Assignment 0 – Hadoop Setup

For the first part of the semester, your programming assignments will use Hadoop. In order to do the assignments you need to either:

1. Set up an account on a Hadoop cloud service, or
2. Create a local install of Hadoop on your personal machine and manage Hadoop yourself.

For cloud-based Hadoop, I will be demonstrating use of the AWS (Amazon Web Services) Hadoop offering called Elastic Map Reduce (EMR). You can create an AWS account and get credits so that you can use the service for free.

**Amazon Web Services**

Here are the options for academic usage of AWS as described by AWS Support.

All AWS-approved accredited educational institutions, educators, and students will need to apply to Educate individually. Students have two options:

* **Educate with an AWS account**
  + The student is able to activate an AWS account which requires a valid credit card
  + Once an application has been approved, Educate credits will be provided to each student via email and will need to be redeemed manually to their AWS account.
  + Any charges accrued that are not covered under any available credits will be the responsibility of the account holder. Note: This means your credit card will be charged.
  + New AWS accounts will be eligible for Free Tier Promotions for the first 12 months.
  + To see what's covered under Free Tier: [https://aws.amazon.com/free/ (Links to an external site.)Links to an external site.](https://aws.amazon.com/free/)
* **Educate Starter Account**
  + Offers free access to a specified, capped amount of AWS cloud resources without requiring a credit card for payment. Once credits have been exhausted, the student is no longer able to utilize services.
  + Unlike a regular AWS account, which is run and managed by you, the AWS Starter Account is run and managed by qwikLABS.
  + Credits will be added automatically to their qwikLABS account, so no credit codes will need to be redeemed.
  + To see what's included with the Starter Account
    - [https://s3.amazonaws.com/awseducate-starter-account-services/AWS\_Educate\_Starter\_Accounts\_and\_AWS\_Services.pdfLinks to an external site.](https://s3.amazonaws.com/awseducate-starter-account-services/AWS_Educate_Starter_Accounts_and_AWS_Services.pdf)

Students will only be approved for one option or the other but not both.

To view our FAQ's: [https://www.awseducate.com/apex/FAQs#fa0Po00000043dVbEAILinks to an external site.](https://www.awseducate.com/apex/FAQs#fa0Po00000043dVbEAI)

To apply: [https://aws.amazon.com/education/awseducate/Links to an external site.](https://aws.amazon.com/education/awseducate/)

Here are some step-by-step instructions for setting up your AWS account: [http://docs.aws.amazon.com/ElasticMapReduce/latest/ManagementGuide/emr-gs-prerequisites.htmlLinks to an external site.](http://docs.aws.amazon.com/ElasticMapReduce/latest/ManagementGuide/emr-gs-prerequisites.html)

**MicroSoft Azure**

MicroSoft also offers a cloud-based Hadoop service in their cloud offering called Azure. If you are interested in using Hadoop via Azure, you can obtain an account as described here: [https://azure.microsoft.com/en-us/free/Links to an external site.](https://azure.microsoft.com/en-us/free/)

**Local Install of Hadoop**

If you are interested in setting up a local install of Hadoop, see: [https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html (Links to an external site.)Links to an external site.](https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html)

Note: Using a local install of Hadoop will provide you with faster runs of your programs, as cloud-based services often take several minutes to start-up. But a local install requires you to manage the Hadoop install and configuration yourself, which is extra work.

Assignment 1 – Word Count

**Goals:**

* Compile a Java map-reduce program
* Create a JAR file containing the map-reduce code, ready for upload to AWS or Hadoop cluster
* Login to your Hadoop cloud service (if not using a local install)
* Create sub-folders in your account for jar files, logs, inputs, and outputs. I recommend "jars", "logs", "data", and "output".
* Upload the JAR file
* Define a cluster and job steps, and run the map-reduce job
* Examine files generated by the map-reduce job execution
* Download the output

In this assignment you will learn how to start with Java source code and end with the output generated by that map-reduce job running on AWS (Amazon Web Services). The Java source code and a Maven pom.xml file are provided. Although you are not creating Java code in this assignment, I encourage you to set up the environment you will be using to edit and compile your Java code, and import the example code there. I recommend using an IDE such as Eclipse (open source) or Intellij IDEA Community Edition (free) or Student Edition (free). Also make sure that you have the capability to run the maven script (pom.xml) that can generate your JAR file for each assignment, as this will be a required artifact that you must submit for assignments. This allows Vivek and I to execute and evaluate your code.

