CLICK THROUGH RATE PREDICTION PROBLEM: ANALYZING THE SCALABILITY OF A LOGISTIC REGRESSION-BASED SOLUTION USING APACHE SPARK

Meethil Vijay Yadav
Department of Computer Science,
University at Buffalo.

Daniel Nazareth
Department of Computer Science,
University at Buffalo.

Abstract:

Online advertising is a billion-dollar industry consisting of three major players- Publishers such as the New York Times, ESPN etc. which make money by displaying ads on their websites, advertisers, typically product based companies which pay to have their products displayed on the publisher's page and matchmakers such as Google, Yahoo, Microsoft etc. which decide dynamically which kind of ads to display for various search and other pages and earn revenue based on how often a user clicks. Since user engagement can easily go as low as 1%, the click through rate prediction problem aims to estimate the conditional probability that a user will click on an ad based on a massive dataset of predicted features such as ad content, historical performance, user and publisher specific information wherever possible and much more. We address the problem using logistic regression and analyze the scalability and efficiency of our solution on a dataset of approximately 40 million rows of anonymized user-ad interaction data.

Spark can be used using various configurations of the environment. The parameters that can be varied are nodes, executors, cores per executor, and memory per executor. We show that performance is highly dependent on the configuration of spark. We have tried to find the optimal values of these parameters for the logistic regression.

1. Implementation

We implement, test and tune our solution to the click through rate problem using Apache Spark, a parallel processing engine capable of efficiently performing fine-grained transformations and calculations on massive datasets. Spark also comes equipped with a powerful Machine Learning library MLib which we draw upon when running logistic regression on our dataset. Once we have successfully trained a model on the dataset using logistic regression, we then assess the scalability of the solution and attempt to tune various Spark in order to evaluate their effect on the solution's overall performance. In summary, our implementation steps are:

- Reading in the dataset and cleaning if necessary.
- Converting categorical features of the dataset into numerical ones so that we can run logistic regression on them.
- Evaluating strong and (if necessary) weak scalability.
- Exploring the tunable, fine-grained controls available within the Spark infrastructure to find the fastest possible implementation of our solution.

Feature hashing: One of the biggest challenges of the solution is finding a way to convert the anonymized categorical data in our problem to a corresponding numerical representation, which can then be inputted into our logistic regression function. The two frequently used methods for this type of conversion for the purposes of the machine learning are called *one-hot encoding* and *feature hashing.* We use the latter method since it gives us a much more concise final set of numerical features.

Feature hashing as the name suggests is based on a method of data mapping using hash functions which are used frequently in computer science applications to map objects to specific containers (called buckets) in such a way that looking up this data takes on average O(1) i.e. constant time. The buckets are represented by a large vector initialized to all zeroes. In the context our application every feature category is the object and instead of storing the large number of overall features in a dictionary (the

aforementioned *one-hot encoding* method), we hash each feature category. After applying the hash function to a feature category we obtain a value, say 'V'. This V modulo the total number of buckets at our disposal is the index where we can now find the feature and the resultant index of the vector is incremented by 1. The final vector after hashing all feature categories is a numerical representation of the dataset as we desire.

Logistic regression (method and implementation in MLib): Regression analysis is a widely used method in inferential statistics. Simply put, regression aims to find the *conditional expectation (average value)* of a dependent variable given a set of independent ones. In the context of our dataset the independent variables are the set of features numericalized using feature hashing and the dependent variable is the probability of the desired response-i.e. a click. For logistic regression, we assume that this probability for any given data point is the weighted sum of the numeric features for that data point. If the weight vector is represented by 'w' and the feature vector by 'x', then the logistic regression model will output a probability given by the logistic function i.e.

$$f(z)=1/(1+e^{-z})$$

where $z=w^{T}x$. The beauty of using this logistic function is no matter the weighted product of the feature the logistic function will always "squash" it into a corresponding weighted value between 0 and 1 which is a useful as an estimate of the probability of a click on that data point (ad).

2. Execution Environment

We implement the above steps using *Pyspark,* a Spark API that exposes the Spark programming model to Python. We then run our experiments (each outlined individually) on a high performance cluster with a large number of high-memory nodes and cores at our disposal.

3. Experimental Results

Our first order of action is to determine an optimum configuration for Spark. Since Spark has a number of tunable parameters which impact performance, we wish to *regularize* these experimentally before carrying out our strong scalability analysis on an optimum set of parameters.

Spark Controls: All Spark application runs consist of one driver process and multiple executor processes which run in parallel across multiple nodes on the cluster. The parameters which can be defined at the time of execution of a particular Spark job and which can affect performance are:

- Number of executors
- Number of cores allocated to each executor (and, by extension, the total number of nodes.)
- The amount of available RAM(memory) allocated to each executor.
- Whether or not the driver process runs independently on its own node.
- Number of partitions of dataset.
- Choice of serializer

