Lecture 4: Counting at Scale
Modeling Social Data, Spring 2017
Columbia University

February 10, 2017

Notes from as5194

1 Combining and Reshaping Data

1.1 GroupBy

- Can implement the split/apply/combine framework with group_by
- There are a lot of red herrings while using group_by. We should always ungroup for performance and correctness issues. Examples can be checked in the Jupyter notebook.
  - Ungroup for performance
  - Ungroup for correctness
  - Unintentionally overwriting columns
  - Tricky variable scoping(lazy variable naming)

1.2 Joins

- There are various types of joins which can be used in different scenarios. Fortunately, we don't need to implement these joins, as there are functions in R to use them. We only need to learn which join to use in which situation.
- Inner Join
  - Returns the records which match in both the tables in some specific column
  - We can do an inner join on more than one data frames by nesting the joins
  - Symmetric - The order of the arguments doesn't matter
- Left Join
  - Returns all the records of the left table, even if they might not have matches with the right table
  - Not symmetric
- Right Join
  - Complement of left join; returns all the records of the right table, even if they don't have matches with the left table
  - Not symmetric
- Full Join
  - Combines both left and right join in a way; doesn't drop any records and returns missing values for unmatched values
  - Symmetric
- Anti Join
  - Opposite of inner join; shows the rows in first table which don't have a match with the second table
  - Shows what's being dropped; can be checked to see if any rows were dropped from the left table(number of rows in anti-join should be zero)
  - Not symmetric

1.3 Tidyverse

- Data pre-processing is a very important part of data science and one that which takes up a disproportionate amount of time. Tidyr is one R library which helps us in doing that with two functions - spread and gather. Examples can be seen in the Jupyter notebook on Github.
  - Spread - Can convert a "long" table to a "wide" table, by specifying a column as a key and the value column, which returns one column per key, and value filled in the cells
  - Gather - Does the opposite operation of spread; converts a wide table to a long table

2 MapReduce and Hadoop

2.1 MapReduce

- It helps us to solve the split/apply/combine problem in a distributed manner.
- A distributed solution is required because we can scale vertically only up to a point, after which we'll need multiple machines working on the same problem.
- For example, it takes roughly 4 hours to read 1 TB of data from a commonly used hard disk. Using MapReduce, we can sort 1 TB of data in 62 seconds and a 1 Peta Byte of data in 16.25 hours.
- There are three major parts of Map Reduce.
  - Mapper - Transforms the input records to (key, value) pairs
  - Shuffler - Collects all the intermediates records by key, and assigns them to the reducers by the function hash(key) % num_reducers, where num_reducers is the number of reducers that we have. The only priority of shuffler is to make sure that the records for the correct machines.
  - Reducer - Transforms all the records with given key to final output
- Programmer specifies the map and reduce functions, but he doesn't have to worry about fault tolerance, synchronization and other issues which are taken care of under the hood.

2.2 Curse of Last Reducer

- Suppose we're counting the ticket sales of various movies over a weekend, and we make the name of the movie as a key
- There might be one movie which is a blockbuster and has done a great amount of business, whereas the other movies might have done a mild business
- We'll have to wait for a great time until the tickets for the blockbuster movie have been counted, even when all the other movies have already given counts
- In this case, the idea of map reduce breaks down. The reason is the skewness in the data. To rectify this problem, we should have a clever shuffling step in between. We could even give hints to the system by telling it the name of columns which we expect might take more time. Otherwise, we might have to wait for hours.

2.3 Hadoop

- Hadoop is an open-source implementation of the Map Reduce framework. It was named so after a toy of the developer's kid. Hadoop comes along with various utilities/sub-projects like Hadoop Common, Chukwa, HBase, HDFS, Hive, MapReduce, Pig, Zookeeper etc.
- Hadoop streaming is one such utility which allows the user to create and run map/reduce jobs with any executable or script as the mapper and the reducer.

