ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/337801616

Benchmarking and Performance Modelling of MapReduce Communication Pattern

Conference Paper · December 2019

DOI: 10.1109/CloudCom.2019.00029

CITATIONS
0

READS
102

3 authors, including:

Sheriffo Ceesay
University of St Andrews

4 PUBLICATIONS **8** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Master Thesis View project

All content following this page was uploaded by Sheriffo Ceesay on 06 December 2019.

The user has requested enhancement of the downloaded file.

Benchmarking and Performance Modelling of MapReduce Communication Pattern

Sheriffo Ceesay, Adam Barker and Yuhui Lin
University of St Andrews, UK
Email: {sc306, adam.barker, yl205 }@st-andrews.ac.uk

***Abstract*—Understanding and predicting the performance of big data applications running in the cloud or on-premises could help minimise the overall cost of operations and provide opportunities in efforts to identify performance bottlenecks. The complexity of the low-level internals of big data frameworks and the ubiquity of application and workload configuration parameters makes it challenging and expensive to come up with comprehensive performance modelling solutions. In this paper, instead of focusing on a wide range of configurable parameters, we studied the low-level internals of the MapReduce communication pattern and used a minimal set of performance drivers to develop a set of phase level parametric models for approximating the execution time of a given application on a given cluster. Models can be used to infer the performance of unseen applications and approximate their performance when an arbitrary dataset is used as input. Our approach is validated by running empirical experiments in two setups. On average the error rate in both setups is ±10% from the measured values.**

Communication Patterns, Modelling, MapReduce, Big Data

I. Introduction

Traditional data storage and processing like Relational Database Management Systems (RDBMS) by design are inefficient and rigid to store and handle big data. Over the years researchers have developed big data processing frameworks and storage systems to handle these challenges. Like any new system, it is crucial to model and understand its performs under various conditions. This could be a tedious and challenging task because big data application under the hood runs on frameworks comprising of complex and complicated pipeline of phases and operations. For example, applications deployed using MapReduce [5] and Apache Spark [20] frameworks go through several computation or communication stages. Phases and stages on a basic level are the transitions that the data goes through while being processed. More details on MapReduce phases are discussed in Section II. Considering this challenge, we argue that a phase by phase modelling approach can gain a better understanding of the performance drivers of such frameworks. Some of the previous works in this area focuses only on high-level performance drivers like data size [7, 8]. This approach may work well for some applications. For example, from sort operations shown in Figure 1, we can see a clear linear relationship between the data size and the time taken to complete the execution. In this case, the input data size is equal to the output data size. However, as evident word count (WC) plots, the relationship between processing time and input data size is non-trivial and therefore there need for more investigation. The amount of data that passes through the various phases of the MapReduce pattern is crucial to its overall performance. This leads us to dig further into the low-level details of the framework. This work can also serve as a foundation for performance modelling of big data frameworks like Apache Spark, a general purpose computation engine that does of superset of what MapReduce does. The findings could be useful in size-based schedulers to run small jobs even when the cluster is loaded with long running and expensive jobs.

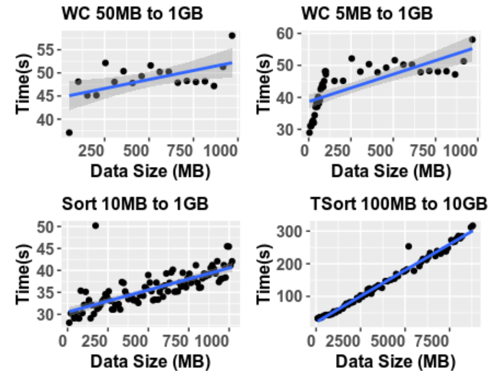

Fig. 1: **Execution time for varying datasets for MapReduce WordCount, TeraSort and Simple-Sort Programs**

The research questions we seek to address in this paper are:

- RQ0: Considering the ubiquity of MapReduce workloads, can we use a generic approach to model the performance of MapReduce applications?
- RQ1: Considering how expensive it is to improvise a production size Hadoop cluster, can we identify major performance drivers for the MapReduce framework and use them to model the performance of applications?
- RQ2: Can we measure how good the effect of RQ1 would have on an arbitrary cluster?

In this paper, we develop a parametric model to estimate execution time of a given MapReduce application for a given cluster of nodes running YARN [14] (MapReduce 2.0 or greater). To achieve this, we first study and identify the performance drivers of each generic phase of the MapReduce application execution life cycle. This is achieved by executing generic MapReduce workloads to collect relevant performance metrics from logs. The combination of varying modelling approaches and cross-validation methods are used to obtain

the models with their respective parameters. We have also developed a proof of concept application to test the models with different datasets. This links for the application and source code are is available at:
https://sc306.host.cs.st-andrews.ac.uk/cbm/ and
https://github.com/sneceesay77/mr-performance-modelling

The main contributions of our work are:

- A parametric models for predicting the execution time of a big data application running on a YARN cluster.
- A generic performance modelling approach to model the execution time of MapReduce (YARN) [14] applications tested on different clusters and data sizes.
- An insight into the performance characteristics of the generic phases of MapReduce.
- An understanding of how MapReduce applications of different design patterns perform.

