Experiences of using Cassandra for molecular dynamics simulations

Roger Hernandez, Cesare Cugnasco
Barcelona Supercomputing Center
Barcelona, Spain
Email: {roger.hernandez, cesare.cugnasco}@bsc.es

Yolanda Becerra, Jordi Torres, Eduard Ayguadé
Universitat Politècnica de Catalunya - Barcelona Tech.
Barcelona supercomputing Center
Barcelona, Spain
{yolandab, torres, eduard}@ac.upc.edu

Abstract—In response to the requirements of applications that work with large amounts of data, various NoSQL databases have appeared to deal specifically with these challenges. These systems have become popular in environments such as data analytics and OLTP, however these are not the only data-intensive applications that can benefit from these databases. In the life sciences domain, there are many applications that still use flat files as a medium to store data, and they see themselves very limited in terms of scalability and performance, as well as code complexity. We present an analysis on the viability of using these databases for applications with data demands that differ in some of the characteristics from what these systems were originally designed for. By using these databases, we can also observe that the design of the data model, queries and other configuration parameters can have a considerable impact on performance, thus we present examples of different data and system configurations to analyse their effects on performance. With the executions that are presented in this paper we can see performance gaps of a factor of up to almost 5 between using different models, queries and configuration parameters.

Keywords—Big data; Non-relational databases; Cassandra; Performance evaluation; Life Science applications

I. INTRODUCTION

Every day there are more data-dependent applications whose performance is heavily affected by input and output operations. Requirements to read and write this big amount of data has brought up limitations in traditional storage systems that cannot be overcome without defining new paradigms and mechanisms. Non-relational databases ([1], [2], [3], [4]) arise as a new proposal to solve these limitations, in fact they provide great features aimed at the challenges posed by the newly appeared requirements [5].

The most typical user base of these data store systems is that which is working with data analytics and Online Transaction Processing applications (OLTP), which favour this recent technology as a solution to the data management problems they face. Data involved in this kind of applications is usually highly dynamic and composed of a lot of small records without a fixed structure, which are accessed simultaneously by many different applications and clients from the cloud. Therefore, these data store systems have been designed in order to fit the requirements of these applications, thus they provide high scalability, the ability to favour many small queries simultaneously, and allow the dynamic insertion of unstructured records.

However, there are other types of applications with different access patterns that could also make use of the features offered by non-relational databases. For example, the scalability and accessibility they provide, and their ability to store and to allow large parallel accesses to a big amount of information in the cloud.

We have observed in our supercomputing centre the case of life science applications. Every day, the life sciences community is tackling with data challenges that keep growing as technology and ambitions advance [6]. Improvements to sampling methods and precision, increasingly produce more and larger data. This situation poses a technical barrier for many fields in the scientific community whose main speciality does not lie in data management.

Life science applications usually keep data in flat files that are typically stored in centralized file systems that work on a client-server architecture [7]. However, at the current data growth rate, this technology is not able to keep up with the resource demands and scalability requirements to store all this data. Scalability is not the only factor that scientists have to struggle with, there are plenty of other drawbacks bound to flat files. For example, the lack of indexing complicates the required code to select specific parts of the dataset. In order to avoid productivity penalties, life science applications have taken the approach of reading whole samples into memory when needing to manipulate data, which allows them to select the exact subset of data required for a given analysis. This process entails disk and network bandwidth squandering from reading unnecessary data, which negatively affects application performance. Some efforts have been devoted to provide life science applications with indexing support in order to overcome this lack of productivity by trying to reduce resource usage, for example, using the HDF5 data model [8] or relational databases [9], [10]. However, both of these solutions were not designed to support the increasing data requirements of current installations. Thus, tuning and configuring them to scale gets incredibly challenging as data sizes increase ([11], [12], [13], [14], [15]) and easily becomes the limiting factor of performance.

Non-relational databases are good candidates in order
to offer a highly productive environment with support to indexed accesses using scalable cloud storage [16] because they were designed specifically for this kind of environments. However, even with these, programmers face new challenges which are, on one hand that of designing a suitable data model to organize the information, and on the other hand, choosing the right database parameters in order to get an efficient execution environment.

