Parallel Data with MapReduce and Spark (Part 1, Introduction)

Daniel Lawson — University of Bristol

Lecture 12.1 (v2.0.0)

Summary

- In this lecture we cover:
  - **Big Data**
  - Streaming
  - Hadoop Distributed file system (HDFS)
  - Hadoop MapReduce

Big Data

- Some key concepts for understanding big data are:
  - **Volume**: The defining feature of big data is that there is lots of it!
  - **Velocity**: To create lots of data, the data arrive with high frequency. It must be appropriately dealt with immediately.
  - **Variety**: With unfiltered volume, anything that can be seen, will be seen. It may also change over time. Systems must be able to cope with this.
  - **Veracity**: This is the trustworthiness of the data, e.g. does the data represent the truth?

Frameworks

- There are different **computational frameworks** for handling data with each type.
  - High **Velocity** data is dealt with using **streaming** computation.
  - High **Volume** data is stored using a **distributed file system**.
- The underlying engineering systems typically interact:
  - A **streaming decision** is made about whether to store the data, and any immediate decisions;
  - The **distributed file system** allows retrospective, costly analysis or machine learning.
- Variety and Veracity are dealt with at the choice-of-algorithm stage, and do not impose engineering constraints.

Streaming Context

- **Streaming** is about working with data as it arrives, in real time.
  - High **Velocity** means that data can't always be stored.
  - Immediate decisions must be made in $O(1)$ time.
  - Special algorithms called **streaming algorithms** are used to handle this.
- Streaming is jointly an engineering and an algorithms problem.
  - There are an entire class of algorithms for working in the streaming context.
  - There are a class of statistical quantities that can be calculated or approximated in the stream.
  - We focus on the data exploration role of big data.

Some streaming algorithms

- High level perspective:
  - We want to design an estimator for some quantity of interest.
  - We want to have access to the estimator at any given time.
  - We require $O(1)$ effort for each incoming data point.
  - This requires storing our knowledge in an updatable manner.
- To compute the **streaming mean**:
  - We are used to calulating the mean $\overline{x}_n = \sum_{i=1}^{n} x_i/n$;
  - This can be re-written as a streaming algorithm as $\overline{x}_n = \overline{x}_{n-1} + (x_n - \overline{x}_{n-1})/n$.
- To compute the **streaming variance**:
  - We are used to calculating $\hat{s}_n^2 = \sum_{i=1}^{n}(x_i - \overline{x}_n)^2/(n-1)$;
  - This can be rewritten as $d_n^2 = d_{n-1}^2 + (x_n - \overline{x}_n)(x_n - \overline{x}_{n-1})$,
  - With $\hat{s}_n^2 = d_n^2/(n-1)$,
  - This is shown in the Worksheet.

Further streaming algorithms

- Additional algorithms include:
- The "**exponential moving average**": $\overline{x}_n = \frac{\sum_{i=1}^n w_i x_i}{\sum_{i=1}^n w_i}$, where $w_{i-1} = \alpha w_i$ for some $\alpha \in (0,1)$.
  - Streaming has the simple form:

$$\overline{x}_n = (1-\alpha)\overline{x}_{n-1} + \alpha(x_n - \overline{x}_{n-1}).$$

  - A wide class of exponential weighting schemes are possible.
  - Streaming clustering e.g. k-means can work in this way.
- **Sliding window** averages or other statistics.
- Sub-linear time algorithms, most notably Sketching (see Algorithms for Data Science).