II. Background: The MapReduce Pattern

MapReduce [5] initiated by researchers from Google is one of the most prevalent patterns in big data processing. The open-source community then created an open-source version of MapReduce known as Hadoop MapReduce. MapReduce 2.0 uses YARN [14] (Yet Another Resource Negotiator) for cluster resource management. In the context of YARN, a container is as a logical unit of computation assigned with CPU and memory where individual MapReduce tasks of an application are executed. They run on nodes and are managed by a Node Manager process. The node manager sends periodic reports of task progress to Application Master which monitors and manages the entire application progress. After a successful resource allocation by YARN's Resource Manager, the MapReduce execution starts by first reading data read from HDFS [2] and fed to mappers for processing. The mappers then write their intermediate results to disk. If a reducer is defined, these results are shuffled and processed by the reducers, and the final results are written to the disk. As shown in Figure 2, the time taken by a job depends on the execution of the last reducer as there is a write barrier at the end of a job. Although there are faster and newer big data processing frameworks like Apache Spark [20] MapReduce is still relevant in the big data processing domain.

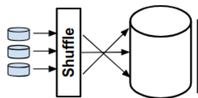

Fig. 2: **The MapReduce Communication Pattern**

In order to define a representative model to predict the performance of a MapReduce application, all the phases shown in in Figure 3 should be considered. The phases can be grouped into user-defined and framework-defined phases. User-defined phases are implemented by the programmer to provide customised functionality, e.g. the *map()* and *reduce()* functions. Framework-defined phases are defined and controlled by the framework, and users cannot modify those functionalities. Examples are *read*, *collect*, *spill*, *merge*, *shuffle* and *write*. In Section III, we will present detailed explanation for each of the phases of the MapReduce pipeline illustrated in Figure 3.

III. Theoretical Models of The MapReduce Pattern

In this chapter, we present the lessons learnt from the in-depth exploration of the MapReduce source code and its internals.

A. *Map Phase*

As shown in Figure 3, the Map Phase consists of the following sub-phases: *Read*, *Map*, *Collect*, *Spill* and *Merge*.

*1) **Read**:* This phase reads a configurable input split or data block from the *Hadoop Distributed File System* (HDFS). Each record in the input split is then sent to a customised $map()$ function for processing. The cost in the read phase depends on the size of the input split. Given a specific cluster, we can define the cost model of the read phase with a linear function of the size of the input split:

$$T_{read} = f(d) = \beta_{0r} + \beta_1 d \quad (1)$$

where d represents the size of the data and β_0 and β_1 represents the unknown parameters of the linear function. However, for a given MapReduce job, the time taken to read each block of 128MB of data is usually the same.

*2) **Custom Map**:* This is the second sub-phase of the Map Phase. The map function contains a user-defined code to specify how to process records consumed from the input splits. Since this phase is user-defined, we, therefore, devised a way to approximate the time as shown below.

$$T_{map} = \frac{\sum_{i=1}^{n} T_{rec}}{M} / N_c \quad (2)$$

Where n is the number of records in the input split fed to the map function, T_{rec} is the time taken to process each record, M is the number of mappers and N_c is the number of containers.

Moving forward, the amount of data that passes through each stage is crucial to the accuracy of the model, therefore the data emitted by the map phase M_d should also be calculated as shown in Equation (3). d is the total input data for task t and M_{sel} is the map selectivity ratio

$$M_d = d * M_{sel} \quad (3)$$

*3) **Collect**:* The output of each map task is not directly written to disk and consumed by the reducers. Instead, they are buffered and presorted in memory. This phase also depends on the amount of data emitted by the mappers. A linear cost model of the collect phase is given as below:

$$T_{collect} = f(M_d) = \beta_{0c} + \beta_3 M_d \quad (4)$$

*4) **Spill**:* In this phase, map output data is partitioned, sorted in memory and written to local disks. Writing data to local disk is the main bottleneck. The more data to spill, the more time it will take. We can, therefore, represent the cost model using the linear relationship shown below:

$$T_{spill} = f(M_d) = \beta_{0s} + \beta_4 M_d \quad (5)$$

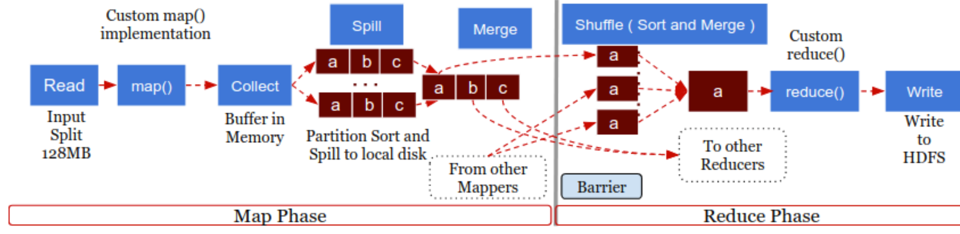

Fig. 3: **A MapReduce Workflow For Each Task [19]. The blue blocks are the different phases, whiles the brown blocks are the process data in transition from one stage to another.**

