BIG DATA ANALYSIS & ANALYTICS WITH MATLAB®

D. Willingham#, MathWorks, Sydney, Australia

Abstract

In today's world, there is an abundance of data being generated from many different sources in various industries across engineering, science & business. For example, DESY's 1.7 mile-long PETRA III accelerator is capable of generating 20 gigabyte's of data per second [1]. Using Data Analytics to turn large volumes of complex data into actionable information can help improve design and decision making processes. Big data sets may often be too large for the available memory, take too long to import and process, or just stream too quickly to store. Standard algorithms are usually not designed to process big data sets in reasonable amounts of time or memory usage. MATLAB® is the high-level language and interactive environment used by millions of engineers and scientists worldwide. It lets you explore and visualize ideas and collaborate across disciplines including signal and image processing, communications, control systems, and computational finance. Using a generic case study we demonstrate efficient ways to manipulate, compute and visualise on large, multidimensional data sets on light weight machines. The techniques presented here can also be used to develop predictive models, scale up for high performance computing on clusters, or in the cloud and deployable within databases, Hadoop and other Big Data environments.

INTRODUCTION

Big data describes a massive volume of data that is so large or complex that it's difficult to manage and process using traditional database and software techniques [2]. Big Data is commonly described by the 3 V's

- Volume – which references the amount of data.
- Velocity – is the speed at which data is generated or needs to be analyzed.
- Variety – which describes the range of data types and sources from which data is obtained.

There are a number of challenges that arise when analyzing big data sets, especially for domain experts who are not necessarily experienced software programmers. These include:

- How to get started and gain insight into the basic structure and format of the data especially when the volume exceeds memory limits?
- Rapid exploration of the data, and development of algorithms which can easily be scaled for use with big data
- Using big data algorithms within business systems

As an interesting example of the volume and velocity of data that is being generated in the world. In 2000, the world generated 2.5 Exabyte's of data a day [3]

In this paper, we present a generic case study identifying and predicting patterns of the domestic flights in the US between 1987 and 2008, which is 12gB in size [4]. We address the challenges mentioned previously by demonstrating two capabilities of MATLAB®, namely, datastore and mapreduce and the links with Big Data storage platforms such as Hadoop.

This paper is organised as follows. We highlight 3 techniques that can be used to handle importing and processing big data;

1. Scale the problem down
2. Parallelize the Problem
3. Scale up if needed along with the MATLAB functions that support these.

TECHNIQUES FOR HANDLING BIG DATA

Scale the Problem Down

For simpler problems where data can be too big to be analyzed at once, but where distinct sub-problems can be calculated and the results easily aggregated together, we can simply loop through the various files or records, load the subset of data we're interested in, analyze it, store the result and repeat. MATLAB's `datastore` enables this to be achieved in the easy efficient way.

What is *datastore*?

A *datastore*[4] is an object for reading a single file or a collection of files or data. The *datastore* acts as a repository for data that has the same structure and formatting. For example, each file in a *datastore* must contain data of the same type (such as numeric or text) appearing in the same order, and separated by the same delimiter. A *datastore* is useful when:

- Each file in the collection might be too large to fit in memory. A *datastore* allows the ability to read and analyze data from each file in smaller portions that do fit in memory
- Files in the collection have arbitrary names. A *datastore* acts as a repository for files in one or more folders. The files are not required to have sequential names.
- Not all the data needs to be loaded into memory. For example, you may only wish to analyse 10 columns out of 50 from a data file, so only import the 10 that are needed.

david.willingham@mathworks.com.au

ISBN 978-3-95450-148-9

Create and Read from a Datastore

Use the *datastore* function to create a *datastore*. For example, create a *datastore* from the sample file, airline.csv [5]. This file includes departure and arrival information about individual airline flights.

```
ds = datastore('airline.csv')
```

After creating the *datastore*, the data can be previewed without having to load it all into memory. Variables can be specified as columns of interest using the SelectedVariableNames property to preview or read only those variables.

```
ds.SelectedVariableNames=
('DepTime','DepDelay'};
preview(ds)


ans =
    DepTime    DepDelay
    _______    ________
     642        12
    1021         1
    2055        20
    1332        12
     629        -1
    1446        63
     928        -2
     859        -1
```

