An Object Mapping for the Cassandra Distributed Database*

Pedro Gomes, José Pereira, Rui Oliveira

Universidade do Minho

Abstract. New data management systems aimed at huge amounts of data, such as those pioneered by Google, Amazon, and Yahoo, have recently motivated a profound revolution in database management. These systems are characterized by simple data and query models and by their relaxed consistency, that contrasts with the ubiquitous and widely known relational and ACID model. These changes, in combination with the absence of a high level language for query and manipulation, make it harder for developers to port applications or to leverage existing know-how. In fact, the major proponents of such technologies are precisely those companies that can employ highly skilled developers.

This paper bridges this gap between the common programmer and the increasingly popular Apache Cassandra distributed database by providing an object mapping for the Java language in the style of object-relational mappers that are commonly used with SQL databases. In detail, we describe how the object-relational paradigm can be mapped to the new data model and experimentally evaluate the result with a combination of the standard TPC-W benchmark and a representative application from the telecom industry.

Keywords: Non Relational Databases, Cassandra, Object Mapping

1 Introduction and Motivation

The current trend in a wide variety of applications, ranging from the Web and social networking to telecommunications, is to collect and process very large and fast growing amounts of information leading to a common set of problems known collectively as "Big Data". This reality caused a shift away from relational database management systems (RDBMS), a mainstay for more than 40 years, to multiple new data management methods and tools. In fact, when scaling traditional relational database management systems the user can only chose between the expensive and limited vertical scale-up or the non-straightforward horizontal scale-out, where many of the niceties that make these solutions valuable like strong consistency guarantees and transactional support are rendered unusable [4]. To face these challenges, it is now widely accepted in industry and academia that a new paradigm is needed, leading to the NoSQL movement. This moniker encompasses a large variety of data and consistency models, programming interfaces, and target areas from small web development oriented databases to large

* Partially funded by FCT through project ReD (PDTC/EIA-EIA/109044/2008) and by PT Inovação.

scale solutions. In this paper we focus in the later, that include Amazon's Dynamo [2], Google's BigTable [1], and Facebook's Cassandra [5] that is today an Apache open source project.

These databases were designed from scratch to scale horizontally with the help of mechanisms such as consistent hashing and quorum-based operations [2] where new nodes can be added and removed in a much more straightforward way than with a typical sharded relational database. Highly available hardware and its associated costs can thus be avoided, as one can create a cluster with commodity hardware that can easily scale up and down and exhibit high reliability. As consistency, availability and partition tolerance cannot be achieved at the same time [3], these databases are built on the assumption that a trade-off must be accepted. They thus allow some data inconsistency to guarantee availability at all moments, mainly, and by offering the user the choice of defining the number of nodes involved in each operation. This ensures that the system stays on-line even when some nodes fail or become isolated, but is a considerable challenge to porting existing applications.

A second challenge is the data model presented to developers. The multi-level sparse map model introduced in BigTable and then adopted by HBase and Cassandra has been been the preferred so far, as it has no fixed columns and allows the storage of the data in ways that enable a limited set of queries to run very efficiently. But this model, with its multiple levels of storage and associated nuances, represents a major mind shift for many programmers. Moreover, typical application programming interface (API) don't include a high level languages for data extraction like SQL. In fact these interfaces can sometimes cut the language impendence and speed up the development, but they can also be painful to handle as they forece the user to master a set of new concepts and routines and to deal explicitly with consistency. For instance, by having to provide time stamps to order requests.

In this paper we address this second challenge by introducing a Java object mapping for Cassandra, giving the programmers a tool that allows them to use this database much in the same way they have been using relational databases by means of object-relational mappers like Hibernate[1] and Datanucleus[2]. This should provide a bridge between the familiar object oriented paradigm, including different relation types, and the Cassandra data model, allowing the developer to abstract the persistence layer and focus on the application code.

