture-components-beginners-dummies/>

Рис. 214: Компоненты HBase ⁶²

16.5 Доступ к HBase

Наконец, рассмотрим основные способы взаимодействия с системой HBase. Базовым способом является использование интерактивной оболочки командной строки — HBase Shell. Кроме того, HBase предоставляет полноценный Java API, с помощью которого можно программно обращаться к данным, управлять таблицами и выполнять операции чтения и записи.

Также поддерживается доступ к HBase через Avro Server⁶³ и библиотеку PyHBase⁶⁴, которая реализует клиент на языке Python, использующий интерфейс Avro. Одним из популярных решений для удаленного взаимодействия с HBase является REST-сервер (см. Рисунок 215), поддерживающий передачу данных в форматах JSON и XML. Еще один подход — использование Thrift-сервера⁶⁵, обеспечивающего мультиплатформенную поддержку и возможность работы с HBase на Ruby, C++, C# и т.д.

⁶³<http://avro.apache.org/>

⁶⁴Python client for HBase Avro interface <https://github.com/hammer/pyhbase>

⁶⁵<http://thrift.apache.org/>

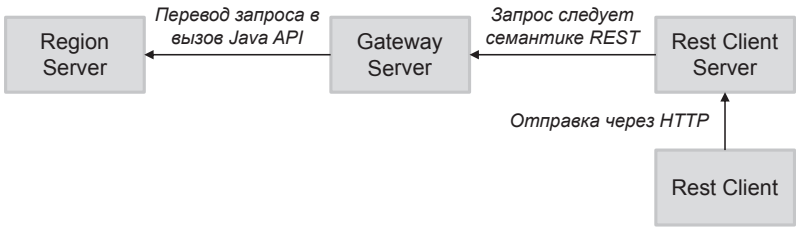


Рис. 215: REST

На Рисунке 216 представлены различные средства доступа к HBase. Напомним, что физическое хранение данных осуществляется в HDFS, а за распределенную координацию отвечает ZooKeeper. Взаимодействие с HBase может осуществляться через Java-клиент (напрямую или через программы на MapReduce, Hive, Pig), либо через серверы Thrift и REST. Кроме того, возможно создание собственных Java-приложений, использующих клиентскую библиотеку HBase.

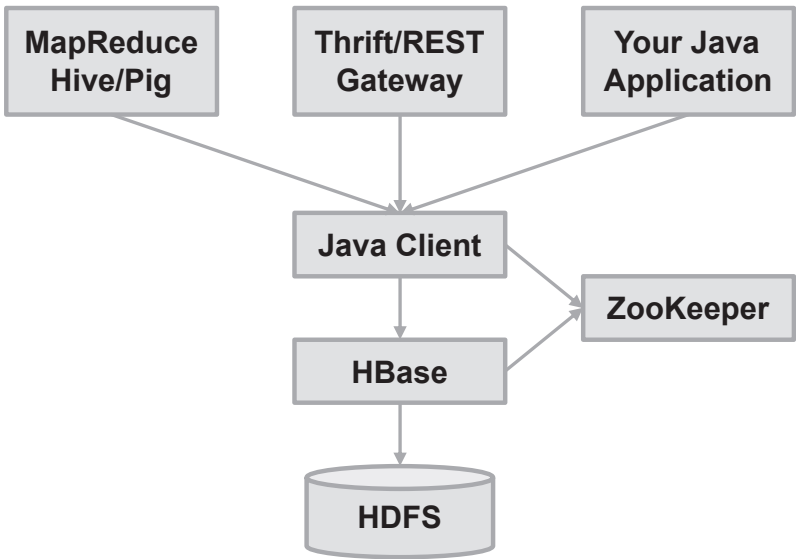


Рис. 216: Схема доступа к HBase

16.6 Контрольные вопросы

Для закрепления материала предлагается дать ответы на следующие вопросы:

- Что представляет из себя архитектура HBase?
- Что такое регион?
- Опишите процесс разбиения региона на сплиты.
- Что такое WAL? Зачем он необходим?
- Опишите различные способы доступа к HBase.

Заключение

В рамках данного пособия рассмотрены ключевые технологии и архитектурные решения современных методов хранения и обработки больших данных. На базе классической модели MapReduce и распределенной файловой системы GFS показано, как горизонтальное масштабирование кластера позволяет линейно увеличивать производительность при работе с терабайтными наборами данных. Рассмотрены этапы развития Hadoop: переход от монолитного планировщика JobTracker к двухуровневой модели YARN, появление высокоуровневых инструментов Hive, Pig и Spark.

