SQLTurk: A Human Interface to Relational Databases

Master Project Report *

Kerui Huang
Computer Science Department
University of California, Santa Cruz
khuang7@ucsc.edu

ABSTRACT

In many real life scenarios, people need to interact with relational databases without knowing SQL, the prevalent query language in current relational databases. To address this issue, this project introduces a novel system named SQLTurk, which provides a human interface between natural language and SQL query language. SQLTurk takes advantage of crowdsourcing to implement the two major phases – authoring and ranking queries. Additionally, to optimize the query results, SQLTurk also employs four algorithms to aggregate results. We use the World and TPC-H databases in our experiments to evaluate the performance of SQLTurk. Furthermore, we are interested in the factors which possibly influence the results. The experimental results show that: 1) aggregating few ranked queries can lead to a good performance; 2) showing sample data to workers is helpful; 3) among the four aggregation algorithms, our Intersection and Union have high precision, while Full Disjunction and Full Disjunction Plus have high recall; 4) the complexities of query and schema are not very influential.

1. INTRODUCTION

In many practical scenarios, non-expert users need to interact with relational databases in order to extract the data they want. For instance, natural scientists need to retrieve the information from very large datasets stored in relational databases. However, most people have no background in computer science and are not familiar with SQL, the prevalent query language in current relational databases. On the one hand, without knowing SQL query language, it is impossible for them to query a database themselves and get data as needed. On the other hand, the learning curve for SQL query language can be quite steep, which is very difficult to overcome for people who have no background in computer science. Even though learning SQL query language is a possible way for them, it will probably also take a considerable amount of time and effort before reaching a SQL query formulation that produces the desired results when run over the available data, which distracts them from their own research. In this sense, it is very desirable to provide non-expert users of relational database systems a capability that enables them to extract information stored in the databases via questions in natural language, instead of specialized technical languages such as SQL.

The rapid growth of crowdsourcing within research and industry produces a novel idea aimed at dealing with this problem – translating natural language statements into SQL queries. We can think of crowdsourcing as a specific kind of the combination of cloud computing and outsourcing. On crowdsourcing platforms, such as Amazon Mechanical Turk [1], CrowdFlower [2], oDesk [3], there are many different kinds of people, named crowds, with various skills working on miscellaneous tasks all over the world. In real world, it is not hard to find some certain tasks which are very difficult for computers but relatively easy to carry out by human workers. Two prominent examples of such tasks are image categorization and natural language processing. This observation incurs the emergence of crowdsourcing as a brand-new solution to these problems.

Though natural language processing can be used for converting a language to another language, these techniques are not very successful in domain specific implementations of natural language due to the difficulty in understanding semantics. However, humans can understand semantics easily, which makes crowdsourcing a potential solution to this specific issue. Our purpose is to build a system where non-expert users can give the natural language descriptions of the characteristics of the desired results, such as the information they want to retrieve from databases, and then get the data as needed with the help of worldwide on-demand freelance workforce in crowdsourcing environment. As a popular platform of crowdsourcing, Amazon Mechanical Turk [1], which is especially good for simple tasks, serves our purpose in an ideal way. We leverage this platform as our infrastructure under the hood for crowdsourcing, involving human intelligence as part of our solution. Besides, we also make use of four algorithms to aggregate the intermediate results in order to reach a better final result.

The outline for the rest of this project report is as follows. Section 2 gives an overview of the crowdsourcing platform we use in our project – Amazon Mechanical Turk. Section 3 describes the two primary phases of our crowdsourcing tasks, authoring query and ranking query respectively. In section

*This project is under the guidance of Professor Neoklis Polyzotis, joint work with Professor Wan-Chiew Tan and Dr. Bogdan Alexe.

.

4, we discuss the four algorithms for result aggregation. In section 5, we discuss the performance metric we used for evaluating SQLTurk and our experimental results.

2. THE AMAZON MECHANICAL TURK

Amazon Mechanical Turk [1] is a widely-used crowdsourcing infrastructure, where thousands of tasks are posted and taken by people every day. These transactions help people solve many problems which are fairly difficult or even impossible for computers. The original usage of Amazon 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.

