Natural Language Interface to Relational Databases via Crowdsourcing

CMPS290H - Spring 2010 Project Report
Instructor: Prof. Neoklis Polyzotis

Bogdan Alexe
abogdan@cs.ucsc.edu

Sig Myers
sig@soe.ucsc.edu

ABSTRACT

In many practical scenarios, nonexpert users need to interact with relational databases without being familiar with SQL, the prevalent query language for such databases. In this work we investigate a novel approach to solving the problem of translating natural language questions into SQL queries. We propose the use of crowdsourcing as the backbone of our solution: leveraging large communities of workers remunerated by micropayments for drafting and refining translations. In addition to describing some initial ideas towards this end, we introduce a measure that quantifies the quality of produced translations, and present an experimental evaluation of our work using this measure.

1. INTRODUCTION

In many real life scenarios, users need to interact with database systems without being familiar with the query languages employed by these systems. As an example, natural scientists with no background in computer science need to extract information from very large datasets stored in relational databases. The learning curve they need to overcome before becoming familiar with SQL, the prevalent query language for relational databases, can be quite steep. Moreover, instead of directing their efforts towards the focus of their research, these users need to spend a considerable amount of time and effort before reaching a SQL query formulation that produces the desired results when run over the available data. It would thus be very desirable to provide to nonexpert users of relational database systems a capability that allows them to retrieve information stored in the database via questions in natural language, instead of specialized technical languages such as SQL.

Previous work focused on automatic translation of natural language statements into SQL queries by leveraging techniques developed in the natural language processing community. However, since semantic processing of natural language constructs is notoriously difficult, these techniques have lead to not very successful, domain specific implementations of natural language interfaces to relational databases. In this report we present a new approach to providing such a natural language interface. More precisely, we propose the use of a specific kind of cloud computing, namely crowdsourcing, in translating natural language statements into SQL queries. We envision a system where nonexpert users can delegate to a large worldwide on-demand freelance workforce the task of formulating SQL queries, given natural language descriptions of the characteristics of the desired results. We describe the first steps we took towards implementing the natural language interface, and evaluate the effectiveness of our initial techniques.

In a traditional cloud computing setting, a client that needs to carry out a computation task uploads the task to a service provider which is responsible to distribute the task to a (large) number of computation nodes and returning the results to the client. A fundamental observation that lead to the emergence of crowdsourcing as a type of cloud computing is that certain tasks are very difficult for computers, but relatively easy to carry out by human users. Two prominent examples of such tasks are image categorization and natural language processing. In a crowdsourcing scenario, the computation nodes are replaced by human workers executing such tasks that require human intelligence in order to be carried out effectively.

The most widely used platform for crowdsourcing, especially in the realm of simple to medium complexity tasks, is the Amazon Mechanical Turk [3]. Using this infrastructure, requesters can post task descriptions, while workers can browse through these tasks. They can choose some tasks to work on, and submit their results back to the requesters. The requesters can, based on the quality of the work, either accept the results and thus trigger the payment for the workers, or reject them and block payment.

In this work, we took some initial steps towards using the Mechanical Turk as an intermediary between nonexpert users posing natural language questions and relational databases. Among the many possible approaches to this problem, we focused first on a rather straightforward technique. In a first step, ask the workers to give their version of an SQL query corresponding to a natural language statement. As a second step, among the answers, choose the queries that appear to match the closest the semantics of the given natural language statement through a voting round where workers express their preferences by voting among multiple query formulations. Finally, aggregate the results corresponding to the queries receiving most votes, and return the aggregated results to the end user. This is a very simple workflow and there is ample room for alternate, more sophisticated techniques. Some of our ideas regarding directions we would like to pursue are presented in Section 6.

The outline for the rest of this report is as follows. In Section 3, we give an overview of the Amazon Mechanical Turk [3] service. In Section 4, we describe in more detail our techniques for translating

natural language questions into SQL queries via the Mechanical Turk. In Section 5, we introduce a quality metric for the result of a translation process, and use this metric to experimentally evaluate our approach. Moreover, we present some interesting observations that emerged while conducting the experiments. In Section 6 we present some possible directions for future work, while Section 7 concludes our report.

