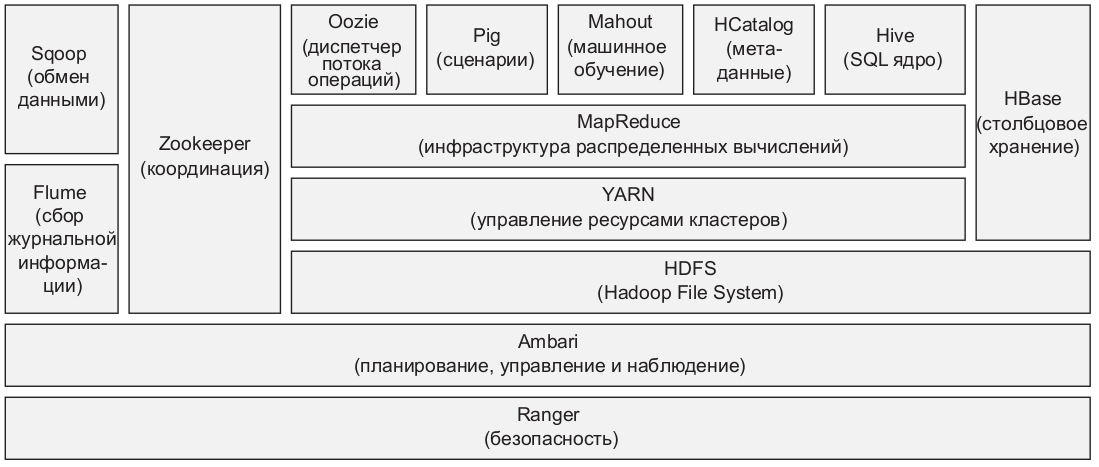
В Hadoop центральное место занимают:

*  Распределенная файловая система (HDFS).
*  Метод крупномасштабного выполнения программ (MapReduce).
*  Система управления ресурсами кластеров (YARN).

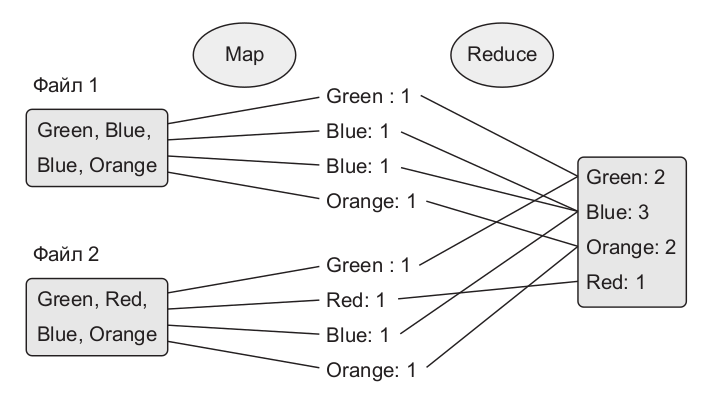
На этой базе сформировалась экосистема приложений, например базы данных Hive и HBase, и инфраструктуры машинного обучения (такие, как Mahout).

Для реализации параллелизма Hadoop использует технологию программирования,



называемую MapReduce. Алгоритм MapReduce выполняет разбивку данных, обрабатывает их параллельно, а затем сортирует, объединяет и обобщает результаты. Однако алгоритм MapReduce плохо подходит для интерактивного анализа или итеративных программ, потому что данные записываются на диск между этапами вычислений. При работе с большими наборами данных запись обходится достаточно дорого.

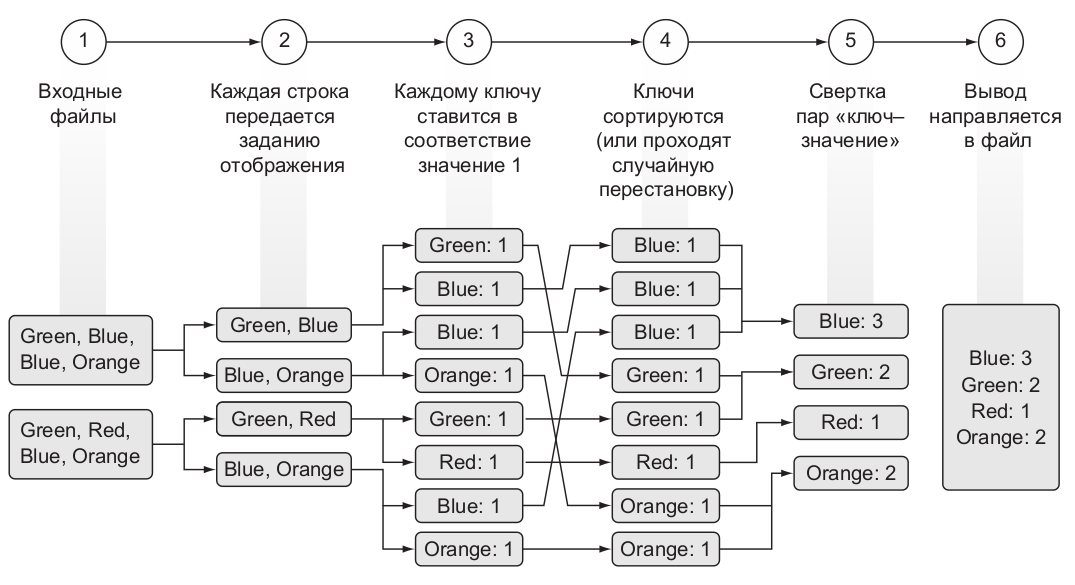
Рассмотрим работу MapReduce на небольшом вымышленном примере. Вы — директор компании, занимающейся производством игрушек. Каждая игрушка окрашена в два цвета; когда клиент заказывает игрушку с веб-страницы, файл заказа размещается в Hadoop с указанием цветов. Ваша задача — определить, сколько емкостей с краской нужно подготовить. Для подсчета цветов будет использоваться алгоритм в стиле MapReduce.



Как следует из названия, процесс делится приблизительно на две крупные фазы:

*  Фаза отображения (Map) — документы делятся на пары «ключ—значение». До выполнения свертки в данных будет много дубликатов.
*  Фаза свертки (Reduce) — в целом аналогична конструкции SQL GROUP BY . Разные уникальные экземпляры собираются воедино, и в зависимости от функции свертки могут быть получены разные результаты. В данном случае нужно получить количество вхождений каждого цвета, поэтому именно оно возвращается функцией свертки

Весь процесс описан в следующих шести шагах.



