An Analysis of the Impact of Hash Codes in Apache Hadoop and Spark

Johannes Vamos, Tong Wu, Xin Yang
Department of Electrical and Computer Engineering
Rutgers University
{*johannes.vamos,tong.wu96, xin.yang*}*@rutgers.edu*

ABSTRACT

Hash functions are an integral part of MapReduce software, both in Apache Hadoop and Spark. If the hash function performs badly, the load in the reduce part will not be balanced and access times spike. To investigate this problem closer, we run a WordCount program with numerous different hash functions on Amazon AWS. In particular, we will leverage the Amazon Elastic MapReduce framework. The paper investigates on general purpose, cryptographic, checksum, and special hash functions. With the analysis, we present the corresponding runtime results.

1. INTRODUCTION

MapReduce is a Framework to extract information from large datasets efficiently. It has to major components: i) a Mapper, which reads and transforms data and creates key-value pairs, and ii) a Reducer, which combines multiple Mapper outputs. Each Mapper and Reducer can run on an individual machine. The most expensive operation in this setup is transferring data between machines. Thus, the necessary data exchange should be kept minimal. Additionally, for the Reducer to work correctly, it needs all data which correspond to the same key. To address those two problems hash functions are used. They ensure that each key has the same, shorter hash value and these hash values are assigned to Reducer. The Reducer then works on their own problem and generate the result for a certain key. Hash functions have two core contributions: i) ensure that each key produces the same hash value, and ii) create a uniformly distributed. The latter is necessary to evenly distribute the workload.

Both Apache Hadoop [1] and Apache Spark [2] are two commonly used MapReduce frameworks. If the hash functions are slow or distribute data badly, the execution time will potentially increase. A slow hash function will add additional computation time at each node and thus increase the overall computation time. If a hash function does not distribute the data accordingly, multiple keys might fall into the same hash value and thus increase the load on a single Reducer. The parallelism is reduced and the execution time is increased.

In this paper, we want to give an analysis of common hash functions if they are used in Apache Hadoop or Spark. For this purpose, we selected 14 functions which generate a hashed value. We use each function in a WordCount example in both Apache Hadoop and Spark. Additionally, we use these hash functions in the PageRank [3] algorithm on Apache Spark. With the resulting execution times, we show how hash functions impact the performance of MapReduce problems and how they can also influence the performance of machine learning algorithms with Apache Spark.

This paper is structured as follows: Section 2 presents previous work and main contributions. Section 3 presents the implementation details behind the analysis. In Section 4 we present the test environment. Section 5 provides the results of the analysis. Finally, Section 6 concludes this paper.

2. RELATED WORK

Although hash algorithms are crucial to Apache Hadoop and Spark, research about how dedicated hash function perform is hard to find. Thus, the found related work is comparably limited.

In 2008, He et al. [4] did investigate on using graphics card processing on Apache Hadoop and mentioned the potential impact of a good or bad hashing function. However, they did not discuss the actual impact of a bad hash function.

Other work from Katsoulis et al. [5] does use hash functions extensively, i.e. for joins, but again does not discuss any performance impacts of hash functions. However, they further strengthen the reliance on hash functions with their research.

Bertolucci et al. [6] did discuss big data partitioning in Spark in 2015 but again did not focus on hash functions. They focused on the difference between dynamic and static partitioning.

Lastly, a closely related work from Kocsis et al. [7] proposed a method to repair broken Apache Hadoop hash functions but did not discuss the performance impact of bad, broken hash functions in detail. Their key contribution is an algorithm to automatically fix – or

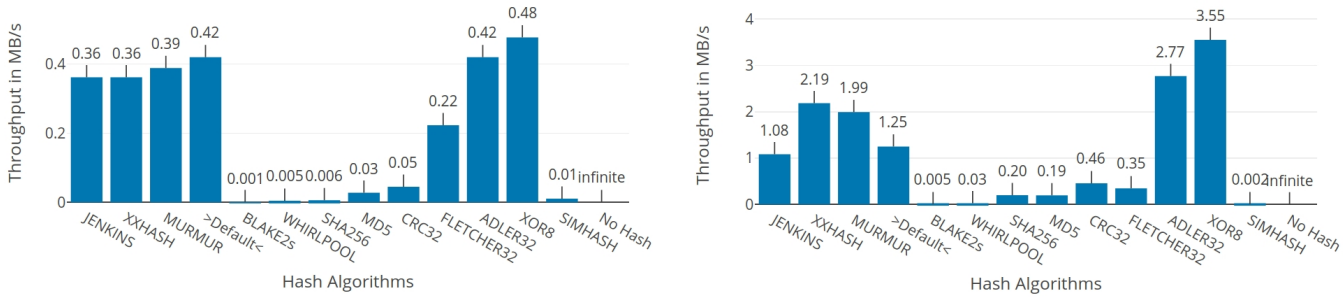

Figure 1: The possible throughput of the different hash functions on a single 3.6 GHz core clustered by type. On the right side the throughput of a 11-byte input and on the left side the throughput of a 200 MB input. The four leftmost on each side are part of the General Purpose Hash Functions. The next four are in the category Cryptographic Hash Functions. The following four are Cyclic Redundancy Checks for error detection. The last two are in the Special Hash Function category.

