Lecture 4: Counting at Scale: MapReduce, Spring 2017 Columbia University

Jason Lei

February 10, 2017

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"*