1. Чтение входных файлов.
2. Каждая строка передается заданию отображения (mapper job).
3. Задание читает из файла цвета (ключи) и выводит для каждого цвета файл с количеством его вхождений (значение). Или, в терминологии программирования, ключ (цвет) отображается на значение (количество вхождений).
4. Ключи сортируются, чтобы упростить обобщение данных.
5. Фаза свертки суммирует количество вхождений каждого цвета и выводит для каждого ключа один файл с общим количеством вхождений этого цвета.
6. Ключи собираются в выходной файл.

**Spark** — инфраструктура кластерных вычислений, сходная с MapReduce. Однако Spark не занимается ни хранением файлов в (распределенной) файловой системе, ни управлением ресурсами. Для этого Spark пользуется услугами таких систем, как Hadoop File System, YARN или Apache Mesos. Таким образом, Hadoop и Spark являются взаимодополняющими системами. В процессе тестирования и разработки вы даже можете запускать Spark в своей локальной системе.

**Как Spark решает проблемы MapReduce?**

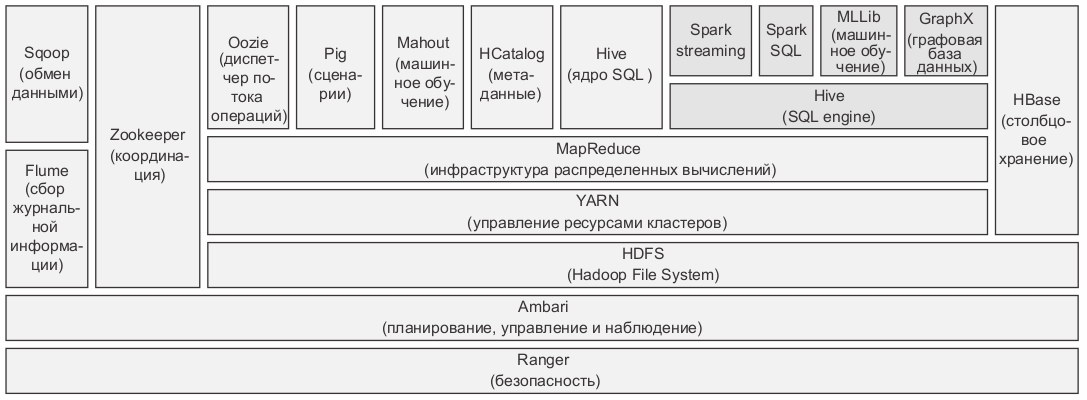
Spark создает своего рода общую оперативную память для компьютеров вашего кластера (хотя мы немного упрощаем ситуацию для наглядности.) Это позволяет разным рабочим процессам использовать общие переменные (и их состояние), а следовательно, снимает необходимость в записи промежуточных результатов на диск. Если же вы предпочитаете более научное и формальное объяснение, Spark использует RDD (Resilient Distributed Datasets) — распределенную абстракцию памяти, которая позволяет программистам выполнять вычисления в памяти в больших кластерах с защитой от ошибок. Так как система работает в памяти, она избегает затратных дисковых операций.

**Компоненты экосистемы Spark**

Ядро Spark предоставляет среду NoSQL, хорошо подходящую для интерактивного, исследовательского анализа. Spark может работать как в пакетном, так и в интерактивном режиме и поддерживает Python.

Spark содержит еще четыре крупных компонента,

1. Spark Streaming — инструмент анализа в реальном времени.
2. Spark SQL предоставляет интерфейс SQL для работы с Spark.
3. MLLib — инструмент машинного обучения в инфраструктуре Spark.
4. GraphX — графовая база данных для Spark.

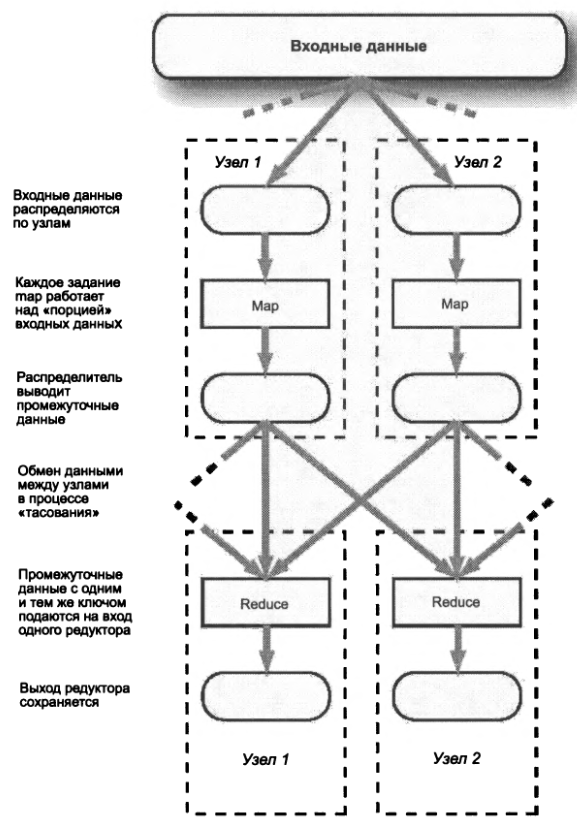


**Архитектура MapReduce**

MapReduce-программа обрабатывает данные посредством манипулирования парами ключ/значение. В общем виде это выглядит так:

* map:(Kl,Vl)->list(K2,V2)
* reduce: (K2,list(V2)) -> list(K3,V3)

Конечно, такое представление потока данных слишком общее.



**Типы данных в Hadoop**

В каркасе MapReduce определен некий способ сериализации пар ключ/значение для передачи их через границы машин в кластере, и только классы, поддерживающие эту сериализацию, могут выступать в роли типов ключей и значений.