**Compiling Java code, building the uber JAR.**

The Java source code for WordCount and the pom.xml file used to build are available under Files / Assignment 1. I recommend that you get the file assign1.tar. This file contains the Java source code for WordCount (and WordCountTest) and the pom.xml file. If you untar this file (command: tar xvf assign1.tar), the Java source files and pom.xml file will be in the appropriate directories for compiling with maven, and for creating a project in Eclipse or IntelliJ. When creating a project in either of these IDEs, simply import the pom.xml file to create the project.

Once you have untar'd assign1.tar, you can compile the Java code and create the uber JAR for upload to AWS. The command is:

mvn clean package

The generated JAR file is in the target subdirectory.

**Using AWS:**

Login to AWS Management Console, select the S3 service. (Note: You can move this and other service icons to the toolbar if you like.) Click on the bucket name. The display now shows the folders in the S3 bucket. Click the "Create Folder" button and create folders as desired. I recommend you create subfolders named: data, jars, logs, and output. You will use these folders for:

* data - uploading data files you create to test your map-reduce program.
* jars - uploading your JAR files
* logs - capturing the logs of a map-reduce job execution - useful for debugging and understanding the execution of your map-reduce job
* output - capturing the output of your map-reduce job

Now you're set up to upload your JAR file, define a cluster and job step, and run a map-reduce program. Click on the jars folder. Once in this directory, click on the Upload button. Find your JAR file and upload it to S3. If you have a data file for testing, you can upload it to the data folder in a similar manner. For this assignment, you should use the data file dataSet1.txt (available on Canvas) to produce the output to submit to Canvas.

Now let's define a cluster and job step to run your map-reduce program. Select the EMR ("Elastic MapReduce") service. You may notice several clusters listed there. If no clusters are visible, you'll see Welcome to Amazon Elastic MapReduce page. In either case, click the "Create Cluster" button.

If your page starts with "Create Cluster - Quick Options", click on the text "Go to advanced options" just to the right.

* **Step 1: Software Configuration:**
  + Step type: Select "Custom JAR", and click the Configure button.
  + In the "Add Step" dialog box:
  + Name: the name of this step. I suggest something like Assign1Run1 (you'll likely have multiple steps on future assignments).
  + JAR Location: s3://yourBucketName/jars/bdp-0.1.jar
  + Arguments: This map reduce job takes three arguments:
    - Full pathname of the class: com.refactorlabs.cs378.assign1.WordCount
    - The input file: s3://yourBucketName/data/dataSet1.txt
    - The output folder: s3://yourBucketName/output/Assign1Run1
  + Action on failure: Leave as "Continue"
  + Click "Add".
  + Note: If the output folder already exists, the job will fail, so you need to specify a new folder each time for the output, or delete the output folder before running the job.
  + Vendor: Amazon Release: emr-5.8.0
  + Applications: Hadoop 2.7.3. Unselect another other software packages, as they only slow down the cluster startup process.
  + Edit software settings (optional): nothing to enter here
  + Add steps (optional): Here is where you define the details of the map-reduce job to run.
* Auto-terminate: I recommend that you say "No" here. The reason being that this terminates your cluster when the step completes, whether or not your map-reduce job was successful. In most cases you'll want to keep the cluster up and running so you can add another step (after uploading a new JAR file that contains code changes, or running our map-reduce job on a different data set). Cluster startup is somewhat slow, and this will improve turn-around time when you are debugging your code. Also, the minimum charge for any job is one hour of whatever resources are used. So if you run for 10 minutes or 58 minutes, you are charged for one hour. If you need to re-run the job, it's easy to add a step and run that step (remember to delete the output folder if you reuse it in another step). In keeping the cluster running, you don't want to inadvertently leave it running for many hours and run up the charges on your account.
* **Step 2: Hardware Configuration:**
  + Leave "Uniform Instance groups" selected
  + Network: no change
  + EC2 Subnet: no change
  + Root device EBS volume size: no change
  + Master - Click on the edit icon (a pencil) next to the Instance Type, "m3.xlarge". I recommend selecting "m1.medium", as this is sufficient for the jobs we will run, and uses minimal resources so the usage charge is minimized.
  + Core - Set the instance count to 0
  + Task - Set the instance count to 0
* **Step 3: General Options:**
  + Cluster name: Give the cluster a name (for example, Assign1 Run1)
  + Logging: Enable logging, and enter the pathname of your log folder. Assuming you created a folder named "logs", the logging folder S3 location will look like: s3://yourBucketName/logs. Log files will be placed under a folder there with a unique name assigned to the mapreduce job by Hadoop.
  + Debugging: Unselect
  + Termination Protection: Unselect
  + Scale down behavior: No change.
  + Tags: Nothing to do here.
  + Additional options: Nothing to do here.
  + Bootstrap Actions: None.
* **Step 4: Security Options:**
  + EC2 key pair : Select the EC2 key you created with your account (not required).
  + Permissions: Nothing to do here.
  + Finally, click on Create Cluster at the bottom right, and your cluster will start up. Once it has completed provisioning and run bootstrapping code, your map-reduce job will run. Assuming that your job is successful, you can examine the log files and download and examine the output.