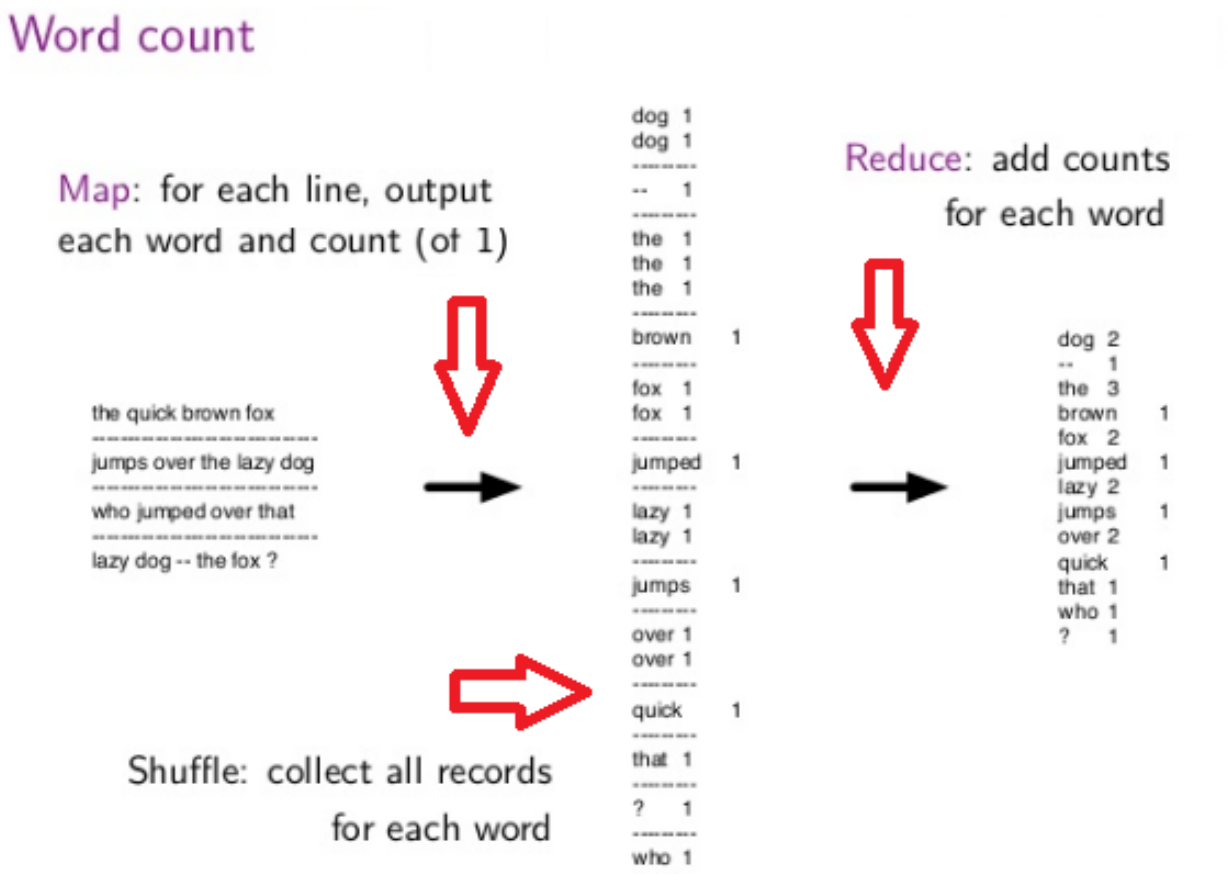


Figure 1: Taken and edited from the slides

- Higher level languages like Pig and Hive provide robust implementations for many MapReduce operations like filter, sort, join, group_by, along with allowing custom made map and reduce operations.

Notes from dmj2130

Lecture 4 can be broadly separated into two parts. The first is an expansion on methods for data preparation and cleaning, which we covered last week. The second expanded on our discussion of counting, talking about how to work on a substantially larger scale than before, using Hadoop and MapReduce.