This has been tried by several small projects that appeared previous to this work like the HelenaORM [3] or the OCM[4]. However, these projects still lacked several features like relation mapping and did not support a standard accepted interface that is a vital point in this work. A standard interface like JPA or JDO is in fact many times important for integration, and was been already implemented in other large scale databases like Hbase [5].

[1] http://www.hibernate.org
[2] www.datanucleus.org/
[3] github.com/marcust/HelenaORM
[4] github.com/charliem/OCM
[5] www.datanucleus.org/products/accessplatform_2_1/hbase/support.html

To qualitatively evaluate our proposal we first ported the well known TPC-W database benchmark [6] to Cassandra with and without the object mapper. This industry standard benchmark highlights difficulties in using Cassandra directly and then to what extent these difficulties are masked by the object mapping. In a second phase we then adapted a representative application from the telecom industry that proves that an existing system can easily be ported to Cassandra. Finally, we quantitatively evaluate the impact of the additional middleware layer in the performance of both these applications.

The rest of the paper is structured as follows. Section 2 briefly describes the Cassandra distributed database and its data model. The object mapping is then introduced in Section 3. Section 4 describes the application of our proposal to a case study and Section 5 concludes the paper.

2 Background

There are nowadays several open source databases that aim at high scalability and present themselves as solutions to the challenges of “big data” . Among these solutions that include also HBase or Riak, Cassandra is one of most representative of the NoSQL movement and establishes an interesting bridge between the scalable architecture of Amazon’s Dynamo and the flexible data model from Google’s Big Table. With a straightforward deployment in one or more nodes, this solution allows to have a cluster of several machines ready and running in a matter of minutes, providing a highly available service with no single point of failure. Initially developed at Facebook, Cassandra was open sourced and is today an Apache top level project used at large web names like Digg, Reedit and Twitter.

On the other hand, when coming from a relational data model with static tables and foreign keys, the hybrid column model can be difficult to understand. This model is based in a sparse multidimensional map indexed by key in what we call a keyspace. In practice, a keyspace defines a collection of keys that can be sorted (lexicographically or by their bytes) or randomly distributed by a hash function through all the nodes of the database.

Inside a keyspace the data is then divided in Column Families. These are in some aspects equivalent to relational tables, as they usually contain data related to similar entities, like for instance the Book family or the User family. These units map, for each of the keys in the database, a collection of Columns, the base storage unit in the system. A column is a triplet that contains a name, a value and a timestamp, that is ordered in the row by the first of these elements in a user defined way. One important difference to the traditional relational model is that for each one of these rows mapped to a column family there is no theoretical limit in number or name for the columns that it can contain, bringing dynamism to the schema.

Column Families can also assume a different form, where instead of containing a single list, each key is associated to several lists of columns, each one mapped by a binary name. These structures are suitable to map information that is accessed as a whole, as sub columns under these structures are not indexed, and the retrieval of one of these elements implies the deserialization of all others. An example of such structures can be seen on Figure 1.

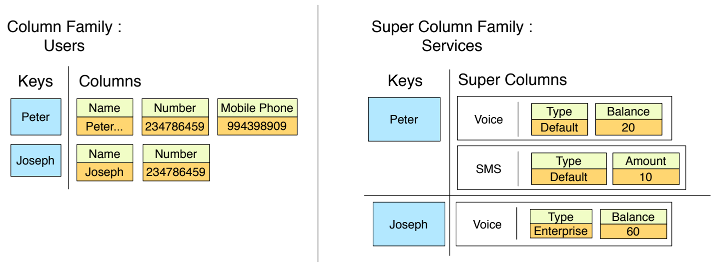

Fig. 1. Example of column and super column families.

But the data model is not the only issue where Cassandra diverges from the mainstream relational paradigm. Indeed, the base API supported in Cassandra is far from the high level interaction offered by SQL in a common RDBMS. Based on the Apache Thrift framework, it offers just basic get and put operations that can be executed individually or grouped in a single database request. Besides these basic operations, ranges are also available, but operations such as "get range from A to B" return meaningful results only when the keys are logically ordered in the database. All these instructions are executed atomically for each row, regardless the number of columns involved. No multi row guarantees are given though.

