Disambiguating Natural Language Queries with Tuples

Christopher Baik, Zhongjun Jin, Michael Cafarella
University of Michigan
Ann Arbor, MI, USA
{cjbaik, markjin, michjc}@umich.edu

ABSTRACT

Enabling natural language access to relational databases is challenging and often requires the disambiguation of a user's natural language query by selecting the target interpretation from among many candidate structured query interpretations. Existing approaches such as natural language explanations are promising, yet in certain situations users may benefit from an alternate clarification mechanism. We propose the *distinguishing tuple interaction model* to help the user disambiguate candidate queries and provide a formal problem definition. We introduce a solution strategy involving a greedy algorithm and demonstrate in selected experiments that our algorithms can reduce the number of user interactions over state-of-the-art baseline approaches.

PVLDB Reference Format:
Christopher Baik, Zhongjun Jin, and Michael Cafarella. Disambiguating Natural Language Queries with Tuples. *PVLDB*, 13(xxx): xxxx-yyyy, 2019.

1. INTRODUCTION

With the rise of virtual assistants such as Google Assistant, Siri, and Alexa, voice search is increasingly becoming a popular medium for users to interact with devices. ComScore estimates that by 2020, 50% of all searches will be voice searches[1]. At the same time, relational databases are far and away the most popular type of database in use today[2], given their reliability and ability to process transactions quickly. The times seem to naturally call for an integration of the two technologies, where a user would be able to invoke a "skill" on Alexa or "action" on Google Assistant to issue queries on a large relational database.

Unfortunately, enabling such an interaction is an open challenge, due to the difficulty of natural language (NL) understanding and "bridging the semantic gap" [1] between a user's specification and the formal SQL query in the context of a specific database schema. As a result, NL interfaces typically generate multiple candidate queries (CQs), as demonstrated in the following example.

EXAMPLE 1. *Sharon has been a car parts dealer in the USA for 15 years and has access to a relational database of part sales that a consulting firm created for her. After hearing recent news that tariffs would be enforced on goods flowing into and out of China, she wants to know which of her largest customers would be affected to inform them.*

Unfortunately, she has little knowledge of SQL or of the database schema she is querying. As such, she uses a NL interface that was set up for her on the database to issue the query: "What are the names and addresses of those in China who bought more than $10,000 from us?"

Internally, the NL interface tries its best to resolve the ambiguities in the query. In particular, "those in China" can refer to either customers or suppliers, and the amount "$10,000" can refer to various price fields. Some sample CQs are:

1. ```
   SELECT s.name, s.address
   FROM supplier s
     JOIN partsupp ps ON ps.sid = s.sid
     JOIN part p ON p.pid = ps.pid
   WHERE p.price > 10000
     AND s.address LIKE '%China%'
   ```
   ***Meaning**: Select the name and address of suppliers selling parts costing more than $10,000 with an address containing the substring 'China'.*

2. ```
   SELECT c.name, c.address
   FROM customer c JOIN nation n ON c.nid = n.nid
     JOIN order o ON o.oid = c.cid
   WHERE o.price > 10000 AND n.nation = 'China'
   ```
   ***Meaning**: Select the name and address of customers in the nation China who made orders (i.e. collections of line items) of more than $10,000.*

3. ```
   SELECT c.name, c.address
   FROM customer c JOIN nation n ON c.nid = n.nid
     JOIN order o ON o.oid = c.cid
     JOIN lineitem li ON o.oid = li.oid
   WHERE li.price > 10000 AND n.nation = 'China'
   ```
   ***Meaning**: Select the name and address of customers in the nation China who made an order with a line item costing more than $10,000.*

A full list of the top 20 SQL CQs are directly displayed to Sharon, who is promptly overwhelmed by the options and finds it difficult to select her target query.

[1] https://www.campaignlive.co.uk/article/just-say-it-future-search-voice-personal-digital-assistants/1392459
[2] https://db-engines.com/en/ranking_categories

Copyright is with the authors.
1st Int'l Workshop on Conversational Access to Data (CAST)
in conj. with the 45th Int'l Conference on Very Large Data Bases (VLDB)
Los Angeles, California - August 26-30, 2019.
   ```

	name	address	CQs
✗	Steeler Car Parts	555 China St, Pittsburgh, PA	1
✓	Beijing Auto Parts	Beijing, China	2, 3
	Great China Auto	Shanghai, China	2, 3
✗	Guangdong Auto	Guangzhou, China	2

Table 1: Example distinguishing tuple interaction.

