Journal of Universal Computer Science, vol. 21, no. 13 (2015), 1810-1829
submitted: 4/5/15, accepted: 6/7/15, appeared: 28/12/15

A Distributed Recommendation Platform for Big Data

Daniel Valcarce
(Information Retrieval Lab, Computer Science Department
University of A Coruña, Spain
daniel.valcarce@udc.es)

Javier Parapar
(Information Retrieval Lab, Computer Science Department
University of A Coruña, Spain
javierparapar@udc.es)

Álvaro Barreiro
(Information Retrieval Lab, Computer Science Department
University of A Coruña, Spain
barreiro@udc.es)

Abstract: The vast amount of information that recommenders manage these days has reached a point where scalability has become a critical factor. In this work, we propose a scalable architecture designed for computing Collaborative Filtering recommendations in a Big Data scenario. In order to build a highly scalable and fault-tolerant platform, we employ fully distributed systems without any single point of failure. We study the use of data replication and data distribution technologies. Additionally, we consider different caching techniques. Taking into account these requirements, we propose particular technologies for each component of the platform. Next, we evaluate the response times of storing, generating and serving recommendations using MySQL Cluster and Cassandra showing that the latter technology is much more adequate for that purpose. Finally, we conduct a simulation for evaluating the impact of a memory caching system.
Key Words: Recommender systems, big data, scalability, architecture, NoSQL, cache.
Category: H.3.3

1 Introduction

It has been reported that, in 2013, the web traffic generated by search engines dropped 6% meanwhile that originated in the social networks increased more than 100% [Wong 2014]. This change in the users' behaviour is an indicator of the importance of recommender systems. Users are expecting more suggestions from the systems instead of explicitly formalize their information needs in the form of a query. The traditional use of the World Wide Web used to consist in browsing and manually searching. In contrast, nowadays the so called Web 2.0 offers the users a lot of content recommendations: contact suggestions, personalised timelines, automatic tagging, etc. Thus, the importance of recommender systems is growing in order to attend users' demands.

Recommender systems [Ricci et al. 2011] aim to predict items that may be of interest to users. In this way, there is no need of an explicit request for information: the system learns about users and generates personalised suggestions for them. Unfortunately, diversity of scenarios and domains makes the task of finding relevant items a non easy one. Different approaches have been proposed for this task. Traditionally, recommender systems are classified [Ricci et al. 2011] in Content-Based methods [Resnick and Varian 1997] which exploit the similarity of candidate items with the ones already assessed by the user, Collaborative Filtering [Schafer et al. 2007] techniques which exploit the information about the preferences of similar users to the subject of recommendation and, finally, hybridisations of both families.

In this work, although it is not devoted to the discussion of recommendation algorithms, we will focus on Collaborative Filtering (CF) methods. CF is a popular approach in multiple recommendation scenarios. The rationale behind this fact is that CF exploits the preference patterns existing in any community (regardless of the type of items) offering personalised recommendations. Commonly, the users' preferences are explicitly elicited in the form of ratings. In other cases, those preferences are obtained from implicit feedback such as clicks. All this information (either ratings or clicks) has to be efficiently stored in real-time. Additionally, it has to be easily accessed by recommendation algorithms during their executions.

There exist several application domains for recommendation where the number of users, items and rating increased dramatically turning the recommendation problem into a big data challenge (e.g., music, web-pages, videos, friends or tweets recommendations). Nevertheless, the amount of scenarios reaching big data scales is increasing day after day.

In this large-scale context, the architecture of a recommender system is becoming critical to its success. In this paper, extending the analysis presented in [Valcarce et al. 2014], we describe a complete recommender platform capable of generating recommendations in a big data landscape. This new analysis also covers the study of the effects of the caching technique in the platform. Additionally, we update the technology elections made in [Valcarce et al. 2014]. In particular, we replace the distributed caching system since the reason for choosing Memcached over Redis is no longer valid. Furthermore, we discuss in more detail different NoSQL solutions.

We can distinguish three main components in our proposed architecture: a web front-end which consists in a web application, a recommendation engine that uses MapReduce [Dean and Ghemawat 2008] and a hybrid persistence layer. We propose specific technologies for storing, processing and serving web-scale information. To this end, we evaluate different families of storage technologies (relational databases, NoSQL systems [Sadalage and Fowler 2012] and inverted indices) showing that a mixed solution fits our persistence needs. In particular, we perform a series of experiments to study the efficiency of a distributed relational solution and a NoSQL column-store, under realistic circumstances. Additionally, we analyse the impact of a distributed memory caching system by simulating a real environment.

2 Recommender System Architecture for Big Data

The development of a general recommender architecture capable of processing large-scale data is currently a challenge [Cortizo et al. 2010]. In spite of the fact that this problem is an important concern for the industry, it has attracted little attention in the academia. Therefore, in this section, we describe the complete architecture of a recommendation platform justifying the technology selection in detail.

Our architectural proposal was designed to achieve two main goals: high scalability and high availability at every level. In addition, the infrastructure should be capable of storing continuous updates of the users' ratings history and provide high quality and fresh recommendations. We consider three different main components: the user interaction layer, the recommendation engine and the persistence component. Each subsystem can be either monolithic/centralised or distributed. When we scale a recommender system out to the big data landscape, any of those components is a potential point of failure in the architecture. Thus, we need to choose distributed and fault-tolerant technologies to satisfy the high availability requirement.

The overall design of the proposed recommendation platform is exhibited in Figure 1. We can easily distinguish its main components: the web front-end (a web application), the recommendation engine using MapReduce framework and the data storage component. In the following sections, we describe these subsystems justifying the choices made in each component with respect to the desired goals.