Extra complexity is also introduced by the consistency and conflict resolution mechanism that is associated to each one of these database operations. In fact, these are two main traits of this database that seem highly strange and confusing to new users. Each piece of data inserted with a timestamp in the database is usually replicated in N nodes and for each operation the user will define how many of these N shall guarantee its success. Asking for the Quorum nodes is usually the advised strategy, as it guarantees that when reading, at least one of the nodes has the last written data, getting strong consistency and fault tolerance for a reasonable price.

In the end, being a valuable option in a large scale scenario, the adoption of Cassandra as a solution presents three many problems that we address:

New concepts: Several are the concepts and mechanism that are completely new to most users when adopting Cassandra. Understanding the model, its characteristics and how to take part of it in the operations can sometimes be overwhelming. With our approach this layer is invisible to the user, that focuses only on the object model.

The consistency model also brings complexity with its levels and timestamps, but this is however something that we do not intend to address here. Even if we simplify the use of such parameters we do not present here any consistency mechanism whatsoever and we advise every user to fully understand the drawbacks of using such a solution before choosing it.

Code: The use of low level libraries such as the ones offered by Thrift gives the user an opportunity to choose the development language and tools unconstrained. Nevertheless these libraries have their own drawbacks, like code verbosity and poor readability. This is aggravated by the programming paradigm inherent to Cassandra's simple API. As the data is only accessed through its primary key or indexed columns and operations like `group by` or `order by` are not available, most of the code that was expressed in SQL queries is now done in the client side.

As expected, higher level libraries are available for various languages allowing an easier integration. Even so, these libraries often fail to hide many of the low level implementation details in a friendly way. In order to alleviate this problem our approach eases the processes of data persistence and retrieval, allowing the user to stay focused on the object model.

Integration: The last challenge faced is moving existing applications and the associated knowledge to this new paradigm. With a complete change in the development paradigm and no common ground between the old and the new applications, the code has then to be rewritten. This is often an unacceptable scenario due to the implied costs.

The recent introduction of the CQL language[6] is an important addition to the Cassandra project, and although immature and prone to change, it allows the relational-minded users an easier integration with the project with its SQL inspired syntax. But they are two different paradigms and besides the many operators that are missing in one or the other, the high level application code still needs to be rewritten.

Being based on a standard and widely known persistence interface, our proposal aims at an easy migration of existing ORM code with the advantage that it can be done in a progressive way and can be only partial. Going back to the consistency issue, this approach allows in the same code base to redirect the objects and related code that needs consistency guarantees to a database that provides ACID transactions while maintaining the rest of the model in a large scale data store. Other main advantage is that existing programming knowledge is also preserved.

3 Object mapping

We now present our approach for a Java object mapping tool for Cassandra. For this purpose and to leverage the existing work in the area, we build our mapper over the DataNucleus platform. This allows us to reuse many components, such as the parsing of Java classes and mapping files, and maintain the focus on the mapping to Cassandra's model. In fact, the DataNucleus platform offers a great advantage as it based in a set of extension points that allows the user to progressively implement different features when developing a new backend plug-in.

3.1 Object model

The main challenge when developing a object mapper is to find the most efficient way to store and retrieve the mapped entities while maintaining intact the relations among them. For this purpose and in the context of the Java language there are today two common specifications: the Java Data Objects (JDO) API and the Java Persistence API (JPA). These specifications, both supported by the DataNucleus platform, bear a set of

[6] `www.datastax.com/docs/0.8/api/cql_ref#cql-reference`

rules related to class mapping, persistence and querying. In the lower layers of the platform, the different implementations such as the one presented here are then responsible for the persistence, retrieval and deletion of objects from the database.

