Improving a Semantic Parser through User Interaction

Pius von Däniken

January 2021

Contents

1 Introduction

The dawn of the information age in the latter half of the past century and the advent of the internet have thoroughly revolutionized the way humanity stores, retrieves, and interacts with knowledge and information. The written word, which had superseded oral traditions, has been the dominant store of knowledge for centuries. This legacy persists to this day, with the internet still largely consisting of so-called unstructured text. Of course language and, as a result, texts are anything but unstructured, with linguists dedicating their lives to prying open that underlying structure. The problem of querying and extracting knowledge from such texts with the help of computers is studied by the field of *Information Retrieval* (Mitra and Chaudhuri, 2000). The last twenty years have shown staggering progress in that field, propelling the likes of *Google* to their current behemoth status.

At the same time, new and so called structured ways of representing knowledge have been introduced to facilitate processing by computer systems. The most prominent are Relational Database Management Systems (RDBMS), which are based on the mathematical formalism of relational algebra introduced by Codd (1970, 1989). They organize knowledge as a collection of tables that can refer to each other. An alternate representation is the Resource Description Framework (RDF[1]), which represents knowledge in the form of triples of the form *subject - predicate - object*. In either case, interaction with stored knowledge is mediated by a *query language*, such as *SQL* in the case of RDBMS or *SPARQL* for RDF. To successfully retrieve an answer, a user not only needs to be able to wield those programming languages, but also needs to know exactly how the world, or domain, is modelled in the specific instance. This is usually referred to as the *schema* of the database.

While these technologies are part of any undergraduate computer science curriculum, very few people outside the field know how to use them, making the knowledge stored in such databases inaccessible to most. One way to close this accessibility gap is to provide a way to query such systems using natural language. Such *Natural Language Interfaces for Databases (NLIDB)* are an active area of research. The core task of an NLIDB system is to provide an answer to a question posed in natural language by querying the underlying database. For example, given access to the IMDb[2] database, the NLIDB system should be able to answer questions about movies, such as "What are the most popular movies?" We will give a brief glimpse of some of these systems in Section 2 and refer readers interested in a thorough overview to a recent survey by Affolter et al. (2019).

Unfortunately, current NLIDB systems are still far from perfect and there are many sources of errors. One of the most challenging problems is the inherent ambiguity of

[1] `https://www.w3.org/TR/rdf-mt/`
[2] `https://www.imdb.com/`

natural language. Therefore, it is often not enough to present the user with just the raw answer of the system, since the correctness of the answer by itself can be hard to judge without context. If our system produces "Battlefield Earth" as the answer to the question about popular movies, a user might not know that the answer is wrong. Even if the system gives an obviously wrong answer, like "John Travolta", the user has no guidance on how to rephrase the question to get the right answer. To remedy this, a NLIDB system can produce an explanation alongside its answer that gives insight into what might have gone wrong. At the most basic level, one could present the user with an SQL statement that was produced by the system, but this assumes that the user can understand SQL.

We previously developed an approach that produces a synthetic natural language question from the query produced by an NLIDB system (von Däniken et al., 2021). For our example question it would produce: "What are the names of movies with maximum popularity?", which is effectively a paraphrase of the original question. We showed that asking users whether the generated question is equivalent to their original question is a good proxy for the correctness of NLDIB output. In the present work, we take this one step further and let the user manipulate the generated natural language representation to, hopefully, correct the NLIDB output. We show that by adding this simple element of user interaction, we can increase the accuracy of the system from 56.1% to 83.2%.

2 Background

NLIDB. Research into NLIDB goes back to at least the 1960s with systems such as *Baseball* (Green Jr et al., 1961), which was used to answer simple questions about baseball games. Since then, various fields of computer science and artificial intelligence have contributed ideas and approaches.

Systems like *SODA* (Blunschi et al., 2012) are based on ideas from information retrieval. *SODA* extracts keywords from the user's question, such as table names and literals. It then matches these to entries in the database or the associated metadata. During this process, a keyword could match multiple concepts in the database or metadata. Therefore, all candidate sets of matches are ranked according to a heuristic, before being translated into SQL.

Athena (Saha et al., 2016) and its successor *Athena++* (Sen et al., 2020) use extensive linguistic analysis, such as dependency parsing and symbolic reasoning over a domain ontology to translate a human question to an intermediate representation they call *ontology query language*. The rule set is quite elaborate and extensive and *Athena++* is powerful enough to handle complex nested queries.

Finally, we point out one system in particular that is very similar to what we are proposing in this work, namely *TR Discover* (Song et al., 2015). It uses first order logic as the intermediate representation of a query and uses a feature-based context free

grammar to continuously parse an input question. To encourage the input strings to conform to its grammar, it provides suggestions while the user is typing their question. Affolter et al. (2019) refer to such systems as *grammar based*. While the framework we lay out in Sections 4 and 5 would conceptually allow for a very similar workflow , we instead investigate how this approach can be used as a post-processing step. We will allow the user to provide an arbitrary natural language utterance, retrieve the intermediate representation from a semantic parser, apply our grammar to produce a natural language representation of the query, and let the user manipulate that representation in a way that conforms to the grammar.

Semantic Parsing. In the natural language processing and computational linguistics communities the NLIDB problem is known as part of *Semantic Parsing*. In general, semantic parsing is the problem of mapping a natural language utterance to a formal semantic representation. For NLIDB, this corresponds to mapping a question to a query that can be executed on the database. More generally, this could also apply to translating natural language to any kind of programming language. The recent boom of neural network based machine learning models has led to variety of models for the task.

A seminal model for a majority of recent advances was *GrammarNet* (Yin and Neubig, 2017). The model was originally built to translate natural language descriptions to programs. Similar to models popular in machine translation, it uses an encoder-decoder architecture. The encoder is a recurrent neural network (RNN), such as LSTM (Hochreiter and Schmidhuber, 1997) or GRU (Cho et al., 2014), that will map the sequence of words in the input question to a high dimensional vector representation. The decoder network is another RNN that, starting from the encoder representation, will generate a sequence of production rules. These production rules correspond to a context free grammar and applying them, in order, will result in a parse tree for that grammar, such as an abstract syntax tree of some program or, in our case, a query. The model can be extended with a pre-training step inspired by auto-encoders. During pre-training, the text encoder is substituted by another encoder that is given the task of mapping a tree conforming to the grammar to a high dimensional vector. This makes it easier for the overall system to learn the structure of the grammar.

In the meantime, various extensions of this framework have been considered. A recent culmination of this refinement is *RAT-SQL* (Wang et al., 2020). One issue when using *GrammarNet* for text-to-SQL parsing is that it is unaware of the underlying database schema. Bogin et al. (2019) address this issue by providing a learned representation of the schema as an additional input. That representation is based on a graph neural network. The acronym *rat* in *RAT-SQL* stands for relation aware attention. Its encoder consists of several self attention layers (Vaswani et al., 2017) that have been augmented with information about the schema. Another common extension, which is also employed by *RAT-SQL*,

is to use word representations from large language models, such as *BERT* (Devlin et al., 2019).

Datasets. Over the years, various benchmark datasets for text-to-SQL semantic parsing have been created. Such datasets provide a collection of natural language questions and their associated SQL queries. Depending on the dataset, the number of unique SQL queries can be smaller than the number of natural language utterances if paraphrases are included. An early dataset is the *ATIS* corpus (Dahl et al., 1994), which contains a database and questions about flights between cities in North America. The most prominent dataset today is *Spider* (Yu et al., 2018). It contains 10181 natural language questions corresponding to 5693 SQL queries, spanning 200 databases. Notably, it features an online leaderboard[3] and is currently the de-facto standard for assessing the performance of a semantic parser. Another SQL dataset is *WikiSQL* (Zhong et al., 2017). It is based on data from *Wikipedia*[4]. Despite its size of over 80000 utterances, it is less popular due to the queries being simple and not containing complex *join* operations. *LC-QuAD 2.0* (Dubey et al., 2019) is an example of a dataset containing SPARQL queries. It is based on Wikidata and DBpedia, two large online knowledge graphs. In this work, we will use the *OTTA* corpus by Deriu et al. (2020). They propose an intermediate query representation named *Operation Trees* that represent a simplified SQL syntax. We will give more details on this formalism in Section 3.

Back-translation. The goal of translating a structured query back into natural language is to provide non-technical users with an explanation of the query. The main approach, so far, has been to define template phrases that are combined when traversing the AST of the query. Systems based on this idea have been developed for SQL (Koutrika et al., 2010), SPARQL (Ngonga Ngomo et al., 2013), and Operation Trees (von Däniken et al., 2021). This work is largely based on Operation Trees and we will give a detailed overview in Section 4. Wang et al. (2015) show that such systems can also be leveraged to build a semantic parser. They generate natural language utterances for queries expressed in lambda calculus. They then let humans paraphrase the generated utterances and train a model to tell whether a given paraphrase and generated utterance match. They can then use that model to search high scoring queries for a given human question, resulting in a semantic parser.

This task can also be tackled by applying deep learning methods. One approach is illustrated by Xu et al. (2018). They use a graph neural network to encode the AST of a SQL query and a text decoder to generate the corresponding text. They show that this outperforms more naive approaches that encode the query sequentially. The main

[3] `https://yale-lily.github.io/spider`
[4] `https://en.wikipedia.org/wiki/Wikipedia`

promise of neural approaches is that they can produce more fluent utterances, especially for complex queries where the template based approaches often generate unwieldy and unnatural utterances. On the other hand, neural methods cannot yet guarantee that the generated utterance is complete in respect to the query. There is nothing preventing them from, for example, forgetting to generate text to express certain nodes.

