Scalable Parallel Processing with MapReduce

- Understanding the challenges of scalable parallel processing
- Leveraging MapReduce for large scale parallel processing
- Exploring the concepts and nuances of the MapReduce computational model
- Getting hands-on MapReduce experience using
  - MongoDB

Manipulating Large Amounts of Data

- Manipulating large amounts of data requires tools and methods that can run operations in parallel with as few as possible points of intersection among them.

- Fewer points of intersection lead to fewer potential conflicts and less management.

- Such parallel processing tools also need to keep data transfer to a minimum.

- I/O and bandwidth may become bottlenecks preventing fast and efficient processing
- With large amounts of data the I/O bottlenecks can be amplified and can potentially slow down a system to a point where it becomes impractical to use it.
- Therefore, for large-scale computations, keeping data local to a computation is of enormous importance.

- Given these considerations, manipulating large data sets spread out across multiple machines is neither trivial nor easy.

Super Computers

- Over the years, many methods have been developed to compute large data sets.
- Initially, innovation was focused around building super computers.
- Super computers are meant to be super-powerful machines with greater-than-normal processing capabilities.
- These machines work well for specific and complicated algorithms that are compute intensive but are far from being good general-purpose solutions.
- They are expensive to build and maintain and out of reach for most organizations.

Grid Computing

- Grid computing emerged as a possible solution for a problem that super computers didn’t solve.
- The idea behind a computational grid is to distribute work among a set of nodes and thereby reduce the computational time that a single large machine takes to complete the same task.
- In grid computing, the focus is on compute-intensive tasks where data passing between nodes is managed using Message Passing Interface (MPI) or one of its variants.
- This topology works well where the extra CPU cycles get the job done faster.
- However, this same topology becomes inefficient if a large amount data needs to be passed among the nodes.
- Large data transfer among nodes faces I/O and bandwidth limitations and can often be bound by these restrictions.
- In addition, the burden of managing the data-sharing logic and recovery from failed states is completely on the developer.

SETI@Home and Folding@Home

- Public computing projects like
  - SETI@Home (http://setiathome.berkeley.edu/)
  - Folding@Home (http://folding.stanford.edu/)

  extend the idea of grid computing to individuals donating "spare" CPU cycles for compute-intensive tasks.

- These projects run on idle CPU time of hundreds of thousands, sometimes millions, of individual machines, donated by volunteers.

- These individual machines go on and off the Internet and provide a large compute cluster despite their individual unreliability.

- By combining idle CPUs, the overall infrastructure tends to work like, and often smarter than, a single super computer.

MapReduce

- Despite the availability of varied solutions for effective distributed computing, none listed so far keep data locally in a compute grid to minimize bandwidth blockages.
- Few follow a policy of sharing little or nothing among the participating nodes.
- Inspired by functional programming notions that adhere to ideas of little interdependence among parallel processes, or threads, and committed to keeping data and computation together, is MapReduce.
- Developed for distributed computing and patented by Google, MR has become one of the most popular ways of processing large volumes of data efficiently and reliably.
- MapReduce offers a simple and fault-tolerant model for effective computation on large data spread across a horizontal cluster of nodes.
- This lecture explains MapReduce and explores the many possible computations on big data using this programming model.

MapReduce

- MapReduce is explicitly stated as MapReduce, a camel-cased version used and popularized by Google.
- However, the coverage here is more generic and not restricted by Google’s definition.
- The idea of MapReduce is published in a research paper, which is accessible online at
  - http://labs.google.com/papers/mapreduce.html
  - Dean, Jeffrey & Ghemawat, Sanjay (2004)
  - “MapReduce: Simplified Data Processing on Large lusters”

Understanding MapReduce

- We already introduced MapReduce as a way to group data on MongoDB clusters. Therefore, MapReduce isn't a complete stranger to you.
- We start out by using MapReduce to run a few queries that involve aggregate functions like sum, maximum, minimum, and average.
- The publicly available NYSE daily market data for the period between 1970 and 2010 is used for the example.
- Because the data is aggregated on a daily basis, only one data point represents a single trading day for a stock.
- Therefore, the data set is not large.
- Certainly not large enough to be classified big data.

Example

- The example focuses on the essential mechanics of MapReduce so the size doesn’t really matter.
- We use a document database, MongoDB, in this example.
- The concept of MapReduce is not specific to these products and applies to a large variety of NoSQL products including sorted, ordered column-family stores, and distributed key/value maps.
- MapReduce with Hadoop and HBase is to be discussed later.