1 Data Preparation Part 2

1.1 Groupby and Vectorization

We discussed groupby again, which is a function in R that we'll be using quite extensively. As a refresher, it allows you to break up a dataframe into groups, with each containing all the observations that share certain entries. For instance, let's say that you had a dataframe containing information about various people, with columns for gender, eye color, etc. Grouping the data by one of these columns breaks up the data according to the values in that column, so that you can consider the groups separately. For instance, grouping by eye color would create groups for each option–say, a group containing all of the blue-eyed people, another for the brown-eyed, etc. These could then easily be compared separately and concisely. Some important observations:

- Groupby is powerfully combined with the pipe command. For instance,

  ```
  iris %>% group_by(Species) %>% summarize(count=n(),mean_length=mean(Sepal.Length))
  ```

- The majority of important R verbs (like filter) will respect the groups within groupby and process accordingly.
- The dataframe outputted by a groupby operation is modified, containing information about the categories. This makes it simple to perform grouped operations, but can make many operations give unexpected answers or perform poorly when this is not the functionality you are looking for. Be careful about when you are using the grouped and ungrouped tables!

1.2 Pointers and Common Mistakes

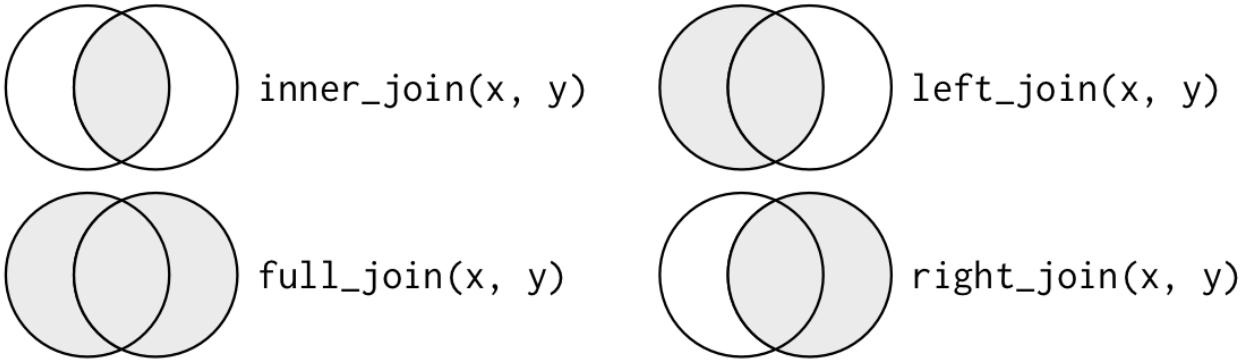
We then talked about some of the basics of R, and how to write effective and functional programs in it. Here are a few tips I thought were worth reemphasizing:

- In data processing, it's often handy to pass the results of one function into another in relatively complex ways. In R, you can do this in the standard way allowed by most programming languages, just nesting the functions within one another. This works fine, but can result in unreadable code if your pipeline is quite long. There is also the %>% function, which acts similarly to a Linux pipe. It normally just pipes the output of the function to its left to the first argument of the function on its right, but you can control the ordering. df %>% f() evaluates to f(df), df %>% f(x, .) evaluates to f(x, df).
- Variable scoping can be a major source of confusion and unexpected behavior. A common error is creating local variables with the same name as columns in a dataframe, which can cause lots of frustration. Avoid this if at all possible!
- Be sure to vectorize your functions if you're writing ones that ought to be applied to columns or rows of a dataframe! Functions designed to process a single item will be mighty confused when passed an array of such items. There's a built in vectorize function in R that you can pass your functions to in order to simplify this process.
- If else statements can often be unsightly in R, especially when applied to vectors. There's an ifelse function that can be of help to you, functioning like a ternary operator. An example to which you can pass a vector of integers, which will provide a vector of strings as a result:

  ```
  ifelse(a%%2==1,"odd","even")
  ```

