Applying MapReduce to Learning User Preferences in Near Real-Time

Ian Beaver and Joe Dumoulin

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

Abstract. When computer programs participate in conversations, they can learn things about the people they are conversing with. A conversational system that helps a user select a flight may notice that a person prefers a particular seating arrangement or departure airport. In this paper we discuss a system which uses the information state accumulated during a person-machine conversation and a case-based analysis to derive preferences for the person participating in that conversation. We describe the implementation of this system based on a MapReduce framework that allows for near real-time generation of a user's preferences regardless of the total case memory size. We also show some preliminary performance results from scaling tests.

Keywords: case-based reasoning, dialogue systems, natural language

1 Introduction

Next IT is a company in Spokane WA, USA that builds natural language applications for the worldwide web and for mobile device applications. Recently we have been working on features to improve the performance of machine directed conversations. This paper describes one of those enhancements, the development of a scalable system for learning user preferences from past experience in near real-time.

The system we developed is closely tied to our existing natural language system (NLS) so this paper will begin with a discussion of the behaviour of this system and how it interacts with people. The following sections discuss the architecture, operation, and performance testing of our CBR-based learning system.

1.1 User-Directed and Machine-Directed Conversations

The NLS in question enables designers to craft user-directed and machine-directed conversation templates. Users of the system can initiate these templates

** The final publication is availible at link.springer.com http://link.springer.com/chapter/10.1007\%2F978-3-642-39056-2_2

by starting a conversation with the system. The conversation can take the form of a chat or a task to complete. For example, we can assume that the application is running in an airline domain. A person can ask the system "how much does it cost to check an extra bag?" and the system may respond with a simple answer like "$10." This is a user-directed conversation.

In contrast, a machine-directed conversation turns the user's request into a task to be completed, and the system asks the user a series of subsequent questions to support that task. The system keeps track of what information has been gathered and what is still required to complete the task in a *form*, or conversation state object. For example, suppose the person wants to book a flight. They may start by saying "I need to fly to Seattle on Tuesday." The system can then ask a series of questions of the user to fill in the remaining information needed to complete the task.

There may be a validation step to perform for each form component, or *slot*, to ensure that the user has supplied valid information and to re-prompt the user in the case of invalid data. When the user has filled in all of the slots in the form through conversation, the system has enough information to complete the task by booking the flight the user requested.

To summarize, a user-directed conversation is conducted by the user asking questions of the system. A machine-directed conversation is conducted by a system asking questions of a user. Systems which perform both of these tasks are sometimes called mixed-initiative systems. Mixed-initiative dialogue systems have been proposed as far back as the 1970's[1] and have been shown to both perform well[2] and adapt well to different domains[3].

1.2 Machine-Directed Conversations and Personal Preferences

Designing a machine directed conversation includes a number of steps. The first step is creating the tasks the system will be capable of performing (defining the set of available forms). For each task in the system, the next step is determining the data needed to complete the task (defining the slots in the form to be filled). Once the task is well-defined, a final optimization step is looking at ways to minimize the number of questions the system must ask in order to complete the task. This is especially important if a person is expected to return to the system and complete this task many times, as may be the case when booking travel.

Something we noticed while investigating the need to minimize the number of questions being asked is that, for a given person, the answers to certain questions are consistently the same. For example, if a person is a frequent traveller and books flights through the system, there is a high likelihood that the flight's departure airport will be the home airport of the person. In another example, suppose the person prefers aisle seats. This will become evident after only a small number of bookings. We concluded that in order to shorten the task completion turns, we could learn a person's preferences based on previous conversations with the user.

The remainder of this paper describes the system that we constructed to learn user preferences in the scope of any machine-directed conversation and to make

that learned behaviour accessible to subsequent sessions with the same person. This learned preferences system can then be applied to any of the domains in which our natural language system is deployed. In Section 2 we outline the implementation goals and the component architecture used to achieve them. In Section 3 we describe how we applied case-based reasoning methods to address some of the implementation goals. Section 4 describes the testing methodology we used to ensure that our system could perform adequately when installed on production hardware and under load. Finally, Section 5 outlines our Conclusions.

