# Введение в Map-Reduce

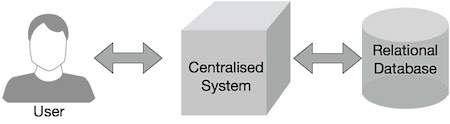
MapReduce – это модель программирования для написания приложений, которые могут обрабатывать большие данные параллельно на нескольких узлах. MapReduce предоставляет аналитические возможности для анализа огромных объемов сложных данных.

**Что такое большие данные?**

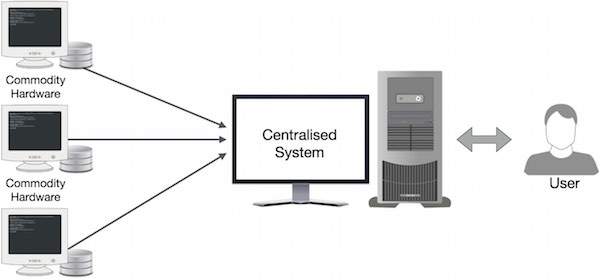
Большие данные – это набор больших наборов данных, которые не могут быть обработаны с использованием традиционных вычислительных технологий. Например, объем данных, которые Facebook или Youtube должны требовать для ежедневного сбора и обработки, может подпадать под категорию больших данных. Однако большие данные касаются не только масштаба и объема, но и одного или нескольких из следующих аспектов – скорость, разнообразие, объем и сложность.

***Почему MapReduce?***

Традиционные корпоративные системы обычно имеют централизованный сервер для хранения и обработки данных.  Иллюстрация изображает схематическое представление традиционной системы предприятия. Традиционная модель, безусловно, не подходит для обработки огромных объемов масштабируемых данных и не может быть размещена на стандартных серверах баз данных. Более того, централизованная система создает слишком много узких мест при одновременной обработке нескольких файлов.



Google решил эту проблему, используя алгоритм MapReduce. MapReduce делит задачу на маленькие части и назначает их многим компьютерам. Позже результаты собираются в одном месте и объединяются для формирования результирующего набора данных.



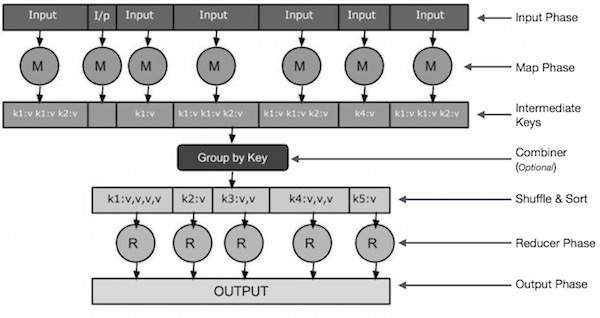
**Как работает MapReduce?**

Алгоритм MapReduce содержит две важные задачи, а именно Map и Reduce.

* Задача «Карта» берет набор данных и преобразует его в другой набор данных, где отдельные элементы разбиваются на кортежи (пары ключ-значение).
* Задача Reduce принимает выходные данные из карты в качестве входных данных и объединяет эти кортежи данных (пары ключ-значение) в меньший набор кортежей.

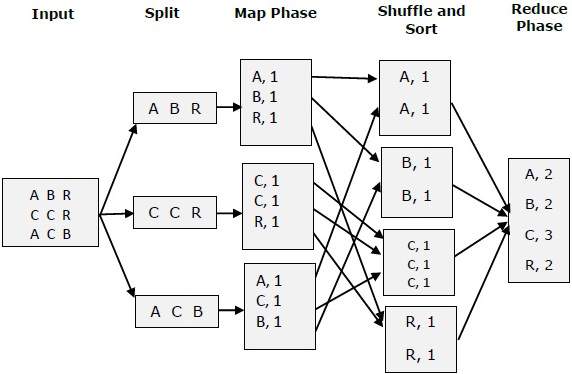
Задача уменьшения всегда выполняется после задания карты.

Теперь внимательно рассмотрим каждый из этапов.



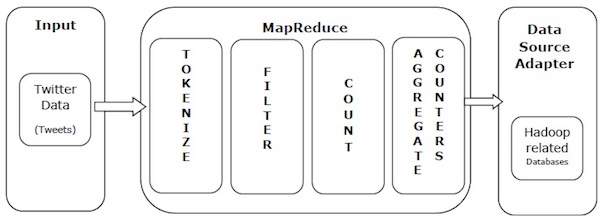
* **Фаза ввода** – Здесь есть Record Reader, который переводит каждую запись во входной файл и отправляет проанализированные данные в маппер в виде пар ключ-значение.
* **Карта** –это пользовательская функция, которая принимает серию пар ключ-значение и обрабатывает каждую из них, чтобы сгенерировать ноль или более пар ключ-значение.
* **Промежуточные ключи** – пары «ключ-значение», генерируемые картографом, называются промежуточными ключами.
* **Объединитель** – это тип локального редуктора, который группирует аналогичные данные из фазы карты в идентифицируемые наборы. Он принимает промежуточные ключи от преобразователя в качестве входных данных и применяет пользовательский код для агрегирования значений в небольшой области одного преобразователя. Он не является частью основного алгоритма MapReduce; это необязательно.
* **Перемешать и отсортировать** – задача «Восстановитель» начинается с шага «Перемешать и сортировать». Он загружает сгруппированные пары ключ-значение на локальный компьютер, на котором работает редуктор. Отдельные пары ключ-значение сортируются по ключу в больший список данных. Список данных группирует эквивалентные ключи вместе, так что их значения могут быть легко повторены в задаче Reducer.
* **Редуктор** –принимает сгруппированные парные данные ключ-значение в качестве входных данных и запускает функцию Редуктор для каждого из них. Здесь данные могут быть агрегированы, отфильтрованы и объединены различными способами, что требует широкого спектра обработки. Как только выполнение закончено, он дает ноль или более пар ключ-значение для последнего шага.
* **Фаза вывода.** На этапе вывода у нас есть выходной форматер, который переводит конечные пары ключ-значение из функции Reducer и записывает их в файл с помощью средства записи.

