Estimating Cache Benefits for NoSQL Databases

Savannah Wheeler and Dr. Dilma Da Silva
Texas A&M University

Abstract. Caching is a very well understood concept in computer science, with implementations that resulted in significant efficiency improvements in the realm of computer architecture, operating systems, database systems, storage systems, and distributed systems. Our general research goal is to assess how existing caching policies behave in the context of cloud-based data services for Internet-of-Things. During this summer project we took initial steps towards a benchmarking infrastructure to enable such caching studies. We implemented a deployment scenario to determine experimentally which type of distributed NoSQL database connection would be more efficient. We experimented with databases hosted on the local server and on servers accessible through the local area network (LAN). From this experimental framework, our tests can be extended to contrast the behavior between local databases and databases hosted remotely, for example on a public cloud infrastructure. For database backend technology we used a popular data storage engine, MongoDB. Using Eclipse, we built synthetic workloads of database insertion and query operations. Under this simple (and unrealistic) workload, we carried out experiments that show an average insertion time of 3652.2 milliseconds and an average query time of 136 milliseconds for locally-hosted databases. The remote scenario had an average insertion time of 4126.5 milliseconds and an average query time of 209 milliseconds. Such results are counterintuitive and indicative of errors in our benchmarking infrastructure. We are currently refining our experimental test to fix problems and achieve reliable experimental data.

1 Introduction

With the rise of smartphones and the use of apps, along comes the massive amount of data requests needed in order to offer users the functionality they want. Currently, software developers consider when to keep the data local on the mobile device and when to go to an external server ("the cloud") to retrieve or store it. Many factors influence this decision, such as impact on battery life, data plan usage, and tolerance for communication delays. The goal is to achieve efficiency without compromising user experience.

The current proliferation of wearable technology and Internet-of-Things (IoT) devices present to the application designer a novel scenario where, again, efficiency is at premium. Existing services tend to keep the data repository on the cloud, achieving simplicity of access and reducing the requirements on the device hardware by not including storage capabilities.
Our work aims at assessing the impact of keeping some data from the wearable and IoT devices cached locally — either at the device or or at another locally available hardware — on the performance of data access operations.

Our overall purpose is to determine if a combination of locally-hosted caching storage and remotely-hosted database can improve data access costs. This report describes our experience over the summer with using NoSQL databases as a backend to the data and measuring database

operating costs in terms of latency and bandwidth consumed. We describe our experience with a specific NoSQL database (mongodb), our learning in terms of caching policies, and our planned implementation of an LRU caching policy [3].

2 Background

Caching has proved itself essential for many forms of computing. When computers were slower, the speed in which your favorite webpage was delivered did not matter, as long as it showed up. In modern times, computing is all about the speed in which underlying computations happen. One way to improve performance in every computer is to accelerate memory access by adopting caching. Caching, in its simplest sense, is a component used for storing small amounts of data so that future requests can be handled quickly and efficiently [4]. In the context of CPUs, this means that the processor can get to some data orders of magnitude faster, without having to generate requests to the main memory. In the context of storing data in disks, without caching data requests would need to make their way to the hard drive all the time, never escaping from the large latency involved in storage technology. In the context of accessing web pages in the Internet, cached data is stored in files on the computer issuing the request or at other computers in the infrastructure, so that whenever a previous request is seen again — let's say for a web page visited recently — the data on that page is loaded very quickly because either it was temporarily stored in local file system therefore incurring no network access or it was stored at a nearby computer, saving additional network hops and congested network areas. Many caching systems exist to implement easy data access, exploring different caching policies that proved valuable for a specific domain. Caching policies like Least Recently Used (LRU), Least Frequently Used (LFU), or Most Recently Used (MRU), all use the similar model of keeping small amounts of data for quick queries [6]. They vary on the way that they handle the fact that the cache is not big enough for everything, and therefore at some point in order to add a new element to the cache, it is necessary to evict an existing entry. Our goal is to experiment with the most commonly used caching policies, but for the purpose of this summer project we focused on LRU. Such broad experimentation with caching policies represents the first step in determining how existing caching policies behave for cloud-hosted bases. Future work is needed to derive a conclusive view.

3 The Learning Approach

The tasks necessary for this project required familiarity with many computer science topics usually covered as 300 or 400 level classes. The student in this DREU project (the first author in this report) has just finished her freshman year, so an important aspect of the project was the practice of learning just enough about an established area in order to enable forward progress. The learning began with reading about standard relational databases [5], i.e., typical database systems, like MySQL [7]. From there, we looked into the concepts involved in a NoSQL cloud-hosted database, focusing on the widely deployed MongoDB [1]. These activities involved learning basic operations in the Standard Query Language (SQL), resulting a better understanding of how general databases worked. After the table-based system and the basics of SQL were grasped, learning about a document-based data model, as available in MongoDB, was essential. Learning the model behind the cloud system, and the beginning CRUD (create, read, update, and delete) operations in the MongoDB shell were needed in order to eventually learn

how to use the database in an application developed in Java using the Eclipse Integrated Development Environment (IDE) [2].

