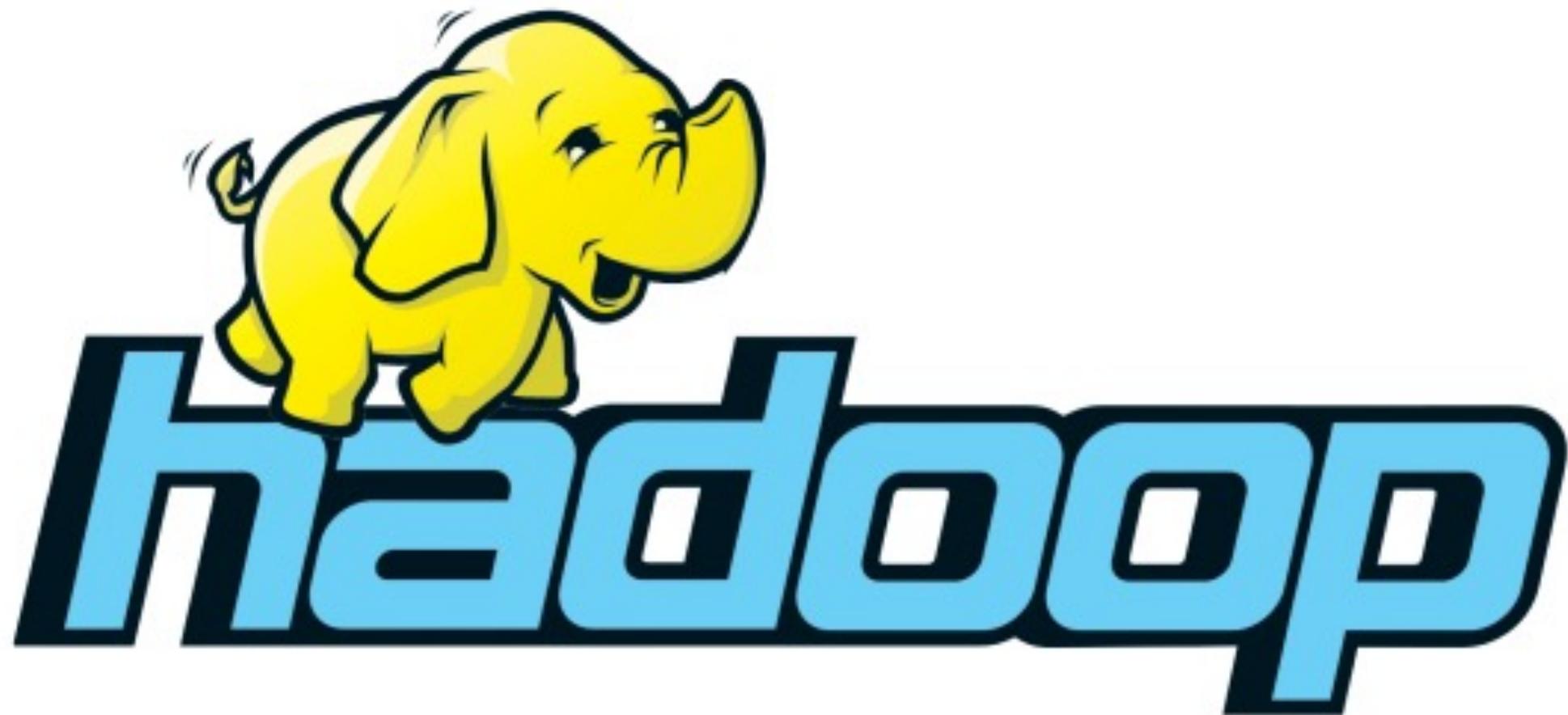


Обзор Hadoop



О чём поговорим?

- 1) Экосистема Hadoop
- 2) HDFS
- 3) YARN
- 4) MapReduce
- 5) Некоторые команды в Hadoop
- 6) Amabri
- 7) Waggle Dance

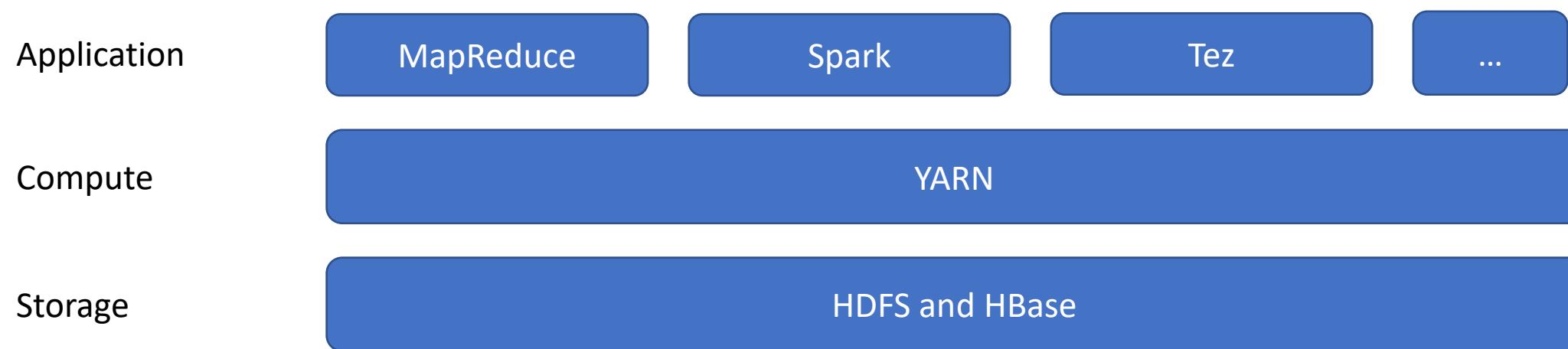


Экосистема Hadoop

Экосистема Hadoop

- HDFS – распределенная файловая система
- YARN – менеджер ресурсов кластера (CPU, RAM...)
- MapReduce – API для распределенных вычислений

Экосистема Hadoop



Зачем это все?

Раньше очень хорошо все работало на реляционных базах данных. Данных больше – купили сервер побольше и проблем нет. Но как-то не виден потенциал сильной масштабируемости, правда?

M - масштабируемость

- Раньше данных было меньше и они чаще были структурированными. Заранее продумали схему реляционной базы, нужные запросы и все, сложили данные в таблицы.
- Как ускорить обработку в RDBMS (Relation Database Management System)? Правильно, купить сервер побольше (вертикальная масштабируемость).
- В Hadoop для ускорения обработки нужно больше средних серверов (горизонтальная масштабируемость), что значительно дешевле.

RDBSM или Hadoop?

	RDBMS	Hadoop
Количество серверов	Один мощный	Много средних
Объем данных	Маленький	Большой
Скорость запросов	Быстрый ответ	Ответ с задержкой
Формат данных	Таблицы (структурированные)	Файлы (неструктурированные)
Требования ACID*	Выполняются	Не требуются

ACID – требования к сохранности данных в терминах атомарности, согласованности, изолированности и надежности.

RDBMS или Hadoop?

В хадупе нет индексов, есть задержка ответа, но зато мы точно ответим. Но вот в хадупе нет требований, он под пакетную обработку, есть вероятность потери части данных при обработке. Но опять же, это та парадигма, которая была в самом начале пути Hadoop.



HDFS

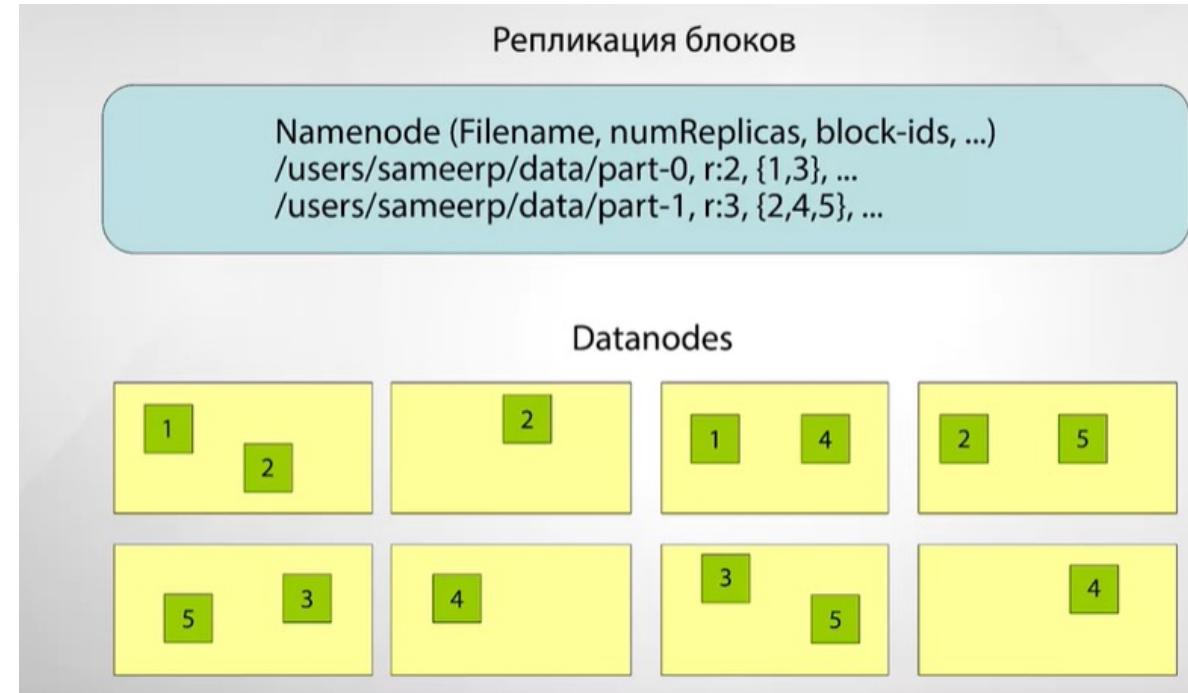
HDFS – Hadoop Distributed File System

- Файлы хранятся по кусочкам на машинах и велика вероятность потерять часть данных, если одна из тачек накроется.
- Для этого существует репликация и контролируется фактором репликации, по умолчанию 3.
- Нужен справочник по типу имя файла – блок. Это хранится на специальной тачке.

HDFS

- Файлы разбиты на блоки, которые хранятся на разных машинах (Data Nodes)
- Каждый блок реплицируется на нескольких Data Nodes, количество которых настраивается при конфигурации системы
- Соответствие имя файла → блоки хранится в памяти специальной машины (Name Node)

Репликация блоков



Зачем нужна репликация

Давайте вспомним теорию вероятностей...

- Пусть сервер ломается с вероятностью 0.001
- Какова вероятность, что 1 из 500 серверов сломается?
- $1 - (1 - 0.001)^{500} = 0.3936$

Чем больше серверов, тем выше вероятность поломки => нужен приемлемый фактор репликации

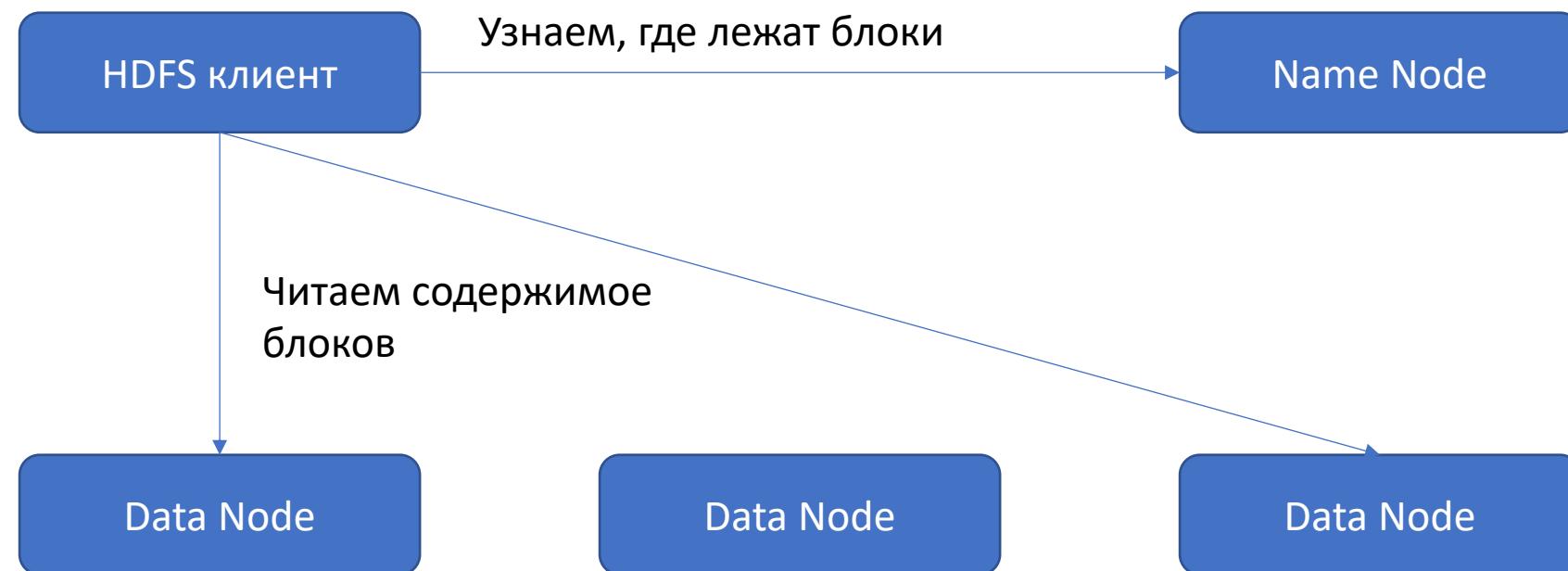
Более приземленный пример

Прошлый слайд скорее относится к тяжелым кластерам типа ЛД БР, но даже на примере наших типовых кластеров:

- 20 нод: $1 - (1 - 0.001)^{20} = 0.0198$ или 2%
- 30 нод: $1 - (1 - 0.001)^{30} = 0.0296$ или 3%

Уже не мало, учитывая, что мы работаем с ними регулярно.