In this paper, we analyze the performance of a non-relational database when managing data generated by molecular dynamic simulations, which is a widely common source for many life science applications. We work on real data samples generated by the life sciences department of our supercomputing centre, and we analyze how different configurations and data models influence on query performance. For all these executions we chose Apache Cassandra [2] as a representative example of a non-relational database. We show that by using an adequate data model and configuration parameters it is possible to get a speedup of up to 9 times when comparing it to using flat files. But we also show that choosing the right scenario is not straightforward, and using an inadequate data model or configuration parameter can divide query performance by a factor of about 4.7. This result suggests that on one hand, migrating this type of applications to non-relational databases is worth it in terms of performance, and on the other hand, that users should be provided with tools and mechanisms to alleviate them from the task of selecting data models and configuration parameters.

The rest of the paper is organized as follows. Section II describes the background and environment in which this work was done. Section III and Section IV describe the evaluation scope of this paper and the benchmark characterisation that we use, and in Section V we show the result of our experiments. Finally, Section VI and Section VII contain related work and conclusions respectively.

II. BACKGROUND

Apache Cassandra is a distributed and highly scalable key-value database backed up by the DataStax company and a strong user community. Cassandra implements a non-centralized architecture based on peer-to-peer communication, in which all nodes of the cluster can receive and serve queries. Each node is assigned a token, and a partitioning function uses this token to decide the data distribution among the ring-shaped nodes. This partitioning algorithm is configurable by the system administrator, and the Murmur3Partitioner is the default option.

Column Families are the equivalent of a table in an SQL environment, and data is stored in them by rows that are identified by a user chosen key value. In each row, users can add multiple columns also identified by a chosen column name. Cassandra stores data by rows, in a manner where one node is responsible for hosting a specific row, this node is chosen through the partitioning algorithm and the choice depends on the row key and the node token. In a setup with N replicas, it is also necessary to choose the nodes that will hold each replica. The default replication algorithm selects the N-1 nodes that come before the target node in the ring’s order to store those replicas.

When using replication (replication factor ≥ 1), Cassandra offers various features and consistency options. Users can query Cassandra with a specific consistency level, choosing between consistency and performance as they see fit. For non-quorum reads Cassandra will automatically attempt background read repairs in order to keep consistency among the replicas, and the frequency of these checks is determined by the read_repair_chance parameter (i.e. setting it to 0.1 would perform repairs for 10% for the reads).

When receiving a query, if the entry point has to choose which node serves the request between multiple options, it will choose the node with the best recent response time. Excessive switching between responding replicas can be detrimental for response time because it defeats the purpose of caching. In order to avoid small changes in the difference of performance to trigger a change of response candidate, the dynamic_snitch_badness_threshold can be tuned. For example, if it set to a value of 0.1 the selected responding replica will only be replaced in the event of the other replica being at least 10% faster. For instance, setting this option to a high enough value would make a replicated Cassandra cluster to behave as a cluster without any other replicas besides the original.

III. SCOPE OF THE ANALYSIS AND SCENARIOS

There are many facts, features and choices to have into account when querying data in Cassandra, and considering them all at once will result in a very complex scenario when trying to determine which are the individual factors impacting the results, and how much they do. For this reason, we have performed the same set of experiments on four incremental scenarios that comprise particular characteristics of Cassandra and other non-relational databases. These will incrementally combine the previous ones as they are executed and analysed. These experiments take as a model a life sciences application, however it will only take into consideration the sections that manage and query data, which formerly used flat files.

The first scenario we have defined sets the replication factor to 1 and uses only one client with a single entry point to the Cassandra cluster. This is an unrealistic scenario that simplifies a non-relational database to meaninglessness in a common production environment. The system will not be able to exploit the potential benefits of using more than one replica, reducing load balancing capabilities and execution performance. It is for this reason that this experiment will serve us as a baseline or starting point that will draw some conclusions and allow us to let the environment grow with
further complexity. More precisely, in this scenario, we can evaluate the impact on performance different queries may have on different data models, thus allowing us to quantify the relevance that a data model can have on data retrieval performance.

The second scenario sets the replication factor to 3 which, besides reliability, offers load balancing. In this scenario, we evaluate the effects of replication on performance from the application point of view by changing the dynamic snitch value.