optimize – hash functions to perform better. Also, Ramakrishna et al. [8] describe the importance of good hashing functions for high-performance computers, but limit themselves to a theoretical approach.

To the best of our knowledge, there is no related work which discusses how a hash function impacts the overall performance of Apache Hadoop or Spark. While there is research using and relying on hash functions, we could not find any which puts numbers on the actual impact. However, it is commonly accepted that hash functions do influence the performance of Apache Hadoop and Spark significantly. Nevertheless, a detailed analysis was not found.

3. IMPLEMENTATION

3.1 Hash Functions

We selected 14 different hash function approaches and divided them into four categories. Our defined categories are General Purpose Hash Functions, Cryptographic Hash Functions, Checksums as Hash Functions, and Special Hash Functions. The first category contains hash functions which are used to divide data. They should serve the purpose of Apache Hadoop and Spark perfectly as they are supposed to be well balanced between speed and hash value distribution. General purpose hash functions mostly use a limited amount of operations and are not difficult to implement. The second category contains hash functions for cryptographic purposes. They are less efficient but provide the certainty that there is no calculating back from the hash value to the original value. One way to ensure this is the Feistel structure. The input is manipulated in multiple rounds and in each round it is split up into two segments. Those segments are exchanged. One of them gets XORed with a set of predefined numbers and the other gets again XORed with this result. Through many such iterations both security and distribution are ensured. Although this security not necessary for Apache Hadoop and Spark, they are good markers for *bad* hash functions, as they are usually very slow. In the third category, common checksum algorithms can be found. They are designed to be quick in generating but are created only to detect transmission errors and thus distribution is disregarded. In Fig. 2 that such functions tend to have certain values which are more likely the result than others. Their inner workings typically rely on a single iteration in which mathematical operations – also modulo operations – are applied in combination with predefined numbers. Thus, their hash value distribution might be worse than standard hash functions but their throughput is higher The last category contains special functions which have a hash like behavior with certain additional properties. For each category four representative are chosen. The Special Hash Functions category contains two functions.

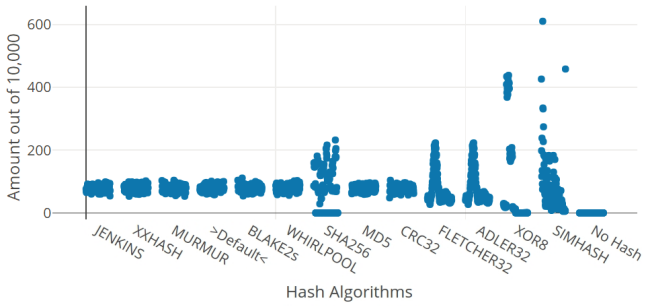

Figure 2: The number of entries of resulting hash values modulo 128 of 10,000 numbers in text format. Each dot represents one of the 128 modulo buckets. Both general-purpose hash functions and cryptographic hash functions have a similar distribution. Checksums and special hashes have an expected worse distribution. The 10,000 dot of *No Hash* is not displayed.

An overview of how the individual functions performed on a standard machine is depicted in Fig. 1. On the left side the throughput of an 11-byte input and on the right side the throughput of a 200 MB file. As you can see, some hash functions perform better for big inputs and some perform better for small inputs. The results are grouped by the categories and the throughput is calculated in Megabytes per second with a single 3.6 GHz processor on Ubuntu 16.04. Fig. 2 shows the distribution of the hash functions. It shows how many out of 10,000 numbers in text format fall in a single of 128 buckets. If the bar is thinner and lower, it represents a better distribution. A scattered plot is a bad distribution. The general purpose hash functions have a better distribution than expected. For the cryptographic hash functions, SHA256 is a surprising outlier. The others are marginally better than the general purpose hash functions. Both checksums and special hash functions have a bias in the distribution which can end up in reducing parallelism.

All evaluated hash functions are implementations which can be found on the Internet. This approach reduces potential implementation errors.