*5) **Merge**:* Each time the collect buffer reaches its configurable threshold, a new spill file is created. Similar to the spill phase, the cost depends on the amount of data to merge-sort and write back to disk. From the analysis Hadoop source code, this phase uses the merge-sort algorithm which has an average complexity of $nlogn$. Since the focus of this work is to identify key performance drivers, we ignore the less significant I/O cost at this stage. Hence, the relationship to model this phase is:

$$T_{merge} = f(M_d logM_d) = \beta_{0m} + \beta_5 M_d(logM_d) \quad (6)$$

B. *Reduce Phase*

As shown in Figure 3, Reduce Phase of each task consists of the following sub-phases: *Shuffle*, *Reduce* and *Write*. The cost analysis of each phase will be discussed as follows.

*1) **Shuffle**:* When all map outputs have been copied, data are then merged into larger ones while maintaining their sorting order to be consumed by the next stage. To model this phase, we calculated the data that are shuffled through to each reducer. The total data processed by each reducer can be represented as:

$$S_d = \frac{d * M_{sel} * M_t}{R_t} \quad (7)$$

Where M_t is the total number of mappers, and R_t is the total number of reducers. Using Equation (7), the cost model of the shuffle phase can be formulated as:

$$T_{shuffle} = \beta_{0f} + \beta_6 S_d + \beta_7 M_t \quad (8)$$

*2) **Custom Reduce**:* For each key in a data partition, the reduce function is executed. We use the formula below to approximate the time it takes for the reduce phase to complete.

$$T_{reduce} = \frac{\sum_{i=1}^{n} T_{key}}{R_t} / N_c \quad (9)$$

Here, N_c is the total number of containers.

*3) **Write**:* This is the last phase of the MapReduce pipeline. The output of custom reduce function is collected and written to HDFS. To model this phase, first, we modelled the total data that the reducer task emits, i.e.,

$$R_d = S_d * R_{sel} \quad (10)$$

where S_d is the total shuffle data fed to a reduce task and R_{sel} is the ratio of input and output sizes. Using the relations in Equation (10), we can, therefore, define the cost model of the write as a linear function of the reduce output data size, which is:

$$T_{write} = f(R_d) = \beta_{0w} + \beta_8 R_d \quad (11)$$

C. *Combining it All Together*

Now that we have proposed the cost model for the individual phases of the MapReduce pipeline. Putting them all together for the two main phases, we have:

$$T_{mt} = T_{read} + T_{map} + T_{collect} + T_{spill} + T_{merge} \quad (12)$$

Now substituting the phases with their corresponding linear cost models and replacing all the constants β_{0*} as β_0 together to have

$$T_{mt} = \beta_0 + \beta_1 d + \beta_2 d + \beta_3 M_d + \beta_4 M_d + \beta_5 M_d logM_d + T_{map} + \epsilon$$

Similar to the Map Phase, we can combine all the initial formula in the sub phases of the reduce phase, i.e.,

$$T_{rt} = T_{shuffle} + T_{reduce} + T_{write} \quad (13)$$

Putting the cost models together, we have:

$$T_{rt} = \beta_0 + \beta_6 S_d + \beta_7 S_d + \beta_8 R_d + T_{reduce} + \epsilon \quad (14)$$

D. *Cost Model For The Entire Process*

The models presented in the previous section represent models for a single task. MapReduce runs several tasks in parallel. Depending on the system resources, all task may run in one round, or in most cases, there will be several rounds of tasks before the entire job finishes. Before YARN, map and reduce slots were used to determine the number of tasks that can run concurrently. This approach does not fully utilise the cluster resource and previous performance models using this approach cannot be applied to YARN-based MapReduce application. YARN uses *containers* for task execution. The number of containers in a cluster determines the number of tasks that can run concurrently. Assuming that a cluster has 20 containers N_c and job j has a total of 100 map tasks M_t, there will be at least five rounds of 20 map phase with each phase running 20 tasks concurrently. Using this logic we can modify both the Map and Reduce final formulas as follows:

$$\begin{aligned} T_{mt} &= \frac{M_t}{N_c}[T_{read} + T_{map} + T_{collect} + T_{spill} + T_{merge}] \\ &= \frac{M_t}{N_c}[\beta_0 + \beta_1 d + \beta_2 d + \beta_3 M_d + \beta_4 M_d + \\ &\quad \beta_5 M_d logM_d + T_{map}] + \epsilon \end{aligned} \quad (15)$$

Similarly, the same can be done for the reduce phase:

$$\begin{aligned} T_{rt} &= \frac{R_t}{N_c}[T_{shuffle} + T_{reduce} + T_{write}] \\ &= \frac{R_t}{N_c}[\beta_0 + \beta_6 S_d + \beta_7 S_d + \beta_8 O_d + T_{reduce}] + \epsilon \end{aligned} \tag{16}$$