The third scenario uses a multithreaded client which can fully exploit the potential parallelism of a Cassandra cluster. This application performs multiple concurrent queries each of them to a different Cassandra entry point. Therefore, we exploit parallelism at two different levels: the first being at an application level, where various concurrent queries are performed on different nodes, and the second one at a Cassandra node level, in which, given a query, the database determines the target nodes responsible for each row and each request is forwarded to them in parallel. In this scenario we evaluate the benefits of exploiting this parallelism and its effects on the database.

Finally, the fourth scenario will evaluate the benefits the client can gain from possessing knowledge of the cluster topology in order to try and attempt to take advantage of it.

IV. DATA CHARACTERIZATION

This section will describe the data sample that is used for all the experiments through this paper. The Life Sciences department in our centre provided us with real data generated by molecular dynamic simulations that they are using for drug discovery applications. Notice that we have selected a small workload to speed the execution of the experiments up. However, the conclusions of our experiment are not affected by the size of this workload because even if the data sample could fit into memory completely, all queries access just once to each element, thus there are no benefits from caching.

A. Dataset

The dataset is a molecular dynamics trajectory aimed to study drug behaviour, consisting of a series or snapshots (also known as frames) of atom positions in space, therefore each atom will contain a set of three coordinates describing its location in space. The trajectory comprises about 10.000 snapshots of approximately 32.000 atoms that are grouped into atom clusters defined as residues (e.g. residue 34 contains atoms 312-320). Those always have the same atoms among the different frames and each atom can only be placed in one residue, there are about 10.000 residues in total.

B. Queries

There are multiple common queries to perform in a trajectory, these can be split into two major interests: time and residues. The queries that are analysed in the experiments section will be the following three:

- **Read All**: The whole trajectory will be requested, comprising all the frames and residues (.316.700.000 atom positions)
- **Frame set**: All the residues will be queried for a specified number of frames (.9.600.000 atom positions)
- **Residue set**: For all the frames, just a pocket (residue subset that define the drug binding area of the trajectory) will be retrieved (.11.720.000 atom positions)

C. Models

Two models are presented from the given dataset: in the first one, *FrameXResidue* or *FXR*, all the data is placed in the same Column Family, where each row represents a different frame and each column a different residue, thus each row/column combination contains a residue (set of atoms) in a specific frame of time. The second model is called *ResidueXFrame* or *RXF* and is a reversal of the first case, in which we have residue indexes as rows, and frame indexes as columns, again with a residue element for each combination of these. To implement these models, it was required to define and integrate a new data type into Cassandra, that is the *ResidueType* which will consist of an array of floats representing the different positions of the atoms inside this residue.

V. EXPERIMENTS

In this section, we show the results of our experiments averages, which are characterized by the data models, dataset and queries described in Section IV and by the scenarios described in Section III.

The data are deployed on an Apache Cassandra database, installed on 16 homogeneous nodes each with an Intel Xeon L5630 @2.13GHz CPU, 24Gb RAM, an SSD at 256MB/s bandwidth for storage, and interconnected in a 2Gbps network. All the queries are requested from a node of equal characteristics and in the same network which is not participating in the Cassandra cluster.

The Cassandra version we use is 1.2.2, and unless otherwise stated, the values of the configuration parameters are the default ones, with *SimpleSnitch* and *Murmur3Partitioner*. For all queries and in every scenario we use the most relaxed consistency level, which is **ONE**, and the feature to do background checks of consistency among replicas is disabled (setting to 0 the parameter read_repair_chance). These checks are not necessary for our scenarios because data is read-only and thus it is not possible to run into consistency problems.

A Java client that uses the Hector API for Cassandra has been implemented to interact with the database. This client can be configured to divide a query into threads that will request data from different entry points, therefore, with 16 nodes, up to 16 threads are supported. When the Java client
is started, the maximum memory allocation value is set to 8GB (-Xmx8g).

A. First scenario: Data model affinity

In this subsection, we show the results obtained from executing the three queries we are considering (read all, frame set and residue set) on the two data models we have designed (FXR and RXF) for the first scenario, with a replication factor of 1 and a single entry point.