**Artifacts to submit:**

* Assignment1Build.zip or tar - all files (Java, pom.xml) in the directory structure required by maven and buildable with your pom.xml file.
* Assignment1Code.zip - all files (Java) in a flat directory for easy inspection for grading
* Assignment1Output.txt - output file containing the word counts

Assignment 2 – Custom Writable

For assignment 2, the task is to compute the mean and variance for each word that appears in one or more paragraphs of the input document. Each line in the input file represents one paragraph.

* Input file: dataSet2.txt
* Output file content: Document count, mean, and variance for the words that appear in the document.
* Output format: Text file with fields: word, \t, paragraph (document) count, mean, variance.

**Required elements:**

* Name your class WordStatistics.
* Implement a custom class that implements the Writable interface (name this class WordStatisticsWritable), and use this class for map output, and reducer input and output.
* Implement and use a combiner.
* Remove punctuation (commas, periods, double quotes, ....) and convert the text to lowercase. I suggest that you first run the WordCount program on dataSet2.txt to see what punctuation exists in the document. This document contains footnote citations like this:  "[174]".  Consider these as words.

**Major Hints:**

Declaration of the map class for WordStatistics should look like this:

... extends Mapper<LongWritable, Text, Text, WordStatisticsWritable>

Declaration of the reduce class for WordStatistics should look like this:

... extends Reducer<Text, WordStatisticsWritable, Text, WordStatisticsWritable>

**Artifacts to submit:**

* Assignment2Build.tar - all files (Java, pom.xml) in the directory structure required by maven and buildable with your pom.xml file. ZIP is OK is you don't have tar.
* Assignment2Code.zip - all files (Java) in a flat directory for easy inspection for grading
* Assignment2Output.txt - output generated by your WordStatistics map-reduce program

Your Java code should include comments that describe your approach, so that it can be read and easily understood for grading.

**Extra Credit** (2 points):

In addition to computing statistics for the words in the input, also calculate the mean and variance for paragraph length. For the extra credit, use the following files as input:

* dataSet2Xa.txt
* dataSet2Xb.txt

To specify multiple input files, where you would list a single file as the input to a map-reduce job, give both filenames separated by a comma, like this:

* s3://bucketName/data/dataSet2Xa.txt,s3://bucketName/data/dataSet2Xb.txt

To read multiple input files, you'll need to make this change in the main() method of WordStatistics (I suggest you use this in all future mapReduce programs):

FileInputFormat.addInputPaths(job, appArgs[0]);

Additional artifact to submit:

Assignment2XOutput.txt - output generated by your WordStatistics map-reduce program using extra credit input.

Assignment 3 – Inverted Index

Created an inverted index as described in class (and in the textbook), where the output is a file of key/value pairs, with the key being an email address and the reference type (To, From, Cc, Bcc), and the value being a list of email IDs in which that address was referenced in the email.

For this assignment you will need to parse the emails in the input file (one email per line in the input file, separated by newlines).  You should only be concerned with parsing the header of the email (fields like To, From, Cc, ...), and you need not do anything with the email content or embedded/forwarded emails other than skip over them. In the data set (two files) the emails have these fields in the header:

* Message-ID:
* Date:
* From:
* To:
* Subject:
* Cc:  [optional]
* Mime-Version:
* Content-Type:
* Content-Transfer-Encoding
* Bcc:   [optional]
* X-From:
* .....

**Input files:**

dataSet3a.txt, dataSet3b.txt

**Artifacts to submit:**

* Assignment3Build.tar - all files (Java, pom.xml) in the directory structure required by maven and buildable with your pom.xml file. ZIP is OK is you don't have tar.
* Assignment3Code.zip - all files (Java) in a flat directory for easy inspection for grading
* Assignment3Output.txt - output generated by your InvertedIndex map-reduce program

Example line for the output file:

Bcc:ceb@tozzini.com.br   <17233551.1075849805047.JavaMail.evans@thyme>,<32775907.1075849805605.JavaMail.evans@thyme>

**Bonus points** (2 pts. each)

* List of message IDs for an email address has no duplicates
* List of message IDs for an email address is sorted (lexicographical order).