This process does involve more than a few writes or reads though, as it is also the platform's responsibility to establish how will the objects and their relations be mapped in the database. These relations, characterized in direction as being unidirectional or bidirectional and in terms of cardinality as one-to-one, one-to-many and many-to-many, depending in number of connections that an entity has, have then to be adapted to the database schema and characteristics. This is done as follows:

Class mapping Similar to the relational solutions, the base ground for the mapping process is that each class is persisted to a separated column family being the associated fields stored in individual columns. This approach besides being straightforward to implement has also the advantage of allowing the easy extraction of data by other processes in a later phase, as the data is clearly mapped by the name of its class and fields.

One to one/Many to one relations In one-to-one relations, objects of a class have an associated object of another class (only one associated object). Again this is fairly easy to do, being the associated object stored in its corresponding column family, and its key is then stored in the field column. There are, however, two main differences between this and relational approaches. The first relates to deletions where in the absence of foreign keys and referential integrity, deleting a object will not cascade to associated objects. To guarantee their deletion, the object should then be marked as dependent. The other difference is that its not possible to map a bidirectional relation with only one column family holding the reference as seen in relational solutions. This type of relation can only be achieved with both sides having a reference to its associate.

The same approach is used in one to many relations where an object of a class has an associated object of another class (only one associated object) like an account that has a connection to a user. The difference here is that the same user can have several accounts and they all have a link to it. As the scenario is identical, the same implementation and considerations about deletions are maintained. Bidirectional relations imply a different implementation though, similar to the one to many relations described below.

One to many/Many to many relations In one to many relations and in many to many relations, an object of a class A has several associated objects of a class B, being that in the second case, each instance of the class B refers also multiple objects of the class A. In order to map such relations we need a way to store the several connections that each one of the instances contain.

Several approaches were devised was shown on Figure 2. The first one is the storage of all the information in an extra column family. This column family would then contain for each one of the instances of the original object the keys to all the associated objects. This approach is straightforward but requires, a retrieval of all the associated keys and objects in two separated phases every time such field is fetched. For this reason this option is discarded.

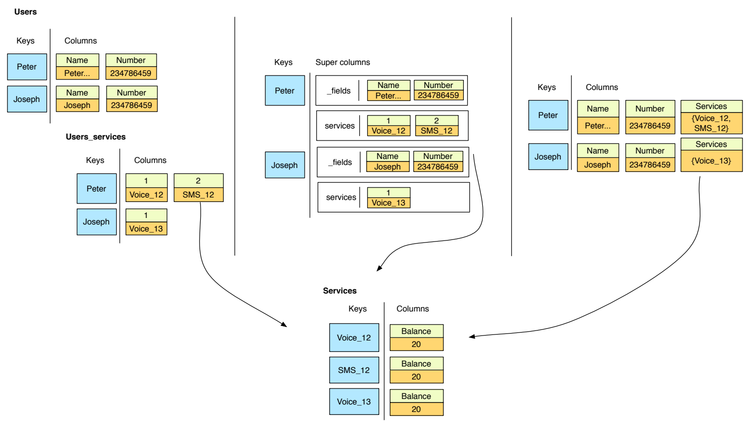

Fig. 2. Mapping strategies.

The second option is having all the information for the class stored in one super column family. In this option for each object/key, all the base fields are stored in one super column with a distinct name and the others are then used to store the keys to the associated objects. This presents a good way to store the relations in a clear way and when reading this field there is no additional fetch to an external mapping column family. However, this approach has no major advantages in performance when compared to the third solution and introduces a lot more complexity to the platform.

So the chosen approach is the third one where the mapping information is first constructed using the keys to the associated objects and its then serialized and stored in the column correspondent to that field. This approach as the same advantage on reads as the above, but even if the stored data is more obscure, the associated code is simpler and faster. In relation to the deletions the above considerations are maintained here.

4 Other implementation topics

Besides the mapping problem, the platform has also to deal with other subjects such as connection management, consistency levels and the querying mechanism. Starting with the connection management, this is a relevant part of the platform due to the distributed nature of the Cassandra database. In fact, many clusters can have dozens of nodes and for that purpose we included a node discovery mechanism that can be activated by the user in the configuration file. With just one node, the platform can then discover the rest of the cluster and distribute the load within it.

