Crowdsourced Query Understanding and Optimization

Big Data Course, EPFL

2015

Florian Chlan, François Farquet, Joachim Hugonot, Simon Rodriguez, Kristof Szabo, Florian Vessaz, Guo Xinyi, Vincent Zellweger

T.A. : Immanuel Trummer

Contents

1 Introduction

Crowdsourcing is a process to use human workers to do simple tasks. For a lot of tasks, humans are better suited than computers, because they can correctly understand the context of a variety of tasks by themselves, for which machines fail. Picture analysis *"Is there a cat in this image?"* or semantic analysis *"Is this comment sarcastic?"* provide good examples of things, that humans are good, but computers routinely fail at.

There are some things that are currently not possible with traditional database systems. To query a database, one needs to have the information and the information structure already inside the database. Our goal is to create a system to retrieve data from the vast amount of unstructured information available on the internet right now. We do this by distributing work to the Amazon Mechanical Turk (mturk) platform.

2 Project details

2.1 Global structure

Our projects consists mainly of three layers. The part were we parse the query and create the associated abstract syntax tree, the part were we use this tree to create Human Intelligence Tasks (HITs) and finally the part which communicates with the Amazon Mechanical Turk platform , sends those HITs and retrieve answers from the workers.
To visualise the status of our queries, we implemented a user-friendly web interface on which the user can find relevant information for each query.
The majority of the code is written in Java and Scala but we also use HTML/CSS and JavaScript for the web interface.

An overview of the distribution of work among the team members during the whole project can be found in Appendix A.

2.2 Usage of Amazon Mechanical Turk

mturk provides an easy way to send work to tens of thousands of anonymous workers and collect the results afterwards. Each worker is paid for their results based on a reward chosen by the requester. We use mturks huge, scalable workforce to help with retrieving data for and processing the query.
To interact with Amazon Mechanical Turk service, we use the available GET interface. We developed our own AMT communicator, in charge of interpreting our internal abstract representation of a task (a HIT) to generate the corresponding URL request. We provide a Question abstract class and a number of subclasses to represent a wide range of question types, e.g. FreeText answers, URL answers, True/False answers. Each Question subclass is responsible for providing its mturk-complient XML scheme.
Once a HIT is sent, we encapsulate it into an AMT Task and add it to the list of pending jobs. We can then keep track of those tasks and poll their results at

a fixed interval of time. Answers to our requests are sent back in XML format and parsed by the AMT communicator, that generates Answer objects and uses a callback to alert higher level classes of the updates.
In our implementation, we currently don't use qualifications. This means that the answers from each and everyone of the workers are treated equally.

2.3 Query language and parsing

The parsing is inspired by the SQL query language. We tried to include a lot of features from SQL, such as the ORDER BY, the WHERE and the JOIN, into our language. To be more flexible, we have introduced two main things. First a hierarchy on the different commands and second the ability to parse queries recursively. This means that it is possible to have subqueries within a query.

A detailed description of the language parser and its features can be found in Appendix B.

2.4 Implementation choices

Once the query is parsed we use the query tree to generate a plan to answer the query, this plan is critical to obtain decent results, here a description of the main choices and optimizations we did to generate the plan:

- For a given query, we decompose it into sub-tasks which are generally really close to the semantic. This means that if we have a "WHERE" in the query we will have a dedicated task to handle this "WHERE".
- We **always** start with the "FROM" clause, where we ask **one** worker to give either a website or a list of primary keys. After that we can continue the execution of this query by using the results given in this first step.
- For each task we try to use simple and understandable formulation, as well as proper description and keywords for a better visibility on the mturk platform.
- The parallelization has been implemented with Scala Futures. The result of each HIT is contained in a Future. As soon as the result arrives, the onSuccess method of the Future is called sending a new HIT if possible. For example, if we send a query containing SELECT and WHERE tasks, some WHERE tasks can be submited even if not all the select tasks are finished.
- Parallelization of JOIN tasks works a bit differently. The two branches of the JOIN are executed in parallel but the JOIN task is waiting for the results of both parts before beginning. Changing this is a possible improvement for the future.

2.5 User interface

The UI is served over HTTP to the user and can be used in any recent web browser. Our application acts as an HTTP server[1] which delivers a web page to the user and allows to create, abort and get details on the pending queries via simple RPC. The server answer the various procedure calls with JSON[2] documents. It is thus possible to use our application as is with a different frontend.