4. Analysis

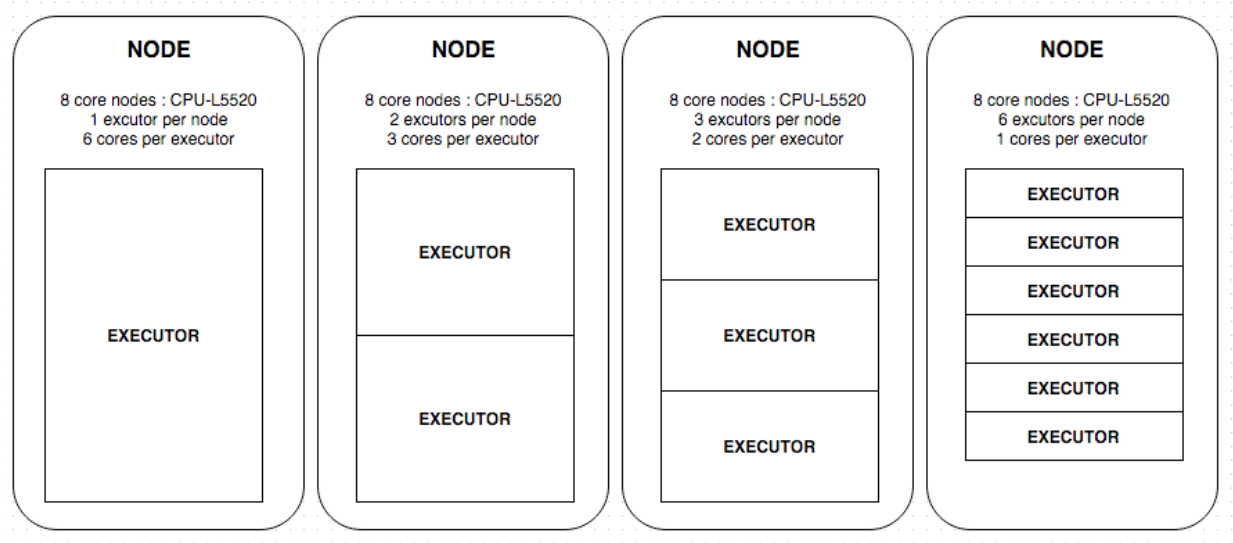
4.1 Number of executors / Number of cores per executor

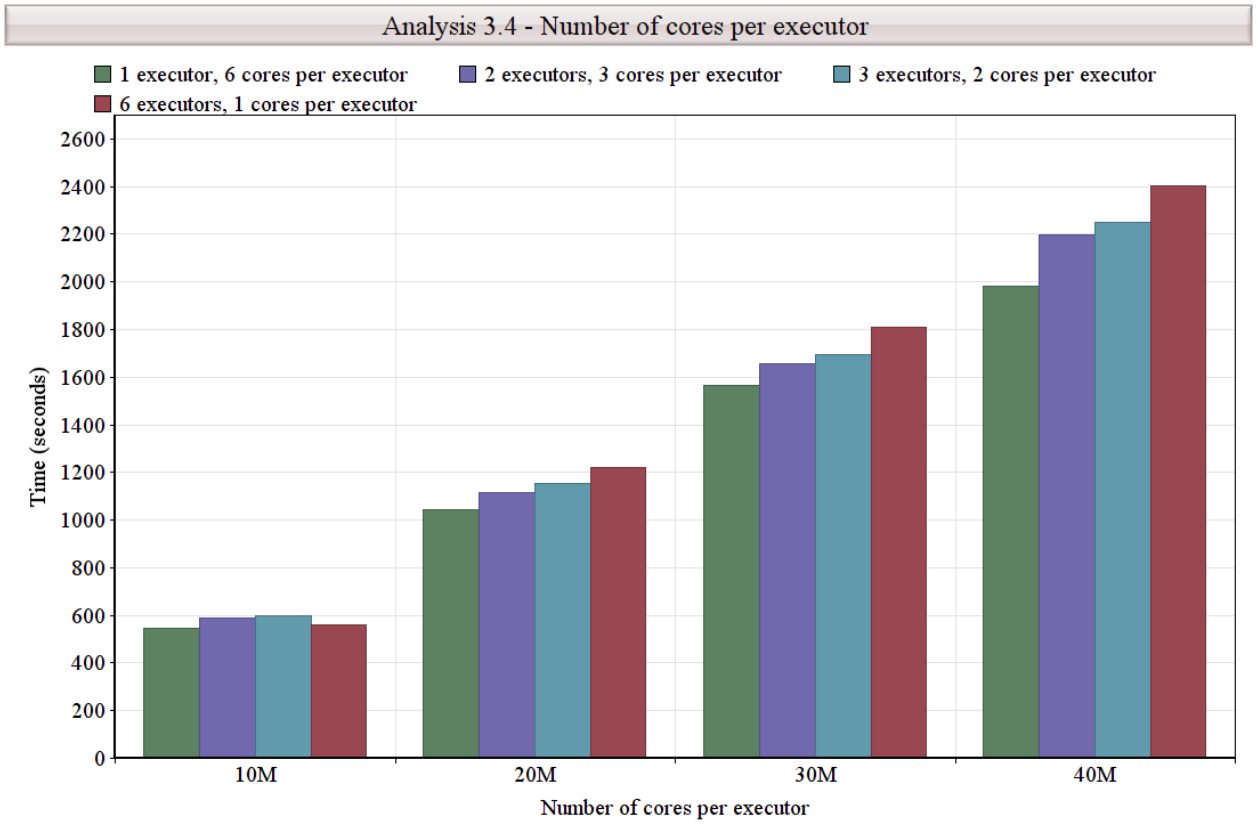
Description: Here, we examine the effect of number of executors and number of cores per executor has on performance.

For this experiment we use a single node with 8 cores. The master or driver process and the executors run on the same node. 6 of the 8 cores of the node are used by the executors while 1 core is used by the master process. We then plot a graph of the speedup obtained.

