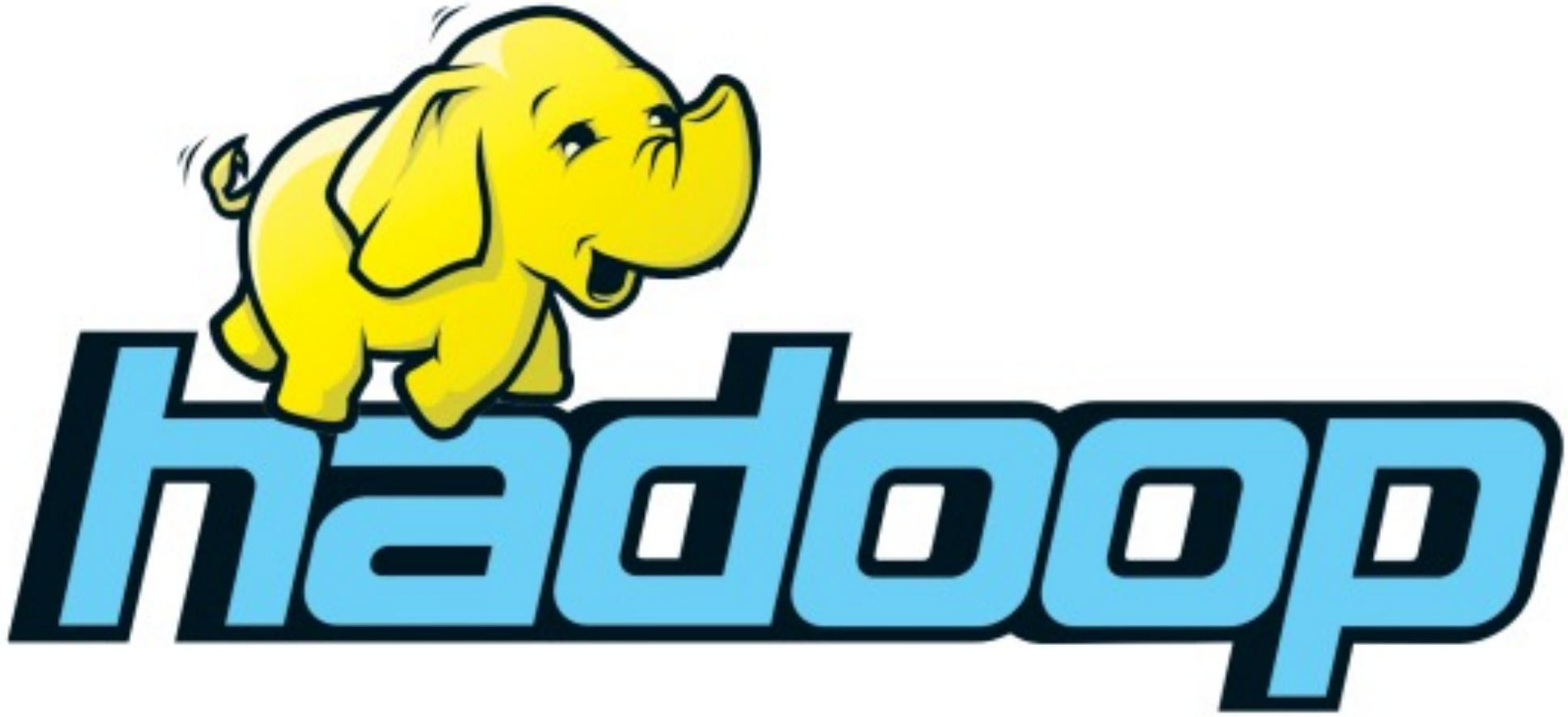
Обзор Hadoop

О чем поговорим?

