Disambiguating Queries in Conversational Interfaces

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

ABSTRACT

Enabling conversational access to relational databases is a challenging task and often requires the disambiguation of a user's natural language query by selecting the target interpretation from among many candidate structured query interpretations. Performing this disambiguation on a conversational interface is difficult as such interfaces are often implemented on devices with small screens or no screen at all, requiring system responses to be succinct and to occupy little screen real estate. We propose the *distinguishing tuple interaction model* to help the user disambiguate candidate queries via a conversational interface and provide a formal problem definition. We introduce a general solution strategy involving a greedy algorithm with branch-and-bound and heuristic variants, and demonstrate in selected experiments that our algorithms can reduce the number of interactions with the user over baseline approaches.

PVLDB Reference Format:
Christopher Baik, Zhongjun Jin, and Michael Cafarella. Disambiguating Queries in Conversational Interfaces. *PVLDB*, 13(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

With the rise of virtual assistants such as Google Assistant, Apple's Siri, and Amazon's Alexa, voice search is increasingly becoming a popular medium for users to interact with devices. A study of 1400 smartphone users performed by Google in 2014 found that 55% of teens and 41% of adults use voice search at least once a day[1], while ComScore estimates that by 2020, 50% of all searches will be voice searches[2].

[1]https://www.prnewswire.com/news-releases/teens-use-voice-search-most-even-in-bathroom-googles-mobile-voice-study-finds-279106351.html
[2]https://www.campaignlive.co.uk/article/just-say-it-future-search-voice-personal-digital-assistants/1392459

Proceedings of the VLDB Endowment, Vol. 13, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

At the same time, relational databases are far and away the most popular type of database in use today[3], 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 understanding and "bridging the semantic gap" [1, 6] between a user's description of an information need and the formal SQL query in the context of a specific database schema. As a result, natural language interfaces to databases typically generate multiple candidate queries (i.e. SQL interpretations), 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 natural language 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 natural language interface tries its best to resolve the ambiguities in Sharon's query. In particular, "those in China" can refer to either customers or suppliers, and the amount "$10,000" can refer to various price fields. A few sample candidate queries 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%'
```
(i.e. Select the name and address of suppliers selling parts of 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'
```