3.1.1 General Purpose Hash Functions

The first representative is the default hashCode() [9] implementation of Java. It is commonly used in Java software and implemented through native code. Thus, the implementation is platform dependent but usually delivers an output well balanced between speed and distribution.

The second hash function in the category is the Jenkins [10, 11] hash. In our test environment, it performs slightly worse than the default implementation of hashCode(). It should serve as a good reference point for how well the default implementation is on the AWS setup compared to the local one.

As a third hash MurmurHash [12, 13] is chosen and xxHash [14,15] as the fourth hash. Both are fairly recent developments and focus on the throughput. xxHash is almost capable of twice the throughput than the default Java implementation on Ubuntu 16.04.

3.1.2 Cryptographic Hash Functions

The first chosen representative of this group is BLAKE2s [16,17]. It is a novel cryptographic hash function which promises superior speed and safety. Strangely, the implementation we use performs worse than the other cryptographic hash functions. Nevertheless, we kept it, as it gives the behavior of a very slow hash function.

The second cryptographic hash function is Whirlpool [18,19]. It is a rather recent development which promises good performance with superior security. In our test setup, it is the second slowest representative in this category.

The last two representatives are MD5 [20, 21] and SHA256 [22]. Both have a similar and higher throughput than the others in this category. While MD5 is officially insecure, SHA256 has not suggested for use anymore. However, both are still commonly used in real-world applications, which makes them good choices.

Since all cryptographic hash functions result in more than 32-bit output, only the first 32 bit is used in our analysis.

3.1.3 Checksums as Hash Functions

Both Fletcher32 [23, 24] and CRC32 [25, 26] have a similar throughput behavior. They are developed specifically for error detection and not for code distribution. However, they do convert an arbitrary input to a 32-bit output and thus can function as hash functions.

With Adler32 [27, 28] we chose a better performing checksum algorithm for comparison. Similarly to the previous ones, it generates 32 bit of output. The superior speed is traded for a slightly worse error detection – resulting in less distribution.

Lastly, we added the XOR checksum for comparison. It is the only function, which generates 8-bit output by simply applying the XOR operation to every byte of the input. Thus, it has the worst distribution of all but outperforms any other function.

3.1.4 Special Hash Functions

Additionally, we added two special hash function to the analysis. With SimHash [29,30] we evaluate a hash function which produces similar hash values for similar input values. Although this should not result in an advantage for Apache Hadoop and Spark, we chose it for curiosity purposes.

Lastly, we added a *no hash* simulation for comparison purposes. This function always returns the same hash values for arbitrary input. Thus, all the data will be gathered at a single reducer and the load is not balanced at all. However, it returns the value instantly and does not need any operations.

3.2 Word Count

The source packages of both Apache Hadoop and Spark provide example implementations for the WordCount problem. We modified these samples and use them for our analysis.

In the Apache Hadoop case, we override the standard *org.apache.hadoop.io.Text* class with our custom implementation. This implementation implements a custom *hashCode()* implementation, which applies a selected hashing algorithm for the input. By overriding the *hashCode()* function of the *key*, it is automatically applied whenever a hash code is necessary.

For Apache Spark, we create our own class *MyString*. It acts as a wrapper class for *Strings* and overrides

the *equals()* and *hashCode()* functions. The *hashCode()* functions selects the hashing algorithm to use. We use this class for the computations with Apache Spark and thus the custom hash function is in use wherever possible.

3.3 PageRank

The PageRank algorithm is only used in Apache Spark due to its iterative nature. There exists a standard implementation in the source code of Apache Spark which we modified and used for this paper. Similarly to the Apache Spark implementation of the WordCount example, we created the class *MyString*. It acts as a wrapper class for *Strings* and overrides the *equals()* and *hashCode()* functions. The *hashCode()* functions selects the hashing algorithm to use. With some modifications of the provided example, we use the *MyString* class instead of normal *Strings* and thus use the custom hashing algorithm wherever possible.

4. TEST ENVIRONMENT

4.1 AWS Infrastructure

Amazon provides a bunch of effective frameworks like Amazon Elastic Compute Cloud(EC2) and Amazon Elastic MapReduce(EMR). Normally EC2 provides remote servers for a user to customize, in this project, we find that spending a large amount of time setting up a cluster is ineffective since we will run on both Hadoop and Spark and our target is the hash algorithms. In this case, we select EMR as our remote environment. Users can easily edit a configuration of framework and the whole cluster can be fully functional within ten minutes. All EMR instances are completely based on Amazon EC2. As commonly known, the file system of Hadoop and Spark is Hadoop Distributed File System(HDFS), on EMR, you can choose to use the traditional HDFS or EMRFS with Amazon S3. Considering the limited funds, our project runs on the traditional HDFS. Amazon Command Line Interface(CLI) is the interface for input.