The other point that was approached was the consistency level parameter that must be included in most database operations. In a basic approach we use quorum reads and writes by default, parameter that can be changed during the execution but always in a global fashion. A simple enhancement to the JDO API though, will in the future allow the use of transaction isolation properties to define such consistency levels. In a similar manner, the timestamps must also be included in most write or deletion operations being this process automated by the platform.

Finally, a limited subset of JDOQL query language was implemented. JDOQL aims at capturing the power of expression contained in SQL queries and apply it to the Java object model. Range query capabilities were then implemented in the database allowing the user to get all keys within an entity, or just a range with defined start and end. Naturally, the range option is only useful when keyspace keys are ordered as otherwise it will return a random set of keys. Related to this point index support was also added in a posterior phase to enhance the querying capabilities of the platform. In a first phase, when indexes were not implemented in Cassandra, our approach consisted in the creation of extra column families that stored all the associated keys to that field value. This allows the user to query with a single non primary field. In later versions the index implementation was moved to Cassandra and it is now possible to query information using more than one indexed field.

5 Case study

In order to test and evaluate the overall impact of our approach we first implemented a subset of the TPC-W benchmark over Cassandra using the native interface, and then using the object mapper. This well known benchmark was chosen as it represents a typical Web application, representative of an area that is nowadays opening to new NoSQL solutions and where these solutions are increasingly being used. Although other options such as the OO7 benchmark [7] could be more fit to evaluate the raw performance of Cassandra, this is not our goal here. With a small description of the implementation and few snippets of code we intend here to describe the advantages of the adoption of such a solution. Latency measures are also shown to measure the drawbacks of this solution.

In a second phase we show how an existing telecom application, that simulates an real industrial problem of scale, can be ported to large scale database like Cassandra with this solution. With a small description of the system and its associated problems we describe how it was migrated. We then present results in terms of latency and throughput in the system, evaluating the performance of the object mapper in a real world scenario.

5.1 TPC-W

The Cassandra lower level API, currently over Thrift, offers to developers a multi language interface containing read, write and range operations among others. To be generic and include several cases of use, this interface requires from the user the knowledge about a set of API related concepts and structures, operational details, and also how to manage the consistency levels. When new to the system, the developer thus faces a steep learning curve. Even if the same applies to object-mapping solutions, these were by design intended for code simplicity and corresponding know-how to them associated is almost ubiquitous.

The first step that separates implementations is the coding of the entities and related meta-information that includes the identification of primary key fields and possible relations in the non native solution. With the use of JDO annotations, or with the XML

[7] `pages.cs.wisc.edu/~{}dewitt/includes/benchmarking/oo7.pdf`

mapping file, the coding of meta-data information is relative easy, as the basic settings are simple to learn and use. In fact, this first phase when we model the classes to be persisted is fundamentally different from the development paradigm connected to the NoSQL movement that favours the construction of data models that optimize the expected queries in the system. Even so, the development of this mapper over a mature platform as the one being used, allows the use of such optimizations as the fetching groups that allow to extract only some of the fields in a object.

When coding the persistence of entities, the simplicity introduced by the platform is also visible, as the user only has to invoke the platform persistence method with the target object as parameter. Loading methods also makes it easy to fetch entities based only on their identifications. In the native platform such simple operations would require the conversion of the key object and the value itself in a byte buffer in order to store it, not forgetting the various parameters to add such as the consistency levels and timestamps.

