Exploiting key-value data stores scalability for HPC

Cesare Cugnasco, Yolanda Becerra, Jordi Torres, Eduard Ayguadé

Barcelona Supercomputing Center
Department of Computer Architecture - Universitat Politècnica de Catalunya
{cesare.cugnasco,yolanda.becerra,jordi.torres,eduard.ayguade}@bsc.es

***Abstract*—BigData revolutionised the IT industry. It first interested the OLTP systems. Distributed Hash Tables replaced Traditional SQL databases as they guaranteed low response time on simple read/write requests. The second wave recast the data warehousing: map-reduce systems spread as they proved to scale linearly long-running computational workloads on commodity servers. The focus now is on real-time analytics. Being able to analyse massive quantities of data in a short time enables multiple HPC applications and interactive analysis and visualization. In this paper, we study the performance of a system that employs the DHT architecture to achieve fast in local analysis on indexed data. We observed that the number of keys, nodes, and the hardware characteristics strongly influence the actual scalability of the system. Therefore, we developed a mathematical model that allows finding the right system configuration to meet desired performance for each kind of query type. We also show how our model can be used to find the right architecture for each distributed application.**

I. Introduction

In the last few years, distributed key-value databases have become a standard and powerful solution to achieve high-performance in distributed applications. In our research group, we developed several HPC applications [5] [8] [9] which run on key-value databases to achieve performance and scalability. However, each of these applications had different requirements and characteristics, so we had to find the best architecture and configuration for each case. This means we have to speculate about which aspect may be the primary bottleneck in our distributed system; create a prototype; test and profile the prototype to finally check if our assumptions are correct. If they are not, we have to iterate the whole processes. It goes without saying that this method can be extremely expensive, especially as we observed that each case tends to suffer from different problems. Also, in many cases, we have a trade-off between the various factors, thus it is difficult to find the right balance. A typical case is deciding when to use a master-slave or a peer-to-peer approach: a master with a centralised logic is easier to implement but the capability of a single node might constrain the performance.

Likewise, we found a similar trade-off when assigning jobs to nodes: we can use pseudo-random policies, which are faster and do not need a master, but then we might have workload imbalance. Additionally, we can alleviate the imbalance by partitioning the work in more and smaller tasks or by replicating the information in more servers so that clients can pick the least loaded replica. Both cases have an overhead in terms of CPU, storage or cache affinity.

How can we find a good balance between these aspects? Can a one-size-fits-all solution exist? Should a system that aims to few milliseconds response time have the same infrastructure of a batch-oriented one?

In this paper, we propose a benchmarking methodology that allows summarising in few metrics the behaviour of distributed systems so that we can understand which are the limits of such a design. Also, we provide a mathematical model that - once fed with the results of our benchmarking - can guide the improvement of the system architecture by giving precious insights about which components of the system are or will be the bottleneck at any given level of parallelism. Thanks to this model, a developer can, in front of a set of technologies and SLAs, choose the right architecture for its system.

The paper is structured as follow: Section II describes some background required to understand some parts of the work presented, Section III introduces the case of study that motivated this work, Section IV describes our benchmarking methodology while Section V applies such methodology to a particular case and analyses the results. Section VI presents our performance model while Section VII illustrates few examples of insight we can derive from the model. Section VIII provides the related works. Finally, in Section IX we draw our conclusions, and we discuss possible future works.

II. Background

A well designed distributed system should clearly distinguish between business logic design and how the system distributes mainly for two reasons: reusability and separation of concerns. Indeed, we want to write code that works well for different applications; without requiring any algorithm modification. We prefer to entrust the underlying distributed platform to configure and optimise the algorithm's execution.

Following this principle, modern distributed BigData platforms, such as Apache Hadoop, Spark or PyCOMPSs, allow writing an algorithm as a chain of operations to perform on data, leaving the platform to decide where and how to execute each step. Similarly, NoSQL databases, abstract the distribution of the data among servers, relieving the client from choosing where to search for an item or how to handle failures. The developer is only required to describe the application data model, while the database takes care of uniformly distributing the load and preserving consistency.

However, the data modelling and system behaviour are not entirely orthogonal [6] [8]. For instance, in Distributed

 DOI: 10.1109/ICPPW.2017.25

Hash Table (DHTs) databases, the cardinality of the key has a significant influence on how the workload splits among servers. Indeed, each distinct key randomly maps to a server with the optimistic assumption that - with a "high enough" cardinality - requests will spread uniformly. In other words, we have the undesired situation where a business logic related decision prejudices the system performance.

We can use Apache Cassandra [10] as an example, a well-known NoSQL database which adopts a wide columnar layout. In abstract terms, we can describe it as a partitioned distributed *HashMap* where each entry contains another *SortedMap*. Indeed, Cassandra has two levels of indexing. The outer level is a distributed hash table built on what is named *Partition Key*: the database randomly assigns each key to a server, and all values with the same *Partition Key* stay on the same machine. Then, each Cassandra node creates a local index for all elements sharing the same *Partition Key*. This index is built on what is called *Clustering Key*, which represents the second level of indexing. As it stores the entries ordered, it allows efficient access to ranges of grouped elements.

This double layer of indexing gives the user the possibility to decide which items sort and group together and which ones spread randomly among the cluster.

For example, let's suppose we want to index each phone number in the world: we can choose to group each record by country - e.g. using the national prefix as the *Partition Key* - by city or, at the end of the spectrum, store individually each record. Each choice has its benefits regarding which kind of query can serve efficiently: while the last configuration allows accessing only single users, the others also permit reading and aggregating by country or city respectively.

In the first case, we will have around 200 keys: one for each country. In the second one, we can estimate about one million keys while something of the order of the billions for the single user indexing. Now the question is: if we want to store all this data in ten servers, will it spread uniformly? We can use Formula 1 to estimate with high-probability how many more keys - in proportion - will go in the most loaded node. The formula, here briefly introduced, has a wider description in Section VIII.

$$p \approx \sqrt{\frac{\log n * n}{m}} \tag{1}$$

Where m is the number of keys and n the number of nodes. With Formula 1 we can estimate for the first case that one of the ten nodes will have 27 countries assigned- which is about $\sqrt{\frac{\log 10*10}{200}} = 0.339 \approx 34\%$ more of what would have been a perfect distribution. In the two other cases, as the imbalance decreases with the number of keys, we will expect an unbalance of 0.5% and 0.015%. In the first situation the unbalance problem is evident, but in the second one, when grouping by city, we can encounter a similar problem. Even though the cardinality is high enough to distribute the cities uniformly among servers, some cities are much bigger than others. As a matter of fact, about half of the population lives in the 500 most populated cities, so with such a layout half of queries would have an unbalanced distribution: we can expect to have one node with 21% more load than average. Even worst, doubling the server increases the imbalance to 35%.