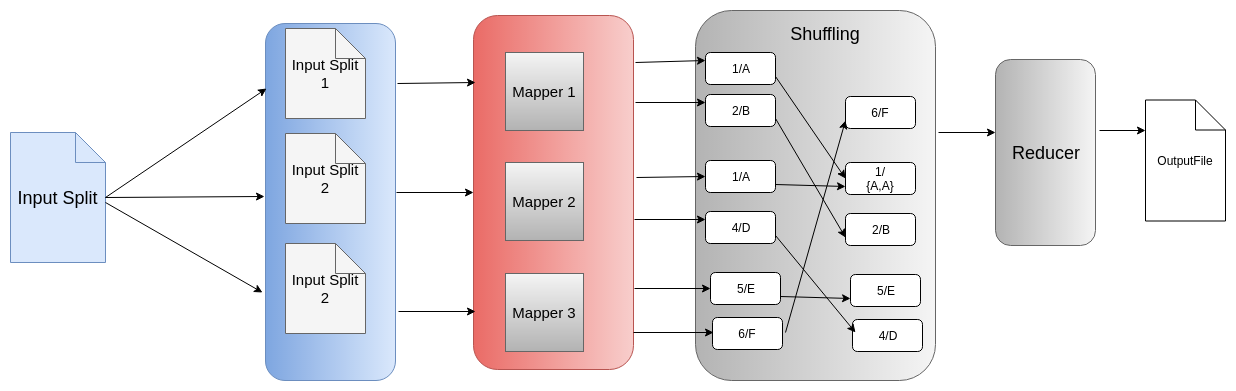
Точнее говоря, классы, реализующие интерфейс Writable, могут быть типами значений, а классы, реализующие интерфейс WritableComparable<T>, - типами как ключей, так и значений. Интерфейс WritabieComparabie<T> представляет собой комбинацию интерфейсов Writable и java.lang.Comparable<T>. Требование сравнимости ключей выдвигается потому, что на этапе редукции они сортируются, тогда как значения просто передаются напрямую.

В состав Hadoop входит ряд готовых классов, реализующих интерфейс WritabieComparabie, в том числе классы-обертки для всех простых типов данных.

Наиболее употребительные типы ключей и значений. Все эти классы реализуют интерфейс WritabieComparabie

|  |  |
| --- | --- |
| Класс | Описание |
| BooleanWritable | Обертка стандартного типа Boolean |
| ByteWritable | Обертка одного байта |
| DoubleWritable | Обертка типа Double |
| FloatWritable | Обертка типа Float |
| IntWritable | Обертка типа Integer |
| LongWritable | Обертка типа Long |
| Text | Обертка для хранения текста в кодировке UTF8 |
| NullWritable | Заглушка для случая, когда ключ или значение не нужны |

**Процесс обработки данных**

**InputFileFormat**: процесс MapReduce начинается с чтения файла, хранящегося в HDFS. Эти файлы могут быть любого определенного типа, например, «Text», «Avro» и т.д. Обработка файла контролируется InputFormat. Есть несколько реализаций для InputFormat.

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;

}

**RecordReader и input split**: входной файл делится на куски, и эти куски называются input split. Разделение ввода - это не что иное, как отдельные фрагменты файла, а размер контролируется параметрами mapred.max.split.size и mapred.min.split.size. По умолчанию размер входного разбиения совпадает с размером блока, и вы не должны изменять его, если это не требуется для конкретного случая. Для неразделяемых форматов файлов, таких как .gzip, разделение ввода будет равно размеру одного файла .gzip, что означает, что если будет 12 файлов .gzip, то будет 12 входных разделений, и для каждого входного разделения будет запущен один Mapper для его обработки.

Функция RecordReader отвечает за чтение данных из input split, хранящихся в HDFS. Формат входного файла по умолчанию - TextInputFileFormat, а разделитель RecordReader - / n, что означает, что RecordReader будет рассматривать одну строку как одну запись. Помните, что вы всегда можете настроить поведение RecordReader, передав реализацию собственного RecordReader. RecordReader знает, как читать записи из input split. По умолчанию RecordReader считывает запись с разделителем новой строки для TextInputFileFormat. Однако вы можете изменить поведение RecordReader, передав свою собственную реализацию. RecordReader читает запись и передает ее мапперу.

**Mapper**: класс Mapper отвечает за обработку входного разбиения. Функция RecordReader считывает запись из входного разбиения и передает каждую запись в функцию сопоставления mapper. Mapper содержит метод map, который принимает входные данные из RecordReader и обрабатывает запись. Функция map выполняется для каждой записи, что означает, что если один input split содержит 100 записей, то функция map будет выполнена 100 раз.

Mapper также содержит методы настройки (setup) и очистки (cleanup). Метод настройки выполняется до того, как mapper начнет обработку записей для разделения ввода, поэтому любая операция инициализации, такая как чтение из распределенного кэша и инициализация соединения, должна выполняться внутри метода установки. Метод очистки выполняется после того, как все записи в input split были обработаны, и поэтому любая операция очистки должна выполняться внутри этого метода.

Mapper обрабатывает запись и выдает выходные данные, используя **объект контекста**. Объект контекста позволяет Mapper и Reducer взаимодействовать с другими системами Hadoop, такими как предоставление конфигурации для mappers и reducers, запись mappers и reducers в файл и т.д. Он также обеспечивает связь между Mapper, Combiner и Reducer.

**Partitioner:** работа Partitioner состоит в том, чтобы назначить номер раздела записи, генерируемой mapper, чтобы записи с одним и тем же ключом всегда получали один и тот же номер раздела, что гарантирует, что эти записи всегда будут идти к одному и тому же редуктору. Для каждой записи существует определенный индекс раздела, и значение индекса раздела вычисляется внутри. mapper Context.write (). Общая формула индекса раздела: partitionIndex = (key.hashCode () & Integer.MAX\_VALUE)% numReducers

**Перестановка и сортировка**. Процесс переноса данных из Mapper в Reducer известен как перестановка. Редуктор запускает потоки для чтения данных с машины Mapper и считывает все принадлежащие им разделы для обработки с использованием протокола HTTP. Разные mappers могут иметь запись для одного и того же, и, таким образом, редуктор сортирует записи по ключу. Фазы перемешивания и сортировки происходят параллельно, что означает, что во время выборки выходов они объединяются, так что редуктор получает несколько записей для одного и того же ключа в списке в качестве значения.

**Reducer**: Количество редукторов, которые может запустить среда Hadoop, зависит от количества выводов Mapper и других параметров, но мы также можем контролировать количество редукторов, которые могут быть запущены. Формула для расчета количества редукторов выглядит следующим образом:

1,75 \* № of nodes \* mapred.tasktracker.reduce.tasks.maximum.