Assignment 4 – Using AVRO

The goal of this assignment is to familiarize yourself with:

* defining objects using AVRO
* generating the Java bindings
* building/using/referencing AVRO objects in your map-reduce code
* understanding output of AVRO defined objects

In this assignment you will:

* define an object (using AVRO) that contains the data produced by the WordStatistics app:
  + document count
  + total count
  + min
  + max
  + sum of squares
  + mean
  + variance
* Call this AVRO defined object WordStatisticsData
* Modify your WordStatistics app from Assignment 2 to use WordStatisticsData, and collect stats for words and paragraph length, including min and max (word count in a paragraph, or paragraph length)

Use the example code from WordCountA.java as your guide for using an AVRO defined object. This file is available on Canvas (Files / Assignment 4).

For maven to see your AVRO schema file (file suffix is:  .avsc), you need to place it under the directory: src/main/avro

 A new pom.xml file is available that defines the AVRO dependencies and AVRO version you should use. Use this pom file, as AWS Hadoop and AVRO are sensitive to version differences, and these cause you lots of headaches if you get them wrong. The first time you build with maven, the required AVRO jars and other jars AVRO depends on will be retrieved. This may take a few minutes, as there are quite a few new jar files.

**Required elements**:

* Define an AVRO object named WordStatisticsData
* Mapper output types: Text, AvroValue<WordStatisticsData>
* Reducer output types: Text, AvroValue<WordStatisticsData>
* Output file format: TextOutputFormat
* Use dataSet2Xa.txt and dataSet2Xb.txt for input
* As we did in Assignment 2:
  + Convert messages to lower case
  + Compute the message length statistics.
  + You will be able to compare your results (mean and variance) with the results for Assignment 2 extra credit (available on Canvas).

Use example code in WordCountA as your guide (available on Canvas: Files / Assignment 4).

**Artifacts to submit**:

* Assignment4Build.zip or .tar - Java code and AVRO schema (in the directory structure recognized by the pom.xml file), and pom.xml.
* Assignment4Code.zip - Java code and AVRO schema (all files in one directory).
* Assignment4Output.txt - output generated by your solution.

**Files available on Canvas (Files / Assignment 4)**:

* pom.xml
* WordCountA.java
* WordCountMapper.java
* wordCountData.avsc

**Notes on Using AVRO.**

Useful links:

[http://avro.apache.org/docs/1.7.4/spec.htmlLinks to an external site.](https://avro.apache.org/docs/1.7.4/spec.html) [http://avro.apache.org/docs/1.7.4/gettingstartedjava.htmlLinks to an external site.](http://avro.apache.org/docs/1.7.4/gettingstartedjava.html)  
[http://avro.apache.org/docs/1.7.4/api/java/index.htmlLinks to an external site.](http://avro.apache.org/docs/1.7.4/api/java/index.html)

If you get an error message like this:

java.lang.Exception: java.lang.NoSuchMethodError: org.apache.avro.generic.GenericData.createDatumWriter(Lorg/apache/avro/Schema;)Lorg/apache/avro/io/DatumWriter;

this indicates a mismatch between the AVRO version you compiled against, and the AVRO version in your classpath (the version used at runtime).

Assignment 5 – User Sessions

For Assignment 5, you'll define an Avro object that represents a user session. Your mapReduce job will read in individual log entries, create an Event object that represents each log entry, and assemble these into user sessions. A session should be created for each unique userId, as the userId identifies activities by the same user.

You should download dataSet5Small.tsv and dataSet5Header.tsv from Canvas and examine the content to familiarize yourself with what is represented in a log entry. The log files are tab separated (.TSV), and dataSet5Header.tsv defines the fields of an individual log entry.

Your first task is to run WordCount against dataSet5a.tsv and dataSet5b.tsv to get a list of the values in the various fields. The following fields have a large number of unique values, so you need not explore the values for these fields (they'll result in a large amount of output):

* event\_timestamp
* mileage
* price
* user\_id
* vin

Modify your WordCount app to output the following info:

fieldname:value    count

This will show you the values that occur in each field including which fields can have no value or a null value.

Include all the fields in your Avro object. A starting definition for the Avro schema has been provided (session.avsc, on Canvas).

The event\_type field of a log entry should be broken into two fields (both enums) in your Avro schema: event\_type and event\_subtype. The different event\_type values are already defined in session.avsc. You need to fill out the possible values for the event\_subtype, which is determined by the remainder of the string value in the log entry event\_type field.