1.3 Joins

R also includes functionality to allow for SQL-style joins between tables. This is quite important in the real world, as it allows you to combine data from a number of sources into one dataframe. There are several methods for joining data, depending on how you want data that appears in one data set but not the other to be treated. Here's a handy diagram from /r4ds.had.co.nz/diagrams/join-venn.png that I found helpful:



- There's a "by" argument that allows you to specify what columns in each dataframe you would like to join by comparing. For instance, $inner_join(df1, df2, by = c('firstname' = 'namefirst'))$ will perform an inner join on df1 and df2 by comparing column "firstname" in df1 to column "namefirst" in df2. R will try to guess how to perform the join if you don't include this argument, but it's best to keep it for the sake of clarity.
- R also has an anti-join command which can be used to see all of the values that are dropped when performing a join such as an inner join that does not preserve all of the information. This can be useful to ensure that you're not accidentally dropping lots of your information!
- In this class, it's considered bad style to use right join. For the sake of clarity and consistency, it's preferred that you switch the order of the arguments and do a left join instead.

1.4 Spread and Gather

Spread and gather are two functions introduced by tidyr, which allow for you to "widen" or "lengthen" data by splitting up categorical values in rows into columns or by doing the reverse. This is very useful in plotting results. I found a neat and helpful example of spread on the internet (http://garrettgman.github.io/tidying/), which I'll reproduce here. It shows how you can turn the categories in a column full of categorical keys and a data column into two neatly organized columns.

```
table2

##        country year        key      value
## 1  Afghanistan 1999      cases        745
## 2  Afghanistan 1999 population   19987071
## 3  Afghanistan 2000      cases       2666
## 4  Afghanistan 2000 population   20595360
## 5       Brazil 1999      cases      37737
## 6       Brazil 1999 population  172006362
## 7       Brazil 2000      cases      80488
## 8       Brazil 2000 population  174504898
library(tidyr)
spread(table2, key, value)
##       country year  cases population
## 1 Afghanistan 1999    745   19987071
## 2 Afghanistan 2000   2666   20595360
## 3      Brazil 1999  37737  172006362
## 4      Brazil 2000  80488  174504898
```

Gather will do the reverse operation, converting specified columns into a categorical variable. It takes an additional argument, the indexes of the columns, to determine which columns should be converted into categories. Here's an example from the same site of gather in action:

```
table5  # population

## Source: local data frame [3 x 3]
##
##       country       1999       2000
## 1 Afghanistan   19987071   20595360
## 2      Brazil  172006362  174504898
## 3       China 1272915272 1280428583

gather(table5, "year", "population", 2:3)

## Source: local data frame [6 x 3]
##
##       country year population
## 1 Afghanistan 1999   19987071
## 2      Brazil 1999  172006362
## 3       China 1999 1272915272
## 4 Afghanistan 2000   20595360
## 5      Brazil 2000  174504898
## 6       China 2000 1280428583
```

2 Intro to MapReduce and Hadoop

2.1 MapReduce

MapReduce is a distributed solution to computational problems exceeding the processing power of a single machine. It has become extremely prominent within computer and data science in the last decade or so, as it is abstract enough to be applied as a framework to a large variety of parallelizable tasks. Essentially, it asks the programmer to implement a map function, which transforms the data into key-value pairs, as well as a reduce function, which lays out the computations that ought to be performed on each group. There's also a shuffle record that works behind the scenes, to collect all of the records with the same key together for the reduce function. This is advantageous if the processing takes a long time or if the disk access takes a long time, since both can be done in parallel. Some important points:

- Fault tolerance and synchronization are implemented under the hood, so the programmer doesn't need to worry about them when working on a new task.
- MapReduce is designed to scale elegantly and transparently.
- Strongest on tasks that are reading and bandwidth intensive.
- Batch, offline, simple tasks are most frequently implemented in this way.
- Some implementations can struggle when the tasks assigned to each computer take dramatically different amounts of time to finish.
- MapReduce is useful for a great many tasks! But it's not always the right answer.