2.1 Front-end

The user interaction layer of the proposed platform consists of a web application where users can search and rate items. This website also offers personalised suggestions to the users. In order to allow them to perform complex search queries, we implemented faceted search. This technique gives the users the opportunity to explore an item collection by applying several filters as a complement to the recommendations with the aim of increasing users' ratings since we want to optimise all aspects of the user experience. In fact, it has been reported [Amatriain 2012] that 25% of played films in Netflix are not based on recommendations.

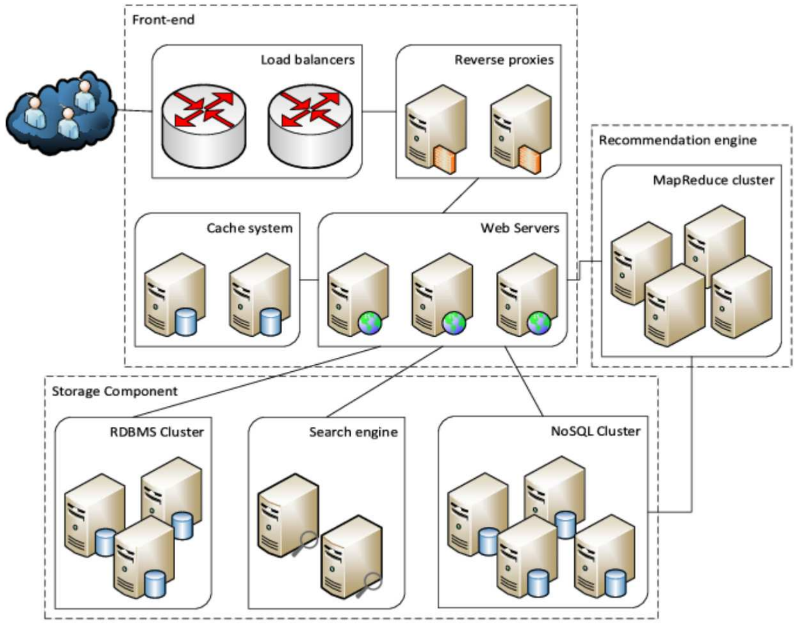

Figure 1: Overall system architecture

Guided by our goals (scalability and availability), we deployed redundant web servers with a load balancer subsystem on top. A cluster of web servers is needed for scaling horizontally (i.e., adding more nodes increases system capacity to serve requests). We should note that not only do we improve throughput with more servers, but also reliability: if a node goes down, others can replace it. The task of the load balancer module consists in distributing web traffic evenly across the cluster nodes.

We developed a web application using the Django framework [see 1] because it is a well-known tool that easily integrates with the rest of technologies described below. Django uses a shared-nothing architecture, that is, every web server can operate independently. Moreover, it has been previously used in big data environments such as Disqus, Pinterest or Instagram with success.

For the load balancing task, we chose Perlbal [see 2] following the recommendation by [Holovaty and Kaplan-Moss 2009], but there exist other adequate options. Actually, a load balancer appliance (i.e., a device that implements load balancing in hardware) would be a more efficient solution than a specialised

[1] `http://www.djangoproject.com`
[2] `http://github.com/perlbal/Perlbal`

software. In this case, the trade-off between inversion and desired performance is the key factor in the decision. We need to make sure that we employ at least two load balancers in a failover configuration to achieve high availability. In this way, if a balancer fails, the other will continue to provide service.

2.1.1 Caching technologies

Additionally, we decided to integrate two types of caching techniques in the platform. The rationale behind this decision is based on the fact that each request that hits the server involves some kind of computation and, probably, database accesses. In high-traffic web-sites, caching is fundamental to reduce server load and avoid repetitive database queries. In our context, the most frequent case is when the users browse their recommendations or when they look for information about the last popular releases. Without caching, each time a user requests this data, the system sends a relatively expensive query to the storage component.

The first type of cache is a coarse-grained approach. The idea is to put reverse proxies to cache entire HTTP petitions. As Disqus operation team reported [Robenolt 2014], using a caching HTTP reverse proxy may greatly reduce the number of requests processed by the web framework and, consequently, the database. These proxies are situated between the load balancer and the web server cluster and its aim is to cache responses to the web requests in order to lessen web traffic to the servers. Nowadays, Varnish [see 3] and Squid [see 4] are two strong competitors in this field. We favour Varnish over Squid because it demonstrates better performance in benchmarks [Migliorisi 2014].

The second level of cache is used inside the web application for caching expensive tasks like the result of some database queries or the user sessions. The success of such approach is a very well studied topic in Web Information Retrieval, where the so called *answer caching* [Baeza-Yates et al. 2007] has been demonstrated as the most effective caching approach. We considered two technologies for this task: Memcached [see 5], a distributed memory object cache system, and Redis [see 6], an advanced key-value store. Traditionally, Memcached was the de facto standard for caching web applications since it is a simple and mature technology for storing small HTML fragments. Nevertheless, Redis can store any type of object (even collections), has tunable persistence to disk and its size limits are higher than Memcached. In our previous work, we favour chose Memcached over Redis because of its clustering capabilities [Valcarce et al. 2014]. Nowadays, this reason is no longer valid since Redis 3.0 provides out-of-the-box

[3] `http://www.varnish-cache.org`
[4] `http://www.squid-cache.org`
[5] `http://memcached.org`
[6] `http://redis.io`

clustering features: data is automatically split among multiple nodes and failures on a subset of nodes do not compromise the availability of the system.

