An analysis of software parallelism in big data technologies for data-intensive architectures

Felipe Cerezo[1[0000-0002-0128-0783]], Carlos E. Cuesta[1[0000-0003-0286-4219]] and Belén Vela[1[0000-0003-0604-7312]]

[1]VorTIC3 Research Group
Universidad Rey Juan Carlos
C/ Tulipán s/n, 28933 Móstoles, Madrid, Spain
jf.cerezo.2019@alumnos.urjc.es, carlos.cuesta@urjc.es, belen.vela@urjc.es

Abstract. Data-intensive architectures handle an enormous amount of information, which require the use of big data technologies. These tools include the parallelization mechanisms employed to speed up data processing. However, the increasing volume of these data has an impact on this parallelism and on resource usage. The strategy traditionally employed to increase the processing power has usually been that of adding more resources in order to exploit the parallelism; this strategy is, however, not always feasible in real projects, principally owing to the cost implied. The intention of this paper is, therefore, to analyze how this parallelism can be exploited from a software perspective, focusing specifically on whether big data tools behave as ideally expected: a linear increase in performance with respect to the degree of parallelism and the data load rate. An analysis is consequently carried out of, on the one hand, the impact of the internal data partitioning mechanisms of big data tools and, on the other, the impact on the performance of an increasing data load, while keeping the hardware resources constant. We have, therefore, conducted an experiment with two consolidated big data tools, Kafka and Elasticsearch. Our goal is to analyze the performance obtained when varying the degree of parallelism and the data load rate without ever reaching the limit of hardware resources available. The results of these experiments lead us to conclude that the performance obtained is far from being the ideal speedup, but that software parallelism still has a significant impact.

Keywords: Data-intensive architecture, software parallelism, partitioning, big data technologies, linear scalability.

1 Introduction

The last decade has witnessed an exponential rise in the magnitude of information to be processed, causing the current interest in Big Data technologies. Even the advances in modern hardware have been unable to cope with this growth, and in order to bridge this gap, it has been necessary to resort to the widespread adoption of parallelism. When more processing power is required, more cores are added to a server, or more nodes are

added to a processing cluster. These strategies, known as vertical and horizontal scalability [1], distribute the work between these new elements, which are partially or completely used by the process. In general terms, this approach to parallelism leads to a *linear* performance speedup [2][3], although some contention factors [2][4] must also be taken into account.

But simply increasing the processing power will not make it possible to achieve the required results. This has triggered a new interest in *software parallelism* [5], defined here as the partitioning and distribution strategy of our software. This provides the basis for z-axis (diagonal) scalability [1], in which data elements are partitioned and scattered in the parallel structure, and which is obviously relevant in the context of data-intensive architectures.

However, an adequate use of parallelism in big data tools has often been neglected. Existing tools provide several alternatives that can be used to define this, but they typically lack any sort of guidelines with which to define an optimal setting. This configuration usually depends on the number of threads available, and/or the number of parallel storage elements (partitions, shards) in which to allocate data entities.

This paper intends to analyze how the data partitioning mechanisms that define internal software parallelism in big data tools influence their performance from a comparative perspective. Our goal is to achieve a better comprehension of how these mechanisms affect the system as a whole and how they may affect the design of data-intensive software architectures [6]. In particular, the claim often made is that their parallelism provides the same sort of linear scalability [7] usually expected from parallel computer architectures. It is our intent to verify the validity of these assumptions.

As our goal is to study *software* parallelism, we have designed an experiment in which the hardware is stable and the resources are constant, explicitly avoiding the saturation regime. Rather than employing the approach normally used in hardware performance benchmarks (i.e., maintaining a constant workload while resources are increased), our experiment utilizes the complementary approach, that is, maintaining the same (constant) set of resources while scalability is tested by:

1. Increasing the number of internal parallelism elements, as defined by the system parameters of our big data tools, and
2. Increasing the data load rate in order to verify how much this software scales.

Our working hypothesis is that when the data insertion rate increases, an adequate configuration of these tools should provide a linear speedup of their performance.