Теперь попробуем понять две задачи Map & f Reduce с помощью небольшой диаграммы



***MapReduce-пример***

Пример из реальной жизни, чтобы понять мощь MapReduce. Twitter получает около 500 миллионов твитов в день, то есть почти 3000 твитов в секунду. На рисунке показано, как Tweeter управляет своими твитами с помощью MapReduce.



Как показано на рисунке, алгоритм MapReduce выполняет следующие действия:

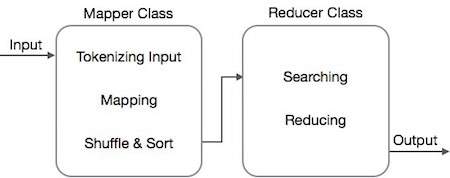
* **Tokenize** – токенизирует твиты в карты токенов и записывает их в виде пар ключ-значение.
* **Фильтр** – Фильтрует нежелательные слова из карт токенов и записывает отфильтрованные карты в виде пар ключ-значение.
* **Count** – генерирует счетчик токенов на слово.
* **Сводные счетчики** – готовит совокупность аналогичных значений счетчиков в небольшие управляемые единицы.

***MapReduce – Алгоритм***

Алгоритм MapReduce содержит две важные задачи, а именно Map и Reduce.

* Задача карты выполняется с помощью Mapper Class
* Задача уменьшения выполняется с помощью класса редуктора.

Класс Mapper принимает входные данные, маркирует их, отображает и сортирует их. Выходные данные класса Mapper используются в качестве входных данных классом Reducer, который, в свою очередь, ищет совпадающие пары и сокращает их.



MapReduce реализует различные математические алгоритмы, чтобы разделить задачу на маленькие части и назначить их нескольким системам. С технической точки зрения алгоритм MapReduce помогает отправлять задачи Map & Reduce на соответствующие серверы в кластере.

Эти математические алгоритмы могут включать в себя следующее:

* Сортировка
* поиск
* индексирование
* TF-IDF

***Сортировка***

Сортировка является одним из основных алгоритмов MapReduce для обработки и анализа данных. MapReduce реализует алгоритм сортировки для автоматической сортировки выходных пар ключ-значение из преобразователя по их ключам.

* Методы сортировки реализованы в самом классе mapper.
* На этапе перемешивания и сортировки после токенизации значений в классе сопоставления класс **Context** (определенный пользователем класс) собирает совпадающие значения ключей в виде коллекции.
* Чтобы собрать похожие пары ключ-значение (промежуточные ключи), класс Mapper использует класс **RawComparator** для сортировки пар ключ-значение.
* Набор промежуточных пар ключ-значение для данного редуктора автоматически сортируется Hadoop для формирования значений ключа (K2, {V2, V2,…}) до их представления редуктору.

***поиск***

Поиск играет важную роль в алгоритме MapReduce. Это помогает в фазе объединителя (опция) и в фазе редуктора.

***Пример***

В следующем примере показано, как MapReduce использует алгоритм поиска, чтобы узнать подробности о сотруднике, который получает самую высокую зарплату в данном наборе данных сотрудников.

* Предположим, у нас есть данные о сотрудниках в четырех разных файлах – A, B, C и D. Предположим также, что во всех четырех файлах есть дубликаты записей о сотрудниках из-за повторного импорта данных о сотрудниках из всех таблиц базы данных. Смотрите следующую иллюстрацию.



* **Фаза Map** обрабатывает каждый входной файл и предоставляет данные о сотруднике в парах ключ-значение (<k, v>: <emp name, salary>). Смотрите следующую иллюстрацию.



* **Фаза объединителя** (метод поиска) будет принимать входные данные из фазы карты в виде пары ключ-значение с именем сотрудника и зарплатой. Используя технику поиска, комбинатор проверит всю зарплату сотрудника, чтобы найти сотрудника с наибольшим окладом в каждом файле. Смотрите следующий фрагмент.

<k: employee name, v: salary>

Max= the salary of an first employee. Treated as max salary

if(v(second employee).salary > Max){

Max = v(salary);

}

else{

Continue checking;

}

Ожидаемый результат заключается в следующем –

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  | | --- | | <сатиш, 26000> | |  | |  | | --- | | <гопал, 50000> | |  | |  | | --- | | <Киран, 45000> | |  | |  | | --- | | <Маниша, 45000> | |  | |

* **Этап сокращения** – Сформируйте каждый файл, вы найдете самый высокооплачиваемый сотрудник. Чтобы избежать избыточности, проверьте все пары <k, v> и удалите дублирующиеся записи, если таковые имеются. Тот же алгоритм используется между четырьмя парами <k, v>, которые поступают из четырех входных файлов. Окончательный результат должен быть следующим:

<gopal, 50000>

***индексирование***

Обычно индексация используется для указания на конкретные данные и их адрес. Он выполняет пакетную индексацию входных файлов для определенного Mapper.