User Interaction. Recently, interest has grown in incorporating user interaction to improve semantic parsers. Labutov et al. (2018) and Elgohary et al. (2020) consider a setting where the user poses their question, the system then provides an explanation of its parse, and the user can ultimately provide corrective feedback in natural language. As an example, lets assume the user asked: "What are Brad Pitt's most popular movies?" and the parser misunderstood it as: "What are the names of movies starring Brad Pitt?". The user will then be able to provide feedback similar to: "Yes, but I meant those with maximum popularity." The authors investigate various ways to incorporate this feedback in the training process to improve the underlying parser. Yao et al. (2019) extend this by explicitly modelling the interaction between the user and the system. They incorporate a model that tries to predict errors in the parse and proactively prompts the user for feedback. Their approach allows for several rounds before the final answer is returned. They further (Yao et al., 2020) refined the approach by embedding it in an imitation learning framework. The idea of prompting the user for specific feedback is also present in *Photon* (Zeng et al., 2020). It will detect confusing spans in the input question and prompt the user to rephrase them.

In contrast, our system will only allow the user to manipulate the query representation in a limited way. The parsed query will be translated into natural language and the user will be presented with a user interface that allows them to manipulate the synthetic question in a manner that respects the underlying grammar.

3 Operation Trees

The *Operation Tree (OT)* representation for structured queries was introduced by Deriu et al. (2020), as way to collect pairs of natural language questions and corresponding queries in a more efficient manner. An OT is a binary tree where each node represents an operation to be executed on the result sets of its children, with leaf nodes reading data from the underlying database. Figure 1 shows an example OT based off of a database related to movies and represents the question:"What is the average revenue of movies produced in Japan?"

Formally, OT follow the context free grammar shown in Table 1. The nodes correspond to operations known from relational algebra with some extensions. Therefore, there is a close correspondence to other query representations, such as SQL.

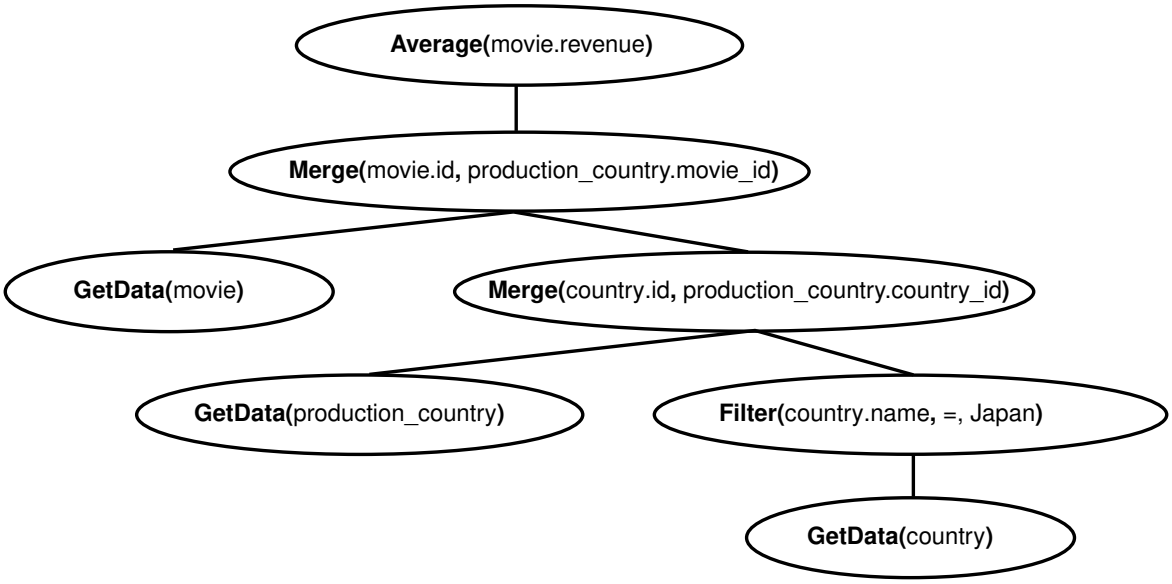

Figure 1: *Operation Tree* for the question: "*What is the average revenue of movies produced in Japan?*" The root node is the final operation, which averages the movie revenue column. There are two merge (join) operations involved that combine the countries and movies via the production-country relation table. The filter selects all countries with the name "Japan." (Figure taken from von Däniken et al. (2021))

S	::=	$Done(R) \mid IsEmpty(R) \mid Count(R) \mid Sum(T, A) \mid Average(T, A)$
R	::=	$ExtractValues(T, A)$
T	::=	$Min(T, A) \mid Max(T, A) \mid Distinct(T) \mid Filter(T, A, OP, LIT) \mid Merge(T, T, A, A) \mid GetData(TN) \mid Union(T, T, A, A) \mid Intersection(T, T, A, A) \mid Difference(T, T, A, A) \mid AverageBy(T, A) \mid SumBy(T, A) \mid CountBy(T, A)$
TN	::=	table name
A	::=	attribute
OP	::=	$<\mid>\mid\leq\mid\geq\mid=\mid\neq$
LIT	::=	literal

Table 1: Full OT grammar as defined by Deriu et al. (2020). The terminals *table name*, *attribute*, and *literal* stand for the sets of tables, columns of those tables, and entries in the database respectively.

The following is a short overview of the function of the different nodes:

- Sum(T, A) and Average(T, A) will return the sum or average of the values in column A of the result of the child query T.
- Count(R) will return the cardinality of the child query.
- Done(R) will return the whole result set of the child query.
- IsEmpty(R) returns whether the result of the child query is empty or not. It is used

to represent yes-no questions.

- ExtractValue(T, A) selects the column A of its child query.
- GetData(table_name) reads the whole table referred to by table_name from the database.
- Filter(T, A, OP, LIT) corresponds to a *where* clause in SQL. It selects rows of the child query by comparing the value in column A to the literal LIT with the operator OP.
- Min(T, A) and Max(T, A) will select all rows of the child query with minimum or maximum value for column A.
- Distinct(T) selects only unique rows of the child query.
- Merge(T_1, T_2, A_1, A_2) will create an inner join of the result sets T_1 and T_2 using columns A_1 and A_2 respectively.
- *Union*, *Difference*, and *Intersection* correspond to set operations on the specified columns of their sub-queries.
- *SumBy*, *AverageBy*, and *CountBy* group and aggregate the result by some column A.

3.1 The OTTA Corpus

The traditional approach to collecting data for NLIDB systems is to have experts write SQL queries for a given natural language question. For example, the *Spider* corpus (Yu et al., 2018) was created in this manner. The creators of *Spider* let computer science students create SQL queries and corresponding natural language questions for every database.

To create the *OTTA* corpus, Deriu et al. (2020) automated the query generation process by sampling suitable OT directly from the grammar in Table 1. Deriu et al. (2020) then had students familiar with databases write matching natural language questions. As a result of this, they could considerably speed up the annotation process.

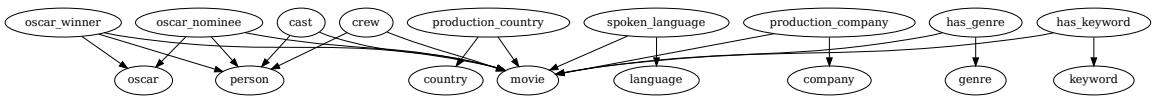

Figure 2: Overview of the *moviedata* database. Nodes represent tables and edges correspond to foreign keys referencing another table.

The full *OTTA* corpus contains the data of five databases: chinook (online music store), college, driving school, Formula 1, and moviedata (based on IMDB). This study will

restrict itself to the *moviedata* domain. We show the schema of the *moviedata* database in Figure 2. At the time of its first publication, *OTTA* contained 1148 pairs of OT and natural language questions.

The performance of a semantic parser is assessed by comparing the OT that is produced by the parser for a given question to the corresponding gold OT from the corpus, for every sample in the corpus. We can then compute the fraction of correct parses as the parser's accuracy. In general, there are different ways to compare OT. A simple method is to execute both OT on the underlying database and compare the results. This tends to overestimate the performance, as result sets can be equal by chance, especially for OT with a *IsEmpty* root node. A better way is to compare the structure of the trees. When doing this, one has to be careful to allow for certain re-orderings of nodes, if they do not change the semantics of the tree. Otherwise, one would underestimate the accuracy. We adopt such a structural equality metric to compute the accuracies reported in Section 5.2.

4 Operation Trees to Text (OT3)

S_1	::=	$Sum(T, A) \mid Average(T, A)$
S_2	::=	$Done(R) \mid IsEmpty(R) \mid Count(R)$
R	::=	$ExtractValues(T, A)$
T	::=	$Min(T, A) \mid Max(T, A) \mid Distinct(T) \mid Merge(T, T, A, A) \mid F$
F	::=	$Filter(F, A, OP, LIT) \mid GetData(TN)$
TN	::=	table name
A	::=	attribute
OP	::=	$<\mid>\mid\leq\mid\geq\mid=\mid\neq$
LIT	::=	literal

Table 2: Reduced Grammar used by the Operation Tree to Text framework by von Däniken et al. (2021).

In a previous study, we developed an approach to generate a natural language representation for arbitrary OTs (von Däniken et al., 2021), which we will explain before specifying the applied modifications. The *Operation Tree to Text* or *OT3* algorithm works on a simplified grammar shown in Table 2. The most important change is the removal of all set and grouping operations. These operations are difficult to express in natural language and would lead to long unwieldy synthetic utterances. Prior experience from working with *OTTA* indicates that even humans struggle with expressing those operations in a natural way. The other change is that *Filter* operations have to appear directly before a *GetData* operation. This is to ensure that relative clauses generated from *Filter* nodes (see Section 4.2) stay close to their associated noun phrases. Note that this change does not impact the expressivity of OT, as one can freely move *Filter* nodes up and down the tree without impacting the semantics of the OT in the original grammar.