In this crowdsourcing environment, the first step of the workflow is to make tasks available to workers. The requesters post their tasks as HITs with task descriptions, requirements and sample results and so on, while workers can browse through these tasks and pick up the tasks they like to work on. When the tasks are completed, the workers submit the results to Amazon Mechanical Turk and wait for their payments. After the requesters collect the results, they can review these results to ensure the quality of the answers and decide to accept or reject the results. If the results are rejected, the workers providing these results are not paid. If the requesters are satisfied with these results, the payment is triggered automatically for these result providers after the results are accepted.

The basic and atomic task unit on Amazon Mechanical Turk is Human Intelligence Task, or HIT for short. According to different complexity of the work, the prices of different HITs can range from 1 cent to several dollars. However, the majority of crowdsourcing tasks on Mechanical Turk are small tasks, and they are what Mechanical Turk is good at. By completing a HIT, the workers get a certain amount of financial reward set by the requesters. In addition, this platform is a global platform where you can gather all the workers all over the world to work for you [4]. It is so flexible that many tools can be found to formulate your task and control the workflow, such as command line tools and development SDKs in major programming languages like Java, Python, Ruby and so on. With this powerful infrastructure, we can easily build our tasks on top of it and use it as the interface of the translation from natural language statements to SQL queries. The non-expert users can use this intermediary for collecting the initial results, rather than learning SQL query language themselves. This process is transparent for non-expert users. And because price is not the dominant factor in the quality of the work as is shown in [**?**], people can pay only a small amount of money to have solutions of high quality, rather than paying as much as some traditional ways such as hiring over-qualified and expensive experts to write queries against the target databases. We use this crowdsourcing as the backbone in the initial phases of our SQLTurk.

3. QUERY AUTHORING AND RANKING VIA CROWDSOURCING

In this project, we take some initial steps towards using Amazon Mechanical Turk as an intermediary between non-expert users and relational databases. In the first phase, we ask the workers to write the queries based on given conditions and requirements. After obtaining these queries, we move to the second phase which also involves using crowds to solve the problem. In the second phase, in order to have a better sense of which queries are probably of high quality, we let workers rank the candidate queries coming from the first phase. By getting the ranking results, we can sort candidates according to their ranks. In this way can we choose top k for our aggregation work, which is supposed to lead to a better outcome than the outcome resulting from another mode that we only employ a singe worker and use the only one query from this worker. The first two phases, authoring and ranking queries, involve crowdsourcing, while the aggregation phase does not. The aggregation phase mainly emphasizes on exploiting database techniques to optimize the results. The overall workflow is shown in Figure 1. This section describes how authoring and ranking queries work in SQLTurk.

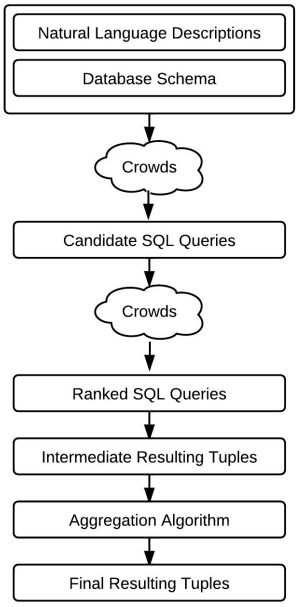

Figure 1: The overall workflow of SQLTurk.

3.1 Query Authoring

At the very beginning, we employ workers to write queries based on our needs. Such tasks are presented with some natural language descriptions about the task and the related database schema. These pieces of information are supposed to come up with enough information for a worker to acquire the entire picture of this particular task, such as what information to query about, what are the basic rules to follow, how the database schema looks like and so forth.

To help authors to write correct queries, we supply a syntax validation button, which informs them whether their queries are syntactically valid or not against the current particular database instance. If a query does not pass this validation, a red line of error message shows up beneath the incorrect query, specifying what portion of the syntax threw what kind of error. Then workers can modify their queries based on this piece of information and validate their queries again until they reach the syntactically-correct queries. If a query with syntax error is submitted, this query will be re-

jected during the review process and the worker will not be paid. Therefore, besides the tips of how to correct queries, this is also of great help for workers to ensure the syntactic validity of their queries and thus they will get paid afterwards.