2. RELATED WORK

Amazon's Mechanical Turk is a popular platform for outsourcing tasks that are generally difficult for computers to do [3]. These tasks often fall into one of two categories. Tasks that are subjective and require a human opinion, and tasks which computational methods can't currently achieve. An example of the latter would be a worker tasked on Mechanical Turk with describing an image and giving a "tag" for that image. We discuss the utility and merits of Mechanical Turk further in the next section.

Psychologists, HCI experts and economists have investigated and analyzed the effects of incentive-based web applications such as Amazon's in [10], [7] and [9]. These studies discuss that a correlation does not necessarily exist between a financial incentive and quality of work. In fact, they show that this is not the case at all! This counter-intuitive result can possibly be attributed to workers in Mechanical Turk not taking their tasks seriously or not having a real need for financial gain [9].

Our problem space of converting natural language to SQL queries is not a new endeavor. Examples of such research can be found in [7,8]. To date, however, no method we're aware of has had strong results in converting natural language to SQL queries. Microsoft attempted to have a natural language converter for its SQL Server product, but the project was later abandoned [2]. Research is also being done in a similar context under the banner of Generative Programming, made popular by Czarnecki and Eisenecker in [5]. They describe many methods and generative techniques; of particular interest is the use of transformations that occur between a natural language set of software requirements and the resulting executable code to be generated.

A few organizations have taken the approach of moderately training a user in order to create valid SQL. Microsoft's Access introduced the notion of having a GUI available to aid an end-user (with a bit of knowledge and training) on how to execute queries over a dataset based on their high-level requirements. Microsoft's Access software has a relatively similar interface to its Excel software, but allows a user to construct a query by "design" or "wizard," the latter being a step-by-step walkthrough of the user's requirements and the dataset with which they intend to derive their results [1]. Similarly, OpenOffice's Base software also allows for SQL code generation via a graphical user interface and is made freely available [4].

Where Access and Base take the approach of educating the user to be able to execute SQL queries, domain specific languages (DSL's) rely on intimate user knowledge within a specific area of interest. A decent example of a DSL might be Mathematica, a utility that is often used to interpret mathematical expressions and compute results in the form of charts, graphs and datasets. The main benefit of a domain specific language is that, once created, the DSL can serve the purposes of an end-user (who has little or no coding knowledge) and eliminate the need to interface with a software engineer in order to get a result or product.

Many techniques exist in trying to bridge the gap between natural language and executable SQL (or code). These techniques often take one of the three formats described above: educating the end-user about a domain (as with Microsoft Access), tailoring the software to meet a vertical need (as with Mathematica) or a pure means of converting natural language to SQL (where research has not produced significant results to date).

Utilities are also starting to appear which enable users to take advantage of Mechanical Turk's API in a streamlined fashion. TurKit [8] is one example of a software utility that enables users to store a persistent state for tasks that are iterative by nature. TurKit allows users to input a javascript-based HTML form and for the duration of that particular task's execution, only one HIT ID (i.e. task ID) is created, as opposed to one HIT ID being generated for every iterative task. As discussed in the implementation section, we weighed the pros and cons of using this utility for our unique needs before deciding to build our own infrastructure.

3. THE AMAZON MECHANICAL TURK

Amazon's Mechanical Turk [3] is a product that builds on the notion of Human Intelligence Tasks, or HITS. At a high level, HITS can be any sort of task that a user can complete via a web browser. HITS are used for tasks that generally are hard for computers to solve on their own with modern day algorithms. Amazon's original usage of Mechanical Turk was for removing duplicate product listings on its website. A few months before Amazon released many of its cloud-computing services such as the Elastic Compute Cloud, SimpleDB, and Simple Storage Service, they released Mechanical Turk for public usage.