As demonstrated above, NL queries often contain ambiguity that are difficult even for humans to interpret. One promising means of disambiguation is presenting NL explanations. However, existing approaches demand costly overhead from the database administrator, whether in manually mapping attribute pairs to NL phrases [5], mapping database entities to an ontology [5], or requiring that the user's initial query is grammatically sound [3]. Automated default approaches can help to circumvent this overhead, but these can lack the precision to produce distinct explanations for distinct queries with subtle differences. For example:

EXAMPLE 2. *Consider a schema with tables* `orders` *and* `companies`*, where* `orders` *contains two attributes (*`cid, sid`*) which are both foreign keys to the primary key* `id` *of* `companies`*. A user issues the NL query: "Find all companies in the United States who ordered from X." Two CQs result:*

1. ```
   SELECT c1.name FROM companies c1
     JOIN orders o ON o.cid = c1.id
     JOIN companies c2 ON o.sid = c2.id
     WHERE c2.name = 'X'
   ```
   ***Meaning**: Companies that **bought an order from** X.*

2. ```
   SELECT c2.name FROM companies c1
     JOIN orders o ON o.cid = c1.id
     JOIN companies c2 ON o.sid = c2.id
     WHERE c1.name = 'X'
   ```
   ***Meaning**: Companies that **sold an order to** X.*

Existing NL explanation mechanisms are unable to produce distinct explanations for these two CQs without additional annotation, specifically because the meanings of foreign keys `cid` *and* `sid` *are unclear for automated approaches.*

Consequently, we argue that clarification mechanisms beyond explanations are often necessary to disambiguate NL queries. We propose the *distinguishing tuple interaction model, which can be used exclusively or in conjunction with existing explanation approaches* to clarify the user's intent.

Interaction Model — We leverage the *distinguishing tuple* (DT) interaction model to help users select a target query from a CQ set produced by a NL interface. The system suggests tuples from the result set of the CQs and asks the user whether their target query should contain them. The model aims to conserve user effort by *distinguishing multiple CQs at once* given user feedback on the suggested tuples.

In Example 1, the system could present the example tuples in Table 1 and ask Sharon whether her desired query should produce each tuple. Sharon **rejects** (✗) the first tuple because the company is clearly not in China, eliminating CQ1. She also rejects the fourth tuple, knowing that Guangdong Auto only ever purchases small parts, and so eliminates CQ2. She **accepts** (✓) the second tuple, remembering that she sold an expensive part to Beijing Auto Parts earlier in the year. She **ignores** the third tuple because she can't remember her interactions with Great China Auto. Sharon's feedback would then be ingested by the system to determine that only Sharon's target query, CQ3, remains.

The DT interaction model has several advantages. First, the model can be a helpful ***companion to NL explanations*** for clarification purposes. Second, tuples are a ***common representation*** used in various interaction models [2] and ***requires no user expertise*** in SQL or the schema. Finally, given a database instance, a tuple can ***precisely distinguish two queries*** so long as such a tuple exists.

Of course, the effectiveness of the interaction model hinges on the user's knowledge of the database. Technical users familiar with the database are likely to possess the requisite knowledge to label most tuples. For such users, the DT model may be a convenient alternative to the time-consuming process of manually examining candidate SQL queries. On the other hand, non-technical users with domain expertise, such as Sharon (Example 1), may also have enough knowledge of the data to "know it when they see it." For these users, the interaction model serves a more critical function in enabling them to synthesize a SQL query without training in SQL or external assistance.

Technical Challenges — Given our interaction model, we want to save the user time and effort by arriving at their target query while displaying as few tuples to them as possible. This entails that we select the smallest set of tuples to whittle down the CQ set to the target query. As we show later, this problem is NP-hard.

In addition, since the suggested tuples can only be retrieved by executing CQs on the database, this process may require the user to wait a long time for CQs to execute, depending on the size and schema of the database and the CQ workload. We aim to reduce the time to select a tuple by intelligently avoiding a full execution of all CQs.