2.2 Storage Component

Django framework is designed for building database-driven web applications and it provides support for several database technologies. A recommendation platform needs to store different types of information. First, we need to manage large amounts of users' ratings (which are read by the recommendation algorithm) and their recommendations (which are computed in a batch process). Second, we also store information about all the items. Finally, we have to deal with web application data such as user profiles or sessions. To address the storage problem, we studied three different paradigms: relational databases and NoSQL systems and information retrieval structures.

2.2.1 Relational Databases

At the first look, Relational Database Management Systems (RDBMS) can be the ideal solution for storing data needed by the web application or even the information about items. In contrast, there may be serious performance issues if we want to manage large-scale data such as the users' ratings and recommendations. Officially, Django supports natively four RDBMS: SQLite, PostgreSQL, Oracle and MySQL. The analysis of each solution is described in the next paragraphs.

SQLite [see 7] was rejected considering it is designed for light databases and embedded systems.

PostgreSQL [see 8] is an object-relational database system. There exists some tools like pgpool-II [see 9] that add support to data replication (which give us fault-tolerance and read scaling), however it does not provide transparent sharding across nodes in a cluster, i.e., the capability of distributing horizontal partitions (collections of rows of the same table) between different machines. In this big data environment, write scaling is also crucial and it is achieved with sharding. Postgres-XC [see 10] is designed to provide a transparent write-scalable cluster solution. Nevertheless, either replication or distribution has to be selected in the table creation process. We excluded PostgreSQL considering we are looking for a system able to handle these two features at the same time.

Oracle RDBMS [see 11] is also a object-relational DBMS like PostgreSQL. It offers an option called Oracle RAC (Real Application Clusters) for support-

[7] `http://sqlite.org`
[8] `http://www.postgresql.org`
[9] `http://www.pgpool.net`
[10] `http://postgres-xc.sourceforge.net`
[11] `http://oracle.com/database`

ing clustering and high availability. In contrast to Postgres-XC where a shared-nothing approach is followed, Oracle RAC is based on a shared-everything architecture. Therefore, all database instances need access to the same shared storage instead of using their own private disk. This architecture involves a significant investment in Storage Area Networks (SAN). To avoid that this part becomes a single point of failure in the cluster, expensive replication infrastructure should be used. Furthermore, the use of a SAN instead of a DAS (Direct-Attached Storage) is not generally recommended for big data analytics [Bai and Wu 2011].

Finally, we analyse MySQL Cluster [see 12]. This product is based on a shared-nothing clustering architecture and it claims to be an ACID (Atomicity, Consistency, Isolation and Durability) compliant system with no single point of failure. It provides read and write scalability due to its replication and auto-sharding features. Even though MySQL Cluster uses an in-memory storage by default, it can be configured to also store non-indexed columns in disk using main memory as a cache. This configuration offers good performance meanwhile huge amounts of information can be managed. Based on these features, we chose MySQL Cluster as the storage system for the relational data (such as item details and web framework data). In Section 3, we evaluate the suitability of this solution for storing ratings and recommendations.

2.2.2 NoSQL Databases

Nowadays, there exist multiple NoSQL solutions [Cattell 2011]. These systems do not use tabular relations like RDBMS and they claim being more scalable and flexible. There are different approaches for classifying NoSQL datastores, but it is common to distinguish four categories: key-value, column-oriented, document-oriented and graph databases [Sadalage and Fowler 2012]. RDBMS fulfil ACID properties whereas many NoSQL solutions follow the BASE consistency model (Basically Available, Soft state, Eventual consistency). In our case, the relaxation of the ACID constraints is not a problem. The recommendation process can be done without a few ratings, the suggestions will be only slightly modified. Moreover, since the recommendations will be computed periodically, the BASE model guarantee that eventually all the users will have fresh recommendations.

Key-value stores are the simplest NoSQL databases and, thus, very fast. These databases only allow to get, set and delete a value by key. Its simplicity provides a high scalability making them suitable for caching purposes. In fact, Redis and Memcached are key-value databases. Nevertheless, these technologies are not suitable for storing ratings and recommendations since they are identified by a composite key (user and item), but we also want to make queries using only the user as a key.

[12] `http://www.mysql.com`

Document-oriented databases (such as MongoDB or CouchDB) stores documents in the form of XML, JSON, BSON... These hierarchical data structures need not to have the same structure and may contain complex collections such as sets or lists. These databases are able to store ratings and recommendations. However, since ratings and recommendations have the same simple structure, the use of document-based datastores is not ideal: we are not using many of the features that document-oriented databases offer and there exist other NoSQL solutions with better performance.

Graph databases (such as Neo4j) store relationships between entities as first-class citizens. They can be used for efficiently store many different types of relationships. In addition, they also provide ACID transactions. In our case, we have ratings and recommendations between users and items. With only two types of relationships, the use of a graph-based database is probably not the most adequate.

Finally, column-oriented databases (also known as extensible record stores) provide high scalability and are well-suited for storing users' ratings and recommendations. These databases store data in rows (in our case, a row is a user). Each row contains a variable number of columns which represent an item and its rating or recommendation score.

As will be described in Section 2.3, we decided to use Hadoop MapReduce framework in order to make personalised recommendations. Because of that, we studied two column stores generally used with Hadoop: Cassandra and HBase. The CAP theorem [Gilbert and Lynch 2002] says that a distributed system can only guarantee two of the following three properties: Consistency (all nodes see the same data at the same time), Availability (every petition receives a response indicating if it succeeded or failed) and Partition tolerance (the system can operate in spite of arbitrary partitioning produced by network failures). Cassandra focuses on delivering availability and partition tolerance while HBase sacrifices availability.