Чтение из HDFS

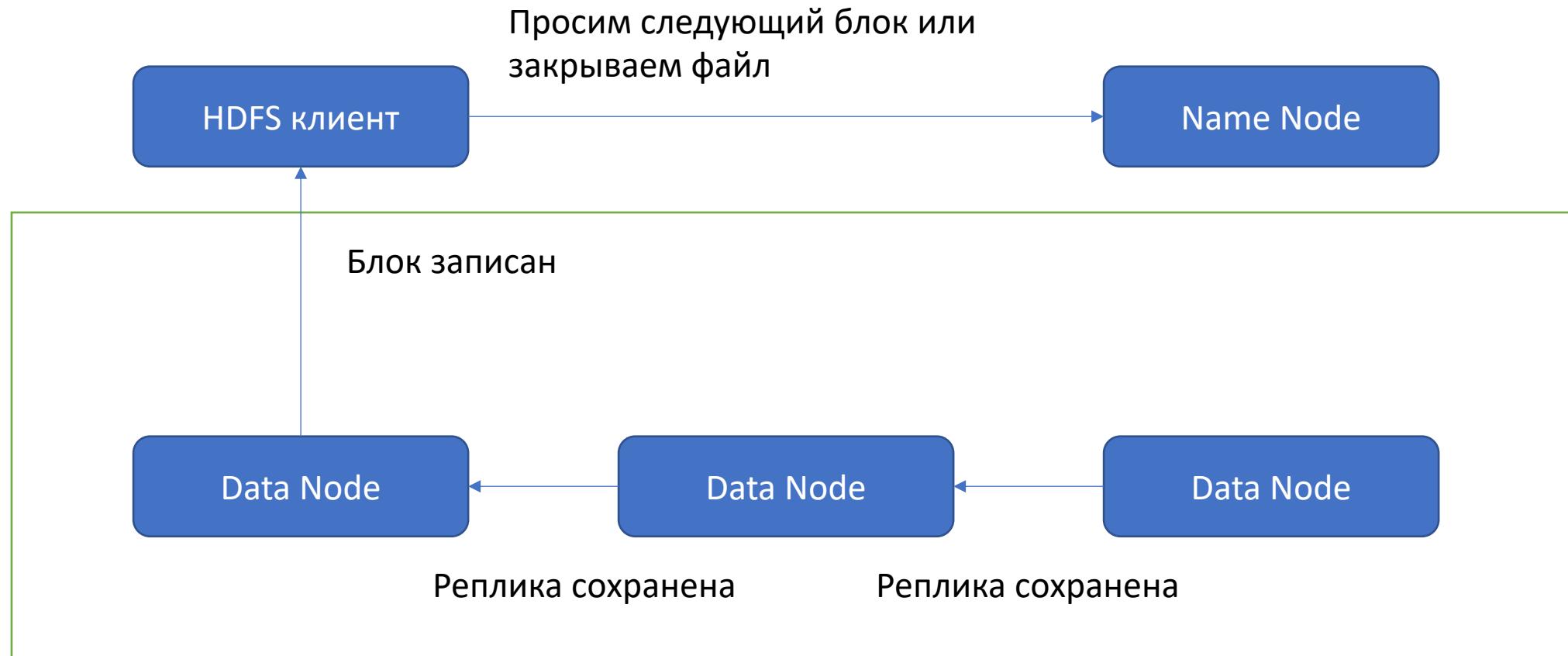


Запись в HDFS

Запись выглядит не так просто, как чтение данных.

- Спрашиваем у Name Node, можно ли записать файл по выбранному пути, она проверит и отдаст ответ.
- Пишем блоками с репликацией. Name Node скажет, куда записать, так как она балансирует нагрузку.
- Данные пишутся через *streaming* на первую Data Node.
- После первая Data Node сообщает второй Data Node, что у нее есть данные для реплики, а вторая говорит третьей и делается это асинхронно.
- То есть на скорость записи реплики не влияют и мы можем писать дальше (ну кроме нагрузки на диски).

Запись в HDFS

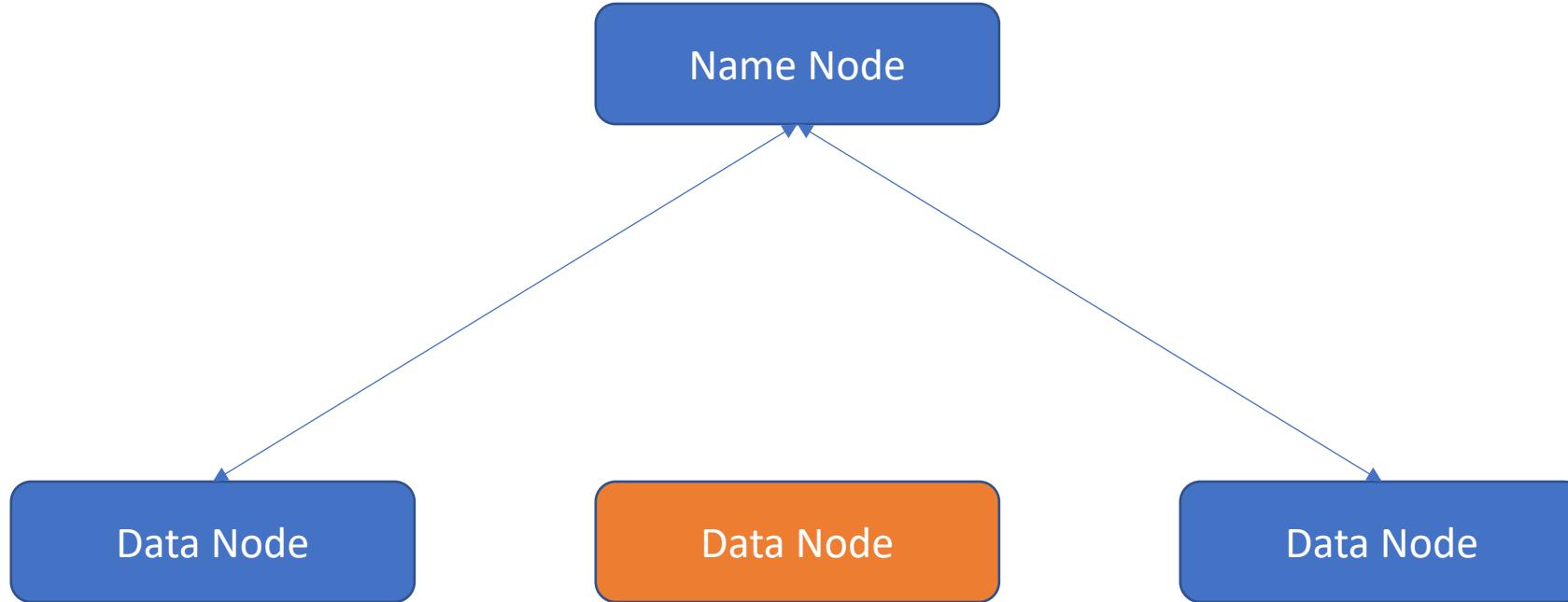


Устойчивость HDFS

Data Nodes регулярно сообщают Name Node, что они все еще живы и способны работать (heartbeat). Если сообщения нет, то беда! Name Node может принять решение вывести машину из кластера (decommission) с репликацией всех ее блоков на другие машины.

Простой пример: собираете транзакции и вдруг вместо 30 нод на кластере доступны только 10, остальные отдыхают.

Устойчивость HDFS



Умерла нода – копируем данные и поддерживаем фактор репликации!

Нюанс устойчивости HDFS

Не стоит делать блоки слишком маленькими. Тогда их количество будет огромным и Name Node не справится, понадобится очень мощный сервер.

Parquet

Parquet — это бинарный, колоночно-ориентированный формат хранения данных, изначально созданный для экосистемы Hadoop.

Одна из уникальных особенностей parquet заключается в том, что в таком формате он может хранить данные сложенными структурами. Это означает, что в файле parquet даже вложенные поля можно читать по отдельности без необходимости читать все поля во вложенной структуре. Для хранения вложенных структур Parquet использует алгоритм измельчения и сборки (shredding and assembly).

Файлы имеют несколько уровней разбиения на части, благодаря чему возможно довольно эффективное параллельное исполнение операций поверх них:

Row-group — это разбиение, позволяющее параллельно работать с данными на уровне Map-Reduce

Column chunk — разбиение на уровне колонок, позволяющее распределять IO операции

Page — Разбиение колонок на страницы, позволяющее распределять работу по кодированию и сжатию

Parquet

Представим таблицу:

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

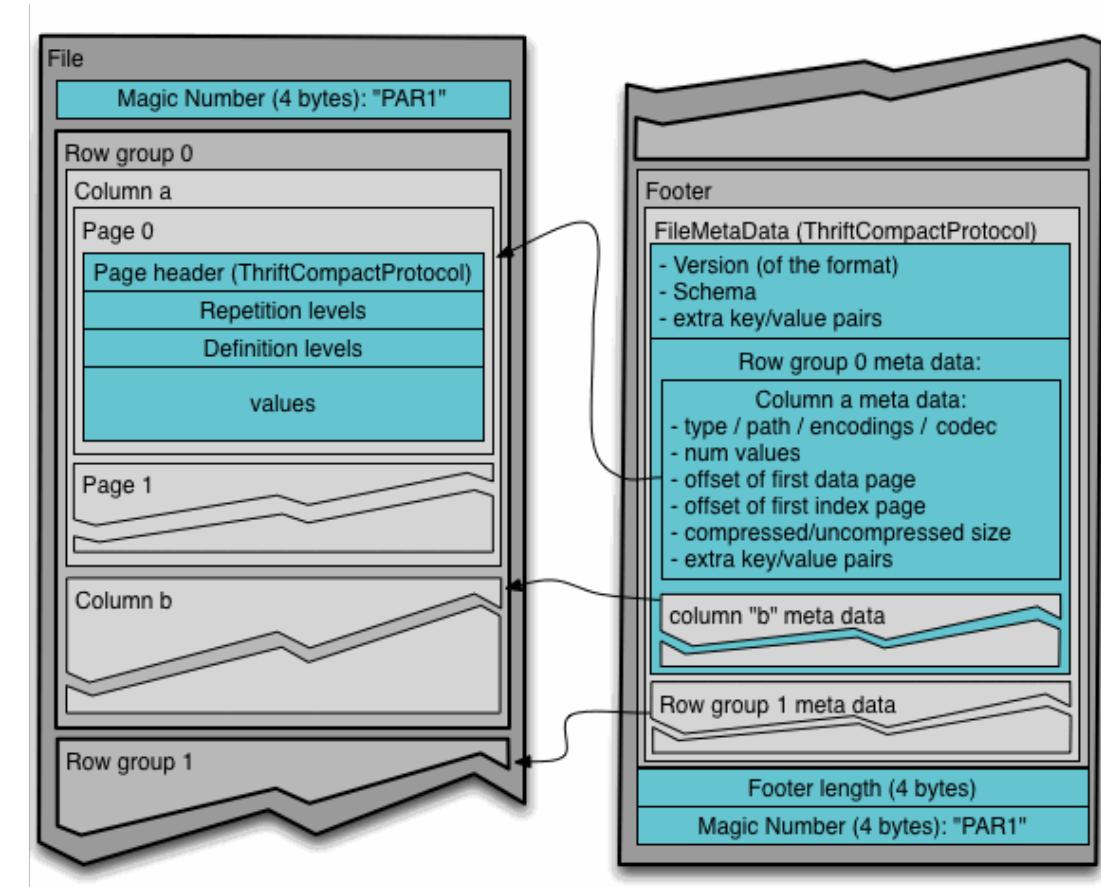
Тогда в текстовом файле, скажем, csv мы бы хранили данные на диске примерно так:

A1	B1	C1	A2	B2	C2	A3	B3	C3
----	----	----	----	----	----	----	----	----

В случае с Parquet:

A1	A2	A3	B1	B2	B3	C1	C2	C3
----	----	----	----	----	----	----	----	----

Благодаря этому мы можем считывать только необходимые нам колонки.



