Early Performance Evaluation of Supervised Graph Anomaly Detection Problem Implemented in Apache Spark

Artem Mazeev[1], Alexander Semenov[1], Dmitry Doropheev[2], Timur Yusubaliev[3]

[1] JSC NICEVT, Moscow, Russia
[2] Moscow Institute of Physics and Technology, Moscow, Russia
[3] Quality Software Solutions ltd, Moscow, Russia
{a.mazeev,semenov}@nicevt.ru, dmitry@dorofeev.su, ytr@kpr-it.com

Abstract. Apache Spark is one of the most popular Big Data frameworks. Performance evaluation of Big Data frameworks is a topic of interest due to the increasing number and importance of data analytics applications within the context of HPC and Big Data convergence. In the paper we present early performance evaluation of a typical supervised graph anomaly detection problem implemented using GraphX and MLlib libraries in Apache Spark on a cluster.

Keywords: machine learning · MLlib · Spark · graph processing · supervised anomaly detection · performance evaluation

1 Introduction

In recent years, data intensive applications have become widespread and appeared in many science and engineering areas (biology, bioinformatics, medicine, cosmology, finance, social network analysis, cryptanalysis etc.). They are characterized by a large amount of data, irregular workloads, unbalanced computations and low sustained performance of computing systems. Development of new algorithmic approaches and programming technologies are urgently needed to boost efficiency of HPC systems for similar applications thus enabling advancing of HPC and Big Data convergence [13].

Spark [15] is a framework which optimizes programming and execution models of MapReduce [6] by introducing resilient distributed dataset (RDD) abstraction. Users can choose between the cost of storing an RDD, the speed of accessing it, the probability of losing part of it, and the cost of recomputing it. Apache Spark [1] is a popular open-source implementation of Spark. It supports a rich set of high-level tools including MLlib for machine learning and GraphX for graph processing.

Anomaly detection in graphs occurs in many application areas, for example, in the analysis of financial markets, in spam filtering, as well as in detection of cyber attacks.

In the paper we evaluate performance of a typical supervised anomaly detection problem implemented using GraphX [10] and MLlib libraries in Apache

Spark. We use synthetic graph generator for performance evaluation. In our approach we calculate features based on community extraction. Then we fit model using supervised machine learning techniques. One can apply this model to new objects, thus performing anomaly detection to new data sets.

2 Related Works

Performance evaluation of Big Data frameworks is drawing more attention due to increasing number and importance of data analytics applications within the context of HPC and Big Data convergence. There exist many papers that present performance evaluation of Spark, e.g. [3, 11, 7]. Chaimov et al. [5] ported, tuned and evaluated Spark on Cray XC systems developed in production at a large supercomputing center. They have reached scalability up to 10,000 cores of the Cray XC system.

Performance evaluation of the initial version of the GraphX library is considered in [10]. The paper presents strong scaling of the PageRank algorithm.

Some papers consider performance evaluation of machine learning applications implemented in Spark. In [14] the MLlib library is not used. In [12, 8] there is no scalability results of the MLlib library.

In this paper, we present performance and scalability results of the typical machine learning application implemented using the latest version (2.1.1, May 2017) of standard GraphX and MLlib libraries in Apache Spark on a cluster equipped with Angara and 1 Gbit/s Ethernet interconnect.

3 Supervised Graph Anomaly Detection Problem

We consider an anomaly detection problem for synthetic graphs to evaluate performance and scalability of graph processing and the machine learning techiques in Apache Spark.

We consider a random uniform weighted directed graph $G = (V, E)$ [9], $|V| = N, |E| = M$. Each edge connects two random vertices of the graph G so that there is no self-loops. Each edge has attributes. The list of attributes includes integer edge weight (a random value in $[0, 10^5)$) and another integer values; *max_degree* is a maximal degree of each vertex.