Create enumerations for these fields:

* body\_style
* cab\_style
* vehicle\_condition

Other fields will be strings, numbers, or booleans as determined by the data. Fields with values "t" or "f" should be represented as boolean.

Events for a single user\_id are grouped into one session, and they should be in order of event\_timestamp.

**Input**: dataSet5a.tsv,dataSet5b.tsv

**Outputs**:

* Field value counts using your modified WordCount program
* Avro text representation of user session instances your map-reduce job creates (use TextOutputFormat)
* Key and value types for the session generation reducer output:
  + AvroKey<CharSequence>  See WordCountB for an example
  + AvroValue<Session>

**Required elements**:

* Modified WordCount app that outputs the counts for each unique value of each field (ignore the fields listed above).
* Avro schema representing a user session.
  + Include all fields defined in dataSet5Header.tsv
  + User sessions have an array of events, sorted by event time.
* The values in the features field (":" separated) should be extracted and placed in an array, in sorted order.
* The fields body\_style, cab\_style, and vehicle\_condition should be enums.
* The field free\_carfax\_report should be boolean.
* Key and value types for the session generation reducer output:
  + AvroKey<CharSequence>
  + AvroValue<Session>
* Remove duplicate events.

**Artifacts to submit**:

* Assignment5Build.zip or tar - all files (Java, avsc, pom.xml) in the directory structure required by maven and buildable with your pom.xml file.
* Assignment5Code.zip or tar - all files (Java, avsc) in a flat directory for easy inspection for grading
* Assignment5Output.zip or tar containing:
  + output from the modified WordCount app
  + output from session generator app - text representation of your Avro user session objects

Assignment 6 – Reduce Side Join

For this assignment, we'll join two different data sources using a reduce-side join. You'll use MultipleInputs for this assignment, with a mapper defined for each data source.

The first input will be user sessions like you created in assignment 5. First, modify your assignment 5 code to output an AVRO container file (binary) by using the output file format: AvroKeyValueOutputFormat

Output types for your reducer should be:

AvroKey<CharSequence>, AvroValue<Session>

Note: Substitute your session class name for "Session" if you used different name.

The first mapper will read the AVRO container file with your sessions. You'll need to specify AvroKeyValueInputFormat as the input format in order to read this file. Mapper input types must match the output types you used to write the AVRO container file:

AvroKey<CharSequence>, AvroValue<Session>

The second mapper will read a CSV file with the following content and format:

vin,impression\_type,count

where VIN is a string like "1J4FA49S41P344436", impression\_type is one of these two values (strings), "SRP" or "VDP", and count is an integer representing the number of unique users that viewed the VIN either on a search results page (SRP) or a vehicle detail page (VDP) on car search web sites.

Both mappers should output data using the AVRO schema I've provided (see the file vinImpressionStats.avsc on Canvas). The output types for both mappers will be:

Text, AvroValue<VinImpressionCounts>

The reducer will combine (join) the data from each data source using the VIN as the common key. The join operation should sum the counts from each source (sessions, or vinImpressionStats) and then combine the sums in an instance of the AVRO defined class VinImpressionCounts.

**The AVRO object VinImpressionCounts**

The first mapper should read your session file and write one instance of VinImpressionCounts for each VIN in the session. The fields should contain the following data:

* unique\_users - Counts the number of user sessions that reference this VIN.
* clicks - For each event of type "CLICK" for this VIN, add the event subtype to this map (with count 1 for the current user - we are counting unique users, not total clicks).
* edit\_contact\_form - For each event type/subtype EDIT/CONTACT\_FORM, count 1 for the current user if this event occurred at least once in the session for the VIN.

The second mapper will take the count field from its input file and record it in the marketplace\_srps or marketplace\_vdps field. The key output by both mappers is the VIN.

**Reduce step**

The particular join we are doing is left outer join, where your session data is considered the left input. There might be VINs in the session data (left side) that do not appear in the VIN impression data (right side), and there will be VINs that appear only in the VIN impression data (right side). Left outer join implies that you should output data for all VINs that occur in your session data (left side), but only those VINs from the VIN impression data (right side) that also occur in the session data (left side).

Hints for the run() method of your app:

* Define the input paths for your mappers using MultipleInputs.addInputPath(). The input format for the mapper that processes sessions will be AvroKeyValueInputFormat, and the input format for the mapper that process VIN impression counts will be TextInputFormat.
* You will also need to add these calls in order to read the AVRO container file: AvroJob.setInputKeySchema(), and AvroJob.setInputValueSchema().
* Use TextOutputFormat for reducer output (we want this to be human readable).