Apache Cassandra [see 13] is a highly scalable, eventually consistent and distributed DBMS. Eventual consistency guarantees that, if no new updates are made to an object, eventually all accesses to that object will return the last updated value. Cassandra is fault-tolerant thanks to replication, allows to add new nodes meanwhile keeping a linear read and write scalability (thanks to transparent partitioning and distribution) and there exists no single point of failure because of its distribute design. Roughly speaking, it can be said that Cassandra takes from Google Bigtable [Chang et al. 2008] its data model and from Amazon Dynamo [DeCandia et al. 2007] its distributed architecture.

Apache HBase [see 14] is also a distributed and linear scalable DBMS that

[13] `http://cassandra.apache.org`
[14] `http://hbase.apache.org`

uses Hadoop distributed storage file system (HDFS). It is inspired in Google Bigtable and it is designed for hosting billions of large rows on commodity hardware. Contrary to Cassandra where strong consistency is optional, in HBase it is guaranteed using logging and locking.

In spite of the fact that they have similar features, we chose Cassandra over HBase based on their performance and availability. Rabl et al. [Rabl et al. 2012] showed that Cassandra clearly outperforms HBase in almost all examined scenarios. We can afford a reduced loss of consistency in our recommendation platform, considering users do not usually modify their ratings, in exchange for higher efficiency and availability. Moreover, only in a scenario where recommendations are calculated in real-time, eventual consistency could be a problem.

2.2.3 Search Engines

As well as storing ratings and making recommendations, our platform is designed to be able to process complex search queries, specifically faceted search. We propose the use of search engines because the previously described database systems (either relational and NoSQL ones) are not well-suited for this task. Apache Lucene [see 15] is probably the most famous information retrieval software library and it supports full text indexing and searching features. Below, two popular search engines built on top of Lucene are described.

On the one hand, Apache Solr [see 16] is a mature and fast search engine. It includes distribution and fault tolerance features as sharding and replication under the name of Solr Cloud. On the other hand, Elasticsearch [see 17] is a modern distributed real-time search and analytics engine. Both of them support faceted search and therefore are valid alternatives to address our needs. We chose Apache Solr because its mature and consolidated nature.

2.3 Recommendation Engine

Now we describe the core of the proposed recommendation platform: processing data for producing recommendations. One of the most successful approaches to face this problem at big data scale is MapReduce [Dean and Ghemawat 2008], a functional programming model designed to treat large datasets in a distributed platform. We propose the use of Apache Hadoop [see 18], an open source implementation of MapReduce model, together with Apache Mahout [see 19], a machine learning library that contains different distributed algorithms built on top of Hadoop.

---

[15] `http://lucene.apache.org`
[16] `http://lucene.apache.org/solr`
[17] `http://www.elasticsearch.org`
[18] `http://hadoop.apache.org`
[19] `http://mahout.apache.org`

Hadoop is designed for doing batch calculations; hence, recommendations are precalculated and stored. In order to provide fresh recommendations, we suggest pipelining MapReduce jobs as the Youtube recommendation system does [Davidson et al. 2010]. A recommendation pipeline consists in launching new recommendation processes before the actual MapReduce job has finished. In this way, the suggestions are updated more frequently and provide a fresher appearance to the user.

At the time of writing this paper, Mahout implements two distributed Collaborative Filtering algorithms: Item-Based and Matrix Factorization with Alternating Least Squares. In Section 3.2 we examine the performance of the first method using MySQL Cluster and Cassandra as data sources. Due to our desire of benchmarking different data storage technologies, we decided to use the first algorithm, the least computationally expensive technique, because it gives us the opportunity of focusing on data consumption costs.

3 Storage Component Evaluation

Since our aim is to find the most suitable database for storing ratings and recommendations, here we do not focus on evaluating recommendations quality. Instead, in this benchmarking effort, we studied two different approaches to address the big data challenge: MySQL Cluster, a clustered RDBMS solution, and Cassandra, a fully distributed NoSQL DBMS. Although Cassandra and MySQL have been compared, to the best of our knowledge, this is the first rigorous experimental study of Cassandra and MySQL Cluster performance under this scenario.

Both data storing technologies have demonstrated linear scalability adding new nodes [Rabl et al. 2012, Oracle 2012]. However, it should also be noted that MySQL Cluster, in contrast to Cassandra, does not provide any load balancer policy. In our tests, we implemented a round-robin policy to face this issue.

The cluster used for the benchmarks consists of four nodes with two Intel Xeon E5504 CPUs, 16 GB of RAM and two 1 TB disks connected with a Gigabit Ethernet switch. The tests we performed include concurrent rating insertion, recommendation generation and concurrent recommendation serving. Each database is configured to work with a replication factor of two.

We used the dataset that was made available for the Netflix Prize [see 20]. This huge collection includes 100,480,507 ratings that 480,189 users gave to 17,770 films. Nevertheless, the proposed system is independent of the dataset. Any type of collection with ratings between users and items is susceptible of being used in our platform. The rationale behind the decision of choosing this particular dataset is its large dimensions. Below, we introduce the applied methodology.