The edge is considered as anomaly if its weight is greater than a given threshold. We consider $ANOMALY\_EDGES\_FRACTION * M$ random edges as anomalous by adding to their weights random values in $[0, 10^9)$. We consider $ANOMALY\_EDGES\_FRACTION = 0.05$. Other edges are normal.

The edge weight is an opaque anomaly feature, it allows to build a training set and a test data set for our synthetic supervised problem. In the problem we have to fit model using supervised machine learning techniques. It is needed to classify whether the edge is anomalous or normal.

Eventually, the computation process consists of two stages: feature calculation and supervised (machine) learning. First, it is necessary to calculate the

features for each edge. Feature calculation includes community extraction procedure.

We define a community around vertex u as a set of vertices $v : dist(u, v) \leq R$, where $dist$ – the shortest distance between u and v. Edges are considered as undirected during the community extraction. We extract two communities around both vertices of each edge of the training set. We consider communities with $R = \{1, 2\}$.

Feature calculation is heavily based on the extracted community. The total number of features is 52. The set of features for an edge includes:

- degrees of the edge vertices, weight of the edge, other parameters from the edge attributes,
- minimum, maximum and average for degree, indegree and outdegree of the community,
- number of edges and vertices in the community,
- sum of weights of all edges in the community.

After the feature calculation stage, a machine learning stage is performed.

We believe that our synthetic supervised graph anomaly detection problem is typical because during the data mining research it is needed to calculate graph features many times, and then to fit a model. Especially, it is required in the beginning of a research while trying to choose a suitable set of features and to select an appropriate machine learning technique.

3.1 Time Complexity

We consider time complexity of the feature calculation stage. We calculate features for each edge of the graph. Each edge has two incident vertices. The complexity of feature calculation for one vertex is the number of vertices in the community with $R = 2$ around this vertex, i.e. max_degree^2 operations in the worst case. So, time complexity for one vertex is $O(1)$.

The number of vertices in the graph for which we calculate features is $O(\min(N, M))$ because if $M < N$, then we calculate features only for the relevant vertices. So, theoretical time complexity of the feature calculation stage is $O(\min(N, M))$. Of course, we can run it by parallel and then with p processes time complexity will be $O(\min(N, M)/p)$.

4 Implementation

We implement the algorithm and the synthetic graph generator with using of Scala language, the GraphX system and the MLlib library on top of Apache Spark, version 2.1.1.

In our implementation each edge has a string which stores integer attributes delimited by a ',' symbol. After generating vertices and edges we create a graph using `Graph` method from the GraphX library.

In the work we use Sparks RDD program interface. Resilient distributed dataset (RDD) is the main abstraction in Spark, which represents a read-only collection of objects partitioned across a set of machines. Users can explicitly cache RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations [15]. The latest Spark program interface DataFrame [4] seems to be more efficient, we plan to use it in the future work.

The feature calculation stage works as follows. We use `degrees, inDegrees, outDegrees` methods from GraphX library, in the current implementation version we calculate other features by using `join` and `map` RDD operations. We calculate features for all edges of the graph because it costs almost the same time as calculating features only for edges which are used in the machine learning stage.

Our implementation uses simple operations (for example, `map, filter`) that can be performed independently for each element from dataset. Also, the implementation uses expensive operations: `distinct, subtract, join, and groupBy`. These operations are potentially expensive because they include a shuffle. The shuffle is a Spark mechanism for re-distributing data so that it is grouped differently across partitions of data. This typically involves copying data across the cluster, making the shuffle a complex and costly operation. In our implementation we often use `cache` method to store data in the main memory.

In the machine learning stage we use `LogisticRegressionWithLBFGS, SVMWithSGD, RandomForest` methods from the MLlib library. Currently, there is no feature selection stage in our solution, but we plan to add it in future work.