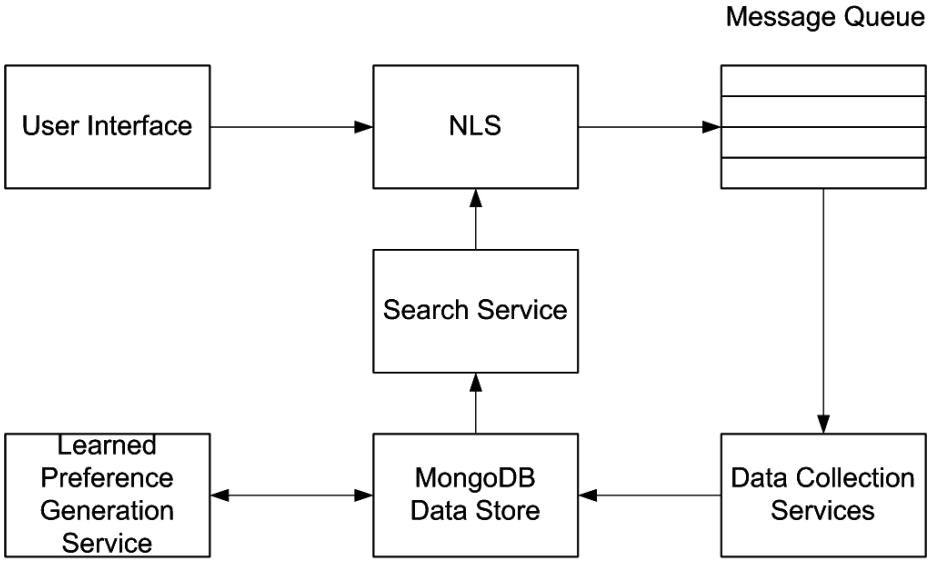

Fig. 1. A component view of the NLS and the CBR Learning system and the services that support it.

2 Implementation Goals and Architecture

There were several general properties that we hoped to obtain from adding a learning component to our existing NLS. We will later show how the case-based reasoning (CBR) system we developed satisfied each of these properties:

- Reduce the number of turns required to complete a task for a returning user, thereby increasing the user's satisfaction with the system.
- Allow the user to adjust the level of personalization the system displays to them.
- Store the learned preferences in a way that they can be quickly and easily retrieved and displayed to the end user.

- Create preferences from user input in near real-time.
- Confirm new preferences with the user to prevent unexpected user experiences.
- Allow users to change their preferences at any time.
- Scale easily as the number of users and input history grows.

Figure 1 shows a simplified component view of the entire NLS with the CBR Learning System. There are two integration points for the CBR Learning System:

The first is where the NLS queues user responses for the Data Collection Services to process and insert into the Data Store. The actual processing steps are performed in the Learned Preferences Generation Services (LPGS) outlined in Section 3.

The second area of integration is the Search Service. The NLS uses the Search Service to check for learned preferences, which we refer to as *rules*, that might be available to help derive information for the in-focus task. The NLS also retrieves rules to verify with the current user through the user interface. The user can then indicate whether or not the system should keep the retrieved rule.

The results of the LPGS processing is a set of rules that is placed in the Data Store. The Data Store contains:

1. User inputs for analysis (case memory) - Individual task-related user interactions with the NLS. Includes the user input text and meta data such as input means, timestamp, and NLS conversation state variables. The complete structure of each case is detailed in Appendix A.
2. Learned preferences (case-base) - Rules created from successful cases that have been analysed and retained for use in future cases. The complete structure of each rule is detailed in Appendix B.
3. 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.

3 Application of Case-Based Reasoning

The primary design idea for the learning system was to look at entire user histories and group their NLS interactions by task. If a user consistently provides the same value for a given slot, we can assume it instead of prompting in future conversations. This was a perfect fit for a CBR methodology since we are looking at specific instances in a user's history and reusing that information to solve a future problem, namely minimizing the number of steps required to repeat the task in the future.

Similar systems have been proposed to learn user preferences for the purpose of creating user recommendations [4]([5] compares many such approaches) or generating a user profile[6, 7]. The primary difference in our system is that instead of learning preferences for the purpose of filtering or ordering information presented to the user, we are attempting to anticipate responses to specific NLS

prompts. Another key difference is that we require these rules to be constructed in near real-time as the user is aware of exactly when a rule should exist based on their personal settings. Our system must be able to maintain this near real-time property as the number of users and cases grow, as we have many existing natural language systems deployed to large companies with very large numbers of users.

Taking the scale into consideration, we decided on a MapReduce programming model as it allows functions to be applied to large datasets in an automatically parallelized fashion[8]. As the dataset grows it can be partitioned across more resources as needed to maintain the near real-time requirement without additional complexity or code changes.