Whereas Amazon's other offerings allow users to create many instances of databases and servers, Mechanical Turk allows organizations to quickly leverage a large set of users through crowdsourcing. The individuals participating in crowdsourcing are referred to as workers on Mechanical Turk who have the capability of completing tasks made available by requesters, which are the organizations/persons who create the tasks.

Workers are offered (in general) a small financial reward for doing a given task. The listings on Mechanical Turk indicate that one or two cents is a normal wage for many tasks. We refer the reader to [10] for more information on wage/income related issues with Mechanical Turk. For an interesting article on worker geographic and demographic information, please see [6].

Users can create tasks on Mechanical Turk in two ways: a programmatic API, and a task designer. The task designer includes a basic WYSIWYG editor that produces an HTML form. When a worker completes a task, the entire contents of the completed HTML form are stored by Mechanical Turk and are able to be downloaded and stored as a CSV file.

Mechanical Turk's API allows programmers to interface with Mechanical Turk through a variety of different programming languages and virtually any piece of information, short of personal information about workers, is accessible to programmers. Code samples for many major programming and scripting languages such as C#, Java, PHP and JavaScript are available, as well as forums that are often contributed to by the development teams at Amazon. A sandbox allows the programmers to test their applications in the Mechanical Turk environment as either a worker or a requester. We found the sandbox to be extremely valuable in generating our tasks

Figure 1: Mechanical Turk's user interface is a means for workers to complete tasks provided by requesters from all over the world.

for testing purposes.

Both the API and task designer can be used to produce HTML forms which are then stored and processed by Amazon and accessible via the API. Whenever a worker completes a task, a timestamp of the worker's completion is stored, along with a unique ID of that worker, the reward, and any answers/choices made by that worker. If a requester finds their task's results to be inadequate, requesters have the option of denying particular workers payment for subpar jobs. Figure 3 shows the main GUI that workers/requesters see when logging into Mechanical Turk.

4. TRANSLATION FROM NATURAL LANGUAGE TO SQL

The system we built on top of Mechanical Turk is a Java/PHP based application coined SQLTurk. The high level specification we had at the onset of this project was that we had to: 1) have an easy means of creating a large number of tasks, 2) be able to store/update the state of our tasks while the application is in execution mode, and 3) be able to verify and analyze our results upon completion of our tasks. These three ideas encapsulate many points of interest within our implementation of converting natural language to SQL.

Every task we sent to Mechanical Turk required a few pieces of information. We supply a database schema and a natural language query over the schema for each of our tasks. Each natural language query has a resulting "truth" query associated with it. For our test datasets, we supplied example natural language and truth SQL queries. When workers write queries for one of our tasks, they are also offered a validation button, which tests the syntactic validity of the SQL they have created in a roundtrip call to our database. If a worker does not give a syntactically valid SQL statement we output an error message specifying what portion of the syntax threw an error message. This means of real-time validation ensures that workers get paid for submitting correct queries and helps to prevent syntactically incorrect SQL from entering our system.

We abstractly defined a workflow for every task we created in Mechanical Turk in order to maintain and update the state of our task. Workflows store elements such as the number of workers to complete the same task, the reward given to each completion of a task, a database schema with a corresponding natural language query, the resulting answers provided by every worker as well as their worker ID's and completion times. In this case, our answers provided by workers consists of the actual SQL statements the workers have written. We preserve the state of each of our workflows in a text file format which is regularly updated/modified based on the progress of our task (i.e. how many workers have completed the task at any given point in the tasks lifetime). Once we received answers for each of our tasks (generally 2, 3, 5 or 8 workers were asked to complete each task) or our time limit expired, we have completed our first checkpoint. In summary, this first checkpoint means that we've sent out a batch of tasks, assigned various rewards and numbers of workers assigned to each task, and stored each task's result set in a text file.