Configuration used:

8 core nodes: CPU-L5520
1 node used
6 cores used for executors
Master on one of the executor nodes





Conclusion:

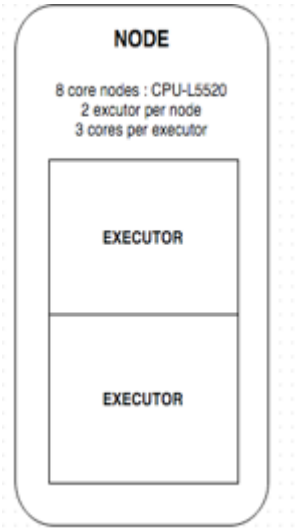
We can see that although the number of cores used is same throughout i.e. 6, the performance is different. The best performance is obtained with 1 executor with 6 cores which shows that latency due to inter-executor communication is significant and must be accounted for. With the increase in the number of executors, the communication between these executors increases which increases latency. This also implies that the number of cores per executor must be selected according to the cores available per node so as to minimize the latency due to inter-executor communication.

4.2 Memory

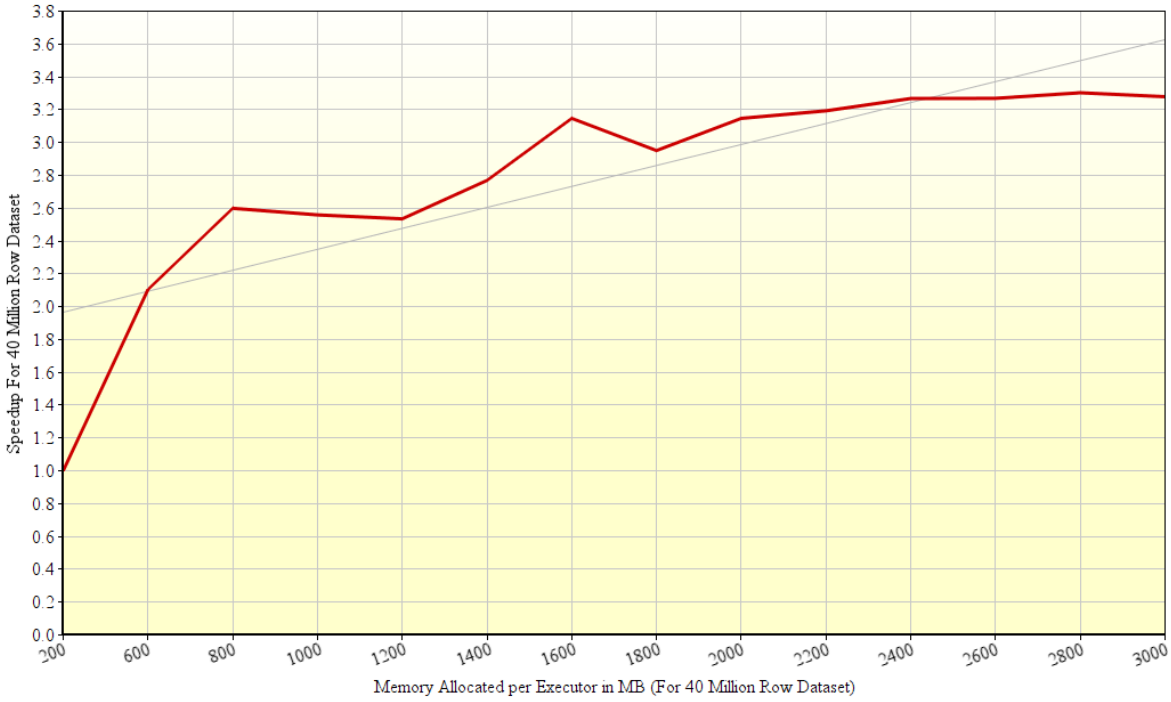
Description: Here, keeping a performant set of parameters from the previous experiment constant, we vary only the memory allocated to each executor and plot the results to assess the effect of memory on performance. It is observed that all things being constant, performance improves with increasing memory and then plateaus at a certain point and stays constant until the nodes run out of memory to allocate altogether.

Configuration used:

8 core nodes – CPU-L5520
2 executors per node
3 cores per executor
40 Million Row Dataset



Speedup Vs Memory Allocated Per Executor(Apache Spark,2 executor,3 cores/per executor using 8 core nodes)



Conclusion:

Clearly, performance improves significantly, as we allocate more memory to the executor processes. This is natural since more memory allocated means less expensive cache lookups/disk i/o. However, at a certain point the performance peaks and does not improve even if more memory is allocated indicating that there is sufficient memory available to store the entire dataset which means more memory allocated makes no difference.