3.1 Case Structure

The major issue that had to be addressed was how to structure the case feature set so that it was general enough that unique information like timestamps and sequences would not prevent a match with a similar conversation, but specific enough to maintain a distinction between conversations with different outcomes.

We overcame this problem by requiring the task designers to specify the set of features in the conversation state that were important for each slot in a task. Since the task designer is the knowledge expert that is defining the ontology of the system, they know ahead of time what feature variables exist in the conversation state and which ones are important to the slot that is being prompted for. When the set of available forms is generated for the NLS system to consume, a separate file is created that defines these relationships, which the learning system uses to identify features for each case.

This solution can still be error prone when feature variable names are too generic or are reused for several different values. For example, suppose we are trying to learn a preferred email address for the user and the designated variable is named *EmailAddr*. If Sally were to ask the system to "Send an email to Fred", and the same variable name is re-used for both source and destination addresses, it can lead to a rule never being generated for Sally's preferred email address because the addresses contained in *EmailAddr* will never agree. After a few iterations of conversation template deployment and testing most of these variable name issues became apparent and were resolved.

3.2 Case Storage

Every task-related user interaction with the NLS is saved as a case. Simple user-directed interactions are not retained as they are not a multi-step process that we can attempt to optimize for repeat users.

MongoDB was chosen as the Data Store for its schema flexibility[9] and its ability to easily scale as the number of cases increases[10]. It also includes a MapReduce mechanism which allows for complex parallel searching of the case memory. Any MapReduce framework and distributed file system could have been

used for the learning system such as Hadoop or Disco, but we chose MongoDB as it supported our requirements in a single simple to deploy package.

3.3 Case Retrieval

When a user begins a task in the NLS, before the NLS prompts the user for each slot in the form (the new case to be solved), it first sends the new case features to the Search Service to see if there is an existing preferential value to place in the slot from the case-base. Fig. 2 shows an example of preferring learned preferences over asking the user.

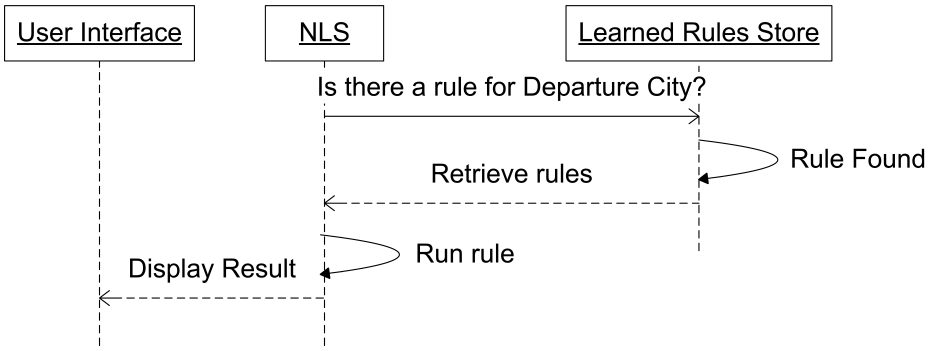

Fig. 2. Flow diagram for a portion of 'Book A Flight' dialogue. The system is resolving whether or not rules exist in order to avoid an explicit prompt for 'Departure City'.

The features of this new case are defined by:

1. A Universally Unique Identifier (UUID) generated for this new case.
2. The current UserID.
3. The array of variable-value pairs that make up the current conversation state, for example: [(IsLoggedIn,True), (LatLong,[47.656918,-117.418356])]
4. The active task the user has initiated in the NLS.

By ordering the array of context pairs by variable name, and then generating a SHA-1 hash of the string representation of the array, a search key is generated. Using this search key, the UserID, and TaskID, we query the case-base for an existing preference rule. If no preference is found in the case-base, the NLS will then prompt the user to fill in the slot.

3.4 Case Reuse

If a preference is found that is both active and valid, it will be used to fill in the slot value, and the NLS will then repeat this process to try and fill in the next slot in the form.

3.5 Case Revision

If a preference is matched and it has been inactivated, it will not be used. If it is active but has not been validated by the user yet, the NLS will ask the user if they would like to use the historical value from now on. If so, the learned preference will be validated and used without prompting the user in the future. Otherwise, it will be marked inactive and will not be returned by future queries. Fig. 3 shows how the verification event takes place during a conversation. This example was taken from a live system and shows how the user's repeat task is optimized by the learning system, satisfying our primary goal of reducing the number of steps for repetitive tasks. Note that the initial conversation will need to be repeated as many times as this specific user has defined in their preferences before a rule will be constructed.