Техника индексирования, которая обычно используется в MapReduce, называется **инвертированным индексом.** Поисковые системы, такие как Google и Bing, используют метод перевернутой индексации.

***пример***

Следующий текст является вводом для инвертированной индексации. Здесь T [0], T [1] и t [2] – имена файлов, а их содержимое заключено в двойные кавычки.

T[0] = "it is what it is"

T[1] = "what is it"

T[2] = "it is a banana"

После применения алгоритма индексирования мы получаем следующий вывод:

"a": {2}

"banana": {2}

"is": {0, 1, 2}

"it": {0, 1, 2}

"what": {0, 1}

Здесь «a»: {2} подразумевает, что термин «a» появляется в файле T [2]. Аналогично, «is»: {0, 1, 2} подразумевает, что термин «is» появляется в файлах T [0], T [1] и T [2].

# Пример TF-IDF

TF-IDF – это алгоритм обработки текста, сокращенный от Term Frequency – Inverse Document Frequency. Это один из распространенных алгоритмов веб-анализа. Термин «частота» относится к числу раз, когда термин появляется в документе.

***Термин частота (TF)***

Измеряет, как часто конкретный термин встречается в документе. Рассчитывается по количеству появлений слова в документе, деленному на общее количество слов в этом документе.

TF(the) = (Number of times term the ‘the’ appears in a document) / (Total number of terms in the document)

***Частота обратных документов (IDF)***

Он измеряет важность термина. Он рассчитывается по количеству документов в текстовой базе данных, деленному на количество документов, в которых появляется конкретный термин.

При вычислении TF все термины считаются одинаково важными. Это означает, что TF подсчитывает частоту термина для обычных слов, таких как «есть», «а», «что» и т. Д. Таким образом, нам нужно знать частые термины при увеличении количества редких, вычисляя следующее:

IDF(the) = log\_e(Total number of documents / Number of documents with term ‘the’ in it).

***пример***

Рассмотрим документ, содержащий 1000 слов, в котором слово **улей** появляется 50 раз. Тогда TF для **улья** (50/1000) = 0,05.

Теперь предположим, что у нас есть 10 миллионов документов и слово « **куст»** появляется в 1000 из них. Затем IDF рассчитывается как log (10 000 000/1000) = 4.

Вес TF-IDF является произведением этих количеств – 0,05 × 4 = 0,20.

**MapReduce – API**

Рассмотрим:

* Интерфейс JobContext
* Класс работы
* Mapper Class
* Класс редуктора

**Интерфейс JobContext**

Интерфейс JobContext – это суперинтерфейс для всех классов, который определяет различные задания в MapReduce. Это дает доступ только для чтения к заданию, которое предоставляется задачам во время их выполнения.

Приведены подинтерфейсы интерфейса JobContext.

|  |  |
| --- | --- |
| **S.No.** | **Подинтерфейс Описание** |
| 1. | **MapContext <KEYIN, VALUEIN, KEYOUT, VALUEOUT>**  Определяет контекст, который предоставляется Mapper. |
| 2. | **ReduceContext <KEYIN, VALUEIN, KEYOUT, VALUEOUT>**  Определяет контекст, который передается редуктору. |

Класс Job – это основной класс, который реализует интерфейс JobContext.

**Класс работы**

Класс Job является наиболее важным классом в API MapReduce. Это позволяет пользователю настраивать задание, отправлять его, контролировать его выполнение и запрашивать состояние. Методы set работают только до отправки задания, после чего они генерируют исключение IllegalStateException. Обычно пользователь создает приложение, описывает различные аспекты задания, а затем отправляет задание и отслеживает его выполнение.Вот пример того, как подать заявку –

// Create a new Job

Job job = new Job(new Configuration());

job.setJarByClass(MyJob.class);

// Specify various job-specific parameters

job.setJobName("myjob");

job.setInputPath(new Path("in"));

job.setOutputPath(new Path("out"));

job.setMapperClass(MyJob.MyMapper.class);

job.setReducerClass(MyJob.MyReducer.class);

// Submit the job, then poll for progress until the job is complete

job.waitForCompletion(true);

**Конструкторы**

Ниже приводится краткое описание конструктора класса Job.

|  |  |
| --- | --- |
| **S.No** | **Сводка конструктора** |
| 1 | **Работа** () |
| 2 | **Job** (Конфиг конфигурации) |
| 3 | **Job** (Конфигурация conf, String jobName) |

**методы**

Вот некоторые из важных методов класса Job:

|  |  |
| --- | --- |
| **S.No** | **Описание метода** |
| 1 | **getJobName ()**  Заданное пользователем имя задания. |
| 2 | **getJobState ()**  Возвращает текущее состояние задания. |
| 3 | **завершено()**  Проверяет, закончена ли работа или нет. |
| 4 | **setInputFormatClass ()**  Устанавливает InputFormat для работы. |
| 5 | **setJobName (имя строки)**  Устанавливает заданное пользователем имя задания. |
| 6 | **setOutputFormatClass ()**  Устанавливает формат вывода для работы. |
| 7 | **setMapperClass (класс)**  Устанавливает Mapper для работы. |
| 8 | **setReducerClass (класс)**  Устанавливает Редуктор для работы. |
| 9 | **setPartitionerClass (класс)**  Устанавливает Partitioner для работы. |
| 10 | **setCombinerClass (класс)**  Устанавливает Combiner для работы. |

**Mapper Class**

Класс Mapper определяет задание Map. Сопоставляет входные пары ключ-значение с набором промежуточных пар ключ-значение. Карты – это отдельные задачи, которые преобразуют входные записи в промежуточные записи. Преобразованные промежуточные записи не обязательно должны быть того же типа, что и входные записи. Заданная входная пара может отображаться на ноль или на множество выходных пар.