One of the goals of this experiment is to make a comparison of the current commonly used methodology to store data in life science applications with our proposal, that is to compare flat files on a centralised file system, to a non-relational database. The original data consists of two AMBER [17] formatted files: a trajectory file of 7.2 GB which contains all the coordinates, and a topology file with the description and metadata of the trajectory (information such as atom types, number of atoms, residues and their components, etc.). The size of the topology file is both small and irrelevant, this is because besides providing additional information, the topology file served to provide structure to the trajectory file, thus it is only used as support when loading the coordinates to Cassandra.

Figure 1 shows the speedup comparing the query response time on Cassandra with that of using flat files. Reading the whole trajectory from the SSD takes 40.6 seconds in average. As we can observe, the read all query is faster when using flat files than when using the database. This is something that was expected because this query retrieves all the trajectory data, and since the trajectory of this experiment was relatively small, it fits in a local file, and the file system remains unstressed. Notice however, that as both the size of the trajectory data and the number of concurrent users increases, file system performance will decrease, and opportunities will arise to exploit the characteristics of a distributed data store system. Knowing this, distributed data store systems are expected to overcome barriers and performance of centralized file systems. Moreover, considering a data store system that aims to export a high amount of large trajectories, centralized file systems will not be an option.

However, when querying partial data through both the frame set query or the residue set query, the speedup of Cassandra over using flat files ranges from 3.5 to 5 times, from the worst to the best scenarios respectively. The main cause of this benefit comes from the fact that databases offer a fast and efficient method to read just the amount of data required by the query, whilst with plain files the methodology is to read all the data into memory, to select the required portion of data afterwards.

The same Figure 1 allows us to compare the impact on performance when using different data models. Response time is better when querying a frame set from the FXR model rather than from the RXF one, with the difference being about 8%. Alternatively, when querying for a residue set, the better performing model is RXF, with a difference in performance of about 28%. That is, performance is better when the selection attribute is based on row keys rather than column names, and the explanation for this is related to the implementation of the read path in Cassandra. The assignment unit to nodes in Cassandra is the row, therefore a row is never divided across multiple nodes. Also, the row is usually the disk request unit: if the row size is below a threshold Cassandra never performs partial reads of a row, therefore regardless of the amount of columns involved in a request, Cassandra reads all the columns from the row that is involved. For example, let us consider the frame set query, which gets the information for all residues on a particular set of frames. If it is executed on FXR, it will request all the columns from a set of rows. So this combination of query and data model involves requesting a few rows to each node and reading just the required data from disk. However, if instead it is executed on the RXF model, it will query a set of columns from every row. Thus, for the frame set query, both the number of requests received by a node and the amount of data requested from disk is higher with this model than with FXR. For the read all query, performance is equivalent in both models. This is due to the absence of the read path influence for the read all query since it is dependent on the model (for FXR it reads by frame, and for RXF it reads by residue), and because all the data is being fully read in both cases.

The results obtained in this section quantify the influence that data models have on query performance, and even though we could see that any model we chose was already superior to flat files, they additionally serve as a lead to decide which data organization, in combination with a given query, should be used in order to achieve optimal query performance. The three described queries are a requirement of the application from the presented domain, and both models showed affinities for different scenarios, thus in order to maximise performance both data organizations should be kept in the database.
B. Second Scenario: Influence of the replication factor

The second scenario sets the replication factor to 3, which is the default value in Cassandra. The purpose of this scenario is to check whether data replication benefits query performance or not. The rationale for this expectation is that having more replicas should increase the opportunities to improve data locality in the entry point nodes, which should decrease network traffic in the cluster, and alleviate the influence of slow or non-responsive nodes in the cluster by providing alternative nodes with the same data. Cassandra implements this optimization by dynamically deciding which of the nodes holding a replica of the requested data is responsible for answering the request according to its recent response time. With a read consistency level of \textit{ONE} the responsibility to answer falls on a single node, which is decided by the Cassandra snitch without considering other replicas. If the read operation latency of the chosen node decreases by an amount higher than a configurable threshold, then Cassandra transfers this responsibility to one of the other replicas holding the data.

Notice that we are executing our experiments in a homogeneous cluster and in an isolated environment. Thus, the only factors that could cause alterations in the assignment of responsibilities for data requests are either unbalanced situations, or data locality.