Initial Conversation(s)	**Verification of New Rule**	**Rule in Use**
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?	**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?	**User**: text Fred **NLS**: What do you want the text to say?

Fig. 3. Conversations from the same user over time using the learning system.

3.6 Case Retainment: Learning Preferences from Case Memory

The NLS forwards all attempts to fill slots to the Learning System by placing them in the Message Queue where they will be stored in case memory by the Data Collection Service. The NLS does this even when a learned preference rule was used to fill the slot.

The LPGS receives events that new cases are available and does reanalysis on the affected user's entire case memory. The LPGS is looking for repetitive behaviours by the specific user that lead to the same outcome in completing a slot for a given task. This is accomplished by two MapReduce jobs applied to the user's history in the case memory collection.

First MapReduce Job The first MapReduce job compresses continuous user inputs that are trying to complete the same slot within the same task. It may take the user several interactions with the NLS to resolve a specific slot. 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. The system evaluates a few scenarios to determine the correct cases to use for the learning step.

In some scenarios, we know that the slot was not successfully completed because the NLS re-prompted the user for the same slot variable. Multiple prompts are merged into a single case. The earlier case's slot value and starting context are combined with the later case's slot value and ending context.

This reduction process continues until either 1) the slot was satisfied or abandoned and the NLS prompted for a new slot; or 2) no more cases are found in that sequence of interactions, meaning that either the user is currently in the middle of the conversation or they abandoned the entire conversation without completing the task.

When this first job completes, its combined results are stored with the single cases where the slot was resolved in a single interaction.

Second MapReduce Job The LPGS starts a second MapReduce job on the first job's results. This job attempts to count all of the slot outcomes for this user that are equivalent. The job first groups all of the cases where the end context was the same. When it finds two cases that meet this criteria, it checks to see if the final answers matched. If that is not the case, later analysis will be done to determine which answer (if any) is most appropriate to learn.

Finalize After this step completes, the LPGS starts a finalize function over the resulting groups. It filters out any group whose slot was never satisfied. For example, if the starting value for the slot is "null" and, after all of the attempts to resolve it, the value remains "null", we know that the slot was never satisfied.

The finalize step also filters out any groups where the count of observed cases is less than an adjustable threshold. This threshold is in place to define how many times a user needed to repeat a behaviour before it was considered a preference.

For example, the very first time a user completes a slot, it should not be considered a "preference" since there is not enough historical data to support that they will do it again. However, each user may have a different expectation of how many times they need to repeat an answer before it is saved.

Since one of our system goals is to allow a user to adjust the level of personalization, we exposed this threshold to them in a "Settings" screen within the application. This setting is saved with their user preferences and looked up when that user's inputs are analysed by the LPGS. The finalize function filters out any answers that do not meet the user's threshold for repeat behaviours. There

are pre-set defaults so that users are not required to configure the system before using it and we impose a lower bound of two, for the above mentioned reason. Currently this setting is used for all forms but we have considered implementing form "groups" of similar tasks and allowing users to control the settings per group. We do not expect that this would add significant load to the LPGS since it would only involve adding a group key to the user setting query that it is already performing. It would, however, add more complexity to the application interface, and possibly add confusion to users so more research needs to be done on the benefits of adding this feature.

Analysis of Results At this point we have a second temporary collection that contains cases where the slot was satisfied, and the count of how many times each answer was observed in the user's history. The LPGS now analyses these cases to determine if an answer can be assumed.

For the purposes of this analysis, the LPGS runs a set of functions over the data. Each of these functions can output rules, but they use different criteria to determine the rules. Currently, we have two functions in the set but we anticipate using more in the future.

The first function looks for a user-specified number of contiguous cases with identical values for the slot in a group. If found, a rule is created for the slot using the value of the case.

The second function checks for a percentage of final slot values to be the same. The specific percentage used is user-defined.

The value of this second function is that a user does not have to repeat the same answer many more times if they had a single different answer for some reason. For example, consider a frequent traveller. Most of the time they book a flight from their home airport, but sometimes they are in a different city and they book a flight leaving from there. We can still learn which airport they prefer since the majority of the time they choose their home airport.

More specifically, suppose they have their percentage set to 75% and the sequential threshold set to 3 for example. If their answers for the DepartureAirport slot from case memory ordered by time looks like [SEA,SEA,LAX,SEA], we can still assume that they prefer to leave from Seattle, even though the sequential function could not assume that.