The final cost model of the entire job can be obtained by merging Equation (15), Equation (16), the custom phases (Equation 2 and Equation 9).

$$\begin{aligned} T_{job} &= [\frac{M_t}{N_c}[T_{mt}] + \frac{R_t}{N_c}[T_{rt}]] + \epsilon \\ &= \frac{M_t}{N_c}[\beta_0 + \beta_x{}'d + \beta_y{}'M_d + \beta_5 M_d log M_d + T_{map}] + \\ &\quad \frac{R_t}{N_c}[\beta_0 + \beta_z{}'S_d + \beta_8 R_d + T_{reduce}] + \epsilon \end{aligned} \tag{17}$$

IV. Phase Profiling Methodology

The profiling process is inspired by [21], but different in terms of the underlying system, workloads used and the experimental approach. It includes running generic benchmarks on the target cluster multiple times with various configurations. For brevity, we assume that the network is stable, no node failure and a non-shared environment and therefore we have configured YARN to use the FIFO [14] scheduling algorithm. For each phase of MapReduce, data is collected from the corresponding YARN logs on the cluster.

$$BM_i = (D_i, M_{sel_i}, B_{size_i}) \tag{18}$$

Each generic benchmark has the following parameters: D_i is the size of the input dataset, M_{sel} represents the map selectivity and B_{size} represents the block-size. For each profiling step, we vary D_i to read data size ranging from 500MB to 5GB with an interval of at most 500MB. This parameter mainly profiles the read phase. We use M_{sel} to parameterise map selectivity which is the ratio of map input to the map output. It is the amount of data that proceeds to the later stages of the MapReduce pipeline. For each input data size we vary the M_{sel} from 10% to 100% using 10% interval.

This parameter affects the collect, spill, merge and shuffle phases. We also vary B_{size} using 64MB and 128MB. For example, for a given input data size of 5GB, we executed the benchmark 20 times (10*2, where 10 is from the various value of M_{sel}, and 2 is from values of B_{size}).

As shown in Figure 4, there are three main steps involved in the generic benchmarking setup. First, we modified the MapReduce 3.0 source code and added the necessary codes to obtain the running times of each stage. Since MapReduce uses counters to present task and job-related statistics to the user, we added six new counters to represent the six generic phases. The Mapper $interface$ has a logic in the $run()$ function that adheres to the map selectivity setting. The reduce function outputs the entire shuffled data. To implement the logic of map selectivity, we used the *Teragen* program provided by the MapReduce framework to generate the dataset. Each row generated by Teragen has a size of 100 bytes. Therefore, to process only 10% of a 500MB input, we invoked Teragen to generate [(500MB to Bytes) /100] rows, and we override the $run()$ function of MapReduce interface to stop after the map had processed the 10% threshold. As shown in Figure 4, the second stage in the pipeline is the YARN log parser. For each job and tasks, YARN logs are collated, aggregated and parsed to extract the relevant counters and their respective values for further processing. These new data are grouped into the various phases of the MapReduce framework. The final stage of the phase profiling pipeline uses linear regression to generate the actual model parameters using the R statistical programming language

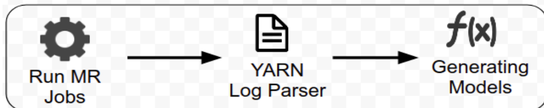

Fig. 4: **A Phase Profiling Pipeline Methodology**

V. Generating the Actual Models and Parameters Using Regression and Cross Validation

Figure 5 gives the results of the models on the processed YARN log data, using a 10 fold cross-validation on each dataset. The observations for each are discussed in the following subsections. The plot shows the results of the predicted and actual values. To select the right parameters and evaluate the accuracy of the model of each phase, we consider the best practices for model selection. For each model, we study the effect of p-value, *Root Mean Squared Error* $RMSE$ [1], $Adjusted - R - squared$ [10] and $Multiple - R - squared$ [10]. First, to select the most important parameters of the model, we performed backwards-elimination and accept any variable whose p-value ≤ 0.05. $RMSE$ measures the standard deviation of the residuals or prediction errors. In determining the model accuracy, we evaluate how small the value of $RMSE$ is considering the range of the dependable variable we are using. In our case, the dependent variable is *execution time* represented on the Y-axis of Figure 5.

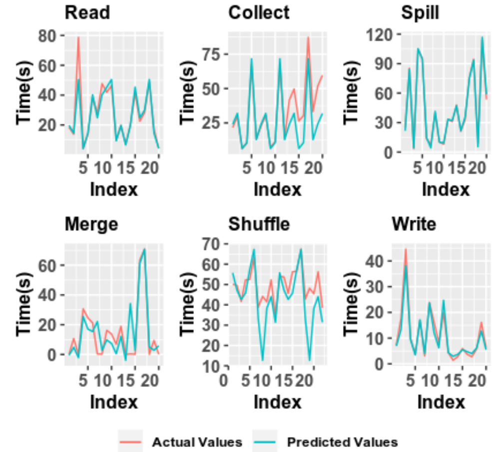

Fig. 5: **Results of 10 Fold Cross-Validation and Prediction on the 8 Nodes cluster**

Our general assumption is that in most cases the performance of these phases depends on the amount of data that it processes. This assumption is confirmed in Figure 6. We can see that, in most cases as we increase the size of the dataset, the processing time generally increases and therefore a possible linear relationship. The last two phases in Figure 6 i.e. shuffle and write have a less linear relationship compared to the first four phases. To select the best learning method, we modelled the data using different Machine Learning (ML) approaches. As shown in Table I, the best ML approach for both the Shuffle and Write phase is the support vector machine algorithm but not by a wide margin compared to linear regression. The algorithm has the highest R-squared value and the best RMSE in our case. Note that for equations all data measurements are in megabytes and time in milliseconds.