**метод**

**карта** является наиболее известным методом класса Mapper. Синтаксис представлен

map(KEYIN key, VALUEIN value, org.apache.hadoop.mapreduce.Mapper.Context context)

Этот метод вызывается один раз для каждой пары ключ-значение во входном разбиении.

**Класс редуктора**

Класс Reducer определяет задание Reduce в MapReduce. Это уменьшает набор промежуточных значений, которые разделяют ключ, до меньшего набора значений. Реализации редуктора могут получить доступ к Конфигурации для задания через метод JobContext.getConfiguration (). Редуктор имеет три основных этапа – перемешивание, сортировка и уменьшение.

* **Перемешать** – Редуктор копирует отсортированный вывод из каждого Mapper, используя HTTP по всей сети.
* **Сортировать** – платформа объединяет сортировку входов Редуктора по ключам (поскольку разные Mappers могут выводить один и тот же ключ). Фазы тасования и сортировки происходят одновременно, т. Е. Во время выборки выходов они объединяются.
* **Сокращение** – На этом этапе метод Redu (Object, Iterable, Context) вызывается для каждого <ключа, (набора значений)> в отсортированных входных данных.

**метод**

**Редукция** – самый известный метод класса Редукторов.  Синтаксис представлен–

**reduce** (KEYIN key, Iterable<VALUEIN> values, org.apache.hadoop.mapreduce.Reducer.Context context)

Этот метод вызывается один раз для каждого ключа в коллекции пар ключ-значение.

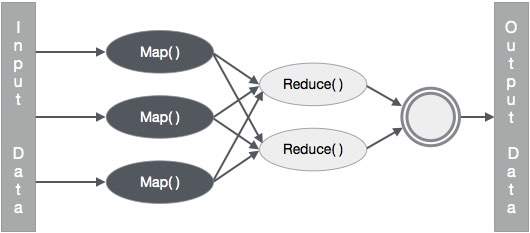
# MapReduce – реализация Hadoop

MapReduce – это инфраструктура, которая используется для написания приложений для надежной обработки огромных объемов данных на больших кластерах аппаратного оборудования.

**Алгоритм MapReduce**

Обычно парадигма MapReduce основана на отправке программ сокращения карт на компьютеры, где хранятся фактические данные.

* Во время задания MapReduce Hadoop отправляет задачи Map и Reduce на соответствующие серверы в кластере.
* Каркас управляет всеми деталями передачи данных, такими как выдача задач, проверка выполнения задач и копирование данных вокруг кластера между узлами.
* Большая часть вычислений происходит на узлах с данными на локальных дисках, что снижает сетевой трафик.
* После выполнения заданной задачи кластер собирает и сокращает данные, чтобы сформировать соответствующий результат, и отправляет их обратно на сервер Hadoop.



**Входы и выходы (перспектива Java)**

Каркас MapReduce работает с парами ключ-значение, то есть каркас рассматривает входные данные для задания в виде набора пар ключ-значение и создает набор пар ключ-значение в качестве выходных данных задания, предположительно различных типов.

Классы ключа и значения должны быть сериализуемы платформой, и, следовательно, необходимо реализовать интерфейс Writable. Кроме того, ключевые классы должны реализовывать интерфейс WritableComparable для облегчения сортировки в рамках.

Оба формата ввода и вывода задания MapReduce представлены в виде пар ключ-значение

|  |  |  |
| --- | --- | --- |
|  | **вход** | **Выход** |
| **карта** | <k1, v1> | список (<k2, v2>) |
| **уменьшить** | <k2, список (v2)> | список (<k3, v3>) |

**Реализация MapReduce**

В таблице приведены данные, касающиеся потребления электроэнергии в организации. Таблица включает ежемесячное потребление электроэнергии и среднегодовое значение за пять лет подряд.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **январь** | **февраль** | **март** | **апрель** | **май** | **июнь** | **июль** | **август** | **сентябрь** | **октябрь** | **ноябрь** | **декабрь** | **в среднем** |
| 2017 | 23 | 23 | 2 | 43 | 24 | 25 | 26 | 26 | 26 | 26 | 25 | 26 | 25 |
| 2018 | 26 | 27 | 28 | 28 | 28 | 30 | 31 | 31 | 31 | 30 | 30 | 30 | 29 |
| 2019 | 31 | 32 | 32 | 32 | 33 | 34 | 35 | 36 | 36 | 34 | 34 | 34 | 34 |
| 2020 | 39 | 38 | 39 | 39 | 39 | 41 | 42 | 43 | 40 | 39 | 38 | 38 | 40 |
| 2021 | 38 | 39 | 39 | 39 | 39 | 41 | 41 | 41 | 00 | 40 | 39 | 39 | 45 |

Нужно написать приложение для обработки входных данных в данной таблице, чтобы найти год максимального использования, год минимального использования и так далее. Эта задача проста для программистов с конечным количеством записей, поскольку они просто напишут логику для получения требуемого вывода и передадут данные в написанное приложение. Теперь поднимем масштаб входных данных. Предположим, мы должны проанализировать потребление электроэнергии всеми крупными отраслями конкретного государства. Когда мы пишем приложение для обработки таких массовых данных,

* Они займут много времени, чтобы выполнить.
* При переносе данных из источника на сетевой сервер будет большой сетевой трафик.