2.2 Hadoop

Hadoop is one of the most popular implementations of MapReduce principles. It's an open source program, which allows for existing code to be easily incorporated into a parallel structure. In many cases, it can save you from writing an entirely new program, since Hadoop as a streaming feature which allows you to just specify existing functions (like Unix builtins) as mappers and reducers. In the slides, there's an example where it's shown that a word counting program can be implemented in Hadoop without writing any Java code, using two lines of code and the Hadoop Streaming utility. Here's a great example taken from the slides, illustrating how easy it is to configure a program that would run locally to run in a distributed fashion:

wordcount.sh

Locally:

```
# cat data | tr " " "\n" | sort | uniq -c
```

⇓

Distributed:

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming.jar \
    -input README.txt \
    -output wordcount \
    -mapper 'tr " " "\n"' \
    -reducer 'uniq -c'
```

Though using Hadoop and MapReduce concepts in traditional languages such as Python and Java is straightforward, there are also a number of high-level languages that were designed with bulk data processing in mind. Hadoop's team makes one called Pig that implements many common data analysis tasks as MapReduce jobs and is usable both on a single computer and in a larger production environment. We also mentioned Hive, another language for large scale data processing that is similar to SQL.

Notes from jl3825

1 Data Manipulation in R [Part 2]

1.1 Operations

- **Group-by** allows you to perform operations by group. For example, if we group students by gender, instead of computing an average grade for an entire class, we can compute the average grade for boys and for girls.
- **Summarize** allows you to make aggregations of data. It collapses a data frame to a single row.
- **Mutate** allows you to add new columns based on the existing ones. Notably, the operations here are automatically vectorized (you don't need to write your own for-loops to do operations on each corresponding set of data).
- While it is possible to chain all of your operations without the **pipe operator(|)**, using the pipe operator can make these operations easier to read.

1.2 Dplyr Tips

Common Logical Errors:

- **Remember to ungroup** your data when you're done working with the groups.
- **Be descriptive with column names** or else run the risk of unintentionally overwriting existing columns or accessing the wrong column.
- **Be careful about variable scope.** Lazy variable naming (for example, using a column name as a variable name) can result in logical incorrectness. Note that, duplicate variable names will access the variable with the more-inner scope, meaning the variable with the global scope is accessed last.
- **Remember to vectorize functions.** Unvectorized functions will still return values; however, they will return one result for the first elements of the vectors, which are then copied by Dplyr. You can do $g = vectorize(f)$ to remedy this issue.

Common Performance Errors:

- **Forgetting to ungroup.** Group-by modifies the dataframe, so operations actually loop within each group, which can get expensive due to the constant overhead for each loop. In class, the sample 1 million operations took 2 seconds when grouped vs. 0.007 seconds when the operation was fully vectorized.

1.3 Joins

Reviewing the different types of joins.

- **Inner-join:** takes the matching values from the common columns. Inner-joins are symmetric. Note: if a table has more than one column of the same name, it will join on all of them. In addition, while joins can be done on more than two data frames at a time, generally this is a nested operation.