The UI, in the form of a web page is implemented in Javascript and HTML 5. Bootstrap[3] was used to achieve a modern look. Users can submit their queries by entering them into textbox. Queries must be conform to the grammatical rules defined in our language, otherwise an error message from the parser will be printed.

To facilitate the usage of the application without knowing the grammar, a form available by clicking on the "Assisted query creation" button allows the user to perform a simple query composed of SELECT, JOIN, and WHERE easily. The user can simply fill out the blanks.

After successfully generating a query, you can see the Querying ID, status, Number of results, Start time, Finish time. If you generate many queries, a list of all the queries is displayed. You can refresh the list by clicking on the corresponding button. The list result also automatically refreshed every 15 sec.

By clicking on a query, the corresponding query detail results will pop out. In the detail window you could see the query string and the final query result. You can also see the subtasks' information, like subtask Task ID, Status, Operator, Generated Hits, Completed hits and every subtask results. If you are not interested in the query any more, you can easily abort the query by clicking the Abort button at the bottom of the window.

2.6 Performance Analysis

In general, the runtime of the queries directly correlates with the complexity of them. It turns out that the general base price we have chosen for most of the tasks, $ 0.01, is not a high enough incentive for a lot of workers to work on our HITs. Increasing this price by 500 % to $ 0.05 increased the rate of answers by about 80 - 90 %. We suspect that a lot of the workers use a filter on the mturk platform to only display tasks paying more than a given threshold.

One would think that a higher reward would lead to more effort on the workers side when answering the HITs. Interestingly, this is not the case. Our experiments show that on average, there is no substantial, positive impact on the correctness of results when paying more. Actually, for certain tasks like the data extraction, a slight decrease in answer quality can be observed. One has to note that those results could as well be explained by statistical variance.

[1] Our application embeds the Grizzly HTTP server `https://grizzly.java.net/`

[2] `http://json.org/`

[3] Bootstrap is a widely used front-end framework: `http://getbootstrap.com/`

On the other hand, we have observed a considerable impact by the formulation of the questions sent to the HITs. The workers seem to have an easier time with tasks using short sentences and simple English, much like having a To-do list to work on. Instead of just trying one's luck and and submitting a HIT even if they don't really understand what to do, workers are more inclined to just ignore tasks they don't understand. This could be explained by the incentive to keep their personal "approval rate on assignments" metric consistently high.

Fine-tuning our question formulation increased performance by about 60 % and answer quality by about 13 %. This could be partly explained by the ever increasing amount of foreign workers whose native language isn't English looking for tasks on the mturk platform.[4]

To further calibrate and tweak the HIT creation, we propose using some sort of sentiment analysis on mturk to find the optimal formulations.

We depend as well on other implementation specificities : the FROM step, for instance, can lead to huge accuracy differences and should be finely tuned and verified. This reinforces the necessity for our HITs to adapt to the behaviour or mturk workers.

3 Future improvements and conclusion

The possibilities of improvement for this project are countless but the following points seem to be the most relevant for the moment by order of importance:

- Add a verification phase to the very first task, which is usually the "FROM" task. This step is critical for the whole query due to the fact that if the website returned by the very first worker is not of high enough quality, subsequent worker will have a lot of trouble extracting the required information and we won't be able to have any satisfying results.

- Implement a majority vote for some of the questions. The results will be more precise but also more expensive, as more workers need to be asked the same questions. Using a majority vote for multiple choice or true/false questions is trivial. For cases where free text answers or numeric answers are expected, implementing a majority vote will be more sophisticated, as the domain of possible answers is sheer endless.

- Implement more operators.

- Use qualifications to get more reliable results. This would mean that a set of sample questions has to be created, for which the answers are already known. Workers are then at first asked to answer the sample questions to proof their abilities and then the real questions afterwards. It is then possibly to rank, weight or filter workers based on their performance and this would ultimately lead to better and more precise results.

[4] `https://archive.nyu.edu/bitstream/2451/29585/2/CeDER-10-01.pdf`

- Conduct a sentiment analysis survey to find the best formulations for the HIT questions.

