When Recommenders Met Big Data
An Architectural Proposal and Evaluation

Daniel Valcarce, Javier Parapar, and Álvaro Barreiro

Information Retrieval Lab
Computer Science Department
University of A Coruña, Spain
{daniel.valcarce,javierparapar,barreiro}@udc.es

Abstract. Nowadays, scalability is a critical factor in the design of any system working with big data. In particular, it has been recognised as a main challenge in the construction of recommender systems. In this paper, we present a recommender architecture capable of making personalised recommendations using collaborative filtering in a big data environment. We aim to build highly scalable systems without any single point of failure. Replication and data distribution as well as caching techniques are used to achieve this goal. We suggest specific technologies for each subsystem of our proposed architecture considering scalability and fault tolerance. Furthermore, we evaluate the performance under realistic scenarios of different alternatives (RDBMS and NoSQL) for storing, generating and serving recommendations.

Keywords: Recommender systems, big data, scalability, architecture, NoSQL.

1 Introduction

During 2013, web traffic generated by search engines dropped 6% meanwhile that originated in the social networks increased more than 100% [25]. These figures point out the importance of the recommender systems in a landscape where users increasingly expect system suggestions instead of explicitly formalize their information needs.

Recommender systems [20] intend to predict items of interest for users without the need of an explicit request for information. The heterogeneous variety of scenarios make the task of providing with relevant items a non easy one. Different approaches have been proposed for this task being generally classified [4] in content-based methods [19] that exploit the similarity of candidate items with the ones already assessed by the user, collaborative filtering [22] techniques that exploit the information about the preferences of similar users to the subject of recommendation and hybridisations of both families.

In this work, although it is not devoted to the discussion of recommendation algorithms, we will work with a Collaborative Filtering (CF) method. CF is very popular in multiple recommendation scenarios because it is able to exploit

the preference patterns existing in any kind of community in order to provide with personalised recommendations. Most of the times, those users' preferences are explicitly presented in terms of rates by the users to the items. Nowadays, there exist several application domains for recommendation where the number of users, items and rates increased in a dramatic way turning the recommendation problem into a big data challenge. Web-pages, videos, friends or tweets recommendations are examples of this facet of the problem, but the number of scenarios reaching big data scales is increasing day after day.

In this context, the contributions of this paper are a detailed description of a system architecture capable of storing, processing and serving web-scale information for the purpose of developing a recommender platform in a big data context and the evaluation of the persistence layer under realistic circumstances. We evaluate two different families of storage systems, RDBMS and NoSQL, showing that a mixed solution fits our persistence needs.

2 Recommender System Architecture for Big Data

Traditional approaches to build recommender systems consider three different main components: a user interaction layer, a recommendation engine and a persistence component, every of which is subject to be either monolithic/centralised or distributed constructed. When scaling a recommender system to the big data landscape, any of those components is a potential point of failure in the architecture. Thus, our architecture proposal was designed to achieve two goals: high scalability and availability at every level. In addition, the infrastructure should be capable to store continuous updates of the user ratings history and provide high quality and fresh recommendations.

The overall design of the proposed recommendation platform is exhibited in Figure 1. We can easily distinguish three main components: the user interaction layer (front-end) provided by a web application, the recommendation engine using MapReduce framework, both of them consuming and saving information from/to the data storage component. In the following sections, we describe our proposal justifying the choices made in each component w.r.t. 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 should also provide users with recommendations. In order to allow them to perform complex search queries, we implemented faceted search. This technique gives to the users the opportunity to explore an item collection by applying several filters as a complement to the recommendations with the aim of increasing user rates. In fact, it has been reported [2] that 25% of played films in Netflix are not based on recommendations.

Guided by our goals (scalability and availability), we propose to deploy redundant web servers with a load balancer subsystem on top. Using a cluster of