After a function analyses the composite cases, any new preferences found by it are saved into the case-base. New preferences will be verified or ignored the next time the user initiates the task. The two temporary collections are then deleted and the MapReduce jobs are applied to another slot in the user's task history. This re-analysis process is completed for each user that has added new cases since the last time the LPGS fetched updated users.

4 Testing and Performance

4.1 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 the same way their configured amount of times, 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 was shown in Fig. 3 above.

4.2 System Performance

The system was functioning as intended, but we had to ensure that the solution would be able to scale to a production capacity. Since the majority of the analysis work is done within the MapReduce jobs, the ability for the LPGS to scale is closely tied to the ability for the MapReduce engine (MongoDB) to scale. 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. There is an expectation by the user that, for example, 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. This would happen if this user has their learned threshold setting at three. 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 performs with the testing hardware available to us and know there will be larger computing capacity needed to cover domains with fast preference availability expectations.

To test that the system 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 how long it took to analyse a single user for the set of tasks in the NLS and the total time it took the system to analyse all users in the data set 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.

4.3 Test Dataset Creation

In order to create a data set to test with, 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 $1 \ldots 10$ 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. A custom Data Collection Service was used that simply marked all of the users as having new cases available instead of reading off the Message Queue and marking users with queued messages. This way the LPGS would have to look at all users at the same time, creating the maximum load on the system.

4.4 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 v2.2.3-rc1.

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 learning service itself 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 would be worked on in parallel. This number is configurable based on the computing power of the machine the learning system is running on. Multiple instances of the learning system can be started on multiple machines in order to reduce the total analysis time.

4.5 Performance Results

Table. 1 shows the results of running the LPGS against all of the users in the database. After each run more users were imported in case memory and the database cluster was fully restarted in order to clear out any cached data that may skew the benchmarks. The *Total Cases* column shows the number of cases in the case memory collection, all of which would have to be looked at if all user histories are analysed. The *Analysis Time* column is the wall-clock time from when the LPGS started to the time it completed the last user. The *Avg. User Time* column is the wall-clock time it took to complete all of the historical analysis on a single user.

Table 1. Performance (MongoDB v2.2.3-rc1)

Total Cases	Users	Avg. User Cases	Analysis Time (H:M:S)	Avg. User Time (S)
369,536	1,000	369	0:07:22	2.839
743,719	2,000	371	0:14:39	6.279
3,622,196	10,000	362	0:47:12	8.416
7,200,767	20,000	360	1:34:00	8.407

MongoDB handles the load of 32 parallel MapReduce jobs on completely separate (meaning uncached) data very well. The total time it takes to process 20,000 users would be acceptable in most domains without needing to use multiple instances of the LPGS. The *Avg. User Time* meets our definition of near real-time given the size of the data we tested with. This is also testing the absolute worst case, in a real world case where 20,000 unique users would need to be reviewed every 1.5 hours would in all likelihood mean there was a great deal more total users in the system. Notice that the *Avg. User Time* does not change when the case memory size is doubled from 3.6M to 7.2M cases. Once MongoDB has reached a stable load, the job time appears to flat line, at least until the indexes would no longer fit in memory[11]. Further testing is needed with larger data sets to know if this assumption holds true and where the limitations of our testing hardware are.

In order to compare how the performance changes with a different M/R engine, the same cluster was rebuilt using a developer preview version of MongoDB 2.4 (2.3.2) using the V8 JavaScript engine[12], which supports multi-threaded MapReduce execution compared to the single-threaded execution in the SpiderMonkey engine used in versions 2.2 and below. This release proved to be still unstable, with several memory leaks observed and occasional long pauses in MapReduce job execution. In spite of this, as Table. 2 shows[1], the results are very promising as the average user analysis time was sped up between 50-62%[2].

Table 2. MongoDB v2.3.2 Performance

Total Cases	Users	Avg. User Cases	Analysis Time (H:M:S)	Avg. User Time (S)
754,748	2,000	377	0:12:44	2.364
4,130,057	10,000	413	0:25:50	4.175

5 Conclusion

We have shown how we applied CBR to the problem of automatically learning user preferences for repeat users. We have also shown how this system satisfied all of our initial goals, as well as shown that it has the ability to scale very well to millions of cases and tens of thousands of users. Given that our test cluster used 4 shards when MongoDB supports up to 1,000[13], we are confident that the solution we presented would continue to scale several orders of magnitude more than our test data size.