Obviously, in this simple example, we could just redistribute the data since we know which grouped elements will have more load. But what if we do not have such information? Or if the popularity of items rapidly changes over time? With this work we aim to study this particular problem, providing a methodology and a mathematical framework that allows evaluating beforehand the performance consequence of any particular data model design.

III. Case of Study

Virtually all distributed applications running on key-value databases have to face the problem of key-value cardinality. However, our research started from a particular use case which pointed out the limit of this problem: the D8tree. In our previous contribution [5], we studied the performance of a novel multidimensional indexing algorithm that employs data denormalization to ensure performance scalability on key-value databases. The D8tree denormalizes by replicating elements into one or more cubes so that the algorithm can complete a query following different reading paths. In simpler terms, we can arbitrarily decide the number of keys we need to access to run a query. However, we lacked a framework that could guide our decision. The dataset we used in our tests is the index produced by the D8tree on the results of the Alya simulator executed on top of the Apache Cassandra database. Alya [16] is an in-house HPC-based multi-physics simulation code. The problem that we have chosen studies how the particles are dragged into the bronchi during an inhalation.

IV. Methodology

In the methodology that we propose we have two different phases: the first one is composed of several steps and it is based on performing a broad range of tests to analyse how different configurations influence the system; the second one consists on defining a statistical model to guide future application designs. In the following subsections we describe the steps that we have performed.

A. Scalability analysis and data model influence.

The first step in our methodology is to analyse how the data model affects the trade-off between a more balanced workload and a higher job fragmentation and thus influences the performance and scalability of the application. As Formula 1 states the expected workload imbalance in a DHT system is inversely proportional to the root of the number of keys. In simpler words, the more keys, the more uniform workload the node will have. On the other hand, more partitions result in more operations, messages, index entries and thus disk accesses.

To study this trade-off, we created a testing prototype to evaluate the behaviour of three data models. All three data models partition the dataset into blocks, which is the data unit to store in the database, and all of them differ in the number of elements per partition. Differences in the amount of blocks in

each model are enough to affect the uniformity of the workload distribution and the number of operations necessary to perform a given query. In section V we describe the results of our experiment and we analyse the influence the data model has on the scalability of our case study.

B. Definition of stages

As we will see in section V, the work imbalance is not always enough to explain the lack of scalability, so further analysis is required. To understand which part of a distributed system is responsible for the lack of scalability it is necessary to study and to analyse in detail the application performance. However, we had to face two problems: first, the system is running asynchronously on multiple servers which means that we have to record, gather and correlate metrics from multiple sources. Second, as requests last less than a second, the common performance tools like Ganglia [1] are useless since they are designed to detect long-running phenomena. In a previous contribution, we proposed Aeneas [6], a distributed high-resolution metrics system which allowed us to store, analyse and correlate a massive quantity of parameters from each of the distributed nodes. At first, we tried to embrace the enormous amount of data by developing analysis and visualisation methods, but we then realised that this approach was ineffective, and we had to take a step back. System metrics, such as page faults, IO requests and many other are essential in understanding if a component is misbehaving, but they do not give any insight on which part of a distributed system is the actual performance bottleneck. We found out that the best approach is to identify the primary data flow phases and to record the time that requests spend in each of them. This approach simplifies the identification of common patterns such as bottlenecks or workload imbalances.

C. Bottleneck identification and analysis

Once we identified the critical phases in our application, we can analyse the timings by searching for possible architecture weaknesses. For example, if we observe that requests spend a considerable amount of time during phase **master-to-slave**, we can hypothesise we have a network problem. Similarly, if we see that requests spend significantly more time **in-queue** in one node, we might investigate on the workload imbalance.

Section V presents the results of the battery of experiments that we have performed to analyse each bottleneck and how the conclusions of our analysis allow us to improve the implementation and the deployment of our application.

D. Statistical model

While the performance analysis allows improving the efficiency of the system, our final goal is to generalise our results into a statistical model which allows exploring the feasibility of new architectures. Indeed, with the mathematical model, not only we can find the optimal trade-off between requests granularity and workload balance, but we can also simulate the performance characteristics of new and arbitrary complex architectures, giving us a valuable tool to research which is the best solution for each particular use case and platform. In section VI we describe how we have defined the model for our case study.

V. Performance analysis

We have implemented a prototype to run on real data generated by the Alya simulator and indexed by our D8tree indexing system. In this prototype multiple nodes have to compute a simple aggregation — count by type — over one million elements grouped in blocks of various size. The tests starts when a master asks the slaves to compute the aggregation over data stored locally. Each slave accesses only the local blocks using the underlying key-value data store. In this simplified prototype, the master knows from the beginning which are all the requests it has to issue. The dataset is composed of one million elements and we used three data models with different workload distribution characteristics. We named the three data workloads as follow:

coarse-grained: 100 partitions, of 10,000 elements each.
medium-grained: 1,000 partitions of 1,000 elements each.
fine-grained: 10,000 partitions of 100 hundred elements each.

As the data is indexed with the D8tree system, the elements are stored in space partitions named **cubes**. We selected - in a pre-query phase - all the cubes with sizes that matched the three workloads. We picked at random cubes with one hundred, one thousand and ten thousand elements and we pre-computed the list of keys each workload has to read.

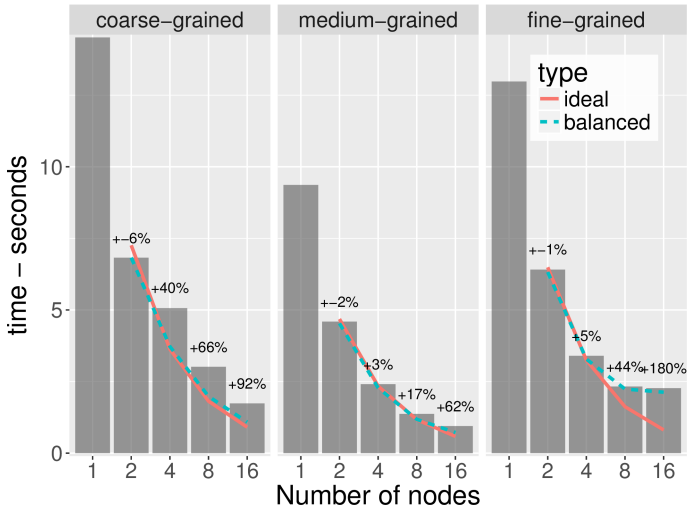
Our tests ran on on-premises servers equipped with two Intel Xeon Quad-Core L5630 with a maximum clock speed of 2.13GHz for a total of 8 cores and 16 threads and 24 GB of NUMA RAM. Each server had a SATA2 SSD and a rotational Hard disk and Intel Gigabit Ethernet as network interface. The three tests ran on the same database table. We run the tests on clusters of increasing sizes: 1, 2, 4, 8 and 16 nodes.