4.1 Time Complexity in Apache Spark

The sort algorithm used for the shuffle operation is not specified in the Apache Spark documentation. We assume that optimal complexity of parallel sort algorithms is $O(n * log(n)/p)$, where p is a number of processes which can not be more than n. We use it as a rough bound of the time complexity for the sort algorithm inside the shuffle operation. Therefore, the complexity of shuffle is $O(n * log(n)/p)$, where n is the number of elements in RDD or DataFrame, p is the number of processes.

The programs hot spot is the calculation of communities with $R = 2$. This operation contains a join of the two RDDs, the first RDD consists of pairs (a neighbour of the vertex and the vertex) and the second consists of the reversed pairs (the vertex and a neighbour of the vertex), i.e. after the join operation we have vertices with $dist = 2$ for any vertex in the graph. Both RDDs consist of $O(max\_degree * \min(N, M)) = O(\min(N, M))$ elements. So, complexity of the shuffle operation is $O(\min(N, M) * log(\min(N, M))/p)$.

Complexity of the local calculations after the shuffle operation is $O(\min(N, M)/p)$, p — the number of parallel processes because for each element of the first RDD there exist max_degree elements of the second RDD in the worst case (we build an extension of the community with $R = 1$ to the community with the $R = 2$).

Our theoretical and profiling analysis shows that the time of the remaining programs operations is insignificant. Eventually, our time complexity evaluation

of the feature calculation stage implemented in Apache Spark is $O(\min(N, M) * log(\min(N, M))/p)$.

5 Performance Evaluation

Table 1. System configuration of the Angara-K1 cluster

Cluster	Angara-K1
Chassis	SuperServer 5017GR-TF
Processor	E5-2660 (8 cores, 2.2 GHz)
Memory	DDR3 64 GB
Number of nodes	36
Interconnect	Angara 4D-torus $3 \times 3 \times 2 \times 2$ 1 Gbit/s Ethernet
Operating system	SLES 11 SP4
Spark	Apache Spark 2.1.1
Scala Compiler	sbt 0.13.13

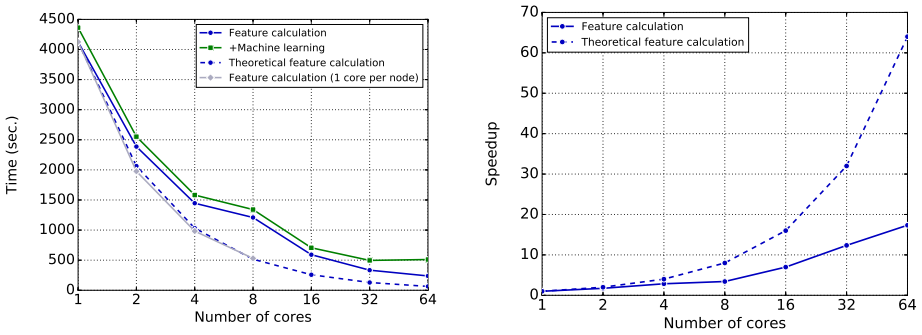
All presented results has been obtained on the Angara-K1 cluster. It has 36 nodes, but in the paper we use only 8 nodes. Table 1 provides information about the architecture and a software overview of the Angara-K1 partition. All Angara-K1 nodes are connected to each other by the Angara and 1 Gbit/s Ethernet interconnects. High-speed Angara interconnect is developed in NICEVT, performance evaluation of the Angara-K1 cluster with Angara interconnect on scientific workloads is presented in [2].

We use the following default graph parameters: $N = 2^{19}$, $M = 2^{22}$. We suggest that the graph size is large enough for scalability evaluation, but performance evaluation consumes reasonable time.

In the figures the dashed line shows theoretical evaluation. We plot this line as follows. We take left point in the corresponding obtained results line and multiply this value on the ratio of corresponding asymptotic values, for example, for weak scaling we get time value X for the single core point, then for the 64 cores point we get theoretical time $X*((64*N)*log(64*N)/64)/(N*log(N)/1) = X * log(64 * N)/log(N)$, where N is the number of vertices.