**Входные данные**

Приведенные выше данные сохраняются как **sample.txt** и передаются в качестве входных данных. Входной файл выглядит так, как показано ниже.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **2017** | 23 | 23 | 2 | 43 | 24 | 25 | 26 | 26 | 26 | 26 | 25 | 26 | **25** |
| **2018** | 26 | 27 | 28 | 28 | 28 | 30 | 31 | 31 | 31 | 30 | 30 | 30 | **29** |
| **2019** | 31 | 32 | 32 | 32 | 33 | 34 | 35 | 36 | 36 | 34 | 34 | 34 | **34** |
| **2022** | 39 | 38 | 39 | 39 | 39 | 41 | 42 | 43 | 40 | 39 | 38 | 38 | **40** |
| **2021** | 38 | 39 | 39 | 39 | 39 | 41 | 41 | 41 | 00 | 40 | 39 | 39 | **45** |

**MapReduce – Partitioner**

Секционер работает как условие при обработке входного набора данных. Фаза разбиения происходит после фазы Map и до фазы Reduce.

Количество секционеров равно количеству редукторов. Это означает, что разделитель разделит данные в соответствии с числом редукторов. Следовательно, данные, передаваемые с одного разделителя, обрабатываются одним редуктором.

**Разметка**

Разделитель разделяет пары ключ-значение промежуточных выходов Map. Он разделяет данные, используя пользовательское условие, которое работает как хэш-функция. Общее количество разделов совпадает с количеством заданий Reducer для задания. Давайте рассмотрим пример, чтобы понять, как работает разделитель.

**Внедрение MapReduce Partitioner**

Для удобства предположим, что у нас есть небольшая таблица Employee со следующими данными. Мы будем использовать этот пример данных в качестве нашего входного набора данных, чтобы продемонстрировать, как работает разделитель.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **название** | **Возраст** | **Пол** | **Оплата труда** |
| 1201 | Гопал | 45 | мужчина | 50000 |
| 1202 | Маниша | 40 | женский | 50000 |
| 1203 | Халил | 34 | мужчина | 30000 |
| 1204 | Prasanth | 30 | мужчина | 30000 |
| 1205 | Киран | 20 | мужчина | 40000 |
| 1206 | Лакшми | 25 | женский | 35000 |
| 1207 | Бхавайя | 20 | женский | 15000 |
| 1208 | Reshma | 19 | женский | 15000 |
| 1209 | kranthi | 22 | мужчина | 22000 |
| 1210 | Сатиш | 24 | мужчина | 25000 |
| 1211 | Кришна | 25 | мужчина | 25000 |
| 1212 | Аршад | 28 | мужчина | 20000 |
| 1213 | Лаванья | 18 | женский | 8000 |

Мы должны написать заявление для обработки входного набора данных, чтобы найти работника с наибольшим окладом по полу в разных возрастных группах (например, ниже 20 лет, от 21 до 30 лет, выше 30 лет).

**Входные данные**

Приведенные выше данные сохраняются в **файле input.txt** в **каталоге** «/ home / hadoop / hadoopPartitioner» и передаются в качестве входных данных.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1201 | Гопал | 45 | мужчина | 50000 |
| 1202 | Маниша | 40 | женский | 51000 |
| 1203 | Khaleel | 34 | мужчина | 30000 |
| 1204 | Prasanth | 30 | мужчина | 31000 |
| 1205 | Киран | 20 | мужчина | 40000 |
| 1206 | Лакшми | 25 | женский | 35000 |
| 1207 | Бхавайя | 20 | женский | 15000 |
| 1208 | Reshma | 19 | женский | 14 000 |
| 1209 | kranthi | 22 | мужчина | 22000 |
| +1210 | Сатиш | 24 | мужчина | 25000 |
| 1211 | Кришна | 25 | мужчина | 26000 |
| 1212 | Аршад | 28 | мужчина | 20000 |
| 1213 | Лаванья | 18 | женский | 8000 |

**Задачи карты**

Задача карты принимает пары ключ-значение в качестве входных данных, пока у нас есть текстовые данные в текстовом файле. Входные данные для этой задачи карты следующие:

**Ввод** – ключом будет шаблон, такой как «любая специальная клавиша + имя файла + номер строки» (пример: ключ = @ input1), а значением будут данные в этой строке (пример: значение = 1201 \ t gopal \ t 45 \ т мужской \ т 50000).

**Метод** – Работа этой задачи карты заключается в следующем –

* Прочитайте **значение** (запись данных), которое поступает в качестве входного значения из списка аргументов в строке.
* Используя функцию split, разделите пол и сохраните в строковую переменную.

Прочитайте **значение** (запись данных), которое поступает в качестве входного значения из списка аргументов в строке.

Используя функцию split, разделите пол и сохраните в строковую переменную.

String[] str = value.toString().split("\t", -3);

String gender=str[3];

* Отправьте информацию о поле и значение записи в виде пары выходной ключ-значение из задачи сопоставления в задачу **разбиения** .

context.write(new Text(gender), new Text(value));

* Повторите все вышеперечисленные шаги для всех записей в текстовом файле.