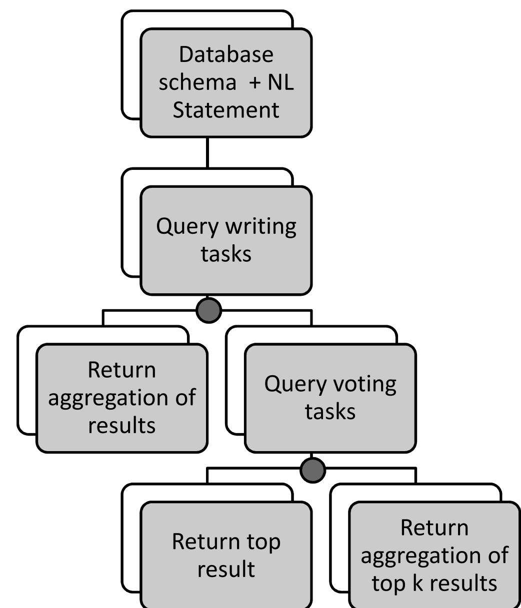

Figure 2: Our translation workflow displays the steps involved in generating SQL from a natural language statement. The query writing and query voting tasks are performed by workers on Amazon's Mechanical Turk.

At our next step, we have the choice of either taking an aggregation of all of our results (i.e. returning the intersection of every valid SQL query and analyzing the results), or creating a simple voting workflow which allows workers on Mechanical Turk to vote for the best answer when supplied with a schema, natural language query and the set of all answers to a given query writing workflow. Our voting workflow is quite similar to the original workflow in

that we send out a task to be completed by a number of workers, but in addition to the natural language statement and schema, we supply the list of valid SQL queries from the first checkpoint and the worker picks what they consider the best answer.

Once our tasks are completed or the lifetime of our voting workflow has expired, we again reach a checkpoint. At this point we post-process the results submitted by the workers. We considered two alternative ways of doing this. We can take the query with the highest number of votes (or arbitrarily pick the highest in the event of a tie), or take the intersection of the results produced by a predefined number of top queries. A summary of our translation processes are shown in Figure 4.

To illustrate the intersection of two SQL queries, we offer a brief example. The schema of the intersection result consists of all common attributes selected in each of those queries. For instance a query of "select * from Employee" might return five distinct attributes (first name, last name, ID number, etc.). A second query of "select firstName from Employee" would return just one attribute and it's corresponding values. The intersection between these two queries means that we take only the one attribute of "firstName" when performing an analysis on our results. To understand whether or not the attribute "firstName" is the same between two tables, we make use of equivalence classes which tell us this information. Equivalence classes maintain information pertaining to which columns are semantically equivalent for a given set of relational tables. Equivalence classes are derived by analyzing foreign key constraints within a table or set of tables. It is quite possible to have a foreign key constraint within the same table, such as with a "managerID" attribute and "employeeID" attribute that both correspond to the same type of data within the same table.

We make use of foreign key constraints for finding attributes in different tables that are semantically equivalent (for example an employee name in the Employee table might be equivalent to an employee name in an Invoice table).

5. EXPERIMENTAL EVALUATION

We will first introduce the performance metric we used to quantify the effectiveness of our techniques. Then, we will present our experimental settings consisting of a diverse query workload and two relational datasets. Finally, we will present the results of the experimental evaluation of our techniques in these settings.

5.1 Performance Metric

For a given query translation task, we would like to compare the results returned by our techniques to the results (which will be called *reference results*) produced by executing the "correct" SQL formulation for a specific natural language statement. Our intention is to use a precision/recall style metric that would give us an indication of how much of the returned result is actually valuable, as well as how well the returned result covers the reference result. However, traditional precision and recall measures applied at tuple level have a level of granularity that is too coarse for our needs. The schema of the returned result may be different than the schema of the reference result. Moreover, the returned result may contain only some partially correct tuples, which would be discarded by traditional tuple level measures. We would like to take these aspects into account in our measure.

