Scaling User Preference Learning in Near Real-Time to Large Datasets

Ian Beaver and **Joe Dumoulin**
NextIT Corporation,
421 W. Riverside Ave, Spokane WA 99201 USA
{ibeaver,jdumoulin}@nextit.com
http://www.nextit.com

Abstract

In previous research we have shown the architecture and application of a case-based reasoning (CBR) system used to discover user preferences in an existing mixed-initiative dialogue system. In this paper we apply this CBR system to increasingly large datasets to test its ability to maintain near-real time performance in generating new user preferences. We also propose possible future applications of the system.

Introduction

Next IT is a company in Spokane WA, USA that builds natural language applications for the worldwide web and for mobile devices. As a way to increase user satisfaction and reduce the number of turns required to complete tasks by returning users, we developed a scalable system based on MapReduce for learning user preferences from past experience in near real-time. In a prior publication (Beaver and Dumoulin 2013) we describe the architecture and operation of this system as well as provide some preliminary performance testing results. We refer to this CBR system as the Learned Preferences Generation Services (LPGS) in the prior paper as well as this one.

Since the publication of that paper, MongoDB, the data store we chose to implement the system on, has switched to using the V8 JavaScript engine internally (MDB 2013). Thus the developer preview version we originally tested, 2.4, is now stable and in production. This allows us to test the scaling performance of the LPGS on a multi-threaded MapReduce engine, to see how it handles larger dataset sizes that we would expect to see in a real world deployment.

In the next section we briefly review the LPGS design and application, all of which is covered in more detail in the original publication. We then cover the testing and performance of the system, followed by some future applications we hope to soon support.

System Overview and Application

The LPGS was implemented using a CBR approach due to the fact that we are attempting to partially automate a conversation on behalf of a returning user leveraging specific knowledge of previous conversations with the same user. This is a key differentiator of CBR from other major AI approaches that focus more on drawing generalizations and associations from data and then applying them to specific cases (Aamodt and Plaza 1994). By looking at specific instances in a user's history and reusing that information to minimize the number of steps required for the user to repeat the same tasks we can make the Natural Language System (NLS) more efficient and increase the user satisfaction over time.

The CBR system architecture we designed is made up of a Data Store, the LPGS, and a Search Service. The Data Store is implemented in MongoDB, chosen for its schema flexibility (Berube 2012) and its ability to easily scale as the number of cases increases (Bonnet et al. 2011) while also providing a built-in MapReduce framework eliminating the need to deploy a separate MapReduce system. The Data Store contains:

User inputs for analysis (case memory). These are individual task-related user interactions with the NLS and include the user input text and meta data such as input means, timestamp, and the NLS conversation state variables.

Learned preferences (case-base). Rules created from successful cases that have been reviewed and retained for use in future cases.

User defined settings. Settings such as if the use of preferences are enabled for a user, and per user thresholds of repetitive behaviour before creating a preference solution.

The Search Service is implemented as a lightweight HyperText Transfer Protocol service that translates requests for prior learned behaviours from the NLS into efficient queries against MongoDB and returns any matching cases.

The LGPS is implemented as a pipeline of MapReduce jobs and filter functions. The inputs are the users conversational history over a specific prompt in a specific task from case memory, and outputs are any learned preferences, which we refer to as *rules*, that can be assumed for that prompt. A prompt by the NLS is an attempt to fill in a *slot* in a form of information needed for the NLS to complete a specific task. By pre-populating slots with a users past answers, a task can be completed faster and with less back and forth prompting and responding with the user.