Минусы parquet

- Не поддерживает ACID;
- Обновление схемы данных = перезапись;
- Фильтр строк – full scan;
- Не любит, когда забирают все столбцы.

Оптимизированный строково-столбчатый формат файлов (Optimized Row Columnar) предлагает очень эффективный способ хранения данных и был разработан, чтобы преодолеть ограничения других форматов. Хранит данные в идеально компактном виде, позволяя пропускать ненужные детали — при этом не требует построения больших, сложных или обслуживаемых вручную индексов.

Преимущества формата ORC:

Один файл на выходе каждой задачи, что уменьшает нагрузку на NameNode.

1. Поддержка типов данных Hive, включая DateTime, десятичные и сложные типы данных (struct, list, map и union).
2. Одновременное считывание одного и того же файла разными процессами.
3. Возможность разделения файлов без сканирования на наличие маркеров.
4. Оценка максимально возможного выделения памяти кучи на процессы чтения/записи по информации в футере файла.
5. Метаданные сохраняются в бинарном формате сериализации, который позволяет добавлять и удалять поля.

ORC хранит коллекции строк в одном файле, а внутри коллекции строчные данные хранятся в столбчатом формате.

Файл ORC хранит группы строк, которые называются полосами (stripes) и вспомогательную информацию в футере файла. Postscript в конце файла содержит параметры сжатия и размер сжатого футера.

По умолчанию размер полосы составляет 250 МБ. За счет полос такого большого размера чтение из HDFS выполняется более эффективно: большими непрерывными блоками.

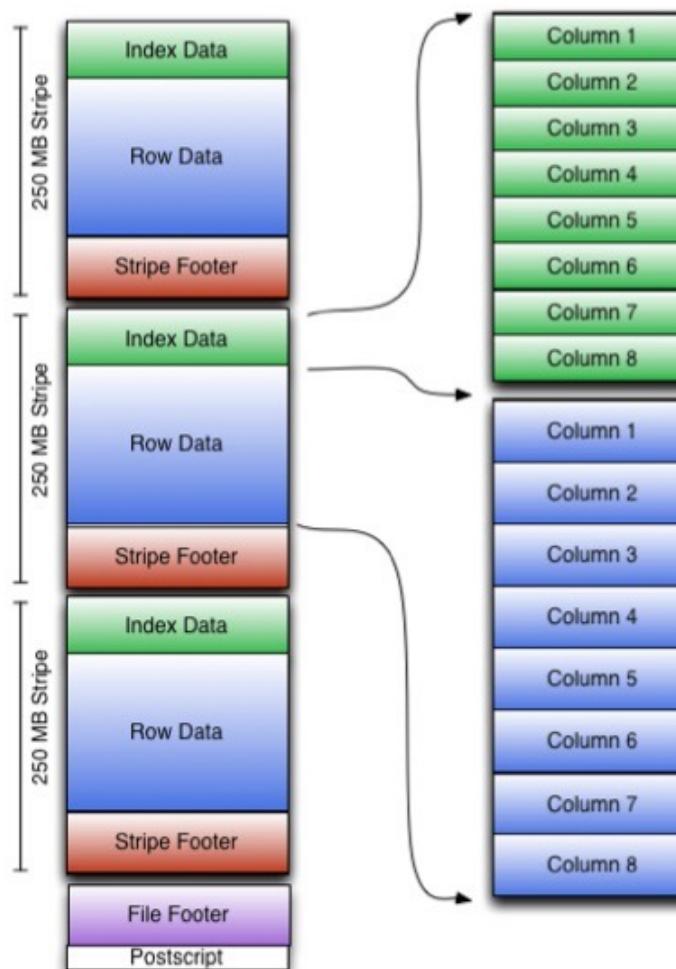
В футере файла записан список полос в файле, количество строк на полосу и тип данных каждого столбца. Там же записано результирующее значение count, min, max и sum по каждому столбцу.

Футер полосы содержит каталог местоположений потока.

Строчные данные используются при сканировании таблиц.

Индексные данные включают минимальные и максимальные значения для каждого столбца и позиции строк в каждом столбце. Индексы ORC используются только для выбора полос и групп строк, а не для ответа на запросы.

ORC



ORC vs Parquet

1. Parquet лучше хранит вложенные данные.
2. ORC лучше приспособлен к проталкиванию предикатов (predicate pushdown).
3. ORC поддерживает свойства ACID.
4. ORC лучше сжимает данные.



YARN

YARN – Yet Another Resource Negotiator. Модуль отвечает за управление ресурсами кластера и управление задачами.

Под управлением YARN могут быть как MapReduce-программы, так и иные поддерживающие YARN распределенные приложения.

Что было до YARN

В Hadoop версии 1.* YARN отсутствовал, был только JobTracker. Он пытался рулить всем и падал при большом количестве задач не позволяя масштабироваться.

В YARN один менеджер, который следит за всем, но для каждой задачи создается свой «короткоживущий менеджер», который отвечает за свою задачу в рамках своего процесса, что дает возможность параллельной работы.

Также Hadoop проектировался только для MapReduce, YARN расширил возможности.

Компоненты YARN

- ResourceManager
- ApplicationMaster
- NodeManager
- Container

ResourceManager

Менеджер ресурсов, задачей которого является распределение ресурсов, необходимых для работы приложений, и наблюдение за вычислительными узлами, на которых эти приложения выполняются.

Состоит из:

- Scheduler - планировщик, ответственный за распределение ресурсов между приложениями, которые нуждаются в ресурсах, с учетом ограничений вычислительных мощностей, наличия очереди и т.д. Scheduler не ведет мониторинга и не отслеживает статус приложений. Он также не дает никаких гарантий о совершении перезапуска неудачных задач из-за сбоя в работе приложения или аппаратного сбоя.
- ApplicationManager - компонент, ответственный за прием задач и запуск экземпляров ApplicationMaster, а также мониторинг узлов (контейнеров), на которых происходит выполнение, и предоставляет сервис для перезапуска контейнера ApplicationMaster при сбое.

ApplicationMaster

Компонент, ответственный за планирование жизненного цикла, координацию и отслеживание статуса выполнения распределенного приложения. Каждое приложение имеет свой экземпляр ApplicationMaster. ApplicationMaster управляет всеми аспектами жизненного цикла, включая динамическое увеличение и уменьшение потребления ресурсов, управление потоком выполнения, обработку ошибок и искажений вычислений и выполнение других локальных оптимизаций.

NodeManager

Агент, запущенный на вычислительном узле и отвечающий за отслеживание используемых вычислительных ресурсов (ЦП, RAM и т.д.), за управление логами и за отправку отчетов по используемым ресурсам планировщику менеджера ресурсов ResourceManager/Scheduler. NodeManager управляет абстрактными контейнерами, которые представляют собой ресурсы на узле, доступные для конкретного приложения.

Container

Это набор физических ресурсов, таких CPU, RAM и прочее.

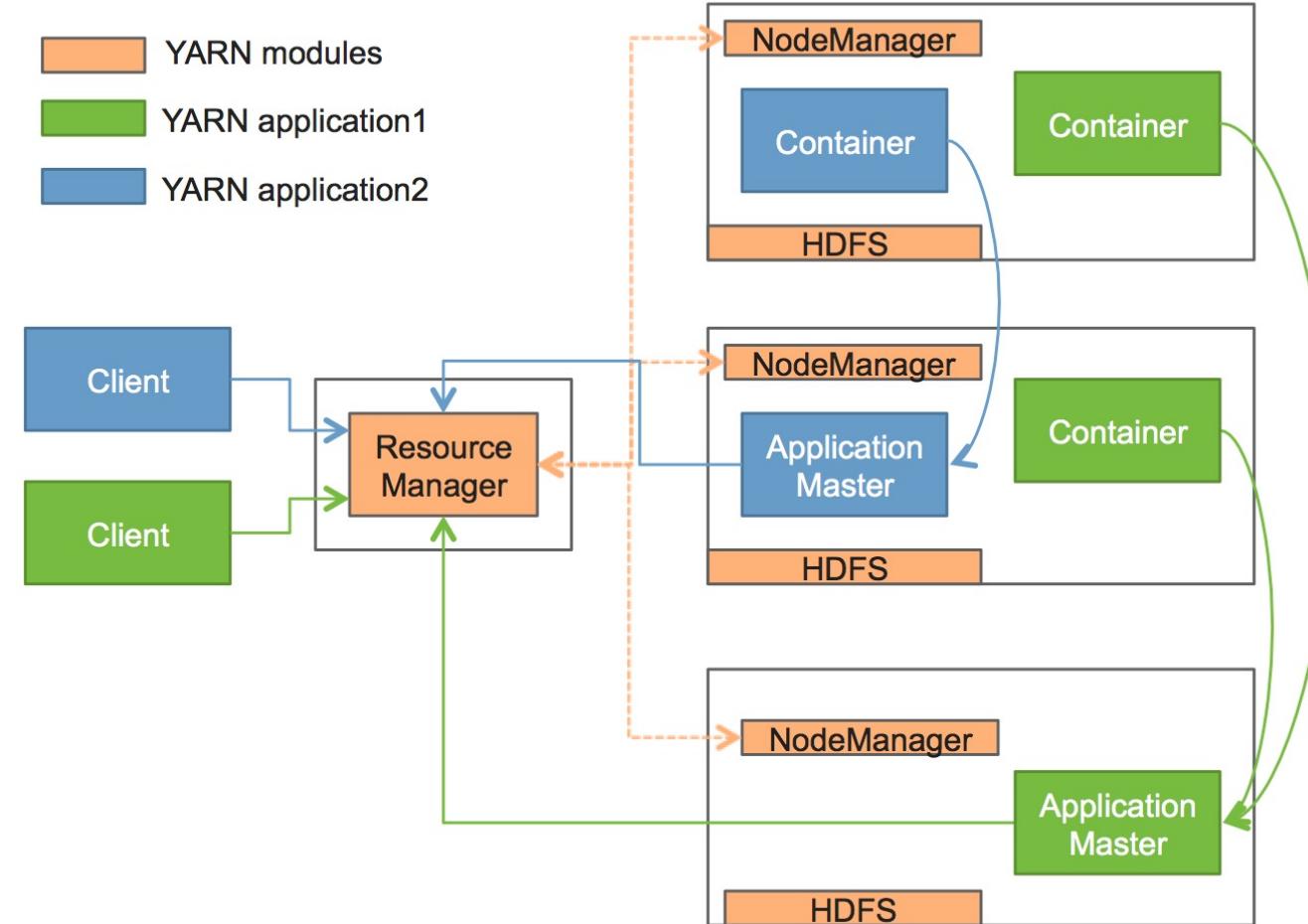
Принцип работы YARN

- Приходит запрос от клиента
- ResourceManager выделяет необходимые ресурсы для контейнера и запускает ApplicationMaster для обслуживания указанного приложения.
- Запуском контейнеров управляет *Container Launch Context (CLC)*, приходящий в качестве запроса от ApplicationMaster к NodeManager.
- ApplicationMaster выделяет контейнеры для приложения в каждом узле и контролирует их работу до завершения работы приложения.
- Чтобы запустить контейнер, NM копирует все необходимые зависимости - файлы данных, исполняемые файлы, архивы - в локальное хранилище.
- Когда задача завершена, ApplicationMaster отменяет выделения контейнера в ResourceManager, и цикл завершается.