In order to examine how samples affect human's performance, we divide our authoring tasks into two types, with or without samples respectively. Every authoring HIT contains two tasks. The first task is to write queries only based on the given natural language statements and the corresponding database schema. The second task is to write queries based on not only these given pieces of information as above but also some sample data from the databases. We choose five tuples for each table as the sample data and present them to workers. The sample data may visually aid people to obtain a better sense of the database. In addition, workers can also move their mouse onto an attribute to trigger a bubble which has more information about the attribute, such as the type of the attribute, whether it is a foreign key and so on. The two tasks differ from each other, which means each task can be considered as a separate task of authoring a query for a natural language question, because it does not make any sense to repeat a same task twice. Also, this can avoid the side effect that the first task may be somehow helpful for workers to work on the next one. To be fair enough, the two questions existing in the two tasks in a single HIT have same complexity but are literally different, which is supposed to rule out other factors except the samples if there is any difference observed between these two tasks.

3.2 Query Ranking

Due to the diversity and flexibility of crowdsourcing, it is difficult to say that one can have the best result by employing only one worker. A better way is to involve more workers and then choose the better results among these candidate results. After the first phase, we have collected the candidate queries from our workers. The next phase is to figure out which queries are better queries.

This phase is quite similar to the previous phase. We also make use of crowdsourcing to complete the ranking tasks. Each ranking HIT is equivalent to a previous authoring HIT. The two tasks in a ranking HIT are almost the same as the two tasks in a authoring HIT. The only difference lies in that the text field which used to serve as the place for inputting the query is replaced by a list of candidate queries from the first phase. There is a box associated to each candidate query and workers are asked to enter the rank of the corresponding query in the box based on their own point of view. Likely, each ranking HIT also employs a few workers to rank the candidate queries within it, under the same assumption that more than one workers are always better than a single one. Therefore, after this phase, all the candidate queries have a list of rankings for them. For instance, we may ask ten workers to write a query for a given natural language question in the first phase. Then, in the second phase, after we collect all the ten candidate queries from these ten workers, we post the HIT asking another ten workers to rank the ten queries from the first phase. Therefore, when the second phase is done, we have ten rankings for these ten candidate queries.

Since we use external page as our implementation method interacting with Amazon Mechanical Turk, we have the full control of the workflow on it. We wrote a function in Javascript to check the validity of answers. When a worker submits the answer, if the answer is invalid, such as having duplicate rankings or the ranking is beyond the scope, the task page pops up a warning window indicating the error. In this way can we guarantee each submission are eligible for our ranking purpose. After the submission passes the validation function, the answers are submitted to a particular form specified by Amazon Mechanical Turk. Then, the answers are recorded by Mechanical Turk automatically. When all answers for a HIT are available, we can then approve all these tasks, pay the workers and download the ranking results for future use.

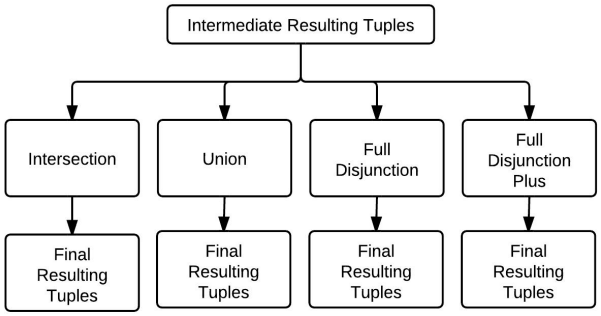

Figure 2: The aggregation process of SQLTurk.

4. AGGREGATION FOR OPTIMIZATION

The first phase provides us a list of candidate queries for each certain natural language question, and we assume that the task worked by a group of people is always better than a single worker. Furthermore, in the second ranking phase, workers imply their preferences on the quality of queries by ranking these query formulations. After the second phase, all the candidate queries are ranked, which helps us to sort these queries by their quality.

To consolidate the per-worker rankings, median-rank algorithm is applied to derive the final ranking [**?**]. The algorithm takes all the rankings into account and finally comes up with the final ranking. This enables us to integrate all rankings from different workers and come to an agreement.