**Inputs**

* AVRO container file with your user sessions.
* VIN impression counts:
  + dataSet6VinCounts1.csv
  + dataSet6VinCounts2.csv

**Required elements**

* Create an AVRO container file for input by modifying your assignment 5 session generator app. (Don't include this file of generated sessions in your assignment submission.)
* Define two mappers.
* Use MultipleInputs to associate each mapper class with its input file. Note: Since there are two files with VIN impression count data, you'll need a call to addInputFile() for each input file.
* Implement left outer join in your reducer, where the data from your user sessions is considered the "left" input.

**Artifacts to submit**

* Assignment6Build.zip or tar - all files (Java, avsc, pom.xml) in the directory structure required by maven and buildable with your pom.xml file.
* Assignment6Code.zip - all files (Java, avsc) in a flat directory for easy inspection for grading
* Assignment6Output.txt - output file containing the text representation of VinImpressionCounts objects

**Extra Credit (2 points)**

Implement and use a combiner that shares the reducer code for combining VinImpressionCounts instances.

Assignment 7 – Filtering and Multiple Outputs

For assignment 7, you'll read user sessions (from Avro container file), apply a filter, and output these user sessions into different categories based on the type of each user session. The output to different categories will be accomplished using multiple outputs from each mapper, so this will be a map-only job.

User session categories are defined as follows:

* SUBMITTER - session has any of these events: CHANGE, EDIT, or SUBMIT CONTACT\_FORM
* CLICKER - Not a Submitter session, has a CLICK  event
* SHOWER - Not a Clicker session, has a SHOW or DISPLAY event
* VISITOR - Not a Shower session, has a VISIT event
* OTHER - None of the above.

**Required elements:**

* Input the user sessions you created for Assignment 6 (AVRO container file).
* Filter out all sessions with more than 100 events.
* There are an uneven number of sessions in the categories, so in order to make the number of sessions output in each category roughly equivalent you will implement filters that randomly sample sessions as follows:
  + CLICKER sessions - random sample 1 in 10 (or 10%)
  + SHOWER sessions - random sample 1 in 50 (2%)
  + Keep all sessions of type SUBMITTER, VISITOR, and OTHER
* Add counters that record the number of sessions that were filtered out. You'll need three such counters, one for large sessions (> 100 events), one for CLICKER sessions, and one for SHOWER sessions.
* Use the AvroMultipleOutputs class to output sessions to different files based on category.
  + You will need to determine the category for each session using the definition of session category given above.
  + Use the getText() method of enum SessionType (provided on Canvas) to the get the name for each "named output" used with the AvroMultipleOutputs class.
* Map-only job, so set the number of reducers to zero.
* Use AvroKeyValueOutputFormat for your output format.
* Use the setCountersEnabled() method of AvroMultipleOutputs to create counters to count the number of each session type that you write out. These counts will be reported in the syslog file, which you will turn in for this assignment.

**Important hint:**

In addition to defining the output format class (AvroKeyValueOutputFormat.class) and the schemata for the output key and output value in the AvroMultipleOutputs.addNamedOutput(), you'll also need to specify the output format and the key and value schemata using:

* AvroJob.setOutputKeySchema()
* AvroJob.setOutputValueSchema()
* job.setOutputFormatClass()

**Artifacts to submit**

* Assignment7Build.zip or tar - all files (Java, avsc, pom.xml) in the directory structure required by maven and buildable with your pom.xml file.
* Assignment7Code.zip - all files (Java, avsc) in a flat directory for easy inspection for grading
* Assignment7Output.txt - syslog file

Assignment 8 – Job Chaining

For assignment 8, you'll implement a map-reduce program that launches multiple jobs, some executing serially and some executing concurrently.

Start with assignment 7 code: output sessions into bins based on the session type, but only filter out large sessions. When this job has completed, add the creation, launch, monitoring, and completion of four map-reduce jobs that determine event subtype statistics in submitter, clicker, shower, and visitor sessions. You'll create and configure these jobs in the run() method, launching each job so that they run concurrently. Your code should then wait for all four of these jobs to complete.  When these four jobs have completed, create, configure, and launch a fifth job that aggregates the statistics across all four session types by reading the output files generated by the four previous jobs. You will use the Job class methods we discussed in class (submit(), isComplete()). All this code will be in the run() method, or called from the run() method of your (single) map-reduce program.