1) Экосистема Hadoop
2) HDFS
3) YARN
4) MapReduce
5) Некоторые команды в hadoop

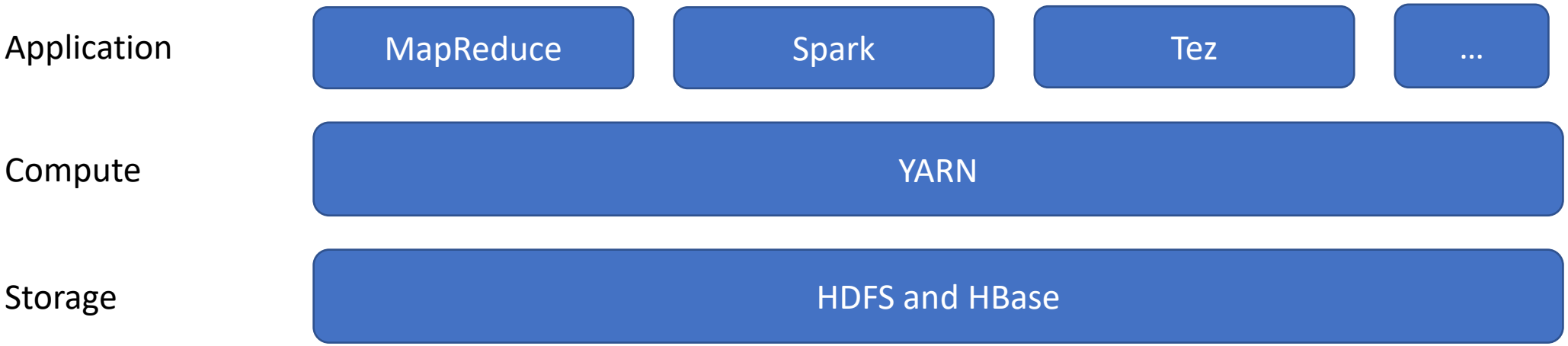


Экосистема Hadoop

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.

RDBMS 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 и 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.