Клиент может отслеживать состояние приложения, обращаясь к ResourceManager или напрямую к ApplicationMaster, если он поддерживает такую функцию. При необходимости клиент также может завершить работу приложения.

Поскольку АМ сам является контейнером, работающим в кластере, он должен быть устойчивым к сбоям. YARN обеспечивает некоторую поддержку для восстановления в случае сбоев, но поскольку отказоустойчивость и семантика приложений тесно связаны, большая часть нагрузки все равно ложится на АМ .

Принцип работы YARN





MapReduce

Парадигма MapReduce

Все началось с google и задачи word count. Блоки хранятся на разных машинах и передача данных по сети не самое лучшее решение, хочется иметь возможность обрабатывать блоки там, где они и хранятся (быстрое чтение с диска). Идеальный мир:

- Если в задаче можно обрабатывать блоки независимо, то можно достигнуть идеальной масштабируемости (embarrassingly parallel);
- Например, задача фильтрации строк в файле идеально масштабируется.

Задача WordCount

- В HDFS лежат сохраненные тексты со всего Интернета
- Хотим узнать частоты всех слов, а потом может и на tf-idf замахнемся

Как решать?

- Для каждого блока посчитаем частоты слов в нем (вроде идеально масштабируется)
- Сложим частоты по всем блокам

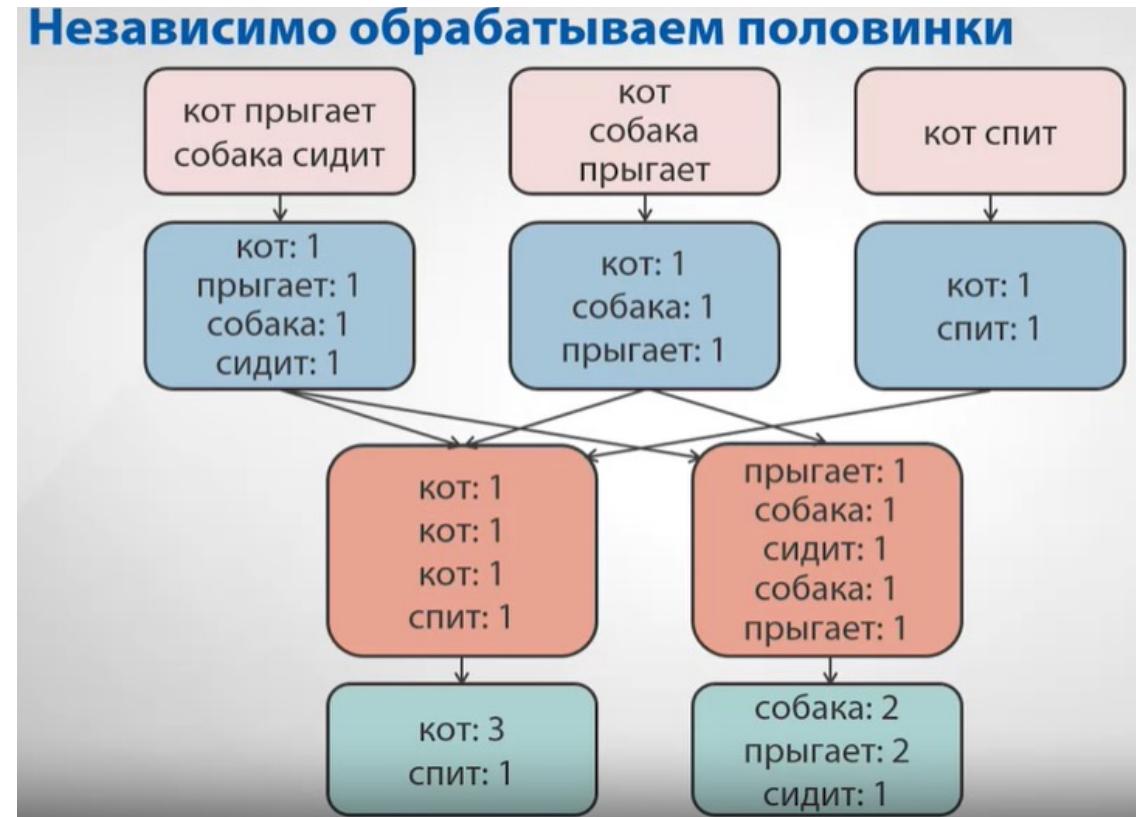
А где масштабируемость?

- Надеемся, что словари будут небольшими и при пересылке данных по сети мы не упадем.
- Отправляем все на одну машину и она там пусть суммирует.
- Вроде все просто, только вот нет масштабируемости, пока все передадим, пока посчитаем...



Как добиться масштабируемости?

А давайте перемешаем слова между двумя машинами и просуммируем тоже по частям? Само собой, что на одной машине должна быть вся информация о конкретном слове.



Обобщим масштабируемость

- Мы хотим разделить задачу на N частей;
- Делить будем по значению хэша слова $\text{hash}(\text{word}) \% N$, которое и покажет нам, на какую машину нужно отправить слово;
- Если в хэш-функции все значения равновероятны, мы не должны получить скоса распределения на каких-то тачках;
- Например, можно взять полиномиальную функцию $\text{hash}(x) = x[0] + x[1]p^1 + \dots + x[n]p^n$

х – строка

p - фиксированное простое число

x[i] - код символа

А вот и MapReduce

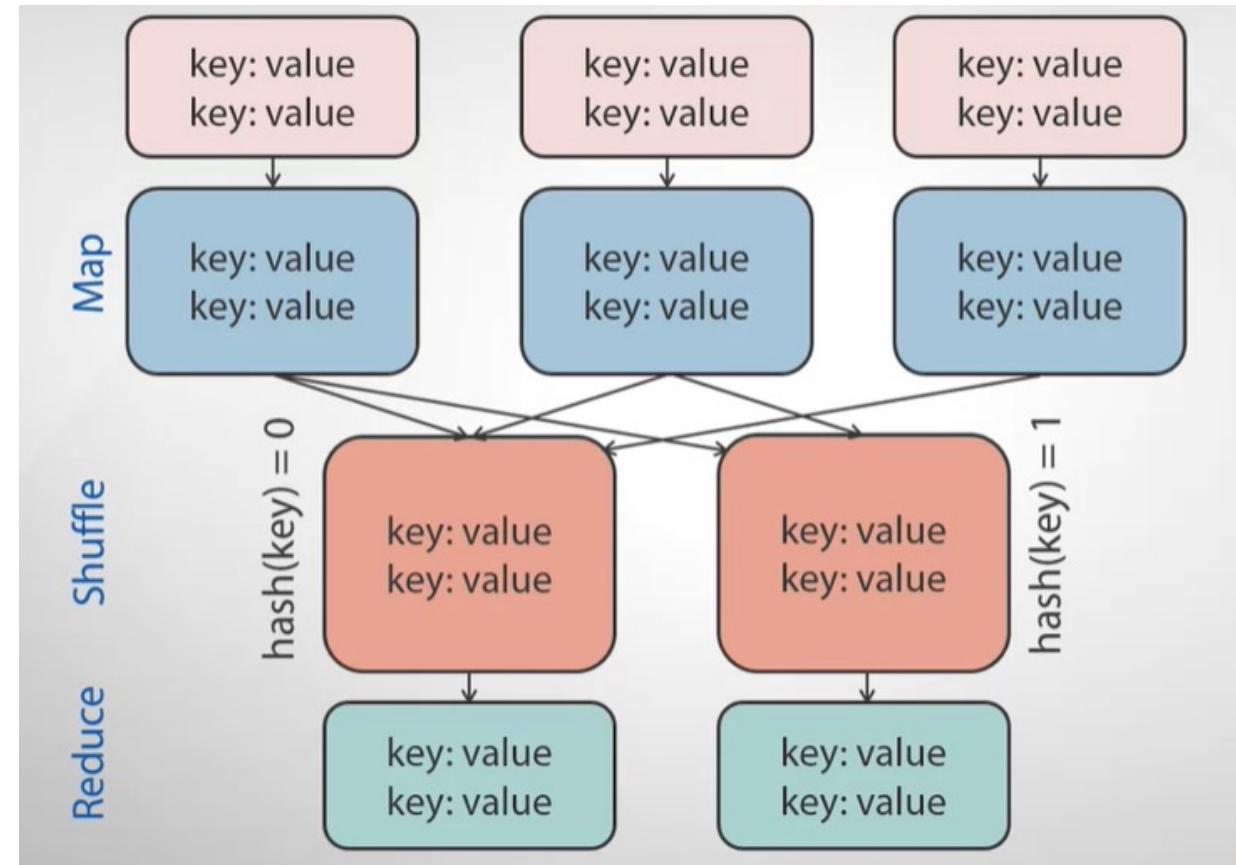
Стоп. Все слышали MapReduce и если спросить что там происходит, то ответят: "сначала мапим, потом редьюсим". Но мы забываем про важный этап, который часто занимает существенную часть времени..знакомьтесь – Shuffle.



Обобщим парадигму MapReduce

В итоге надо сводить задачу к уже решенной, как мы и любим. Сводим вот к такому виду.

Надо задать Map и Reduce, Shuffle работает сам детерминировано (видимо поэтому про него забывают).



Опишем алгоритм

- Map
 $(K1, V1) \rightarrow List(K2, V2)$
(номер строки, "кот спит") $\rightarrow [("кот", 1), ("спит", 1)]$
- Shuffle
Ключи делим по $hash(key) \% N$ на N частей. Каждую часть сортируем по key (независимо). Тогда значения для одного ключа будут обрабатываться непрерывно.
- Reduce
 $(K2, List(V2)) \rightarrow List(K3, V3)$
(("кот"), (1, 1, 1)) $\rightarrow [('кот', 3)]$

Hadoop MapReduce

У данной реализации есть свои фишки.

Самая тяжелая часть в MapReduce – это shuffle, то есть вычисление хеша, сортировка и обмен данными между нодами.

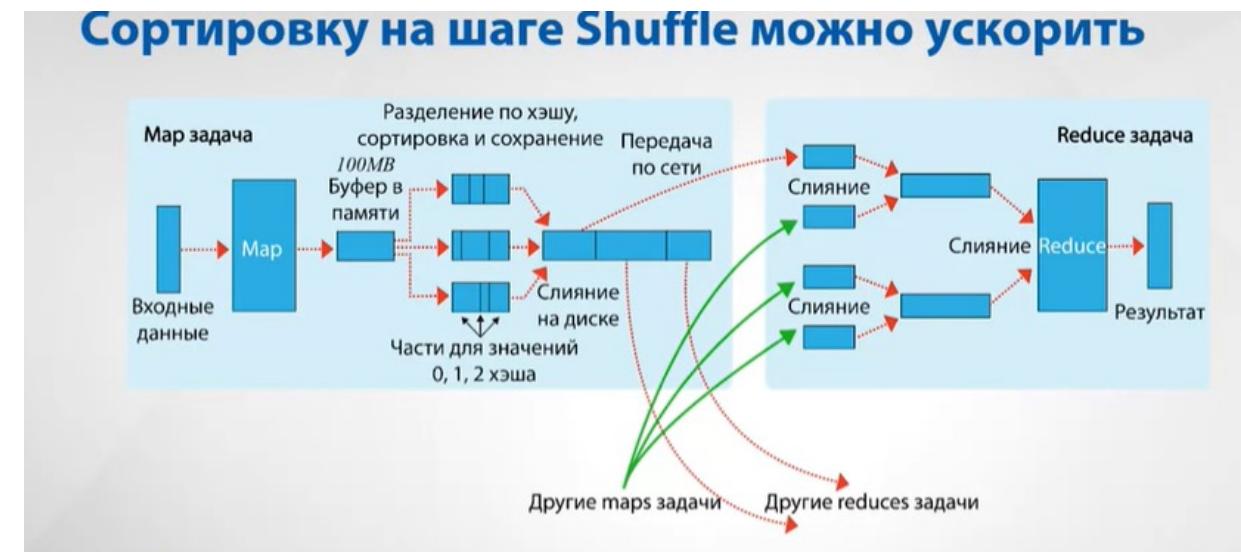
Можно сортировать внутри мапера, делая это кусками по 100 Мб в оперативке.

3 отсортированных списка за 1 проход объединяются в один отсортированный. Reduce получает уже отсортированные данные. То есть в Hadoop идет предсортировка мапером и сортировка слиянием на стороне редюсеров.