Given an OT defined over some database DB, the OT3 algorithm will traverse the tree recursively and expand every node by applying deterministic expansion rules and combining question fragments from its child nodes. All composition rules and the majority of expansion rules are defined in a domain-agnostic way. Nevertheless, it is necessary to provide domain-specific information for DB. In the following sections, we will give an overview of all production rules and point out which parts have to be declared manually per domain. In Section 4.4, we will summarize and exemplify the necessary domain-specific declarations.

4.1 Domain-agnostic productions

	Root Nodes
SUM	$[[Sum(T, A)]]$ = What is the total $[[A]]$ of all $[[T]]$?
AVERAGE	$[[Average(T, A)]]$ = What is the average $[[A]]$ of all $[[T]]$?
COUNT	$[[Count(R)]]$ = How many $[[R]]$ are there?
ISEMPTY	$[[IsEmpty(R)]]$ = Are there any $[[R]]$?
DONE	$[[Done(R)]]$ = What are the $[[R]]$?
	ExtractValues
EXTRACTVALUES	$[[projection(T, A)]]$ = $[[A]]$ of $[[T]]$
	Aggregations
MIN	$[[min(T, A)]]$ = $[[T]]$ with minimum $[[A]]$
MAX	$[[max(T, A)]]$ = $[[T]]$ with maximum $[[A]]$
	Distinct
DISTINCT	$[[Distinct(T)]]$ = distinct $[[T]]$

Table 3: Overview of domain independent production rules.

In the following sections, we will use $[[N]]$ to mean the expansion of some node N to a natural language utterance. We show all domain independent production rules in Table 3. In general, expansions of attributes $[[A]]$ will result in the canonical name of the attribute in either singular or plural form, depending on the parent node. The attribute names themselves have to be provided externally for each specific database. Given these rules, simple queries can easily be translated to natural language:

- Sum(GetData('movie'), 'movie.revenue') → "What is the total revenue of all movies?"
- Min(GetData('movie'), 'movie.runtime') → "movies with minimum runtime"
- Count(ExtractValues(GetData('oscar'), 'oscar.category')) → "How many categories of oscars are there?"
- Done(ExtractValues(Max(GetData('movie'), 'movie.runtime'), 'movie.title')) → "What are the titles of movies with maximum runtime?"

Throughout all of these examples, nodes of type $GetData(TN)$ were expanded to the plural form of the table's name (TN). Those names are defined externally. In general, not all *GetData* operations can be expanded in that way. This will be expanded upon in Section 4.3.

4.2 Filter Operations

All Filter operations are expressed as relative clauses. The exact wording depends on the *type* of the attribute that is filtered on. We call the simplest attribute type *Generic*. It is used as the default attribute type where none of the more specific types apply. In particular, we assume that literals of generic attributes are not ordered.

Generic attributes will be expanded as:

Filter(T, A, OP, LIT) $\rightarrow$ [[T]] whose [[A]] [[OP]] [[LIT]]

[[A]] as defined earlier, expands to the attribute name and [[LIT]] will expand to the literal itself. Since generic attributes are not ordered, the only legal values for OP are $=$ and $\neq$, which we expand to "is" and "is not" respectively. These are two concrete examples:

- Filter(GetData('person'), 'person.name', =, "Brad Pitt") $\rightarrow$ "people whose name is 'Brad Pitt'"
- Filter(GetData('movie'), 'movie.tagline', $\neq$, "A Street Romance") $\rightarrow$ "movies whose tagline is not 'A Street Romance'"

In some cases, it is more elegant to express the attribute as a verb phrase instead of as a noun phrase. For example, the construction "customers whose city is 'New York'" sounds unnatural and would be better expressed as "customers who are living in New York." For such cases, we define the *VerbPhrase* attribute type. For these attributes, we have to provide the auxiliary verb ('are'), the participle ('living'), and the preposition ('in') to express the phrase.

Another big category of attributes are covered by the *Numeric* type. Their expansion is similar to generic attributes:

Filter(T, A, OP, LIT) $\rightarrow$ [[T]] with a [[A]] [[OP]] [[LIT]]

Unlike their generic counterparts, numeric attributes are ordered and as such allow for all values of OP, which we expand as follows:

- $>$ $\rightarrow$ "of more than"
- $\geq$ $\rightarrow$ "of at least"
- $<$ $\rightarrow$ "of less than"

- $\leq \rightarrow$ "of at most"
- $= \rightarrow$ "of"
- $\neq \rightarrow$ "other than"

When declaring a numeric attribute, we also allow the unit to be provided, such as "dollar" for amounts of money (*revenue* or *budget* for movies). This can make the resulting expression more comprehensible to the end user. These are some examples:

- Filter(GetData('movie'), 'movie.budget', <, "1000000") $\rightarrow$ "movies with a budget of less than 1000000 dollars"
- Filter(GetData('movie'), 'movie.runtime', $\geq$, "60") $\rightarrow$ "movies with a runtime of at least 60 minutes"
- Filter(GetData('movie'), 'movie.popularity', $\neq$, "3") $\rightarrow$ "movies with a popularity other than 3"

We also define an attribute type for dates, for which we combine the approaches from both *VerbPhrase* and *Numeric* attributes. An example is the *release_date* attribute for movies:

- Filter(GetData('movie'), 'movie.release_date', <, "1991-24-12") $\rightarrow$ "movies who were released before '1991-24-12'"

The final types are for primary and foreign keys, which we assume only appear in *Merge* nodes but would be treated like *Generic* attributes otherwise.

4.3 Join Operations

To properly handle *Merge* nodes, we first have to introduce the distinction between *entity* and *relation* tables. For *entity* tables, we assume that they do not contain any foreign key attributes, while for *relation* tables we assume the opposite, namely that they exclusively contain foreign key attributes. These are simplifying assumptions to illustrate our approach and we will show how they can be relaxed at the end of this section.

All *Merge* operations will be between a primary key attribute of an *entity* table and a foreign key of a *relation* table.

Consider the example OT in Figure 1. It contains two *entity* tables, *movie* and *country*, that are combined through a *relation* table, *production_country*. To express such a relation, we again rely on external information, in the form of templates. In this concrete instance, the template is "$movie that were produced in $country", where *$movie* and *$country* serve as placeholders for *movie* and *country* entities.

Note that in the OT format, the order in which entity branches attach to relations does not generally matter. This is not the case in natural language, where the order of the entities is important. An alternative template for the *production_country* relation is: "$country in which $movie were produced", which has the order of the entities reversed. This is important when considering the complete OT, as only the first will lead to a coherent final utterance, since the root node asks for the revenue of movies. This means that we have to define one template for each component entity for every *relation* table.

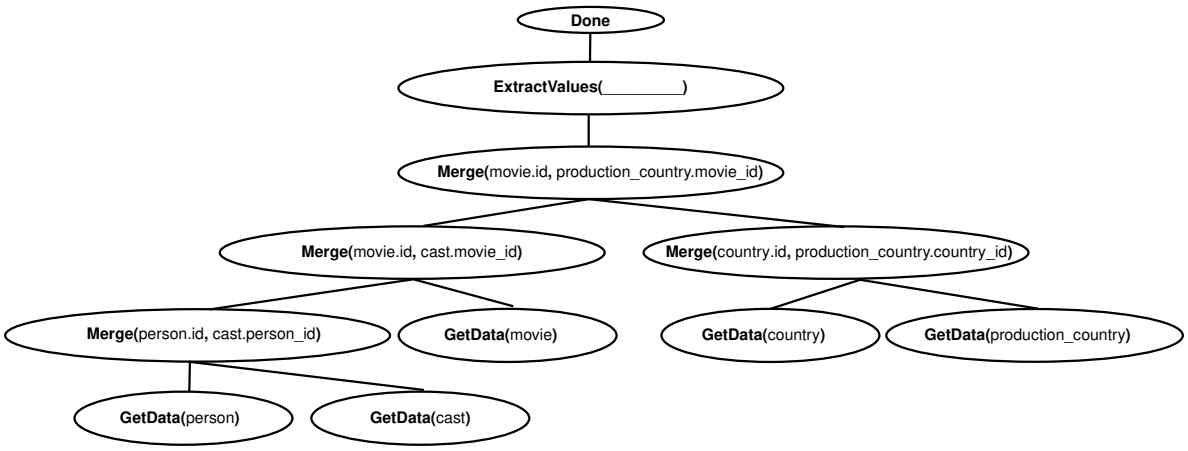

Figure 3: Example OT with two *relation* tables joining three *entity* tables.

Entity order is also relevant when multiple relations, and therefore multiple templates, are involved. Consider the extended example in Figure 3 where we added another relation, *cast*, between the *movie* and the additional *person* entities. Depending on the attribute argument of the *ExtractValues* node, there are four possible ways to combine the templates:

- "country.name" → "What are the names of countries in which movies starring people were produced?"
- "person.name" → "What are the names of people starring in movies produced in countries?"
- "movie.title" → "What are the titles of movies starring people produced in countries?" *or* "What are the titles of movies produced in countries starring people?"

There are two valid ways of combining the templates "$movie produced in $country" and "$movie starring $person": either by inserting the whole second template into the *$movie* argument of the first, or vice-versa. The order in which *Merge* nodes are processed decides which one of the two options is used.

In summary, expanding $[[GetData(table)]]$ will either result in the table's canonical name for *entity* tables or in a predefined template for *relation* tables. The selection of the

correct template depends on the *relation*, as well as the structure of the OT. Expanding a *Merge* node will first expand both child nodes and then insert one sub-phrase into a placeholder of the other. The whole OT3 process for the OT in Figure 1 is illustrated in Figure 4.