Different big data tools use different parallelism mechanisms, even when several of them are variants of the same theoretical concepts. In order to obtain independent results, it is necessary to use several different tools. We focus on different approaches to *data* parallelism, as provided by horizontal partitioning and data sharding.

We designed an experiment in which we studied the impact of internal data partitioning mechanisms with several different configurations, when both the degree of parallelism and the data load rate were increased. The comparable alternatives chosen as representative of internal parallelism were Apache Kafka and Elasticsearch, two well-known and highly performant data-centered big data tools, both of which are widely used and established in industry.

Our results suggest that these configurations provide good initial results, with almost linear scaling, even with quite different speedup factors. But our experiment also shows that there are unexpected limits, clearly imposed by the software itself. The performance results obtained for the two tools are significantly dissimilar, and these scaling limits manifest themselves at very different stages of the experiment.

The remainder of the paper is organized as follows. Section 2 includes a comparison of the parallelization mechanisms of the chosen tools. In Section 3, we provide detailed descriptions of the design of the experiment and of the different choices about the parameters that affect internal parallelism and the increasing data load rate. We then go on to discuss the results of these experiments in Section 4. Finally, our main conclusions and future work are summarized in Section 5.

2 Software parallelism in big data tools

In order to perform an analysis of the parallelization mechanisms and their impact in data-intensive architectures, it was necessary to define the criteria for selecting the adequate Big Data tools. The first one was that we wished to select **open source and independent** big data tools whose use does not require additional software. The second requirement was that the tool should have a **configurable** internal parallelism mechanism. The third one was that we wished to analyze **standalone tools**, we discarded **execution frameworks** as they require the development of additional code in order to test them. The fourth was that the tools should be considered as **mature tools**, that means that they have a history of use and versions and are used widely in the industry. Finally, we also wished to select tools with a large **support and development community**, for which extensive documentation was available. After evaluating several tools, we chose Elasticsearch and Kafka because they comply with all these requirements.

Apache Kafka [8] is an independent, open source, big data tool whose first version dates to January 2011. Apache Kafka aims to provide a unified, high-throughput, low-latency platform to handle real-time data feeds. Kafka is developed in Java and Scala and allows the manual configuration of its internal parallelism mechanism.

Elasticsearch is an independent, open source, big data tool whose first version appeared in February 2010. ElasticSearch provides a distributed, multitenant-capable full-text search engine with web interface and schema-free documents. Elasticsearch is developed in Java. Its internal parallelism mechanism can be configured manually.

The internal data partitioning mechanisms are implemented as follows:

- In Kafka, the data processing unit is the topic (queue), and each topic can be configured with a number of different *partitions*. Each partition is fed and accessed independently, and both the performance obtained and the resources used for each topic will, therefore, depend on how many partitions are configured.
- In ElasticSearch, the data storage unit is the index. Each index can be configured in a number of different storage partitions (called *shards*). When writing to or reading from an index, each different thread actually writes to or reads from each of the shards, thus the application is generating an additional parallelism of its own.

3 Experiment design

In this paper we conduct an experiment with the two big data tools chosen, Kafka and Elasticsearch, in order to analyze their performance (while maintaining the resources constant), varying the degree of internal parallelism and the data load rate.

The objective of this experiment was to verify two different aspects:

- How the performance is affected by the internal data partitioning mechanism,
- How the performance is affected when then load is increased.

The internal parallelism should increase the performance of the system. We studied the impact of different configuration alternatives on the performance.

In our experiment we focused on data load insertion, which requires an intensive use of all the hardware resources: CPU, disk, memory and network.

The following independent variables were, therefore, considered for both tools: (1) number of elements for internal partitioning, (2) data insertion rate.

We began our experiment by using a **basic configuration**: one parallelism element for each instance within the cluster. Smaller configurations do not make sense, as each instance must store at least a part of the data in order to take proper advantage of parallelism. We increased the number of storage elements per instance in each iteration.

We built a **multithreaded process.** Each atomic thread in this process was programmed to insert data at a constant rate. The insertion rate was increased by increasing the number of threads. When we refer to a "x3 load", we are increasing the insertion rate by the factor of three.