TABLE I: RMSE and R-Squared values for Shuffle and Write

Algorithm	RMSE(ms)	R-Squared	Phase
SVM	4655.42	0.96	Shuffle
Random Forest	5027.23	0.95	Shuffle
Decision Tree	8370.37	0.78	Shuffle
Linear Regression	4774.75	0.95	Shuffle
SVM	2747.380	0.94	Write
Random Forest	2940.09	0.93	Write
DecisionTree	9366.400	0.68	Write
Linear Regression	6256.47	0.88	Write

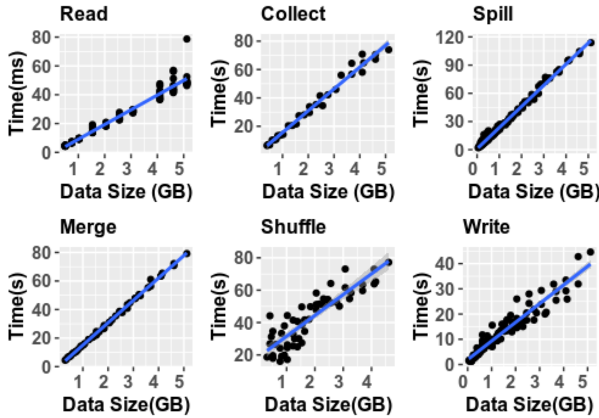

Fig. 6: **A plot and best fit for each of the generic phase: The main purpose of this plot is to show how performance(ms) changes with respect to data size (MB).**

A. *Read Model*

The Read Phase of Figure 6 supports our initial assumption in (1) that there is a linear relationship between the size of input data and the processing time. The model obtained using linear regression is presented in Equation (19). The read plot in Figure 5 and $RMSE$ value of 4.08 proves a strong case for the accuracy of the model.

$$T_{read} = 0.01 \times D + 1.33 \quad (19)$$

B. *Collect Model*

Map output data is collected as soon as map tasks complete; once the circular buffer of each map task reaches the predefined threshold of 80% data is then written to disk. This behaviour can be observed from Figure 6, a spike in the amount of time taken is at 80% M_{sel}. The model for the collect phase in our test environment is illustrated as below:

$$T_{collect} = 0.01 \times M_{sel} + 0.97 \quad (20)$$

Again, the $RMSE$ value of 3.036 and the Collect plot in Figure 5 supports our model.

C. *Spill Model*

As shown in Figure 6, our linearity assumption between data size and processing time is confirmed. The model obtained after the cross-validation is presented in Equation (21). The model has $RMSE$ value is 2.522 which would be considered a good model in this case.

$$T_{spill} = 0.02 \times M_{sel} + 0.98 \quad (21)$$

D. *Merge Model*

Spill files are merged together to form bigger files. From the plot, we can see a close relation between the actual and the predicted values with an $RMSE$ of 9.36. The parameters of the model are shown in Equation 22

$$T_{merge} = 0.002 \times M_{sel} log M_{sel} + 4.80 \quad (22)$$

E. *Shuffle Model*

This phase is the most expensive of all because it involves inter-node communication. The model main parameters are the amount of data being shuffled and the number of Mappers task. The $RMSE$ value obtained after is 4655.

$$T_{shuffle} = 10.45 \times S_d + 579.48 \times M_t + 6144.6^1 \quad (23)$$

F. *Write Model*

From Figure 6 (Write Phase), the fitted line shows a less-linear relationship between the amount of data and the time taken to write that data to HDFS. The cross-validation plot also shows a strong relationship when the test values are used in the predictive model. The model has an acceptable $RMSE$ of 2427

$$T_{write} = 6.94 \times R_d + 2139.98 \quad (24)$$

G. *Custom Map and Reduce*

Since these phases have a custom implementation that depends on the problem being solved, we, therefore, used Equation (2) and Equation (9) for approximation.

H. *Generating the Input Parameters*

With the parameters from profiling, T_{job} in (17) is:

$$\frac{M_t}{N_c}[(0.01 \times D + 1.33) + T_{map} + (0.01 \times M_{sel} + 0.97) + (0.02 \times M_{sel} + 0.98) + (0.002 \times M_{sel} log M_{sel} + 4.80)] + \frac{R_t}{N_c}[(10.45 \times S_d + 579.48 \times M_t + 6144.6) + T_{reduce} + (6.94 \times R_d + 2139.98)] + \epsilon$$

Users now need to provide the actual values for the input parameters listed in Table II. To get the values for each of

[1]Linear equation, however we used the SVM model in the prediction.