[20] `http://www.netflixprize.com`

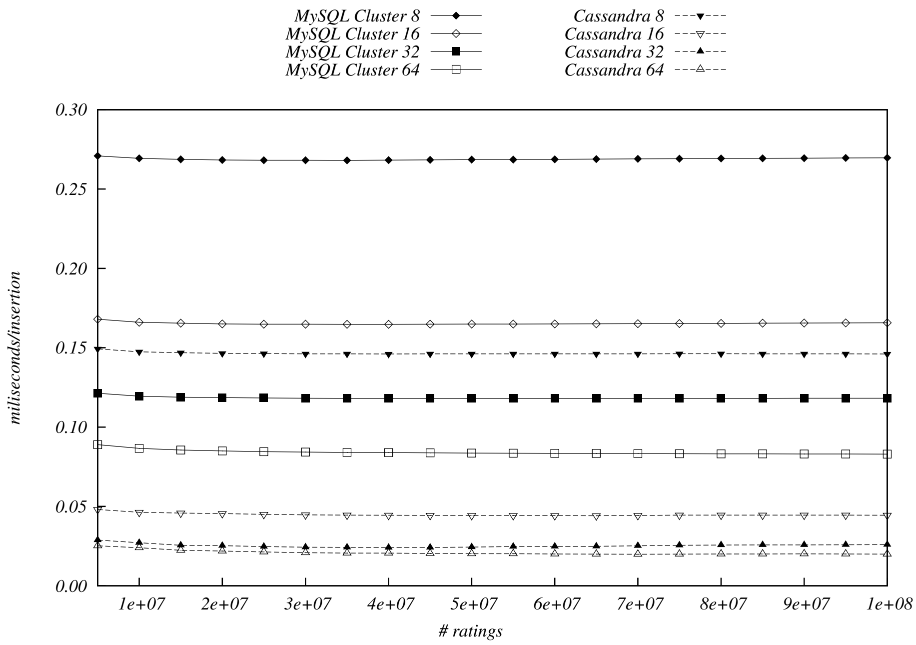

Figure 2: Cassandra vs MySQL Cluster per insertion time when using 8, 16, 32 or 64 concurrent petitions and moving the number of ratings from 10 to 100 million users ratings. Times (in milliseconds) were obtained in a cluster of four nodes

3.1 Rating Insertion

We measured writing times of inserting film ratings from the Netflix dataset using different number of concurrent connections.

The results of inserting all Netflix dataset ratings, illustrated in Figure 2, show that Cassandra outperforms MySQL Cluster in every scenario. In addition, it should be noted that the operation times hardly increase with the number of inserted ratings although a warm-up overhead can be observed at the start.

3.2 Recommendation Generation

Using the data inserted in the previous experiment, we configure Mahout's item-based CF algorithm to fetch and store data from/to MySQL Cluster and Cassandra. We measure the overall time of making recommendations for the whole Netflix dataset (recomendations for 480,189 users) averaged by three executions.

The recommendation algorithm worked well in conjunction with Cassandra. However, we were not able to store data directly into MySQL Cluster because Hadoop outputs all recommendations in bulk using `DBOutputFormat` class. This leads to massive transactions causing database crashes. This event does not happen with Cassandra because `CqlOutputFormat` inserts recommendations as soon as they are generated. To overcome this problem in MySQL Cluster, we wrote recommendations into HDFS (Hadoop Distributed File System) and then used Sqoop [see 21] to export the data from HDFS to MySQL.

The results of the tests are 68.85 minutes using Cassandra (8.6 ms per user in average) and 274.73 minutes using MySQL (34.3 ms per user in average, being the use of Sqoop the crucial factor in the differences). Recommendation generation is not on demand, we conceive the recommendation generation as an off-line process, however, when fresh recommendations are needed frequently in the domain of application, the recommendation algorithm can be pipelined in order to provide a high updating rate, i.e., starting different recommendation generation processes in parallel when a given amount of change is detected in the rating information.

3.3 Recommendation Serving

Lastly, we focus on providing users with recommendations. In this test, we analysed the read times of querying the top items for a user. In order to be able to serve recommendations in real-time, we need to retrieve the first top suggestions efficiently. Depending on the underlying technology, this is achieved by different manners. In the case of MySQL Cluster, we had to create a B-tree index on user and score columns in addition to the primary key index on user and item columns. Since MySQL Cluster maintains all indexed columns in distributed memory, this action worsens the scalability of this technology. In contrast, in Cassandra, when we create the table, we can specify the clustering order of the columns. In this way, all recommendations are stored sorted and they can be retrieved efficiently.

Our experiment consists in querying the top 10 recommended items for 25 million users. Considering that Netflix dataset has about half a million users, many queries will be repetitive. In this test, we want to test the reading performance of the database, thus, the described caches in Section 2.1 are not used. We study serving times under different number of concurrent queries. The results illustrated in Figure 3 show that Cassandra consistently improves MySQL Cluster in every scenario.

[21] `http://sqoop.apache.org`

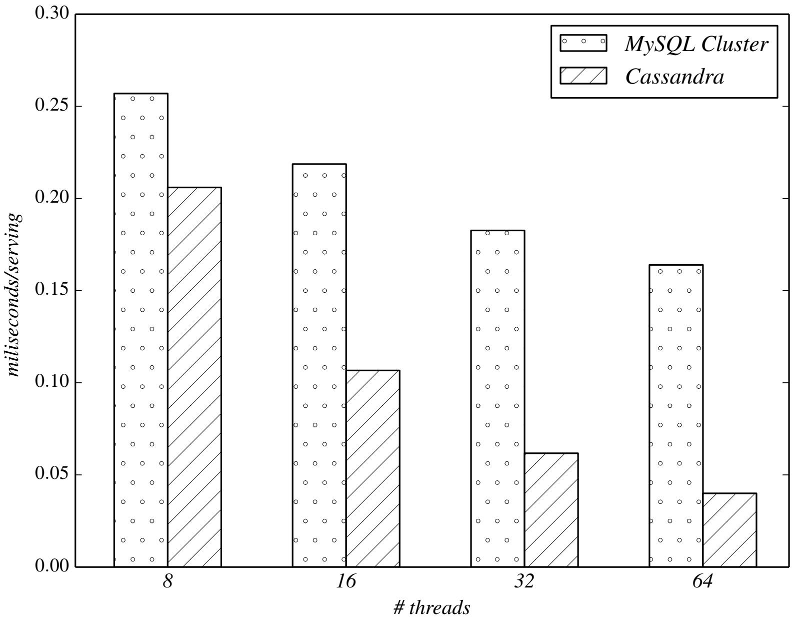

Figure 3: Cassandra vs MySQL Cluster serving time per user recommendation when using 8, 16, 32 or 64 concurrent requests. Times (in milliseconds) were obtained in a cluster of four nodes

3.4 Final choice

In view of the results, we chose Cassandra over MySQL Cluster. We justify this decision in the following paragraphs.