**Вывод.** Получим готовые данные и данные записи в виде пар ключ-значение.

**Задача Partitioner**

Задача секционера принимает пары ключ-значение из задачи карты в качестве входных данных. Разделение подразумевает разделение данных на сегменты. В соответствии с заданными условными критериями разделов входные парные данные ключ-значение могут быть разделены на три части на основе возрастных критериев.

**Ввод** – все данные в коллекции пар ключ-значение.

ключ = значение поля пола в записи.

значение = значение всей записи данных этого пола.

**Метод** . Процесс логики разбиения выполняется следующим образом.

* Считайте значение поля возраста из пары ключ-значение.

String[] str = value.toString().split("\t");

int age = Integer.parseInt(str[2]);

* Проверьте значение возраста с соблюдением следующих условий.
  + Возраст не более 20
  + Возраст больше 20 и меньше или равно 30.
  + Возраст старше 30.

if(age<=20)

{

return 0;

}

else if(age>20 && age<=30)

{

return 1 % numReduceTasks;

}

else

{

return 2 % numReduceTasks;

}

**Вывод** . Все данные пар ключ-значение сегментированы на три набора пар ключ-значение. Редуктор работает индивидуально на каждую коллекцию.

**Уменьшить задачи**

Количество задач секционирования равно количеству задач редуктора. Здесь у нас есть три задачи секционирования, и, следовательно, у нас есть три задачи редуктора, которые нужно выполнить.

**Ввод** – Редуктор будет выполняться три раза с различным набором пар ключ-значение.

ключ = значение поля пола в записи.

значение = все данные записи этого пола.

**Метод** – следующая логика будет применяться к каждой коллекции.

* Прочитайте значение поля Зарплата каждой записи.

String [] str = val.toString().split("\t", -3);

Note: str[4] have the salary field value.

* Проверьте зарплату с помощью переменной max. Если str [4] является максимальной зарплатой, присвойте str [4] значение max, в противном случае пропустите шаг.

if(Integer.parseInt(str[4])>max)

{

max=Integer.parseInt(str[4]);

}

* Повторите шаги 1 и 2 для каждой коллекции ключей (мужские и женские – ключевые коллекции). Выполнив эти три шага, вы найдете одну максимальную зарплату из коллекции ключей для мужчин и одну максимальную зарплату из коллекции ключей для женщин.

context.write(new Text(key), new IntWritable(max));

**Вывод.** Наконец, получим набор данных пары ключ-значение в трех коллекциях разных возрастных групп. Он содержит максимальную зарплату из коллекции мужчин и максимальную зарплату из коллекции женщин в каждой возрастной группе соответственно.

После выполнения задач Map, Partitioner и Reduce три набора данных пары ключ-значение сохраняются в трех разных файлах в качестве выходных данных.

Все три задачи рассматриваются как задания MapReduce. Следующие требования и спецификации этих заданий должны быть указаны в Конфигурациях –

* Название работы
* Форматы ввода и вывода ключей и значений
* Отдельные классы для задач Map, Reduce и Partitioner

Configuration conf = getConf();

//Create Job

Job job = new Job(conf, "topsal");

job.setJarByClass(PartitionerExample.class);

// File Input and Output paths

FileInputFormat.setInputPaths(job, new Path(arg[0]));

FileOutputFormat.setOutputPath(job,new Path(arg[1]));

//Set Mapper class and Output format for key-value pair.

job.setMapperClass(MapClass.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(Text.class);

//set partitioner statement

job.setPartitionerClass(CaderPartitioner.class);

//Set Reducer class and Input/Output format for key-value pair.

job.setReducerClass(ReduceClass.class);

//Number of Reducer tasks.

job.setNumReduceTasks(3);

//Input and Output format for data

job.setInputFormatClass(TextInputFormat.class);

job.setOutputFormatClass(TextOutputFormat.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

**MapReduce – Combiners**

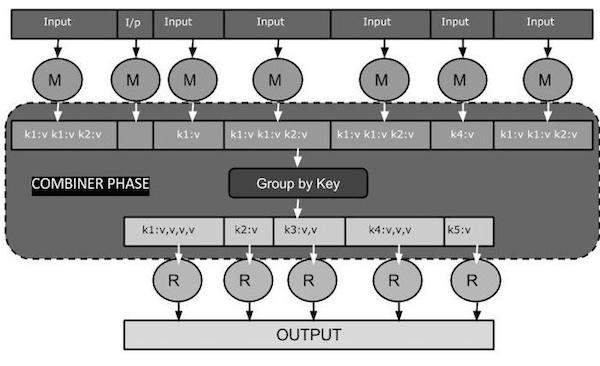
Combiner, также известный как **полуредуктор,** является необязательным классом, который работает, принимая входные данные из класса Map и затем передавая выходные пары ключ-значение в класс Reducer.

Основная функция Combiner состоит в том, чтобы суммировать выходные записи карты с одним и тем же ключом. Выход (сбор значения ключа) объединителя будет отправлен по сети фактической задаче «Редуктор» в качестве входных данных.

**Сумматор**

Класс Combiner используется между классом Map и классом Reduce для уменьшения объема передачи данных между Map и Reduce. Обычно выходные данные задачи карты большие, а данные, передаваемые в задачу сокращения, большие.

На диаграмме задачи MapReduce показана ФАЗА КОМБИНИРОВАНИЯ.