Downloading NYSE Market Data

- To get started, download the zipped archive files for the daily NYSE market data from 1970 to 2010 from
  - http://www.infochimps.com/datasets/nyse-daily-1970-2010-open-close-high-low-and-volume
  - Extract the zip file to a local folder.

- The unzipped NYSE data set contains a number of files.
- Among these are two types of files:
  - daily market data files
  - dividend data files
- For the sake of simplicity, we use only the daily market data files into DB collection
- This means the only files you need from the set are those whose names start with NYSE_daily_prices_ and include a number or a letter at the end.
- All such files that have a number appended to the end contain only header information and so can be skipped.

MongoDB

- The database and collection in MongoDB are named mydb and nyse, respectively.
- The data is available in comma-separated values (.csv) format, so
- I leverage the mongoimport utility to import this data set into a MongoDB collection.

MongoImport Utility

```
mongoimport --type csv --db mydb --collection nyse --headerline NYSE_daily_prices_A.csv

connected to: 127.0.0.1
4981480/40990992 12%
89700 29900/second
10357231/40990992 25%
185900 30983/second
15484231/40990992 37%
278000 30888/second
20647430/40990992 50%
370100 30841/second
25727124/40990992 62%
462300 30820/second
30439300/40990992 74%
546600 30366/second
35669019/40990992 87%
639600 30457/second
40652285/40990992 99%
729100 30379/second
imported 735027 objects
```

Other daily price data files are uploaded in a similar fashion. To avoid sequential and tedious upload of 36 different files you could consider automating the task using a shell script

infochimps_nyse_data_loader.sh

```
#!/bin/bash
FILES=./infochimps_dataset_4778_download_16677/NYSE/NYSE_daily_prices_*.csv
for f in $FILES
do
  echo "Processing $f file..."
  # set MONGODB_HOME environment variable to point to the MongoDB installation
folder.
  ls -l $f
  $MONGODB_HOME/bin/mongoimport --type csv --db mydb --collection nyse --
headerline $f
Done
```

Once the data is uploaded, you can verify the format by querying for a single document as follows:

```
> db.nyse.findOne();
{
        "_id" : ObjectId("4d519529e883c3755b5f7760"),
        "exchange" : "NYSE",
        "stock_symbol" : "FDI",
        "date" : "1997-02-28",
        "stock_price_open" : 11.11,
        "stock_price_high" : 11.11,
        "stock_price_low" : 11.01,
        "stock_price_close" : 11.01,
        "stock_volume" : 4200,
        "stock_price_adj_close" : 4.54
}
```

Next, MapReduce can be used to manipulate the collection. Let the first of the tasks be to find the highest stock price for each stock over the entire data that spans the period between 1970 and 2010.

Map Function and Reduce Function

- MapReduce has two parts:
  - a map function
  - a reduce function
- The two functions are applied to data sequentially, though the underlying system frequently runs computations in parallel.
- Map takes in a key/value pair and emits another key/value pair.
- Reduce takes the output of the map phase and manipulates the key/value pairs to derive the final result.

Map Function

- A map function is applied on each item in a collection.
- Collections can be large and distributed across multiple physical machines.
- A map function runs on each subset of a collection local to a distributed node.
- The map operation on one node is completely independent of a similar operation on another node.
- This clear isolation provides effective parallel processing and allows you to rerun a map function on a subset in cases of failure.
- After a map function has run on the entire collection, values are emitted and provided as input to the reduce phase.
- The MapReduce framework takes care of collecting and sorting the output from the multiple nodes and making it available from one phase to the other.