For this project, we implement our programs on a cluster consisted of three m4.large nodes powered by two cores of Intel Xeon E5-2686 v4 or E5-2676 v3 with 8 GB memory. The version of Hadoop is 2.7.3, and for Spark is 2.2.0. The number of partitioners in Hadoop is set to be 5. However, the number of partitioners in Spark is set automatically to be 16 and won't be affected by spark-submit commands, which can better exploit the strongpoint of parallelization.

4.2 Local Execution

For local running we set up Hadoop on two machines to distribute our work, and the features of one machine are Intel Core i5-7360U with 4 GB memory and another is Intel Core i5-5257U with 8 GB memory. Hadoop is installed in pseudo distributed mode with 5 partitioners, and Spark is running in local mode with 5 partitioners as well.

4.3 Datasets

Three datasets are used for Word Count, first is the air quality index comes from the meteorological stations in China [31], the number size is around 200,000, which contains both integers and decimals. Second is the Lorem Ipsum generated from an online website [32], Lorem Ipsum is a text consists of meaningless words generated randomly to minimize the influence of words' meanings in design field, Lorem Ipsum does have duplicate words and is easy to control the size with accessible generators, and is similar to the structure of real articles. The size for local test is 20,777,978 Bytes and for EMR is 10,388,990 Bytes. On local the size is doubled to make sure that Hadoop can generate enough partitioners. Last is a 49.4 MB duplicate questions file from Quora, which includes over 400,000 question pairs [33] that have the potential to be duplicate, this dataset is closer to real work and can better simulate the situations in real tasks.

As to the PageRank part, we wrote a small program that generates a list of websites with their names represented as numbers, followed by a random PR value. We generated a list of 100,000 websites for EMR and a 1,300,000 websites one for local test.

5. EVALUATION

5.1 Performance

5.1.1 *Word Count Performance on Spark*

We took the overall duration time of collective the part on EMR to mimic the practical time elapsed in a cluster, and for local, we took the same part of the time from the terminal output to get the most accurate result for experimental purpose. The test is first launched on Spark EMR, and for every dataset we run it ten times to get the mean value except for some extremely slow hash algorithms, only three to five times is launched to make sure the slow performance is not an exception.

First comes the result of air quality case, performances of hashes running on air quality can be seen in Fig. 3. Overall the algorithms except BLAKE2s have the similar time performances, Adler32 and xxHash can even be a little faster than the default hashCode in some rounds in such a relatively small datasets.

Time consumption from the Lorem Ipsum test is given in Fig. 4. The performance of cryptographic hashes and special functions start to show a trend of slowing down especially BLAKE2s and it's noteworthy that the performance of xxHash even slightly exceeds the one of

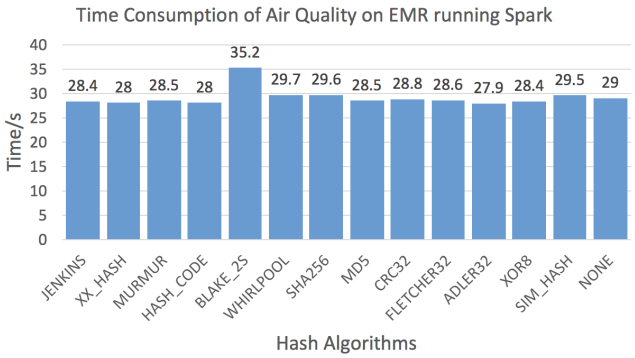

Figure 3: The average time consumption of Spark Word Count program in collective part. There's no significant performance difference in this case and even ADLER32 can be slightly faster than default hash function.

hashCode in this case. However, the differences between other hashes are still not clear enough that we couldn't make any reasonable speculation.

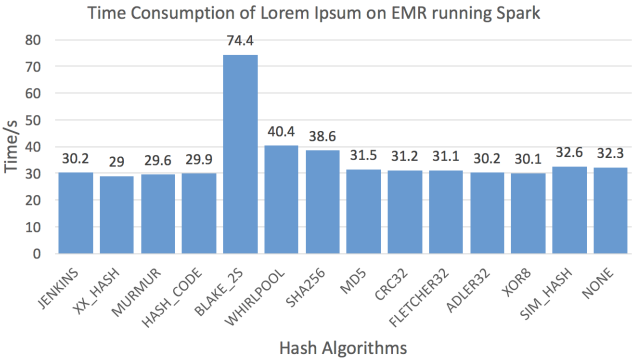

Figure 4: Time consumption of running Lorem Ipsum on Apache Spark shows the relatively worst performance of cryptographic hashes and slightly slow performance of special hashes.

