A Methodology for Creating Question Answering Corpora Using Inverse Data Annotation

Jan Deriu[1], Katsiaryna Mlynchyk[1], Philippe Schläpfer[1], Alvaro Rodrigo[2],
Dirk von Grünigen[1], Nicolas Kaiser[1], Kurt Stockinger[1], Eneko Agirre[3], and Mark Cieliebak[1]

[1]Zurich University of Applied Sciences (ZHAW), Winterthur, Switzerland
{deri, mlyn, scrp, vogr, stog, ciel}@zhaw.ch
[2]National Distance Education University (UNED), Madrid, Spain
alvarory@lsi.uned.es
[3]University of the Basque Country (UPV/EHU), Donostia, Spain
e.agirre@ehu.eus

Abstract

In this paper, we introduce a novel methodology to efficiently construct a corpus for question answering over structured data. For this, we introduce an intermediate representation that is based on the logical query plan in a database called Operation Trees (OT). This representation allows us to invert the annotation process without losing flexibility in the types of queries that we generate. Furthermore, it allows for fine-grained alignment of query tokens to OT operations.

In our method, we randomly generate OTs from a context-free grammar. Afterwards, annotators have to write the appropriate natural language question that is represented by the OT. Finally, the annotators assign the tokens to the OT operations. We apply the method to create a new corpus *OTTA* (Operation Trees and Token Assignment), a large semantic parsing corpus for evaluating natural language interfaces to databases. We compare OTTA to Spider and LC-QuaD 2.0 and show that our methodology more than triples the annotation speed while maintaining the complexity of the queries. Finally, we train a state-of-the-art semantic parsing model on our data and show that our corpus is a challenging dataset and that the token alignment can be leveraged to increase the performance significantly.

1 Introduction

Question Answering (QA) over structured data, also called Natural Language Interfaces to Databases (NLI2DB) or Text-to-SQL, is a key task in natural language processing and the semantic web. It is usually approached by mapping a natural language question (NL question) into executable queries in formal representations such as logical forms, SPARQL or SQL.

The state-of-the-art in this problem uses machine learning techniques to learn the mapping. Unfortunately, the construction of labeled corpora to train and evaluate NLI2DB systems is time- and cost-intensive, which is slowing down progress in this area. In particular, it usually requires recruiting SQL or SPARQL experts to write queries for natural language questions. For instance, in Spider (Yu et al., 2018), the authors recruited students to write SQL queries. They worked 500 person-hours to generate 5,600 queries, which corresponds to more than 5 minutes per question.

As a more cost-effective alternative to writing formal queries manually, some authors propose to use templates to generate them automatically. For instance, LC-QUAD 2.0 (Dubey et al., 2019) used 22 templates based on the structure of the target knowledge graph. Constructing templates is also time-consuming, and the expressiveness of the automatically produced queries is limited.

Apart from the high cost of generating queries, the natural language questions in current datasets do not necessarily cover the whole range of data present in the database. In Spider, the coverage is limited by the creativity of the students, and in LC-QUAD 2.0 by the templates.

In this paper, we propose a new procedure to increase the speed of the annotation process. For this, we first introduce an intermediate representation of the structured queries, which we call *Operation Trees* (OTs, see Figure 1). Our OTs follow a context-free grammar and are based on logical query plans that can easily be mapped to SPARQL or SQL, making our system more versatile. In addition, it has been shown that working on abstract tree representations instead of sequences yields better results (Guo et al., 2019). Recent work by (Cheng et al., 2019) shows the successful use of tree-like abstractions as an intermediate representation to parse text into semantic representations, reinforcing our choice of operation trees as the main representation language.

Our annotation process works as follows. First,

we use the context-free grammar to sample random OTs for a given database. We then let annotators in a first round write the corresponding NL questions for the sampled OTs. In a second, optional, round the annotators perform an assignment of tokens from the NL question to the operations in the OT. This additional annotation enriches the information in the dataset, and, as we will show below, allows for performance gains, especially in low data regimes.

Our approach to producing datasets has the following advantages with respect to the methodology used in previous work: 1) It reduces the time needed for an annotation (less than 2 minutes, compared to more than 5 in Spider). 2) It allows us to cover the whole range of data present in the database structure and not to focus on the most prominent examples. 3) Our annotation procedure provides alignments between operations in the formal language and words in the question, which are an additional source of supervision when training.