Отдельное внимание уделено системам хранения. Рассмотрены текстовые, последовательные и колоночные форматы (RCFile, ORC), их роль в сокращении коммуникационных затрат и ускорении обработки выборок данных. На примере HBase и Google Bigtable показано, как колоночная модель с Column Families, журналом WAL и HFile-хранилищем обеспечивает произвольный доступ к миллионам столбцов и версируемость данных, оставаясь отказоустойчивой за счет HDFS и ZooKeeper.

Практическая часть охватывала методы оптимизации: секционирование и кэширование в Spark, Broadcast Join и External Shuffle Service, механизмы compaction в HBase и стратегии стоимостного планирования запросов. Показаны способы интеграции систем и принципы организации вычислительных сценариев через JobControl, Oozie или цепочки Map/Reduce.

Освоение архитектур и инструментов обработки больших данных представляет собой важный этап профессионального становления. Знание внутреннего устройства систем Spark, Pig, Hive, HBase и Bigtable формирует целостное инженерное понимание современных решений в области распределенных вычислений и создает прочную основу как для научной, так и для прикладной деятельности в сфере анализа и обработки данных.

Список литературы

- [1] White Tom. Hadoop: The definitive guide. — "O'Reilly Media, Inc. 2012.
- [2] Pentreath Nick. Machine learning with Spark. — Packt Publishing Birmingham, 2015.
- [3] Кузнецов Сергей. К свободе от проблемы Больших Данных // Открытые системы. СУБД. — 2012. — no. 2. — P. 1–22.
- [4] Sparke Linda S, Gallagher III John S. Galaxies in the universe: an introduction. — Cambridge University Press, 2007.
- [5] Ghemawat Sanjay, Gobioff Howard, Leung Shun-Tak. The Google file system // Proceedings of the nineteenth ACM symposium on Operating systems principles. — 2003. — P. 29–43.
- [6] Dean Jeffrey, Ghemawat Sanjay. MapReduce: simplified data processing on large clusters // Communications of the ACM. — 2008. — Vol. 51, no. 1. — P. 107–113.
- [7] The Hadoop distributed file system / Shvachko Konstantin, Kuang Hairong, Radia Sanjay, and Chansler Robert // 2010 IEEE 26th symposium on mass storage systems and technologies (MSST) / IEEE. — 2010. — P. 1–10.
- [8] Lam Chuck. Hadoop in action. — Simon and Schuster, 2010.
- [9] Sammer Eric. Hadoop operations. — "O'Reilly Media, Inc. 2012.
- [10] Apache Hadoop YARN: Yet another resource negotiator / Vavilapalli Vinod Kumar, Murthy Arun C, Douglas Chris, Agarwal Sharad, Konar Mahadev, Evans Robert, Graves Thomas, Lowe Jason, Shah Hitesh, Seth Siddharth, et al. // Proceedings of the 4th annual Symposium on Cloud Computing. — 2013. — P. 1–16.

- [11] Omega: flexible, scalable schedulers for large compute clusters / Schwarzkopf Malte, Konwinski Andy, Abd-El-Malek Michael, and Wilkes John // Proceedings of the 8th ACM European Conference on Computer Systems. — 2013. — P. 351–364.
- [12] Mesos: A platform for fine-grained resource sharing in the data center / Hindman Benjamin, Konwinski Andy, Zaharia Matei, Ghodsi Ali, Joseph Anthony D, Katz Randy H, Shenker Scott, and Stoica Ion // NSDI. — 2011. — Vol. 11. — P. 1–22.
- [13] Dominant resource fairness: Fair allocation of multiple resource types / Ghodsi Ali, Zaharia Matei, Hindman Benjamin, Konwinski Andy, Shenker Scott, and Stoica Ion // NSDI. — 2011. — Vol. 11. — P. 1–24.
- [14] Leskovec Jure, Rajaraman Anand, Ullman Jeffrey David. Mining of massive data sets. — Cambridge university press, 2020.
- [15] Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing / Zaharia Matei, Chowdhury Mosharaf, Das Tathagata, Dave Ankur, Ma Justin, McCauly Murphy, Franklin Michael J, Shenker Scott, and Stoica Ion // Presented as part of the 9th Symposium on Networked Systems Design and Implementation. — 2012. — P. 15–28.
- [16] Learning Spark: lightning-fast big data analysis / Karau Holden, Konwinski Andy, Wendell Patrick, and Zaharia Matei. — "O'Reilly Media, Inc. 2015.
- [17] Karau Holden, Warren Rachel. High performance Spark: best practices for scaling and optimizing Apache Spark. — "O'Reilly Media, Inc. 2017.
- [18] Energy-aware cluster reconfiguration algorithm for the big data analytics platform Spark / Duan Kairong, Fong Simon, Song Wei, Vasilakos Athanasios V, and Wong Raymond // Sustainability. — 2017. — Vol. 9, no. 12. — P. 2357.
- [19] RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems / He Yongqiang, Lee Rubao, Huai Yin, Shao Zheng,