Reduce Function

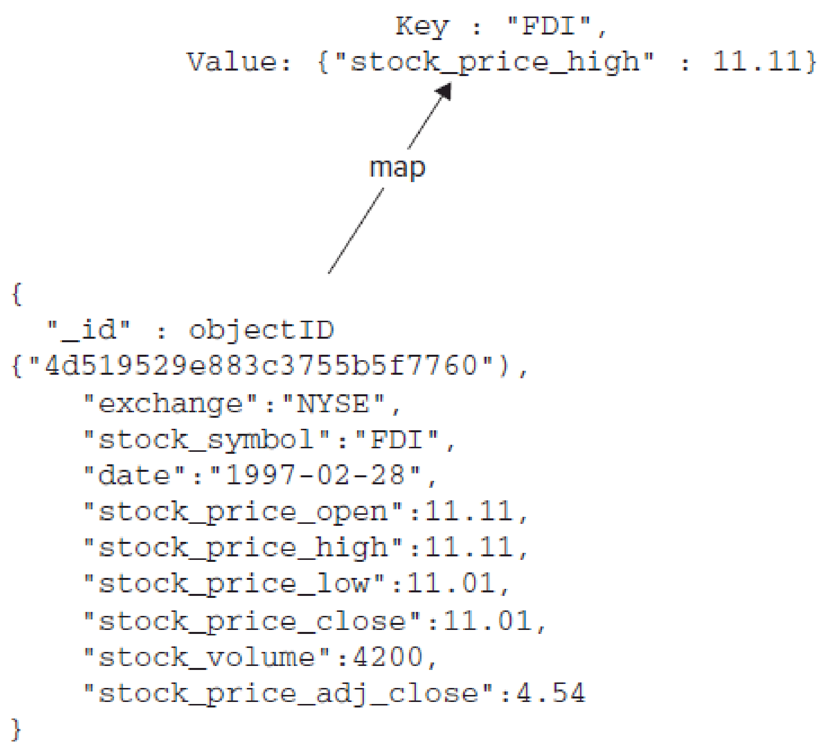
- The reduce function takes in the key/value pairs that come out of a map phase and manipulate it further to come to the final result.
- The reduce phase could involve aggregating values on the basis of a common key.
- Reduce, like map, runs on each node of a distributed large cluster.
- Values from reduce operations on different nodes are combined to get the final result.
- Reduce operations on individual nodes run independent of other nodes, except of course the values could be finally combined.
- Key/value pairs could pass multiple times through the map and reduce phases.
- This allows for aggregating and manipulating data that has already been grouped and aggregated before.
- This is frequently done when it may be desirable to have several different sets of summary data for a given data set.

Finding the Highest Stock Price for Each Stock

Getting back to the first task of finding the highest price for each stock in the period between 1970 and 2010, an appropriate map function could be as follows:

```
var map = function() {
  emit(this.stock_symbol, { stock_price_high: this.stock_price_high });
};
```

This function will be applied on every document in the collection. For each document, it picks up the stock_symbol as the key and emits the stock_symbol with the stock_price_high for that document as the key/value pair.



```
             Key : "FDI",
Value: {"stock_price_high" : 11.11}
```

```
{
  "_id" : objectID
{"4d519529e883c3755b5f7760"),
    "exchange":"NYSE",
    "stock_symbol":"FDI",
    "date":"1997-02-28",
    "stock_price_open":11.11,
    "stock_price_high":11.11,
    "stock_price_low":11.01,
    "stock_price_close":11.01,
    "stock_volume":4200,
    "stock_price_adj_close":4.54
}
```

The key/value pair extracted in the map phase is the input for the reduce phase. In MongoDB, a reduce function is defined as a JavaScript function as follows:

MongoDB supports only JavaScript as the language to define map and reduce functions.

```
var reduce = function(key, values) {
  var highest_price = 0.0;
  values.forEach(function(doc) {
    if( typeof doc.stock_price_high != "undefined") {
      print("doc.stock_price_high" + doc.stock_price_high);
      if (parseFloat(doc.stock_price_high) > highest_price) { highest_price =
parseFloat(doc.stock_price_high); print("highest_price" + highest_price); }
    }
  });
  return { highest_stock_price: highest_price };
};
```

The reduce function receives two arguments: a key and an array of values. In the context of the current example, a stock with symbol `"FDI"` will have a number of different key/value pairs from the map phase. Some of these would be as follows:

```
    (key : "FDI", { "stock_price_high" : 11.11 })
    (key : "FDI", { "stock_price_high" : 11.18 })
    (key : "FDI", { "stock_price_high" : 11.08 })
    (key : "FDI", { "stock_price_high" : 10.99 })
    (key : "FDI", { "stock_price_high" : 10.89 })
```

Running a simple count as follows: `db.nyse.find({stock_symbol: "FDI"}).count();`, reveals that there are 5,596 records. Therefore, there must be as many key/value pairs emitted from the map phase. Some of the values for these records may be undefined, so there may not be exactly 5,596 results emitted by the map phase.

The reduce function receives the values like so:

```
    reduce('FDI', [{stock_price_high: 11.11}, {stock_price_high: 11.18},
     {stock_price_high: 11.08}, {stock_price_high: 10.99}, ...]);
```

Now if you revisit the reduce function, you will notice that the passed-in array of values for each key is iterated over and a closure is called on the array elements. The closure, or inner function, carries out a simple comparison, determining the highest price for the set of values that are bound together by a common key.

The output of the reduce phase is a set of key/value pairs containing the symbol and the highest price value, respectively. There is exactly one key/value pair per stock symbol. MongoDB allows for an optional finalize function to pass the output of a reduce function and summarize it further.