We also considered the possibility of performing simultaneous loads on different data entities. Therefore, in another set of tests, we ran two instances of the multithreaded process. Each instance loaded data into a different entity. The different loads within each process were combined in order to obtain a total load from x2 to x8.

Resource usage **monitoring** was key in our experiment. It was necessary for the tools to work without reaching, in any case, the limit of the available hardware resources; that is, CPU, disk, RAM memory or network bandwidth. If this resource saturation had been reached, the results would have been distorted.

A **test time** of 30 minutes was determined for each configuration (internal parallelism and load). This was sufficient time for the tools to pass through the transient start-up period of the load processes and to work in the normal regime.

We chose a **representative dataset** from the big data domain: 2.142.000 records, 25 fields of information with different datatypes, an average length of 312 bytes per record and a standard deviation in the size of the records of 28 bytes. The distinct fields have different cardinalities: from 2 different values up to 2 million.

We carried out our experiment using Elasticsearch and Kafka.

- In the case of Elasticsearch, we ran the insertion experiment 36 times. This covered all the combinations of a degree of internal parallelism from 1 to 6 and a data insertion load from x1 to x6.
- Kafka can support higher loads, so internal parallelism degrees used were 1 to 6, 8, 10 and 12 partitions in each instance, and a data insertion load from x1 to x8.

The use of all system resources, CPU, network, disk and memory remained below the maximum limits of the servers in all cases for both tools. The resources used were fairly homogeneous between the servers throughout the test and remained almost constant throughout the test.

In the part of the experiment in which we insert into two independent data entities, we have executed them only for a selected set of values, in order to evaluate internal data partitioning. We chose those values that had performed best in the previous tests: 2 elements for Elasticsearch (a total of 60) and 12 elements for Kafka (a total of 360).

We also verified whether different proportions of insertions had an impact on the global insertion rate. The fact that more than one combination provides the same total load (for example, x4 can be obtained as x2 + x2 or as x3 + x1) made it possible to assess this aspect.

4 Results

In this section, we present the results of the experiment with regard to the following two aspects: a) internal parallelism b) load increasing.

The following figure shows the number of insertions per second. The horizontal axis shows the **internal parallelism** (number of shards in Elasticsearch and number of partitions in Kafka) used for the experiment. It is important to keep in mind that we used 30 instances, signifying that the numbers that appear are always multiples of 30. Each line represents a different, and increased, data insertion rate (data load).

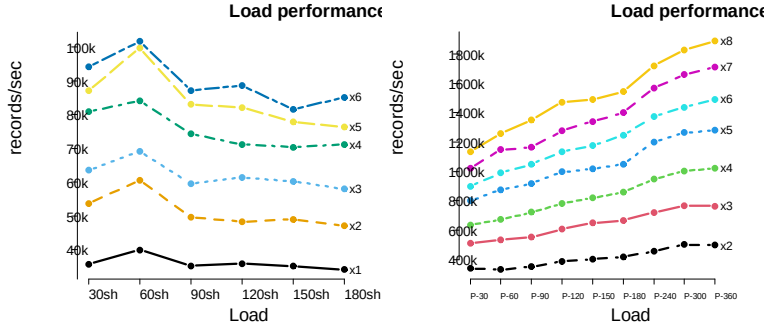

Fig. 1. Data load in Elasticsearch (a) vs. Kafka (b)

The data partitioning mechanism implemented in these big data tools, when compared, show a very different behavior in each case.

In Elasticsearch there is a clear **optimal value of 2 elements per instance,** while in Kafka, a value of *12 elements per instance* was still attained, always increasing the performance of the data insertion process.

With regard to the **load increasing**, the figure below shows the number of insertions into **a single entity** per second. The horizontal axis shows the data insertion rate (data load) used for the experiment. Each line represents the number of parallel elements (number of shards in Elasticsearch and number of partitions in Kafka).

When the degree of internal parallelism is kept constant for different data loads in each iteration of the experiment, the result can be approximated by a straight line. It is possible to consider that the growth with the insertion load is linear, the lowest correlation coefficient for the loads was r=0.983 (Elasticsearch) and r=0.996 (Kafka).