After we merge all the rankings to reach a final ranking, we now attain a list of finally ranked queries. Each candidate query can be used for querying database and then have a table containing all the resulting tuples. We utilize the multiple queries and their ranking for aggregation, which leads to a better final result. Here are four basic algorithms for aggregation. Figure 2 illustrates the process of aggregation in SQLTurk. We discuss the four aggregation algorithms in this section with more details. They are Intersection, Union, Full Disjunction and Full Disjunction Plus.

4.1 The Intersection

The first algorithm for aggregation is our Intersection. This algorithm is different from traditional intersection. To illustrate it, we introduce the concept of equivalent class. The equivalent class is introduced to find the attributes which are semantically equivalent but in different relations with different names. We use foreign key constraints for finding the attributes in different relations that are equiv-

alent. If there is a foreign key constraint between two attributes in separate relations, we say these two attributes are in same equivalent class. The Intersection between two relations means that we take only the attributes which are in the same equivalent class when performing an analysis on our results. For instance, a query of "select * from Employee" might return three distinct attributes (first name, last name, ID number). Another query of "select firstName from Employee" might return one attribute "firstName". If the two returned attribute are common attributes or there is a foreign key constraint between them, the intersection result is the tuple with only one attribute "firstName" if their values are equal.

4.2 The Union

The second algorithm is our Union algorithm, which also vary from the traditional union algorithm. The Union algorithm is sort of similar to the Intersection algorithm. However, instead of rule out the tuples which mutually disagree with each other in the resulting relations, the Union algorithm tries to include as many tuples as possible. That is, the Union algorithm accept all the tuples from each resulting relation no matter whether they agree on equivalent attributes or not. Moreover, duplicate tuples are eliminated in the final result. It apparent that the Intersection algorithm may result in missing correct tuples, while the Union algorithm may result in keeping incorrect tuples.

4.3 The Full Disjunction

The Full Disjunction algorithm employed by SQLTurk was proposed by Anand Rajaraman and Jeffrey D. Ullman [6]. We can observe that both Intersection and Union focus on the operations on tuples. However, they can incur information loss. As an example, consider a tourist who is choosing a destination. Among the plethora of sites that contain tourist information are sites that specify climates, sites that elaborate on tourist attractions and sites that specialize in hotels. On the one hand, for a more straightforward view of the information, the tourist need a relation which joins all the consistent tuples together. On the other hand, to obtain the entire picture of his possible choices, the tourist needs all the pieces of information, which means even though some attributes of a tuple are null, the tuple is still supposed to be kept in the resulting table. Namely, we need a way to effectively query a collection of data sources and combine the data together in a coherent fashion without losing any information. The Full Disjunction is a variation of the join operator that maximally combines tuples from connected relations. It can be seen as a natural extension of the binary outerjoin operator to an arbitrary number of relations and is a useful operator for information integration.

4.4 The Full Disjunction Plus

The original Full Disjunction algorithm uses the consistency of common attributes to do the join operation. A possible variant of Full Disjunction is to take advantage of the provenance resulting tuples. We modify the original definition of Full Disjunction so that join operations also take into account provenance, a great property of data. This new version Full Disjunction is called Full Disjunction Plus. Specifically, we employ why-provenance and tableaux in our Full Disjunction Plus. Tableaux capture possible data associations through foreign key constraints on a database schema. There is one tableau for each relation in the schema and each tableau is constructed starting from a single relation via chase with outgoing foreign key constraints. We first use tableaux to rewrite the queries by appending the tableaux involving the queries. Then, we use why-provenance to keep track of the provenance of each resulting table, which is used for checking if two tuples are from the same provenance. When we use the framework of the original Full Disjunction to do the join operation, we add a new condition for determining whether two tuples should be joined that these two tuples are from the same provenance. Therefore, the new conditions for joining two tuples are not only based on whether they agree on common attributes but also depends on whether they stem from the same provenance. Since provenance is introduced in Full Disjunction Plus, it is more strict than the original Full Disjunction algorithm to join two tuples together, but this can be useful to increase the accuracy of final results.