- **Left-join:** returns an Inner-join as well as all of the remaining rows from the left data frame that don't have a corresponding row in the right data frame. Easy way to think about left-joins: left-joins are guaranteed to have the same number of rows in the output data frame as in the left input data frame. This operation is not symmetric.
- **Anti-join:** shows the difference (what gets dropped) when a Left-join is performed. Anti-joins are helpful to use as sanity checks to make sure you aren't losing useful data. This operation is not symmetric.

- **Right-join:** The opposite of a Left-join. It keeps all unique keys from the right column. User tip: Always use Left-join and put the feature, whose keys you wish to maintain, in the left column. Left-join(df1, df2) = Right-join(df2, df1).
- **Full-join:** ensures that the join maintains the union of all input keys. Will add an NA whenever there is a missing value for some key.

1.4 Re-shaping Data

- **Spread:** creates a wide version of your data frame. Takes two columns (key and value) and spreads to multiple columns.
- **Gather:** makes "wide" data longer. Takes multiple columns and gathers them into key-value pairs.

2 Hadoop

2.1 What?

Hadoop is an open-source software framework that supports the processing and storing of large data sets in a distributed computing environment. Our work with Hadoop is centered around MapReduce, which is essentially a distributed group-by operation. It provides us a distributed system for solving the split/apply/combine problem at scale.

2.2 Who/When?

- Doug Cutting and Mike Cafarella create Hadoop to support distribution for Nutch in mid-2000s -¿ Hadoop becomes offical project
- Google: internal MapReduce programming model

2.3 Why?

- Much faster for high volume data. E.g. It takes 4 hours to read 1 TB from a commodity hard disk, but takes Hadoop 62 seconds to sort the same amount of data.
- **Typical use case:** "How many search queries match 'icwsm', by month?"

2.4 How?

MapReduce takes any key, hashes it to a number, and then mods it by the number of reducers. It is designed to be generic, meaning that programmers only need to specify the map and reduce functions and the rest is handled internally. Allows programs to scale transparently with regard to the size of the input. Emphasizes local computation (bringing the code to the data, instead of moving data around).

- **Map:** transforms input record to an intermediate (key, value) pair.
- **Shuffle:** collects all intermediate records by key.
- **Reduce:** transforms all records for a given key to final output. Record assigned to reducers by $hash(key)\%num-reducers$

2.5 Pros/Cons

- **Strengths:** batch, offline jobs; simple computations (not computation-bound, but I/O bound); usually, write-once
- **What It's Not:** not a lot of synchronization, not a low-latency random access relational database.

2.6 Hadoop Streaming

Because writing Java is no fun!

- Allows programmers to create and run map/reduce jobs with any executable or scrip as the mapper and/or the reducer.
- Hadoop handles the rest (note: you still need to make sure your dependencies are packaged into the executable)
- MapReduce for *nix geeks: *"# cat data | map | sort | reduce"*

Notes from sm3604

1 What is Hadoop?

- An abstraction for parallel data analysis
- consists of many subprojects
  - we will mostly focus on Map/Reduce
- deals with distributed data
- born out of open source web indexing, crawling, and searching software
- Map/Reduce revealed by Google in 2004
  - added to Hadoop, which is adopted by Yahoo! in 2006

2 Why do we need Map/Reduce?

- There is still a lot of latency when dealing with a lot of data
- Read speed of commodity hard disk is about 1 TB/4hrs
- Using Hadoop, 1PB can be sorted in 16.5 hours! petabytesort

3 Map/Reduce

- break into parts
- process in parallel
- combine results

For example, if we wanted to count the number of occurences of each word in a book:

1. For every word on every page
2. Map to (word, count) e.g. ("cat", 1)