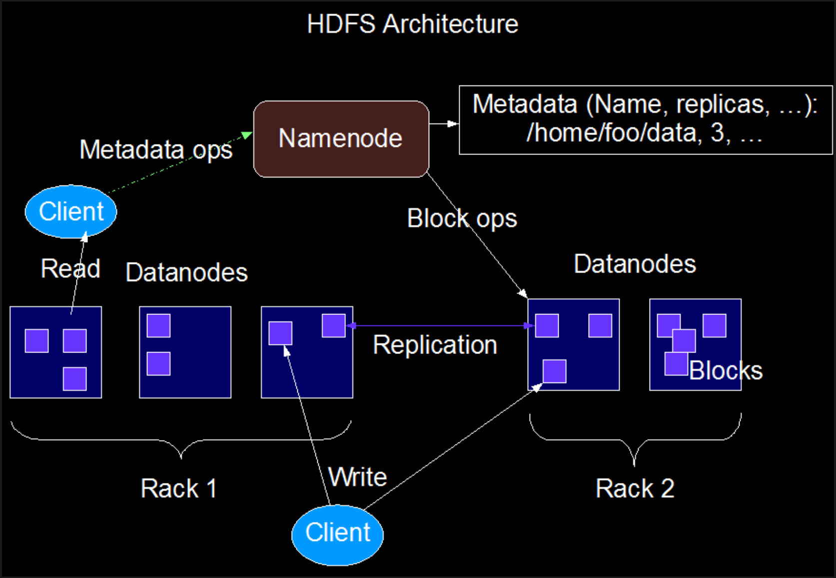
High volume data with HDFS

- When there is so much data that it can't all be stored in one location, a **distributed file storage** is required to quickly query it.
  - By placing compute with storage, filtering and other operations can be applied rapidly at scale.
- The **Hadoop Distributed File System** (HDFS) has become an industry standard because:
  - It is fault tolerant,
  - It was the first industry standard to be open-sourced,
  - It has remained supported and developed,
  - It is integral to several key tools.

HDFS architecture

- HDFS uses:
  - A **Namenode**, which keeps track of where the data are. It is typically run on a dedicated node.
  - Many **Datanodes**, which each keep track of numerous data blocks (typically millions). Each datablock is tracked on several datanodes (typically 2-5).
- Why this complexity? Because at scale, **devices fail** all the time.
  - The data are **duplicated** so that the probability of all duplicates of data becoming corrupted is low.
  - Each copy is also processed, meaning that **compute failures** are also tolerated.

HDFS architecture



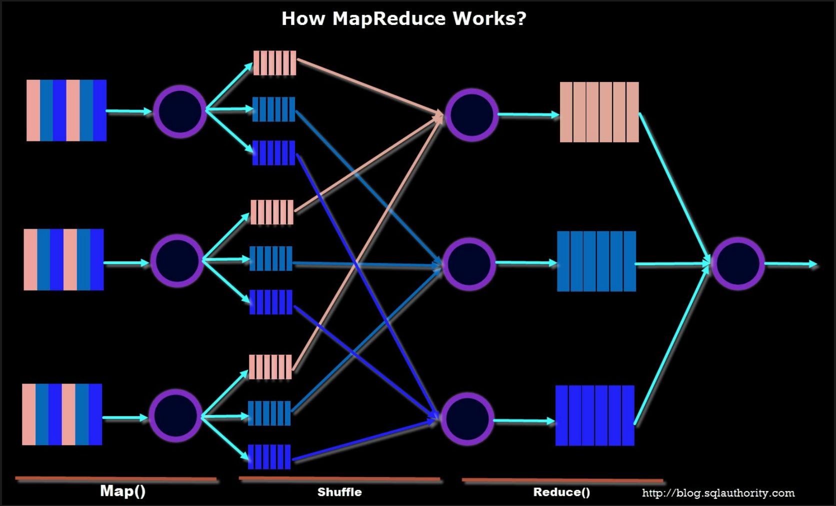
HDFS implementation

- HDFS is implemented in Java.
- It provides a **virtual filesystem** interface that treats the entire set of data blocks as if they were on a regular filesystem.
- **Data blocks** are simply regular files and can be read in the regular way.
- Input and output are by default structured around blocks (as medium sized files, several Mb) in a single virtual directory.

Hadoop

- Apache Hadoop is an open-source implementation of Map/Reduce optimised for distributed data.
- Using N_m input data blocks and N_r reducers, Hadoop performs for following stages of computation:
  1. A **Map** which produces exactly one file M_{out} for each input file M_{in} (N_m in total). This is run in parallel by the host of the data block.
  2. A **Sort** which ensures that each key appears in exactly one file R_{in} (N_r in total). This is a distributed sort operation, which places the output into the pre-allocated memory of the hosts of the reduce block.
  3. A **Reduce** which produces exactly one output file R_{out} per input R_{in} (N_r in total). This is again performed in parallel.
- Because data are distributed, it may be that some hosts are busier than others. Each stage can be completed when just one copy of the processing has completed.

Hadoop


How MapReduce Works?
Map()
Shuffle
Reduce()
http://blog.sqlauthority.com

Practical concerns

- Unless you want to code in Java, you want to use **Hadoop Streaming**.
- This is **not** streaming as discussed above! It is instead:
  - An interface to allow **any** binary to be used as a mapper and reducer.
  - To do this, you need to work with **stdin** and **stdout**,
  - In which each line is **processed independently** as a record.
  - In bash this is handled by **pipes**, e.g.
  - `cat file.txt.gz | gunzip -c`: extracts a compressed file with a pipe, pass this to `gunzip` as a stream, print to stdout.