[3]https://db-engines.com/en/ranking_categories

(i.e. Select the name and address of customers in the nation China who made orders 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'
```
(i.e. Select the name and address of customers in the nation China who made an order with a line item costing more than $10,000.)

As demonstrated above, natural language queries often contain ambiguity that are difficult even for humans to interpret. Some viable approaches to disambiguation include asking some clarifying questions ("Were you referring to customers or suppliers?"), applying domain knowledge, or executing some queries and validating results to select the appropriate query interpretation.

We propose the *distinguishing tuple* interaction model to facilitate disambiguation of natural language queries in the context of conversational interfaces, which are often implemented on devices with small screens or no screen at all.

Interaction Model — While non-technical users may lack knowledge of SQL or the schema, they often have domain knowledge that can help disambiguate candidate queries (CQs). One specific means by which a system can elicit this knowledge is to display examples from the result sets of CQs to the user, who can provide feedback on whether those examples should or should not belong in the result set of their desired query. We call this the *distinguishing tuple* interaction model, and aim to conserve user effort by *distinguishing multiple CQs at once* given user feedback on the suggested tuple. Distinguishing tuples can be used exclusively or in conjunction with orthogonal techniques such as natural language explanations [3] to clarify the user's intent.

In Example 1, the system could present the example tuple `("USA Car Parts", "Chinatown, NY")` produced by the first CQ (and also possibly by other undisplayed CQs), and ask Sharon whether her desired query should produce that tuple: *To clarify, one possible query includes "USA Car Parts" in "Chinatown, NY". Is this consistent with what you were looking for?*. Sharon can then reply, "No," eliminating any CQs that produce that tuple. Conversely, if she replied "Yes," then only CQs producing the tuple would be preserved. She might also say, "I'm not sure," in which case alternate tuples could be provided to Sharon.

The distinguishing tuple interaction model has several benefits. First, providing feedback on tuples ***requires no user expertise*** in SQL or the database schema. Second, a tuple can ***precisely distinguish two queries*** given a specific database instance. Finally, both tuples and tuple feedback ***require little screen real estate*** and can easily be communicated via a conversational interface.

Technical Challenges — Given our interaction model, we want to save the user time and effort by arriving at their target query in as few iterations as possible. This entails selecting the shortest sequence of tuples that will narrow 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, the number of CQs, and the number of tuples returned per CQ. We aim to reduce the time to suggest a tuple to the user by intelligently avoiding a full execution of all CQs.

In summary, our technical challenges are to: (1) *arrive at the target query with a minimal number of iterations/tuples*, and (2) *perform each iteration in interactive time.*

Our Approach — We aim to minimize the number of tuples presented to the user by constructing an *optimal split tree*, which is a flowchart of potential tuples the system should present to the user depending on the user's feedback. We develop a *greedy algorithm* for constructing such a split tree, as well as *branch-and-bound* and *heuristic-based* variants of the approach, which improve on the runtime of the vanilla greedy algorithm.

Contributions — We offer the following contributions:

- We introduce the *distinguishing tuple* interaction model as a means of disambiguating natural language queries in a conversational interface. We provide a *formal definition* of the MinDistTuples problem of minimizing user effort in the interaction model.
- We introduce a general 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 baseline approaches.*

2. OVERVIEW

In this section, we provide an overview of the distinguishing tuple interaction model and a formal problem definition.

2.1 Interaction Model

The user begins by providing a natural language specification of their target query $\hat{q}$, 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 $\hat{q}$, while a rejected tuple is expected not to be in the output of $\hat{q}$. If a 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 again returns a tuple from the remaining CQs. This process iteratively continues until the interaction allows the system to arrive at a single CQ satisfying all of the user's feedback.

2.2 Problem Definition

In this section, we introduce some necessary concepts, then formalize our problem definition. We assume that 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 from a natural language interface that a certain CQ is the target

query. We note that the $w(q)$ value is not required to reflect a probability distribution and our problem setting does not require that $w(q)$ values sum to 1.

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 $\mathcal{Q}^t_\top$. 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 distinguishing tuple interaction 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 uniquely identifies a single CQ consistent with the user's feedback from a set of CQs:

DEFINITION 2. *Given a CQ set $\mathcal{Q}$ and a user function $\mathcal{U} : \mathbb{T} \to \{\top, \bot, \varnothing\}$, 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:*

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

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. Our main problem can now be formalized as follows:

PROBLEM 1 (MINDISTTUPLES). *Given a set of CQs $\mathcal{Q}$ and a user function $\mathcal{U}$, find the smallest distinguishing tuple set S.*

We make a few assumptions in this problem formulation in order to guarantee that a solution always exists to MINDISTTUPLES. First, we assume that the target query has a non-empty result set. Second, we assume that the user always provides feedback consistent with their target query. Finally, we assume that there is exactly one target query in the set $\mathcal{Q}$ and that no other CQ in $\mathcal{Q}$ has the exact same result set as the target query.

Unfortunately, solving this problem is non-trivial; in fact:

THEOREM 1. MINDISTTUPLES *is NP-hard.*

The theorem can be proved by demonstrating that the decision problem variant of MINDISTTUPLES is in NP and that SETCOVER [4] 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 overall solution strategies to tackle the NP-hard MINDISTTUPLES problem of minimizing the number of tuples we present to the user.

3.1 Split Trees

We introduce the *split tree* to represent the space of interactions in the distinguishing tuple interaction 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:

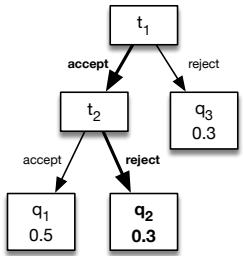


Figure 1: Example split tree. The bolded execution leads to q_2 as the target query.

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 distinguishing tuple interaction model can be mapped to a path from the root to the leaf labeled with the target query, where at each internal node, the left edge is taken if the user accepts the tuple, and the right edge if the user rejects it. While not shown in the figure, 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 distinguishing tuple sets for MINDISTTUPLES.

3.1.1 Optimal Split Tree

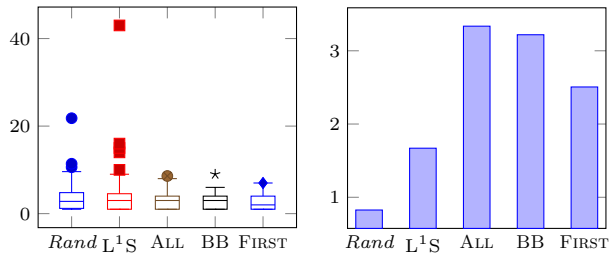
One of the reasons why MINDISTTUPLES is difficult is because the system has no way of knowing which CQ is the target query apart from a trial-and-error approach of feeding tuples to the user. We attempt to tackle this challenge by minimizing the root-to-leaf path length for all CQs on a single split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

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

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 natural language 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 distinguishing tuple sets to be explored to the root-to-leaf paths of this split tree:



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

Figure 2: Results on the IMDB dataset.

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 [5], 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 [5] 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 adopt a 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 [7] which contains corresponding pairs of natural language and SQL queries. We executed the natural language queries for each task using a natural language interface based on the design of [7] to produce a set of candidate queries which vary in terms of selected projections, predicates, and join paths. The original annotated SQL query was labeled as the target query for each task.

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^1S approach from [2]. We did not compare against the bottom-up and top-down algorithms from [2] because they were only applicable to join predicate CQ workloads, and also did not evaluate against L^2S as it leveraged a similar heuristic approach to L^1S yet was demonstrated in their evaluation to often be an order of magnitude slower than L^1S. 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^1S.

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^1S 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 conversational interface. We introduced a general solution strategy involving a greedy algorithm with branch-and-bound and heuristic variants, 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. *arXiv e-prints*, page arXiv:1902.00031, Jan 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] R. M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[5] 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.

[6] Y. Li and D. Rafiei. Natural language data management and interfaces: Recent development and open challenges. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1765–1770, New York, NY, USA, 2017. ACM.

[7] 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.