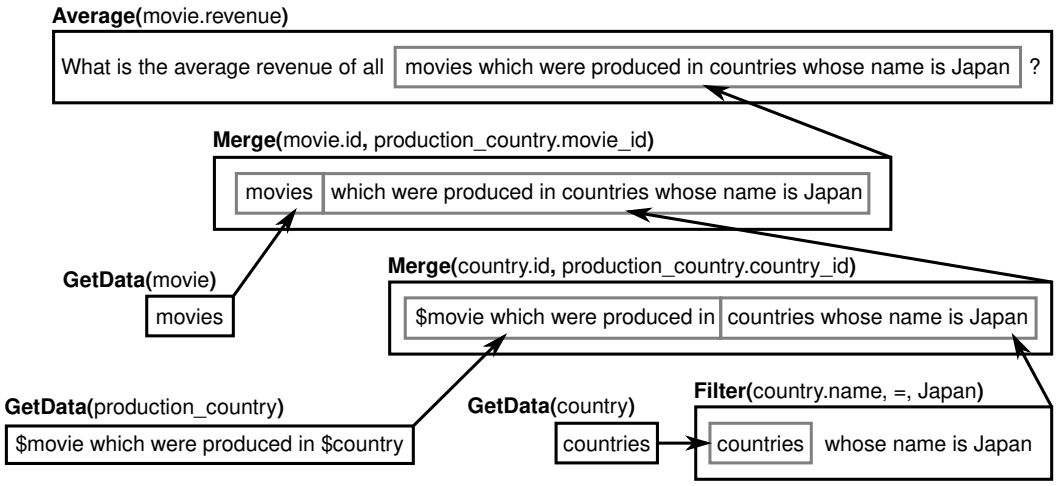

Figure 4: Example of how a natural language question is composed from an OT and domain specific metadata. We write "$T" to denote placeholders where phrases from sister subtrees will be inserted. (Figure taken from von Däniken et al. (2021))

4.3.1 Relaxing the distinction between Entities and Relations

In practice, most databases will rarely contain only pure relation tables, as described above. In some cases, tables that are best modelled as entities will have foreign key attributes. OT3 will treat such tables either as *entities* or *relations*, depending on whether there is a *Merge* node involving a foreign key of that table.

In the simple case where there is no such *Merge*, the table is treated as an entity. Otherwise, we can consider it a relation that has one of its entity components already pre-filled. Therefore, we have to define templates for such tables.

There are also cases where a *relation* table has additional attributes. Our concrete implementation of OT3 handles this by extending the templating system.

4.4 Declaration of Domain-specific metadata

In Listing 1, we show an example of how an *entity* table is declared. It contains the name for the table in the underlying database and the name for the entity it represents. These will not always be the same, as is the case here. Table names are usually expressed without spaces, as such programmers will use snakecase or camelcase as table names, which is unsuitable for non-technical end-users. The same is also the case for naming attributes. As a concrete example, consider the *vote count* attribute of the *movie* entity.

```
Entity(
    table_name='movie',
    entity_name='movie',
    attributes={
        "id": PrimaryKey(),
        "title": Name(is_default=True),
        "revenue": Numeric(name='revenue', unit="dollar"),
        "overview": Text(name="overview"),
        "runtime": Numeric(name="runtime", unit="minute"),
        "tagline": Text(name="tag line"),
        "vote_count": Numeric(name="vote count"),
        ...
    }
)
```

Listing 1: Example declaration of the *movie* entity table in the *moviedata* domain. Some attributes have been omitted for brevity.

```
Relation(
    table_name="production_company",
    attributes={
        "movie_id": ForeignKey(name="movie_id", target_table="movie"),
        "company_id": ForeignKey(name="company_id", target_table="company"),
    },
    templates={
        "movie": "$movie which were produced by $company",
        "company": "$company which produced $movie",
    }
)
```

Listing 2: Example declaration of the *production_company* relation table in the *moviedata* domain. Note that we declare a template for every component of the relation.

To declare attributes, we provide a mapping from attribute names to concrete instances of the attribute types described previously. Note that the *Text* attribute type is equivalent to the *Generic* type. The title attribute is tagged as the default attribute, which will be relevant in Section 5.

In Listing 2, we show an example declaration of a *relation* table. For *relation* tables, we additionally have to provide templates for how they are to be expressed in natural language. The templates are declared as a mapping from the name of the head of the relation to a template string. The component entities are represented by placeholders in the form *$table_name*.

The amount of manual effort to provide these declarations scales linearly with the size of the database. We originally developed OT3 for the *moviedata* domain and providing new declarations for the *chinook* domain was a two hour process.

4.5 Extension of OT3

Our original implementation of OT3 returned a raw *string* for a given OT. For our new use case, we need more fine-grained information about which node generated what part of the utterance.

We, therefore, updated OT3 to produce a sequence of *tokens* instead. A *token* consists of one or more words of the generated utterance, a reference to the node which generated those words, as well as more specific information for words which were generated due to node attributes. We define the following token types:

- Token(content, node): content was generated based on the expansion of some node $[[N]]$.
- AttributeToken(content, node, attribute_name): content was generated based on the attribute argument of the respective node. It corresponds to an expansion of the form $[[A]]$.
- LiteralToken(content, node, attribute_name): content was generated based on the literal of a Filter node. It corresponds to an expansion of the form $[[LIT]]$.
- ComparatorToken(content, node, attribute_name): content was generated based on the comparison operator of a Filter node. It corresponds to an expansion of the form $[[OP]]$.
- TableToken(content, node, table_name): content was generated based on a GetData operation on an entity table. It corresponds to an expansion of the form $[[TN]]$.

The original natural language utterance can be recovered by concatenating the contents of the token sequence. We show an example token sequence in Table 4.

Token Text	What is the average	revenue	of all	movies	which were produced in	countries	whose	name	is	Japan	?
Token Type	Token	AttributeToken	Token	TableToken	Token	TableToken	Token	AttributeToken	ComparatorToken	LiteralToken	Token
Node	Average('movie.revenue')			GetData('movie')	GetData('production_country')	GetData('country')	Filter('country.name', =, 'Japan')				Average('movie.revenue')

Table 4: Sequence of tokens produced by updated OT3 for OT in Figure 1.

5 Operation Tree Correction Framework

One of the use cases of OT3 presented in von Däniken et al. (2021) was to evaluate the output OT a NLIDB system produces for a given human question *without access to the underlying true OT*. The main idea was to present the user with the synthetic utterance generated by OT3 and ask them whether it is semantically equivalent to their question. We have shown that such a binary semantic textual similarity (Agirre et al., 2013) judgement correlates strongly with the correctness of the parse. Indeed, untrained

crowdworkers were able to detect wrong parses with up to 77% accuracy. One advantage of that approach is that the user does not need any prior training to provide such feedback.

In this work we expand upon that idea and extend OT3 to create a framework that lets an end-user directly modify and correct a wrong OT in natural language, by manipulating the text generated by OT3, as our main contribution.

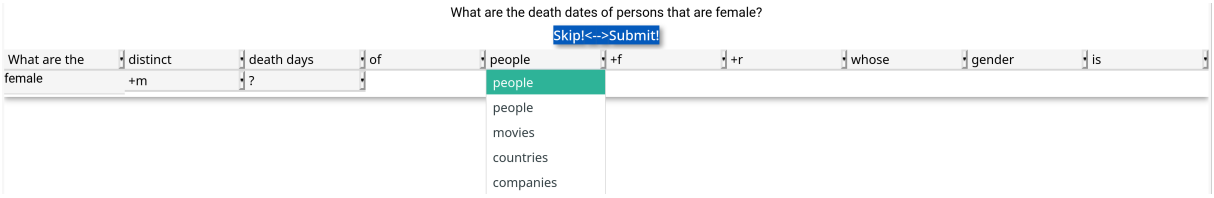

Figure 5: Screen capture of the OT correction framework in use. Refer to the text for a detailed explanation.

In Figure 5, we show a simple example of the correction framework in use. At the top is the question as provided by a human. There is a row of buttons in blue underneath the given question. The *skip* button lets the user skip the sample if they get stuck or do not know what to do with the sample. The buttons labelled `<-` and `->` are to *undo* a erroneous modification or *redo* a previously undone action. The *submit* button is to indicate that the user is done with their correction. Lastly, there is the utterance generated by OT3 from the OT that was produced by a semantic parser for the human question. The utterance is broken up by tokens (see Section 4.5) and every box is a drop-down menu (or text field in case of literals) for the various possible modifications of the utterance and the underlying OT. Concretely shown is the opened menu that would change the *GetData* node for the *person* entity to another entity (which is unnecessary in this case as the utterance already seems to match the human question). The menus labelled *+f*, *+r*, and *+m*, are to add a *Filter* node, insert a *relation*, or add a *Min* or *Max* node, respectively.

At the most basic level, we can modify an OT by adding, deleting, or replacing nodes. It is crucial to maintain the constraints imposed by the OT grammar as well as relationships between tables and attributes. For example, if we inserted a *Merge* without also adding its second sub-tree, the tree would be incomplete. Similarly, if we replace a *GetData* node for some table but a node higher up refers to an attribute of that table, the tree would be in an inconsistent state. We, therefore, will not allow all kinds of modifications for all nodes.

The correction framework will enumerate all legal modifications to an OT, render them to natural language, and assign them to the corresponding token of the OT3 output. For example, the underlying OT in Figure 5 contains a *GetData('person')* node. The opened drop-down menu corresponds to all allowed replacements of that node.

5.1 How to modify OT

Filter. While filters are among the most complex nodes in their composition, they are relatively easy to modify. Earlier, we described (Section 4.2) how OT3 expresses filters as relative clauses. The whole clause will usually consist of a *Token* introducing the relative clause ("with a", "whose"), an *AttributeToken* expressing the attribute to be filtered, a *ComparatorToken* expressing the comparison made, and a *LiteralToken* representing the literal that is being employed for comparison (see Table 4 for an example).