HDFS

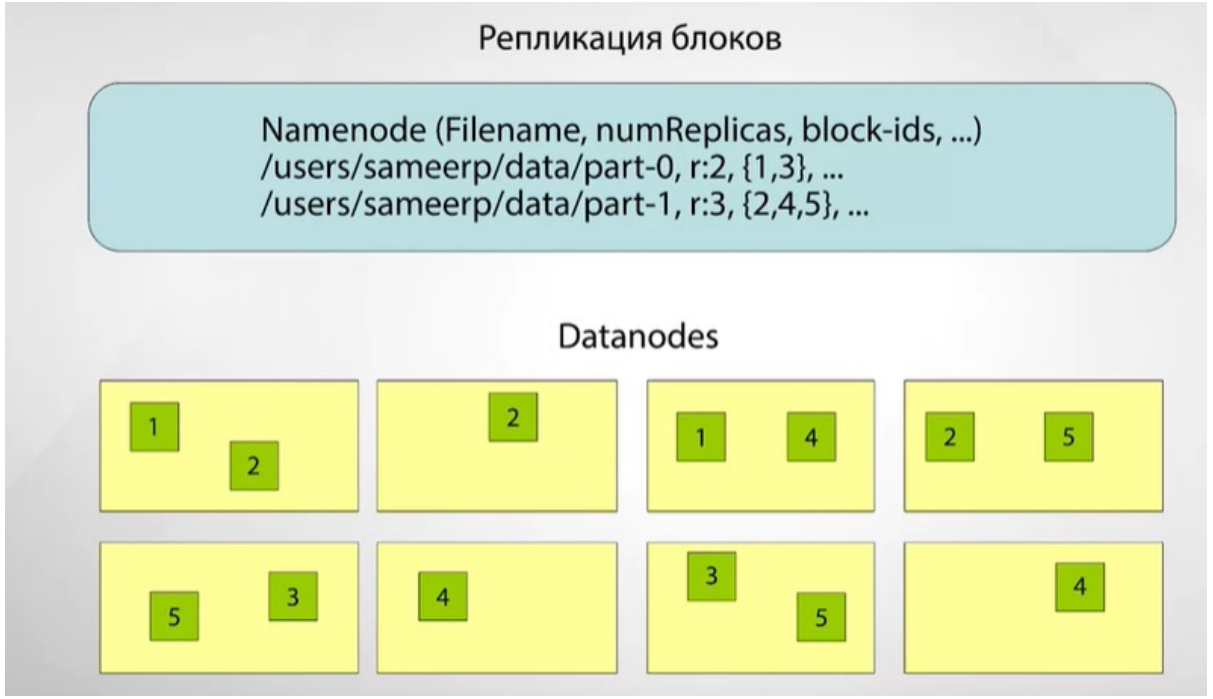
HDFS – Hadoop Distributed File System

- 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
- Соотвествие имя файла –> блоки хранится в памяти специальной машины (Name Node)