Values can be specified in the data which represent missing values. In airline.csv, missing values are represented by NA.

```
ds.TreatAsMissing = 'NA';
```

If all of the data in the *datastore* for the variables of interest fit in memory, it can be read using the readall function.

```
T = readall(ds);
```

Otherwise, read the data in smaller subsets that do fit in memory, using the read function. By default, the read function reads 20000 rows at a time. However, this value can be changed by assigning a new value to the ReadSize property.

```
ds.ReadSize = 15000;
```

Reset the *datastore* to the initial state before re-reading, using the reset function. By calling the read function within a while loop, we can perform intermediate calculations on each subset of data, and then aggregate the intermediate results at the end. This code calculates the maximum value of the DepDelay variable.

```
reset(ds)
X = [];
while hasdata(ds)
    T = read(ds);
    X(end+1) = max(T.DepDelay);
end
maxDelay = max(X)
maxDelay =
        1438
```

If the data in each individual file fits in memory, you can specify that each call to read should read one complete file rather than a specific number of rows.

```
reset(ds)
ds.ReadSize = 'file';
X = [];
while hasdata(ds)
    T = read(ds);
    X(end+1) = max(T.DepDelay);
end
maxDelay = max(X);
```

Parallelize the Problem

As the number and type of data acquisition devices grows annually, the sheer size and rate of data being collected is rapidly expanding. These big data sets can contain gigabytes or terabytes of data, and can grow on the order of megabytes or gigabytes per day. Most algorithms are not designed to process big data sets in a reasonable amount of time or with a reasonable amount of memory. *MapReduce* allows us to meet many of these challenges to gain important insights from large data sets, and importantly can be run using parallel processing on multiple cores, processors or clusters which can reduce computational time.

What is *MapReduce*?

MapReduce[6] is a programming technique for analyzing data sets that do not fit in memory.

mapreduce uses a *datastor*e to process data in small chunks that individually fit into memory. Each chunk goes through a Map phase, which formats the data to be processed. Then the intermediate data chunks go through a Reduce phase, which aggregates the intermediate results to produce a final result. The Map and Reduce phases are encoded by map and reduce functions, which are primary inputs to *mapreduce*. There are endless combinations of map and reduce functions to process data, so this technique is both flexible and extremely powerful for tackling large data processing tasks.

mapreduce lends itself to being extended to run in several environments, on a single PC, on a cluster and or integrated with Hadoop®.

Prepare Data

The first step to using *mapreduce* is to construct a *datastore* for the data set. Along with the map and reduce

ISBN 978-3-95450-148-9

functions, the *datastore* for a data set is a required input to *mapreduce*, since it allows *mapreduce* to process the data in chunks.

```
ds = datastore('airline.csv');
```

Write Map and Reduce Functions

The *mapreduce* function automatically calls the map and reduce functions during execution, so these functions must meet certain requirements to run properly.

1. The inputs to the map function are data, info, and intermKVStore:

- data and info are the result of a call to the read function on the input *datastore*, which *mapreduce* executes automatically before each call to the map function .
- intermKVStore is the name of the intermediate KeyValueStore object to which the map function needs to add key-value pairs. The add and addmulti functions use this object name to add key-value pairs. If none of the calls to the map function add key-value pairs to intermKVStore, then *mapreduce* does not call the reduce function and the resulting *datastore* is empty.

A simple example of a map function is:

```
function MeanDistMapFun(data, info,
intermKVStore)

    distances =
data.Distance(~isnan(data.Distance));

    sumLenValue = [sum(distances)
length(distances)];

    add(intermKVStore, 'sumAndLength',
sumLenValue);

end
```

2. The inputs to the reduce function are intermKey, intermValIter, and outKVStore:

- intermKey is for the active key added by the map function. Each call to the reduce function by mapreduce specifies a new unique key from the keys in the intermediate KeyValueStore object.
- outKVStore is the name for the final KeyValueStore object to which the reduce function needs to add key-value pairs. *mapreduce* takes the output key-value pairs from
- outKVStore and returns them in the output *datastore*, which is a KeyValueDatastore object by default. If none of the calls to the reduce function add key-value pairs to outKVStore, then *mapreduce* returns an empty *datastore*.