We attach the option to delete a filter node to its leading token, as all possible replacements will be associated with more specialized tokens. Deleting a filter node is relatively straightforward and will leave the resulting OT in a legal state.

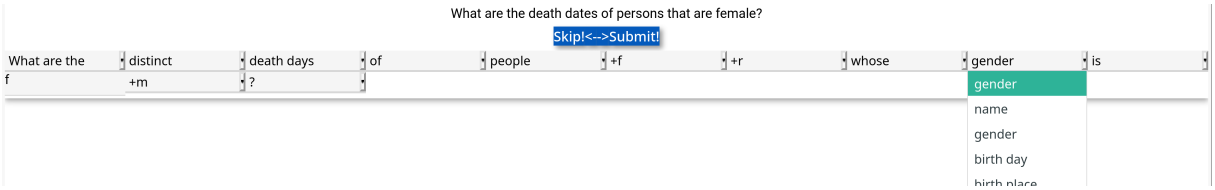

Figure 6: Drop-down menu showing the options to change the attribute of a filter node.

We let the *AttributeToken* represent replacements of the node with a new filter on a different attribute of the same entity. We show these options in natural language using the name of the new attribute. An example can be seen in Figure 6. For the new filter, we copy the literal and comparison operation from the original filter node. In some cases, when changing from an ordered attribute type to a generic one, we have to replace the comparison operation by $=$ for the new filter to be legal.

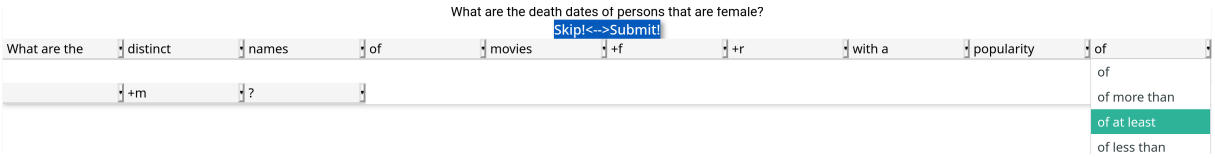

Figure 7: Drop-down menu showing the options to change the comparison operation of a filter node.

Changing the comparison operation is handled similarly to changing the attribute. We attach the options for legal operations to the *ComparatorToken*. The wording for every possible comparison corresponds to the one described in Section 4.2 and an example can be seen in Figure 7. We copy the attribute and literal from the filter node to the new one.

Finally, the text representation of a literal is the literal itself. While we could use a drop-down or similar to let the user select a value from the database, this is impractical for even moderately sized databases. Therefore, it is preferable to pre-fill a text input field with the literal from the OT and let the user modify it. Once the user provides a new literal, we update the filter node accordingly.

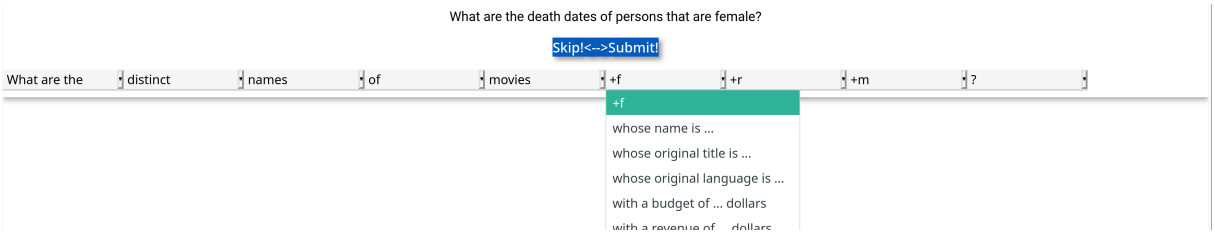

Figure 8: Drop-down menu showing the options to add a new filter node.

We decided to add the placeholder *+f* after entity tokens to represent the option to add a new filter to that entity. We show one option for every attribute of the entity and represent it by the text generated by OT3 for the new filter node. Figure 8 shows an example. We select = as the default comparison operator, and an arbitrary literal from the database for the new node. The new node is inserted as a direct parent of the entity's *GetData* node.

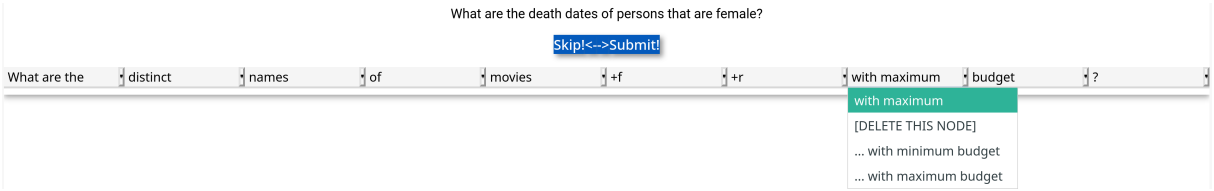

Figure 9: Drop-down menu showing the options to delete or change Min/Max node.

Min and Max. The overall handling of *Min* and *Max* nodes is similar to *Filter* nodes. OT3 will produce a generic token for the node itself and an *AttributeToken* for its attribute. As shown in Figure 9, we attach the options to delete the node, or change it from minimum to maximum, or vice-versa, to the generic token. Since these nodes always have exactly one parent and one child, deleting and replacing them is straightforward.

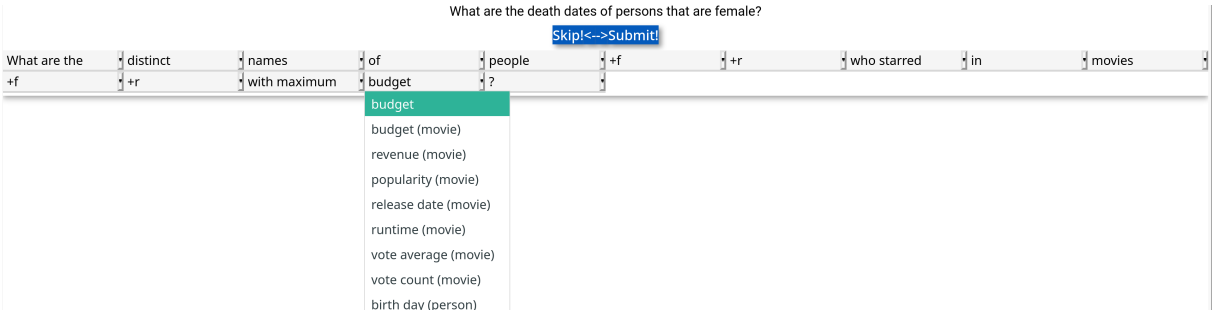

Figure 10: Drop-down menu showing the options to change attribute for a Min or Max node.

The attribute of a minimum or maximum operation can be changed to an ordered attribute of any entity table in the sub-tree below it. Figure 10 shows an example. The drop-down associated with the *AttributeToken* shows both attributes for movies and people and all the attributes follow an ordering.

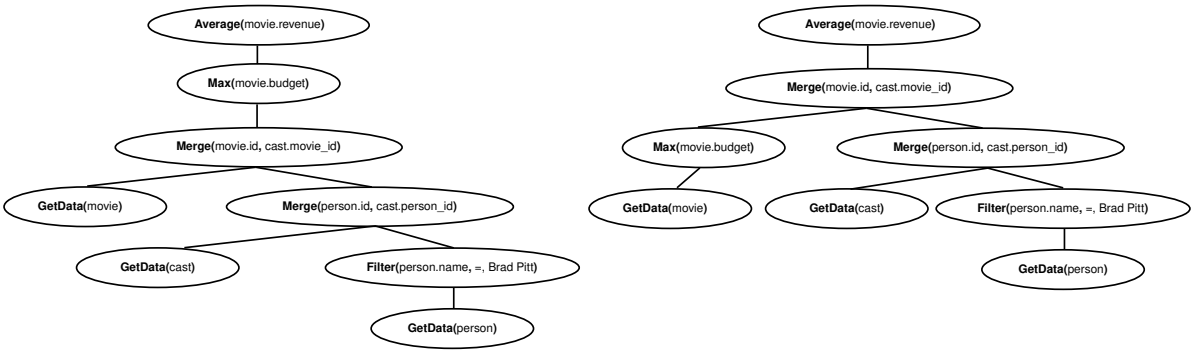

Figure 11: Example of two OT that differ in the position of a Max operation. See text for detailed explanation.

Unlike filters, Min and Max cannot be moved up or down the OT without changing the semantics of the tree. At the same time, this shift in meaning is not easy to capture in natural language. Consider the difference between the two trees in Figure 11. The tree on the left will average the revenue of all movies with maximum budget among the movies that star Brad Pitt. The tree on the right, in contrast, will average the revenue of all movies that have the maximum budget amongst all movies and also star Brad Pitt. These two will not always be the same but the difference is subtle. The *OTTA* corpus handles this ambiguity by only including OT where all Min and Max nodes are ancestors of any Merge nodes. This means that trees like the one on the right in Figure 11 are not allowed.

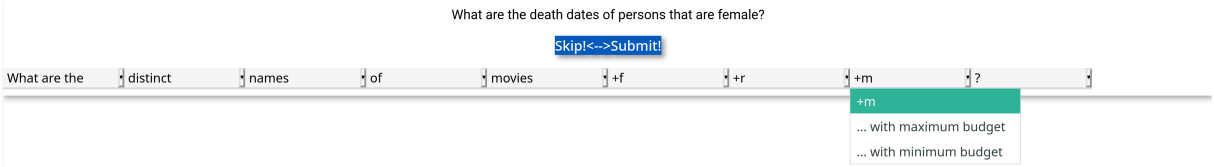

Figure 12: Drop-down menu showing the options to add a Min or Max node.