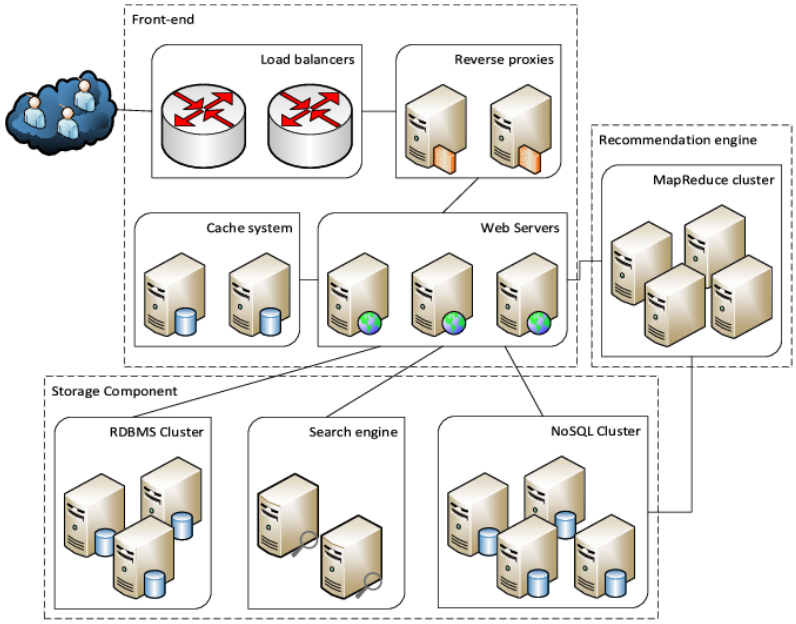

Fig. 1. Overall system architecture

web servers provides us with horizontal scaling (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[1] because it is a well-known tool that easily integrates with the technologies described below and it was successfully used in big data environments such as Disqus, Pinterest or Instagram.

For the load balancing task, we chose Perlbal[2] following the recommendation of Holovaty and Kaplan-Moss [12], but there are many other options available. Actually, a load balancer appliance (i.e., a device that implements load balancing in hardware) would be more efficient being the trade-off between inversion and desired performance the key factor in the decision. We need to make sure we use at least two load balancers in a failover configuration to achieve high availability.

In addition, we chose the integration of two levels of caching techniques in the current platform. The rationale for this decision is based on the fact that each request that hits the server produces some kind of computation and, probably, database accesses. In high-traffic sites caching is fundamental fto reduce server

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

load and avoid repetitive database queries. In our context, the most frequent case is users browsing their recommendations or looking for information about the last popular releases and, without caching, each time a user requests this data a relatively expensive query to the storage component would be processed. The success of such approach is a very well studied topic on Web Information Retrieval, where the so called *answer caching* [3] has been demonstrated as the most effective caching approach. As Disqus operation team reported [21], using a caching HTTP reverse proxy may greatly reduce the number of requests processed by the web framework and, consequently, the database. This kind of proxies are situated between the load balancer and the web server cluster and its aim is to cache responses to user requests in order to lessen web traffic to the servers. Nowadays, Varnish[3] and Squid[4] are two strong competitors in this field. We favour Varnish over Squid because it demonstrates better performance in benchmarks [15].

The second level of cache is used inside the web application for caching expensive calculations like the output of some views or the user sessions. We considered two alternatives: Memcached[5], a distributed memory object cache system, and Redis[6], an advanced key-value store. In spite of the fact that Redis have some features that Memcached lacks of (persistence to disk or collections), we decided to use Memcached because of out-of-the-box integration with Django and better support for cluster environments. However, we must point out that the Redis team is working to fully support clustering in the next mayor release [1]. Thus, probably Redis will be a more powerful choice than Memcached when clustering features are properly implemented.

2.2 Storage Component

Although Django, our web framework, can be used without any database, it is designed to write database-driven web applications. Our suggested recommender platform needs to store different types of information. On the one hand, we need to manage large amounts of user ratings and user recommendations. On the other hand, we also store items and web application data (such as user details). We studied three different approaches to address this problem: relational and NoSQL database systems and information retrieval structures.

Relational Databases: Relational database management systems (RDBMS) can be the ideal solution for storing data needed by the web application or even information about items. However, there may be performance issues for managing big data like user ratings and recommendations. Nowadays, Django supports natively four RDBMS, namely SQLite, PostgreSQL, Oracle and MySQL. The analysis of each solution is described in the next paragraphs.

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

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

PostgreSQL[8] is an object-relational database system. There exists some tools like pgpool-II[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[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[11] is also a object-relational DBMS like PostgreSQL. It offers an option called Oracle RAC (Real Application Clusters) for supporting 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) that become a single point of failure in the cluster. Furthermore, the usage of a SAN instead of a DAS (Direct-Attached Storage) is not generally utilised for big data analytics [24].

Finally, we analyse MySQL Cluster[12]. This product is based on a shared-nothing clustering architecture and it claims to be a 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.

NoSQL Databases: Nowadays, there exists multiple NoSQL solutions [5]. These systems do not use tabular relations like RDBMS and they claim being more scalable and flexible. There are different approaches to classify NoSQL datastores. We focus on column-oriented databases (also known as extensible record stores) because they provide high scalability and are well-suited for storing user ratings and recommendations.

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

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 NoSQL products generally used with Hadoop: Cassandra and HBase.

Apache Cassandra[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 read and write linear 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 [6] its data model and from Amazon Dynamo [11] its distributed architecture.

Apache HBase[14] is also a distributed and linear scalable DBMS that 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. Rabl et al. [18] 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. Moreover, only in a scenario where recommendations are calculated in real-time, eventual consistency could be a problem.