**Как работает Combiner?**

Вот краткое описание того, как работает MapReduce Combiner:

* У объединителя нет предопределенного интерфейса, и он должен реализовывать метод Redu () интерфейса Reducer.
* Комбайнер работает с каждым ключом вывода карты. Он должен иметь те же типы значений выходного ключа, что и класс Reducer.
* Объединитель может создавать сводную информацию из большого набора данных, поскольку он заменяет исходный вывод карты.

Хотя Combiner не является обязательным, он помогает разделить данные на несколько групп для фазы сокращения, что упрощает обработку.

**Реализация MapReduce Combiner**

Следующий пример дает теоретическое представление о комбайнерах. Предположим, у нас есть следующий входной текстовый файл с именем **input.txt** для MapReduce.

What do you mean by Object

What do you know about Java

What is Java Virtual Machine

How Java enabled High Performance

**Record Reader**

Это первая фаза MapReduce, где Record Reader считывает каждую строку из входного текстового файла в виде текста и выдает выходные данные в виде пар ключ-значение.

**Ввод** – построчно текст из входного файла.

**Выход** – формирует пары ключ-значение. Ниже приведен набор ожидаемых пар ключ-значение.

<1, What do you mean by Object>

<2, What do you know about Java>

<3, What is Java Virtual Machine>

<4, How Java enabled High Performance>

**Фаза карты**

Фаза Map принимает входные данные из Record Reader, обрабатывает их и создает выходные данные в виде другого набора пар ключ-значение.

**Вход** – следующая пара ключ-значение – это вход, полученный от устройства чтения записей.

<1, What do you mean by Object>

<2, What do you know about Java>

<3, What is Java Virtual Machine>

<4, How Java enabled High Performance>

Фаза Map считывает каждую пару ключ-значение, делит каждое слово на значение, используя StringTokenizer, обрабатывает каждое слово как ключ, а счетчик этого слова – как значение. В следующем фрагменте кода показаны класс Mapper и функция map.

public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>

{

private final static IntWritable one = new IntWritable(1);

private Text word = new Text();

public void map(Object key, Text value, Context context) throws IOException, InterruptedException

{

StringTokenizer itr = new StringTokenizer(value.toString());

while (itr.hasMoreTokens())

{

word.set(itr.nextToken());

context.write(word, one);

}

}

}

**Вывод** – ожидаемый вывод выглядит следующим образом –

<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>

<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>

<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>

<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>

**Фаза Combiner**

Фаза Combiner берет каждую пару ключ-значение из фазы Map, обрабатывает ее и создает выходные данные в виде пар **набора ключ-значение** .

**Ввод** – следующая пара ключ-значение является вводом, взятым из фазы карты.

<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>

<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>

<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>

<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>

Фаза Combiner считывает каждую пару ключ-значение, объединяет общие слова как ключ и значения как коллекцию. Как правило, код и работа для Combiner аналогичны таковым для Reducer. Ниже приведен фрагмент кода для объявления классов Mapper, Combiner и Reducer.

job.setMapperClass(TokenizerMapper.class);

job.setCombinerClass(IntSumReducer.class);

job.setReducerClass(IntSumReducer.class);

**Вывод** – ожидаемый вывод выглядит следующим образом –

<What,1,1,1> <do,1,1> <you,1,1> <mean,1> <by,1> <Object,1>

<know,1> <about,1> <Java,1,1,1>

<is,1> <Virtual,1> <Machine,1>

<How,1> <enabled,1> <High,1> <Performance,1>

**Фаза редуктора**

Фаза редуктора берет каждую пару сбора ключ-значение из фазы Combiner, обрабатывает ее и передает выходные данные в виде пар ключ-значение. Обратите внимание, что функциональность Combiner такая же, как и у Reducer.

**Вход** – следующая пара ключ-значение является входом, взятым из фазы Combiner.

<What,1,1,1> <do,1,1> <you,1,1> <mean,1> <by,1> <Object,1>

<know,1> <about,1> <Java,1,1,1>

<is,1> <Virtual,1> <Machine,1>

<How,1> <enabled,1> <High,1> <Performance,1>

Фаза редуктора считывает каждую пару ключ-значение. Ниже приведен фрагмент кода для Combiner.

public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable>

{

private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values,Context context) throws IOException, InterruptedException

{

int sum = 0;

for (IntWritable val : values)

{

sum += val.get();

}

result.set(sum);

context.write(key, result);

}

}

**Выходные данные** – ожидаемый выходной сигнал от фазы редуктора следующий –

<What,3> <do,2> <you,2> <mean,1> <by,1> <Object,1>

<know,1> <about,1> <Java,3>

<is,1> <Virtual,1> <Machine,1>

<How,1> <enabled,1> <High,1> <Performance,1>

**Record Writer**

Это последняя фаза MapReduce, где Record Writer записывает каждую пару ключ-значение из фазы Reducer и отправляет вывод в виде текста.