Python Streaming Mapper (1)

- Thanks to Edinburgh Hadoop Streaming…

```
#!/usr/bin/python2.7
# mapper.py
import sys

def map_function(title):
    # Split title to fields using the data delimeter
    fields = title.strip().split('\t')
    # Select the required field
    primaryTitle = fields[2]
    # Split primary title by words
    for word in primaryTitle.strip().split():
        # Use a word as a key
        yield word, 1

<...>
```

Python Streaming Mapper (2)

```
<...>

for line in sys.stdin:
    # Call map_function for each line in the input
    for key, value in map_function(line):
        # Emit key-value pairs using '|' as a delimeter
        print(key + "|" + str(value))
```

Python Streaming Reducer (1)

```python
#!/usr/bin/python2.7
# reducer.py
import sys

def reduce_function(word, values):
    # Calculate how many times each word was encountered
    return word, sum(values)

prev_key = None
values = []
```

Python Streaming Reducer (2)

```
 for line in sys.stdin:
    key, value = line.strip().split('|')

    # If key has changed then
    # finish processing the previous key
    if key != prev_key and prev_key is not None:
        result_key, result_value = \
          reduce_function(prev_key, values)
        print(result_key + "|" + str(result_value))
        values = []

    prev_key = key
    values.append(int(value))

    # Don't forget about the last value!
if prev_key is not None:
    result_key, result_value = \
      reduce_function(prev_key, values)
    print(result_key + "|" + str(result_value))
```

Resource management

- **Resource management** is handled by "YARN" (Yet Another Resource Negotiator) which provides:
  - Management of **data storage**, including data re-duplication,
  - Management of **CPU access**, i.e. job queue,
  - "**Application management**", i.e. load balancing, monitoring of the system, automatic rerunning of failed jobs, etc.
- YARN requires separate installation and is typically handled by a sysadmin.
  - We therefore will not be using it. Just know it exists.

Mapping

- Because the data are distributed, mapping requires all **datanodes containing the data** to run.
- This can lead to **congestion** if (required) data are not balanced.
- There is (a small amount of) **flexibility** if duplicates are ignored, and only one instance of each block is initially analysed.

Sorting

- Sorting is not a trivial thing:
  - sorting on the **correct key** is an integral part of the algorithm.
  - e.g. counting the key “IP” is different to counting the key “IP-PORT”.
- However, you rarely need to worry about the **algorithm used** to perform the sorting.
  - Multiple algorithms are provided.
  - Sorting is typically done via hashing into reducer input blocks.
  - It can be costly (in terms of network bandwidth) if a lot of data made it through to the reduce.
- All nodes involved in Mapping and Reducing are required for sorting.
- Whilst it is straightforward to ensure that **keys** are evenly balanced across nodes, the number of **values** may not be.

Reducing

- The parallelism of reducing is **chosen by the user**.
- By design, Reducers cannot share information across keys. So they should be linear in the amount of data if:
  - the amount of data per key is $O(1)$, or
  - the reducers are $O(1)$ within a key.
- **Good Map/Reduce design** is needed to ensure that one of these conditions hold!

Pseudo-code

- You need to specify the **map** function, the **reduce** function, and the associated **keys**.
  - **Sorting is assumed** and is done on the reducer key.
  - Therefore there is no difference with regular Map/Reduce algorithms.
- An alternative way to describe the Map-Reduce mean algorithm from Block 10 is:
- **class Mapper** maps $(key = k_0, v)$ to $(key = k, (count = 1, value = v))$
  - where $k = k_0$
- **class Reducer** reduces $(k, (count, value))$ to $(k, (count = c, value = v))$
  - where $c = \sum_{i:key=k} c_i$ and $v = \sum_{i:key=k} v_i$
- **Postprocess**: Return $mean = \frac{\sum_{k=1}^{K} v_k}{\sum_{k=1}^{K} c_k}$

Hadoop limitations

- Hadoop is efficient for Map + Sort + Reduce.
- It supports complex manipulations in Java, or arbitrary languages through "**Hadoop Streaming**" (which we will use).
- There are ways to be more efficient, e.g. reduce each map output before sort.
- It is slow for iterative calculations because all content is written to/from disk.
- It is also **stateless** between iterations (though additional files can be provided as reference data).
- It is probably **not** going to be what you use, unless you have a legacy application.

References

- General parallel algorithms:
  - Streaming and Sketching
  - Parallel algorithms for dense matrix multiplication
- Map Reduce
  - Apache Hadoop
  - Gentle introduction to MapReduce
  - A Q&A
  - Lecture@poznan
  - Basic MapReduce Algorithms Design
  - Tutorialspoint Mapreduce
  - Hadoop for Streaming applications