TABLE II: Metrics Extracted From Logs

Name	Variable	Description
Num Of Bytes Read	D	Total Size of Data
Map Output Bytes	M_d	Data Outputted by Mappers
Map Selectivity	M_{sel}	$(M_d/d) * 100$
Bytes Shuffled	S_d	Data shuffled, refer to Eq: (7)
Bytes Written	R_d	Data Outputted by Reducers
Total Mappers	M_t	$\frac{D}{BlockSize}$
Total Reducers	R_t	Optimisable
Number of Containers	N_c	Inferred from Cluster Config.
Map Time	T_{map}	Total time map() fns
Reduce Time	T_{map}	Total time reduce() fns

these parameters, the first step would be to run an application on the profiled cluster using a representative input dataset for the application. The application logs are then parsed to the YARN log parser to collect the values of the parameters for substitution into the corresponding models. This would provide the input parameters to generate the final predictive model. To illustrate, we take the Reduce Side Inner Join algorithm as a target application. The generated values for each of the input parameters are listed in Table III. To facilitate the generation of the values from YARN logs we provided Java and R scripts available on our Github page.

TABLE III: The values for each parameters for Reduce Side Inner Join algorithm extracted from YARN logs.

Variable	Value
D	19584
M_t, R_t	153,11
$T_{map}, T_{reduce}(ms)$	33069,286257
M_d	128
M_{sel}	100%
S_d	19584
R_d	19584
N_c	8

VI. Experiment and Evaluation

A. *Setup*

In order to gauge the applicability of our approach on an arbitrary setup, we conducted two sets of experiments on two different hardware setups. On the first setup, we used a single node YARN cluster with 32GB memory and 8 CPUs. YARN was allocated 24GB and a minimum of 3GB per container In the second setup, we used an 8-node in-house YARN cluster to mimic a real-world deployment scenario. Each of these nodes has 8GB of RAM, 8 vCPUs and a 500GB of storage space. YARN is allocated 48GB of memory, and a maximum of 8 containers can run at a time. From the results of the two setups for the experiments, we have a strong case to conclude that, the approach can be used to profile an arbitrary cluster size. In all the experiments, we have adapted the algorithms discussed in the design patterns book [11] and used the same $StackOverflow$ data source but a more updated version. Table IV summarises all the workloads and their data-sizes used in this work. To have a representative cost model for the various MapReduce phases and the entire process, we studied the most common design pattern algorithms of MapReduce presented in [11]. The motivation of using MapReduce design pattern approach in the experimental setup is to avoid bias towards specific applications only. As shown in Table IV, we have only included large datasets in the experiment to narrow our focus on big data and data-intensive applications. We have identified four common design patterns, and for each of these patterns, we tested at least three algorithms to show how algorithms using the same pattern relates to one another. We assume that algorithms with the same pattern would have similar performance characteristics.

- Summarisation Pattern: This pattern provides a summary of an input dataset. The popular MapReduce program is a good example.
- Filtering Pattern: This pattern filters and returns a subset of a given original dataset. In most cases, there is a reduction in the amount of output dataset compared to the input dataset. Example of algorithms in this pattern are Distributed $Grep$, $Distinct$, and $Top\ K$.
- Data Organisation Pattern: This pattern deals with the reorganisation from one structure to another. Example, transforming table data to JSON structure.
- Join Pattern: This pattern processes related data stored in the input files. The most popular type of Join in MapReduce is the Reduce side join. It works in all cases but can be slow as the size of the data increases.

TABLE IV: Workloads Used

Algorithm	Design Pattern	Data Size
MinMaxCount	Summarisation	16GB
Average Count	Summarisation	16GB
Median and Std.	Summarisation	16GB
Inverted Index	Summarisation	12GB
Grep	Filtering	12GB
Top X	Filtering	2.7GB
Distinct	Filtering	16GB
Structure to Hierarchy	Data Organisation	12GB,16GB
Total Order Sorting	Data Organisation	2.6GB
Shuffling	Data Organisation	2.7GB
RSJ Inner	Join	16,2.7GB
RSJ Left Outer	Join	16,2.7GB
RSJ Right Outer	Join	16,2.7GB
RSJ Full Outer	Join	16,2.7GB

B. *Evaluation of Results*

For each of the two setups, the same experiments are repeated and the results are plotted and discussed side by side. As expected, the time taken by the single node cluster for each of the algorithms is mostly greater than the eight node cluster. Below, we discuss the rest of the results in details.

1) Summarisation Pattern: The algorithms and the results obtained included are listed in V and corresponding plot in Figure 7. The percentage prediction error for both setups is less than 16%. Also note that the algorithms in this pattern have a similar completion time boundaries.