A. Influence of the workload distribution

In this section, we describe the first set of tests we ran on our prototype system, focusing on how the data model influences the scalability of the application. Figure 1 shows how the three cases reveal different scaling profiles when doubling the number of servers in the system. The bar shows the time we observed; the solid line (labelled *ideal*) shows the query time we should have experienced if the query was scaling linearly while the dotted line (labelled *balanced*) shows the time we would have if the workload was distributed perfectly. The labels on the bars state the relative difference between the ideal and real times. The *balanced* line is calculated by counting how many queries each node served and measuring the relative difference between the most loaded node and the average. Finally, we estimated how much time the query would have run if the load was distributed uniformly.

The first thing to notice is that none of the models scale perfectly, with a degradation that increases with the number of nodes. With 16 nodes, we observed times between 62% (**medium-grained**) and 180% (**fine-grained**) worst than an ideal scalability. In both **fine-grained** and **medium-grained**

Fig. 1: Data model influence on scalability.



workloads the *balanced* and *ideal* lines overlap almost perfectly. This suggests that request imbalance between nodes is the primary cause of lack of scalability in these two workloads.

A different case is **fine-grained**, as the *balanced* line diverges from the *ideal* one. With 16 nodes, compensating the imbalance does not significantly counterbalance the lack of performance, so that the *ideal* line is 180% lower than the *balanced* one. Since we ruled out the workload distribution as the cause of the performance degradation, we will later present a more detailed study. As confirmation of our hypothesis about the effect of work imbalance on the application scalability we analysed **coarse-grained**, which is the workload that splits the query into only one hundred keys, each of them containing ten thousand elements. Given the relatively small number of partitions, this policy shows the higher imbalance. As long as each key is stored at random in a server, the smaller is the ratio keys/nodes, the higher is the probability that some nodes will have more work than others.

Fig. 2: Operations per node vs. sub-query time.

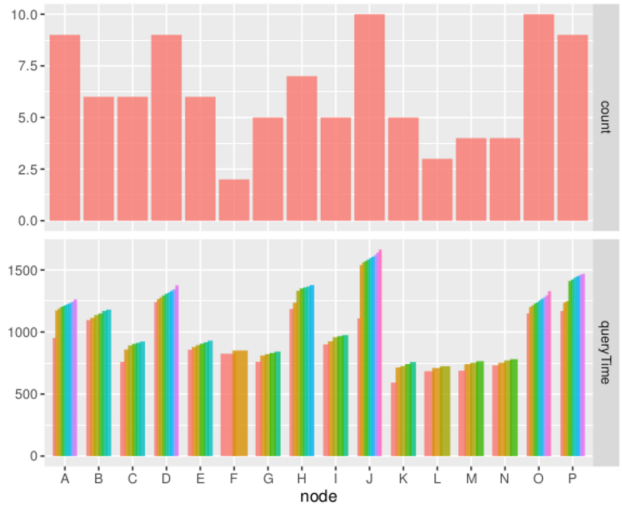
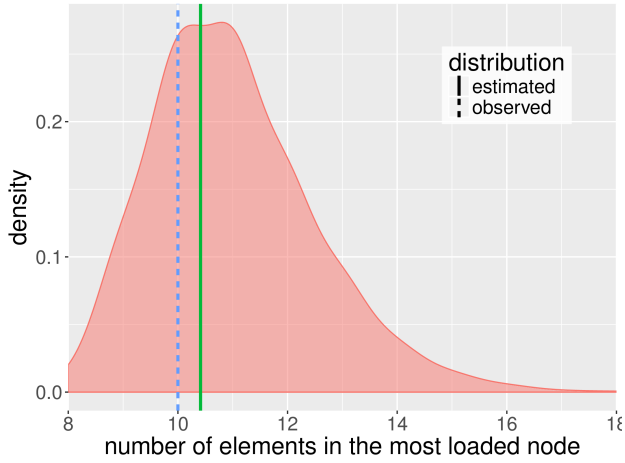


Figure 2 illustrates how the number of keys assigned to each node and thus, the number of operations that each node has to perform, affects the performance of the query. Figure 2 shows the results of the execution of **coarse-grained** workload on 16 nodes and it is composed of two charts: on the top a bar plot that shows the absolute number of requests each node served while the bottom graph shows the time required to complete each request on each node. All requests begin at the same time, so the query ends when the slower request completes.

Fig. 3: fine-grained: probability density with 16 nodes



At a first look, we can see how the two metrics are strongly correlated, but they are not identical. Indeed, we can see that the peaks in the number of operations match roughly the peaks in query time, but we cannot say the same for the lowest points. For example, node **G** completes 5 operations in almost the same time as node **F** finishes 2 of them when queries run in-memory on multiple cores CPU. Even though results show a considerable variance in all our tests, we observed that the node that served more requests is also the last to complete. Therefore, as long as the distributed operation completes when all nodes end their jobs, it is the slowest node to dictate the overall time. Summarising: **1)** the slowest node is the one that dominates the total time **2)** the slowest node is usually the one with the most queries **3)** the workload imbalance is proportional to the number of servers, and thus it explains why **coarse-grained** does not scale linearly. It also suggests that we can estimate this performance drawback using Formula 1.

Intuitively, it might seem wrong how the requests distributed in figure 2. We were demanding 100 keys on 16 nodes so — in a perfectly balanced case — the most loaded node would have to serve $\lceil \frac{100}{16} \rceil = 7$ operations while in our case it served 10, which is 43% more. Therefore, we might wonder if this test is just a highly unfortunate case.