In summary, our technical challenges are to: (1) *minimize the number of tuples presented to arrive at the target query*, and (2) *minimize the system execution time to discover those tuples.*

Our Approach — We minimize the number of tuples presented to the user by constructing an *optimal split tree*, which is a flowchart of potential tuples the system presents to the user depending on the user's feedback. We develop variants of a *greedy algorithm* for split tree construction.

Contributions — We offer the following contributions:

- We introduce the *distinguishing tuple* interaction model as a means of disambiguating NL queries. We provide a *formal definition* of the MINDISTTUPLES problem of minimizing user effort in the interaction model.
- We introduce a solution strategy involving a *greedy algorithm with branch-and-bound and heuristic variants.*
- We demonstrate in selected experiments that our algorithms can *reduce the number of interactions with the user over state-of-the-art baseline approaches.*

2. OVERVIEW

2.1 Interaction Model

The user provides a NL specification of their target query, which the interface translates to a set of CQs. The system selects a tuple from the result sets of the CQs, and presents it to the user. The user can either *accept*, *reject*, or *ignore* the presented tuple. An accepted tuple is expected by the user in the output of their target query, while a rejected tuple is expected not to be in the output of the target query. If the

user ignores a tuple, then an alternate tuple is provided to the user. The system prunes the set of CQs according to the user's feedback, then returns a tuple from the remaining CQs. This process iterates until the system can arrive at a query satisfying all of the user's feedback.

2.2 Problem Definition

In this section, we introduce some necessary concepts and our problem definition. All concepts and definitions provided are in the context of a fixed existing database.

2.2.1 Concepts

First, we define candidate queries:

DEFINITION 1. *A **candidate query** (CQ) q is a conjunctive query with a weight $w(q) > 0$ producing a result set of tuples $R(q)$.*

The weight $w(q)$ of a CQ models the confidence that a certain CQ is the target query. Existing NL interfaces [5, 6] commonly produce such confidence scores to rank their result candidate queries. *We do not assume that weights provided are accurate,* but develop our approach under the assumption that they are more helpful than harmful if provided. If there are no helpful sources of weight information, then the system always has the option of assigning an identical default weight to all CQs.

We denote a set of CQs by $\mathcal{Q} = \{q_1, \ldots, q_n\}$ and extend the definition of result sets and weights to CQ sets such that $R(\mathcal{Q})$ is defined as the union of all result sets of CQs in $\mathcal{Q}$ and $w(\mathcal{Q})$ is the sum of the weights of all the CQs. $\mathcal{Q}^t_\top$ is the subset of CQs in $\mathcal{Q}$ that produce the tuple t in their result set and $\mathcal{Q}^t_\bot$ is the subset of CQs in $\mathcal{Q}$ that do not produce t in their result set. We also use $\mathcal{Q}^t$ as shorthand for $\mathcal{Q}^t_\top$.

A set of queries is considered to be **equivalent** if all member queries produce the exact same result set with respect to the fixed database. Finally, the domain of all possible tuples is given by the symbol $\mathbb{T}$.

2.2.2 Formal Problem

Our goal is to *minimize the number of tuples presented to the user in the DT model.* Given our setting where the target query is unknown a priori and can only be discovered by soliciting user feedback on tuples, we define a *distinguishing tuple set* as a set of tuples which identifies CQs consistent with the user's feedback:

DEFINITION 2. *Given a CQ set $\mathcal{Q}$, a user function $\mathcal{U} : \mathbb{T} \to \{\top, \bot, \varnothing\}$, and an equivalent set of target queries $\hat{\mathcal{Q}}$, a **distinguishing tuple set** is a set of tuples $S = \{t_1, \ldots, t_m\}$ such that each tuple $t_i \in R(\mathcal{Q})$ and:*

$$\bigcap_{t_i \in S} \mathcal{Q}^{t_i}_{\mathcal{U}(t_i)} = \hat{\mathcal{Q}} \tag{1}$$

In Definition 2, we model the user as a function that takes a tuple as input and returns $\top$ (i.e. accepted tuple), $\bot$ (rejected), or $\varnothing$ (ignored) as output. We use an equivalent set of target queries $\hat{\mathcal{Q}}$ instead of a single target query because the DT model is unable to distinguish two CQs that produce identical result sets, and in a fixed database setting we can consider such queries to be identical. The DT model also requires that the result set of all queries in $\hat{\mathcal{Q}}$ are non-empty.