5. EXPERIMENTAL EVALUATIONS

To evaluate our new approach, we deployed our tasks on Amazon Mechanical Turk and collected queries written by workers. Then, we aggregated all these queries with the four algorithms mentioned above. After we get the final results, we used our performance metric to evaluate the performance of SQLTurk. In this section, we will first present the overall workflow of our experiments, and then introduce the performance metric we used for evaluating the results. Finally, we will present the experimental results and discuss about them.

5.1 Workflow

In our experiments, we made use of two datasets. The first database, named World, contains geographical and demographical information which is used as a sample scenario for the MySQL database server. The other one is the TPC-H database which is a decision support benchmark. The World database is less complex than TPC-H database in terms of their schemas.

Then, we brought up 8 questions for each dataset. All these questions are designed to be as natural as possible. By natural, we mean these queries are very likely to be asked in the real world. Besides, these questions are also different in their difficulty or complexity. For example, some simple questions may only involve a table, while some complicated questions may need join operations across several tables. There are 4 different complexity levels, and each level has 2 questions. Intuitively, the two questions at the same complexity level are complexity-equivalent. In light of this, we always have two questions which are equally complicated.

As shown in Figure 1, we first make use of Amazon Mechanical Turk as our crowdsourcing platform. We deployed our tasks on Amazon Mechanical Turk to collect queries for our proposed natural language questions. As mentioned in previous section, each authoring HIT has two separate questions. The first question is given with the database schema and the natural language question. Besides the schema and question, the second question also comes up with some sample data for each table, such as the first five rows of each table. Also, both of the two question are complexity-equivalent, which means the only difference between the two question in a single HIT lies in whether the sample data is displayed or not. Moreover, for each question, we asked ten

workers to work on the question which produced 10 candidate queries each.

After receiving all 10 candidate queries for a question, we posted a related ranking HIT which asked 10 workers to rank these candidate queries, Similarly, the first ranking task in the HIT showed workers the schema and question, while the second question additionally showed them sample data.

At the next step, we utilize median-rank algorithm to integrate the 10 rankings for the 10 queries and then reach the final ranking of 10 queries.

To test whether the ranking is helpful, we choose some top k queries, where k can range from 2 to 10. The top k queries then generate k resulting tables, which will be used for the next step of aggregation.

By querying database with the chosen queries, we have some resulting tables. Then, we use the four aggregation algorithms described above to merge these tables. Note that Full Disjunction and Full Disjunction Plus are computation-intensive, we use an alternate implementation [5] in SQLTurk. After aggregation, all the original resulting tables are integrated into one table, which is the final result of SQLTurk. So far, we have the final result of SQLTurk. The next step is to evaluate this result so as to have a better sense of its performance. Before presenting the experimental results, we introduce the performance metric we used.

5.2 Performance Metric

After the aggregation process, we now have one resulting table for a given natural language question. To measure the performance of SQLTurk, we need to know how far it is away from the "correct" answer. Therefore, we will use the performance metric to compare the results returned by our techniques with the results produced by executing the "correct" query. The performance metric involves two precision and recall. Precision indicates how much of the returned result is actually valuable, while recall indicates how much the returned result covers the correct result. Therefore, both precision and recall are related to measuring the similarity between two relations.

First, we introduce the measure of similarity between two tuples t and t' as follows:

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

This measure captures how many of the values in t can be recovered from the values of t' on compatible attributes. The compatible attributes are the same attributes or distinct attributes belonging to the same equivalenct class. Then, 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':

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

The measure of similarity between two relation instances I and I' is the sum of similarities between each tuple in I and I', normalized by the number of tuples of I:

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

After we have the similarity measure for two relation. With the help of this measure, we can define our precision and recall as follows:

$$Precision = Sim(I,I'), \quad Recall = Sim(I',I)$$

where I is the instance finally returned by SQLTurk, while I' is the instance which is the result of running the correct SQL query.

So far, we have introduced our performance metric. Then, we are now ready to present some results about the effectiveness of SQLTurk.