Figure 2 shows the response time for the \textit{read all} query when executed on both data models (\textit{FXR} and \textit{RXF}) with just 1 replica, in contrast of the execution with 3 replicas when both disabling and enabling the dynamic snitch feature. As it was expected, we can observe that increasing the number of replicas without activating the dynamic snitch has no effect on performance. The less intuitive result is that when enabling the dynamic snitch, query performance remains unaffected. This happens because even when Cassandra is exploiting data locality at the entry point, the data to be gathered from other nodes is so large that it is not enough to compensate for the locality of the entry point.

In this case, we can observe that enabling the dynamic snitch feature causes performance to increase slightly in all cases. This happens because these queries are performing a lot fewer requests to the cluster and thus the drawbacks of an unbalanced system are not enough to neutralize the advantages of data locality.

C. Third Scenario: Influence of parallelism

In order to attempt further improvements on performance, we decided to exploit yet one of the other advantages of having a database over flat files, which is the possibility of having parallel queries. Therefore, we repeat the previous scenarios by splitting the queries into 4, 8 and 16 query groups, each of them executed on a different entry point.

In order to simplify the display of results we have chosen to show figures only for the best performing cases, which are the pairs of model-query that showed an affinity in the previous scenarios. These are \textit{read all} and \textit{frame set} on \textit{FXR}, and \textit{residue set} on \textit{RXF} which is analogous to \textit{frame set} on \textit{FXR} pair. In regard to the remaining queries, they are not presented in figures but have a very similar behaviour.

First of all, by looking at Figure 4, it is shown that having dynamic snitching activated when reading all the data does not bring any changes in response time when comparing to the execution with dynamic snitching disabled. Figure 5 shows the results for the partial query of \textit{frame set} on \textit{FXR} (graphic for the \textit{residue set} on \textit{RXF} case was omitted because it behaves in the same exact manner) and we can see how enabling dynamic snitch is beneficial in every case, although the improvement is very small and it diminishes its effectiveness as more entry points are added.

Furthermore, we see how, for all the queries, performance drastically improves when querying to four simultaneous entry points instead of one, and how with a larger number of entry points the improvement is continuous, finally stalling at roughly 40% of the original version with just one single entry point for both the \textit{read all} and \textit{frame set} queries, and at about 55% for the \textit{frame set} query.
In order to find the factor that is stalling performance when increasing the number of entry points, we proceeded by monitoring the network, and we could observe that for 8 or more entry points network usage topped its effective bandwidth limit at about 120MB/s, becoming the bottleneck in this environment.

Willing to exploit the characteristics of the network and to avoid its bottleneck we have distributed the client application between two querying nodes, such as that one will be querying 8 entry points concurrently, while the other queries the remaining 8, thus duplicating the receiving bandwidth by using two connection links. We refer to this query as 8+8. We have evaluated this distributed version of the client and we have observed that network usage of each of the two links used by the 8+8 query, was exactly the same as with the queries that used 8 and 16 entry points, limited by the maximum bandwidth of 120MB/s. In Figure 6, we can see how the response time of this distributed version of the client compares with the previous evaluated versions with 8 and 16 entry points. We can observe that the 8+8 query took exactly half the time of the queries involving 16 entry points. Since the network is still the limiting factor, response time can be further lowered by either upgrading the network or by distributing even more the application among different clients.

D. Fourth Scenario: Influence of token-aware queries

In order to see if we can further improve performance we upgraded the client to be token-aware. For a token-aware client, we understand that it will have been provided with information regarding the cluster topology, more precisely the token distribution, or in other words which are the tokens that determine how data is sharded. With just this information, the client can pinpoint which node has which rows before connecting to the cluster, allowing it to group queries with the same token assignation, to be executed on their respective nodes afterwards, resulting in a scenario where all read operations will be served locally by the entry points.

We have executed the non-distributed version of the client for the read all query, using 16 parallel entry points, with 3 replicas using token-aware query assignments. Notice that, as we have shown in Section V-C, network bandwidth is the bottleneck of this configuration and thus, we do not expect any improvement on the final response time of the query. However, if this token-aware approach is able to improve the Cassandra query time, that is the total time it requires to prepare the requested data, it means that a distributed client that further exploited network topology could benefit from enhancing data locality in the entry points.