The next part of the work began in a very tricky manner. MongoDB has many drivers that can be installed and used in various languages and IDEs. Out of the many drivers they have, for use with Java we needed the MongoDB 3.0 java driver for the Eclipse IDE. This driver turned out to be extremely verbose in its language, and one of the least used of all of the drivers available in the MongoDB community. Since the driver had been recently updated, documentation on this driver was hardly existent, deprecated code was everywhere, and finding ways of using CRUD operations via the Java language proved to be difficult and very error prone. Nevertheless, we succeeded in developing a Java tool that generated synthetic data to populate the database and synthetic queries to put the database to work. We also succeeded in deploying this tool with database connections to locally- and remotely-hosted MongoDB instances. (We only experimented with local-area connections, but the same code should work with MongoDB instances running on the cloud.)

We assumed for our study a very simple data model, with entities in the database having attributes such as age, names, and phone numbers. We planned to include also images as attributes — enabling larger variation on the data size of elements — but we did not get to it during the summer.

In order to facilitate sharing code between us, we created a Github [8] repository. The integration of Github as the version control system for Eclipse had problems and troubleshooting it took more time than we expected.

With the environment set, we extended out in order to evaluate the generation of randomized data. An initial user search through the database was implemented and two variations, inserting the data and querying the data, were implemented as well. Coding the search aspect of the program required more time than expected due to the lack of experience with databases: a student who recently learned how to search in an array tends to try to implement the search operation instead of using the functionality already available in the database. Implementing a simple array search versus implementing a user ran search through a MongoDB database using the Java driver are on opposite ends of the programming spectrum. However, like all roadblocks, this was overcome through more investigation, learning, and trying out code. The other operations (data insertion and query) didn't take too long to be implemented as they built on the experience implementing the user search feature. When it became apparent that the the program needed to be automated (enabling execution of benchmarking activities without any user intervention), we simplified the infrastructure by not including the search feature. Instead, we ran a number of iterations (k) for both the queries and insertions to generate experimental results. After running multiple iterations, deciding on 100,000 for k seemed to be viable. These iterations were to be measured, and for the use of this experiment we measured the time taken for these two iterations in milliseconds and implemented an average.

Once a connection to the local MongoDB database was established (using an operation within the Java environment that connected the program to the local database) initial experimental results were obtained. Over 100,000 iterations, results indicated an average insertion time

of 3652.2 milliseconds and an average query time of 136 milliseconds for locally-hosted databases. The remote scenario had an average insertion time of 4126.5 milliseconds and an average query time of 209 milliseconds. However, the results for the querying of data appear to be significantly off of what we thought would be observed. Further experimental calculations and changes on how time is measured are being implemented. With these changes, we expect to achieve properly designed experiments.

4 Conclusions and Future Work

During this research project, several aspects of how data is stored and retrieved from databases have been examined. We made very good progress in developing an infrastructure to execute experiments and observe their performance behavior. We have identified the questions to be addressed regarding caching, but unfortunately we stopped short from implementing a cache policy and assessed its behavior.

At this point of our research, the question is: how do these caches and a cloud-hosted database relate? As described earlier, performance requirement in many scenarios makes it necessary to have a cache in the system, often both in terms of hardware support for caching and software data structures that explore a caching approach to improve system performance. In mobile environments, the value of caching data in the device has been demonstrated repeatedly to be effective for many applications. It is well understood that if an app were to constantly request access to data residing the cloud, it could end up costing much more than you have to spend: operations would take longer, likely compromising user experience; they would use more network data access, possibly incurring data plan costs; and they would be consuming more battery due to the increased usage of the network hardware. On the other hand, doing everything locally comes with risks to data integrity and challenges in keeping the local data synced with more recent data stored in the cloud. Therein lies that there has to be a happy medium, one where the database and the cache can be in sync to optimize the costs. With that in mind, as future work we intend to determine the cause of the erroneous query numbers in the program and to implement a caching policy, most likely LRU, to determine what combination of a cloud-hosted database and a caching policy would obtain with collecting more information in addition to time collection. I intend to apply other tests including a test for bytes used for insertions and queries, as well as a test for the cost of energy. Finally, I intend to create to a non-LAN remote connection to MongoDB and apply theses tests to it as well.

It is noted that the program defined in this paper will evolve as research continues. The latest program can always be obtained from:
https://github.com/savannahw/DREU2015/tree/master/DREU15/src.

References

[1]MongoDB. (2015). Retrieved August 1, 2015, from MongoDB:
http://docs.mongodb.org/manual/tutorial/
[2] Foundation, T. E. (2015). *Eclipse*. Retrieved August 1, 2015, from Eclipse.org:
http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/keplersr1

[3]Panchekha, P. (2012, October 16). *Tech Blog*. Retrieved August 1, 2015, from https://blogs.dropbox.com/tech/2012/10/caching-in-theory-and-practice/
[4]Rouse, M. (n.d.). *Tech Target*. Retrieved August 1, 2015, from http://searchstorage.techtarget.com/definition/cache
[5] Rouse, M. (n.d.). *Tech Target*. Retrieved September 7th, 2015, from http://searchsqlserver.techtarget.com/definition/relational-database
[6]Wikipedia. (n.d.). *Cache Algorithms.* Retrieved September 7th, 2015, from https://en.wikipedia.org/wiki/Cache_algorithms
[7] Heng, C. (n.d.). *The Site Wizard.* Retrieved September 7th, 2015, from http://www.thesitewizard.com/faqs/what-is-mysql-database.shtml
[8] Wheeler, S. Da Silva, D. (2015) *Github*. Retrieved September 7th, 2015, from https://github.com/savannahw/DREU2015/blob/master/DREU15/src/RandomData.java