Thus we come to the Quora duplicate questions dataset, the time performance is given in Fig. 5. This datasets is big enough and the test on BLAKE2s and No hash took so long that we can't even get a result, so we simply represent it as N/A.

We can tell that General hashes do benefit from their superb throughput and still remain the best, while the cryptographic hashes and special hashes have been completely unusable for their more than twice time, which makes sense due to their small throughput. The Checksums except XOR8 also shows the correlation trend but the degree is much lower. Under no hash circumstance, all records are delivered to the first reducer, so there is little parallelization in this case and the performance is the worst.

Questions can be raised that XOR8 was supposed to

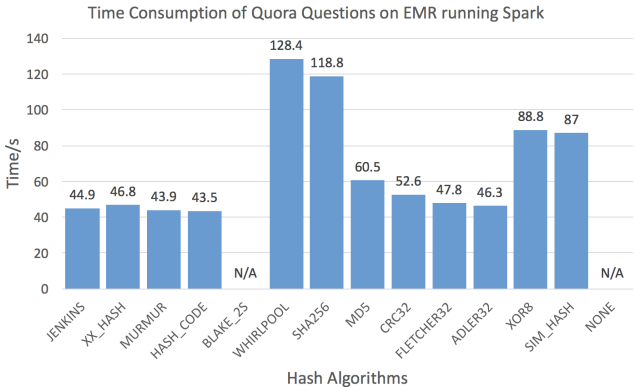

Figure 5: The figure shows the obvious difference between hash types and generally shows a reasonable negative correlation with their throughput level but not completely fitted.

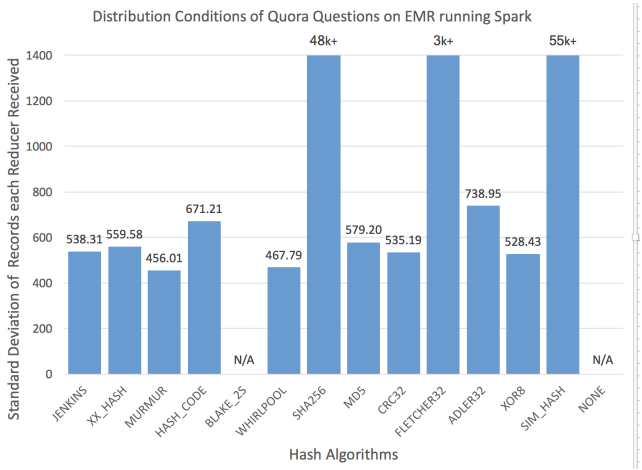

Figure 6: This picture is the distribution conditions taken from standard deviation of records each reducers received.

show a faster speed as the throughput level is good but here is definitely the opposite. And the inappropriate correlation level in general purpose hashes also indicates us that besides throughput, more reasons can be affecting the performance of hashes. So we collected the number of records each partitioner receives to see how records are partitioned and distributed to reducers. We took the standard deviation of data to demonstrate the distribution level, and as the duty of hashes is to let out records as evenly as possible, so here higher distribution level represents a worse job.

However, when we analyze the Spark distribution result, we are surprised to find Adler32 and Fletcher32 function strangely on Spark with large datasets that the total records generated are more than they are supposed to be, but when we verify this situation with small datasets and on Hadoop, they never happened. Nevertheless comparing with other results, the result Fig. 6 still shows the same pattern and we can still take this as a reference.

General hashes with higher throughput unluckily have relatively worse distribution, and it's interesting that the hashes with better throughput right have the worse distribution, this drags down the overall speed and as a result, the difference between general purpose hashes are neutralized. The cryptographic hashes are greatly limited by throughput so the distribution won't make them better. Checksums' performances are hard to predict according to the Fig. 2, the records are both likely to be scattered evenly or not. In this case looks like Fletcher32 shows a scattered pattern while Adler32 is acceptable, and Adler32's better throughput made it better than Fletcher32. Special Hashes shows such a bad distribution that we should avoid implementing them in partitioner.

For the local environment, our data illustrate the similar result Fig. 7,Fig. 8, and the significant difference is from Checksums, Adler32 and Fletcher32 seems to be steady this time but XOR8 turns to be extremely bad. For this time we are able to get the distribution of records of BLAKE2s, which is quite acceptable, but the little throughput still became its bottleneck and showed an unsatisfactory outcome.

5.1.2 *Word Count Performance on Hadoop*

The problem occurred on Spark didn't happen on Hadoop, so the result of Hadoop can better describe the practical performances of Hashes. And for Hadoop, we are mainly going to talk about the result running on local machines since the results from EMR are alike and running on pseudo mode can cut down the unpredictable factors on a distributed cluster as possible, e.g. the network communication latencies between EMR instances.