Another great feature to add, but not so simple to implement, is the caching of previous results. Indeed if you retrieve results from a cache, rather than asking again on mturk, the result will be instantaneous, free and accurate.

In general there are three metrics one has to consider when crowdsourcing queries:

- **Accuracy** of the results
- **Cost** of getting a query answered
- **Time** it takes to obtain the results

It is very hard to achieve all of the three objectives, so one has to choose carefully some tradeoff between them. All in all, we can say that it was a really different experience to work with human workers instead of computers, simply because of the pretty high probability of failure and high indeterminism. Nonetheless, the crowdsourcing idea opens up a whole new world of possibilities and we are really excited about that.

Appendix

A Distribution of Work

- Florian Chlan : Communication with AMT, Analytics.
- François Farquet : Strategies implementation, interface, speaker for final presentation.
- Xinyi Guo : Parsing, interface
- Joachim Hugonot : Administrative, strategies implementation.
- Simon Rodriguez : Communication with AMT, Analytics.
- Kristof Szabo : Parsing, strategies implementation.
- Florian Vessaz : Set-up of the project, Communication with AMT, interface.
- Vincent Zellweger : Parsing.

B Details on the Language Parser

For most of the different steps of the hierarchy, that are optional elements, we can avoid them. For the explanation of the query, we will use the following definition

- Q1, Q2, ... are different queries
- E is an element
- Int is a number
- Str is a string. The string has to be placed between " " and is allowed to contain letters (capital or lower case), integer and _
- NL is some natural language. The natural language has to be inserted into brackets [ ] and is allowed to contain letters (capital or lower case), integer and space.
- Bool is the Boolean type. It could be either True or False

B.0.1 Root Node

The first hierarchy level is all the different links from a query with another query

- JOIN ON: Is a junction between two queries that joins the information of the two queries given a specific attribute.

  ```
  (Q1) JOIN (Q2) ON E
  ```

- IN: Returns all the different rows of Q1 that are in Q2 too

  ```
  (Q1) IN (Q2)
  ```

- NOT IN: Return all the different rows of Q1 that are not in Q2

  ```
  (Q1) NOT IN (Q2)
  ```

- INTERSECT: Return all the different rows that are both in Q1 and in Q2 (give the same result as the IN)

  ```
  (Q1) INTERSECT (Q2)
  ```

- UNION: Return all the different rows of Q1 plus the different rows of Q2

  ```
  (Q1) UNION (Q2)
  ```

- Nothing: If we don't need to use a link with another query

  ```
  Q1
  ```

B.0.2 Limit Node

We could or not ask for a limit on the number of items that we want to return.

- LIMIT: return a specific number of elements

  ```
  (Q1) LIMIT Int
  ```

- Nothing: If we don't need to limit the number of rows

  ```
  Q1
  ```

B.0.3 Order Node

This node is used to order the result in a specific order. The hierarchy is created to be able to limit the X first rows in a specific order. It won't be logic to have an order in an aleatory order

- ORDER BY: Order the rows on one or more elements in a specific order

  ```
  (Q1) ORDER BY E [ASC|DESC] [, E (ASC|DESC)]*
  ```

  The order could be in ascending (ASC) or descending (DESC) order, and it is possible to order in more than one element

- Nothing: If we don't need to order the rows

  ```
  Q1
  ```

B.0.4 Group Node

This node is used to group the different rows of the result given a specific element. It could be useful for SUM or other elements like this one. The hierarchy is created this way because the LIMIT and the ORDER have no impact on the GROUP BY but the GROUP BY gives another way to order

- OGROUP BY: Asks to group the different rows in a specific order

  ```
  (Q1) GROUP BY E
  ```

- Nothing: If we don't need to group the rows

  ```
  Q1
  ```

B.0.5 Where Node

This node is used to remove some rows that don't match a specific condition. The condition is also parsed by the system and is explained in a subsection. The hierarchy is here a little bit tricky and doesn't give us as possibility as SQL, since we couldn't put condition on the sum of elements that come after the GROUP BY. We choose to implement the WHERE before the GROUP BY because in most of the case, we will group elements that match a condition, and not match a condition on grouped element.
It is one of the place in the parsing part where optimisation could occur.