Strong scaling on the default graph is shown in Fig. 1a and Fig. 1b. The speedup of feature calculation from 1 to 8 cores on the single node is 3.41 but speedup from 1 to 8 nodes using 8 cores per node is 5.08. The reason of poor single node scalability is that feature calculation is a memory bound Spark application. Fig. 2 confirms this by showing strong scaling on the same problem with different cores per node number.

From the results in Fig. 1a, we can see that on the single node the feature calculation stage requires more time than the machine learning stage, but feature calculation scales fairly good.



(a) **(b)**

Fig. 1. Strong scaling on the default graph ($N = 2^{19}, M = 2^{22}$). 8 cores per Angara-K1 cluster node

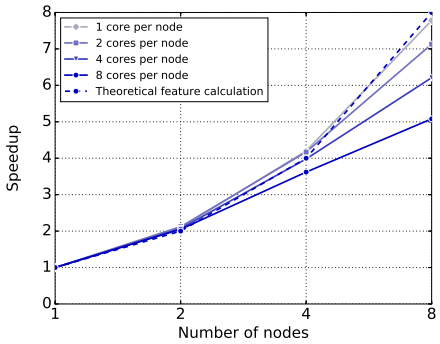


Fig. 2. Strong scaling of the feature calculation stage on the default graph ($N = 2^{19}, M = 2^{22}$) with different values of cores per node

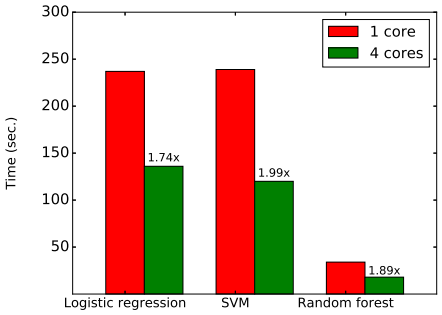


Fig. 3. Strong scaling of the MLlib methods on the default graph ($N = 2^{19}, M = 2^{22}$)

Strong scaling of the different machine learning algorithms is shown in Fig. 3. We consider `LogisticRegressionWithLBFGS`, `SVMWithSGD` and `RandomForest` methods of the MLlib library. These methods scale only to 4 cores of the cluster single node.

Weak scaling is shown in Fig. 4. For p processes (cores) we use graph with $N = 2^{14} * p$ vertices and $M = 2^{17} * p$ edges. Weak scaling results is poor. Among the possible reasons there is a single one that Spark configuration is not optimal. Future tuning can address the problem.

Fig. 5 shows execution time of the program on graphs with different size. We use 8 nodes of the Angara-K1 cluster. The feature calculation results are near to the optimal theoretical line.

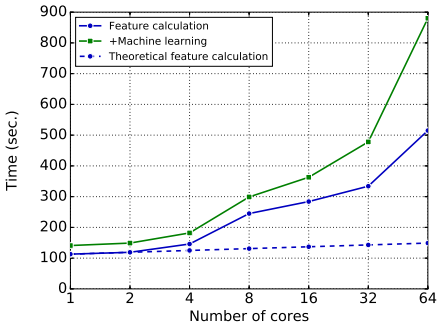


Fig. 4. Weak scaling on the graphs with $N = 2^{14} * P$ vertices and $M = 2^{17} * p$ edges, where p – number of cores. 8 cores per Angara-K1 cluster node

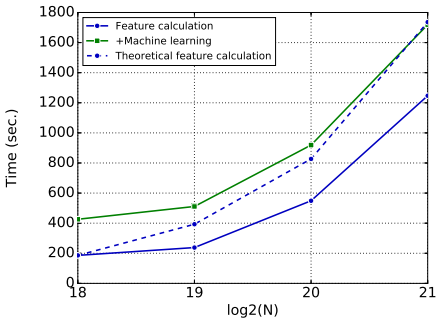


Fig. 5. Program execution on different graphs. 8 cores per each of the 8 Angara-K1 cluster nodes. $M = N * 8$

6 Conclusion

The paper presents performance evaluation of a typical supervised graph anomaly detection problem detection problem implemented using GraphX and MLlib in Apache Spark on the commodity cluster equipped with Angara and 1 Gbit/s Ethernet interconnects.