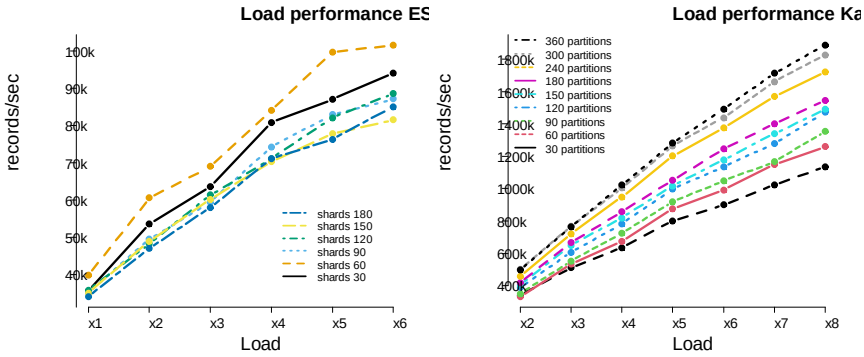

Fig. 2. Load performances of Elasticsearch vs. Kafka

The growth rate should ideally be 100%: one thread obtains 100% output, two obtain 200%, three obtain 300%, etc. Actually, this ratio is 93% in Kafka. Its behavior is very close to that of the ideal theoretical models, with a linear growth and a ratio very close to 100%. In Elasticsearch, this ratio is just 30%. The growth behavior is linear, but its growth rate is very low.

The following figure shows a comparison between the number of insertions (**load increase**) per second into **one** (in red) or **two different entities** (in blue). The ideal behavior is depicted by means of a green line, representing a linear increase with a ratio of 100%. The horizontal axis shows the insertion rate (data load) used for the experiment, while the vertical axis shows the number of insertions per second. Each experimental iteration is represented by a dot; the unfilled dots indicate the data loads that can be generated with different distributions (x4 can be generated as x2 + x2 or x3 + x1). The lines indicate the linear regression.

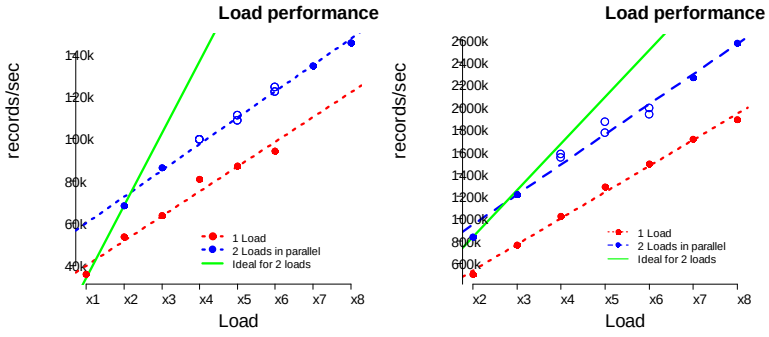

Fig. 3. Load performance when increasing the insert ratio: Elasticsearch vs. Kafka

The results for both tools are very similar. The main difference between them is that Elasticsearch is much further away from the ideal behavior than Kafka is.

The performances of the executions represented by the unfilled dots (different distributions) are very close to each other in every case. It is, therefore, possible to conclude that the performance obtained in these scenarios depends only on the total insertion load, not on how the load is distributed between the two data entities.

The regression line of the two-process load is parallel to the single-process load line for both tools. A higher performance is attained when storing the data in two entities rather than one, and this increase is constant with the load.

In the case of ElasticSearch, there is a low increase in performance (30%) and also a performance bonus in the case of loading in two different data entities. These two insights lead to different architectural scenarios when scaling the data load: (i) a single cluster with a single data entity, (ii) a single cluster with two data entities, (iii) two different clusters, each with a single data entity. The performance will be very different in each of these scenarios. If the insertion ratio is doubled, the performance will increase by 50% (one cluster with one entity, i), 90% (one cluster with two entities) and 100% (two clusters). If the increase is quadruplicated, then we obtain 125% (one cluster with one entity), 180% (one cluster with two entities) and 200% (two clusters).

In the case of Kafka, since the growth with the load is close to 100%, all the scenarios are very similar, and the behavior is consistent.