Figure 7 compares this token-aware execution with the execution that uses 16 parallel entry points and just one replica. It shows both the query and the final response time, that includes query time and time required to deliver the data to the client. Cassandra takes up to roughly 6 times more serving a request that is not token-aware in a non-replicated environment. This result encourages the use of token-aware clients in environments where network bandwidth is not the bottleneck.

E. Experiment conclusions

Overall, the results have proven to be satisfactory. When querying for the whole data, the flat file read was faster but that is to be expected since it is the most, and probably
only, profitable case when reading a file. However, when partially reading it, which is a usual way to query this data, databases have clear benefits when put against flat files, given that these still require a full document read. In this case, we could see the best improvement to be of about 5 times faster than without using a database. Furthermore, by querying Cassandra with different threads this performance was boosted up to 9 times faster than the original, a practice that cannot be achieved with the features flat files offer.

Additionally we could see that having multiple replicas does not improve read performance with this type of dataset which, in this scenario, reduces their usefulness to data consistency, availability and fault tolerance.

We also were able to observe performance gains when reading data with certain models and queries, thus confirming the existence of model-query affinities. The results show over 25% of faster read times when comparing a query to the same redesigned one with better affinity. In an environment where there is a requirement for different queries, the existence of a single model that provides optimality to every query is unlikely. For this reason, decoupling data organization from query design would increase productivity of users in the task of producing well-performing code.

VI. RELATED WORK

Big Data has been a growing concern for the life sciences domain [6], and many approaches have been taken, such as moving to RDBMS, non-relational databases, cloud solutions [18], or even making a database specifically for that [19]. Even though non-relational databases seem to be the preferred choice when dealing with big data, there are claims of Oracle owning 85% of the bioinformatics database market in relational systems [20], which almost nullifies the presence of non-relational database solutions, in fact we have found very few examples of attempts to leverage bioinformatics with such systems [19], and most of them are very, if not completely, undocumented. This large presence of relational databases to store bioinformatics data is also backed up by different major projects [21], [22], [23], [24], [25], [26] that attempt to unite or gather massive amounts of data and repositories. Furthermore, the assembled repositories are individually using relational approaches as well [27], [28].

We have found no information or analysis done on the impacts on performance when using different correct models and queries. However, as can be seen in the original Cassandra and DataStax documentation files [29], [30], [31], it is agreed that performance can be affected by the chosen data schema, or when changing configuration parameters that are common to distributed systems, such as the replication factor.

VII. CONCLUSIONS

Using flat files as storage method for bioinformatics data is an approach with many drawbacks that worsen as science advances. Relational database solutions are a common practice when a repository is created or upgraded, and are a more modern and friendly approach to tackle most requirements, but data are growing in such a challenging manner that even relational databases are often falling short to meet the increasing needs. With Apache Cassandra as an example, we proposed non-relational databases as a solution that integrates both, generic database advantages over flat files, and scalability, which is an issue RDBMS fail to overcome efficiently. Key lookups are incredibly efficient thanks to the distributed and well-performing nature of sharding.

The combination of the obtained results, the multiple configuration parameters Cassandra offers, the different cluster compositions, altogether with some opportunity gaps we spotted in the various configurations and features, open many doors and provide us with many chances and methods to increase performance both within these applications, and within the database system by taking further advantage of more mindful modelling, as well as leveraging replica use further than just for reliability and availability ends.

As part of our future work, we plan to provide programmers with an environment to automatize data organization and queries, hence performance and their productivity should be increased. Additionally, it would be interesting to broaden the analysis of this paper to other BigTable databases, column stores or any other NoSQL solutions overall.

ACKNOWLEDGMENT

We would like to thank Ramon Goñi and Pau Andrio for their help with understanding the characterisation of life science workloads, as well as David Carrera for his support with the cluster configuration. This work is partially supported by the grant SEV-2011-00067 of Severo Ochoa Program awarded by the Spanish Government, the TIN2012-34557 project, with funding from the Spanish Ministry of Economy and Competitity, the European Union FEDER funds, and the SGR 2014-SGR-1051.
REFERENCES