Knowing this, we place the special token, *+m* for adding a new Min or Max node at the end of the question, since they are expressed as relative clauses referring to the whole construction, following the leading entity in the question. This also means that when the user chooses to add a Min or Max node, we insert it as the parent of the Merge node that is closest to the root. Figure 12 shows how adding a Min or Max node is presented to the user.

Entities and Relations. We illustrated how OT3 translates entities and relations to natural language in Section 4.3. In particular, we point out, that entities will appear in the final utterance in a specific order chained together by relation templates. The chain starts with the entity whose attribute appears at the root of the whole tree (the attribute argument of either the *Sum*, *Average*, or *ExtractValues* nodes). We will call that attribute

the *main query attribute* and the corresponding table the *main query table*.

Due to this interaction between the various *GetData* nodes, we only allow the replacement of the main query entity. Other entities can only be changed or removed by modifying the associated relations.

We showed an example of the user interface for changing the main query table in Figure 5 at the start of this section. In the simplest case, where the new entity is already in the tree, we only have to update the main query attribute to promote the selected entity to main query entity. For that purpose, we will select the *default attribute* of the new main query table (see Section 4.4). If, on the other hand, the new main query table is not part of the tree, we will discard all other *GetData* and *Merge* nodes and add a *GetData* node for the new entity. This is extreme but unavoidable, as we would otherwise have to guess and add the relation that connects the new table to the already existing entities.

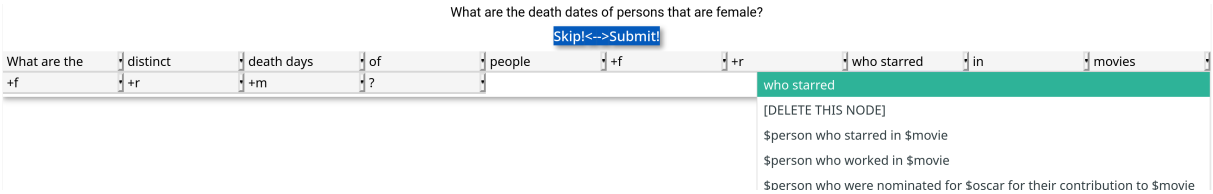

Figure 13: Drop-down menu showing the options to change a relation.

Figure 13 shows the options to delete or change a relation. When deleting a relation, we not only have to delete the *GetData* node for the associated relation table, but any *Merge* nodes that involve foreign keys of that relation table. When deleting a *Merge* node, our implementation will select one of its two sub-trees to replace it and discard the other sub-tree. As a general rule, we will always preserve the sub-tree containing the main query table and prioritise entities appearing earlier in the utterance to later ones. This is a naive solution that will in some cases discard too many nodes, but in practice this has not shown to be problematic yet, as it relatively easy to add the nodes back (see Section 5.2).

We let the user change an existing relation to any relation involving the entity that appears first in the current utterance, assuming that the relation is not already part of the OT. In the example in Figure 13, this would be the person entity. The current implementation will first delete the current relation, as described above, and then insert the new relation. This is unnecessarily destructive and a more sophisticated approach would be to replace the relation table node and adapt the *Merge* nodes up the tree with the new foreign key attributes. We intend to fix this in a later version.

Finally, to insert a new relation we provide another placeholder token, *+r*, directly after each entity *TableToken*. The options are to add any relation involving that entity which is not already part of the OT. Figure 8 shows an example. To insert a new relation,

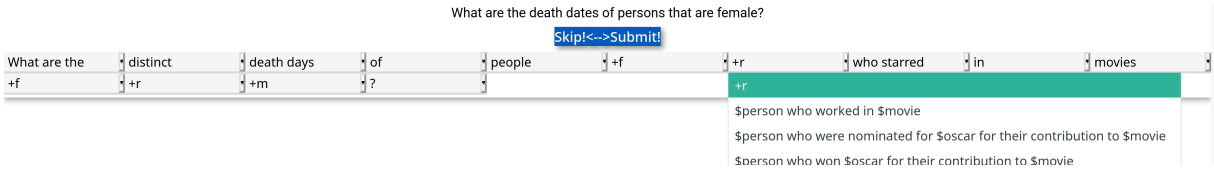

Figure 14: Drop-down menu showing the options to add a new relation.

we add a *Merge* node as parent to the entity's *GetData* node. The second sub-tree of the new *Merge* node will contain the other *GetData* and *Merge* nodes to complete the relation.

We note that the end user never directly manipulates *Merge* nodes but can only influence them through changing relations.

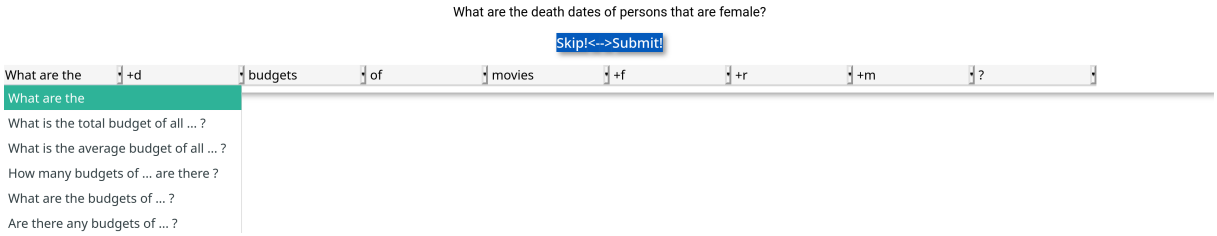

Figure 15: Drop-down menu showing the options to change the question type.

Question Types. According to the grammar in Table 2, there are five different root nodes. These correspond to five different types of questions that can be asked, which were shown in Table 3. An *ExtractValue* node must always appear as a direct child of either a *Done*, *IsEmpty*, or *Count* node. Therefore, we will treat them as a single unit. We attach the option to change the question type to any token associated with one of these nodes. An example can be seen in Figure 15. There are no options to delete or add a new question type, as such operations would not lead to a well-formed OT. When changing the question type, we replace the nodes corresponding to the old question type with ones corresponding to the selected one. We copy the main query attribute from the old nodes to the new ones. We only show the option to change to a *Sum* or *Average* question, when the current main query attribute is summable.

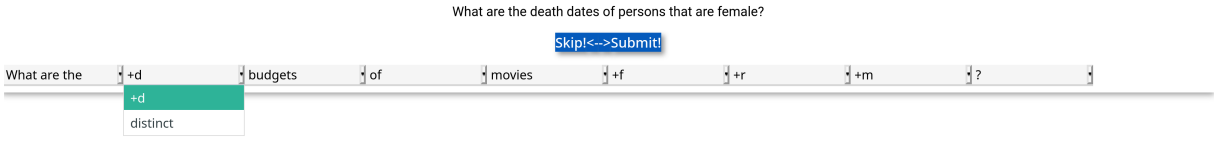

Figure 16: Drop-down menu showing the option to add a *Distinct* node.

Distinct. *Distinct* nodes are conceptually simple, as the only operations that we define for them are to either add a new *Distinct* node or delete one. OT3 will only produce one

token containing the word "distinct" for this node type and we add the option to delete the node to that token.

Our grammar allows *Distinct* nodes to appear anywhere in the tree between the question nodes and filter or *GetData* nodes. Similar to *Min* and *Max* nodes, the actual samples in the *OTTA* corpus follow additional restrictions. In practice, a *Distinct* node will only appear as a direct child of *any* root node. This includes the nodes requiring an *ExtractValues* node. The *Distinct* node will be inserted between them. This is unfortunately in violation of the grammar, as described. Luckily, this is only a minor deviation and does not impact the overall approach. As a result, we place the special token for inserting a *Distinct* node, *+d*, directly in front of the first token expressing the direct child of the root node. This option is only available if there is not another *Distinct* node already present. Figure 16 shows an example.

5.2 Evaluating the OT Correction Framework

To show the efficacy of our correction framework, we have used it to correct the outputs of a real semantic parser. We use a *GrammarNet* parser (Yin and Neubig, 2017) with the same settings as used by Deriu et al. (2020). We gave a brief overview of *GrammarNet* in Section 2. As dataset, we used the OT (and questions) from the *moviedata* domain of the *OTTA* corpus. We only used samples whose OT do not contain any set or grouping operations. The resulting 1116 samples were split into a training set of 894 and a test set of 222 samples, corresponding to 80% and 20% respectively. The resulting parser, trained on the training set, achieves an accuracy of 54.9% on the test set.

We applied our correction framework to 107 randomly selected samples of the test set. An annotator is presented with the original human question as well as the modifiable textual representation of the OT produced by the parser as described in Section 5. They can then apply as many modifications as they deem necessary (including none at all) to make the synthetic utterance match the input question. Since we did not have time to instruct external people in how to use our tool, we annotated the data ourselves.

	Correction Applied		No Correction Applied	
	Result Correct	Result Incorrect	Result Correct	Result Incorrect
Correction Necessary	32	10	-	5
No Correction Necessary	5	3	52	-

Table 5: Overview of corrections performed during our experiment.

In Table 5, we give an overview of the number of trees which were corrected, if the corrections were necessary, and if the result was equal to the reference gold OT. We can see that the parser performed slightly better on our subset of 107 samples compared to the full test set. In 60 cases no correction would have been necessary, meaning that the parser has an accuracy of 56.1%. After our corrections, 89 OT are correct, improving

the accuracy to 83.2%. In 89.4% of cases, where it was necessary, a correction has been applied. This is consistent with previous results showing high recall in humans finding parse errors when presented with the output of OT3 (von Däniken et al., 2021). Similarly, out of all OT needing correction, 84% were corrected, although not always successfully.