- Jain Namit, Zhang Xiaodong, and Xu Zhiwei // 2011 IEEE 27th International Conference on Data Engineering / IEEE. — 2011. — P. 1199–1208.
- [20] Hankins Richard A, Patel Jignesh M. Data morphing: An adaptive, cache-conscious storage technique // Proceedings 2003 VLDB Conference / Elsevier. — 2003. — P. 417–428.
- [21] Copeland George P, Khoshafian Setrag N. A decomposition storage model // ACM Sigmod Record. — 1985. — Vol. 14, no. 4. — P. 268–279.
- [22] Weaving Relations for Cache Performance / Ailamaki Anastassia, DeWitt David J, Hill Mark D, and Skounakis Marios // VLDB. — 2001. — Vol. 1. — P. 169–180.
- [23] Major technical advancements in Apache Hive / Huai Yin, Chauhan Ashutosh, Gates Alan, Hagleitner Gunther, Hanson Eric N, O'Malley Owen, Pandey Jitendra, Yuan Yuan, Lee Rubao, and Zhang Xiaodong // Proceedings of the 2014 ACM SIGMOD international conference on Management of data. — 2014. — P. 1235–1246.
- [24] Bigtable: A distributed storage system for structured data / Chang Fay, Dean Jeffrey, Ghemawat Sanjay, Hsieh Wilson C, Wallach Deborah A, Burrows Mike, Chandra Tushar, Fikes Andrew, and Gruber Robert E // ACM Transactions on Computer Systems (TOCS). — 2008. — Vol. 26, no. 2. — P. 1–26.
- [25] ZooKeeper: wait-free coordination for Internet-scale systems / Hunt Patrick, Konar Mahadev, Junqueira Flavio Paiva, and Reed Benjamin // USENIX annual technical conference. — 2010. — Vol. 8.

Reviewers:

Mikhail Petrovskiy – Ph.D., Associate Professor, Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University;

Alexander Chernov – Ph.D., Associate Professor, Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University

Vasilev, Iulii.

Modern Methods of Distributed Data Storage and Processing : textbook / I.A. Vasilev, S.V. Gerasimov, M.A. Kazachuk. – Moscow : MAKS Press, 2025. – 336 p. ISBN 978-5-317-07462-3
<https://doi.org/10.29003/m4759.978-5-317-07462-3>

This teaching aid contains an extended lecture outline for the course «Modern Methods of Distributed Data Storage and Processing», taught at the Department of Intelligent Information Technologies, Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University.

Keywords: Distributed computing, stream data processing, computation optimization, resource scheduling.

The authors express their gratitude for invaluable assistance in preparing this manual to the students and graduates of the Department of Intelligent Information Technologies, Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University: Dmitry Konovalchuk, Evgeny Kuptsov, Artemy Leshonkov, Vladimir Lisovoy, Kirill Malyshev, Danila Stepochkin, Denis Trofimov, Oleg Fateev, Ilyas Kharunov, Vadim Chernyshev, as well as Nataliia Vasileva for her help in formatting and editing the illustrations.

*The publication of this book was supported by Non-commercial Foundation
for the Advancement of Science and Education «INTELLECT»*

Учебное издание

ВАСИЛЬЕВ Юлий Алексеевич
ГЕРАСИМОВ Сергей Валерьевич
КАЗАЧУК Мария Андреевна

**СОВРЕМЕННЫЕ МЕТОДЫ РАСПРЕДЕЛЕННОГО ХРАНЕНИЯ
И ОБРАБОТКИ ДАННЫХ**

Учебно-методическое пособие

Издательство «МАКС Пресс»
Главный редактор: *Е.М. Бугачева.*
Обложка: *А.В. Кононова*

Напечатано с готового оригинал-макета

Подписано в печать 10.10.2025 г.

Формат 70х100 1/16. Усл.печ.л. 27,3. Тираж 300 (1-40) экз. Заказ 132.

Издательство ООО «МАКС Пресс». Лицензия ИД N 00510 от 01.12.99 г.

119992, ГСП-2, Москва, Ленинские горы, МГУ им. М.В. Ломоносова,
2-й учебный корпус, 527 к. Тел. 8(495)939-3890/91. Тел./Факс 8(495)939-3891

Отпечатано в полном соответствии с качеством предоставленных материалов в ООО «Фотоэксперт»
109316, г. Москва, Волгоградский проспект, д. 42, корп. 5, эт. 1, пом. I, ком. 6.3-23Н