TABLE V: Results for Summarisation on 8-Node Cluster

Algorithm	Predicted (sec)	Actual (sec)	Error%
MinMaxCount	196	201	2
Inverted Index	118	117	-1
Average Count	157	176	11
Median and Std. Dev	158	169	7

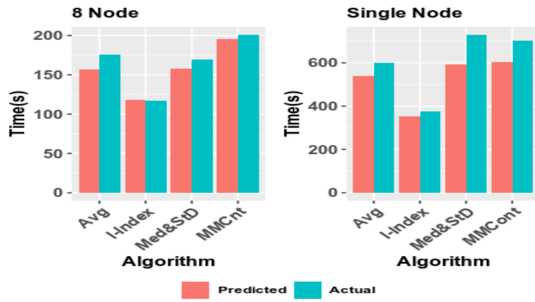

Fig. 7: Summarisation Pattern

2) Filtering Pattern: In filtering, we included, Grep, Distinct and Top 100 algorithm. Table VI shows the results of the experiment and the respective errors of each algorithm. We have also observed that our prediction error is less than 14% from the observed values for both setups.

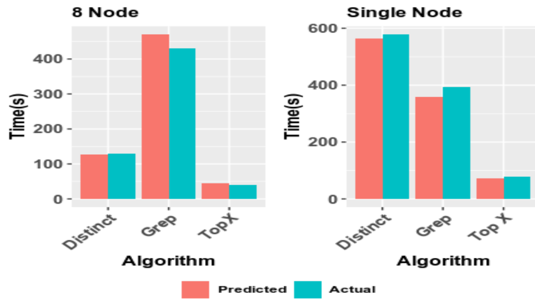

Fig. 8: Filtering Pattern

TABLE VI: Results for Filtering Pattern on 8-Node Cluster

Algorithm	Predicted (sec)	Actual (sec)	Error%
Grep	470	430	-9
Top X	45	40	-12
Distinct	128	130	2

3) Data Organisation Pattern: In data organisation, we have observed that the amount of input data is mostly the same as the amount of output data. Here data is just reorganised and there is no data pruning component. The Question and Answer Hierarchy algorithm merged data from two big data sets. For each Question posted in Stackoverflow, the corresponding answers are collated from the Post file. The results of the three executed experiments are illustrated in Figure 9 and Table VII.

TABLE VII: Results for Data Organisation Pattern

Algorithm	Predicted (sec)	Actual (sec)	Error%
Q&A Hierarchy	1653	2024	18
Total Order	162	160	-1
Anonymise & Shuffling	131	125	-8

4) Join Pattern: Joins are one of the most expensive operations in MapReduce, this is evident in our experimental results. We executed inner, left-outer, right-outer and full-outer joins using the Reduce Side Join Algorithm. We used the Post and Comments dataset curled from Stack Overflow. The

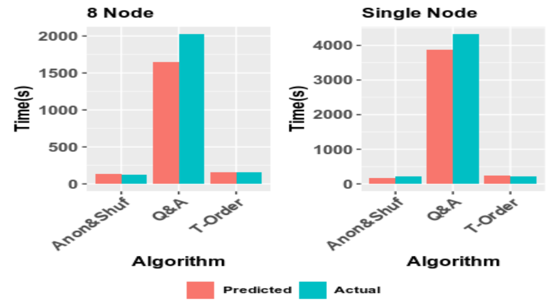

Fig. 9: Data Organisation Pattern

prediction error for both setups is at most 10% of the observed value. The results are shown in Table VIII and Figure 10.

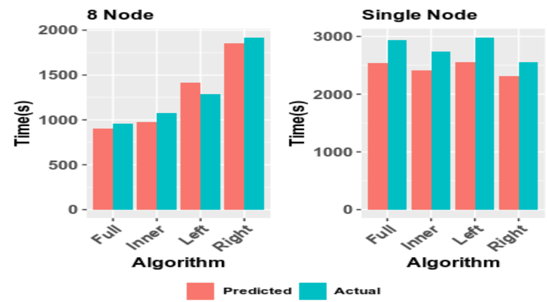

Fig. 10: Join Pattern

TABLE VIII: Experiment Data for Join Pattern

Algorithm	Predicted (sec)	Actual (sec)	Error%
Reduce Side Inner	975	1080	10
Reduce Side L-Outer	1410	1285	-10
Reduce Side Right Outer	1854	1920	3
Reduce Side Full Outer	900	960	6

C. Discussion

Here we revisit our three research questions (RQs) and evaluate how far we have answered them. In answering RQ0, we adopted a generic benchmarking approach by running dummy MapReduce workload on the cluster. The main idea behind this is to avoid benchmarking bias when only a particular set of workloads are used. In addressing RQ1, we have realised that out of the many variables, input parameters such as the number of mappers, number of reducers, number of containers and the amount of data that passes through each stage are the key performance drivers. The final model building process involves feeding these key performance drivers to ML algorithms. In addressing RQ2, we scaled our experimental environment from a single node setup to an eight-node cluster. We have observed that the approach we have used can be replicated to arbitrary cluster size.