This MapReduce pipeline consists of two jobs. The first MapReduce job, as seen in Figure 1, compresses continuous user inputs that are trying to complete the same slot within

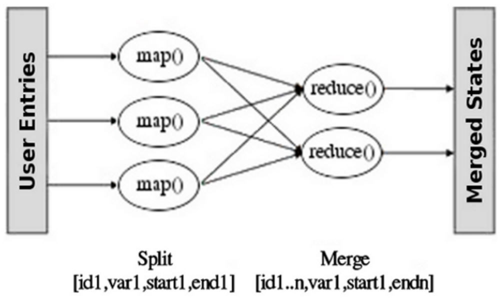

Figure 1: First MapReduce Job compresses prompted and satisfied states into single cases

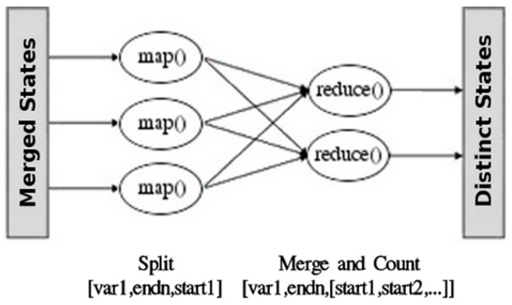

Figure 2: Second MapReduce Job groups all cases together with the same ending context and counts them

the same task. It may take the user several interactions with the NLS to resolve a specific slot since the user may give incorrect or incomplete data, or respond to the system prompt with a clarifying question of their own.

The compression is done by keeping track of when the user was first prompted for the slot, and when the slot was either filled in or abandoned. If the slot is eventually filled in, the prompting case's slot value and starting context are combined with the final case's slot value and ending context. If the slot is never satisfied the conversation is thrown out as there is no final answer to be learned from it.

When this first job completes, the compressed cases are stored along with the cases where the slot was resolved in a single interaction. As shown in Figure 2, the second MapReduce job is then started on the first job's results. This job attempts to count all of the slot outcomes for this user that are equivalent. First by grouping all of the cases by end context and then merging them into a single case, containing a list of all starting states that created the end context.

These final cases are passed through a set of functions that determine if the answer was given often enough to create a rule based on the users current settings. Cases that meet these conditions are saved in the case-base as a learned preference.

This pipeline is applied to a user's history any time the user has a new interaction with the NLS containing tasks that have 'learnable' slots or whenever a user changes their thresholds in the application settings. Domain experts defining the set of tasks in the NLS also define slots within those tasks that may be learned. Certain slots should never be saved as cases for learning such as arrival or departure dates or the body of a text message.

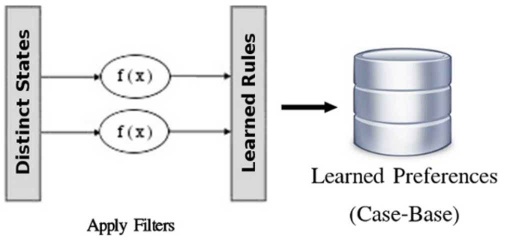

Figure 3: Finalize functions filter out any distinct states that do not meet the users personal repetition thresholds

Testing and Performance

System Evaluation

The primary measure of success for the learning system is reducing the number of steps required for the user to complete a task in the future. To evaluate this measure we needed to ensure that when a user repeats a task as many times as needed based on their settings, a rule is created and that rule is found on the next attempt to complete the task. The evaluation was done following these steps:

1. Create a new user account
2. Choose custom threshold settings or use system defaults
3. Walk through a task in the system conversationally
4. Repeat the conversation enough times to meet the set thresholds
5. Assert that on the next attempt to complete the task a prompt to validate a learned preference appears
6. Assert that on the next attempt to complete the task no prompt appears but the task is completed using the learned preference

Once the system was shown to be working correctly for a single user, we released access to the UI in the form of a mobile application to a limited group of 35 testers. The testers had the ability to enable and disable the use of the learned preferences during their conversation to compare the change in experience. In our limited release testing user feedback was very positive. One user commented that "Using the application without learning enabled is annoying", compared to the experience with it enabled. This was due to the decrease in prompting by the NLS on repeat uses with the learning system enabled. An example of a conversation collected from this evaluation is shown in Table. 1.

System Performance

One of the goals of the system is that the creation of new preferences for a specific user must happen in near real-time from when a user input is received. The definition of near real-time in this context is driven purely by user experience.