Сложно, сейчас глянем на картинке:

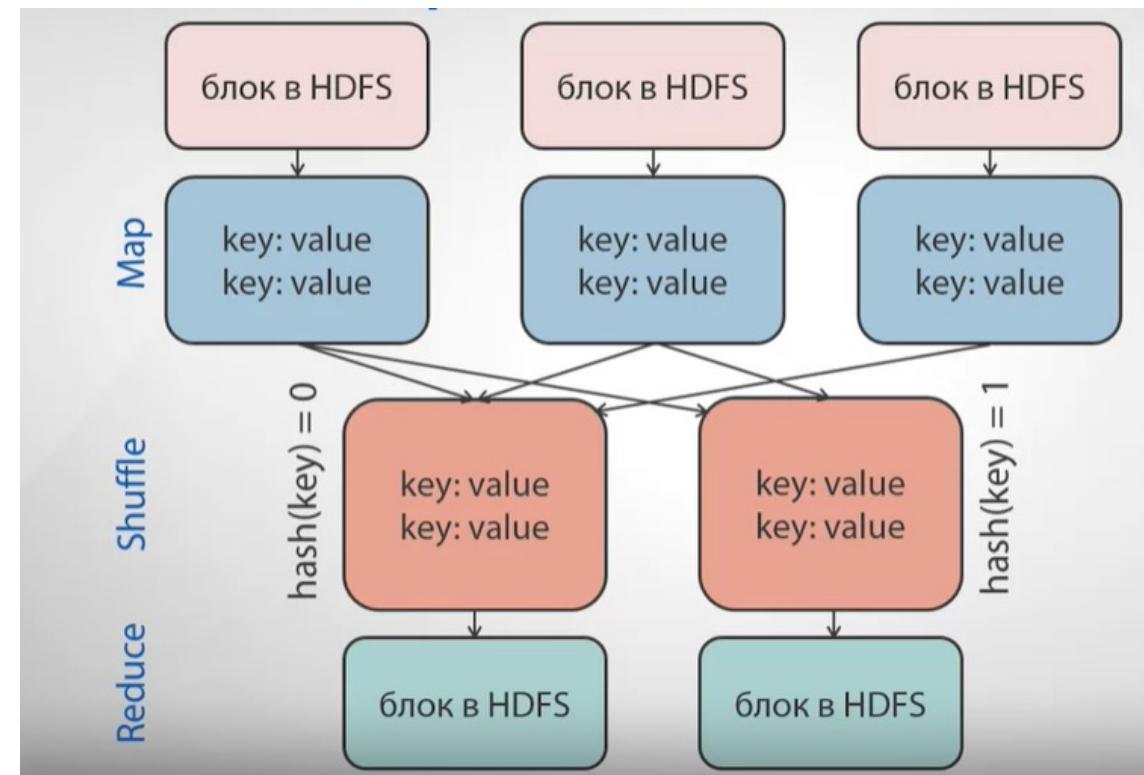
Hadoop MapReduce

- Результаты Мар-шага сортируются локально
- После используем merge sort на шаге Shuffle, делая это за линейное время



Так, но потом надо все собрать для записи

Нет, не надо ничего собирать! Мы же хотим все хранить распределенно, а в Reduce уже все отсортировано и готово к записи, у каждого свой уникальный набор слов (может быть что угодно).



А что если упадет?

- Потеряли маппер – перезапустили задачу только для его блока
- Потеряли редьюсер – собрали по хэшу данные со всех мапперов и перезапустили задачу. Дольше и тяжелее, но не с 0 стартуем=)

Некоторые команды

hadoop fs –du –h hdfs://path – вычислит размер папки по указанному пути

hadoop fs –mkdir hdfs://path – создаст папку по указанному пути

hadoop fs –rm -r skipTrash hdfs://path – удалит файлы по указанному пути

hadoop fs -distcp hdfs://path1 hdfs://path2 – копирование path1 из одного hdfs в path2 другого hdfs

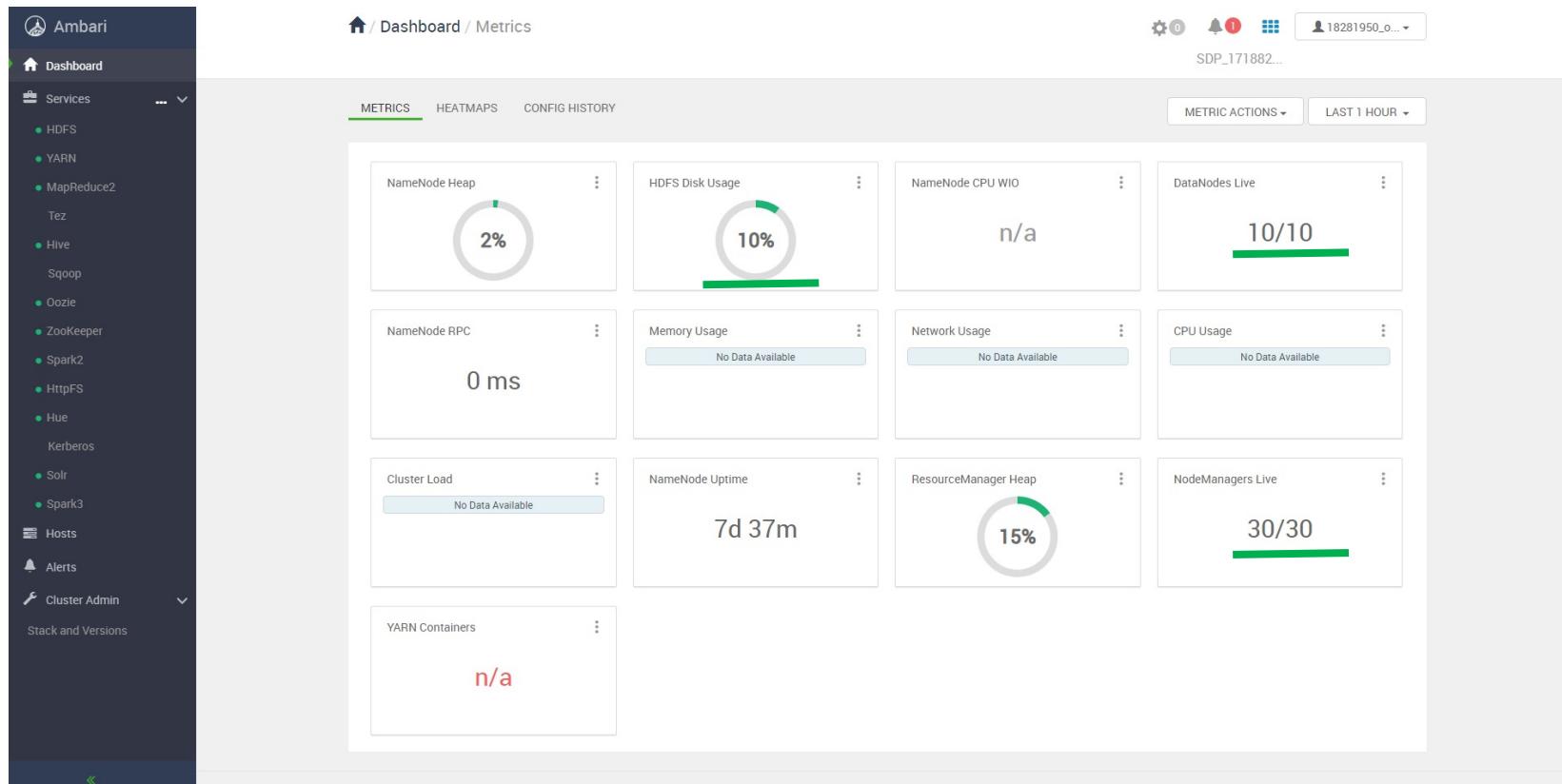
hadoop fs -mv hdfs://path1 hdfs://path2 – перенос path1 из одного hdfs в path2 другого hdfs

hadoop fs –get hdfs://path destination_path – скачать данные из hdfs в папку назначения

hadoop fs –put from_where hdfs://path – загрузить данные на hdfs из папки

Ambari

Ambari у нас используется для мониторинга ресурсов кластера. Тут многие термины, которые мы употребляли можно «пощупать».



Ambari - HDFS

Тут следим за дисками. Забьется – начнут отмирать ноды.

The screenshot shows the Ambari interface for monitoring the HDFS service. The left sidebar lists various services: Dashboard, Services (HDFS, YARN, MapReduce2, Tez, Hive, Sqoop, Oozie, ZooKeeper, Spark2, HttpFS, Hue, Kerberos, Solr, Spark3), Hosts, Alerts, Cluster Admin, and Stack and Versions. The main content area is titled 'Services / HDFS / Summary'. It features four tabs: SUMMARY (selected), HEATMAPS, CONFIGS, and METRICS. The SUMMARY tab displays a 'Summary' section with components and their status: STANDBY NAMENODE (Started), ZKFAILOVERCONTROLLER (Started), ACTIVE NAMENODE (Started), DATANODES (10/10 Started), JOURNALNODES (3/3 Live), NFSGATEWAYS (0/0 Started), and a DATANODES STATUS table. Below this is a 'Service Metrics' section with metrics for BLOCKS, TOTAL FILES + DIRECTORIES, and DISK USAGE (DFS USED). To the right is a 'Quick Links' panel listing links for NameNode UI, NameNode Logs, NameNode JMX, and Thread Stacks. A red arrow points from the text 'Тут можно смотреть графики' to the 'NameNode UI' link in the Quick Links panel.

Ambari

Dashboard

Services

- HDFS
- YARN
- MapReduce2
- Tez
- Hive
- Sqoop
- Oozie
- ZooKeeper
- Spark2
- HttpFS
- Hue
- Kerberos
- Solr
- Spark3

Hosts

Alerts

Cluster Admin

Stack and Versions

Ambari / Services / HDFS / Summary

SUMMARY HEATMAPS CONFIGS METRICS

Actions

SDP_171882...

Summary

Components	Started	Started	Started	Started
STANDBY NAMENODE	STANDBY NAMENODE	ZKFAILOVERCONTROLLER	ACTIVE NAMENODE	ZKFAILOVERCONTROLLER
7d 39m NAMENODE UPTIME	2.5% 415.2 MB / 16.0 GB NAMENODE HEAP	3/3 Live JOURNALNODES	0/0 Started NFSGATEWAYS	
10/10 Started DATANODES	DATANODES STATUS			
10 Live	0 Dead	0	0 Decommissioning	