The considered anomaly detection problem consists of calculation of features and supervised machine learning. The feature calculation requires more time than machine learning stage on the single cluster node, but it scales good. The machine learning stage implemented using the MLlib library does not scale beyond the single cluster node.

Our theoretical analysis shows that the performance results of strong scaling and scaling on graphs with different size on a fixed cluster configuration is relatively good. It seems that Apache Spark is a memory bound application and many cores per cluster node running leads to lower efficiency. This fact and poor performance results of weak scaling of the problem is the subject of future research.

Acknowledgments. Research is being conducted with the finance support of the Ministry of Education and Science of the Russian Federation Unique ID for Applied Scientific Research (project) RFMEFI57816X0218. The data presented, the statements made, and the views expressed are solely the responsibility of the authors.

References

1. Apach Spark Homepage, http://spark.apache.org, http://spark.apache.org/

2. Agarkov, A., Ismagilov, T., Makagon, D., Semenov, A., Simonov, A.: Performance evaluation of the Angara interconnect. In: Proceedings of the International Conference Russian Supercomputing Days. pp. 626–639 (2016)
3. Armbrust, M., Das, T., Davidson, A., Ghodsi, A., Or, A., Rosen, J., Stoica, I., Wendell, P., Xin, R., Zaharia, M.: Scaling spark in the real world: performance and usability. Proceedings of the VLDB Endowment 8(12), 1840–1843 (2015)
4. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., et al.: Spark sql: Relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 1383–1394. ACM (2015)
5. Chaimov, N., Malony, A., Canon, S., Iancu, C., Ibrahim, K.Z., Srinivasan, J.: Scaling Spark on HPC systems pp. 97–110 (2016)
6. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation - Volume 6. OSDI'04, USENIX Association, Berkeley, CA, USA (2004)
7. Dünner, C., Parnell, T., Atasu, K., Sifalakis, M., Pozidis, H.: High-performance distributed machine learning using Apache Spark. arXiv preprint arXiv:1612.01437 (2016), https://arxiv.org/pdf/1612.01437.pdf
8. Dünner, C., Parnell, T.P., Atasu, K., Sifalakis, M., Pozidis, H.: High-performance distributed machine learning using Apache Spark. CoRR abs/1612.01437 (2016), http://arxiv.org/abs/1612.01437
9. Erdős, P., Rényi, A.: On random graphs. Publicationes Mathematicae Debrecen 6, 290–297 (1959)
10. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: GraphX: Graph processing in a distributed dataflow framework. OSDI 14, 599–613 (2014)
11. Hong, S., Kim, S., Jang, J., Choi, C.h., Jung, I.s., Na, J., Cho, W.S., Chi, S.y.: Performance evaluation of apache spark according to the number of nodes using principal component analysis. In: Proceedings of the 2015 International Conference on Big Data Applications and Services. pp. 98–103. BigDAS '15, ACM, New York, NY, USA (2015)
12. Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., Xin, D., Xin, R., Franklin, M.J., Zadeh, R., Zaharia, M., Talwalkar, A.: MLlib: Machine learning in Apache Spark. Journal of Machine Learning Research 17(34), 1–7 (2016)
13. Reed, D., Dongarra, J.: Exascale computing and big data: The next frontier. Communications of the ACM 57(7), 56–68 (2014)
14. Wei, J., Kim, J.K., Gibson, G.A.: Benchmarking Apache Spark with machine learning applications (2016), http://www.pdl.cmu.edu/PDL-FTP/BigLearning/CMU-PDL-16-107.pdf
15. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. HotCloud 10, 1–7 (2010)