**Ввод** – каждая пара ключ-значение из фазы редуктора вместе с форматом вывода.

**Вывод** – он дает вам пары ключ-значение в текстовом формате. Ниже приводится ожидаемый результат.

What 3

do 2

you 2

mean 1

by 1

Object 1

know 1

about 1

Java 3

is 1

Virtual 1

Machine 1

How 1

enabled 1

High 1

Performance 1

**MapReduce – Администрирование Hadoop**

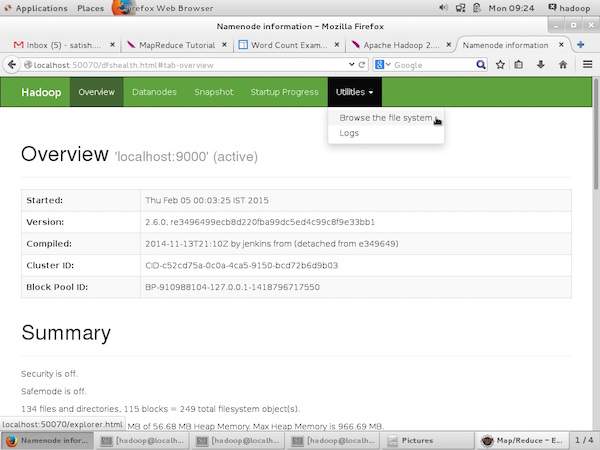
Администрирование Hadoop, включает администрирование HDFS и MapReduce.

* Администрирование HDFS включает в себя мониторинг файловой структуры HDFS, расположения и обновленных файлов.
* Администрирование MapReduce включает мониторинг списка приложений, конфигурации узлов, состояния приложений и т. Д.

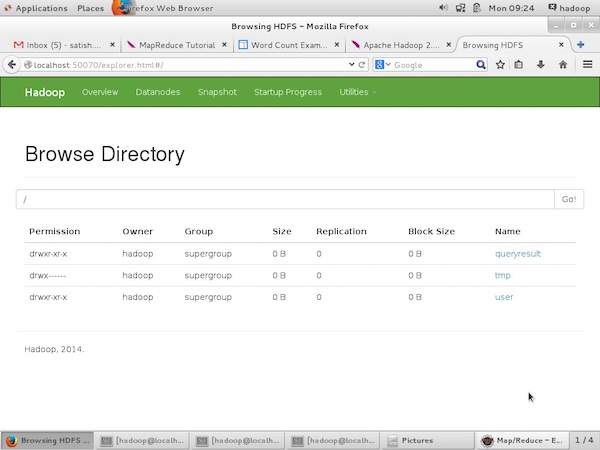
**HDFS мониторинг**

HDFS (распределенная файловая система Hadoop) содержит пользовательские каталоги, входные и выходные файлы. Команды MapReduce, **put** и **get,** для хранения и извлечения.

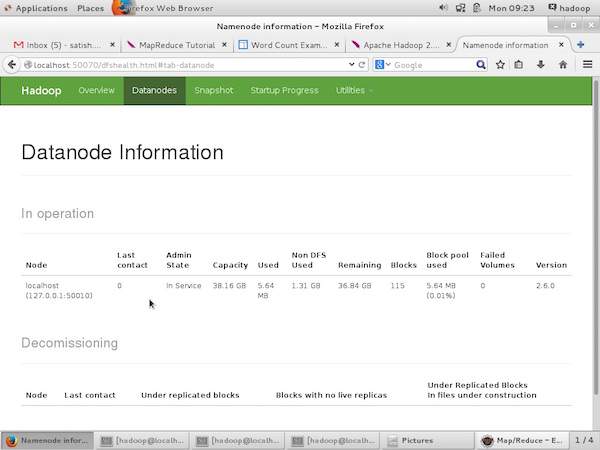
После запуска инфраструктуры Hadoop (демонов), передав команду «start-all.sh» в «/ $ HADOOP\_HOME / sbin», передайте следующий URL-адрес в браузер «http: // localhost: 50070». Вы должны увидеть следующий экран в вашем браузере.



На следующем снимке экрана показана файловая структура HDFS. Он показывает файлы в каталоге «/ user / hadoop».



На следующем снимке экрана показана информация Datanode в кластере. Здесь вы можете найти один узел с его конфигурациями и возможностями.

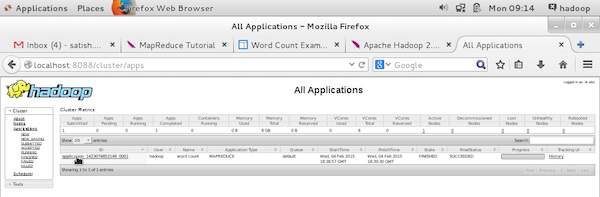


**MapReduce Job Monitoring**

Приложение MapReduce представляет собой набор заданий (задание Map, Combiner, Partitioner и Reduce job). Обязательно следить и поддерживать–

* Конфигурация датододы, где приложение подходит.
* Количество датододов и ресурсов, используемых в приложении.

Чтобы контролировать все эти вещи, обязательно, чтобы у нас был пользовательский интерфейс. После запуска инфраструктуры Hadoop, передав команду «start-all.sh» в «/ $ HADOOP\_HOME / sbin», передайте следующий URL-адрес в браузер «http: // localhost: 8080». Вы должны увидеть следующий экран в вашем браузере.



На приведенном снимке экрана указатель руки находится на идентификаторе приложения. Просто нажмите на него, чтобы найти следующий экран в вашем браузере. Это описывает следующее –

* На каком пользователе запущено текущее приложение
* Название приложения
* Тип этого приложения
* Текущий статус, Финальный статус
* Время запуска приложения, прошедшее (завершенное время), если оно завершено во время мониторинга
* История этого приложения, т.е. информация журнала
* И, наконец, информация об узлах, т. Е. Об узлах, которые участвовали в запуске приложения.

На снимке экрана показаны детали конкретного приложения.