Figure 3 shows the probability density relative to the number of requests the most loaded node has to serve. We generated the graph with brute-force by distributing at random 100 keys between 16 nodes and recording how many keys fell in the most loaded node. Figure 3 shows the recurrences: the two vertical bars represent the imbalance we observed in the experiment — the blue line — while the green one is the value predicted by Formula 1. Figure 3 shows that our observation

Fig. 4: Profile patterns: **medium-grained** and **fine-grained**

was not particularly unfortunate. On the contrary, in 60% of the cases we would have a more unbalanced scenario.

The central part of Figure 1 shows how **medium-grained** has lower imbalance as it uses ten times more rows. For example, with 16 nodes we reduced the overhead from 108% to only 44% compared to policy **coarse-grained**.

Model **fine-grained** shows a distinct behaviour. As we issue ten thousand queries at the same time we would expect an almost homogeneous workload distribution, but the system stops scaling with more than 8 nodes.

B. Definition of stages and identification of the bottlenecks

To investigate the origins of the unexpected lack of scalability on **fine-grained**, we have analysed the code and the prototype architecture to define the main stages in a request execution and then we recorded the times spent by requests in each phase. We identified the critical points of a request execution such as the ones where the system interacts with different software or hardware layer or with remote machines. We identified the following stages:
1) master-to-slaves: the time between the master issues a request and when one slave receives it. **2) in-queue**: the time a query waits in a slave before it is sent to the database. **3) in-cassandra**: the phase where the request is sent, processed and returned by the database. **4) slaves-to-master**: the time for the master to receive a partial result sent by a slave. This four-phase analysis turned out to be crucial for understanding the overall system performance: it allows pointing out which components of a distributed system limits the performance.

Figure 4 shows the duration of each request in each phase in the two executions: one with **medium-grained** — the bottom part — and **fine-grained** on the upper side. Each cluster of segments is a single phase in a specific node, while the length of the bars and their colour describe the time spent by a request in each phase. In such a way, short-lasting events which should be the norm in a well operating system are almost invisible thus highlighting eventual system congestions.

The bottom part shows **medium-grained**: we can see that the **master-to-slaves** phase lasts at most 300 ms, and also that Cassandra is not fast enough to satisfy all of the requests as quickly as they arrive. This is evident by looking at the **in-queue** phase where a lot of requests spend a considerable time waiting. Finally, the plot clearly shows how the workload distribution between Cassandra's nodes - as we saw before - is not uniform, and also that it directly influences performance. Indeed, this workload executes in less than 1 second, which is approximately the time required by the slower slave node — F — to complete all its requests in the **in-cassandra** phase. Hence, we can deduce that Cassandra is the weak link in the chain in this scenario, but also that, we can achieve a significant performance improvement with a more uniform work distribution. On the other hand, **fine-grained** presents an entirely different pattern. In this case, the master requires up to 1.5 seconds to finish sending all requests: an extensive time that leaves Cassandra idle most of the time. We can see it in both the **in-queue** and the **in-cassandra** phase. The first one is empty, meaning that all requests spent practically no time in queue. In the second one, we can see several empty - white - spots. These spots are the proof that Cassandra was processing requests faster than our system was able to issue them. In other words, a consistent portion of the whole execution time was spent idle while waiting for the requests to reach Cassandra. Here the major system bottleneck is the master node: it simply cannot send messages fast enough to keep Cassandra's nodes working at their full capabilities.

It is intrinsic of the master-slave architecture to be limited by the master's capabilities at some point. However, we wanted to understand which was the limiting factor. At first, we investigated the network: our cluster network has a star architecture, so each node is directly connected to a switch that dispatches the packages between nodes. With such an architecture, we suspected that the query saturated the outgoing connection of the master: yet we measured that the outbound traffic was only 7.5 MB split in 15 thousand packets. We measured that such a transmission takes 7ms in our cluster, way less than the 1.5s we observed.
Once excluded the network, we carried out a detailed profiling of the master application. We found out that we were hitting the CPU bottlenecks: we were consuming too many CPU cycles for each message. The CPU net cost of sending a message is the combination of several aspects influencing the actual implementation, such as the programming language and model, as well as the platform and libraries used. When building our prototype, we aimed for a good balance between speed of development and performance, so we used an Actor based library which runs on the Java Virtual Machine platform. Using profiling techniques, we found out that the there are two major contributors to the high message CPU cost. The first,and the most influential one, was the messages serialization. In Java, messages are objects and thus we have to transcode them into their binary representation — the serialization — to send them over the network. The default Java serialization implementation focuses more on flexibility than performance:

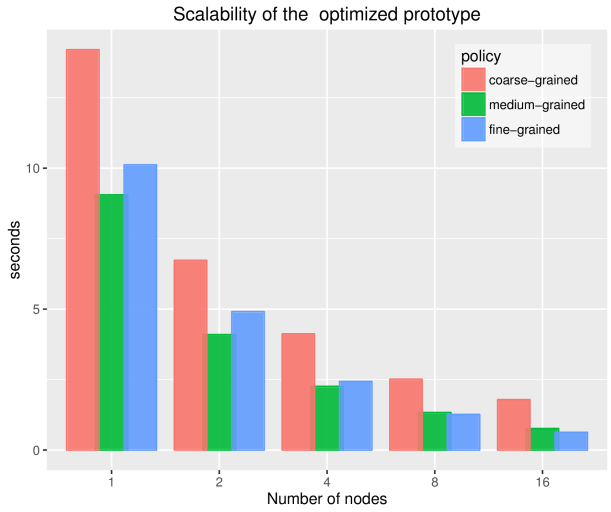


Fig. 5: Performance reducing bottlenecks

it allows serializing at runtime any object, at the cost of adding extra meta-data into each object's byte representation.
In our profiling, we measured that the serialization phase took about 400ms of the whole execution. Serialization is a well-known issue of many distributed platforms running in the JVM, and thus, there are several alternatives which aim to reduce both the amount of bytes size and the CPU cycles required. Among many, we choose Kryo [2], a library that allows to reduce consistently both the bytes and the CPU required, by explicitly declaring which classes need serialization.
The second optimization took place in our prototype code, where we observed that usually inexpensive operations, such as logging and integrity checks, were too costly at such a frequency so we had to work to reduce their effect.
With such optimizations, the master node of our distributed application had a great improvement, and the master node changed from sending ten thousand messages from 1.5s to just 192ms. Breaking it down to the single message, it is moving from 150 to 19 microseconds each, almost one order of magnitude of difference. Also, with a more efficient serialization, the amount of data transferred in the master to the slaves lowered to 900KB, which travels over the network in approx 700 microseconds. As a natural consequence of such an improvement, the results of our tests changed, especially for the **fine-grained** workload, which with the highest number of keys, was the more penalized by the message overhead. Figure 5 shows how performance changed.