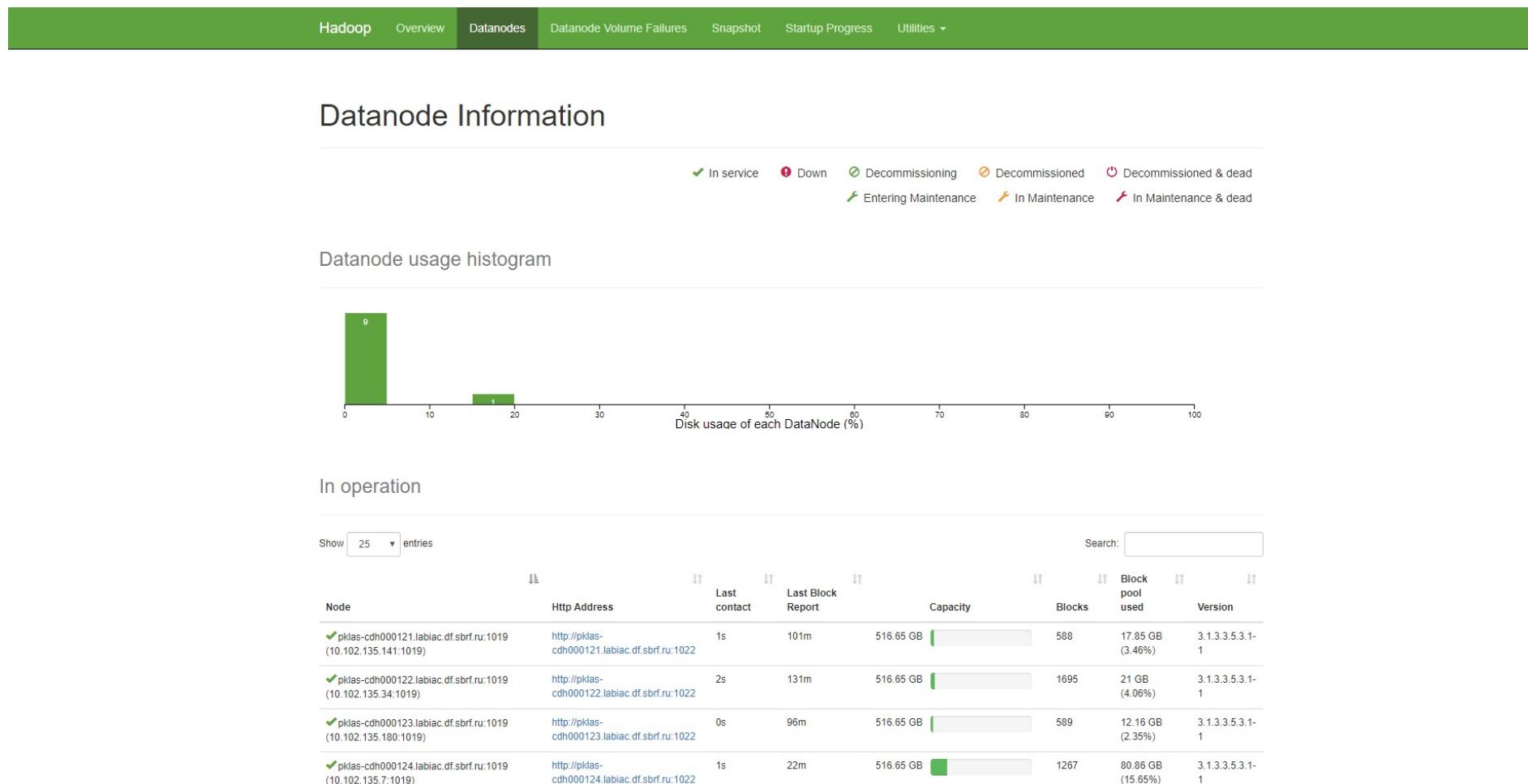
Service Metrics	BLOCKS			
BLOCKS	3716 Total	0 Corrupt Replica	0 Missing	0 Under Replicated
TOTAL FILES + DIRECTORIES	4557	No pending upgrade UPGRADE STATUS	Not in safe mode SAFE MODE STATUS	
DISK USAGE (DFS USED)	4.11% 212.6 GB / 5.0 TB	5.82% 300.5 GB / 5.0 TB	90.07% 4.5 TB / 5.0 TB	DISK REMAINING

Quick Links

- pklas-cdh000122.vm.prod.cloud.sbrf.ru (Standby)
NameNode UI
NameNode Logs
NameNode JMX
Thread Stacks
- pklas-cdh000123.vm.prod.cloud.sbrf.ru (Active)
NameNode UI (Active)
NameNode Logs
NameNode JMX
Thread Stacks

Тут можно смотреть графики

Ambari – HDFS (графики)



Ambari – HDFS (decommission)

Summary

Security is on.

Safemode is off.

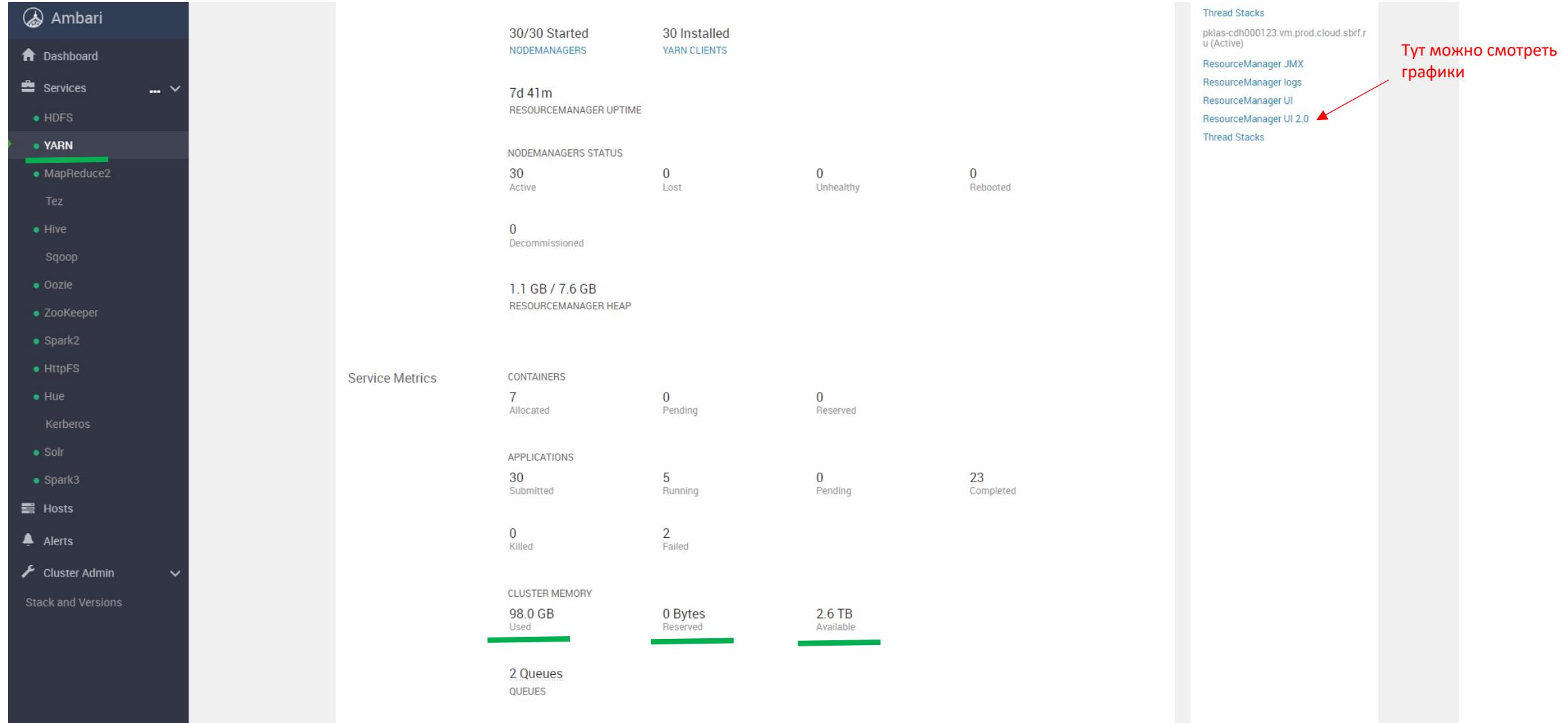
4 557 files and directories, 3 716 blocks (3 716 replicated blocks, 0 erasure coded block groups) = 8 273 total filesystem object(s).

Heap Memory used 303.92 MB of 15.96 GB Heap Memory. Max Heap Memory is 15.96 GB.

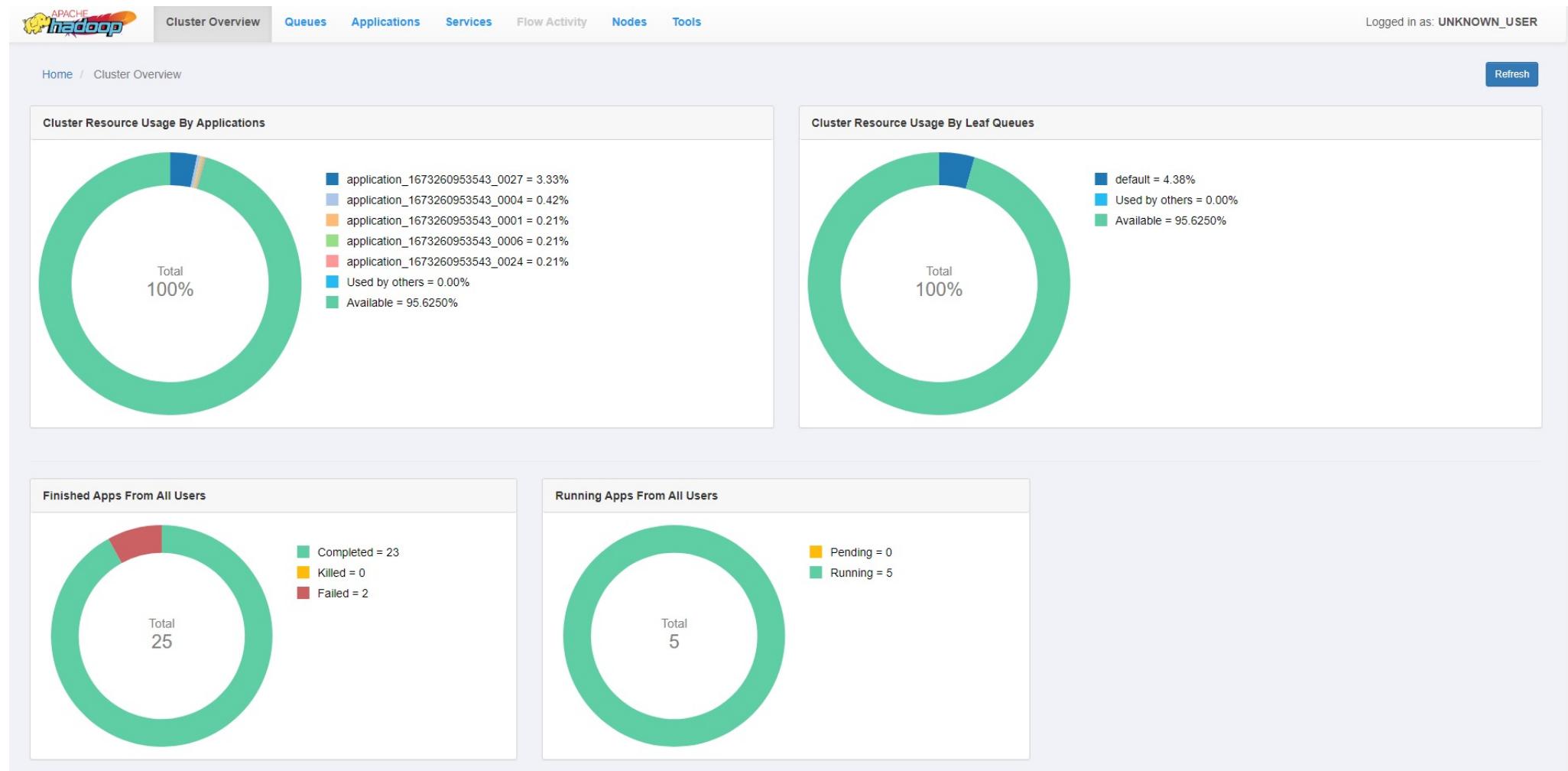
Non Heap Memory used 122.7 MB of 125.58 MB Committed Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	5.05 TB
Configured Remote Capacity:	0 B
DFS Used:	212.56 GB (4.11%)
Non DFS Used:	0 B
DFS Remaining:	4.54 TB (90.07%)
Block Pool Used:	212.56 GB (4.11%)
DataNodes usages% (Min/Median/Max/stdDev):	2.35% / 2.66% / 15.65% / 3.88%
Live Nodes	10 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion (including replicas)	0
Block Deletion Start Time	Mon Jan 09 14:33:31 +0300 2023
Last Checkpoint Time	Mon Jan 16 13:15:32 +0300 2023

Ambari - YARN



Ambari – YARN (ResourceManager)



Ambari – YARN (ResourceManager)

The screenshot shows the Apache Ambari interface for managing YARN applications. The top navigation bar includes links for Cluster Overview, Queues, Applications (selected), Services, Flow Activity, Nodes, and Tools. A user is logged in as UNKNOWN_USER. On the left, there are two filter panels: 'User (32)' and 'State (4)'. The 'User' panel shows filters for 'hive' (109), '18281950_omega-sbrf-ru' (49), '19282897_omega-sbrf-ru' (45), 'spark' (39), '20299882_omega-sbrf-ru' (37), and '19083153_omega-sbrf-ru' (30). The 'State' panel shows filters for FINISHED (466), FAILED (71), RUNNING (5), and KILLED (5). The main content area displays a table of running applications with columns: Application ID, Application Type, Application Name, User, State, Queue, Progress, Start Time, Elapsed Time, and Finished Time. The table lists five applications:

Application ID	Application Type	Application Name	User	State	Queue	Progress	Start Time	Elapsed Time	Finished Time
application_1673260953543_0027	SPARK	SimakovaVB	06832618_o...	● Running	default	<div style="width: 10%;">10%</div>	2023/01/16 10:1...	4h 19m 22s 231...	N/A
application_1673260953543_0024	TEZ	HIVE-57727a2f-...	hive	● Running	default	<div style="width: 0%;">0%</div>	2023/01/16 08:0...	6h 31m 11s 78ms	N/A
application_1673260953543_0006	TEZ	HIVE-99530a3a-...	hive	● Running	default	<div style="width: 0%;">0%</div>	2023/01/09 13:5...	7D 41m 56s 149...	N/A
application_1673260953543_0004	yarn-service	llap0	hive	● Running	default	<div style="width: 100%;">100%</div>	2023/01/09 13:4...	7D 42m 54s 17ms	N/A
application_1673260953543_0001	SPARK	Thrift JDBC/OD...	spark	● Running	default	<div style="width: 10%;">10%</div>	2023/01/09 13:4...	7D 49m 3s 644ms	N/A

Ambari – YARN (ApplicationMaster)

APACHE
hadoop

Cluster Overview Queues Applications Services Flow Activity Nodes Tools

Logged in as: UNKNOWN_USER

Home / Applications / App [application_1673260953543_0027] / Attempts

SimakovaVB UNDEFINED

application_1673260953543_0027

● Running

06832618_omega-sbrf-ru Started at 2023/01/16 10:12:55

Refresh

Attempts List Resource Usage Diagnostics Logs

Application Attempts

Graph View Grid View

2023/01/16 10:12:55 2023/01/16 14:33:33

attempt_27

appattempt_1673260953543_0027_000001

Application Attempt Id	appattempt_1673260953543_0027_000001
Started Time	2023/01/16 10:12:55
Elapsed Time	4 Hrs : 20 Mins : 38 Secs
AM Container Id	container_e27_1673260953543_0027_01_000001
AM Node Id	pklas-cdh000128.labiac.df.sbrf.ru:45454
AM Node Web UI	pklas-cdh000128.labiac.df.sbrf.ru:8042
Logs	Link

Ambari – ApplicationMaster - Jobs

Выполнить граф – Job, который бьется на Stages

The screenshot shows the Spark 3.0.1 ApplicationMaster - Jobs page. The top navigation bar includes links for Jobs, Stages, Storage, Environment, Executors, and SQL. The title is "Spark Jobs (?)" and the sub-page title is "Completed Jobs (107)". The page displays a table of completed jobs with columns: Job Id, Description, Submitted, Duration, Stages: Succeeded/Total, and Tasks (for all stages): Succeeded/Total. The table lists 107 rows of completed jobs, each showing a short description of the task (e.g., "showString at <unknown>:0"), the submission time, duration, stage status (all succeeded), and task completion status (all succeeded).

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
106	showString at <unknown>:0 showString at <unknown>:0	2023/01/16 13:43:54	0.1 s	1/1 (1 skipped)	75/75 (402 skipped)
105	showString at <unknown>:0 showString at <unknown>:0	2023/01/16 13:43:54	0.1 s	1/1 (1 skipped)	100/100 (402 skipped)
104	showString at <unknown>:0 showString at <unknown>:0	2023/01/16 13:43:53	0.4 s	1/1 (1 skipped)	20/20 (402 skipped)
103	showString at <unknown>:0 showString at <unknown>:0	2023/01/16 13:43:53	0.4 s	1/1 (1 skipped)	4/4 (402 skipped)
102	showString at <unknown>:0 showString at <unknown>:0	2023/01/16 13:42:10	1.7 min	2/2	403/403
101	showString at NativeMethodAccessorimpl.java:0 showString at NativeMethodAccessorimpl.java:0	2023/01/16 13:27:26	0.1 s	1/1 (1 skipped)	75/75 (1524 skipped)
100	showString at NativeMethodAccessorimpl.java:0 showString at NativeMethodAccessorimpl.java:0	2023/01/16 13:27:26	0.1 s	1/1 (1 skipped)	100/100 (1524 skipped)
99	showString at NativeMethodAccessorimpl.java:0 showString at NativeMethodAccessorimpl.java:0	2023/01/16 13:27:26	0.1 s	1/1 (1 skipped)	20/20 (1524 skipped)
98	showString at NativeMethodAccessorimpl.java:0 showString at NativeMethodAccessorimpl.java:0	2023/01/16 13:27:25	0.1 s	1/1 (1 skipped)	4/4 (1524 skipped)
97	showString at NativeMethodAccessorimpl.java:0 showString at NativeMethodAccessorimpl.java:0	2023/01/16 13:27:17	8 s	2/2	1525/1525
96	showString at NativeMethodAccessorimpl.java:0 showString at NativeMethodAccessorimpl.java:0	2023/01/16 13:27:04	0.1 s	1/1 (1 skipped)	75/75 (402 skipped)
95	showString at NativeMethodAccessorimpl.java:0 showString at NativeMethodAccessorimpl.java:0	2023/01/16 13:27:03	0.2 s	1/1 (1 skipped)	100/100 (402 skipped)
94	showString at NativeMethodAccessorimpl.java:0	2023/01/16 13:27:03	0.4 s	1/1 (1 skipped)	20/20 (402 skipped)

Ambari – ApplicationMaster - Stages

The screenshot shows the Apache Spark 3.0.1 ApplicationMaster - Stages UI. The top navigation bar includes links for Jobs, Stages (selected), Storage, Environment, Executors, and SQL. The main section displays 'Stages for All Jobs' with 117 completed stages. A table lists these stages with columns for Stage Id, Description, Submitted, Duration, Tasks: Succeeded/Total, Input, Output, Shuffle Read, and Shuffle Write. Red arrows point from the following column headers to specific data points in the table:

- Время выполнения (Duration) points to the duration of stage 147 (91 ms).
- Задачи с progress bar (Tasks: Succeeded/Total) points to the progress bar for stage 147, which shows 75/75.
- Сколько прочитали из HDFS (Input) points to the input size for stage 138 (2.3 GiB).
- Сколько записали в HDFS (Output) points to the output size for stage 128 (1575.9 MiB).
- Сколько прочитали из временного хранилища (Shuffle Read) points to the shuffle read size for stage 128 (249.8 KiB).
- Сколько записали во временное хранилище (Shuffle Write) points to the shuffle write size for stage 128 (249.8 KiB).

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
147	showString at <unknown>:0	+details 2023/01/16 13:43:54	91 ms	75/75			116.0 KiB	
145	showString at <unknown>:0	+details 2023/01/16 13:43:54	0.1 s	100/100			157.3 KiB	
143	showString at <unknown>:0	+details 2023/01/16 13:43:53	0.4 s	20/20			40.2 KiB	
141	showString at <unknown>:0	+details 2023/01/16 13:43:53	0.4 s	4/4			6.0 KiB	
139	showString at <unknown>:0	+details 2023/01/16 13:43:53	0.2 s	1/1				
138	showString at <unknown>:0	+details 2023/01/16 13:42:10	1.7 min	402/402	2.3 GiB			319.6 KiB
137	showString at NativeMethodAccessorImpl.java:0	+details 2023/01/16 13:27:26	0.1 s	75/75			91.2 KiB	
135	showString at NativeMethodAccessorImpl.java:0	+details 2023/01/16 13:27:26	0.1 s	100/100			122.6 KiB	
133	showString at NativeMethodAccessorImpl.java:0	+details 2023/01/16 13:27:26	0.1 s	20/20			31.6 KiB	
131	showString at NativeMethodAccessorImpl.java:0	+details 2023/01/16 13:27:25	0.1 s	4/4			4.4 KiB	
129	showString at NativeMethodAccessorImpl.java:0	+details 2023/01/16 13:27:25	31 ms	1/1				
128	showString at NativeMethodAccessorImpl.java:0	+details 2023/01/16 13:27:17	8 s	1524/1524	1575.9 MiB			249.8 KiB
127	showString at NativeMethodAccessorImpl.java:0	+details 2023/01/16 13:27:04	0.1 s	75/75			91.2 KiB	
125	showString at NativeMethodAccessorImpl.java:0	+details 2023/01/16 13:27:03	0.2 s	100/100			122.6 KiB	
123	showString at NativeMethodAccessorImpl.java:0	+details 2023/01/16 13:27:03	0.4 s	20/20			31.6 KiB	
121	showString at NativeMethodAccessorImpl.java:0	+details 2023/01/16 13:27:01	2 s	4/4			4.4 KiB	
119	showString at NativeMethodAccessorImpl.java:0	+details 2023/01/16 13:27:00	0.4 s	1/1				
118	showString at NativeMethodAccessorImpl.java:0	+details 2023/01/16 13:26:09	51 s	402/402	1531.9 MiB			249.8 KiB

Ambari – ApplicationMaster - Stages

Можно смотреть DAG каждого Stage (обобщенный) и различные метрики

SIMAKOVAVB application UI

Apache Spark 3.0.1

Jobs Stages Storage Environment Executors SQL

Details for Stage 135 (Attempt 0)

Total Time Across All Tasks: 2 s
Locality Level Summary: Process local: 100
Shuffle Read Size / Records: 122.6 KiB / 1331
Associated Job Ids: 100

DAG Visualization

```
graph TD; A[ShuffledRowRDD [1811]] --> B[WholeStageCodegen (33)]; B --> C[mapPartitionsInternal]
```

Show Additional Metrics Event Timeline

Summary Metrics for 100 Completed Tasks

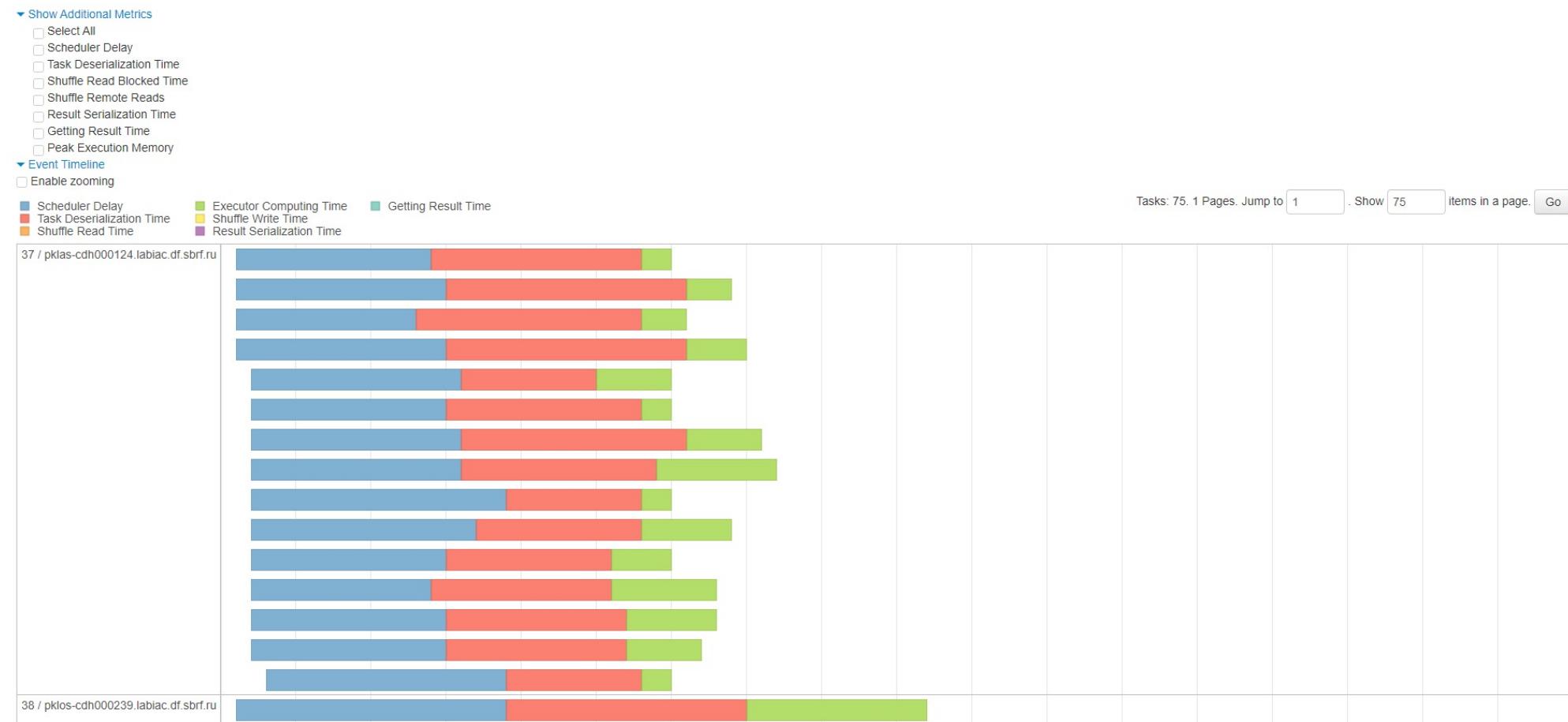
Metric	Min	25th percentile	Median	75th percentile	Max
Duration	2.0 ms	5.0 ms	12.0 ms	25.0 ms	83.0 ms
GC Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Shuffle Read Size / Records	0.0 B / 0	0.0 B / 0	0.0 B / 0	3.1 KiB / 30	6.2 KiB / 116

Showing 1 to 3 of 3 entries

Aggregated Metrics by Executor

Ambari – ApplicationMaster - Stages

Дополнительные возможности для анализа



Ambari – ApplicationMaster - Executors

Тут можно сделать любимое всеми дело – посмотреть логи ошибок executors.