5.3 Experimental Results

In this experiment, we intend to measure various aspects of SQLTurk. The factors which are considered to have impact on the performance of SQLTurk are the complexity of database schemas, the complexity of queries, the user interface such as whether sample data are shown or not, how many queries we use for aggregation computation and the overall impact of different algorithms on the performance. To make the experiments more close to real world scenarios, our default configuration of experiments utilized a question whose correct query was neither too easy nor too difficult as the standard one. In addition, we used top 3 candidate queries from this question for aggregation process by default. Finally, because the requesters may not know anything about the database, the default question was given without any sample data from the database.

5.3.1 The Number of Queries for Aggregation

The work completed by more than one person is usually better than the work done by only one worker, we distributed 10 assignments for each question and then collected the 10 candidate queries from these assignments. When all the 10 queries were returned, we created another 10 assignments for ranking these 10 candidate queries. After the ranking phase, median rank was applied to integrate the 10 rankings to reach the final ranking. Then, we aggregated different numbers of ranked queries to see if the number of queries used for aggregation caused differences. Figure 3 shows that the precision of final result decreased in general when we involved more queries in aggregation, no matter what aggregation algorithm it was. However, the recall of the results generated by Full Disjunction and Full Disjunction Plus almost did not change, but we can also observe a very slight increase. The recall from Intersection or Union decreased a little. This indicates that only a few queries cab probably lead to a good performance, while involving more queries makes aggregation worse. Because the queries were ranked already after the ranking phase, involving more queries also means introducing more noise into future aggregation, which is the reason why the precision and recall went down with the increase of number of queries involved. Also, it tells us that the final result can benefit from the ranking phase, which implies the quality of queries.

5.3.2 The Complexity of Database Schema

Different databases come up with different schemas. The common sense is that simple schemas are easier for people to understand, which are therefore supposed to lead better results. The first factor we examined is the complexity of database schema. As mentioned before, we used two datasets in our experiments – World database from MySQL server demo and TPC-H benchmark. There are only three relations in World database, while the TPC-H database contains eight relations. In the meantime the number of attributes in TPC-H is much more than that in World database. Also, TPC-H has more foreign key constraints

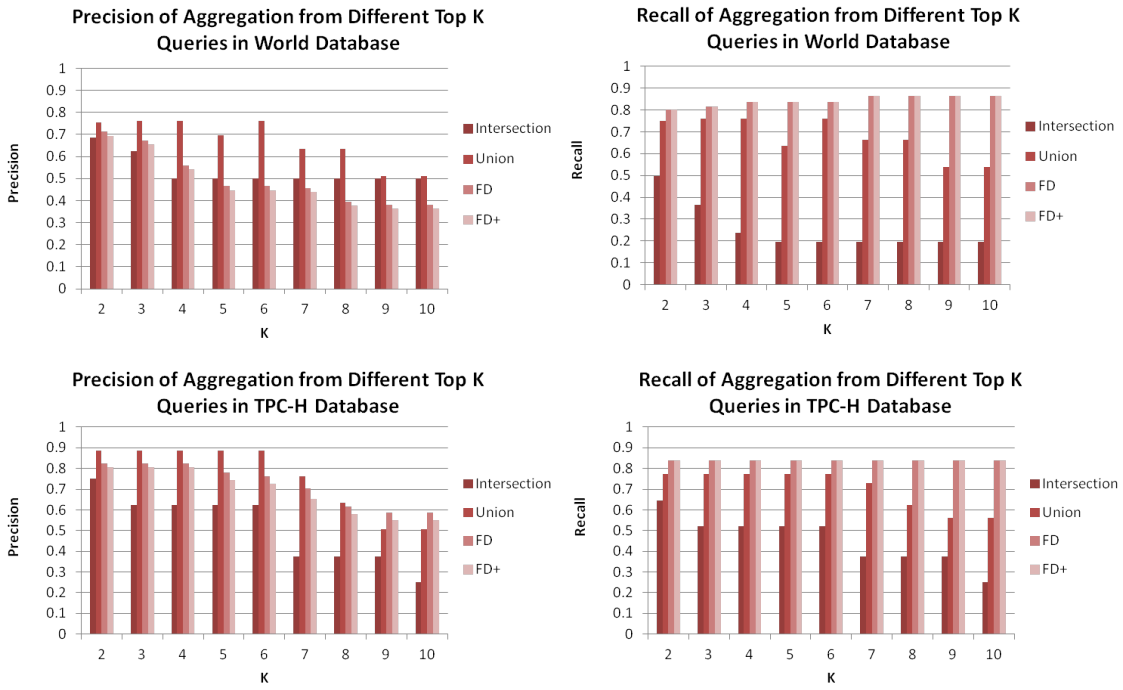

Figure 3: The impact of the number of queries used for aggregation.