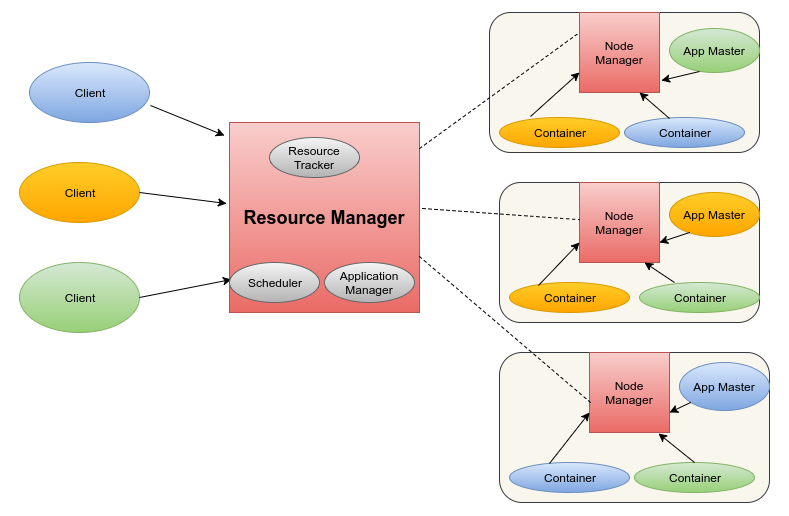
Редуктор содержит Reduce (), который выполняется для каждого уникального ключа, генерируемого преобразователями

**Combiner**: Combiner, также известный как мини-редуктор или локальный редуктор, работает на машине Mapper. Он берет промежуточный ключ, выпускаемый Mapper, и применяет пользовательскую функцию Reduce на той же машине. Для каждого Mapper на машине будет доступен один Combiner. Combiner значительно уменьшает объем перетасовок данных от Mapper к Reducer и, таким образом, помогает повысить производительность. Наличие Combiner между тем, не гарантирует, что они будут выполнены.

**Выходной формат**: выходной формат транслирует выходные данные Reducer пары ключ/значение , а программа записи записывает их в файл в HDFS. По умолчанию ключевые значения выходных данных разделены символом табуляции, а записи разделены символом новой строки. Выходной формат преобразует окончательную пару ключ / значение из функции reduce и записывает ее в файл. По умолчанию record writer разделяет ключ и значение символом табуляции, а отдельные записи - символом новой строки. Поведение по умолчанию можно изменить, внедрив собственный формат вывода.

**YARN and MapReduce**

**Архитектура YARN**

**Диспетчер ресурсов**. Диспетчер ресурсов - это главный демон, который отвечает за управление ресурсами представленных приложений. Он имеет два основных компонента:

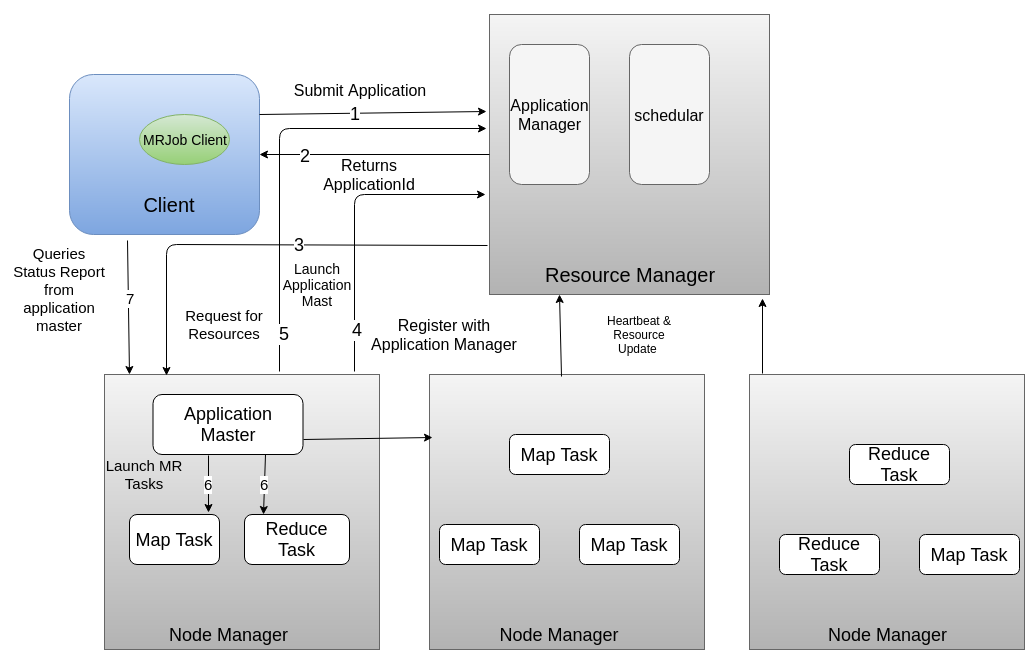
* **Планировщик**. Задачей планировщика Resource Manager является выделение необходимых ресурсов, запрашиваемых мастером приложения для каждого приложения. Задача планировщика заключается в том, чтобы только планировать задание, что означает, что он не отслеживает никакие задачи и не несет ответственности за повторный запуск любого отказавшего контейнера приложения. Приложение отправляет запрос на планирование работы в YARN, а YARN отправляет подробную информацию о планировании, включая объем памяти, необходимый для работы. Получив запрос на планирование, планировщик просто планирует задание.
* **Диспетчер приложений**. Работа диспетчера приложений заключается в управлении каждым мастером приложения. Каждое приложение, представленное в YARN, будет иметь собственную мастер-версию приложения, а диспетчер приложений отслеживает каждую мастер-заявку. Каждый клиентский запрос на отправку задания принимается диспетчером приложений, и он предоставляет ресурсы для запуска мастера приложения для приложения. Он также уничтожает мастер приложения по завершении выполнения приложения. Когда ресурсы кластера становятся ограниченными, диспетчер ресурсов может запросить ресурсы у работающего приложения, чтобы он мог выделить их приложению.

**Диспетчер узлов.** Диспетчер узлов - это ведомое устройство, которое работает на каждом рабочем узле кластера и отвечает за запуск и выполнение контейнеров на основе инструкций от диспетчера ресурсов. Диспетчер узлов отправляет сигналы heartbeat в диспетчер ресурсов, которые также содержат некоторую другую информацию, включая сведения о машине диспетчера узлов и доступную память. Диспетчер ресурсов регулярно обновляет информацию о каждом диспетчере узлов после получения запроса, что помогает планировать и планировать предстоящие задачи. Контейнеры запускаются в Node Manager, а мастер приложения также запускается в контейнере Node Manager.