3. Shuffle to collect all records with the same key (word)
   hash(val) = hashVal mod number of reducers
4. Reduce results by adding count values for each word

To use Map/Reduce, you must specify the Map and Reduce steps

4 Principles

- move code to data
- allow programs to scale transparently

5 Strengths of Map/Reduce

- batch, offline
- write-once read-many
- simple computations
- I/O bound by disk or network bandwith

6 Weaknesses

- does not work well for high performance parallel computing applications
- low latency on random access
- not always the right solution
- difficult to find points of failure

Before you use Map/Reduce, you should make sure it's the right tool for the job.

7 Beware the Curse!

- often there is an assumption that all machines in the Map/Reduce pipeline end up with approximately the same amount of work
- this is not always true
    - not the case when working with skewed data
    - can do clever shuffles to avoid this if there is prior knowledge of skewness in the data

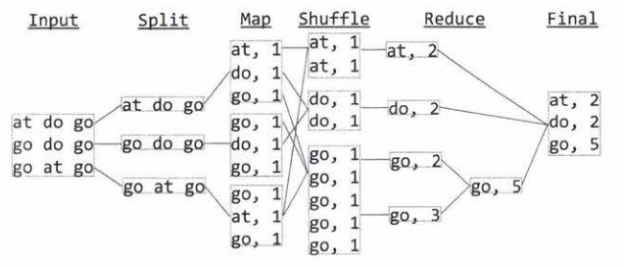


Figure 2: In this example, we see a graphical representation of a word count program using Map/Reduce. The Map step emits tuples of (word, count). The Shuffle step brings all corresponding words together. The Reduce step adds the counts of all the corresponding words. In the final step we have a list of tuples, each with its own word and the count of that word in the original input.

8 Examples

Below are two different examples of MapReduce, taken from "Cracking the Coding Interview, 6th Ed." by G. McDowell.

8.1 WordCount

See above.

8.2 Average Temperature

Say you are given a list of data in the form (City, Temperature, Date). You wish to find the average temperature for each city in a given year. How would you set up the Map and Reduce steps to perform this calculation?

- **Map**: Given a data point, the Map step will emit a (Key, Value) tuple of the form (CityYear, [Temperature, N]). The "Key" will be a particular city and year. The "Value" will be a tuple of Temperature and N, where N indicates the associated temperature is the average of N data points. This will be important in the Reduce step.
- **Reduce**: Given a list of emitted values, the Reduce step will take the weighted average of temperatures for each CityYear, where the weights will be N.

Notes from zq2144

1 How to Make Scribed Note on Github

1. Fork the repository from Course Github
2. Pull the repository to your own machine
3. Add commit on your own machine
4. Push your notes onto your Github
5. Make pull request on Course Github

2 Homework 1

due at 02/23/2017 (Markdown, Notebook, etc. are all acceptable)

1. Problem 1: No coding, just thinking and writing
2. Problem 2: counting exercise(do not send back all data)
3. Problem 3: reproduce the plot

3 Review of the Last Lecture(split/apply/combine with group_by)

To make code more readable, we tend to do group_by and summarize separately. Filter can help you choose a specific row/ column. Mutate can help you create a new column in a specific way easily (avoid complex loops, etc.). Arrange(desc) means to arrange them in descending order.

4 Warnings

1. group_by has side effects, so always ungroup intermediate results
   It's common to store grouped data in an intermediate variable and do something with it downstream. Forgetting about the implicit grouping downstream can lead to unexpected issues, both in performance and correctness of code. For example, when we filter with groups, it still has linear time, but the constant matters. So filter at this time will take so much time. If we mutate with groups, it will make loop within each group which may output wrong answers. So, the solution is to always ungroup when creating intermediate tables.
2. Arranging within groups
   Arrange works within groups, reordering rows on a per-group basis. But it doesn't display the output sorted by group. So we can use the group variable(s) as the first argument(s) to arrange.