**Event SubType Statistics**

For event subtype statistics, collect data just as we did for the WordStatistics assignment, replacing "words" with the session type and event subtype event. One aspect will be different than what we did in WordStatistics: in WordStatistics we computed the mean and variance for the word count only in those documents in which the word appeared. For event subtypes, we want to calculate the mean and variance across all sessions, whether the session had that event subtype or not.  Use the following AVRO schema for the key and value of the event subtype statistics (available on Canvas: Files / Assignment 8):

* eventSubTypeStatsData.avsc
* eventSubTypeStatsKey.avsc

Output of the four jobs collecting stats for specific session types should be written using AvroKeyValueOutputFormat. Output of the last job that aggregates the statistics should be written using TextOutputFormat.

**Some hints on what you'll need to do:**

- For the jobs that collect event subtype statistics, you'll need to explicitly define the mapper output schema (key and value), even though it is the same as the job output schema.  I had to do this to get these jobs to work.

- Feel free to use the session.avsc file and the AVRO container file I generated with this schema for Assignment 6 and 7. Remember that you need to use both files. You can also use your own session generation code and schema if you prefer.

session.avsc is provided on Canvas (Files / Solutions / Assignment 6), as is the AVRO container file.

**Required elements**

* Do the binning as we did in Assignment 7, but only filter out large sessions.
* The four jobs that compute statistics for submitter, clicker, shower, and visitor sessions should run in parallel.
* Final output should have statistics for each event subtype for each session type (submitter, clicker, shower, and visitor), as well as these statistics for these four session types combined.

**Artifacts to submit**

* Assignment8Build.zip or tar - all files (Java, avsc, pom.xml) in the directory structure required by maven and buildable with your pom.xml file.
* Assignment8Code.zip - all files used for this assignment (Java, avsc) in a flat directory for easy inspection for grading
* Assignment8Output.txt - output file containing the event subtype statistics.

**Extra Credit (3 points)**:

In addition to calculating the statistics for each session type, and for all session types, also calculate the statistics for any event subType for each session type, and for all session types together.

Assignment 9 – Spark Setup

For this assignment your task is to download and setup Spark, and run the Java class that implements WordCount for Spark.

Download version 2.2.0 of Spark, that can be found here:

[https://spark.apache.org/downloads.htmlLinks to an external site.](https://spark.apache.org/downloads.html)

Select these options:

* Version 2.2.0
* Prebuilt for Hadoop version 2.7 and later

untar the downloaded file, and you should be ready to go.  Look over the documentation provided here:

[https://spark.apache.org/docs/2.2.0/quick-start.htmlLinks to an external site.](https://spark.apache.org/docs/latest/quick-start.html)

for details on running the Spark shell (using Scala), and self-contained (compiled) applications, like Java.

For this assignment, once you have the Spark install setup, you'll do the following:

* Grab the pom.xml file and WordCount.java files on Canvas (Files / Assignment 9)
* Create a new project structure using the above pom.xml file and Java file.  You can use your existing project structure if you want (I find creating a new project structure easier, your mileage may vary). If you do use your existing project structure, update the pom.xml file to the versions in the new pom.xml file I've provided, add version and dependency info for Spark, and add the configuration filters section to the shade plugin description.
* Build the uber JAR in the same way you did for your Hadoop map-reduce job(s).
* Run the Spark WordCount app
  + In the directory where Spark was installed (in my environment, that is a path ending with:  **spark-2.2.0-bin-hadoop2.7**), run the command:
  + **./bin/spark-submit**
  + Arguments:
    - **--class "com.refactorlabs.cs378.assign9.WordCount"**
    - **--master local[1]**
    - *location of your uber JAR*
    - input file:**README.md**
    - *output directory*

For the output directory, I created a directory named "output" in the Spark installation directory, and then specified a subdirectory for the assignment: output/assign9. Just like Hadoop map-reduce, Spark expects to create the directory, and therefore you should delete that directory (output/assign9 in my example) before running your Spark job. The input file is the readme file that comes with the Spark install.

**Artifacts to submit**

* Assignment9Output.txt - the output file generated by the WordCount app

**Useful links:**

[https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html (Links to an external site.)Links to an external site.](https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html)

[https://spark.apache.org/docs/2.1.0/api/python/pyspark.html (Links to an external site.)](https://spark.apache.org/docs/2.1.0/api/python/pyspark.html)

Assignment 10 – Inverted Index in Spark

For assignment 10, implement an inverted index as you did for Assignment 3, this time using:

* Different input data (described below)
* Spark instead of Hadoop map-reduce.