Firstly, a potential MySQL Cluster drawback is the fact that it needs to store indexed columns in main memory which compromises its scalability if the data stored in a node exceed its main memory. For instance, our first attempt to compare MySQL and Cassandra involved using the Bigflix dataset [Schelter et al. 2013], with 25 million users and 5 billion ratings, but given the MySQL memory limitation using 4 nodes and replication factor 2, the Bigflix collection could not be fitted in this cluster using MySQL. Although MySQL could still be valid in a lot of cases, its choice could compromise the use of this architecture at big data scale.

In addition, Netflix Engineering [Amatriain 2012] reported an insertion rate of 4 million ratings per day (an average of 46 requests per second). Probably rating insertion will not be uniform, in this scenario (64 concurrent requests) Cassandra is 4 times faster than MySQL Cluster.

Moreover, recommendation generation is a very slow process using MySQL Cluster due to the writing performance. Finally, the difference of performance when serving recommendations reinforces the election.

4 Cache Evaluation

The use of caches is one important part of the proposed system. Caching techniques allow the systems to serve requests in large-scale scenarios. In this section, we designed an experiment to evaluate the effect of Redis in the proposed platform.

4.1 Experiment setup

To this end, we use Apache JMeter [see 22] an application for load testing functional behaviour and measuring performance. In order to avoid interaction with other uncontrolled variables such as network latency, we simulate 30 concurrent users accessing the same machine. They arrive at a rate of one user per two seconds plus a Poisson distributed value with $\lambda = 1$ second. In this way, we can simulate random arrival of users with a known average rate using a Poisson distribution.

In this simulation, we aimed to reproduce a standard user behaviour in the web application. Thus, we designed a typical interaction based on an expert judgement. Each user's action requires some time from the user waiting time which is estimated using a Normal distribution. The users' actions as well the waiting time are indicated below:

1. Go to the platform index page: arrival $\sim$ Poisson(1).
2. Log into the system: user interaction time $\sim \mathcal{N}(3, 2)$.
3. Check the user's homepage: user interaction time $\sim \mathcal{N}(10, 6)$.
4. Check personal ratings: user interaction time $\sim \mathcal{N}(10, 6)$.
5. Check tailored recommendations: user interaction time $\sim \mathcal{N}(20, 10)$.
6. Check one random top recommendation: user interaction time $\sim \mathcal{N}(10, 6)$.
7. Log out.

[22] `http://jmeter.apache.org`

For these experiments, we disabled the coarse-grain cache (i.e., Varnish) because we are interesting in measuring the sole effects of the fine-grained cache. First, we evaluated the performance of the system without employing Redis caching system. Second, we cached every SQL request to MySQL Cluster. Finally, we cached not only database accesses but also the result of computing the views of Django, that is to say, the pieces of HTML code that are sent to the users.

4.2 Response time results

Figure 4 illustrates the results of the simulation. As it can be observed, the use of Redis as a caching system improves the response time of the web application. Improvements are noticeable either using only database caching or using both database and views caching. In the case of visiting the homepage or logging out of the system, the response times are quite similar. The reason is that these pages are static and almost no processing is needed. Nevertheless, in these cases, the differences in performance are very small, probably not significant. In contrast, the action in which the user checks the recommendations is the most benefited. This fact is mainly motivated by the popularity bias introduced by any recommender.

The time of login may appear quite noticeable. This is because of the overhead induced by the creation of the user's session and its storage in the database. The use of Redis enables to store the sessions in the cache lightening the login load.

Additionally, it can be appreciated that caching not only accesses to the database but also the views of Django improves even more the response time of the system.

We also experimented with other parameters obtaining the same trends as the ones showed in Figure 4.

In the light of these results, we can affirm that the introduction of caching techniques into the system improves the overall performance of the platform. These results are consistent with the ones reported for Information Retrieval tasks [Baeza-Yates et al. 2007] where caching the query results yields in an improved performance since many queries that users introduce into the system are the same. In the recommendation scenario, this phenomenon also occurs. Although users receive personalised suggestions, popularity is a key factor in the recommendation process. In addition to fashion trends, this causes that some items are more usually requested than others: the so called *long tail* [Anderson 2006]. Consequently, caching these popular elements provide appreciable improvements in response time.