	Initial Conversation(s)
User:	text Fred
NLS:	I found more than one phone number for 'Fred'. Which do you mean? Home - ### Mobile - ### Work - ###
User:	Home number
NLS:	What do you want the text to say?
	Verification of New Rule
User:	text Fred
NLS:	I notice that in the past when you've asked me to send a text message to Fred you chose to use the phone number ###. Would you like me to assume you mean ### now and in the future?
User:	Yes.
NLS:	Great! I'll make a note of it. What do you want the text to say?
	Rule in Use
User:	text Fred
NLS:	What do you want the text to say?

Table 1: Conversations from the same user over time using the learning system.

Since the system allows users to tailor the number of repetitions required before learning an answer to a prompt, there is an expectation by the user that, for example when the number of times required is three, after the third time booking a flight it will not ask them for their departure airport that they have given the last three times in a row. If the system were then to ask them for that information, their expectations would not be met. In this example the definition of near real-time must be less than a realistic window of time before the user would repeat this task.

In this domain of booking flights, several hours may be an acceptable time frame since it is rare that users would book multiple flights in a several hour period leaving from the same airport. There may be domains where the same tasks are completed many times a day, as in a personal assistant domain where the user wants the system to learn that a nickname is associated to a specific contact they write text messages to often. In this domain the acceptable time frame may be only a matter of minutes. Therefore we recognize that since this acceptable time frame varies by domain and expectations of the user base, we can only show how the system scales within the limits of the testing hardware available to us and know there will be larger computing capacity needed to cover domains with fast preference availability expectations or large concurrent user bases.

Scaling Concerns A consideration in the initial system design was to make it easy to scale the system to meet the demands of an ever growing Data Store of user histories, and an ever increasing user base. To meet this need we selected the MapReduce programming model as it was designed to run on large clusters of commodity hardware and automatically partition the data across the machines (Dean and Ghemawat 2008). A motivation of MapReduce is to push the data closer to the processing. As the processing is distributed across commodity servers the data is distributed along with it, allowing the data size to continue to grow without greatly impacting performance. There are many different MapReduce engines available, Hadoop being perhaps the best known. It has been well proven in industry with Hadoop clusters over 5,000 nodes in size existing in production (Morgan 2013). However, as MongoDB includes a MapReduce engine, we use it instead of adding additional complexity by requiring an external engine such as Hadoop.

In this architecture, as the size of the Data Store grows, the load on an individual server can remain constant by simply adding more servers to the cluster and letting the data rebalance across them. MongoDB handles this data partitioning through a mechanism called *Sharding*, where a single collection of data is distributed across multiple servers or *shards* (MDB 2014). Each shard is an independent database that can execute MapReduce functions on its partition of the data. For example, if the Data Store contains 1 terabyte of data, and there are 4 shards in the cluster, then each shard only has to operate on 256GB of data. If there are 40 shards in the cluster, then each shard only needs to operate on around 25GB of data.

To test that the LPGS was capable of scaling to large numbers of cases, we needed to create a test data set in incremental sizes and show how performance degrades. We measure the average time it took to execute a single MapReduce job across the conversation history data, and the average time to analyse a single user for the complete set of tasks defined in the NLS for each case memory size. Since the LPGS only works on users that added new cases since the last time it ran, running against all users would be a test of the worse case scenario in the system.

Test Dataset Creation

In order to create large data sets for the purpose of load testing, we used actual conversations from users of an existing NLS in the personal assistant domain. These conversations were inserted into the case memory directly, truncated in a way that the number of inputs or cases per user would form a Normal Distribution where $\mu = 365, \sigma = 168$ with negatives remapped as

$$f(n) = \begin{cases} \chi & \chi \geq 1 \\ \chi \in 1 \leq \mathbb{Z} \leq 10 & \chi < 1 \end{cases}$$

Where $1 \leq \mathbb{Z} \leq 10$ is generated at random. This larger distribution between 1...10 is to simulate users that try the NLS out of curiosity with no intention of accomplishing any task and then abandon it. We chose μ and σ values based on projected usage expectation in the personal assistant domain after reviewing historical NLS usage in current production environments. We then simply marked all of the users as having new cases available, this way the LPGS would have to look at all users at the same time, creating the maximum load on the system given the size of the Data Store.