5 Conclusions

In this paper we study the impact of software parallelism on data-intensive architectures by means of an experiment, in which two factors have been evaluated, namely internal partitioning mechanisms and increasing the load rate. When conducting our experiment, we chose two representative big data tools: Kafka and Elasticsearch. The results of these experiments lead us to conclude that these tools do not behave as expected, that is, the performance obtained is far from the assumed speedup. But our experiment also shows that the software parallelism mechanisms still have a significant impact on the performance. This improvement is entirely dependent on the implementation of the partitioning mechanism in each of the tools.

Throughout the experiment, our goal was always to analyse the performance without ever reaching the limit of available hardware resources. We considered constant hardware resources, signifying that any increment in the processing power had to be provided by internal parallelism mechanisms. Scalability was checked later by incrementing the data load rate.

In the case of internal parallelism mechanisms, it was reasonable to assume that performance would improve if we increased the number of parallelization elements: partitions in Kafka and shards in Elasticsearch. Our results show that this is not always true. In the case of Kafka, this actually occurred: increasing the internal parallelism also increased the performance obtained. But in the case of Elasticsearch, this occurred only until we attained a value that maximized the performance, after which the performance *decreased* as the number of internal parallelism elements *increased*.

In the case of the increasing data load, a linear growth of performance should be expected. The ratio between the data insertions and the processed data led to highly disparate values in both tools: 93% for Kafka and 30% for Elasticsearch. The tools should ideally have a load/processing ratio of 100%, that is, if we double the data load rate, the performance should scale in the same proportion. However, as stated in [2], there are non-parallelizable elements that can make the ratio obtained lower. In Kafka, the result is very close to the maximum expected value, but in Elasticsearch, the resulting value of 30% clearly indicates that it is far from the ideal expected behavior.

The fact that these tools do not perform according to expectations has important implications for data-intensive architectures in big data context. As mentioned previously, the performance of the tool would be different depending on the architectural scenario chosen (number of clusters, number of entities, number of internal storage elements…). Moreover, the difference between these scenarios as regards the performance rate becomes greater with an increasing data insertion rate.

Another implication of our study is that there is a great dependency on the specific big data tool. It is, therefore, necessary to perform detailed tests regarding the behavior of these tools in order to discover whether there is a performance limit and where this maximum value is located. In the case of Kafka, it will be noted that the speedup is maintained as the internal partitioning increases. It is not, however, possible to guarantee that there will be no upper limit with this tool in another scenario. In the case of Elasticsearch, we discovered that using two shards per instance allowed us to attain of an optimal performance as regards data loading. But it is again impossible to guarantee that this value will be optimal in all cases. It is possible that by modifying the number of physical instances within the cluster, this optimal parameter could also vary.

We therefore conclude that big data tools must be evaluated quantitatively for the particular scenario in which they will be used. When evaluating a data-intensive architecture, we cannot straightforwardly assume an increase in linear performance when applying software parallelism mechanisms.

6 References

[1] Abbott, M.L., Fisher, M.T.: The Art of Scalability. Pearson (2009).
[2] Amdahl, G.M.: Validity of single-processor approach to achieving large-scale computing capability. In: Proceedings of AFIPS Joint Computer Conference. AFIPS (1967).
[3] Gustafson, J.L.: Reevaluating Amdahl's Law. Commun. ACM. 31, 532–533 (1988). https://doi.org/10.1145/42411.42415.
[4] Gunther, N.J.: Guerrilla Capacity Planning: Tactical Approach to Planning for Highly Scalable Applications and Services. Springer (2007).
[5] Pacheco, P. An Introduction to Parallel Programming. Morgan-Kaufmann (2011).
[6] Cerezo, F., Cuesta, C.E. & Vela, B.: Phi: a software architecture for big & fast data. Submitted for publication (2021).
[7] Gunther, N.J., Puglia, P., Tomasette, K.: Hadoop superlinear scalability. Commun. ACM. 58, 46–55 (2015). https://doi.org/10.1145/2719919.
[8] Kafka (2021) Apache Kafka Project: https://kafka.apache.org/. Accessed 21 January 2021.