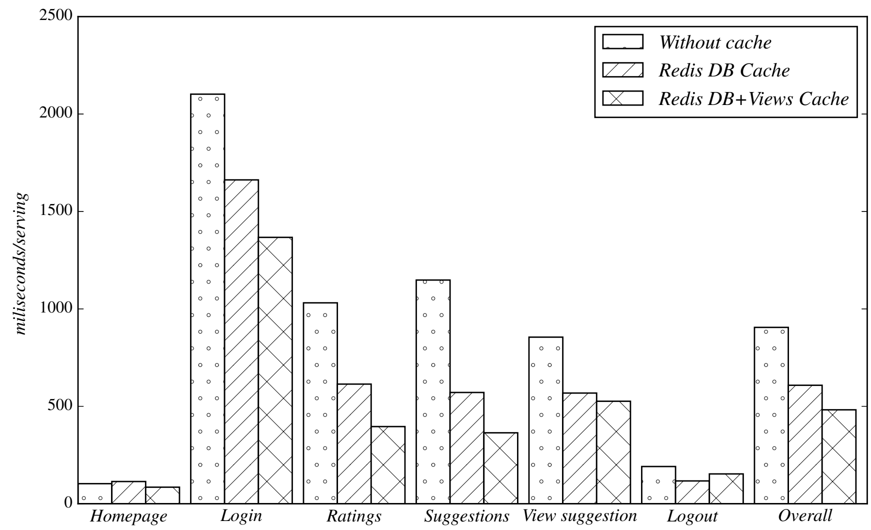

Figure 4: Response times of the system using 30 concurrent users on one node without cache, caching only database accesses and caching database accesses and views.

5 Related Work

Scalability of recommender systems is not a novel topic. In fact, it has been counted as one of the most urgent challenges in recommendation frequently [Cortizo et al. 2010]. This topic has been thoroughly studied by many companies, and nonetheless it continues to be a source of research. For example, Hulu [see 23] and Netflix [see 24], two big film recommendation platforms have reported some details about their architectures in their technical blogs. Unfortunately, because of the strategic importance of this type of information, these blog posts only offer a superficial overview of their platforms.

The architecture described in this paper shares some similarities with the short description of the recommendation system implemented in Youtube [Davidson et al. 2010]. This video portal computes recommendations using pipelined MapReduce jobs. In this way, they can simulate to provide fresh recommendations. On the other hand, they also store the user logs (ratings, clicks, plays, etc.) as well as the computed suggestions in Bigtable [Chang et al. 2008], a column-oriented database on which Cassandra is based. This supports the idea of the

[23] `http://tech.hulu.com/blog/2011/09/19/recommendation-system`
[24] `http://techblog.netflix.com/2013/03/system-architectures-for.html`

convenience of using Cassandra instead of MySQL Cluster.

Other Google service, Google News, also uses Bigtable for storing the history of the users [Das et al. 2007]. They combined three algorithms, two of them are implemented in MapReduce. Google News recommendation architecture is based on three types of servers. First, the MapReduce cluster computes periodically the clustering of the users. Second, the statistics servers store user data in real-time. Last, the news front-end generate in recommendations on demand using the clustering previously computed in the MapReduce cluster.

The Ebay recommendation architecture [Katukuri et al. 2013] also consists of three main components. The data store is responsible for registering the user activity and the learned models. There is a MapReduce cluster that generates the computationally expensive models in an offline way. Finally, there is a component that computes personalised suggestions on demand using the data and models from the store component.

A very different approach to recommendation is the engine built with Apache Solr presented by Laci et al. [Lacic et al. 2010]. In contrast to our proposal where updates are not intended to be immediately processed, they are able to generate recommendations in real-time. This is an alternative for light and simple recommendation algorithms which can be modelled using the basic operations provided by Solr API. However, the major downside of this procedure is the difficulty of implementing complex recommendation algorithms such as the ones based on matrix factorization methods.

6 Conclusions and Future Work

In this paper, we have described a scalable architecture for big data recommendation systems without any single point of failure. This system consists of three main components: the web front-end, the data storage and the recommendation engine. Every subsystem of each component is fully distributed and replicated to achieve scalability and high availability.

We have studied the write scalability as well the read scalability of different storage technologies. We have compared two storage products, MySQL Cluster and Cassandra, for storing ratings and generating and serving recommendations, concluding that the second one is best suited to these tasks. At the end, we have employed a mixed solution constituted of MySQL Cluster, Cassandra and Lucene for the different types of data. Cassandra is used for storing ratings and recommendations, MySQL Cluster for web application data and Lucene for storing data about the items.

Additionally, we have included two levels of caching systems. On the one hand, we have proposed the use of a reverse proxy, a coarse-grain cache, using Varnish. On the other hand, the second layer of cache consists of an in-memory key-value store, Redis. We have tested the performance gains of using Redis in terms of response time of the web application. The results have shown important improvements either by caching only the database accesses or by caching also the web views.

Further research might benchmark more aspects of the architectural proposal. It would be very interesting to analyse the combined effect of both types of caching systems (Varnish and Redis). This study may help in determining which parts of the web application are worth of being cached.

Another possible area for further work is to analyse the use of the storage component that different recommendation algorithms make. In particular, we would like to analyse how effective probabilistic recommendation methods [Parapar et al. 2013, Valcarce et al. 2015] differ in the consumption of resources from the classical, less effective, models.

Acknowledgements

This work was supported by the *Ministerio de Economa y Competitividad* of the Spanish Government under the research project TIN2012-33867 and also by grants GPC2013/070, R2014/034 and R2014/002 from the Galician Government. The first author wants also to acknowledge the support of *Ministerio de Educacin, Cultura y Deporte* from the Spanish Government under the Grant FPU014/01724.

References

[Amatriain 2012] Amatriain, X.: “Netflix recommendations: Big data, smart models, scalable architectures”; GraphLab Workshop 2012; 2012.

[Anderson 2006] Anderson, C.: The Long Tail: How Endless Choice Is Creating Unlimited Demand; Hyperion, 2006.

[Baeza-Yates et al. 2007] Baeza-Yates, R., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., Silvestri, F.: “The impact of caching on search engines”; Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval; SIGIR ’07; 183–190; ACM, New York, NY, USA, 2007.