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 previous described database systems (either relational and NoSQL ones) are not well-suited for this task. Apache Lucene[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[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[17] is a modern distributed real-time search and analytics engine. Both of them supports faceted search and therefore are valid alternatives to address our needs. We chose Apache Solr because its mature and consolidated nature.

[13] `http://cassandra.apache.org/`
[14] `http://hbase.apache.org/`
[15] `http://lucene.apache.org/`
[16] `http://lucene.apache.org/solr/`
[17] `http://www.elasticsearch.org/`

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 [10], a functional programming model designed to treat large datasets in a distributed platform. We propose the use of Apache Hadoop[18], an open source implementation of MapReduce model, together with Apache Mahout[19], a machine learning library that contains different distributed algorithms built on top of Hadoop.

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 [9].

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 storing technologies, we decided to use the first algorithm, the least computationally expensive technique, because it gives us the opportunity of focusing in 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 comparison between Cassandra and MySQL Cluster.

Both data storing technologies have demonstrated linear scalability adding new nodes [18,16]. 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. Below, we introduce the applied methodology.

[18] `http://hadoop.apache.org/`
[19] `http://mahout.apache.org/`

3.1 Rating Insertion

We measured writing times of inserting film ratings from the Netflix Prize[20] using different number of concurrent connections. This dataset includes 100,480,507 ratings that 480,189 users gave to 17,770 films.

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 increases with the number of inserted ratings although a warm-up overhead can be observed at the start.

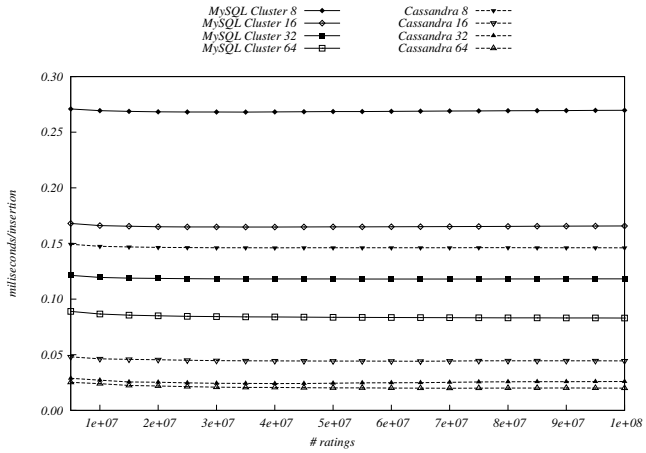

Fig. 2. Cassandra vs MySQL Cluster per insertion time when using 8, 16, 32 or 64 concurrent petitions and moving the chunk size of the ratings from 10 to 100 million users ratings. Times (in milliseconds) were obtained in a cluster of four nodes

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 which causes the database to crash. 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[21] to export the data from HDFS to MySQL.

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

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

The result 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 sorted recommendations in real-time, it is imperative to have an index on predicted score attribute. This action worsen MySQL Cluster scalability because this database solution stores indexed columns in memory. However, in Cassandra we only need to indicate the clustering order in the table creation process.

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.

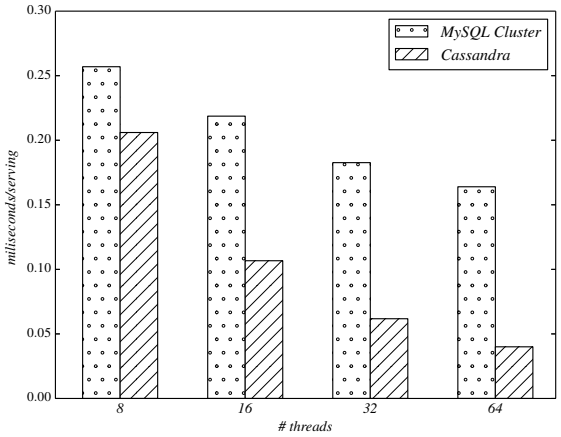

Fig. 3. Cassandra vs MySQL Cluster serving time per user recommendation when using 8, 16, 32 or 64 concurrent requests. Times (in milliseconds) were obtained in a cluster of four nodes

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 exceeds its main memory. For instance, our first attempt to compare MySQL and Cassandra involved using the Bigflix dataset [23], 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 [2] 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 Related Work

Scalability of recommender systems is not a novel topic. It has been counted as one of the most urgent challenges in recommendation on more than one occasion, for instance, Cortizo et al. [7] stated it as one of the main challenges of a general purpose multi-algorithm recommender. Of course, several companies have already addressed the scalability issues such as Hulu or Netflix. Unfortunately, given the strategic nature of such information, the reported details in their technical blogs are minimum and public comparison are missing. Only some short publications in the field of commercial recommender systems are available showing some of the concerns of those corporations about this topic.