**Мастер приложения**. Первым шагом приложения является отправка задания в YARN, и после получения запроса на передачу задания менеджер ресурсов YARN запускает мастер приложения для этого конкретного задания в одном из контейнеров Node Manager. Мастер приложений затем отвечает за управление выполнением приложений в кластере. Для каждого приложения будет выделенный мастер приложения, работающий на некотором контейнере Node Manager, который отвечает за координацию между Resource Manager и Node Manager для выполнения приложения. Мастер приложения запрашивает необходимые ресурсы для выполнения приложения у менеджера ресурсов, а менеджер ресурсов отправляет подробную информацию о контейнере ресурсов мастеру приложения, который затем координирует свои действия с соответствующим менеджером узлов, чтобы запустить контейнер для выполнения задачи приложения. Мастер приложения регулярно отправляет heartbeat импульсы в Resource Manager и обновляет использование своего ресурса. Мастер приложения меняет план выполнения в зависимости от ответа, полученного от менеджера ресурсов.

**Выполнение задач MR в YARN**

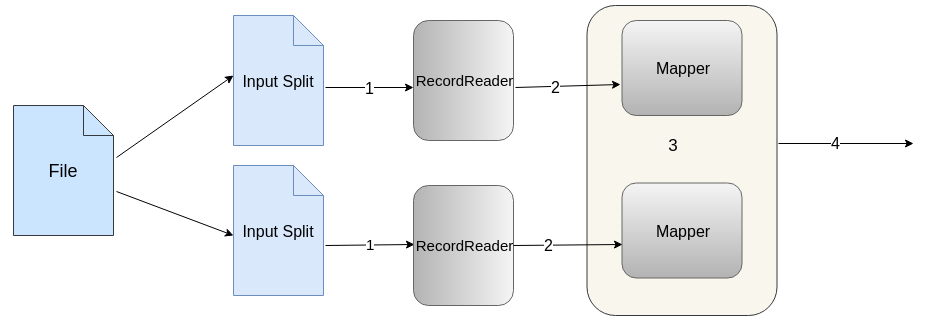
Диспетчер узлов (node manager) в YARN имеет достаточно памяти для запуска нескольких контейнеров. **Application Master** может запросить любое количество контейнеров у менеджера ресурсов, который отслеживает доступные ресурсы в кластере YARN. Тип задания не ограничивается MapReduce; вместо этого YARN может запускать приложения любого типа.



* Клиент задания MapReduce запрашивает новый идентификатор приложения у менеджера приложений. Менеджер ресурсов отправляет уникальный идентификатор приложения клиенту после проверки подлинности и авторизации клиента. Клиент задания MR помещает информацию метаданных о приложении в ApplicationSubmissionContext, который также содержит информацию для запуска мастера приложения.
* Менеджер ресурсов запускает мастер-приложение на одном из менеджеров узлов, который удовлетворяет этому требованию контейнера для мастер-приложения. Планировщик диспетчера ресурсов выбирает диспетчер узлов, в котором будет запущен мастер приложения.
* Мастер приложения создает объект клиента для связи с менеджером ресурсов и менеджером узлов. Мастер приложения регистрируется в диспетчере ресурсов, а последний отвечает такой информацией, как токены доступа, список ACL и т. д.
* Клиент задания MR запрашивает у менеджера приложений информацию о мастере приложения, а затем может напрямую обращаться к мастеру приложения для получения информации о состоянии, счетчике и любой другой информации. Мастер приложения вычисляет количество разбиений и отправляет запрос ресурса для мапперов и редукторов в планировщик менеджера ресурсов. Запрос содержит информацию о памяти и процессорной информации, которая требуется для контейнера.
* Мастер приложения получает контейнер для задачи map и reduce, а затем связывается с определенными менеджерами узлов для запуска контейнеров. Диспетчер узлов в YARN может запускать несколько контейнеров в одном диспетчере узлов.
* Мастер приложения также управляет и контролирует отдельную задачу сна, сокращает задачу и запрашивает дополнительный контейнер у менеджера ресурсов, если это необходимо. Это также гарантирует, что если какая-либо задача не выполнена или не отвечает, она может перезапуститься с новыми ресурсами, пока не будет достигнуто максимальное повторной попытки.
* Мастер приложения запускает операцию очистки задачи после завершения всех задач map и reduce задач. Наконец, мастер приложения отправляет запрос менеджеру ресурсов, завершает выполнение и освобождает (удаляет) занятый контейнер.

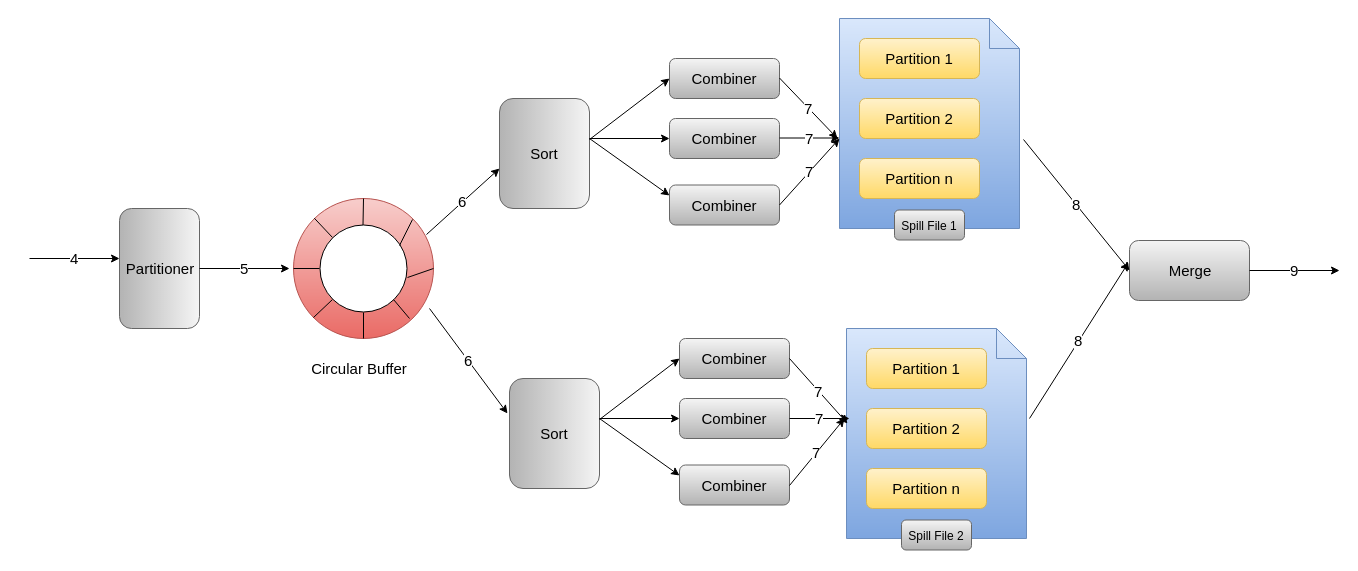
**MR workflow в Hadoop**

Выполнение задач MapReduce проходит через различные этапы, и у каждого этапа есть область для небольшой оптимизации. Давайте кратко рассмотрим поток выполнения MapReduce, который поможет нам понять, как каждый компонент взаимодействует друг с другом. Следующая диаграмма дает краткий обзор потока выполнения MapReduce.