We applied our approach[1] to five datasets, yielding a large corpus called OTTA[2] which consists of 3,792 complex NL questions plus their corresponding OTs as well as the token assignment for one of our domains. Besides, we have adapted a state-of-the-art system (Yin and Neubig, 2017) to work on to operation trees, and included a mechanism to profit from token alignment annotations when training. The system yields better results with up to 7 point increase when trained on aligned OTs.

2 Related Work

In this section, we first review the related work in the area of Natural Language Interfaces to Databases (NLI2DB). Afterwards, we focus on the data resources that are currently available to evaluate these systems.

Natural Language Interfaces to Databases. There is a vast amount of literature on NLI2DB. A recent survey on methods and technologies is provided by (Affolter et al., 2019). Early systems use a *keyword-based* approach with inverted indexes to query the databases (Simitsis et al., 2008; Blunschi et al., 2012; Bast and Haussmann, 2015). *Pattern-based* approaches are able to handle more

complex NL questions (Damljanovic et al., 2010; Zheng et al., 2017). *Parsing-based* approaches use a natural language parser to analyze and reason about the grammatical structure of a query (Li and Jagadish, 2014; Saha et al., 2016). *Grammar-based* approaches only allow the user to formulate queries according to certain pre-defined rules, thus focus primarily on increasing the precision of answers (Song et al., 2015; Ferré, 2017). More recent systems use a *neural machine translation* approach similar to translating natural languages, say, from French to English (Iyer et al., 2017a; Basik et al., 2018; Cheng et al., 2019; Liu et al., 2019; Guo et al., 2019; Cheng et al., 2019).

Data Resources. We will now review the major data resources that have recently been used for evaluating NLI2DB systems. These resources are mainly created following two approaches: (1) Both NL and structured queries are manually created, and (2) structured queries are automatically generated, and then humans create the corresponding NL questions.

Regarding fully manually created resources, (Yu et al., 2018) provided Spider, a dataset with 5,600 SQL queries, over 200 databases and 10,181 NL questions annotated by 11 students, where some questions were manually paraphrased to increase the variability. (Finegan-Dollak et al., 2018) released Advising, with 4.5k questions about university course advising and SQL queries. (Dahl et al., 1994) created ATIS, a dataset with 5k user questions about flight-booking manually annotated with SQL queries and modified by (Iyer et al., 2017b) to reduce nesting. (Zelle and Mooney, 1996) created GeoQuery, with 877 questions about US geography annotated with Prolog and converted to SQL by (Popescu et al., 2003) and (Giordani and Moschitti, 2012). There are also smaller datasets about restaurants with 378 questions (Tang and Mooney, 2000), the Yelp website with 196 questions and IMDB with 131 questions (Yaghmazadeh et al., 2017).

Resources using an automatic step usually rely on generating structured queries using templates created by experts. (Zhong et al., 2017) created WikiSQL, a collection of 80k pairs of SQL queries and NL questions made using Wikipedia. However, SQL queries are relatively simple because each of the databases consists of only a single table without foreign keys. Hence, the queries do not contain joins. (Dubey et al., 2019) developed

---

[1]The annotation tool can be found here: `https://github.zhaw.ch/semql/annotation_tool`

[2]The corpus can be found here: `https://github.zhaw.ch/semql/semql-data/tree/master/annotated_tree_files/single_files`

LC-QuAD 2.0, with 30,000 complex NL questions and SPARQL queries over DBpedia and Wikidata. They used templates to generate SPARQL queries for seed entities and relations, which are lexicalized automatically using other templates. NL questions of both datasets were created by crowdsourcing workers.

All the resources mentioned above required a large amount of effort. In each case, the annotators need an in-depth knowledge of SQL or a similarly structured language. Our approach simplifies the process of generating question-answering corpora while ensuring a large coverage of the underlying database without forfeiting any complexity in the queries.

On the other hand, (Wang et al., 2015) developed a method similar to ours. They begin with a lexicon linking natural utterances with predicates in the database. Then, they use domain-specific grammar to create several canonical phrases associated with queries. Finally, crowdsourcing workers rewrite the canonical phrases and create natural utterances used for training a semantic parser. Similar to our approach, they combine an automatic method with crowdsourcing workers. However, they have to create the lexicon and the grammar for each database, while our method can be applied to any database without creating new resources.