Репликация блоков



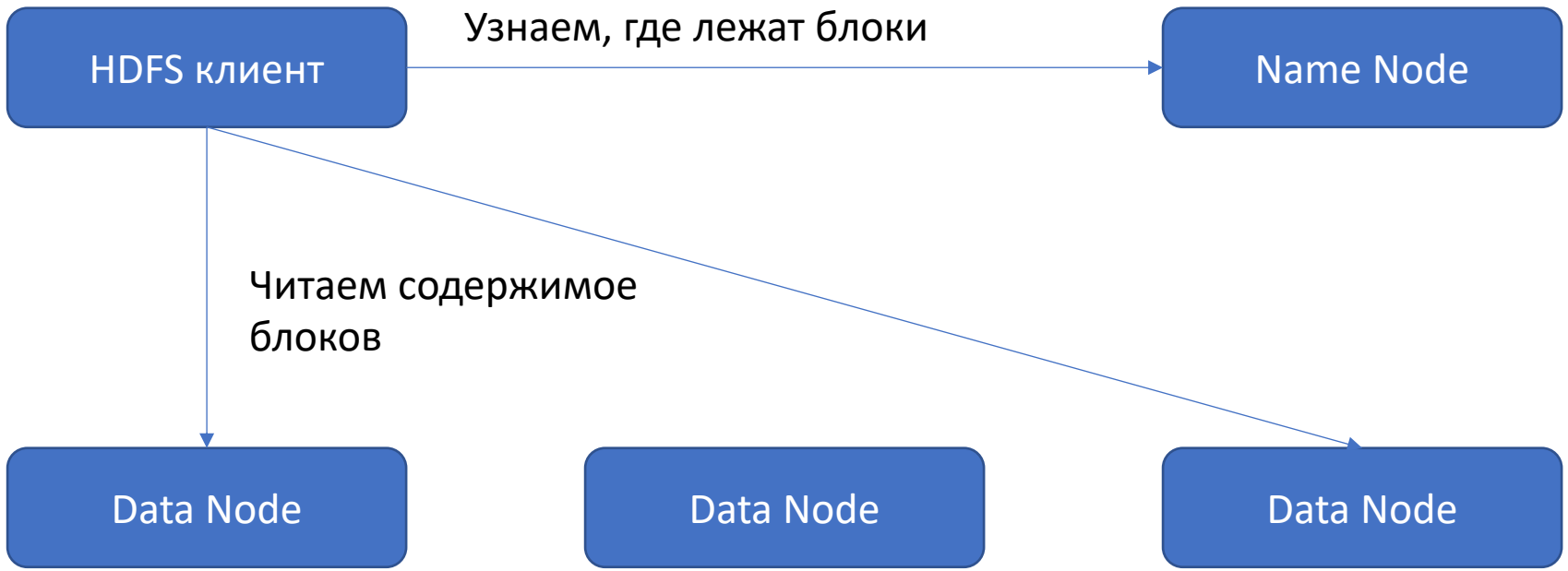
Зачем вообще репликация

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

Чтение из HDFS

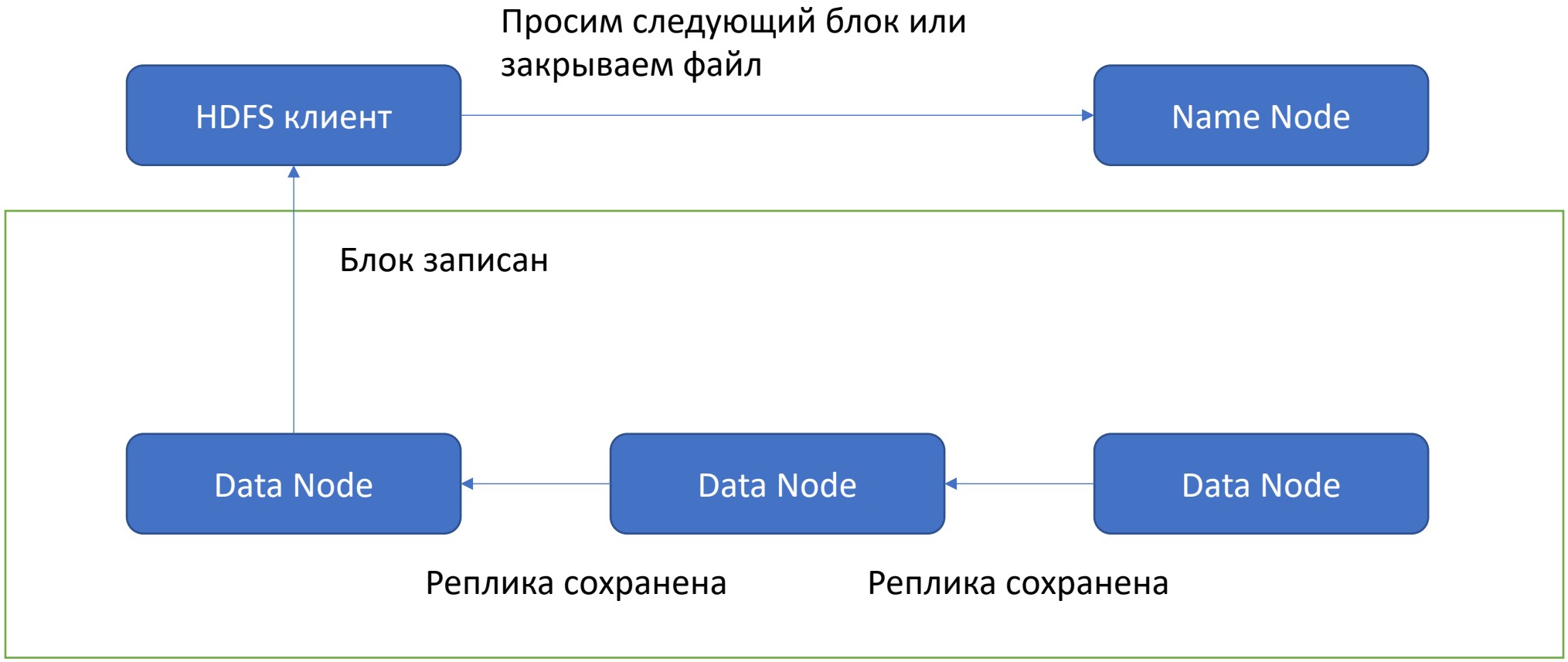


Запись в HDFS

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.