1. InputFormat является отправной точкой любого приложения MapReduce. Он определяется в конфигурации задания в классе Driver приложения, например, job.setInputFormatClass (TextInputFormat.class). InputFormat помогает понять тип ввода и как его читать. Возвращает входные данные split и reader, которые помогают в чтении записей из файла. Размер входного разбиения снова зависит от InputFormat, например, размер разделения TextInputFormat будет равен размеру блока hdfs, который определяется свойством dfs.blocksize. Для нерасщепляемого формата файла, такого как .gzip, размер входного разделения будет равен одному размеру файла, что означает, что если будет 10 файлов gzip, то будет 10 входных разделений.

2. Для каждого входного разбиения будет запущен один Mapper. Это означает, что если getsplit () возвращает пять разбиений, то будет запущено пять задач Map для обработки разбиений. Функция Map, определенная в классе Mapper, будет выполняться для каждой пары ключ-значение входных разбиений, возвращаемых RecordReader. Реализация RecordReader зависит от InputFileFormat и для TextInputFormat, каждая новая строка рассматривается как новая запись.

3. Mapper обрабатывает запись. Для каждой записи мы можем выбрать один или несколько выходных данных и записать их в объект контекста, предоставленный в функции map. Класс, отвечающий за сбор выходных данных Map, определяется с помощью свойства mapreduce.job.map.output.collector.class, а реализацией по умолчанию является класс org.apache.Hadoop.mapred.MapTask.MapOutputBuffer. Следующая диаграмма является продолжением предыдущей диаграммы:

4. Выходные данные, генерируемые функцией map, поступают в класс раздела, который имеет метод getPartition (). Метод вычисляет номер раздела на основе определенного алгоритма и по умолчанию использует HashPartitioner.

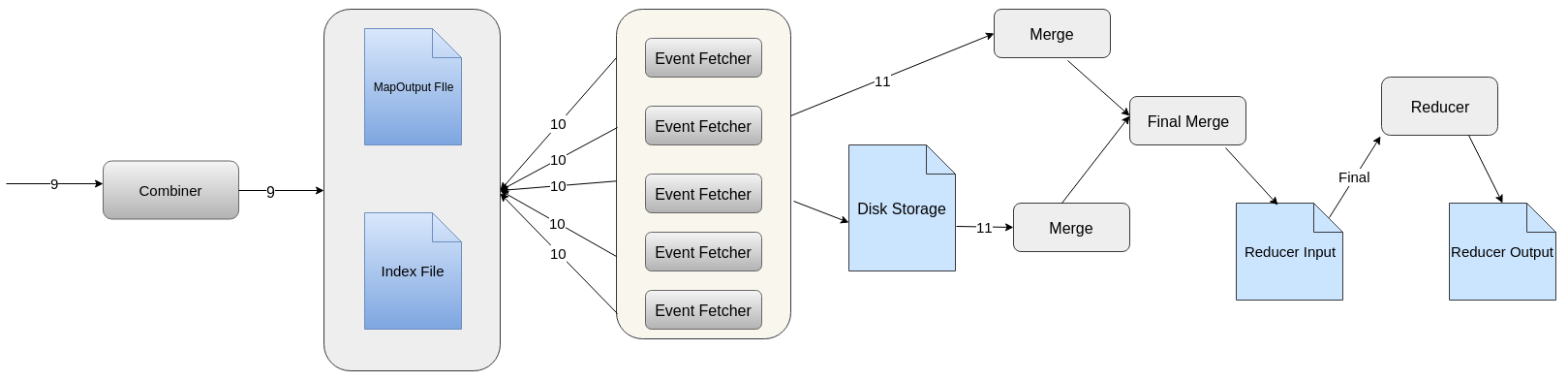
5. Вывод с разделом теперь записывается в кольцевой буфер в памяти. По умолчанию размер кольцевого буфера составляет 100 МБ. Его можно изменить, присвоив новое значение свойству mapreduce.task.io.sort.mb. Если размер выходных данных превышает, данные, хранящиеся в кольцевом буфере, попадут на диск.

6. Перед тем, как данные попадают на диск, они сортируются по ключу раздела, а внутри каждого раздела - по ключам записи. После сортировки combiner обрабатывает записи, чтобы уменьшить объем данных, записываемых на диск. Combiner не гарантирует выполнения, и если размер записи, выдаваемой Mapper, больше, чем размер буфера, то фазы Sort и Combiner будут пропущены, и данные будут записываться прямо на диск.

7. Для каждой операции spill на локальном диске Mapper создается новый файл. Расположение каталога можно настроить с помощью свойства mapreduce.job.local.dir.

8. После обработки всех записей и выполнения spill задач файлы spill объединяются в единый выходной файл Map. Эта фаза называется фазой слияния (merge). Процесс слияния может обрабатывать до 100 файлов по умолчанию. Этот параметр можно изменить, изменив значение mapreduce.task.io.sort.factor.

Если количество файлов превышает указанное значение, шаг слияния будет рекурсивно объединять файлы до тех пор, пока все файлы не будут объединены в один файл. Если общее количество файлов spill больше, чем min.num.spills.for.combine, Combiner будет выполнен для окончательного результата слияния. Файл MapOutput вместе с индексным файлом записываются на локальный диск mapper. Индексный файл содержит такую ​​информацию, как количество разделов, начало, конечные точки раздела и т.д.

9. Следующий шаг - перетасовка данных из Mapper в Reducer. Редуктор запускает поток извлечения событий, задача которого состоит в том, чтобы опрашивать мастер приложения для определения состояния Mappers. Получив состояние завершения выполнения преобразователя, он передает информацию Mapper в другой поток, который начинает копирование данных из Mapper с использованием протокола HTTP. По умолчанию количество потоков считывателя равно пяти, и его можно изменить, назначив новое значение mapred.reduce.parallel.copies. Данные, скопированные потоками считывателя, сохраняются в памяти, а размер памяти по умолчанию составляет 1 ГБ. Память может быть увеличена или уменьшена путем изменения mapreduce.reduce.memory.totalbytes. Если размер выходных данных Map превышает указанный процент памяти, определенный с помощью mapreduce.reduce.shuffle.input.buffer.percent (по умолчанию 0,7) на 70%, сборщик сохраняет выходные данные на локальный диск редуктора.