On Hadoop, the previously concluded results are still

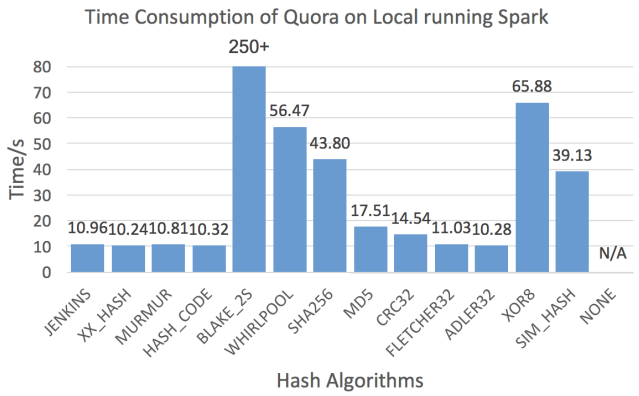

Figure 7: Time consumption on local Spark shows the similar pattern as on a remote cluster on EMR which proves the running mode of Spark won't change the performance of hash algorithms.

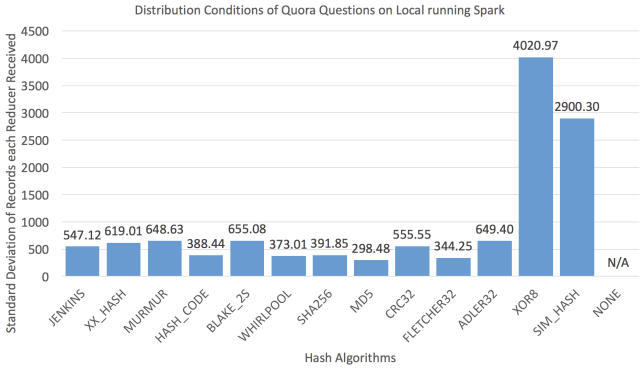

Figure 8: Distribution of local Spark is able to get the data of BLAKE2S, which proves the cryptographic hashes' performances are dominated by throughput regardless of the tolerable spread.

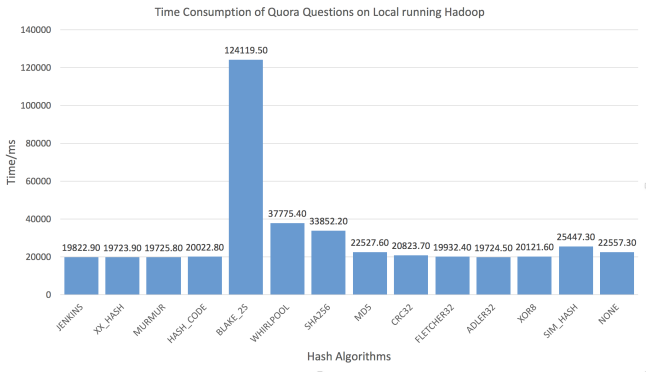

Figure 9: The main discrepancy between Spark and Hadoop is the performance of XOR8, which can be an excellent proof of the impact of distribution on the task speed.

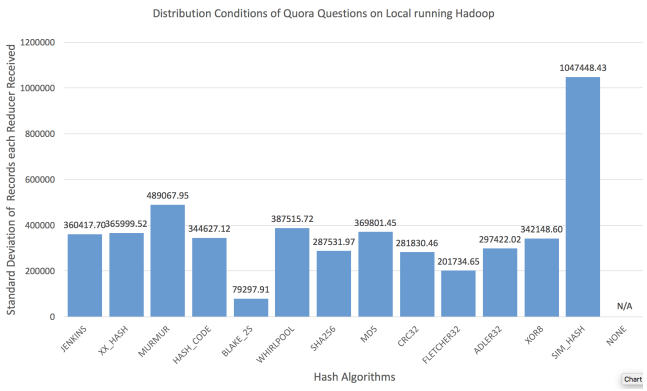

Figure 10: Distribution conditions of XOR8 are greatly improved this time, and combining with its outstanding throughput, the speed increase in Fig. 9 is completely within our expectation.

compatible with most of the hashes and the only noticeable discrepancy between the tests on Spark and Hadoop is the performance of XOR8, which can be completely expected thanks to the occasionally well-distributed records. The results of Hadoop also indicate the consistency of how the Hashes work in both frameworks.