Запись в HDFS

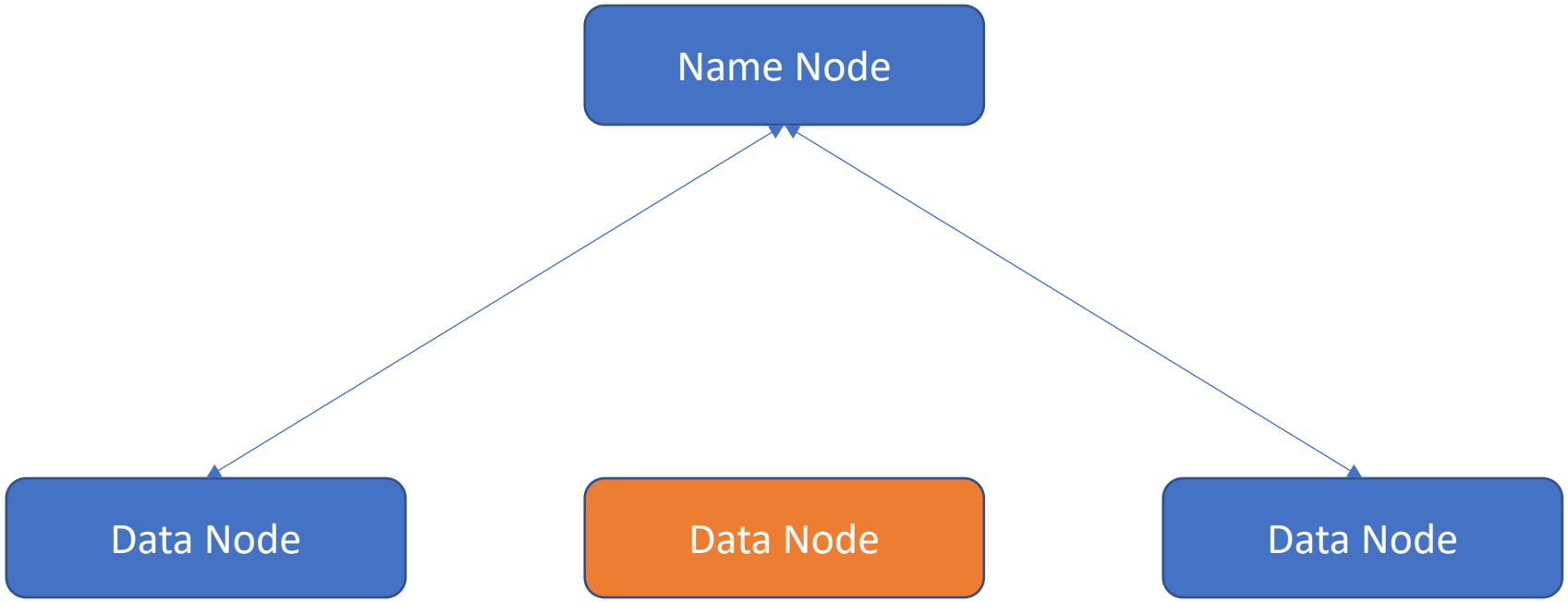


Дожидаемся ответа асинхронно, можем сразу начинать писать следующий блок

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.

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.

YARN

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.

А что 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 мониторингов 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