To alleviate the aforementioned problem, we introduce a finer grained notion of similarity between two relations over possibly different schemas. From this notion of similarity, we then derive the measures of precision and recall that will be used in our experimental evaluation. We start by defining a measure of similarity between two tuples t and t' as follows:

$$\mathrm{Sim}(t, t') = \frac{|\{A \mid \exists A' \; t.A = t'.A' \text{ and } A, A' \text{ compatible}\}|}{|t|}$$

Intuitively, the above measure captures how many of the values in t can be recovered from the values of t', on compatible attributes. By compatible, we mean either the same attribute, or distinct attributes belonging to the same equivalence class, as introduced in the previous section. In our current implementation, two distinct attributes belong to the same equivalence class if there exists a chain of key/foreign key constraints connecting them. This value is normalized by the arity of t. As a next step, we use the similarity between a tuple t and an instance I', as the maximum value of the similarity between t and any of the tuples in I':

$$\mathrm{Sim}(t, I') = \max_{t' \in I'} \mathrm{Sim}(t, t')$$

Finally, the measure of similarity between two instances I, I' is the sum of similarities between each tuple in I and the second instance I', normalized by the number of tuples in the first instance:

$$\mathrm{Sim}(I, I') = \frac{1}{|I|} \sum_{t \in I} \mathrm{Sim}(t, I')$$

Given an instance I returned by our techniques, and a reference instance I' that is the result of running the correct SQL query, we define the precision P and recall R measures as follows:

$$P = \mathrm{Sim}(I, I') \qquad R = \mathrm{Sim}(I', I)$$

Having introduced our performance metric, we are now ready to present some results concerning the effectiveness of our translation techniques.

5.2 Experimental Results

Setting We considered two datasets: a database containing geographical and demographical information that is used as a sample scenario for the MySQL database server, as well as a car model online store database, which is used for demonstration purposes in the Eclipse BIRT project. Our query workload consisted of a mix of 10 queries, ranging from very simple selections based on arithmetic comparisons on single relations, to queries combining 3-way joins, selections, grouping and aggregation.

Findings We considered the resulting queries and several points in our translation workflow and computed the quality of their results in terms of the precision and recall measures defined above.

The first kind of result we considered was the aggregation of all the queries produced in the query writing phase by the workers. As described in Section 4, we computed the intersection of the results produced by all the submitted queries. However, since the quality of these queries varied significantly from one worker to another, the result of the intersection was of very low quality. In most cases, since there was at least one query for which the results had very little overlap with the other queries, the result of the intersection had good precision, but extremely low recall. Given this observation, we discarded the intersection immediately after the query writing phase as a viable option.

The second kind of result we considered was the result of the voting phase: we picked the query with the highest number of votes and