5.1.3 PageRank Performance

PageRank is a iterative algorithm that works completely different from Word Count and the input data contains only pairs of integers. Data collected from EMR and local is separately given in Fig. 11,Fig. 12. The time consumptions and scattered records are the same as what we got from Hadoop, and this shows the independence of hash algorithms and what exactly the task does in this two programs has nothing to do with the performance of hash functions since their duty is to map records from mappers to appropriate reducers, and all they need to care about is the speed and the uniformity.

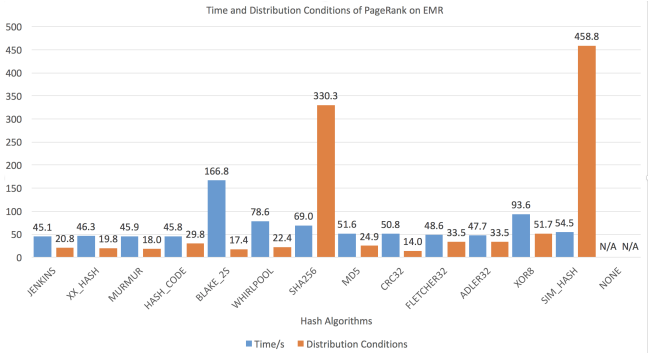

Figure 11: Results of PageRank from EMR have the same features and correlations as the previous results.

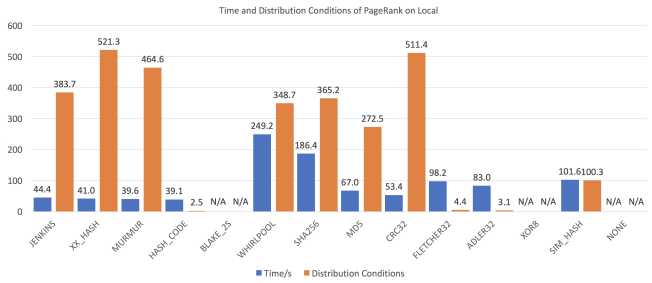

Figure 12: CRC32 shows a bad distribution this time and the speed is correspondingly slower than the performance on EMR. The small throughput can make the decrease of speed less severe than the trend in distribution.

5.2 Future Work

In this program we tested a limited number of hash algorithms try to find out the relationship between the performance and the type of hashes, and only when in small datasets we do find some hashes such as xxHash could have the potential of replacing the default hash function to provide better performance. But there're still more functions worth our implement and test. We do believe that after relatively mature tests, we can provide a guidebook for when and where to implement which kind of hashes to avoid blindly trial and error when the default hash is not suitable under specific circumstances, but that would require a lot of time and effort to collect and test the commonly used hashes.

We do find the correlations between the performance of tasks and the throughput and uniformity of hash algorithms, but these are inferred from small data sample and we even encounter with some strange behaviors about Adler32 and Fletcher32 on Spark with large datasets, so we still need more tests with various datasets types and sizes to prove our conclusion.

The correlations between performance, throughput, and distribution are qualitative rather than quantitative. It would definitely better if we can find some quantitative relation between the three factors, and that can be attempted by implementing machine learning since we do have a bunch of original data.

For the coding part, the records counting part is hasty and can only work for local Hadoop part, it can definitely be debugged and optimized for general purpose implement on Hadoop and Spark and turn into an effective performance monitor tool.

AS mentioned before, there are strange issues with Adler32 and Fletchere32, also the performance of BLAKE2s is unexpectedly slow in that it's widely accepted the performance of BLAKE2s is much better than MD5 and SHA256, we would like to figure out why the common sense performance of hashes is different from what we got in distributed frameworks.

6. CONCLUSIONS

14 hash algorithms divided into 4 groups are tested in this project, and unluckily only xxHash has a slight potential of replacing the default algorithms when running with small datasets. We can tell from the data that general purpose hashes always have the best performance and are highly predictable and similar to each other, replacing with general purpose hashes is a safe way if necessary.

The cryptographic hashes only can compare with others under small datasets, their unsatisfactory throughputs are bottlenecks for the speed, thus as the name said, they are better used for encrypting jobs and completely not your choice for a replacement in parallel systems.

Checksums generally possess the worst uniformity of distribution, as a result, their performances are unpredictable and can sometimes cost heavily, they are also not recommended for distributing records.

Special groups are mainly set for reference and comparison, and they are unusable in most cases, however, their capability can help us understand how a bad hash or even no hash would influence the system, which emphasizes the importance of hash algorithms in the cloud computing.

Basing on the data collected about throughput and distribution, we can also conclude that the performance of hashes in the partitioner in greatly dominated by these two factors, and usually the impact from throughput is the foundation of the distribution and in a mass decides how it functions inside partitioner.