Error Analysis. We will take a closer look at the total 18 cases that were still incorrect after a manual correction attempt, as well as the 5 cases where unnecessary corrections were applied that did not change the correctness of the tree. The latter might strike one as odd, since modifying an OT, that does not warrant it, should intuitively lead to an error. In all of those cases, a single operation to replace the comparison operator of a *Filter* node for a *Date* attribute was performed, updating the comparison operator to the same value (for example replacing $\leq$ by another $\leq$). This stems from an ambiguity in texts produced by OT3, which does not differentiate between strict and non-strict equality in dates. It will produce "before 1984" for both "$<$ 1984" and "$\leq$ 1984". In the drop-down menus of the correction framework those have been disambiguated to "before (not including)" and "before (including)." Since we annotated the data ourselves and were thus aware of this limitation, we made sure to always disambiguate the comparison operation for dates.

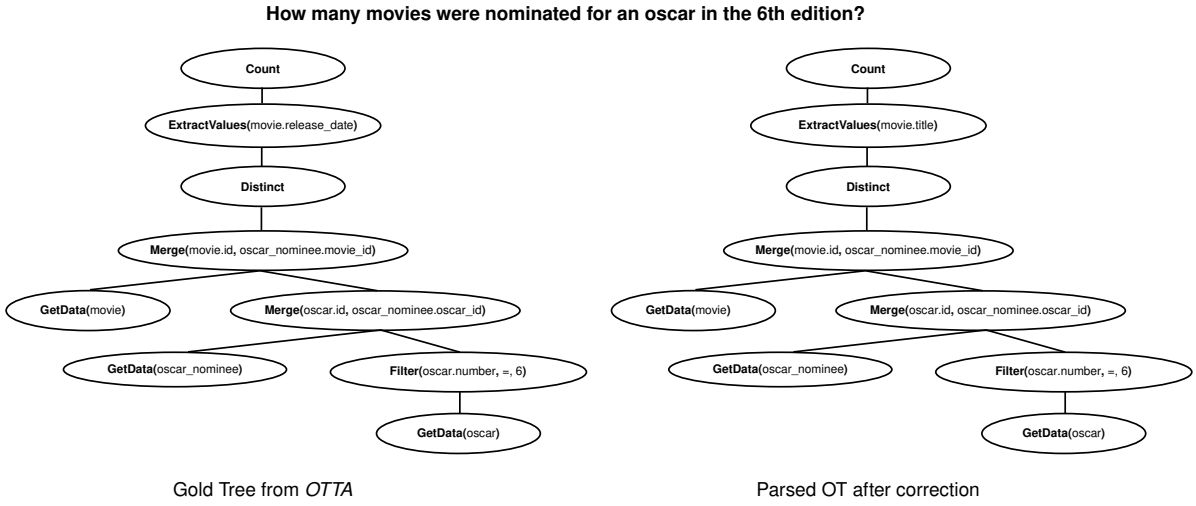

Figure 17: OT for the human input question: "How many movies were nominated for an oscar in the 6th edition?" On the left, the gold OT from the *OTTA* corpus. On the right, the parsed OT after it was corrected.

The most prominent source of errors seems to be ambiguous question types, accounting for 7 out of the 18 errors. There are two sources of ambiguities. The first stems from a human preference for succinctness. Some human annotators of the *OTTA* corpus seemed to drop the main query attribute from the question in some cases for OT with *Done*, *IsEmpty*, or *Count* root nodes, effectively ignoring the *ExtractValues* node. This makes it impossible to correctly recover the original tree. We show an example in Figure 17. As discussed in Section 3.1, the *OTTA* corpus was created by letting human annotators write

questions matching a tree sample from the OT grammar. The original tree can be seen on the right in Figure 17. In the associated question: "How many movies were nominated for an oscar in the 6th edition?", the annotator did not include the *release date* main query attribute. Therefore, it was impossible to recover that attribute for both the parser and the corrector. In fact, the only correction operation that was applied in this instance was to change the attribute of the *Filter* node. In this specific instance, one can argue that the parsed OT matches the input question more closely.

The other source of ambiguity in question types comes up in some aggregation questions, when the root node is either *Sum* or *Average*. In a question such as: "How much revenue did movies generate in which 'Fritz Rasp' was part of the cast?", it is not entirely clear whether we have to sum over all revenues of matching movies, or return them individually.

The second largest source of errors stems from *Distinct* nodes. Human questions in *OTTA* often do not contain explicit words such as *unique*, *different*, or *distinct*, to express the presence of a *Distinct* node. When correcting a tree, it is, therefore, hard to know whether adding or removing such a node is appropriate.

Since we only worked with a very limited set of samples, it is hard to draw any meaningful conclusions from the other errors that were observed. We will mention the few remaining errors that we deem interesting and that might be meaningful. In *moviedata* specifically, there are relations that have some inherent ambiguity. When humans express the *cast* or *crew* relation it is not always entirely apparent which one is meant. A similar confusion exists between the *original language* attribute for movies and the *spoken_language* relation between the tables *movie* and *language*. Finally, there is always the element of human error, both in corrections and in the original annotation process.

Number of Corrections. Another aspect to evaluate is the number of interactions the user has to perform to correct an OT. This depends on two factors, first of which is the quality of the parser. The better the parser, the closer it will get to the true OT and the less there will be to correct. The second aspect is the efficiency of the framework. Ideally, our framework would be expressive enough to enable modifications with as few interactions as possible. We count any time a user selects an option from a drop-down menu as a discrete correction. In Figure 18, we show the histogram of the number of interactions taken during our annotations. Out of the 50 OT that have been modified, more than half, namely 28, required only a single interaction. The maximum number of interactions for any single OT was 6.

We give an overview of the types of modifications applied in Table 6. The most common change was to correct a comparison operator, accounting for around 27.4% of modifications. The second most common modification was to change an attribute. This is followed by adding filters and changing a literal. These two occur together because

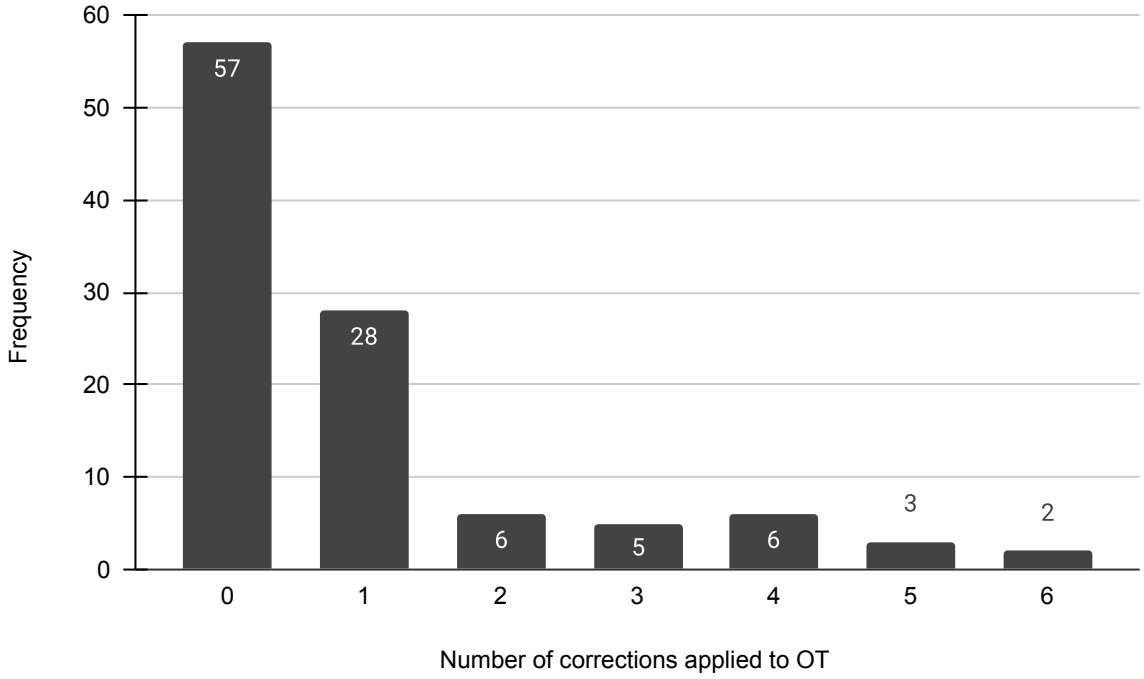

Figure 18: Histogram of the number of corrections applied to OT.

Correction Type	Occurrences
Change Comparison Operator	29
Change Attribute	21
Add Filter	17
Change Literal	17
Delete Node	9
Add Relation	4
Add Distinct	3
Change Main Query Table	2
Change Question Type	2
Add Min/Max	2

Table 6: Types of corrections applied during our experiments.

the correction framework fills in an arbitrary literal when creating a new *Filter* node, which subsequently has to be adjusted too (see Section 5). Similarly, it chooses = as the default comparison operator for new *Filter* nodes, which might also have to be updated and therefore account for some of those operations as well. Therefore, in the worst case, a missing filter can take up to 3 interactions to correct. The distribution of corrections also indicates that, overall, the errors made by the parser are relatively minor. In most instances, *GrammarNet* seems to get the right structure of the question and only struggles with missing *Filter* nodes and choosing wrong attributes and comparison operators.

6 Discussion

At first glance the accuracy of 83.2% that was achieved after the correction step seems underwhelming. Our analysis has shown that an important proportion of the remaining errors are due to ambiguities in the human questions in the *OTTA* corpus that the annotators resolved in a wrong way. In an ideal setting, the person posing the initial question would also be the person correcting the resulting parse. This would eliminate the need for the corrector to guess the exact intention of the question. As such, there is a mismatch in our experiment setting and one could argue that eliminating those potential misinterpretations should lead to an even higher accuracy. On the other hand, since we provided the corrections ourselves, there is also the potential to overestimate the performance. Unlike potential lay users, we are intricately familiar with both *OT3* and the correction framework and untrained users might introduce errors that we would not. Finally, it is difficult to draw strong conclusions from the very limited number of samples that we actually corrected. Overall, to more thoroughly assess the potential of the framework, more extensive experiments are needed.