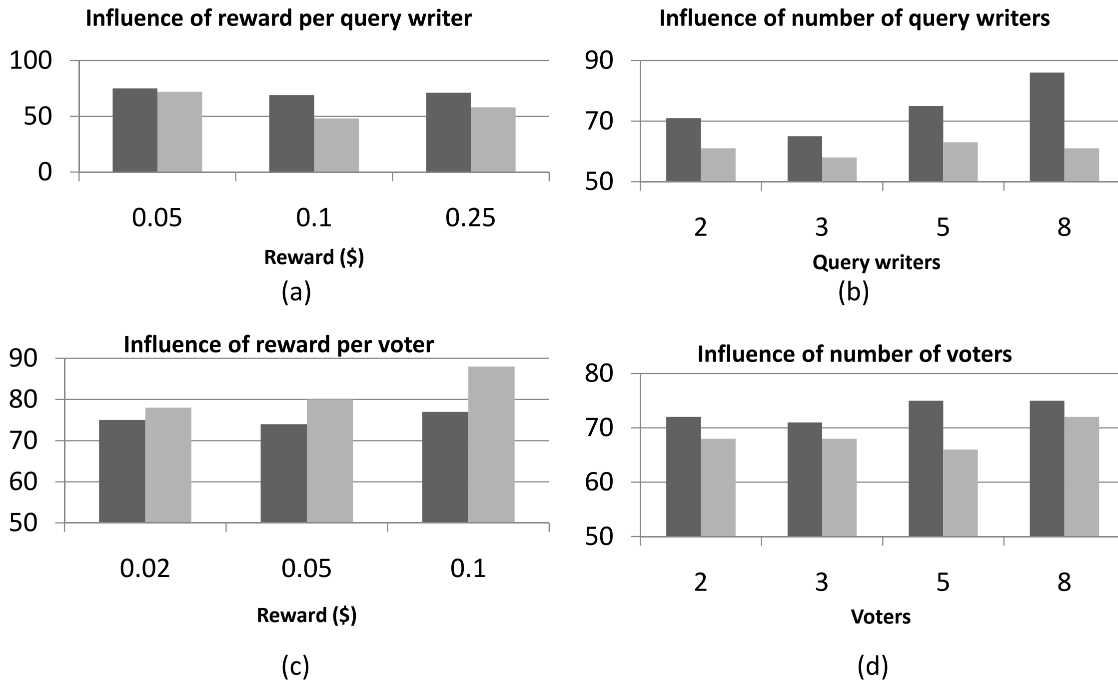

Figure 3: The influence of different translation parameters on the quality of the output result: precision (dark grey) and recall (light grey) in percentage points.

returned its result to the user. We considered different values for the number of query writers, their reward, as well as for the number of voter workers and their reward. A summary of our results is presented in Figure 5.

The first global observation that we can make is that the overall quality of the results is rather good: both precision and recall are in most cases in the 60% - 70% range. This is encouraging for our future work, since these results reflect our initial efforts based on some rather simple ideas. We expect to be able to improve on these numbers with more thought into refining our techniques.

The influence of the reward given to each query writer on the quality of the end result is given in Figure 5(a). We observe that the quality of the results does not increase with the amount of the reward. This is counterintuitive, but it is in line with other studies [9] on crowdsourcing, which show that for creative tasks (such as SQL query writing in our case), the quality of submitted work, above a lower threshold, in many cases does not improve when the reward is increased. Moreover, a second justification of this fact is that we required a rather specialized skill: SQL query writing. The population of workers on Mechanical Turk having this skill is rather limited. Although proposed over nonoverlapping time intervals, our task sets corresponding to different reward levels had some overlap in the groups of workers who solved them. Since the proficiency of workers in SQL was quite unlikely to increase from one set of tasks to another, it is understandable why there were no significant changes in the quality of results from one reward level to another. Another observation, backed by Figure 5(b), is that increasing the number of query writers improves the quality of the results. This is intuitive, and stems from the fact that there is a higher chance of having at least one high quality query among a larger number of candidate queries.

The last two graphs in Figure 5 depict the influence of the reward per voter, and the number of voters, respectively, on the quality of the results. We find that both increasing the reward for each voter, and the number of voters employed in each task, leads to better results. Both of these results are intuitive. Regarding the reward per voter, this is in line with other studies [9, 10] on crowdsourcing who found that for most relatively simple tasks and where creativity does not play a major role, quality of work is directly correlated to the reward. Increasing the number of workers has a positive influence on the quality of results. First, more votes can lower the effect of "outlier" or random votes. Moreover, considering a larger number of "honest" votes has a higher chance of making the result of the vote converge towards the good quality candidate queries.

We also considered the possibility of aggregating the results of the top 2 or 3 queries following the voting phase. For our experimental settings and for the batch of experiments we ran, more often than not it was the case that the top query was significantly better in quality than the query ranked second. This meant that taking the intersection of their results produced an aggregated result of low quality. However, more experiments are needed to clarify whether this is a consistent phenomenon. Our choice of aggregation operator (intersection) played an important role here as well. We plan to consider alternative aggregation operators in our future work.

Further observations. We noticed there were some minimal levels of reward for gaining the interest of workers on Mechanical Turk in solving our tasks. For query writing, this reward was $0.05, while for voting it was $0.02.

We found that the best time to submit tasks to the Mechanical Turk, in terms of minimizing the waiting time for results, was between 6PM PDT and 3AM PDT. This corresponds to post-business hours in the US, and to daytime in East Asia. This result is in sync with

our observations regarding the demographics of the workers, which showed that the majority of the people who solved our tasks originated in South India. In terms of delay until the completion of all of the tasks we submitted in one batch (corresponding to one level of reward), we noted that the number of incoming responses peaked within the first 12 hours (for query writing) and 6 hours (for voting). The vast majority of the responses for each batch were collected within approximately 24 hours (for query writing) and 12 hours (for voting).