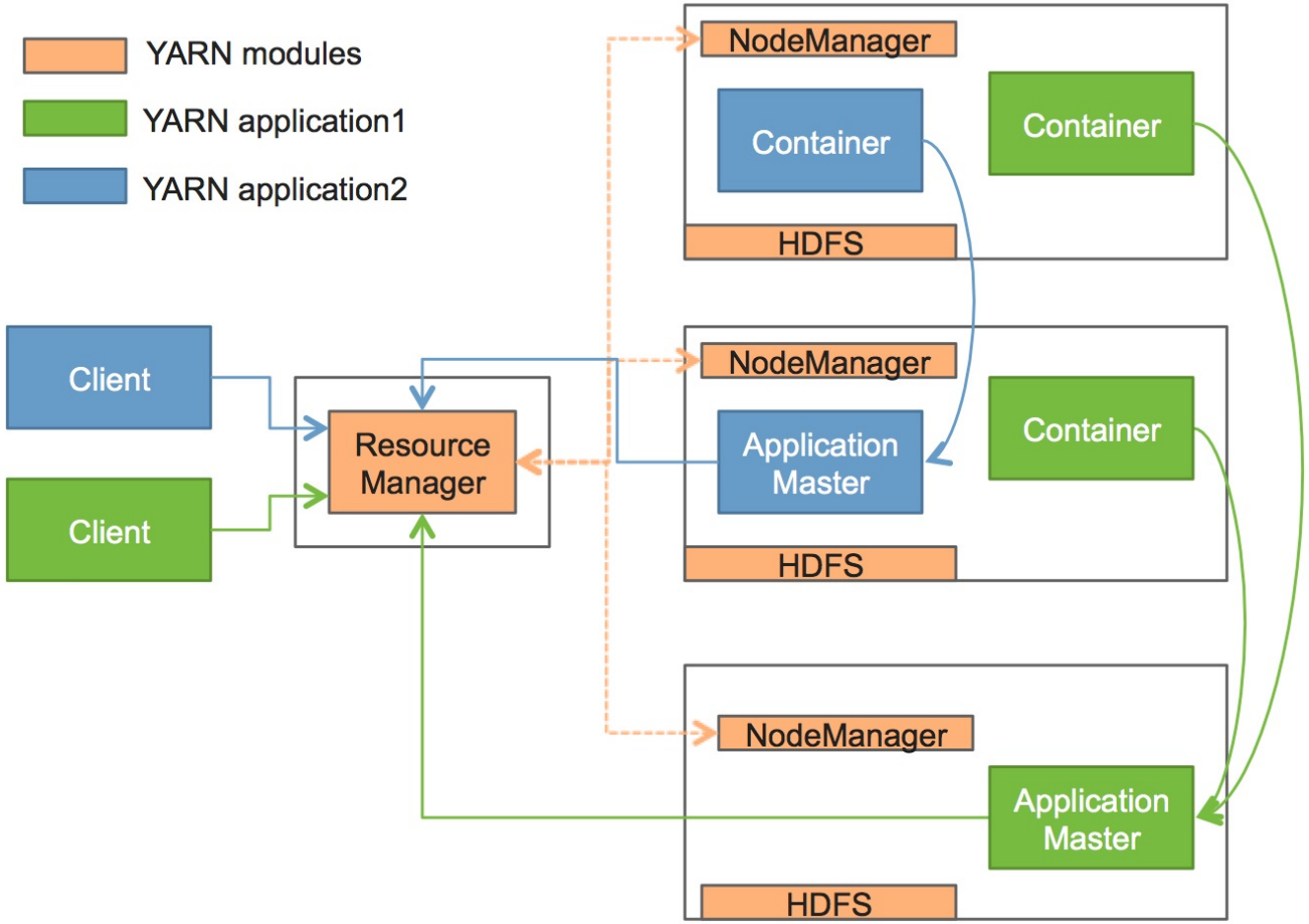
Это набор физических ресурсов, таких CPU, RAM и прочее.

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 AM 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 AM .

Принцип работы YARN



Источник: https://ru.bmstu.wiki/YARN_(Yet_Another_Resource_Negotiator)

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.

Как обобщить масштабируемость?

- Мы хотим разделить задачу на N частей;
- Делить будем по значению хэша слова hash(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 $hash(x) = x[0] + x[1]p^1 + .. + x[n]p^n$