4.3 Location of Master

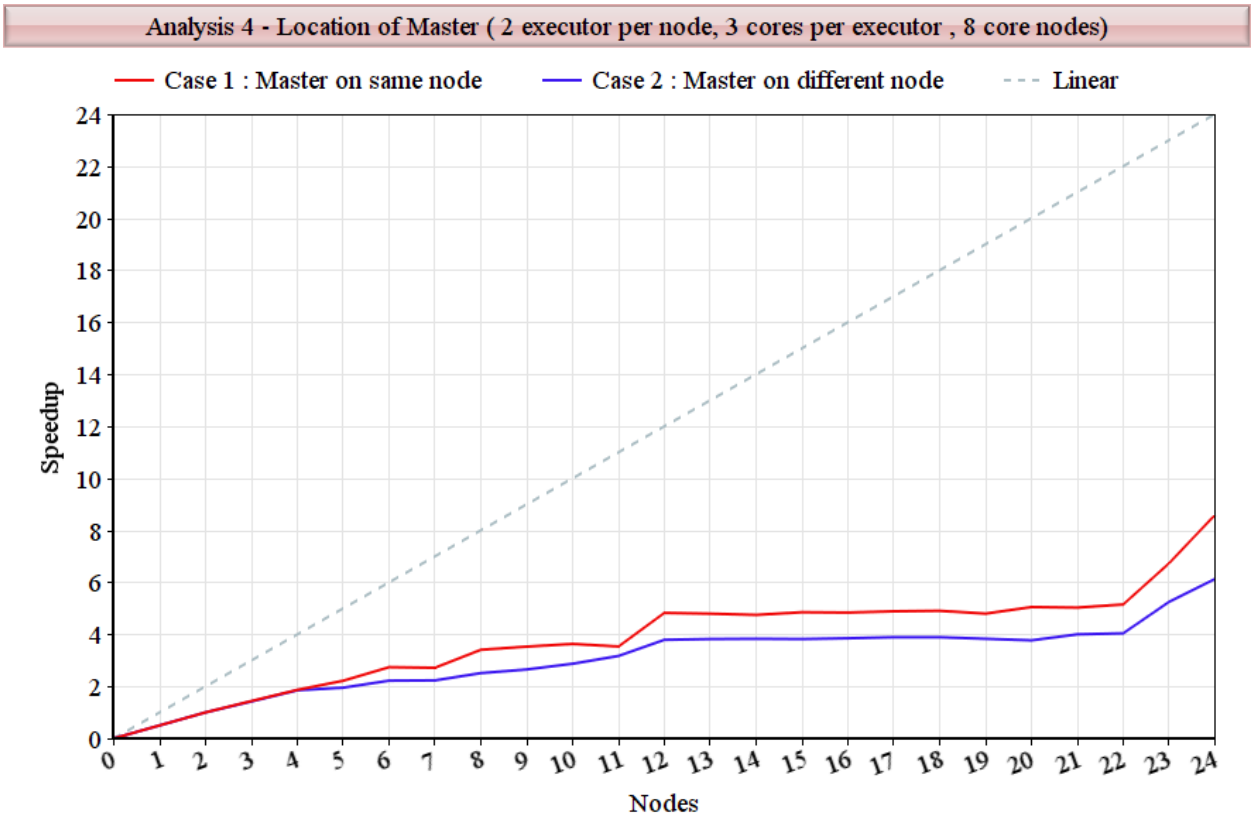
Description:

We mentioned earlier that Spark applications consist of two kinds of processes, an *application master/driver* and *executors/slaves*. The master process requires one core and can run on any of the nodes. In this experiment we find the optimal node where the master can be run to get best performance. Accordingly, we have the following cases.

Case 1: Master on same nodes as executors
Case 2: Master on separate independent node

Configuration used:

8 core nodes: CPU-L5520
2 executors per node
3 cores per executor
Data used: 30M rows (about 4.5GB)



From the above graph we can see that there does exist slight difference in performances of the two cases. The reason for this difference is that running the master on a separate node increases the inter-nodal communication. The above results prove the following:

I. Inter-nodal communication is expensive and should be minimized
II. The cost of inter-nodal communication is much higher than the cost of running the master
III. The optimal location of the master is on one of the executor nodes as it is very light weight and reduces the inter-nodal communication latency

4.4 Strong Scaling

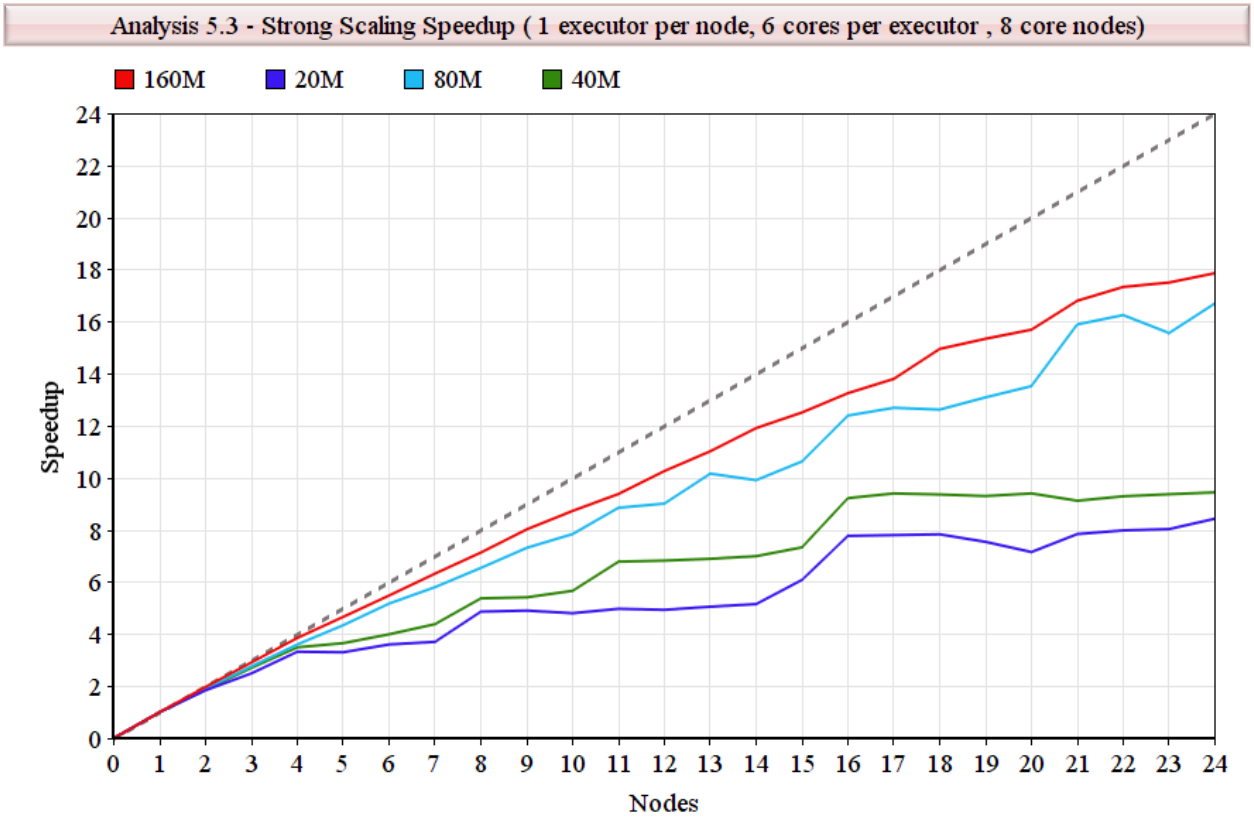
From the above analysis, we see that the following configuration is best suited for our problem:

I. 1 executor per node
II. 6 cores per executor
III. Master on one of the executor nodes

Thus, we use the above configuration to check for strong scalability.

Configuration used:

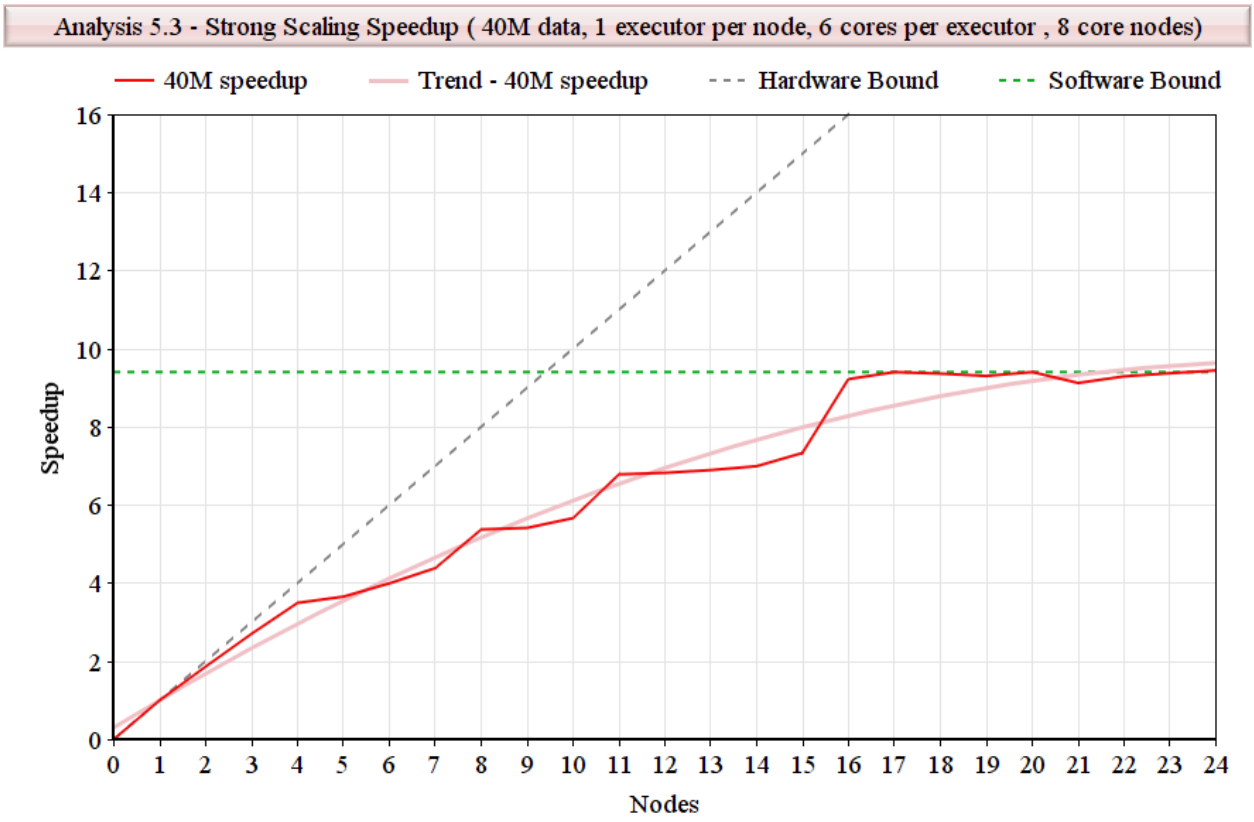
8 core nodes: CPU-L5520
1 executor per node
6 cores per executor
Master on one of the executor nodes
1 GB of memory per executor



We can see from the above graph that the algorithm provides better scalability for larger dataset. The overhead of inter nodal communication is significant for data less than 80 million rows i.e. about 12.5 GB. This overhead remains constant irrespective of the dataset size, as expected. Thus, as the dataset size is increased, the impact of the overhead on the performance becomes less significant and the speedup shifts towards linear speedup.