The input file is a text file or verses, each line start with a verse ID followed by the verse text. For this assignment you will need to extract the verse ID at the beginning of each line. Format of the ID is:

book:chapter:verse

Where book is a string, and chapter and verse are integers. The content of the verse is the remainder of the input line (separated from the ID by whitespace). There will be blank lines in the input, so be prepared to handle that in your mapper.

Your output should be in the following format (each line):

* the word
* verse references (book:chapter:verse), separated by commas

The output file should be ordered by the words being indexed (lexicographical order), and the verse references should be ordered by book, chapter, and verse number. Remove punctuation and make all the words lower case.

The input files can be found on Canvas: Files / Assignment 10

* dataSet10Small.txt - for development
* dataSet10.txt - for assignment submission

Artifacts to submit for the assignment:

* Assignment10Build.tar - all files (Java or your source language, pom.xml) in the directory structure required by maven and buildable with your pom.xml file.
* Assignment10Code.zip - all files (Java or your source language (Python, Scala)) in a flat directory for easy inspection for grading
* Assignment10Output.txt - output generated by your program

Extra credit (3 pts.):

* In addition to the file output for the base assignment, output another file where the lines are ordered by the number of references in the index, and where this number is the same, order by the word being indexed.

Assignment 11 – User Sessions in Spark

For assignment 11 you will create user sessions in Spark, similar to what you did in assignment 5. In particular:

* Remove duplicate events
* When sorting by timestamp, use the event type as a secondary sort, event subtype as a tertiary sort

Instead of using AVRO to represent a session, you should represent a session in a JavaPairRDD where the key is the tuple <userID, city>, and the value is an array of events, where each event is an instance of the class provided here (create a similar structure if you are using Scala or Python):

private static class Event implements Serializable {   
String eventType;  
String eventSubType;  
String eventTimestamp;  
String vin;   
public String toString() { return "<" + eventType + ":" + eventSubType + "," + eventTimestamp + ">";}   
}

The eventType and eventSubType should be derived from the event\_type field from the input, the eventTimestamp should be the event\_timestamp field from the input, and the vin should be the vin from the input. The city that you add to the key (along with the userID) should be the city field from the input file.

**Data Set**

Use dataSet5a.tsv and dataSet5b.tsv (available on Canvas).

**Required elements**

Once events have been organized into sessions, do the following

* For sessions of type "SHOWER", sample these sessions at a rate of 1 in 10.
  + A "SHOWER" session is defined as:
    - The session contains no event whose subtype is Contact Form (not a SUBMITTER session)
    - The session contains no Click event (not a CLICKER session)
    - The session contains at least one event of type Show or Display
* Order the events in each session by timestamp
* Order sessions by userID, then by city.
* Partition user sessions by city using a custom partitioner and apply this partitioning at some point, so that the final outputs are partitioned in this way. Your custom partitioner should distribute the sessions by hashing the city, and should produce 6 partitions.
* Add accumulators to count:
  + Total number of events (2 different counts):
    - After duplicates removed, but before session filtering
    - After session filtering
  + Total number of sessions
  + Total number of sessions of type SHOWER
  + Total number of sessions of type SHOWER that were filtered out
  + Output these counts to System.out (with a descriptive string for each, identifying the particular count) and include this output in the artifacts you submit.

**Artifacts to submit**

* Assignment11Build.zip or tar - all files (Java or your language, pom.xml) in the directory structure required by maven and buildable with your pom.xml file.
* Assignment11Code.zip - all files (Java or your language) in a flat directory for easy inspection for grading
* Assignment11Output.tar - tar or zip the output directory, containing all 6 output files and a file with the accumulator count outputs.

Assignment 12 – Spark DataSets

For this assignment, input the file dataSet12.csv into a DataSet and compute the following using SQL:

* By make/model, the min, max, and average price (excluding VINs with price = 0)
  + Order the output CSV by make, then model
* By year, the min, max, and average mileage (excluding VINs with mileage - 0)
  + Order the output CSV by year
* By VIN, the total for each event type/action (you'll need to split out the event type/action from the event)
  + Order the output CSV by VIN, then event type/action

Note that the same VIN (and associated make, model, price, and mileage) appears multiple times in the events, so only consider each VIN once in the price and mileage statistics

Artifacts to submit for the assignment:

* Assignment12Build.zip or tar - all files (Java or your language, pom.xml) in the directory structure required by maven and buildable with your pom.xml file.
* Assignment12Code.zip - all files (Java or your language) in a flat directory for easy inspection for grading
* Assignment12OutputPrice.txt - output for price statistics
* Assignment12OutputMileage.txt - output for mileage statistics
* Assignment12OutputEvent.txt - output for event statistics