A simple example of a reduce function is:

```
function MeanDistReduceFun(intermKey,
intermValIter, outKVStore)

    sumLen = [0 0];

    while hasnext(intermValIter)

        sumLen = sumLen +
getnext(intermValIter);

    end

    add(outKVStore, 'Mean',
sumLen(1)/sumLen(2));

end
```

This reduce function loops through each of the distance and count values in intermValIter, keeping a running total of the distance and count after each pass. After this loop, the reduce function calculates the overall mean flight distance with a simple division, and then adds a single key to outKVStore.

Run MapReduce

After you have a *datastore*, a map function, and a reduce function, you can call *mapreduce* to perform the calculation. To calculate the average flight distance in the data set, call *mapreduce* using ds, MeanDistMapFun.m, and MeanDistReduceFun.m.

```
outds = mapreduce(ds, @MeanDistMapFun,
@MeanDistReduceFun);
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce 100%
```


ISBN 978-3-95450-148-9

View Results

Use the readall function to read the key-value pairs from the output *datastore*.

```
readall(outds)

ans =

     Key          Value

    _____      __________

    'Mean'     [702.1630]
```

SCALE UP IF NEEDED

Scale Up if Needed

By default, if the Parallel Computing Toolbox is installed with MATLAB, the *MapReduce* will run in parallel on multiple cores. However in big data problems it's common for the scale of the problem to go beyond what a single machine can handle. In such situations, users should look to process the problem on a cluster (or cloud) that link up to big data storage framework, e.g. Hadoop. Note that this method does require a cluster, and works best if the analytics are deployed on the same cluster as where the data is stored.

Using the *MapReduce* and *Datastore* functionality built into MATLAB, we can develop algorithms on our desktop and directly execute them on Hadoop. To get started, access a portion of the big data stored in HDFS with the MATLAB *datastore* function, and use this data to develop *MapReduce* based algorithms in MATLAB on our desktop. We then use MATLAB Distributed Computing Server to execute the algorithms on a cluster within the Hadoop *MapReduce* framework against the full data set stored in HDFS. Additionally we could integrate MATLAB analytics with production Hadoop systems by using the–MATLAB Compiler to create applications or libraries from MATLAB MapReduce based algorithms.

CONCLUSION

While Big data represents an opportunity to gain greater insight and make more informed decisions, but it also presents introduces a number of challenges with no one size fits all solution. In this paper we presented three capabilities of MATLAB to help address these. Using *datastore* to import big data sets efficiently so that they can fit into available memory, Using m*apreduce,*to reduce the computational time to process analytics on the data through parallelization. And finally how to combine the first 2 approaches and scaling up and integrate the analytics with computational clusters and Hadoop.

Looking to the future, Big Data computing is a field that is constantly evolving due to the ever increasing amounts of data being generated each day. MATLAB's *datastore*, *mapreduce* and deployment capabilities will enable users to evolve their analytics to meet this ongoing challenge.

REFERENCES

[1] "DESY and IBM Develop Big Data Architecture for Science" 21 Aug, 2014. https://www-03.ibm.com/press/us/en/pressrelease/44587.wss

[2] Beyer, Mark. "Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data". Gartner. Archived from the original on 10 July 2011.

[3] IBM press release, "IBM Expands PureSystems Family to Help Clients Tame Big Data" https://www-03.ibm.com/press/us/en/pressrelease/39039.wss, Oct 9, 2012.

[4] MathWorks Website "Getting Started with Datasore": www.mathworks.com/help/matlab/import_export/what-is-a-datastore.html

[5] United States Department of Transportation website: http://www.transtats.bts.gov/OT_Delay/OT_DelayCause1.asp

[6] MathWorks Website "Getting Started with MapReduce": www.mathworks.com/help/matlab/import_export/getting-started-with-mapreduce.html

ISBN 978-3-95450-148-9