[1] The apparent discrepancy on Table. 2 in the 2,000 user test between the *Avg. User Time* and the *Analysis Time* appears to be due to one of the jobs hanging for several minutes before returning. This seemed to be repeatable but since this was a developer preview version of MongoDB and the job did eventually return successfully it was not a cause for concern, other than making an inconsistent benchmark.

[2] Because of the stability issues, we were not able to reproduce the 20K data set size to compare.

Appendix A Case Memory Document Structure

ID - The case identifier.
UserID - System wide unique user identifier.
PreviousPrompt - The slot variable the user was previously prompted for.
JustPrompted - The slot variable the NLS just prompted the user for after their answer.
Context - A JSON object holding pairs of conversation state variable names and values at the time the user was prompted for the slot variable.
SearchContext - A case-normalized form of the **Context** stored as an array of [name,value] pairs sorted by variable names.
Answer - The value of the slot variable named by **PreviousPrompt** that was filled in from the users input.
Order - Sequence number used to order inputs for this user.
TimeStamp - Time from web server when user input occurred.

Appendix B Case-base Document Structure

ID - The learned preference identifier.
Active - Flag used to determine if this specific preference is available for use.
Prompt - The slot variable name the user was prompted for.
EndContext - A JSON object holding pairs of conversation state variable names and values representing the state of the system after the user had successfully filled in the slot variable contained in **Prompt**.
Type - The function type that discovered this learned preference.
UserID - System wide unique user identifier.
Verified - Flag used to determine if the user has verified that this preference is acceptable.
StartContext - A JSON object holding pairs of conversation state variable names and an array of values observed at the time the user was prompted for the slot variable contained in **Prompt**.
SearchKeys - Array of SHA-1 hex strings computed for each combination of name:value pairs in **StartContext**.

$$length(SearchKeys) = \prod_{var}^{vars} length(StartContext(var))$$

TaskID - The task that this preference relates to.
SlotFeatures - The set of variable relationships defined for this **Prompt** in this **TaskID** that was created by the domain knowledge expert when the task was defined. It is saved with the learned preference built from it for reporting and system auditing purposes.
Entries - A list of case IDs that contributed to this preference.

References

1. Bobrow, D. G., Kaplan, R. M., Kay, M., Norman, D. A., Thompson, H., Winograd, T.: GUS, a frame-driven dialog system. Artificial intelligence, 8(2), 155-173 (1977)
2. Levin, E., Narayanan, S., Pieraccini, R., Biatov, K., Bocchieri, E., Di Fabbrizio, G., ... Walker, M.: The AT&T-DARPA Communicator mixed-initiative spoken dialog system. In: Proc. of ICSLP, vol. 2, pp. 122–125. (2000, October)
3. Bohus, D., Rudnicky, A. I.: The RavenClaw dialog management framework: Architecture and systems. Computer Speech & Language, 23(3), 332-361. (2009)
4. Saaya, Z., Smyth, B., Coyle, M., & Briggs, P.: Recommending case bases: applications in social web search. Case-Based Reasoning Research and Development, 274-288. (2011)
5. Bridge, D., Göker, M. H., McGinty, L., & Smyth, B.: Case-based recommender systems. The Knowledge Engineering Review, 20(03), 315-320. (2005)
6. Sugiyama, K., Hatano, K., & Yoshikawa, M.: Adaptive web search based on user profile constructed without any effort from users. In Proceedings of the 13th international conference on World Wide Web (pp. 675-684). ACM. (2004, May)
7. Schiaffino, S. N., & Amandi, A.: User profiling with case-based reasoning and bayesian networks. IBERAMIA-SBIA 2000 Open Discussion Track, 12-21. (2000)
8. Dean, J., & Ghemawat, S., MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113. (2008)
9. Berube, D.: Encode video with MongoDB work queues, `http://www.ibm.com/developerworks/library/os-mongodb-work-queues`
10. Bonnet, L., Laurent, A., Sala, M., Laurent, B., & Sicard, N.: 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 (pp. 483-488). IEEE. (2011, August)
11. Horowitz, E.: Schema Design at Scale. `http://www.10gen.com/presentations/mongosv-2011/schema-design-at-scale` Presentation, MongoSV (2011)
12. MongoDB 2.4 Release Notes, `http://docs.mongodb.org/manual/release-notes/2.4/#default-javascript-engine-switched-to-v8-from-spidermonkey`
13. Horowitz, E.: The Secret Sauce of Sharding. `http://www.10gen.com/presentations/mongosf2011/sharding` Presentation, MongoSF (2011)