[Bai and Wu 2011] Bai, S., Wu, H.: “The Performance Study on Several Distributed File Systems”; 2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery; CYBERC ’11; 226–229; IEEE, 2011.

[Cattell 2011] Cattell, R.: “Scalable SQL and NoSQL data stores”; SIGMOD Rec.; 39 (2011), 4, 12–27.

[Chang et al. 2008] 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"; ACM Trans. Comput. Syst.; 26 (2008), 2, 4:1–4:26.

[Cortizo et al. 2010] Cortizo, J. C., Carrero, F. M., Monsalve, B.: "An architecture for a general purpose multi-algorithm recommender system"; Workshop on the Practical Use of Recommender Systems, Algorithms and Technologies; PRSAT 2010; 51; 2010.

[Das et al. 2007] Das, A. S., Datar, M., Garg, A., Rajaram, S.: "Google News personalization: Scalable online collaborative filtering"; Proceedings of the 16th International Conference on World Wide Web; WWW '07; 271–280; ACM, New York, NY, USA, 2007.

[Davidson et al. 2010] Davidson, J., Liebald, B., Liu, J., Nandy, P., Van Vleet, T., Gargi, U., Gupta, S., He, Y., Lambert, M., Livingston, B., Sampath, D.: "The youtube video recommendation system"; Proceedings of the Fourth ACM Conference on Recommender Systems; RecSys '10; 293–296; ACM, New York, NY, USA, 2010.

[Dean and Ghemawat 2008] Dean, J., Ghemawat, S.: "Mapreduce: Simplified data processing on large clusters"; Commun. ACM; 51 (2008), 1, 107–113.

[DeCandia et al. 2007] 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"; SIGOPS Oper. Syst. Rev.; 41 (2007), 6, 205–220.

[Gilbert and Lynch 2002] Gilbert, S., Lynch, N.: "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services"; SIGACT News; 33 (2002), 2, 51–59.

[Holovaty and Kaplan-Moss 2009] Holovaty, A., Kaplan-Moss, J.: The Definitive Guide to Django: Web Development Done Right, Second Edition; Apress, Berkely, CA, USA, 2009; 2nd edition.

[Katukuri et al. 2013] Katukuri, J., Mukherjee, R., Konik, T.: "Large-scale recommendations in a dynamic marketplace"; Proceedings of First Workshop on Large-Scale Recommender Systems; LSRS 2013; 2013.

[Lacic et al. 2010] Lacic, E., Kowald, D., Parra, D., Kahr, M., Trattner, C.: "Towards a scalable social recommender engine for online marketplaces: The case of Apache Solr"; Proceedings of the ACM World Wide Web Conference companion; WWW 2014; ACM, New York, NY, USA, 2010.

[Migliorisi 2014] Migliorisi, B.: "Reverse proxy performance - Varnish vs. Squid (part {1,2})"; http://deserialized.com/caching/reverse-proxy-performance-varnish-vs-squid-part-{1,2}/ (2014); accessed: 2014-03-12.

[Oracle 2012] Oracle: "Mysql cluster benchmarking"; Technical report (2012).

[Parapar et al. 2013] Parapar, J., Bellogín, A., Castells, P., Barreiro, A.: "Relevance-Based Language Modelling for Recommender Systems"; Information Processing and Management; 49 (2013), 4, 966–980.

[Rabl et al. 2012] Rabl, T., Gómez-Villamor, S., Sadoghi, M., Muntés-Mulero, V., Jacobsen, H.-A., Mankovskii, S.: "Solving big data challenges for enterprise application performance management"; Proc. VLDB Endow.; 5 (2012), 12, 1724–1735.

[Resnick and Varian 1997] Resnick, P., Varian, H. R.: "Recommender systems"; Communications of the ACM; 40 (1997), 3, 56–58.

[Ricci et al. 2011] Ricci, F., Rokach, L., Shapira, B., Kantor, P. B., eds.: Recommender Systems Handbook; Springer, 2011.

[Robenolt 2014] Robenolt, M.: "Scaling Django to 8 billion page views"; http://blog.disqus.com/post/62187806135/scaling-django-to-8-billion-page-views (2014); accessed: 2014-03-12.

[Sadalage and Fowler 2012] Sadalage, P. J., Fowler, M.: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence; Addison-Wesley Professional, 2012.

[Schafer et al. 2007] Schafer, J. B., Frankowski, D., Herlocker, J., Sen, S.: "Collaborative filtering recommender systems"; (2007), 291–324.

[Schelter et al. 2013] Schelter, S., Boden, C., Schenck, M., Alexandrov, A., Markl, V.: "Distributed matrix factorization with mapreduce using a series of broadcast-joins"; Proceedings of the 7th ACM Conference on Recommender Systems; RecSys '13; 281–284; ACM, New York, NY, USA, 2013.

[Valcarce et al. 2014] Valcarce, D., Parapar, J., Barreiro, A.: "When Recommenders Met Big Data: An Architectural Proposal and Evaluation"; Proceedings of the 3rd Spanish Conference on Information Retrieval; CERI '14; 73–84; 2014.

[Valcarce et al. 2015] Valcarce, D., Parapar, J., Barreiro, A.: "A Study of Smoothing Methods for Relevance-Based Language Modelling of Recommender Systems"; Proceedings of the 37th European Conference on Information Retrieval; volume 9022 of ECIR '15; 346–351; Springer, 2015.

[Wong 2014] Wong, D.: "Search traffic vs social referrals: How fast are they growing?"; https://blog.shareaholic.com/search-traffic-social-referrals-12-2013/ (2014); accessed: 2014-03-11.