The comparison of Figure 5 and Figure 1 shows how improving the master changed the performance profile of the various models: for instance now **fine-grained** shows almost linear scalability and it became the fastest workload when running on 4 nodes and more. Indeed, **fine-grained** is 12% slower than **medium-grained** on a single node for the higher overhead of issuing ten times more queries, but this handicap is soon compensated when the number of nodes increases. For example, with 8 nodes **medium-grained** has an imbalance of 16% while it is only 4% for **fine-grained**: a delta that nullifies the initial 12% handicap. It is interesting to see how, even in this simple case, a one-size-fit-all solution does not exists and depending on the number of nodes we might prefer one configuration rather than another.

VI. Performance Modelling

The last step in our methodology is to synthesize the results of the performance analysis into a mathematical model, that can guide designers to select the most suitable data organization for their applications. Thus, we created an approximated model for each of the different components that play a major role in a distributed application. So far, we observed three principal aspects: 1) the time the master needs to send all requests 2) the time that the slowest slave takes to finish 3) the time the master needs to fetch all results.

We saw that the request granularity - the number and the size of partitions in which the whole job is split - influences the performance of most of those aspects. Therefore, for each aspect, we created a regression model function of the number of partitions. We built these models upon observation recorded during tests run on our hardware and software stack. While the specific regression models may be realistic only for some hardware/software settings, the overall model and methodology can be applied to any system: it would simply require to run the same tests on the different hardware/software stack and create a new regression.

We saw in the previous experiments that the behaviour of the distributed system was influenced by the slowest of the high-level factor: the speed of the master sending requests, and the time required by the slowest slave to fulfil them. Therefore, at the highest level we can synthesize the model as:

$$max\{master_{speed}, slave_{slowest}, result_{fetching}\} \quad (2)$$

The $master_{speed}$ is the time the master node needs to send a single request to one slave. As we saw in Figure 4, the master can be the major bottleneck in performance. We have been able to speed-up the master by optimising its code implementation, but this is not possible in all situations. For example, if the master has to compute expensive operations to issue each request, or if there is a dependency between them. For instance, navigating through an index, the master needs to examine the content of each call before deciding which are the next elements to read. In both cases, it is beneficial to understand in a designing phase how much time the master can spend in such operations so that a developer can determine which are the lower and upper bounds when adopting a master-slave or peer-to-peer approach.

In this paper, we focused on the simpler case in which the master knows all the keys to visit from the beginning. Therefore its behaviour can be easily modelled as:

$$master_{speed} = keys * time_{msg} \quad (3)$$

In Formula 3, the keys are the number of partitions in the system, while the $time_{msg}$ represents the time spent, end to end, between the moment a request is sent and received.
We saw in Figure 2 that the slowest node is - unsurprisingly - the one that has to complete more requests. For this reason,

Fig. 6: Response time versus row size.

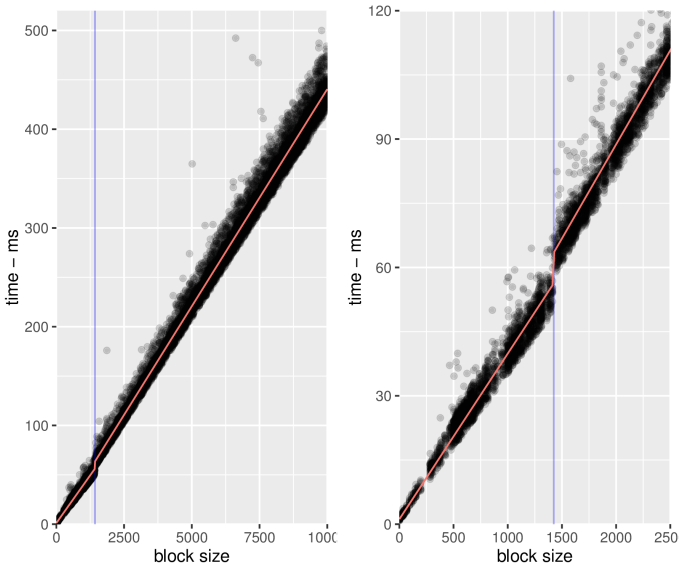


Fig. 7: Speed-up of parallel queries.

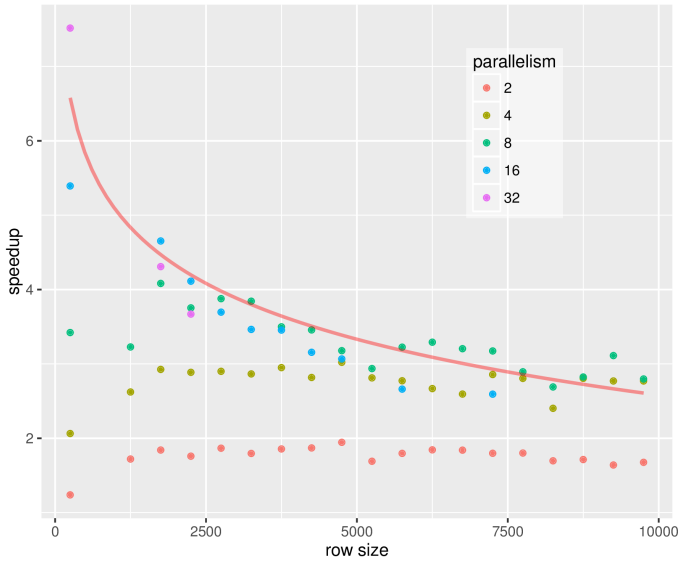
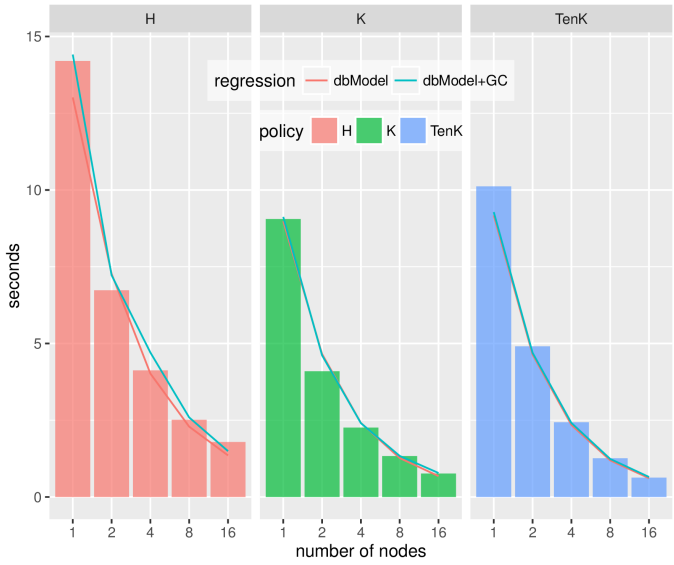