7. REFERENCES

[1] A. Hadoop, "Hadoop," 2009.

[2] A. Spark, "Apache spark: Lightning-fast cluster computing," 2016.

[3] M. Bianchini, M. Gori, and F. Scarselli, "Inside pagerank," *ACM Transactions on Internet Technology (TOIT)*, vol. 5, no. 1, pp. 92–128, 2005.

[4] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 260–269.

[5] S. Katsoulis, "Implementation of parallel hash join algorithms over hadoop," *Master of Science. School of Information University of Edinburgh*, 2011.

[6] M. Bertolucci, E. Carlini, P. Dazzi, A. Lulli, and L. Ricci, "Static and dynamic big data partitioning on apache spark." in *PARCO*, 2015, pp. 489–498.

[7] Z. A. Kocsis, G. Neumann, J. Swan, M. G. Epitropakis, A. E. Brownlee, S. O. Haraldsson, and E. Bowles, "Repairing and optimizing hadoop hashcode implementations," in *International Symposium on Search Based Software Engineering*. Springer, 2014, pp. 259–264.

[8] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Transactions on Computers*, vol. 46, no. 12, pp. 1378–1381, 1997.

[9] Oracle, 2017, https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html#hashCode%28%29.

[10] B. Jenkins, "Jenkins hash," 1997.

[11] vijaykandy, 2017, https://github.com/vkandy/jenkins-hash-java/blob/master/src/JenkinsHash.java.

[12] A. Appleby, "Murmurhash 2.0," 2008.

[13] Viliam Holub, 2017, https://github.com/tnm/murmurhash-java/blob/master/src/main/java/ie/ucd/murmur/MurmurHash.java.

[14] Y. Collet, "xxhash-extremeley fast hash algorithm," 2016.

[15] koron, 2017, https://github.com/koron/java-xxhash/blob/master/src/main/java/net/kaoriya/xxhash/XXHash.java.

[16] J.-P. Aumasson, S. Neves, Z. Wilcox-OâĂŹHearn, and C. Winnerlein, "Blake2: simpler, smaller, fast as md5," in *International Conference on Applied Cryptography and Network Security*. Springer, 2013, pp. 119–135.

[17] legarspol, 2017, https://github.com/legarspol/java-blake2s/blob/master/src/com/uliamar/Blake2s.java.

[18] V. Rijmen and P. Barreto, "The whirlpool hash function," *World-Wide Web document*, vol. 72, 2001.

[19] Vincent Rijmen, 2017, https://github.com/SapphirusBeryl/OSRSCD/blob/master/src/com/sapphirus/osrscd/hash/Whirlpool.java.

[20] R. Rivest, "The md5 message-digest algorithm," 1992.

[21] rosettacode, 2017, https://rosettacode.org/wiki/MD5/Implementation.

[22] D. Eastlake and T. Hansen, "Rfc 6234: Us secure hash algorithms (sha and sha-based hmac and hkdf)," *IETF Std*, 2011.

[23] J. Fletcher, "An arithmetic checksum for serial transmissions," *IEEE Transactions on Communications*, vol. 30, no. 1, pp. 247–252, 1982.

[24] maplog, 2017, https://github.com/trebogeer/maplog/blob/master/src/main/java/com/trebogeer/maplog/checksum/Fletcher32.java.

[25] P. Koopman, "32-bit cyclic redundancy codes for internet applications," in *Dependable Systems and*

Networks, 2002. DSN 2002. Proceedings. International Conference on. IEEE, 2002, pp. 459–468.

[26] Princeton, 2017, https://introcs.cs.princeton.edu/java/61data/CRC32.java.html.

[27] Wikipedia, 2017, https://en.wikipedia.org/wiki/Adler-32.

[28] psiegman, 2017, https://github.com/psiegman/epublib/blob/master/epublib-core/src/main/java/net/sf/jazzlib/Adler32.java.

[29] C. Sadowski and G. Levin, "Simhash: Hash-based similarity detection," 2007.

[30] zhangcheng, 2017, https://github.com/sing1ee/simhash-java/blob/master/src/simhash/Simhash.java.

[31] M. L. R. L. Z. S. E. C. T. L. Yu Zheng, Xiuwen Yi, 2015, https://www.microsoft.com/en-us/research/publication/forecasting-fine-grained-air-quality-based-on-big-data/?from=http%3A%2F%2Fresearch.microsoft.com%2Fapps%2Fpubs%2F%3Fid%3D246398.

[32] lipsum, 2017, https://www.lipsum.com.

[33] K. C. Shankar Iyer, Nikhil Dandekar, 2017, https://data.quora.com/First-Quora-Dataset-Release-Question-Pairs.