VII. Related Work

Standard benchmarks such as TPC-DS [12] has been used in the research community to evaluate the performance of

decision support systems. Big data Benchmarking and Performance modelling recently got lots of attention from researchers [6, 7, 8, 9, 17]. Huang et al. [9] and Wang et al. [18] developed benchmark suites for Hadoop, Spark and Streaming Frameworks. These suites consist of a set of workloads organised into related groups. These workload groups span from basic statistics, machine learning, graph processing and SQL. However, this approach makes it inefficient and cumbersome to test new workloads. Also, as stated in Ceesay et al. [3], deploying these tools can be cumbersome for non-technical individuals. Therefore it becomes a challenge for wider adoption. Zhang et al. [21], presents a MapReduce performance model that measures generic phases of the framework. However, their work focused on an older version of Hadoop (0.20.0) which uses mapper and reducer slots for job processing. The current MapReduce framework uses YARN which efficiently manages cluster resources. Given this fact, it is clear that the approach and the model they used cannot be representative of the current MapReduce paradigm. Secondly, our experimental approach differs as well, while they pick any big data application, we picked ours grouped by the algorithms design patterns, the results of which show some interesting correlation in terms of their performance. Applications in the same pattern mostly have similar performance characteristics in terms of their execution time. Verma et al. [16] proposed ARIA, a job and resource scheduler for the MapReduce framework that aims to allocate the right amount of resource to meet required service level objectives (SLOs). In their work, they extracted information from MapReduce logs as a basis for their framework. Like [21], they used Hadoop 0.20.0, and therefore their approach will not be applicable for Hadoop 2.0 or later versions. Venkataraman et al. [15], built performance models based on a small sample of data and predicting on larger datasets and cluster sizes. However, they focus on a small subset of machine learning algorithms. Popescu et al. [13] introduced an approach for predicting the runtime JAQL queries which focuses on mainly JSON and related data.

VIII. Conclusion and Future Work

In conclusion, we have shown that we can model the performance of big data application in the MapReduce pattern. We have shown that algorithms in the same pattern tend to have similar performance characteristics when the data size is similar. We have also proposed a generic benchmarking approach that can be used to get performance characteristics of the various stages in a MapReduce pipeline. Using the data generated from the benchmarking, linear regression and cross-validation were used to build and validate the models for each phase. These models were used to predict the competition time for a new application, and the results were promising. In future work, we will work on modelling other communication patterns proposed in [4] such as Data Flow with Cycles, which performs computations and subsequent transformations in memory until the user actively persists them to disk. We will also consider validating our approach with a much larger cluster and datasets. Finally, since scheduling and scalability are an integral part of big data systems, we will also study their effects on the performance of our modelling approach.

IX. Acknowledgement

This research is funded by EPSRC EP/R010528/1 and IsDB.

References

[1] Anthony G Barnston. Correspondence among the correlation, rmse, and heidke forecast verification measures; refinement of the heidke score. *Weather and Forecasting*, 7(4):699–709, 1992.

[2] Dhruba Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 53:1–13, 2008.

[3] S. Ceesay, A. Barker, and B. Varghese. Plug and play bench: Simplifying big data benchmarking using containers. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2821–2828, Dec 2017. doi: 10.1109/BigData.2017.8258249.

[4] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36. ACM, 2012.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] Jack J Dongarra, Cleve B Moler, James R Bunch, and Gilbert W Stewart. *LINPACK users' guide*. SIAM, 1979.

[7] Daria Glushkova, Petar Jovanovic, and Alberto Abelló. Mapreduce performance model for hadoop 2. x. *Information Systems*, 2017.

[8] Herodotos Herodotou. Hadoop performance models. *arXiv preprint arXiv:1106.0940*, 2011.

[9] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *(ICDEW), 2010 IEEE 26th*, pages 41–51. IEEE, 2010.

[10] Jeremy Miles. R-squared, adjusted r-squared. *Encyclopedia of Statistics in Behavioral Science*, 2005.

[11] Donald Miner and Adam Shook. *MapReduce Design Patterns*. O'Reilly Media, Inc., 1st edition, 2012. ISBN 1449327176, 9781449327170.

[12] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1049–1058. VLDB Endowment, 2006.

[13] Adrian Daniel Popescu, Vuk Ercegovac, Andrey Balmin, Miguel Branco, and Anastasia Ailamaki. Same queries, different data: Can we predict runtime performance? In *2012 IEEE 28th ICDE Workshops*, pages 275–280. IEEE, 2012.

[14] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[15] Shivaram Venkataraman, Zongheng Yang, Michael J Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, pages 363–378, 2016.

[16] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM ICAC*, pages 235–244. ACM, 2011.

[17] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. Resource provisioning framework for mapreduce jobs with performance goals. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 165–186. Springer, 2011.

[18] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *HPCA, 2014 IEEE 20th*, pages 488–499. IEEE, 2014.

[19] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[20] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[21] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. Benchmarking approach for designing a mapreduce performance model. In *Proceedings of the 4th ACM/SPEC ICPE*, pages 253–258. ACM, 2013.

View publication stats