From the plot for 160 million rows i.e. about 24 GB, it can be seen that linear speedup is obtained approximately until 18 nodes. After which the speedup tends to degrade as expected.

The following graph shows the speedup for 40 million rows of data, the hardware bound and the software bound.



The ***hardware bound*** reflects the limitation imposed by the hardware, and is given by the number ***n*** of available processors. This bound can be achieved only if all ***n*** processors can be kept busy all of the time. The ***software bound*** reflects the limitation imposed by the software, and is derived by noting that, no matter how many processors are available to a system, the execution time must be at least as long as the length of a longest path. Hence, the speedup is at most the ratio of the total service demand to the length of a longest path.

The ***intersection point*** of the hardware and software bounds on speedup is significant: when additional processors are allocated, it is certain that there is not enough parallelism in the software ***to*** keep all ***of*** the processors busy all ***of*** the time. [1]

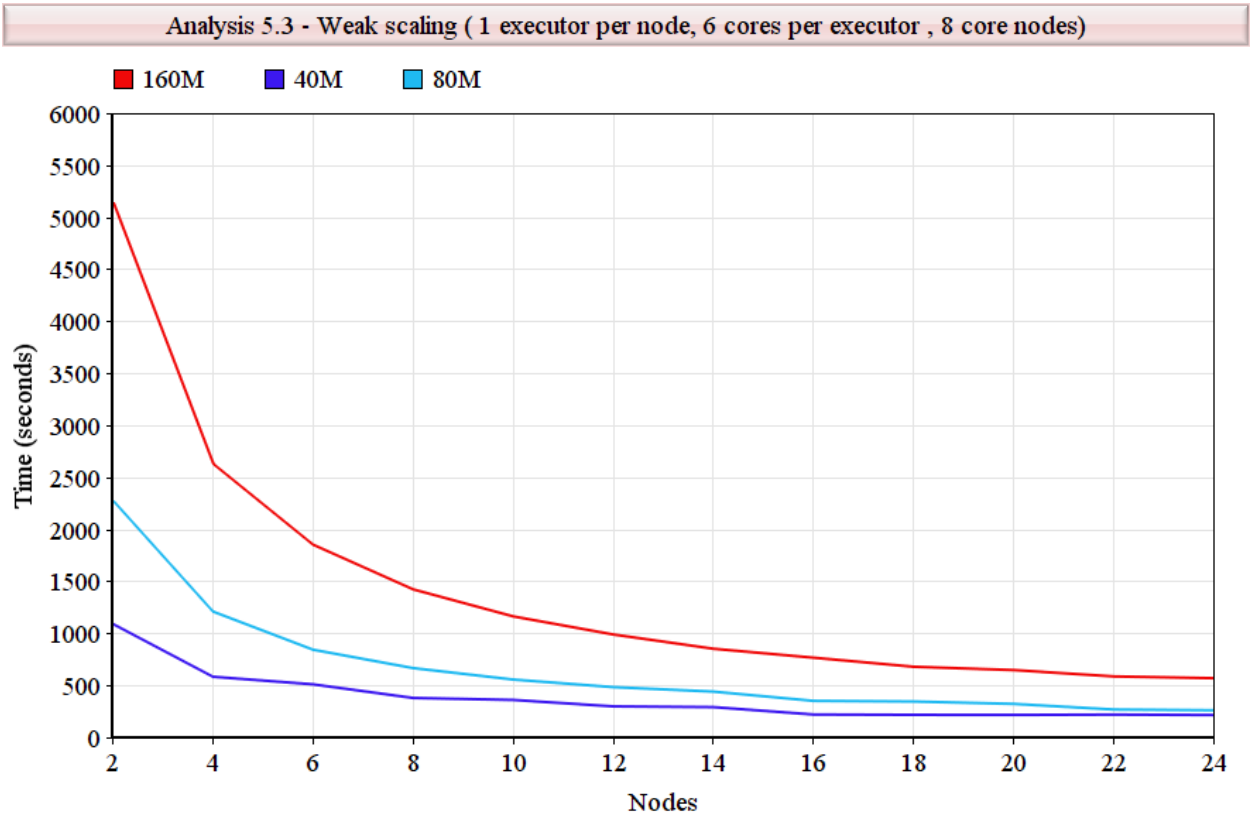
Conclusions:

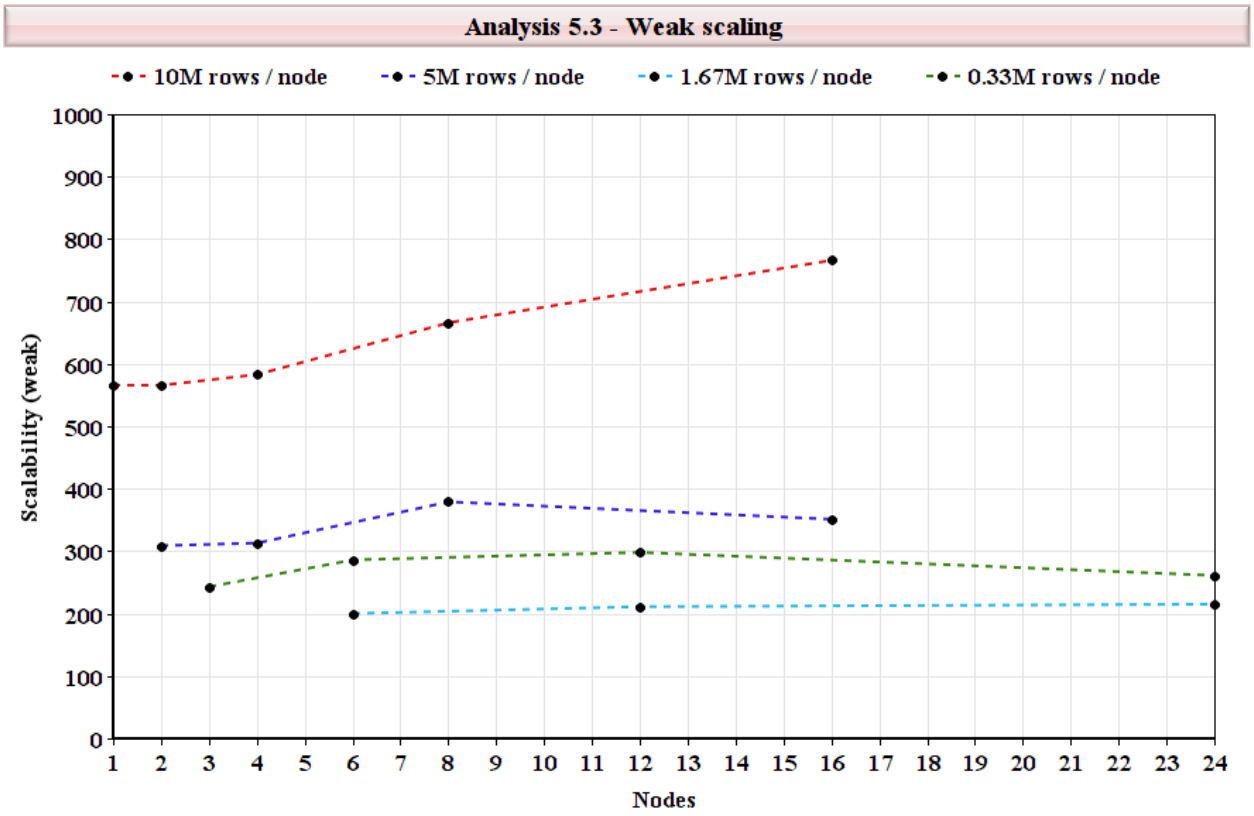
- Clearly, we can infer that speedup improves as number of cores allocated to the job increase. However, as the allocated nodes become more and more, communication between nodes which is network/bandwidth bound begins to take its toll and rate of increase of speedup begins to level off.
- The intersection point of hardware and software bound is 9.3 nodes. This implies that after 9-10 nodes, there is not much parallelism the software can provide.

4.5 Weak Scaling

Configuration used:

8 core nodes: CPU-L5520
1 executor per node
6 cores per executor
Master on one of the executor nodes
1 GB of memory per executor