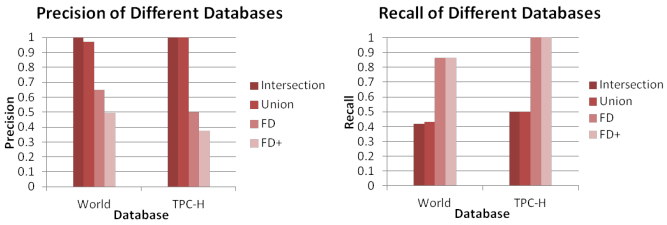

Figure 4: The overall comparison of precision and recall between World database and TPC-H database.

than World does. In this sense, we say the schema database of TPC-H is more complex than the schema of World database.

Figure 4 depicts the changes of precision and recall of the standard query with different databases. On the one hand, even though Full Disjunction and Full Disjunction Plus have slightly lower precision, the difference of the overall precision between the two databases is not significant. On the other hand, the difference of recall between World and TPC-H is insignificant as well. This interesting phenomenon implies that the complexity of database schema has little contribution to the performance of our approach. Namely, this means there is necessary relationship between the complexity of database schema and the performance. Though a query may involve a very complicated database, it does not necessarily mean that the query itself needs to be as complicated as the database schema. On the contrary, the query can possibly be associated with a relatively simple question. In short, there is no certain connection between the complexity of schema and the performance, since the complexity schema does not necessarily mean a complicated query, which confuses workers and thus reduce the precision and recall.

5.3.3 *The Complexity of Query*

Since different complexities of queries may lead to different outcomes, we examine the impact of the complexity of query too. We used four queries with different complexities against each database. For example, the simplest queries are only involve one relation, while the most complex one may require workers to join multiple tables and use more predicates. All the related questions were shown without sample data. Figure 5 shows the experimental results. The simplest queries got higher precision and recall than other queries. We can also see that the precision and recall derived by some algorithm changed. Nevertheless, it is surprising that there is no evident pattern under these changes. This probably means that the complexity of query may not have influence on the results.

5.3.4 *Sample*

The user interface can be another important factor for SQLTurk, since it involves human. More specifically, the user interface we examined in our experiments is whether sample data are displayed to workers or not, which is the most important aspect of user interface we think in SQLTurk. With the help of sample data, people are more likely to have a better understanding of the task. Concrete examples always make people have a better sense of how the database schema looks like, which eases the work process by making it more straightforward. From Figure 6, we can observe that the precision was higher when the question had examples. However, the recall with samples became lower. With some samples, workers are more likely to write accurate queries. Therefore, the precision can be higher if we

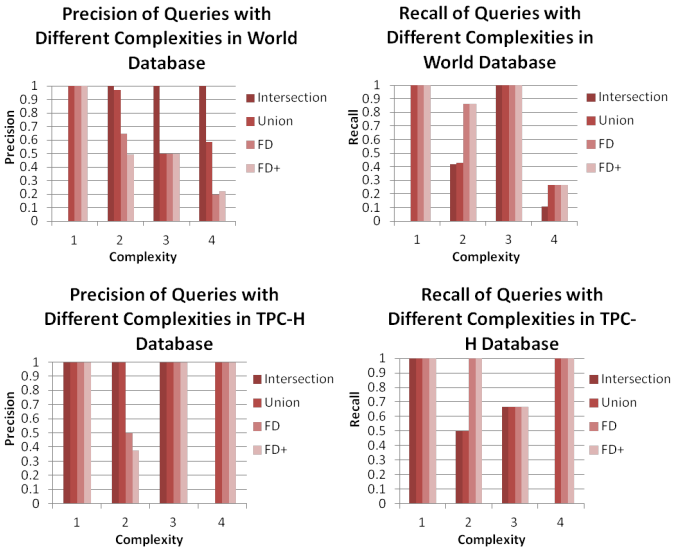

Figure 5: The impact of complexity of query.