6. FUTURE WORK

Our experimental results motivate us to pursue other directions in using the Mechanical Turk for natural language to SQL translation.

Among the variations on our current voting technique, we can consider a setting where different workers vote on different subsets of the candidate queries, followed by vote aggregation. We can also consider using a filtering phase for voting workers: do not allow workers to vote unless they first pass a qualification test involving some basic SQL skills. We do not currently allow workers to modify candidate queries they analyze in the voting phase. One variation would be to allow workers to make changes to proposed queries.

The choice we made for the aggregation operator (intersection) proved to be not very useful since it allowed one query outlier of poor quality in a group of queries to drastically worsen the quality of the final result. We plan to investigate alternative approaches to aggregate results of different candidate queries that have been deemed of having good quality, potentially after a voting phase.

Another change to our voting scheme would be to show workers not alternative query formulations, but alternative results given by the candidate queries on the same synthetic input example. An important challenge here would be the choice of the input example that will be able to "distinguish" among the candidate queries.

A radically different approach to the entire translation process would be to use not the SQL writing skills of the workers, but only their natural language knowledge comprehension skills. This would require doing, in an initial phase, some primitive natural language processing on the input statement. Then, workers would be presented with simple questions about the natural language statement. These questions, which would be problematic to an automatic natural language processor, would provide answers that could be used by our system to infer properties of the desired SQL query. This method poses significant challenges, but has the potential of considerably reducing the cost and time needed for the translation process, since the Mechanical Turk tasks would be much simpler, and thus cheaper.

7. CONCLUSIONS

Our experiments have shown that it is quite possible to use Amazon's Mechanical Turk as a means for producing SQL from a natural language statement. Seeing precision and recall metrics in the 60-70% range in most of experiments indicates that our ambition of closing the gap between natural language and SQL is within grasp. We believe that with further refinement we will be able to improve these metrics even further, as our current methods take a somewhat basic approach in generating a solution.

We found that some of our approaches were more worthwhile than others. Taking an intersection of the top few queries often resulted in an empty result set, which proved problematic. Similarly, we would like to get a more statistically significant set of respondents to our tasks to be able to understand some of the trends we saw in our experiments section. That being said, we were able to see quality results that, with some refinement, may prove to be extremely useful to individuals who have no domain knowledge of SQL.

8. REFERENCES

[1] Microsoft Access, http://office.microsoft.com/en-us/access/default.aspx.

[2] SQL Server NL, http://msdn.microsoft.com/en-us/library/aa174480(sql.80).aspx, 1 2000.

[3] Amazon Mechanical Turk, www.mturk.com, 2010.

[4] OpenOffice, http://www.openoffice.org, June 2010.

[5] K. Czarnecki. Generative programming - principles and techniques of software engineering based on automated configuration and fragment-based component models /. 1999.

[6] P. Iperotis. Mechanical Turk Demographics, http://behind-the-enemy-lines.blogspot.com/2008/03/mechanical-turk-demographics.html, March 2008.

[7] A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing user studies with mechanical turk. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 453–456, New York, NY, USA, 2008. ACM.

[8] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. Turkit: tools for iterative tasks on mechanical turk. In *HCOMP '09: Proceedings of the ACM SIGKDD Workshop on Human Computation*, pages 29–30, New York, NY, USA, 2009. ACM.

[9] W. Mason and D. J. Watts. Financial incentives and the "performance of crowds". *SIGKDD Explor. Newsl.*, 11(2):100–108, 2009.

[10] R. Snow, B. O'Connor, D. Jurafsky, and A. Y. Ng. Cheap and fast—but is it good?: evaluating non-expert annotations for natural language tasks. In *EMNLP '08: Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 254–263, Morristown, NJ, USA, 2008. Association for Computational Linguistics.