Fig. 8: Observed versus predicted time.



to simulate the slowest node, we have to consider how many operations the most loaded node will have to perform, and then estimate the time required by the database to fulfil them. Putting all together, the $slave_{slowest}$ model must take into account: 1) the workload distribution between nodes 2) the time required by the database to compute a request. These aspects result in the following formula:

$$slowest_{slave} = key_{max} * DB_{model} \tag{4}$$

We can deduct key_{max} from Formula 1, and therefore:

$$key_{max} = \frac{keys}{n_{slaves}} + \sqrt{\frac{keys * log(n_{slaves})}{n_{slaves}}} \tag{5}$$

As we discussed before, this formula is influenced by the number of nodes and the number of keys - partitions - in a way that promotes the increasing number of keys when we have more nodes. However, splitting a job into too many smaller partitions has a performance drawback causing overhead in the database.

a) Database model: The design of databases aims to reduce the latency of the average query by adopting greedy strategies; caches, indexes and bloom filters; that minimise the duration of most of the requests at the cost of introducing variance. For example, a miss in a cache of a false positive in a bloom filter can arbitrarily make a request orders of magnitude slower than average. Also, databases optimise the hardware resources by executing multiple operations at the same time, but this introduces a performance degradation caused by the interference accessing shared resources. For such reasons, we found out that the best way to model the database was first to study how the time required to serve a single request varies in relation to its size. In a second step, we estimated the performance degradation caused by the interface of concurrent requests.

To build the DB_{model}, we made a stratified sampling of the rows in our dataset so that we could get the same number of random samples for each range of row size. Then we execute several repetitions of our test reading in random order the rows we selected previously. Figure 6 shows how the query response time changes related to the query size.

Figure 6 shows two plots: the first describes the whole test execution, while the right plot is a close up that shows only the requests with up to 2500 elements. The close up shows an unusual pattern in the Cassandra response time: at around 1425 items per row there is a discontinuity point. We found out that a Cassandra internal parameter -*column_index_size_in_kb* - was the cause of such behaviour. As Cassandra uses two-level indexing, it maintains for each row a column index but, as it is not efficient to index each entry, it records only the first, and the last column each *column_index_size_in_kb*, so rows smaller than 64KB are not indexed. As it turned out, 1425

rows are approximately 64KB and thus the index overhead caused such inconsistency. For such a reason, in Formula 6, we opted for a piecewise function:

$$query_{time} = \begin{cases} 0.773 + 0.0439 * key_{size} & \text{if } key_{size} > 1425 \\ 1.163 + 0.0387 * key_{size} & \text{otherwise} \end{cases} \quad (6)$$

We repeated the tests allowing different numbers of concurrent requests. Increasing the parallelism has an adverse influence on the system variance and the performance of the single queries, but it increases the overall throughput. We observed that the increasing of the throughput is not constant, and it degrades in correlation with the row size. Cassandra seems to be able to perform at a higher parallelism with smaller queries and thus, to estimate the parallelism speed-up, we also have to consider which is the optimal parallelism for such a row size. To figure this out, we formulated another test: we created another stratified sampling of 20 groups, each of them with a row size range of 500 elements. For example, the first group has keys with sizes one to five hundred, the second from five hundred to one thousand, and so on up to ten thousand items per row. We queried all keys, for each group at a time, testing different levels of parallelism. Finally, we computed the maximum speed-up we achieved compared to the time required to get an element at a time.

Figure 7 shows the speed-up we achieve raising the parallelism correlated with the query size. The colour of the dot represents the level of parallelism. The graph shows two general trends. Firstly, the larger the queries are, the lower is the degree of parallelism that performs better: The small queries perform best with 32 requests at a time, the medium with 16 while the large ones with 8. Secondly, the red line in Figure 7 shows that we can get a good approximation of the parallelism speed up as long as it shows a logarithmic proportionality with the row size. Formula 7 expresses the relationship between query size and the speed-up obtainable by running queries in parallel. Figure 7 shows the maximum speed-up we could achieve by raising the parallelism correlated with the query size. As you can see, the two dimensions have a logarithmic relationship that we expressed into Formula 7.

$$parallelism_{model} = 12.562 - 1.084 * log(key_{size}) \quad (7)$$

Finally, putting together Formula 6 and Formula 7 we can define the DB_{model} as shown in Formula 8. The formulas allow modeling the database throughput in relation to the size of the key.

$$DB_{model} = \frac{query_{time}}{parallelism_{model}} \quad (8)$$

b) Validation.: We validated our model by comparing the estimated times with the one we recorded in our previous tests. In Figure 8 the bars show the times we measured while the two lines show the values we estimated with our model. The precision of the estimation is high, especially considering the high variance we observed in the tests. The only correction we had to carry out was for policy **coarse-grain** to compensate the overhead caused by the Java Garbage Collector, which our model does not contemplate as long as its influence is negligible in a properly configured system. Figure 8 also shows the line **dbModel+GC**, which adds the GC time into the model, notably increasing the model accuracy.

Fig. 9: Optimal number of rows and the predicted time.