Testing Environment

The MongoDB cluster was constructed with 8 homogeneous servers with 2xE5450 CPUs, 16GB RAM and 2x73GB 15k rpm drives with RAID0. The system OS is Ubuntu Server 12.04LTS and the database and MapReduce system is MongoDB v2.4.5.

MongoDB was configured as 4 shards of 2-node replica sets. In the case memory and case-base collections, the ID was used as the shard key. In the user settings collection the UserID was used as the shard key. The LPGS was running on a workstation with an Intel i7-3930K CPU and 64GB RAM and was configured to use 32 worker threads, meaning 32 users' histories would be analysed in parallel. This number is configurable based on the computing power of the machine the LPGS is running on. Multiple instances can be started on multiple machines as well in order to reduce the total analysis time as the number of concurrent users grows.

Performance Results

The case memory was initially populated with 1,000 unique user histories. All users were flagged as having new data so that the LPGS would process the entire case memory as a test of a worse case scenario. The average MapReduce wall-clock time and average user analysis wall-clock time were then recorded. After each run more users were imported in case memory and the database cluster and LPGS were fully restarted in order to clear out any cached data that may skew the benchmarks. The next size tested was with 2,000 unique users, followed by 5,000, at which point the case memory was increased by 5,000 users each run. We stopped when we had reached 40,000 unique users, which comprised around 14.7 million cases, as we had reached the limits of the disk space available on our MongoDB cluster.

Anomalies The results of the testing can be seen in Figure 4 and Figure 5. In both figures, there can be seen an anomaly at case memory sizes of 5,000 users and 25,000 users. This is due to the fact that MongoDB's balancer uses by default a range function to partition the data across the shards. As the shard key used was the MongoDB *ObjectID* the most significant bits represent a time stamp. This means that they increment in a regular and predictable pattern.

This monotonically increasing number when inserted through the range function causes the inserted cases to be written into the same chunk of data, until the chunk size limit is reached and the chunk is split into two and moved (MDB 2014). Also as each user history was inserted sequentially when populating the case memory, the majority of a single user's history will reside in a single chunk on a single server. These issues create imbalances in the shard distribution where at certain data set sizes more chunks may reside on one shard than the others creating a disproportionately higher load on that shard and increasing the MapReduce execution times. For our use case it would be better in the future to use a hash function to partition the data as it would lead to a more random distribution of cases across the chunks and shards and a more predictable performance curve.

Memory Saturation The other finding shown in these figures is that around the 25,000 user mark, the average MapReduce time remains near constant for the rest of the case memory sizes. This is due to the fact that around the 25,000 user mark the memory is saturated on the shards and they must swap out to disk. At this threshold the performance of the system in the worst case scenario does not continue to degrade as the Data Store grows. This threshold could be raised using the same server configuration by adding more memory to the shards.

Conclusions Therefore we conclude that given the hardware configuration we tested, the worst it can perform is around 500 milliseconds per MapReduce job given that we are analysing 32 concurrent user histories. We can also conclude that adding more shards to the system would reduce the memory usage per shard and delay the point at which this threshold is reached. An added benefit of adding more shards as opposed to increasing existing shard memory is that the processing load would be distributed across more servers. What still needs to be evaluated is how the number of concurrent users analysed affect the performance degradation and worst case average MapReduce times.

MongoDB handles the load of 32 parallel MapReduce jobs on completely separate (meaning uncached) data very well. The total time it takes to process 40,000 users would be acceptable in most domains without needing to use multiple instances of the LPGS. The *Average User Analysis* time meets our definition of near real-time given the size of the data we tested even as a worse case scenario. In a real world case where 40,000 unique users need to be reviewed concurrently would in all likelihood mean there was a great deal more total users in the system.

Future Applications

Given that the system performance is acceptable and scalable, we have some ideas on using this same processing pipeline to further enhance user experiences.