3 Operation Trees

In our setting, the goal is to generate an Operation Tree (OT) that finds the correct answer for a given question in natural language. An OT is a binary tree that is closely related to a logical query plan in SQL database engines. An OT is composed of a sequence of operations that can be mapped to a database query language such as SQL or SPARQL to retrieve the proper result.

Example. Assume that we have a database about movies that we want to query in natural language. In Figure 1, an example of an OT is depicted for the question "Who starred in 'The Notebook'?". In order to answer this question, the tables *person* and *movie* are selected, then the table *movie* is filtered by movie title *The Notebook*. In the next step, the tables are joined via the bridge-table *cast*. Finally, the *person.name* column is extracted.

We enhance these OTs by associating a reasonable subset of tokens from the NL question to each operation in the tree. For instance, the token "starred" could be associated to the *Join* operation,
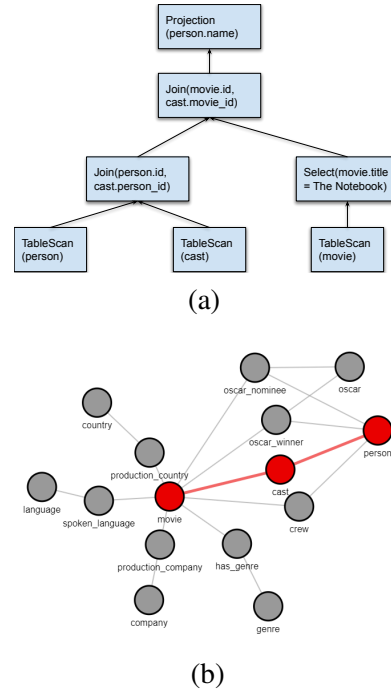

(a)

(b)

Figure 1: (a) Example of an Operation Tree (OT) for the query "Who starred in 'The Notebook'?" (b) The corresponding database schema.

as this operation implies that an actor starred in a movie, whereas the tokens "How many" could be associated to the *Count* operation. This mapping between tokens and operations will help later on to train machine learning algorithms to generate OTs automatically from natural language questions with better quality.

Definition. More formally, the OTs follow a predefined context-free grammar. In the current state, the set of operations includes major operations from the relational algebra with specific extensions. The full grammar is shown in Figure 2.

$$
\begin{array}{ll}
\textbf{S} & ::= \; done(\text{R}) \mid isEmpty(\text{R}) \mid sum\,(\text{T,A}) \mid average\,(\text{T,A}) \mid count(\text{R}) \\
\textbf{R} & ::= \; projection(\text{T, A}) \\
\textbf{T} & ::= \; tableScan(\text{TN}) \mid selection(\text{T, A, OP ,V}) \mid min(\text{T, A}) \mid max(\text{T, A}) \mid \\
& \quad\; distinct(\text{T}) \mid join(\text{T, T, A, A}) \mid union\,(\text{T,T,A, A}) \mid \\
& \quad\; intersection\,(\text{T, T, A, A}) \mid difference(\text{T, T, A, A}) \mid averageBy\,(\text{T ,A}) \mid \\
& \quad\; sumBy\,(\text{T ,A}) \mid countBy\,(\text{T ,A}) \\
\textbf{TN} & ::= \; table\ name \\
\textbf{A} & ::= \; attributes \\
\textbf{OP} & ::= \; < \mid > \mid <= \mid >= \mid == \mid != \\
\textbf{V} & ::= \; values
\end{array}
$$

Figure 2: The set of production rules for the context-free grammar of the operation trees, where *table name* denotes the set of all entity types in the database, *attributes* denotes the set of all attributes of entity types, and *values* denotes the set of all entries in the database. The non-terminal symbols **S**, **T**, and **R** denote the start-symbol, intermediate tables, and result tables respectively.

The OTs can be used to represent queries for any entity-relationship data paradigm. For instance, in SQL databases the entity types are the tables, the attributes are the columns, and the relationships are represented as tables as well. Similar mapping is possible for other paradigms.

Properties. The OTs have several features:

Question Types: There are different types of questions that can be asked. For instance, 1) yes/no questions (*IsEmpty*), 2) questions about a list of items (*Projection* followed by *Done*), 3) questions about the cardinality of a result set (*Count*), and 4) questions about an aggregation (*Sum, Avg,* etc.).