x – строка
p - фиксированное простое число
x[i] - код символа

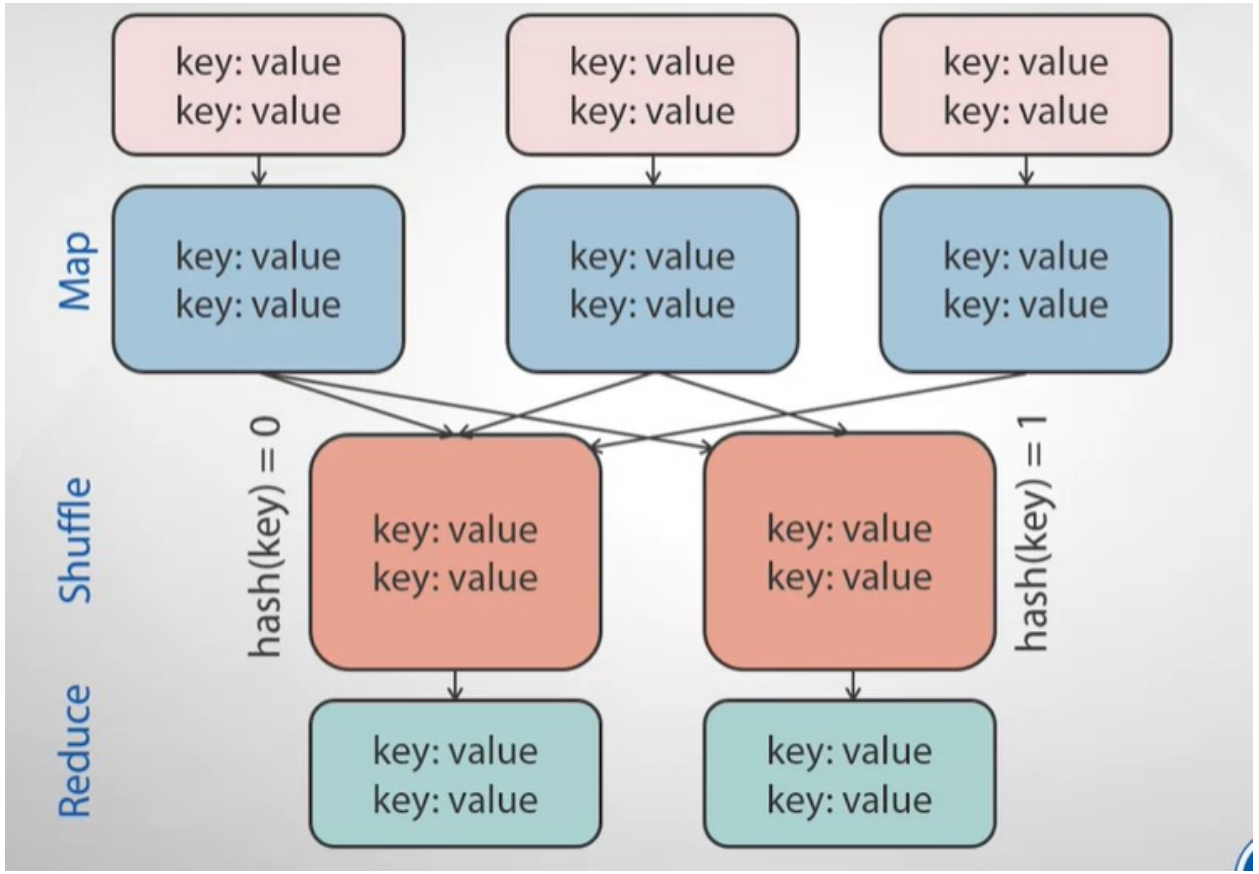
А вот и наш MapReduce

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) -> List(K2, V2)
(номер строки, ”кот спит”) -> [(“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)) -> List(K3, V3)
((“ кот”), (1, 1, 1))-> [(‘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.

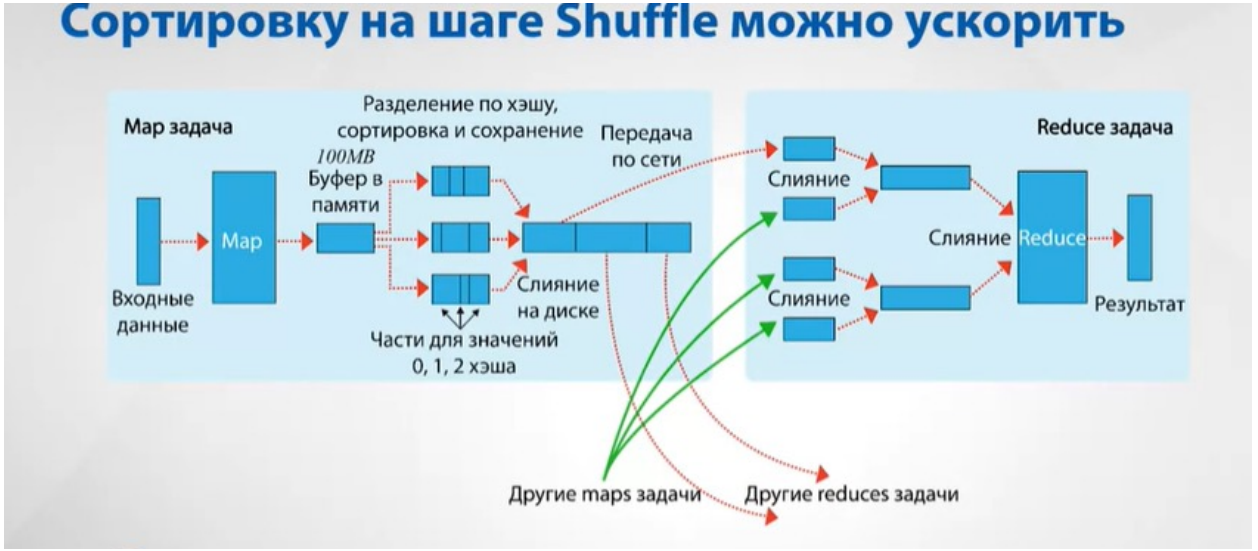
Можно сортировать внутри мапера, делая это кусками по 100 Мб в оперативке.