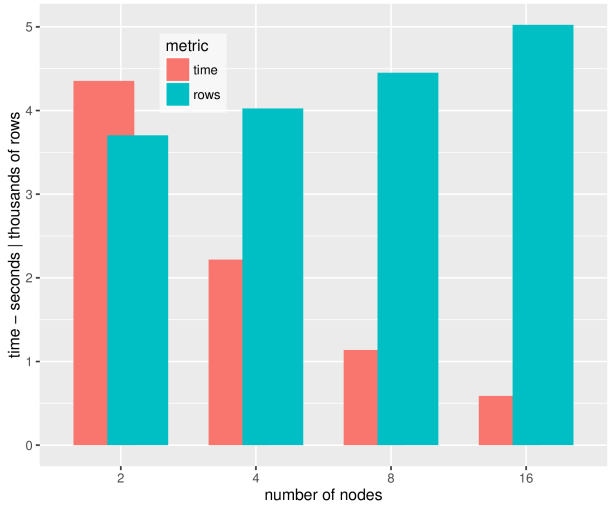
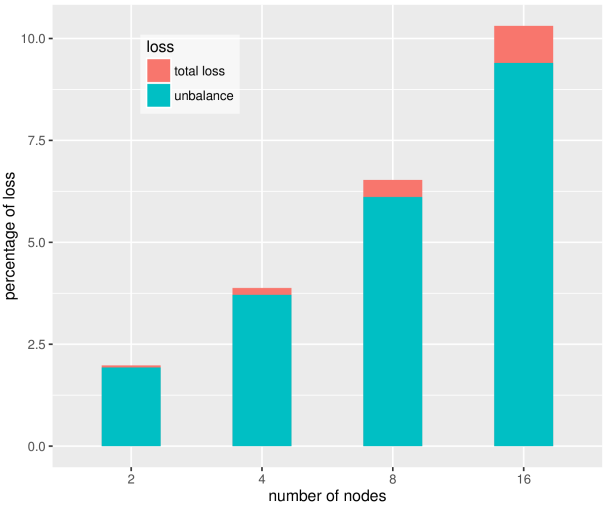


Fig. 10: Optimal settings versus ideal scalability.

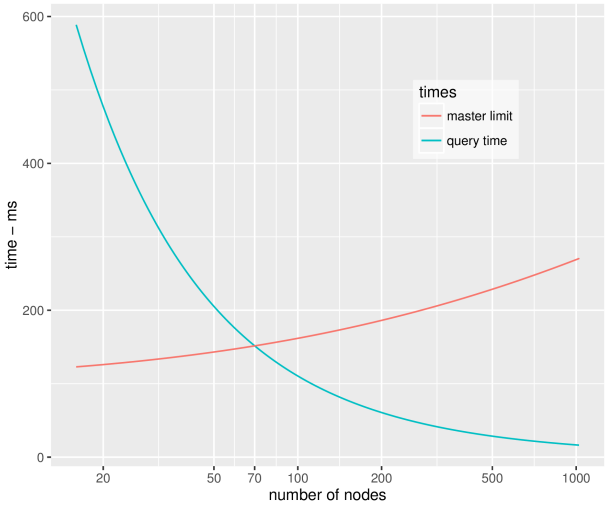


VII. Model Analysis

The flexibility of Formula 2 allows us to get useful insight on many different questions. For example, we can use an optimizer to find which would be the best number of rows for the query we run.

Figure 9 shows which would be the optimal time we could get on our system with the correct number of data partitions. It is interesting to see how the optimizer increases the number of rows when there are more nodes. Cassandra seems to perform at best if we split the one million elements into 3300 rows. However the optimizer is willing to sacrifice some of the database efficiency in exchange for a better work distribution when adding more nodes. It means that we have to mediate

Fig. 11: Load distribution limits for a single master.



between two conflicting aspects: the database efficiency and the workload distribution.

Figure 10 shows in percentage how much more time it takes the query to run on multiple nodes compared to an ideal linear scalability. We can see that even finding the optimal configuration parameters; we still have a consistent loss. For example, with 16 nodes the query requires 10% more of what would have been necessary with a distributed workload. Also, Figure 10 shows the difference between the total amount of loss and the fraction caused by the imbalance increase. This difference quantifies how much database efficiency the optimizer sacrificed to improve the performance. We can look at these results from another perspective. Let's suppose we are replicating the data in multiple nodes and that the master employs a replica selection algorithm so that it can ensure a balanced workload. For simplicity let's round up the numbers: the database performs optimally when issuing 4 thousand rows; the whole query takes 8 seconds on a single node, while the single request takes 11 milliseconds if we are issuing 16 queries in parallel per node. On a cluster of 32 nodes, the query should run in $\frac{8}{32} = 0.25$ seconds if the system scales perfectly. To do so, the algorithm should be able to issue at the very first moment $16 * 32 = 512$ requests, and then continue issuing the same number of requests every 11 milliseconds. However, as we saw before in our prototype, sending a message takes about 19 microseconds, and thus sending 512 of them takes $19 * 10^{-9} * 512 = 9.7ms$, leaving almost no time for the algorithm to run. As a consequence, the time left for the replica selection algorithm reduces so much that it is likely that with more than 32 nodes the master will start to be the major performance bottleneck, and the system stops to scale.

We touch similar limits when distributing the requests at random. Figure 11 shows how the query time decreases by adding nodes. It also demonstrates that with more than 70 servers, the master requires more time to send the requests than the time the database would need to serve them. This limit is higher if compared to the previous case with the replica selection algorithm, and it is so for two main reasons. Firstly, the master has a simpler logic, so we can hypothesize it issues all requests at the beginning of the query. Secondly, with a random distribution, the system does not scale perfectly, consequently leaving more time for the operations of the master.

VIII. Related work

In literature, we can find plenty of contributions regarding the problem of choosing in which node to store an object and how to ensure a balanced workload between nodes. These algorithms aim to select the best node that can store a particular object. In general, there are two main approaches:

1) Global mapping: A simple solution is to use a global master which keeps track of the position of each item's replicas in the cluster. One example is the Google File [7] System and its open source version - HDFS [4]. A master - the *NameNode*- balances the resource utilisation by deciding where to send each replica. This allows for a fast recovery from a slave failure at the cost of adding a single point of failure and a bottleneck in the system. Even if the master may have shadow replicas, as Konstantin [14] analysed, the *NameNode* memory demands grow with the cluster storage size and limits such architecture. Konstantin showed that 10'000 HDFS nodes with a single *NameNode* should scale up to 100 thousand readers and ten thousand writers. However, this estimation assumes a batch processing scenario with file chunks of 64MB, but to achieve low latency response time on indexed data, we have to access few KBs at a time and this dramatically narrows down the upper bound. As proof, McKusick and Quinlan [12] reported that GFS recently evolved to a more complex sharding design with multiple masters thus allowing lower response time and smaller file chunks.