Reasoning About Specific Users As CBR is a methodology for both reasoning and learning (Kolodner 1992) and we are primarily using it for learning, we could use the same data for reasoning about specific users as well. An example would be if a user has the use of learning enabled, but has rejected every potential new rule that has been found for a specific task, we may assume that this user does not want us trying to automate that task. This would allow the user to get the benefit of learned preferences in other tasks without the annoyance of occasionally invalidating new potential rules for a task they do not wish to automate.

Let Users Define Learnable Slots In the current system, the slots that are watched for learning are defined by domain experts when creating the set of tasks the system can preform. As a way to make the learning system more personalized, a user could add a slot to be learnable for them and select context to a rule from the available system context. This could be exposed through a UI element that is shown next to prompts that are part of a task. When the user clicks on the element, a dialogue could appear that lets them select which context elements they think are relevant to the answer.

External Information Sources A possibility we have considered in travel domains is to use data from external sources to take into account weather, delays, flight changes, and other travel info and look at how that affects use of the NLS by users. If we took into account this meta-data from sources outside of users, we could begin to predict usage spikes in the system when alerts like weather changes or flight delays are present.

Time and Location Awareness By adding time and locality information as features in the conversation state, we can look at what conversations are had during what times of day and in what locations. For example, a personal assistant application may notice that this specific user always listens to their Workout playlist in the gym at 9AM Monday through Friday. Therefore the system could learn that when the user wants to listen to music around 9AM on a weekday, and their current location is at the gym, it should just start playing their Workout playlist.

Conclusion

We have shown that our CBR system used to discover user preferences is able to scale to real world workloads and still maintain acceptable performance. In a worst case scenario the system was able to handle the growing case memory size up to the limits of its disk space. Given that our test cluster used 4 shards when MongoDB supports up to 1,000 (Horowitz 2011), we are confident that the system would continue to scale several orders of magnitude more than our test data size. We also proposed some ideas taking advantage of this scalable processing pipeline to leverage the same system to do more than automate conversational tasks for users.

References

Aamodt, A., and Plaza, E. 1994. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications* 7(1):39–59.

Beaver, I., and Dumoulin, J. 2013. Applying mapreduce to learning user preferences in near real-time. In *Case-Based Reasoning Research and Development*. Springer. 15–28.

Berube, D. 2012. Encode video with mongodb work queues. IBM.com. Available online at `https://www.ibm.com/developerworks/library/os-mongodb-work-queues/os-mongodb-work-queues-pdf.pdf`.

Bonnet, L.; Laurent, A.; Sala, M.; Laurent, B.; and Sicard, N. 2011. Reduce, you say: What nosql can do for data aggregation and bi in large repositories. In *Database and Expert Systems Applications (DEXA), 2011 22nd International Workshop on*, 483–488. IEEE.

Dean, J., and Ghemawat, S. 2008. Mapreduce: simplified data processing on large clusters. *Communications of the ACM* 51(1):107–113.

Horowitz, E. 2011. The secret sauce of sharding. MongoSF. Available online at `http://www.mongodb.com/presentations/secret-sauce-sharding`.

Kolodner, J. L. 1992. An introduction to case-based reasoning. *Artificial Intelligence Review* 6(1):3–34.

MDB. 2013. Performance improvements. MongoDB 2.4 Release Notes. Available online at `http://docs.mongodb.org/manual/release-notes/2.4/#v8-javascript-engine`.

MDB. 2014. Sharding and mongodb. MongoDB Documentation Project. Available online at `http://docs.mongodb.org/master/MongoDB-sharding-guide.pdf`.

Morgan, T. 2013. Big hadoop shops are on a hockey stick growth curve. EnterpriseTech Blog. Available online at `http://www.enterprisetech.com/2013/10/30/big-hadoop-shops-hockey-stick-growth-curve`.