The screenshot shows the Apache Spark 3.0.1 Executors page in the Ambari UI. The top navigation bar includes links for Jobs, Stages, Storage, Environment, Executors (which is selected), and SQL. A search bar at the top right contains the text "SimakovaVB application U".

Executors

[Show Additional Metrics](#)

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(2)	0	0.0 B / 50.8 GiB	0.0 B	15	0	0	72	72	5.0 min (6 s)	40.8 MiB	0.0 B	11.8 KiB	0
Dead(40)	0	26.9 MiB / 1017 GiB	0.0 B	600	0	0	9698	9698	3.3 h (9.6 min)	26.1 GiB	1.4 MiB	2.7 MiB	0
Total(42)	0	26.9 MiB / 1 TiB	0.0 B	615	0	0	9770	9770	3.4 h (9.7 min)	26.2 GiB	1.4 MiB	2.7 MiB	0

Executors

Show 20 entries Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	pklas-cdh000124.labiac.df.sbrf.ru:43926	Active	0	0.0 B / 25.4 GiB	0.0 B	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump	
1	pklas-cdh000240.labiac.df.sbrf.ru:38227	Dead	0	2.5 MiB / 25.4 GiB	0.0 B	15	0	0	1828	1828	33 min (1.7 min)	5.4 GiB	22.1 KiB	22.1 KiB	stdout	stderr Thread Dump
2	pklas-cdh000122.labiac.df.sbrf.ru:34749	Dead	0	1.4 MiB / 25.4 GiB	0.0 B	15	0	0	219	219	9.7 min (20 s)	805.6 MiB	0.0 B	0.0 B	stdout	stderr Thread Dump
3	pklas-cdh000242.labiac.df.sbrf.ru:40764	Dead	0	1.2 MiB / 25.4 GiB	0.0 B	15	0	0	244	244	16 min (41 s)	2.1 GiB	0.0 B	0.0 B	stdout	stderr Thread Dump
4	pklas-cdh000244.labiac.df.sbrf.ru:39952	Dead	0	1.3 MiB / 25.4 GiB	0.0 B	15	0	0	278	278	5.9 min (11 s)	850.3 MiB	0.0 B	0.0 B	stdout	stderr Thread Dump
5	pklas-cdh000123.labiac.df.sbrf.ru:35181	Dead	0	1.3 MiB / 25.4 GiB	0.0 B	15	0	0	228	228	11 min (22 s)	819.4 MiB	0.0 B	0.0 B	stdout	stderr Thread Dump
6	pklas-cdh000126.labiac.df.sbrf.ru:33537	Dead	0	1.4 MiB / 25.4 GiB	0.0 B	15	0	0	1672	1672	20 min (31 s)	3.4 GiB	56.4 KiB	319.5 KiB	stdout	stderr Thread Dump
7	pklas-cdh000230.labiac.df.sbrf.ru:44207	Dead	0	0.0 B / 25.4 GiB	0.0 B	15	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	stdout	stderr Thread Dump	

Ambari – ApplicationMaster - Executors

- Storage – хранилище, где отображаются объекты, к которым применяли persist или cache;
- Environment – тут можно посмотреть конфигурацию вашего контекста;
- SQL – тут все про executor'ы Spark SQL, физические и логические планы запросов.

Ambari – Hosts

На вкладке Hosts можно посмотреть на все ноды кластера, их компоненты и вообще их состояние.

The screenshot shows the Ambari interface with the 'Hosts' tab selected. The main area displays a table of nodes, each with a green checkmark icon and a truncated name. A red arrow points from the text 'Ноды' (Nodes) to the 'Name' column header. Another red arrow points from the text 'Что установлено на ноде' (What is installed on the node) to the 'Components' column header. The table includes columns for Name, IP Address, Rack, Cores, RAM, Disk Usage, Load Avg, Versions, and Components. The 'Components' column shows values like '17 Components', '29 Components', '22 Components', etc. The top right corner of the table shows the user ID '18281950_o...'. Below the table, there are pagination controls: 'Items per page: 10' and '1 - 10 of 30'.

Name	IP Address	Rack	Cores	RAM	Disk Usage	Load Avg	Versions	Components
pklas-cdh000121.labiac...	10.102.135.141	/default-rack	16 (16)	125.75GB			SDP-3.5.3.1	17 Components
pklas-cdh000122.labiac...	10.102.135.34	/default-rack	16 (16)	125.75GB			SDP-3.5.3.1	29 Components
pklas-cdh000123.labiac...	10.102.135.180	/default-rack	16 (16)	125.75GB			SDP-3.5.3.1	22 Components
pklas-cdh000124.labiac...	10.102.135.7	/default-rack	16 (16)	125.75GB			SDP-3.5.3.1	13 Components
pklas-cdh000125.labiac...	10.102.135.31	/default-rack	16 (16)	125.75GB			SDP-3.5.3.1	13 Components
pklas-cdh000126.labiac...	10.102.135.121	/default-rack	16 (16)	125.75GB			SDP-3.5.3.1	13 Components
pklas-cdh000127.labiac...	10.102.135.32	/default-rack	16 (16)	125.75GB			SDP-3.5.3.1	13 Components
pklas-cdh000128.labiac...	10.102.135.186	/default-rack	16 (16)	125.75GB			SDP-3.5.3.1	13 Components
pklas-cdh000129.labiac...	10.102.135.101	/default-rack	16 (16)	125.75GB			SDP-3.5.3.1	13 Components
pklas-cdh000130.labiac...	10.102.135.44	/default-rack	16 (16)	125.75GB			SDP-3.5.3.1	13 Components

Ambari – Hosts (Components)

Есть несколько нод с большим количеством компонент, большая часть одинаковые. Где-то DataNode, где-то NameNode, где развернут HUE.

[/ Hosts / pklas-cdh000121.labiac.df.sbrf.ru](#) / Summary

SUMMARY CONFIGS ALERTS 0 VERSIONS

Components

Status	Name	Type	Action
green	HttpFS gateway / HttpFS	Master	...
green	Spark3 History Server / Spark3	Master	...
green	ZooKeeper Server / ZooKeeper	Master	...
green	DataNode / HDFS	Slave	...
green	JournalNode / HDFS	Slave	...
green	NodeManager / YARN	Slave	...
green	HDFS Client / HDFS	Client	...
green	Hive Client / Hive	Client	...
green	Kerberos Client / Kerberos	Client	...
green	MapReduce2 Client / MapReduce2	Client	...
green	Oozie Client / Oozie	Client	...
green	Spark2 Client / Spark2	Client	...
green	Spark3 Client / Spark3	Client	...
green	Sqoop Client / Sqoop	Client	...
green	Tez Client / Tez	Client	...
green	YARN Client / YARN	Client	...
green	ZooKeeper Client / ZooKeeper	Client	...

[/ Hosts / pklas-cdh000122.labiac.df.sbrf.ru](#) / Summary

SUMMARY CONFIGS ALERTS 0 VERSIONS

Components

Status	Name	Type	Action
green	Timeline Service V1.5 / YARN	Master	...
green	History Server / MapReduce2	Master	...
green	Hive Metastore / Hive	Master	...
green	HiveServer2 Interactive / Hive	Master	...
green	HiveServer2 / Hive	Master	...
green	Standby NameNode / HDFS - SDP-17188221-omega-sbrf-ru-uim-rrmc-cluster-10-a78084	Master	...
green	Oozie Server / Oozie	Master	...
green	Standby ResourceManager / YARN	Master	...
green	Spark2 History Server / Spark2	Master	...
green	Timeline Service V2.0 Reader / YARN	Master	...
green	YARN Registry DNS / YARN	Master	...
green	ZooKeeper Server / ZooKeeper	Master	...
green	DataNode / HDFS	Slave	...
green	JournalNode / HDFS	Slave	...
green	Livy for Spark2 Server / Spark2	Slave	...
green	NodeManager / YARN	Slave	...
green	Spark2 Thrift Server / Spark2	Slave	...

HOST ACTIONS ▾

SDP_171882...

LAST 1 HOUR ▾

Host Metrics

NAMESPACE: SDP-17188221-omega-sbrf-ru-uim-rrmc-cluster-10-a78084

NameNode Heap	NameNode CPU WIO
 2%	n/a
NameNode RPC	NameNode Uptime
0 ms	7d 3h 52m

Waggle Dance – зачем он?

Как мы знаем, данные хранятся на разных кластерах в облаке и нужен инструмент, позволяющий работать с данными на разных кластерах так, как будто эти данные лежат на вашем кластере.

Облако нам нужно для того, чтобы в рамках него мы могли создавать отдельные области (кластера) для работы, а не создавать один гигантский кластер.

Однако, Hive разработан на основе архитектуры монолитного кластера и не предоставляет средств для обеспечения межкластерного доступа к данным. Один кластер Hive не может одновременно получать доступ к локальным наборам данных и наборам данных в другом кластере. Это приводит к созданию множества изолированных хранилищ данных в облаке.

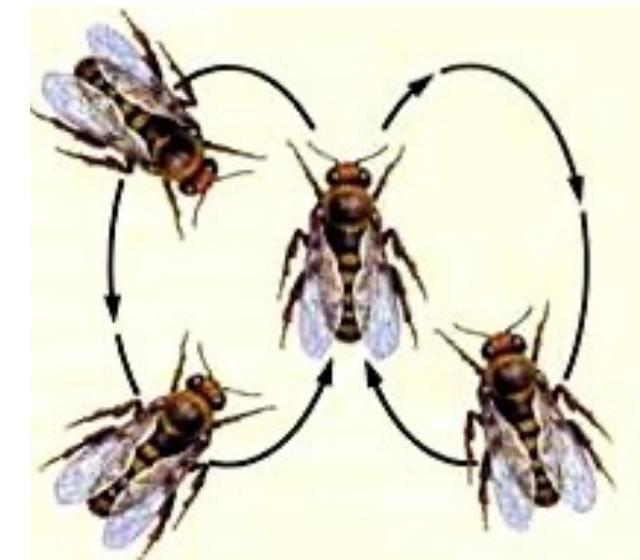
Waggle Dance – как работает?

Создатели вдохновились тем, как реализованы удаленные таблицы в некоторых RDBMS базах.

Waggle Dance предоставляет объединенное представление этих разрозненно расположенных наборов данных, позволяя пользователям в одном кластере исследовать данные на других кластерах и получать к ним доступ.

Waggle Dance работает как прокси-служба маршрутизации запросов через Hive metastore Thrift API. Waggle Dance подключается к нескольким сервисам метахранения, расположенным в других несвязанных кластерах. Он направляет и преобразует запросы метаданных в соответствующие хранилища, используя идентификаторы базы данных, закодированные в коде.

Waggle Dance возвращает ответы из разных хранилищ, эти ответы обычно содержат пути к файлам данных, которые затем используются системой для получения данных.



Waggle Dance - недостатки

- Инфраструктура
 - нужны сервера;
 - нужна память;
 - нужна балансировка нагрузки при перекрестных запросах.
- Ограничения Hive metastore Thrift API
 - waggle dance не синхронизирован с Hive.
- Управление федерациями
 - работает отдельно от Hive metastore, поэтому сам по себе не может управлять источниками, добавление новых источников требует перезапуска системы.
- Перегрузка имен источников и таблиц
 - разные источники могут иметь одинаковые названия сущностей, что решается префиксами или на уровне администраторов баз данных.
- В нашей структуре помним про нагрузку на сеть, а также учимся пользоваться vim.