Indeed small operations, such as the item information extraction, can be easily mapped into Cassandra, but the implementation of complex operations like the best sellers list or the item searches by an author's name or category introduce here several changes the typical TPC-W implementations. What was in the relational model a relative simple query that fetched an item by a non key field, or that filtered all the customer orders in order to discover the most bought item, now implies the creation of indexes and the extraction of thousands of objects to do a client side selection. The mapper presents again an advantage with the automation of the index construction and the JDOQL implementation that, not avoiding the load of thousands of orders, allows to make it with a few lines of code. Due to space limitations we can't show here the difference between the implementations, but they are available on the project page. [8]

Results To run this benchmark we used seven HP Intel(R) Core(TM)2 CPU 6400 - 2.13GHz processor machines, each one with 2 GByte of RAM and a SATA disk drive. The operating system used is Linux, and the Cassandra version is 0.6 that implied client side maintained indexes. Referring to the benchmark parameters, the typical TPC-W load scale value is set to 10 and the number of items is 10000. To run the tests we then use 100 parallel clients with the standard think time.

Figure 3 shows that for small read and write operations the results are similar in both approaches. We than see that either the a) extraction of information for a item or the b) storage of a order after a sale present low values of latencies.

We can see however the price of using a higher level solution such as this one when we move to complex instructions like the ones seen in Figure 4. In 4(a) we can see the results for the search operation where the mapper presents a double behaviour that is associated to the type of this operation. In fact, in this operation that uses indexes, we see many low values in the mapper results that correspond to searches to an author name or book title that has few associated items. On the other hand, when fetching thousands of objects due to an item search by type or the best sellers option shown on 4(b), the loading of fields and relations take its toll when compared to the optimized native solution. These results can be improved though. With the use of fetch plans we can avoid

[8] Available at `github.com/PedroGomes/TPCw-benchmark`.

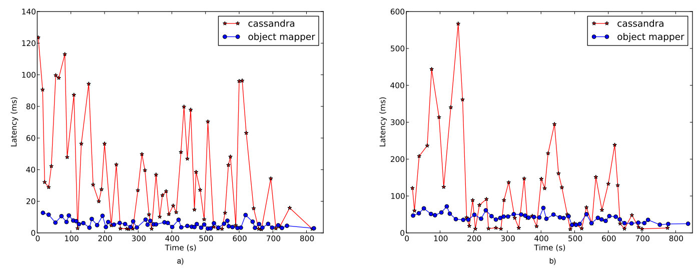

Fig. 3. Latencies for the (a) Item info (b) Buy Confirm operations in TPC-W.

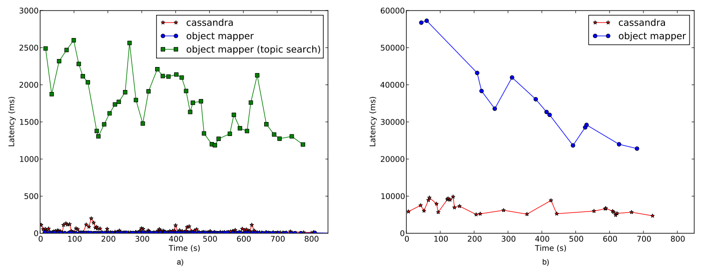

Fig. 4. Latencies for the (a) Item search (b) Best Sellers operations in TPC-W.

the loading of some of the fields and a improvement in the algorithm is schedule for future work.

5.2 A telecom benchmark

Telecom companies have been recently challenged as more and more mobile devices become on-line every day, and consequently more data is added to their data centers. These systems that seem at first sight easy to partition due to the nature of the client data that is self contained, are in fact hard to scale. This affects for example the activation process that all calls must pass to determine if they are valid. This process, that we evaluate here, consists in a series of validation steps that checks the services that the calling client has and its balances, and has then to validate them one by one until a valid is found. This validation processes is where the scale problems begin as they can require access to the list of services of the call receiver to see if they both share the same service, for example.