3. Unintentionally overwriting columns
   Sometimes, we get lazy about variable name and overwrite things, which may cause mis-use in the following steps.
4. Tricky variable scoping
   For example, do not name a vector and a value with the same name.
5. Forgetting to use vectorized functions
   Simple functions usually operates on one value, which means they can not be operated on a vector (mutate, filter, etc.). So, if we want to use it on a vector, we need to use Vectorize() to vectorize it.

5 Joins

tible is like a table but nice in printing and something else

1. inner_join: like a intersection, this function just remains the matching value and drop something which do not match. Besides, we can also ask it to join by which column. For example, Inner_join(df1, df2) equals to inner_join(df2,df1) with different order.
2. left_join: keep all unique values in the left table.
3. anti_join: it will give us what we throw out in the inner_join.
4. right_join: keep all unique values in the right table. But please always use left_join, which means to keep what we care in the left table.
5. full_join: keep all values in both tables

6 Spread and Gather

1. reasons to use spread and gather: When dataset is not in a reasonable form; Or when we want to plot the data in a specific way.
2. spread: spread all values in one column to several new columns and use values in another old column to fill the new table.
3. gather: it collects a set of column names and places them into a single "key" column. It also collects the cells of those columns and places them into a single "value" column. The "key" column and the "value" column are all newly built.

tips: When choosing columns, "-" Means "but".

7 MapReduce

1. Brief History
   Pre-2014: open projects for web-scale indexing, crawling, and search
   2004: MapReduce was used internally at Google
   2006: Hadoop became official Apache project. Cutting joins Yahoo! and Yahoo! adopts Hadoop
2. Why another language/Why a distributed solution: We need so much time(few hours) to read like 1Tb from a commodity hard disk. At the same time, MapReduce can sort 1Tb data in 62 seconds.
3. How MapReduce works
   Map: matching records from different machines to like (YYYY/MM, count = 1)
   Shuffle: to collect all records with same key (just put it to the right machine, no sort)
   Reduce: results by adding count values for each key (merge sort is happening in each of the machine in this step)
   **We only care about Map and Reduce. Shuffle will be done automatically.
4. Weakness of MapReduce
   High-performance parallel computing
   Low-latency random access relational database (CPU capacity)
   Always the right solution
5. Word Count in MapReduce
   Map: for each line, output each word and count (of 1) (wasteful and silly)
   Shuffle: collect all records for each work
   Reduce: add counts for each words
   **Never do word count in Java...

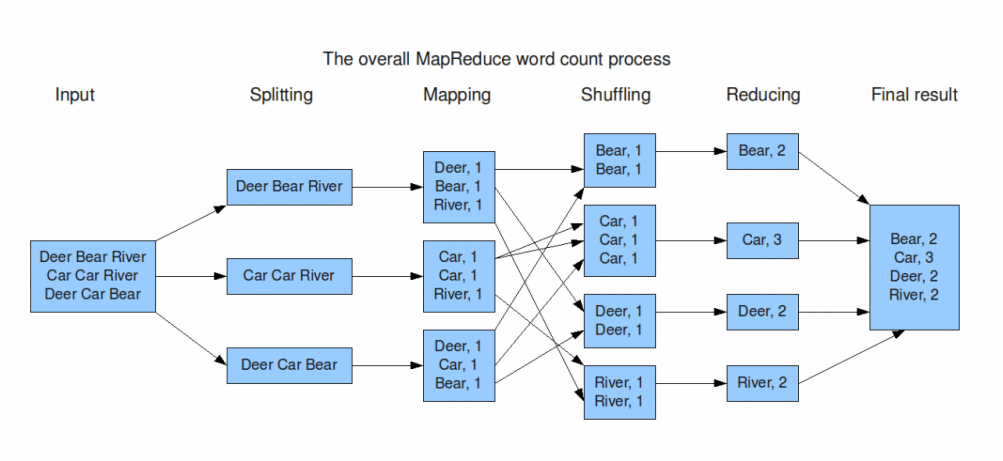


Figure 3: The Overall MapReduce Word Count Process (from http://www.learn4master.com/big-data/what-is-mapreduce-and-how-it-works)

6. Haddop Streaming Introduction: Hadoop Streaming is a generic API which allows writing Mappers and Reduces in any language. Mappers and Reducers receive their input and output on stdin and stdout as (key, value) pairs.
   Example:
   MapReduce for *nix geeks:
   $\#catdata|map|sort|reduce$

7. A Little Pig/Hive (like SQL)
   Differences between Pig, Hive and SQL: what we want to do in SQL is all in one long command. In Pig, we can do it in intermediate way. Hive might be quicker than SQL when doing joins.
   Group in Pig:
   It will give you the answer like word count in MapReduce [Join (JOIN A BY a1, B BY b1)]
   **How we do Join in MapReduce:
   A(1,2,3) $\rightarrow$ [1,(A(1,2,3))]
   B(1,3) $\rightarrow$ [1,(B(1,2,3))]
   $\rightarrow$ reduce (hash join/mergesort join/.....)