10. Слияние - следующий шаг после сборщика. Поток слияния может работать параллельно с заданиями по извлечению и объединяет все извлеченные записи. Последний шаг в общем процессе - запуск задачи редуктора. Редуктор читает запись из объединенного файла и обрабатывает ее в методе Reduce. Для каждого ключа задача reduce будет выполнена один раз, а это означает, что если в выходном объединенном файле имеется 55 ключей, то редуктор запустит функцию reduce 55 раз.

**Оптимизация MapReduce**

Инфраструктура MapReduce предоставляет огромное преимущество для повышения производительности для больших наборов данных, поскольку мы можем добавить больше узлов, чтобы повысить производительность. Такие ресурсы, как узел, память и диск, требуют значительных инвестиций, поэтому только добавление узла не должно быть параметром для оптимизации производительности. Иногда добавление большего количества узлов не помогает повысить производительность, поскольку производительность приложения может быть достигнута чем-то другим, например, оптимизацией кода, нежелательной передачей данных и т. д.

Производительность приложения измеряется общим временем обработки, которое выполняется приложение. MapReduce обрабатывает данные параллельно и, таким образом, уже обеспечивает преимущество в производительности по сравнению с вашим приложением MapReduce. Следующие факторы играют важную роль в оптимизации производительности MapReduce.

**Настройка оборудования**

Настройка оборудования - это первый шаг в установке Hadoop. Производительность всегда зависит от конфигурации оборудования, которая используется для приложения. Система с более высокой вычислительной мощностью всегда будет давать лучшую производительность, чем система с более низкой вычислительной мощностью. Система с большим объемом памяти всегда будет иметь более высокую производительность, чем система с меньшим объемом памяти. В Hadoop полоса пропускания сети также играет важную роль, поскольку задания MapReduce могут потребовать перемещения данных с одного компьютера на другой. Таким образом, нам потребуется большая пропускная способность сети, чтобы завершить процесс как можно быстрее

**Настройка операционной системы**

Операционная система отвечает за большинство задач системного уровня, таких как:  
Transparent Huge Pages (THP): на машинах, используемых в Hadoop, должен быть отключен THP. Давайте разберемся, что такое THP. В большинстве систем Linux размер блока по умолчанию составляет 4 КБ, поэтому в больших файлах будет больше физических блоков. Обработка файлов потребует загрузки большего количества блоков в память и, следовательно, требует большего количества итераций, что может привести к снижению производительности. THP выделяет отдельные адреса памяти для всех блоков, которые называются огромными страницами, и, таким образом, для чтения и обработки файла требуется меньше итераций.

Hadoop уже имеет больший размер блока 128 МБ, и блоки не хранятся в непрерывной памяти. Блоки позволяют Hadoop обрабатывать данные параллельно.

THP плохо работает в кластере Hadoop и вызывает высокую загрузку ЦП. Рекомендуется отключить THP на каждом рабочем узле. Это иногда приводит к повышению производительности. Следующий код можно добавить в файл /etc/rc.local для отключения THP:

if test -f /sys/kernel/mm/redhat\_transparent\_hugepage/defrag; then echo never > /sys/kernel/mm/redhat\_transparent\_hugepage/defrag ;fi

**Избегайте ненужного swapping памяти**: в Hadoop swapping может повлиять на производительность работы, и поэтому мы должны избегать ненужнго swap данных из памяти в файл подкачки, и это следует делать только при необходимости. Настройку swapping можно установить на 0 (ноль), чтобы избежать swap, если только это не требуется. Значение 100 означает, что данные будут немедленно заменены местами. Пространство подкачки поддерживается дисковым пространством и всегда медленнее, чем в памяти. Нам нужно добавить vm.swappiness = 0 в файл /etc/sysctl.conf, чтобы включить этот функционал.

**Конфигурация CPU**: В большинстве операционных систем CPU настроен на энергосбережение и, следовательно, не оптимизирован для таких систем, как Hadoop, где CPU занят выполнением задач большую часть времени. По умолчанию регулятор масштабирования установлен в режим энергосбережения, и нам нужно изменить его режим производительности, выполнив следующую команду:

производительность cpufreq-set -r -g

**Настройка сети**: перетасовка данных занимает значительное время в Hadoop, и, таким образом, оптимальное использование пропускной способности сети может помочь нам достичь более высокой производительности. Мастер и slave взаимодействуют друг с другом, но количество соединений, которые мастер может установить с slave за один раз, ограничено с помощью net.core.somaxconn. В Hadoop соединение между мастером и slave происходит очень часто, поэтому для net.core.somaxconn следует установить более высокое значение. Это можно сделать, добавив или отредактировав /etc/sysctl.conf:

net.core.somaxconn=1024

**Выбор файловой системы**: Дистрибутив Linux поставляется с файловой системой по умолчанию, и он может оказать существенное влияние на производительность Hadoop, поскольку он предназначен для работы с интенсивными операциями ввода-вывода. Последние дистрибутивы Linux поставляется с EXT4 в качестве файловой системы по умолчанию, которая работает лучше, чем файловая система EXT3. Файловая система записывает время последнего доступа для каждой операции чтения в файл и, таким образом, вызывает операцию записи на диск для каждой операции чтения. Параметр ведения журнала можно отключить, добавив атрибут noatime в параметр монтирования файловой системы. В некоторых случаях было отмечено улучшение производительности более чем на 20% за счет добавления noatime.

**Методы оптимизации**

Хороший выбор оборудования выполняет 30% работы, но всегда хорошо иметь оптимизированную и сбалансированную конфигурацию кластера, основанную на требованиях приложения. Давайте рассмотрим несколько методов, которые могут помочь нам повысить производительность приложений MapReduce:

**Использование combiner**: передача данных по сети является дорогостоящей задачей, и, следовательно, передача большего количества данных всегда приводит к увеличению времени обработки. Combiner действует как мини-редуктор, и он работает на машине Mapper. Редуктор не может использоваться во всех случаях, но в большинстве случаев мы можем использовать combiner. Это уменьшает размер данных, которые будут передаваться по сети на этапе передачи.