Result Types: The type of results is defined by the entity types in the result set. For instance, a question can ask about the list of directors that satisfy certain constraints (e.g., all directors that were born in France). In this case, the result type would be the *person* type.

Constraints: The constraints represent the filters that are applied onto the attributes of the entities. For instance, "All directors born in France" sets a constraint on the *birth_place* attribute.

Entity Types: They define which entity types are involved in the query. The selected entity types are combined, usually via a *Join* operation. For instance, in Figure 1 the entity types are *movie* and *person*, which are combined with the table *cast*.

Aggregation Types: They define reduction operations, which are applied to the data. This includes *Min/Max operations* on an attribute, *Set operations* on two sets of relations, and *Group By operations*.

Complexity. In order to categorize the OTs, we define a complexity score similar to (Yu et al., 2018), which is based on the number of components in the tree. The more *Joins* and *Group By* operations, *Aggregations* or *Filters* are in the query, the higher the score. Like (Yu et al., 2018), we define four categories: *Easy*, *Medium*, *Hard*, and *Extra Hard*.

4 Corpus Construction

The evident way to construct a corpus with NL questions and their corresponding OT queries would consist of two main parts: first, collect a set of NL questions, and then create the corresponding OT queries to these questions. However, this approach is very time-consuming and has a major issue. In essence, questions tend to be very narrow in scope, i.e., they do not necessarily cover the

whole range of entity types, attributes and relationships that are present in the database. Moreover, writing the corresponding OT queries for the NL questions requires sufficient SQL skills as well as a mechanism to verify that the OT statements actually correspond to the question.

Thus, we decided to *invert the process*. That is, we first randomly sample an OT using the above-defined context-free grammar, and then annotators write a corresponding question in natural language. In the last step, annotators manually map tokens of the question to the operations. There are several advantages to this procedure: 1) It allows for controlling the characteristics of the OTs, i.e., we can control the question type, the response type, the constraints, and the entity type. 2) It allows them to create more complex questions that better cover the variety of the underlying data. 3) The annotation process is less time consuming, as the annotators do not have to build the trees or write queries. Rather they can focus on writing the question and assigning tokens. We now describe the process of automatic sampling and manual annotation in more detail.

4.1 Tree Sampling

The tree sampling procedure is composed of the following steps:

Question Type: This can be sampled at random or be manually set if a certain type is desired.

Result Type: First, an entity type is randomly sampled. Then a specific set of attributes is sampled from the chosen entity type. Alternatively, the result type can be manually set.

Entity Types: The entity types are sampled based on the graph structure of the entities and relationships in the database schema. For this, we sample from all the possible join-paths, which contain the table of the result type. This is also controllable, as we can specify the length of the paths we want to consider.

Constraints: In the constraints, the filter arguments are sampled. First, the entity types are randomly selected on which the constraints are to be applied. Then we sample an operation and a value at random for each entity type and each attribute. We can limit the number of overall constraints and the number of maximum constraints for each entity type.

Group By: The *Group By* operations (*AvgBy, SumBy, CountBy*) are chosen at random. For a

Group By operation, two attributes need to be selected: a group-attribute, which defines on which attribute to group, and an aggregation-attribute, which defines on which column to apply the aggregation. For instance, we could group by genre and aggregate over the movie budget.

Tree structure: The tree structure is sampled as follows. First, the *Join* operations are applied on the sampled entity types. Second, the set operations (*Union, Intersect, Diff*) are inserted. Third, the *Selection* operations are inserted. Next, the aggregation operations are inserted, i.e., *Group By, Min, Max* operations. Finally, the operations for the question type are sampled. For instance, if the question type is a list of entities, then we use the *Projection* operation, but if it is a cardinality question, we use the *Count* operation.

This procedure may create trees that make no sense semantically. We handle those trees during the annotation phase, which we describe below. Furthermore, we make sure that the trees are executable. For this, we translate the trees into SQL and run them on the database. We also omit trees that return an empty result, as they can lead to confusions during the evaluation, as two different queries that both return an empty result would be counted as being equal.

4.2 Annotation

The annotation process, i.e., writing natural language questions and assigning query tokens to operations in the OT, is performed in two phases. For each phase, we developed a graphical user interface to facilitate the annotation process (for more details, see Appendix D).