Our main problem can now be formalized as follows:

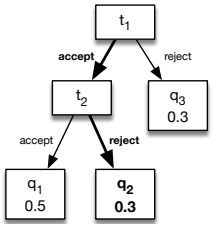


Figure 1: Split tree: bold path leads to target query q_2.

PROBLEM 1 (MINDISTTUPLES). *Given a CQ set $\mathcal{Q}$ and a user function $\mathcal{U} : \mathbb{T} \to \{\top, \bot, \varnothing\}$, find the smallest distinguishing tuple set S.*

MINDISTTUPLES is NP-hard. This can be proved by showing that the decision problem variant of MINDISTTUPLES is in NP and that the set cover problem can be reduced to it in polynomial time. We omit the full proof for space reasons.

3. GENERAL APPROACH

In this section, we introduce our solution strategy to tackle the MINDISTTUPLES problem.

3.1 Split Trees

We adopt the *split tree* [4] to represent the space of interactions in the DT model. The split tree is a type of flowchart that models various possible interaction paths composed of system-suggested tuples and user feedback (i.e. accepting or rejecting the suggested tuples). Formally:

DEFINITION 3. *A **split tree** for CQ set $\mathcal{Q}$ is a rooted binary tree $\mathcal{T}$ in which each node v has a label $L(v)$ such that:*

- *Each CQ $q \in \mathcal{Q}$ has exactly one corresponding leaf node $v_\ell \in \mathcal{T}$ labeled with q: $L(v_\ell) = q$ and each internal node $v_i \in \mathcal{T}$ is labeled with a tuple: $L(v_i) \in R(\mathcal{Q})$.*
- *Any CQ q in the left subtree of an internal node v_i produces the tuple $L(v_i)$ in its result set $R(q)$, while any CQ in the right subtree does not produce $L(v_i)$.*

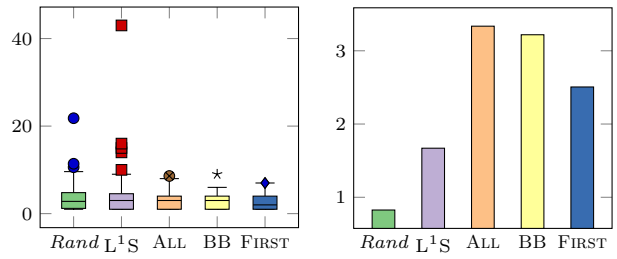
As shown in Figure 1, a single instance of the DT model can be mapped to a path from the root to the leaf labeled with the target query. At each internal node, the left edge is taken if the user accepts the tuple or the right edge if the user rejects it. If the user ignores the tuple, we simply remove it from the pool of candidate tuples and present another tuple. If we enumerated all root-to-leaf paths from all possible split trees, it would be equivalent to enumerating the entire search space of candidate DT sets for MINDISTTUPLES.

3.1.1 Optimal Split Tree

One of the reasons why MINDISTTUPLES is difficult is that the system has no way of knowing which CQ is the target query apart from a trial-and-error approach of feeding tuples to the user. We tackle this challenge by minimizing the root-to-leaf path length for all CQs on a single split tree.

Since the weights of CQs provide information on which CQs are most likely to be the target query, we include this and define the cost of a split tree as the *total weighted cost*:

$$c(\mathcal{T}) = \sum_{i=1}^{n} l_i w(q_i) \tag{2}$$



(a) Iterations for each task (b) System runtime (s)/iter

Figure 2: Results on the IMDB dataset.

where l_i is the length of the path from the root to the leaf node labeled with q_i. While other cost functions such as the worst-case cost of any $q_i \in \mathcal{Q}$ are possible alternatives, we prefer the weighted cost because it takes into account any information provided by the user and/or NL interface to prioritize examining CQs with higher weights.