A promising result is that for most parses, only very few interactions are needed to correct them. This indicates that even though the parser has relatively low accuracy, its errors are minor and that our framework is reasonably efficient at letting the user correct those errors. One inefficiency we discovered is that we need up to three interactions to add a missing filter operation. Of course, one could naively enumerate all possible combinations of attributes, comparison operators and literals as options when adding a filter, but this would make the interaction overly complex. Overall, we are satisfied with the trade-off between number of interactions and their complexities.

We also discovered some potential improvements of the templates of *OT3*. Most importantly, the need to better disambiguate strict and non-strict inequalities for date attributes has been identified.

We want to point out that the overall framework of enumerating and applying legal modifications to OT has potential applications beyond just correcting parses. We see two relatively simple ways to build a semantic parser on top of the base we provided. The first would be to build something akin to *TR Discover* (Song et al., 2015), which we described in Section 2. Concretely, instead of letting the user type an arbitrary natural language question and parse it with another parser, we could just present them with our correction interface initialized with a minimal legal tree (e.g. "What are the names of movies?"). They could then extend and modify that representation the same way they would for corrections until they arrive at a representation of their question. Of course such a system would be pretty far removed from the original goal of NLIDB research, all but eliminating the natural language component. Still, for some applications, it might be an appropriate and pragmatic choice.

The other option would be to pursue the approach of Wang et al. (2015) that we described in Section 2. This would involve training a paraphrase scoring model based on human utterances from *OTTA* and the back-translations by *OT3*. The parser would then consist of a beam search over OT, scored by the paraphrase model. Starting from a set of minimal legal trees, we would apply all possible legal modifications in every iteration to all trees in the current set to create new candidates. For every candidate, the corresponding synthetic utterance is generated by *OT3* and its similarity to the input question scored by the paraphrase model. All but the the top k highest scoring candidates will be rejected and the process repeated until there is no more improvement.

One aspect of our work we have not yet discussed is the user interface itself. We chose the library *REMI*[5] to develop the user interface. It is a relatively simple way to create a browser based interface for *Python* applications. Overall, we consider the graphical user interface as presented to be a mere prototype and there are several improvements to be made. Most importantly, we should rethink the use of drop-down menus to select changes. Their blocky appearance makes the generated question hard to read. The use of special tokens for adding new nodes in particular is unfortunate and hinders readability. A nicer way would be to present a single text field containing the synthetic utterance that the user can interact with.

7 Conclusion

In this work, we introduced a framework that lets users correct the output of a semantic parser by modifying a synthetic textual representation of the parse. Based on the Operation Tree representation of Deriu et al. (2020), we provided functionalities to enumerate and apply legal changes to any operation tree. We combine these functionalities with prior work that allows back-translating an Operation Tree to natural language (von Däniken et al., 2021). As a result, we proved that our correction framework allows users to efficiently modify Operation Trees and improve the accuracy of a semantic parser from 56.1% to 83.2%. We consider changes to the user interface as improvements that can be made within future work and point out potential further applications, such as using it as a starting point to build a semantic parser.

References

Affolter, K., K. Stockinger, and A. Bernstein (2019, Oct). A comparative survey of recent natural language interfaces for databases. *The VLDB Journal 28*(5), 793–819.

Agirre, E., D. Cer, M. Diab, A. Gonzalez-Agirre, and W. Guo (2013, June). *SEM

[5] `https://github.com/dddomodossola/remi`

2013 shared task: Semantic textual similarity. In *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 1: Proceedings of the Main Conference and the Shared Task: Semantic Textual Similarity*, Atlanta, Georgia, USA, pp. 32–43. Association for Computational Linguistics.

Blunschi, L., C. Jossen, D. Kossmann, M. Magdalini, and K. Stockinger (2012). Soda. generating sql for business users. *Proceedings of the VLDB Endowment 5*(10), 932 – 943. 38th International Conference on Very Large Data Bases; Conference Location: Istanbul, Turkey; Conference Date: August 27-31, 2012.

Bogin, B., J. Berant, and M. Gardner (2019, July). Representing schema structure with graph neural networks for text-to-SQL parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Florence, Italy, pp. 4560–4565. Association for Computational Linguistics.

Cho, K., B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio (2014, October). Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar, pp. 1724–1734. Association for Computational Linguistics.

Codd, E. F. (1970, June). A relational model of data for large shared data banks. *Commun. ACM 13*(6), 377–387.

Codd, E. F. (1989). Relational database: A practical foundation for productivity. In *Readings in Artificial Intelligence and Databases*, pp. 60–68. Elsevier.

Dahl, D. A., M. Bates, M. Brown, W. Fisher, K. Hunicke-smith, D. Pallett, E. Rudnicky, and E. Shriberg (1994). Expanding the scope of the atis task: the atis-3 corpus. In *in Proc. ARPA Human Language Technology Workshop '92, Plainsboro, NJ*, pp. 43–48. Morgan Kaufmann.

Deriu, J., K. Mlynchyk, P. Schläpfer, A. Rodrigo, D. von Grünigen, N. Kaiser, K. Stockinger, E. Agirre, and M. Cieliebak (2020, July). A methodology for creating question answering corpora using inverse data annotation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Online, pp. 897–911. Association for Computational Linguistics.

Devlin, J., M.-W. Chang, K. Lee, and K. Toutanova (2019, June). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Minneapolis, Minnesota, pp. 4171–4186. Association for Computational Linguistics.

Dubey, M., D. Banerjee, A. Abdelkawi, and J. Lehmann (2019). Lc-quad 2.0: A large dataset for complex question answering over wikidata and dbpedia. In *Proceedings of the 18th International Semantic Web Conference (ISWC)*. Springer.

Elgohary, A., S. Hosseini, and A. Hassan Awadallah (2020, July). Speak to your parser: Interactive text-to-SQL with natural language feedback. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Online, pp. 2065–2077. Association for Computational Linguistics.

Green Jr, B. F., A. K. Wolf, C. Chomsky, and K. Laughery (1961). Baseball: an automatic question-answerer. In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, pp. 219–224.

Hochreiter, S. and J. Schmidhuber (1997, 12). Long short-term memory. *Neural computation 9*, 1735–80.

Koutrika, G., A. Simitsis, and Y. E. Ioannidis (2010). Explaining structured queries in natural language. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pp. 333–344.

Labutov, I., B. Yang, and T. Mitchell (2018, October-November). Learning to learn semantic parsers from natural language supervision. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium, pp. 1676–1690. Association for Computational Linguistics.

Mitra, M. and B. B. Chaudhuri (2000, May). Information retrieval from documents: A survey. *Information Retrieval 2*(2), 141–163.

Ngonga Ngomo, A.-C., L. Bühmann, C. Unger, J. Lehmann, and D. Gerber (2013). Sorry, i don't speak sparql: Translating sparql queries into natural language. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, New York, NY, USA, pp. 977–988. Association for Computing Machinery.

Saha, D., A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Özcan (2016, August). Athena: An ontology-driven system for natural language querying over relational data stores. In *PVLDB* (PVLDB ed.), Volume 9, pp. 1209–1220. VLDB Endowment.

Sen, J., C. Lei, A. Quamar, F. Özcan, V. Efthymiou, A. Dalmia, G. Stager, A. Mittal, D. Saha, and K. Sankaranarayanan (2020, July). Athena++: Natural language querying for complex nested sql queries. *Proc. VLDB Endow. 13*(12), 2747–2759.

Song, D., F. Schilder, C. Smiley, C. Brew, T. Zielund, H. Bretz, R. Martin, C. Dale, J. Duprey, T. Miller, and J. Harrison (2015). Tr discover: A natural language interface

for querying and analyzing interlinked datasets. In M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, and S. Staab (Eds.), *The Semantic Web - ISWC 2015*, Cham, pp. 21–37. Springer International Publishing.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin (2017). Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), *Advances in Neural Information Processing Systems*, Volume 30, pp. 5998–6008. Curran Associates, Inc.

von Däniken, P., J. Deriu, E. Agirre, and M. Cieliebak (2021). Using textual similarity to evaluate and improve semantic parsing. In *under review*.

Wang, B., R. Shin, X. Liu, O. Polozov, and M. Richardson (2020, July). RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Online, pp. 7567–7578. Association for Computational Linguistics.

Wang, Y., J. Berant, and P. Liang (2015, July). Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Beijing, China, pp. 1332–1342. Association for Computational Linguistics.

Xu, K., L. Wu, Z. Wang, Y. Feng, and V. Sheinin (2018, October-November). SQL-to-text generation with graph-to-sequence model. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium, pp. 931–936. Association for Computational Linguistics.

Yao, Z., Y. Su, H. Sun, and W.-t. Yih (2019, November). Model-based interactive semantic parsing: A unified framework and a text-to-SQL case study. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Hong Kong, China, pp. 5447–5458. Association for Computational Linguistics.

Yao, Z., Y. Tang, W.-t. Yih, H. Sun, and Y. Su (2020, November). An imitation game for learning semantic parsers from user interaction. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Online, pp. 6883–6902. Association for Computational Linguistics.

Yin, P. and G. Neubig (2017, July). A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Com-*

putational Linguistics (Volume 1: Long Papers), Vancouver, Canada, pp. 440–450. Association for Computational Linguistics.

Yu, T., R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev (2018, October-November). Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium, pp. 3911–3921. Association for Computational Linguistics.

Zeng, J., X. V. Lin, S. C. Hoi, R. Socher, C. Xiong, M. Lyu, and I. King (2020, July). Photon: A robust cross-domain text-to-SQL system. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, Online, pp. 204–214. Association for Computational Linguistics.

Zhong, V., C. Xiong, and R. Socher (2017). Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR abs/1709.00103*.