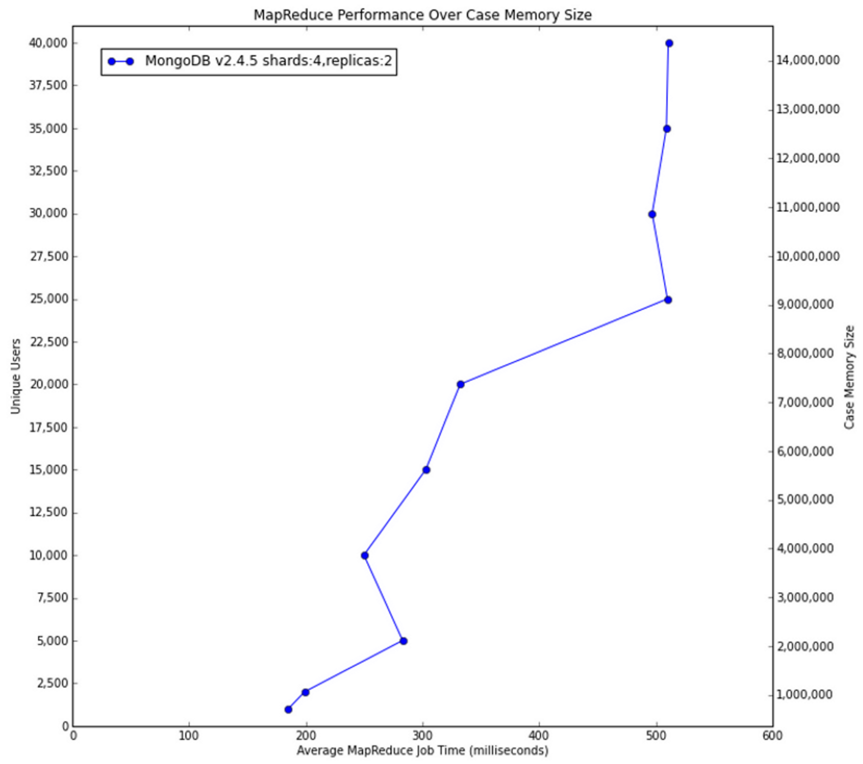

Figure 4: Average wall clock time for a MapReduce job to complete as total dataset size increases

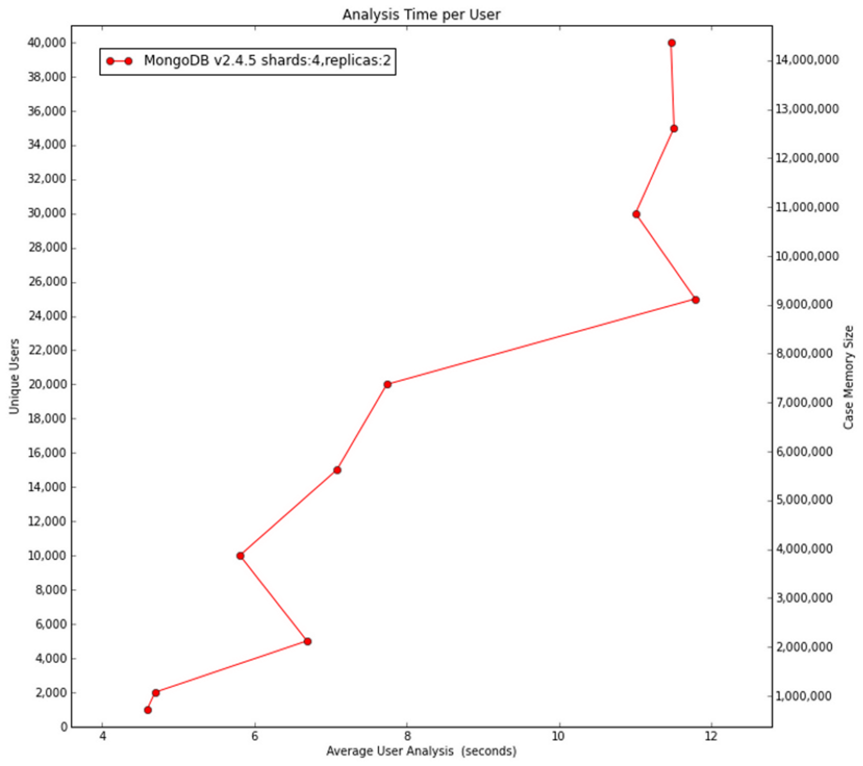

Figure 5: Average wall clock time to complete analysis over a single user's history as total dataset size increases