Indeed, companies find themselves with information about millions of customers that are many times related among themselves. These unpredictable connections among the clients destroy any chance of a clean partition schema in the data that then raises a storage problem. In this context, the Cassandra database was then chosen as possible solution to provide a fast and elastic platform for storing and query the user data. But this proposal raises some problems. The first is that some of the information can't be

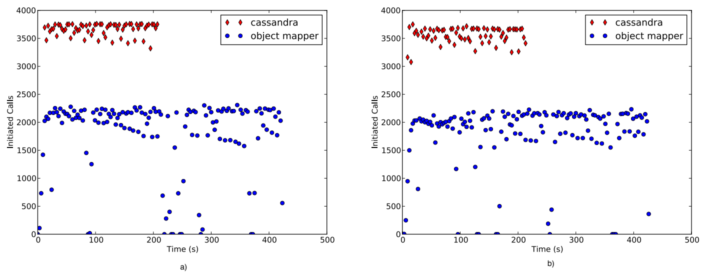

Fig. 5. Throughput with the telecom application for a a) day b) goal scenario.

migrated, as for instance, the client's balance that needs to be consistent at all times, or the company may face complains and/or monetary losses. The other problem refers to the costs and dangers of rewriting the code from scratch, even more when only one part of it can be mapped to Cassandra.

This is the ideal scenario for using our mapper. Even if the original code base is not based in a object mapping solution, the higher level code can probably be adapted and the developer can then work at the object level and have at the same time two persistence sources: one that grants scale and the other consistency.

Results To evaluate this approach, we use a benchmarking tool especially designed to test this kind of system with several kinds of databases. Adapted in a similar process to the TPC-W, this platform allows us to test our mapper in a real world problem and load. Indeed the tests presented here were run over a population of 7 million clients with two different settings that simulate distinct usage scenarios. These, that were named "day" and "goal", represent a typical day of the week with lots of business calls and the load of calls that the users of a service cause in a sports event respectively. These tests were run with a 24 core machine with 128 GB de RAM and RAID1 disks. The client machine was run in a 8 core machine with 16 GB of RAM. The operating system used is Linux, and the Cassandra version is 0.7 meaning that the indexes are now handled by the database. In terms of load, the tests were run with a total of 20 concurrent client request threads.

Looking to the results we see that the native solution that has a specially designed model to run this benchmark preforms better than the mapper by almost 1500 calls per 30 second period. Even so the average of 2300 calls per each 30 second period is reasonable for a solution that exhibits a friendlier API and a more generic model. When looking at the latency values, we notice that for each call there is a mean latency of 12 milliseconds, that even if greater than the native results is in fact a good value.

6 Conclusion

In this paper we present a new object mapping tool for Cassandra aiming at leveraging this scalable database in an easy and fast manner. The main challenge addressed was

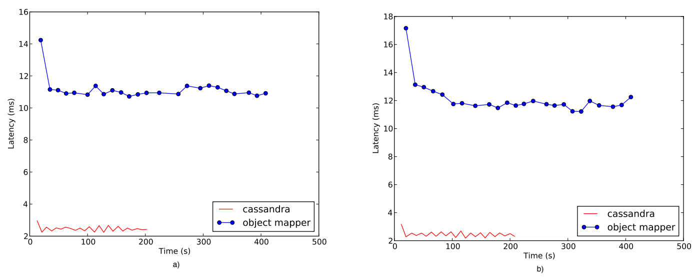

Fig. 6. Latency with the telecom application for a a) day b) goal scenario.

mapping object relations to the data model offered by Cassandra, which was achieved by persistint their meta-data in the corresponding entities. The resulting object mapping, developed as a plug-in to the DataNucleus platform, has been published as open source software. It has since been accepted by both the Cassandra and DataNucleus developer communities. From the early version of this project there was also a fork that being developed in a enterprise environment is now used in production[9].

References

1. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation. pp. 205–218. USENIX Association, Berkeley, CA, USA (2006), `http://portal.acm.org/citation.cfm?id=1298455.1298475`
2. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. pp. 205–220. ACM, New York, NY, USA (2007), `http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf`
3. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2), 51–59 (2002)
4. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data. pp. 173–182. ACM, New York, NY, USA (1996)
5. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44(2), 35–40 (2010)
6. (TPC)., T.P.P.C.: Tpc benchmark w(web commerce) specification version 1.8 (2002)

[9] `https://github.com/tnine/Datanucleus-Cassandra-Plugin`