- WHERE: The bunch of conditions that has to be fulfiled

  ```
  (Q1) WHERE Condition
  ```

- Nothing: If we don't need to ask a specific condition

  ```
  Q1
  ```

Condition The bunch of conditions can accept different kind of stuff. These different conditions can be alone, with another, recursive or with natural language.

- ¡ : Ask an element to be smaller than a condition. The condition has to be a number or a natural language

  ```
  E < [Int|NL]
  ```

- ¿ : Ask an element to be bigger than a condition. The condition has to be a number or a natural language

  ```
  E < [Int|NL]
  ```

- ¡= : Ask an element to be smaller or equal to a condition. The condition has to be a number or a natural language

  ```
  E < [Int|NL]
  ```

- ¿= : Ask an element to be bigger or equal to a condition. The condition has to be a number or a natural language

  ```
  E < [Int|NL]
  ```

- = : Ask an element to be equal to a condition. The condition can be a lot of elements. It can be a number, a string, some natural language or a new sub query. Warning: the natural language and the query have to return only one element (one item and one row) to be consistent. It is not possible to match the equality of an element with a bunch of data.

  ```
  E = [Int|NL|Bool|Q1]
  ```

- IN : Ask an element to be inside a list of elements. It is quite the same as = since the data cab be in a set of elements. Warning : some restriction may still apply. Since the query or the natural language are allowed to give a set of data, it has to give a set of single data, not couple or something else. For example, it is possible to check a name in a set of name, not in a set of couple (name, first name).

  ```
  E IN [([Int|NL|Bool|Q1] [, [Int|NL|Bool|Q2]]*|NL|Q3]
  ```

  The list of elements can be

  - an explicit set

    ```
    (cond1, cond2, cond3, ...)
    ```

  - some natural language
  - another query

- NOT IN : Exactly the same functionality as IN but ask the element to be not in the set.

  ```
  E NOT IN [([Int|NL|Bool|Q1] [, [Int|NL|Bool|Q2]]*|NL|Q3]
  ```

- AND : The parser will parse this part recursively. It asks to have the condition C1 and the condition C2 to be satisfied to accept the condition.

  ```
  (C1) AND (C2)
  ```

- OR : The parser will parse this part recursively. It asks to have the condition C1 or the condition C2 to be satisfied to accept the condition.

  ```
  (C1) OR (C2)
  ```

B.0.6 Select Node

This part is the final part of the node, where we will select the different elements that we want. It could be either only natural language, a selection of fields in a natural language question or a selection of field on a sub query. Since the element cab be quite complex, we will resume it has Elements in the definition and explain them more specifically on another section.

- SELECT FROM NL: Has to extract the specified field from the question asked in natural language.

  ```
  SELECT Elements FROM NL
  ```

- SELECT FROM sub query: Has to extract the specified field from a sub query that will be computed from the root node.

  ```
  SELECT Elements FROM (Q1)
  ```

- NL: Ask something only in natural language

  ```
  NL
  ```

B.0.7 Elements Node

First of all, the different elements in the node element have to be a list of elements. This list is presented like that

```
(E1, E2, E2, ....)
```

We could ask different information about this element. It means:

- The sum of the different elements (given a group by)

  ```
  SUM(E)
  ```

- Have only distinct rows to avoid having ten times the same one (for example after an union or a bad worker)

  ```
  DISTINCT(E)
  ```

And finally the tuple can be of two different kinds

- numeric tuple

  ```
  NUMERIC E
  ```

- normal tuple

  ```
  E
  ```

B.0.8 Comments

During our test, we found some problems with this parsing. First of all the use of the parenthesis before and after the different elements is unsightly. The main point of it is that the recursive parsing creates some infinite loop (was looking for the fist element on the root node, that was recursive for JOIN and look for the first element of root note, ...). So we had to add the parenthesis that remove the first element when asking the recursive function.
Another problem that we tried to solve is that if we ask a query without the parenthesis, it seems to parse it correctly but forget the end of the query. For example :

```
SELECT (name) FROM (king of France) ORDER BY age_of_death
```

parsed well the SELECT and the FROM but seems to ignore the ORDER BY
Another problem is that when we try to parse too long query, the parser seems to stop working correctly. For example, a query using GROUP BY, ORDER and LIMIT seems to work well, but if we insert it in a JOIN, it crashes because it needs the WHERE, which is not needed.