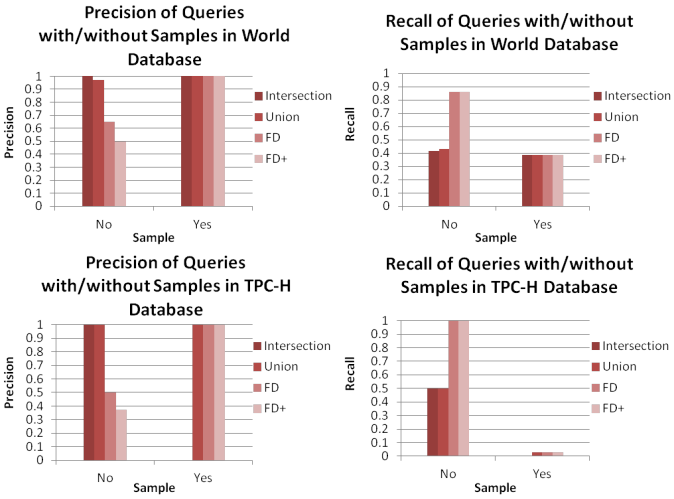

Figure 6: The impact of sample data.

provide samples. Without samples, since it is more difficult for workers to get the whole picture of the problem, their answers probably introduce more noisy data, which retains more data and thereby obtains a higher recall.

5.3.5 *The Overall Comparison between Algorithms*

Finally, we used the standard query to explore the overall performance of each algorithm. Different algorithms have different properties, which can lead to quite different results. Figure 7 demonstrates that Intersection and Union have higher precision, while Full Disjunction and Full Disjunction Plus have higher recall. As discussed before, it is mainly because Intersection and Union only keep the equivalent attributes, while Full Disjunction and Full Disjunction Plus keep as many attributes and tuples as possible.

6. CONCLUSION

In this project, we introduce our new approach SQLTurk

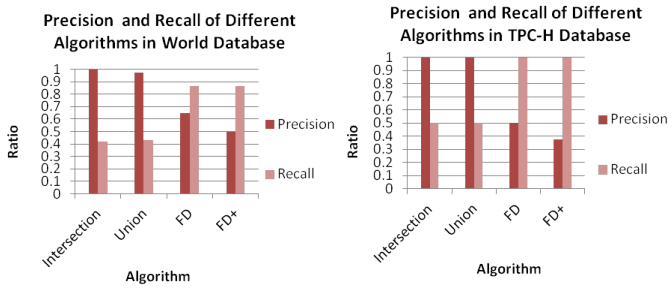

Figure 7: The overall comparison between algorithms.

to solve the problem of translating natural language statements into SQL queries. Intuitively, people bring up their question in natural language while the data are stored in relational databases, where SQL query language is required for retrieving information. There must be a way to translate natural language statements into SQL queries. However, semantics of natural language is the handicap to achieve this goal by computer, since semantics is quite difficult for computer to understand. SQLTurk takes advantage of crowdsourcing to overcome this problem. We make use of human intelligence to gain queries. To optimize the results, we utilize four algorithms to aggregate the results. The experiments show that we can improve the performance of the results in some certain ways, such as using only a few top ranked queries for aggregation, showing sample data to workers. Moreover, we can choose appropriate algorithms to aggregate resulting tuples based on our needs, since different algorithms have different properties and thus have different performance on precision and recall.

7. REFERENCES

[1] Amazon Inc., Amazon Mechanical Turk, https://www.mturk.com/mturk/welcome.

[2] Crowdflower Inc., CrowdFlower, https://www.crowdflower.com.

[3] oDesk Inc., oDesk, https://www.odesk.com.

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

[5] S. Cohen. An incremental algorithm for computing ranked full disjunctions. In *In PODS*, 2005.

[6] E. Ed, A. Rajaraman, and J. D. Ullman. Integrating information by outerjoins and full disjunctions. In *In Proc. 15th Symposium on Principles of Database Systems*, pages 238–248. ACM Press, 1996.