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:

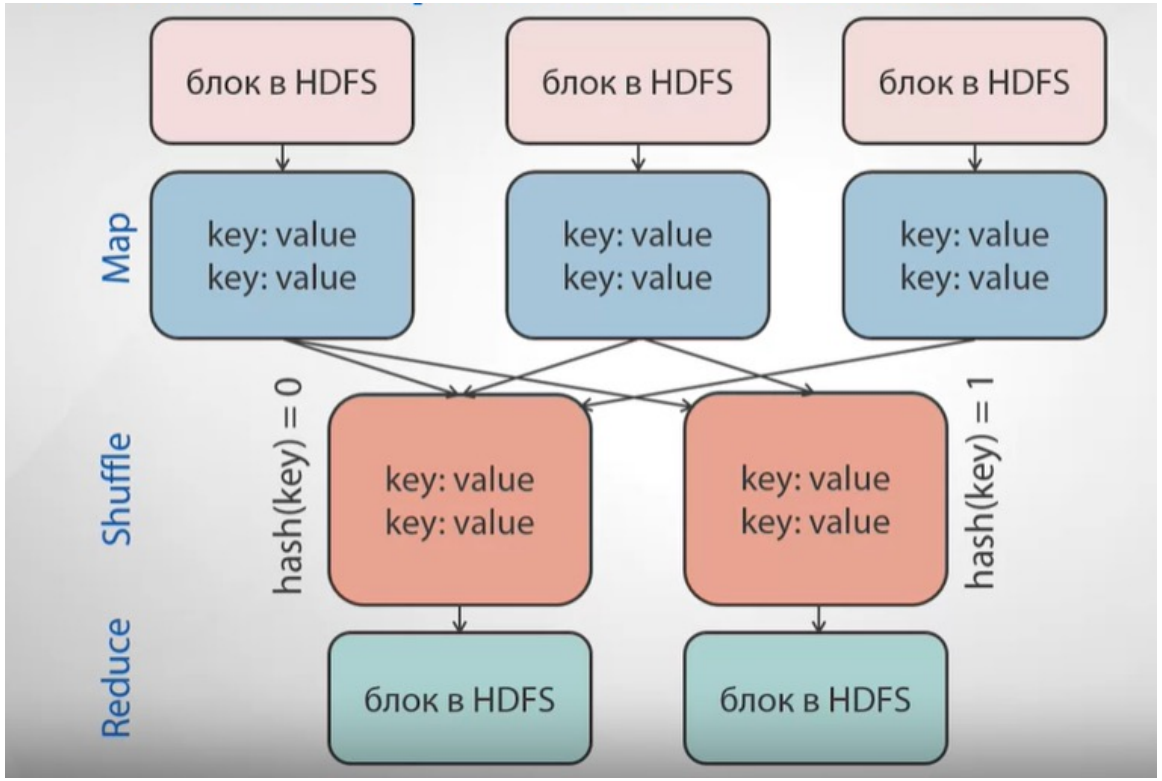
Hadoop MapReduce

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



Предположим, но потом же все надо собрать и записать???

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 если что-то 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