**Сжатие выходных данных map**: Mapper обрабатывает выходные данные и сохраняет их на локальном диске. Этот промежуточный результат может быть сжат с использованием LZO-сжатия, так что он уменьшает дисковый ввод-вывод во время перемешивания. Результаты более заметны в тех случаях, когда Mapper генерирует большое количество выходных данных. Чтобы включить сжатие LZO, нам нужно установить для mapred.compress.map.output значение true.

**Фильтрация записей**: всегда полезно отфильтровать записи на стороне устройства Mapper, чтобы Mapper всегда записывал меньше данных на локальный диск. Все последующие шаги будут выполняться быстрее, так как они должны обрабатывать сравнительно меньше данных, чем раньше. Большая часть времени будет сэкономлена на этапе перетасовки, так как данные должны перемещаться по сети к редуктору.

**Избегайте слишком большого количества маленьких файлов**. Слишком много маленьких файлов может привести к тому, что ваше приложение займет больше времени для выполнения. HDFS будет хранить эти файлы в виде отдельного блока, и будет слишком много времени для запуска слишком большого числа Mapper для обработки этих файлов. Хорошей идеей является сжатие небольших файлов в один большой файл, а затем запуск поверх него приложения MapReduce. В некоторых случаях есть улучшение производительности примерно на 100%.

**Избегайте неразделимых файловых форматов**. Нерасщепляемые файловые форматы, такие как .gzip, будут обрабатываться сразу, а не порциями. Если эти файлы слишком малы по размеру, обработка займет больше времени, потому что для каждого файла будет запущен один Mapper. Если существует 200 файлов, то будет запущено 200 Mapper. Время запуска и остановки Mapper займет больше времени, чем обработка файлов. Лучшей идеей является использование разделяемого формата файла, такого как текст, AVRO, ORC и т.д.

**Конфигурация времени выполнения**

Hadoop поставляется с набором параметров для настройки памяти, диска и оптимизации сетевой производительности заданий Hadoop. Давайте посмотрим на некоторые из них:

**Память Java для задач**: задачи map и reduce - это процессы JVM, и они используют память JVM для выполнения. Больше памяти в конечном итоге приведет к повышению производительности. Размер памяти можно установить с помощью свойства mapred.child.java.opts.

**Map spill memory**. Выходные записи, генерируемые Mapper, хранятся в кольцевом буфере, а размер буфера по умолчанию составляет 100 МБ. Помните, что как только размер вывода превысит 70% от 100 МБ, что означает 70 МБ, данные будут попадать на диск в файле. Таким образом, если будет выполнено 7 операций по удалению, будет 7 файлов. Эти файлы затем объединяются в единый файл. Наша цель - иметь небольшое количество файлов spill и сократить время ввода-вывода для записи файлов spilling на диск. Это может быть сделано путем увеличения буферной памяти в случае, если Mapper создает больше spill. Размер можно установить с помощью свойства io.sort.mb.

**Задача настройки Map**: количество Mapper определяется неявно средой Hadoop, и она контролируется mapred.min.split.size. Идея состоит в том, чтобы контролировать количество Mapper, запускаемых приложением, чтобы обеспечить баланс между размером входных данных и количеством Mapper. Если слишком много маленьких задач выполняются друг за другом, тогда лучше установить mapred.job.reuse.jvm.num.tasks в -1. Пожалуйста, не используйте это свойство, если у вас есть долго выполняющиеся задачи, так как запуск новых издержек JVM не увеличит производительность. Вместо этого он может снизить производительность. В большинстве случаев, если размер входных данных слишком велик, рекомендуется увеличить разбивку входных данных до большего числа.

**Оптимизация файловой системы**

Диск HDFS поставляется с определенными файловыми системами, такими как Ext4, Ext3 или XFS, и настройка файловой системы для повышения производительности значительно повысит производительность обработки. Давайте рассмотрим несколько общеизвестных вариантов настройки HDFS:

**Опция монтирования**: Есть несколько опций монтирования, которые эффективны для кластеров Hadoop. Правильный вариант монтирования обеспечивает хорошую производительность. Не забудьте перезагрузить систему после применения настроек, так как простое изменение конфигурации не будет работать. Вам необходимо перемонтировать систему, а затем перезагрузить ее. Ext4 и XFS должны быть настроены noatime.

**Размер блока HDFS**. Размер блока играет важную роль в повышении производительности NameNode, а также производительности выполнения заданий. NameNode хранит метаданные для каждого блока, который он хранит в DataNode, и, таким образом, занимает больше памяти в случае, если размер блока намного меньше рекомендуемого размера блока. Обработчик, такой как MapReduce, также запускает число Mapper, равное размеру разделения, которое обычно равно размеру блока. Рекомендуемое значение для dfs.blocksize должно быть в диапазоне от 134 217 728 до 1 073 741 824.

Оптимальный размер блока HDFS повышает производительность NameNode, а также производительность выполнения заданий.

**Short circuit read**: операция чтения в HDFS проходит через DataNode, что означает, что клиент запрашивает DataNode для чтения файла, а DataNode отправляет данные файла через сокет TCP клиенту. При чтении с short circuit клиент непосредственно считывает файл и, таким образом, обходит DataNode в процессе, но помните, что это может произойти только тогда, когда данные находятся на клиенте . В большинстве случаев считывание short circuit обеспечивает значительное улучшение производительности. Следующее свойство может быть добавлено в hdfs-site.xml, чтобы включить чтение short circuit:

dfs.client.read.shortcircuit = TRUE  
dfs.domain.socket.path = / вар / Библиотека / Hadoop-HDFS / dn\_socket

**Проблема с небольшими файлами**: Hadoop оптимизирован для хранения больших файлов, поэтому рекомендуется иметь размер файла в списке, равный размеру блока HDFS. Если слишком много маленьких файлов, это увеличит нагрузку на память NameNode и окажет негативное влияние на производительность во время обработки, так как для каждого блока будет запущен новый Mapper. Поэтому рекомендуется выполнять сжатие таких небольших файлов, чтобы сделать их одним большим файлом.

**Устаревший DataNode**: DataNode регулярно отправляет heartbeat в NameNode, чтобы NameNode знал, что его DataNode все еще доступен. Предполагается, что узлы данных, которые не отправляли heartbeat в NameNode в течение определенного интервала, устарели. Мы должны избегать отправки любых запросов на чтение или запись на такие узлы данных. Это можно сделать, добавив следующее свойство в hdfs-site.xml:

dfs.namenode.avoid.read.stale.datanode = TRUE  
dfs.namenode.avoid.write.stale.datanode = TRUE