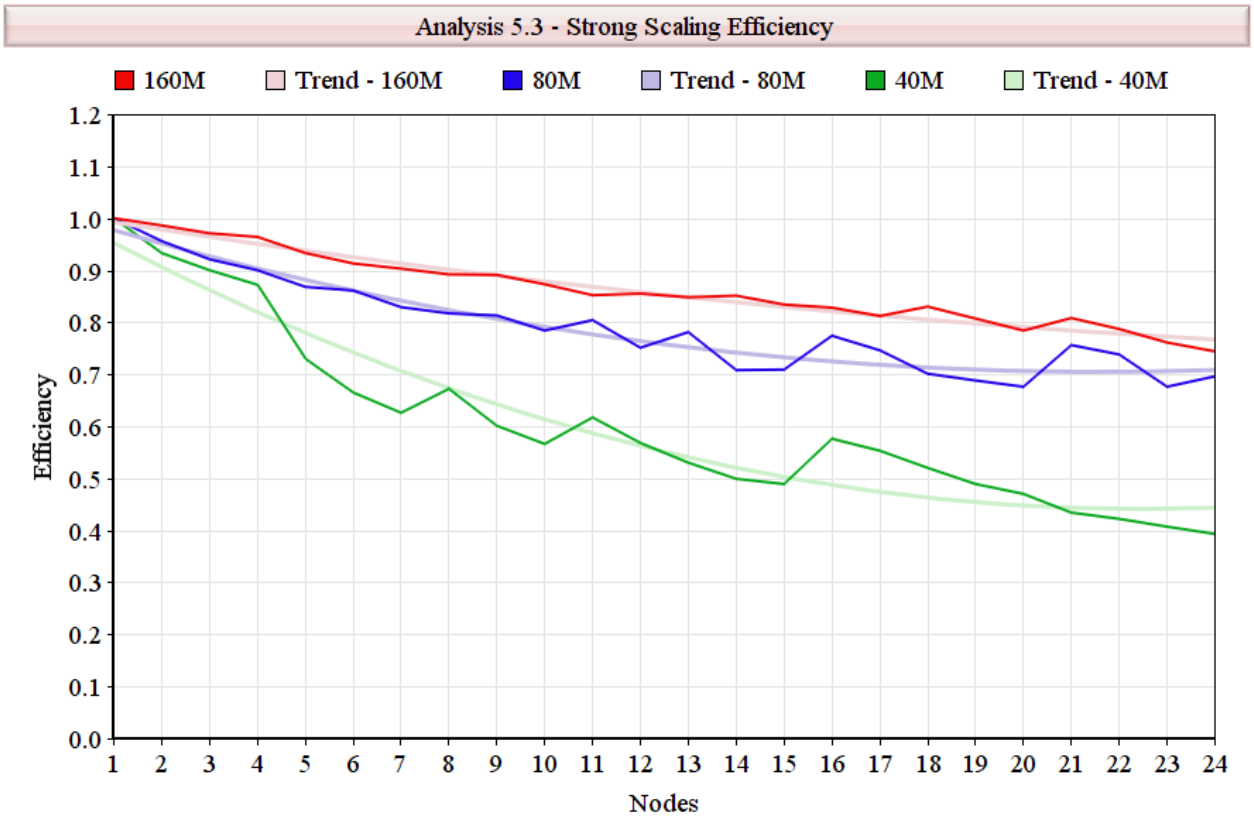




The above graph shows that the algorithm does provide weak scalability and it improves for higher number of nodes.

4.5 Efficiency

To complete our assessment of our solution, we analyze the efficiency of our strong scaling speedup. Since speedup Sn is the ratio T1/Tn and efficiency is the ratio of Sn/n, it is clear that although using more processors can give us a large advantage in terms of speedup, it degrades efficiency (primarily due to communication and network bounds). It is important to find the sweet spot where we have a good speedup and are still not using too many nodes but rather each node is utilized as efficiently as possible. To do so, we plot a graph of the number of nodes being used vs the efficiency.



Conclusions:

From the plot for 40 million row dataset, we see that our solution exhibits the most desirable tradeoff between speedup and efficiency at about 4 nodes for an efficiency of roughly 0.8. In other words, the solution is strongly scalable to about 4 nodes. Later spikes in the plot are also points where the tradeoff is not too bad but in general we can say that the strong scalability extends up to 4 nodes.

Similarly, the best tradeoff for 160 million row dataset can be seen at 18 nodes, after which there is significant degradation of efficiency.

4.7 Number of partitions

8 core nodes: CPU-L5520
1 executor per node
6 cores per executor
Master on one of the executor nodes
1 GB od memory per executor
40M rows dataset

Partitions	Time
186	577
279	623
372	672
465	696
558	817
659	929

Spark partitions the data depending on its size. The minimum default partitions that Spark provides for 40 million row data is 186. We can see that the algorithm performs best for the default number of partitions that Spark provides.

4.7 Serializers

Serialization in Spark refers to the process of converting objects into bytes for distributed computations. As explicitly stated in the Spark documentation, the speed at which this is done plays a key role in how well any Spark application performs serialization and de-serialization of objects is constantly being done across the distributed network.

There are two popular serializers used in PySpark applications:

- **Pickle Serializer:** The default built in PySpark serializer
- **Marshal Serializer:** Faster than Pickle but has the drawback of being unable to serialize all Python objects and is not always usable.

We use the Pickle serializer for our solution since the Marshal is unable to serialize some of the objects in our code.

5. Final Conclusions

In summary, we have presented a viable solution to the Click Through Rate Prediction problem using Apache Spark using the technique of logistic regression. We make the final experimental conclusions below:

- The solution exhibits strong scalability up to about 4 nodes for a 40 million row datasets and up to 18 nodes for 160 million row dataset, but efficiency is reduced thereafter due to the fact that there is not enough parallelism in the solution to keep all the processors sufficiently busy after about 4 nodes.
- The solution exhibits good weak scalability up to as much as 160Million rows of data indicating that if the data increases along with the number of processors, then the processors are kept sufficiently busy.
- The solution is highly tunable using the standard spark parameters of executor count, cores/executor, memory/executor, location of application master and number of partitions as outlined below:
    - **Number of Executors:** We observe that the cost of inter-executor is expensive compared to the gain that multiple executor processes can provide. Therefore, we restrict ourselves to 1 executor process per node.
    - **Cores/Executor:** Although we are using 8 core nodes, we restrict ourselves to about 6 cores per executor which gives us best performance. This makes perfect sense since the node requires some processing power to run the OS, various Hadoop daemons, byte buffers etc. and allocating all cores to executor would ultimately be detrimental.
    - **Memory/Executor:** We observe that allocating more memory per executor improves more performance up to the point that the entire dataset can be held in memory. After this performance does not change no matter how much more memory is allocated. The optimum amount of memory per executor seems to be about 2.4G for 4 million row dataset.
    - **Location of Application Master:** We observe that running the application master process on one of the same nodes as an executor process gives much better performance. This is because the overhead of running the master is very low and comparable to the communication cost between the master and the worker processes.
    - **Number of Partitions:** We observe empirically that the default number of partitions that Spark allocates to the solution (186 for 40 million rows dataset, 745 for 160 million rows dataset) gives optimum performance and that increasing this number marginally increases running time(see section 4.6)

6.References

- "How To Tune Your Apache Spark Jobs" –Cloudera Engineering (http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/)
- A Course In Scalable Machine Learning-Ameet Talwalkar,UCLA for EdX
- "Speedup Versus Efficiency in Parallel Systems" –Eager,Zahorjan,Lozowska-1989