Using this cost metric, our strategy is to approximate MinDistTuples by discovering a single optimal split tree, consequently limiting the candidate DT sets to be explored to the root-to-leaf paths of this split tree:

Problem 2 (OptSplitTree). *Given a set of CQs $\mathcal{Q}$, find the split tree $\mathcal{T}$ minimizing $c(\mathcal{T})$.*

While this problem is also demonstrated to be NP-hard [4], it allows us to move toward a feasible solution strategy.

3.2 Greedy Algorithm

The space of possible split trees that can be generated given a set of CQs is prohibitively large for most tasks, and so we adopt the greedy approach described in [4] to approximate the optimal split tree in an algorithm we call GreedyAll. This greedy approach still requires a full execution of all CQs in order to select an optimal tuple, so we additionally develop branch-and-bound (GreedyBB) and heuristic-based (GreedyFirst) variants of the approach to avoid a full execution and conserve system runtime.

4. SELECTED EXPERIMENTS

We evaluated the effectiveness of our three algorithms with a simulated user that correctly accepted or rejected any tuples presented to it. The input for each task was a set of CQs with equal weight ($w(q) = 1$) with a single target query. For each iteration of the task, the system selected a tuple from the result sets of the CQs and presented it to the user. The system iteratively eliminated CQs given the user's feedback until the system narrowed down the CQ set to a single CQ, which was returned as the target query.

We used the IMDB dataset [6] which contains corresponding pairs of natural language and resulting SQL queries. We executed the natural language queries for each task using a natural language interface based on the design of [6] to produce a set of candidate queries which vary in terms of selected projections, predicates, and join paths.

We compared our algorithms, GreedyAll (All for short), GreedyBB (BB), and GreedyFirst (First) to a baseline approach of randomly selecting any tuple (*Rand*) and to the L^{1}S approach from [2]. We ran 5 trials for each algorithm on each task and averaged the results to get a sense of the average performance.

Figure 2a displays the number of iterations of user feedback on tuples required to find the target query. The box-and-whisker plots display the minimum, first quartile, median, third quartile, and maximum values over all tasks, along with any outliers (values greater than the upper quartile by at least 1.5 times the interquartile range or lesser than the lower quartile by at least that amount) as individual points. Notably, our three algorithms **avoid the worst-case outliers that occur with both *Rand* and L^{1}S**.

Figure 2b displays the mean system runtime per iteration over all tasks. *Rand* has the least overhead because it requires that a single random query is selected and executed with a top-1 query. L^{1}S also runs within 2 seconds per iteration. All, and BB have comparable runtimes over 3 seconds per iteration, while First reduces it to 2.5 seconds per iteration. It is important to note that the total runtime for a task is calculated by $n_{iters}(t_{user} + t_{sys})$, where t_{user} and t_{sys} are the average user response time and system runtime per iteration. In practice, we expect the user response time to be the dominant factor in total task time, and find that it is a reasonable tradeoff for our algorithms take slightly more system runtime than the baselines in order to significantly reduce the total number of iterations.

5. CONCLUSION AND FUTURE WORK

In this paper, we proposed the distinguishing tuple interaction model to enable a user to disambiguate candidate queries in a NL interface. We introduced a general solution strategy involving a greedy algorithm, and demonstrated in selected experiments that our algorithms can reduce the number of interactions with the user over baseline approaches. For future work, we intend to refine our algorithms to improve their performance and extend our experiments to a larger set of benchmarks.

6. REFERENCES

[1] C. Baik, H. V. Jagadish, and Y. Li. Bridging the semantic gap with sql query logs in natural language interfaces to databases. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 374–385, April 2019.

[2] A. Bonifati, R. Ciucanu, and S. Staworko. Learning join queries from user examples. *ACM Transactions on Database Systems (TODS)*, 40(4):24, 2016.

[3] D. Deutch, N. Frost, and A. Gilad. Provenance for natural language queries. *Proceedings of the VLDB Endowment*, 10(5):577–588, 2017.

[4] S. R. Kosaraju, T. M. Przytycka, and R. Borgstrom. On an optimal split tree problem. In *Workshop on Algorithms and Data Structures*, pages 157–168. Springer, 1999.

[5] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Özcan. Athena: an ontology-driven system for natural language querying over relational data stores. *Proceedings of the VLDB Endowment*, 9(12):1209–1220, 2016.

[6] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. Sqlizer: Query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):63, 2017.