2) Hashing: The alternative to a master is to use a deterministic algorithm to decide whether to place a replica. For instance, Distributed Hash Tables systems - such as Cassandra - use a pseudo-random hash function to place an object in one node of a cluster. An issue of DHT systems is how to achieve balance in load and storage utilisation. Its imbalance can be described with the single-choice balls-into-bins problem: we are throwing at random **m** balls into **n** bins, and we wonder how many balls the most loaded bin will have. This problem is the basis of Hashmaps analysis, where a collision means storing multiple items in the same memory cell and is an unwanted situation. The typical approach assumes that the number of cells is much larger than the number of elements. Only in 2006 Berenbrink et all [3] analysed the "*Heavy loaded case*", which interests the DHT systems as we assign many items to each server aiming to uniform distribution. The authors showed the imbalance decreases with the number of records while it increases with the number of nodes. Indeed, when $m \geq m \log m$ - which is the usual case for DHT databases - there is a node that receives $m/n + \mathcal{O}(\sqrt{\frac{m*\log n}{n}})$ items with a high probability. We can also formulate the formula in terms of ratio of imbalance between nodes : $p = \mathcal{O}\left(\sqrt{\frac{\log n*n}{m}}\right)$". On the other hand, Mitzenmacher [13] demonstrated one achieves

a better distribution with the "*power of two random choices*": instead of picking at random a single server, one chooses two of them, and selects the least loaded one. In this case the lower imbalance is to just $\mathcal{O}(\log \log n)$. However, using the multiple-choice algorithm we have to face implementation trade-off. For instance, we can store items uniformly between servers but at the price of penalising reads as the client cannot know which is the chosen server and thus has to question all replicas. Microsoft's Kinesis [11] follows this approach achieving a better load and storage balance by allowing the client to choose **r** replicas over **k** possibles servers. The drawback is that we have to question all k servers during a read operation and this might result in reducing k times the performance as databases system are often limited by the CPU. Alternately, we can store multiple copies of the same item so that the client picks the less loaded replica thus achieving a balanced distribution of read operations. However, it is costly to know the real-time load of each node, and the algorithm should maintain approximated load statistics which might not detect short living imbalances. Also, the second choice penalises caching system: if we ask an item twice from the same node, the second requests will be faster as it is served out of memory. On the contrary, spreading calls to different servers results in a higher page fault number and that might nullify the benefits of a more distributed workload. Indeed, the Cassandra driver selects a replica only if the original node is malfunctioning.

Designing a low latency system is critical to exploit in-memory operations while also distributing work uniformly across nodes. A typical pattern in HPC and Analytics is that the execution is limited by the fact that all nodes have to access to a relatively small partition of the data; a working set might rapidly change over time. We aim to exploit key-value data stores for such applications: our interest is in retrieving any particular subset of data in the minimum time. In this situation, the $m >> n$ hypothesis is not always valid thus resulting in performance degradation.

At the best of our knowledge, this is the first publication which aims to create a holistic analytical model of distributed application on key-value databases.

IX. Conclusions

In this paper, we presented a detailed profiling of which are the major aspects to consider for ensuring the scalability of a distributed application running on a key-value database. Also, we proposed an analytical model that enables developers to estimate the influence of each part of such a system inside the overall performance, so that it is easier to find the right balance for each application requirements. Also, our model allows finding the perfect number of partitions in which to split a distributed query so that it can find the right trade-off between database efficiency and workload distribution. Additionally, we discussed how by using our model we can estimate at which point either the master-slave approach or the replica selection algorithm can limit the performance. We think that our model is very useful to design both HPC and real-time big data applications. Also, we believe it can be employed when deciding which kind of hardware and technologies to use when creating a new cluster, as it is possible to use the formula to predict which hardware characteristics will influence performance the most.

As a future work, we aim to extend our model to consider hierarchical storage architectures such as the recently presented KNL Intel CPU [15]. These systems adopt multiple levels of storage, with an hierarchy between two kinds of ram memory, NVM, and SSD and rotational disks. We aim to extend the model to predict the time of serving requests out of each of these devices thus providing a precious tool for assisting the design of applications that exploit the benefits of such architectures.

Acknowledgement

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 720270 (HBP SGA1). It is also partially supported by grant SEV-2011-00067 of the Severo Ochoa Program awarded by the Spanish Government, the TIN2015-65316-P project, with funding from the Spanish Ministry of Economy and Competitivity, the European Union FEDER funds, and the SGR 2014-SGR-1051.

References

[1] Ganglia monitoring system. http://ganglia.info/. Accessed: 2016-07-29.
[2] Kryo: Java serialization and cloning. https://goo.gl/R8jacQ.
[3] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced Allocations: The Heavily Loaded Case. *SIAM Journal on Computing*, 2006.
[4] D Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 2007.
[5] Cesare Cugnasco, Yolanda Becerra, Jordi Torres, and Eduard Ayguadé. D8-tree: A de-normalized approach for multidimensional data analysis on key-value databases. ICDCN '16. ACM, 2016.
[6] Cesare Cugnasco, Roger Hernandez, Yolanda Becerra, Jordi Torres, and Eduard Ayguadé. Aeneas: A tool to enable applications to effectively use non-relational databases. In *Procedia Computer Science*. Elsevier.
[7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 2003.
[8] R. Hernandez, C. Cugnasco, Y. Becerra, J. Torres, and E Ayguade. Experiences of Using Cassandra for Molecular Dynamics Simulations. *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 288–295, 2015.
[9] A. Hospital, P. Andrio, C. Cugnasco, L. Codo, Y. Becerra, P.D. Dans, F. Battistini, J. Torres, R. Gni, M. Orozco, and J.Ll. Gelý. BIGNASim: A NoSQL database structure and analysis portal for nucleic acids simulation data. *Nucleic Acids Research*, 44(D1), 2016.
[10] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*.
[11] John MacCormick, Nicholas Murphy, Venugopalan Ramasubramanian, Udi Wieder, Junfeng Yang, and Lidong Zhou. Kinesis. *ACM Transactions on Storage*, 2009.
[12] Kirk McKusick and Sean Quinlan. GFS: evolution on fast-forward. *Communications of the ACM*, 2010.
[13] Aw Richa. The power of two random choices: A survey of techniques and results. *Combinatorial ...*, 2001.
[14] Konstantin V Shvachko. Hdfs scalability: The limits to growth. *; login:: the magazine of USENIX & SAGE*, 2010.
[15] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, Mar 2016.
[16] Mariano Vazquez, Guillaume Houzeaux, Seid Koric, Antoni Artigues, Jazmin Aguado-Sierra, Ruth Aris, Daniel Mira, Calmet Hadrien, Fernando Cucchietti, Herbert Owen, Ahmed Taha, and Jose Maria Cela. Alya: Towards Exascale for Engineering Simulation Codes. *Supercomputing*, 2014.