In fact, our approach shares some similarity with Youtube's short description of their current implementation [9] where video recommendations are generated by pipelined calculations with MapReduce. Meanwhile user logs and score recommendations are stored in a Bigtable, a column-oriented database. Google News also uses Bigtable for storing user information [8]. They described three algorithms implemented in MapReduce based on an architecture with three types of servers. Nevertheless, both of them do not provide extensive descriptions.

Ebay recommendation architecture [13] consists of three main components: the data store that contains user data and learned models, the performance system that provide recommendations using Lucene and the offline model training with pipelined MapReduce jobs.

Another approach to recommendation is the engine built with Apache Solr presented by Lacić et al. [14]. 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 recent and very effective collaborative filtering algorithms using matrix factorization methods [17].

5 Conclusions and Future Work

In this paper, we have described a novel and scalable architecture for big data recommendation systems without any single point of failure. This system consists in three components: the front-end, the data storage and the recommendation engine. Every subsystem in each component is fully distributed and replicated to achieve high scalability and high availability. We have also suggested specific software solutions including NoSQL and several cache 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.

In future work, we seek to study and benchmark more aspects of the architectural proposal. In addition, we would also like to try more effective algorithms on the MapReduce framework [17].

Acknowledgements This work was funded by grants TIN2012-33867 and GPC2013/070 from the Spanish and the Galician Governments.

References

1. Redis cluster specification (work in progress). `http://redis.io/topics/cluster-spec`. Accessed: 2014-03-12.
2. Xavier Amatriain. Netflix recommendations: Big data, smart models, scalable architectures. In *GraphLab Workshop 2012*, 2012.
3. Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, pages 183–190, New York, NY, USA, 2007. ACM.
4. Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl, editors. *The Adaptive Web*, volume 4321 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
5. Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.
6. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.
7. Jose C Cortizo, Francisco M Carrero, and Borja Monsalve. An architecture for a general purpose multi-algorithm recommender system. In *Workshop on the Practical Use of Recommender Systems, Algorithms and Technologies*, PRSAT 2010, page 51, 2010.
8. Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google News personalization: Scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 271–280, New York, NY, USA, 2007. ACM.

9. James Davidson, Benjamin Liebald, Junning Liu, Palash Nandy, Taylor Van Vleet, Ullas Gargi, Sujoy Gupta, Yu He, Mike Lambert, Blake Livingston, and Dasarathi Sampath. The youtube video recommendation system. In *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10, pages 293–296, New York, NY, USA, 2010. ACM.
10. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
11. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.
12. Adrian Holovaty and Jacob Kaplan-Moss. *The Definitive Guide to Django: Web Development Done Right, Second Edition*. Apress, Berkely, CA, USA, 2nd edition, 2009.
13. Jayasimha Katukuri, Rajyashree Mukherjee, and Tolga Konik. Large-scale recommendations in a dynamic marketplace. In *Proceedings of First Workshop on Large-Scale Recommender Systems*, LSRS 2013, 2013.
14. Emanuel Lacic, Dominik Kowald, Denis Parra, Martin Kahr, and Christoph Trattner. Towards a scalable social recommender engine for online marketplaces: The case of Apache Solr. In *Proceedings of the ACM World Wide Web Conference companion*, WWW 2014, New York, NY, USA, 2010. ACM.
15. Bryan Migliorisi. Reverse proxy performance - Varnish vs. Squid (part {1,2}). `http://deserialized.com/caching/reverse-proxy-performance-varnish-vs-squid-part-{1,2}/`. Accessed: 2014-03-12.
16. Oracle. Mysql cluster benchmarking. Technical report, 2012.
17. Javier Parapar, Alejandro Bellogín, Pablo Castells, and Álvaro Barreiro. Relevance-based language modelling for recommender systems. *Inf. Process. Manage.*, 49(4):966–980, July 2013.
18. Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proc. VLDB Endow.*, 5(12):1724–1735, August 2012.
19. Paul Resnick and Hal R. Varian. Recommender systems. *Communications of the ACM*, 40(3):56–58, March 1997.
20. Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors. *Recommender Systems Handbook*. Springer, 2011.
21. Matt Robenolt. Scaling Django to 8 billion page views. `http://blog.disqus.com/post/62187806135/scaling-django-to-8-billion-page-views`. Accessed: 2014-03-12.
22. J. Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. Collaborative filtering recommender systems. pages 291–324, January 2007.
23. Sebastian Schelter, Christoph Boden, Martin Schenck, Alexander Alexandrov, and Volker Markl. Distributed matrix factorization with mapreduce using a series of broadcast-joins. In *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pages 281–284, New York, NY, USA, 2013. ACM.
24. John Webster. Storage area networks need not apply. `http://news.cnet.com/8301-21546_3-20049693-10253464.html`. Accessed: 2014-03-14.
25. Danny Wong. Search traffic vs social referrals: How fast are they growing? `https://blog.shareaholic.com/search-traffic-social-referrals-12-2013/`. Accessed: 2014-03-11.