Phase 1. In the first phase, the annotator is presented with an OT, which is automatically sampled as described in the previous section. The task of the annotator is to formulate an appropriate NL question for the sampled OT. In some cases, the sampled tree has contradicting or nonsensical constraints (e.g., compute the average year). For these cases, the annotators can either skip or adapt the OT by changing the constraints.

Phase 2. In the second phase, the annotators perform the token assignment as well as quality control. The annotators are presented with an OT and the NL question, which was written by a different annotator in phase 1. First, they check and correct the NL question, then they assign the tokens to the operations. In order to achieve consistent

annotation results, we set up a guideline on how the tokens are to be assigned (more information in the Appendix).

5 Corpus OTTA

We applied our corpus construction procedure to a set of five databases and produced a new corpus with NL questions and corresponding OTs, called OTTA. In order to compare our results with previous work, we used four databases from the *Spider* corpus (CHINOOK, COLLEGE, DRIVING SCHOOL, and FORMULA I), which we extended with a dump from IMDB[3] that we refer to as MOVIEDATA. For the annotations, we employed 22 engineers with basic knowledge in SQL-databases.

5.1 Corpus Statistics

Table 1 summarizes the dataset. The number of tables per database ranges from 6 to 18, and the number of attributes ranges from 45 to 93 columns per database. For CHINOOK and MOVIEDATA, our corpus has more than 1000 annotated OTs, while it has around 500 annotated OTs for the other three databases. For MOVIEDATA, we also performed the token annotation procedure. For each database, we computed the average *complexity* score. Except for MOVIEDATA, which is *Hard*, all other databases have a *Medium* average query complexity. The average time per question annotation ranges from 77 to 104 seconds (average 97.7 seconds). The token assignment and question correction, on the other hand, took on average 101 seconds per OT.

5.2 Corpus Comparison

In order to examine our corpus, we compare its characteristics to the *Spider* corpus and to the *LC-QuAD 2.0* corpus. We compare the coverage of the queried data, the complexity of the natural language questions and the complexity of the corresponding SPARQL/SQL queries.

Coverage. Table 2 shows the major characteristics of the three corpora. We compare the coverage of the databases in terms of the ratio of tables and attributes which appear in the queries.

The average *attribute coverage* of Spider over all databases equals 62.1%. However, more than half of the databases in Spider contain 5 tables or less. Thus, we also report the coverage of attributes

---
[3]https://www.imdb.com/

	MOVIEDATA	CHINOOK	COLLEGE	DRIVING SCHOOL	FORMULA1
#Tables	18	11	11	6	13
#Attributes	64	63	45	39	93
#Queries	1148	1067	462	547	568
Time per Annotation (sec)	104	104	77	78	104
Avg. Complexity	Hard	Medium	Medium	Medium	Medium

Table 1: Statistics of the new corpus OTTA

	#Questions	#Queries	#DB	#Table/DB	Table Cov.	Attr Cov.	MSTTR	Avg. #Tokens	Ann. Time
Spider	10,181	5,693	200	5.1	0.917 (0.87)	0.621 (0.496)	0.519	12.67	360 sec.
LC-QuaD 2.0	30,000	30,000	1	157,068	0.019	0.187	0.761	10.6	-
OTTA (ours)	3,792	3,792	5	11.8	0.949	0.544	0.67	13.53	98 sec.

Table 2: Comparison of our corpus OTTA to the Spider and LC-QuaD 2.0 corpora. Note that the number of databases in LC-QuaD 2.0 is only 1, since it is an open-domain knowledge base, and the number of tables corresponds to the number of different classes. Numbers in parentheses only consider databases with more than 5 tables.

	#Avg. Join	#Group By	#Order By	#Nested	#Having	#Set Op	#Aggregations	#Boolean
Spider	0.537	0.262	0.234	0.148	0.068	0.076	0.519	-
LC-QuaD 2.0	2.05 hops	0	0.041	0	0	0	0.048	0.089
OTTA (ours)	1.19	0.133	0	0	0.117	0.02	0.4	0.161

Table 3: Comparison of the query complexity based on the ratio of components per query. For the aggregations in LC-QuaD 2.0, we report the number of queries that use a *Count* operation.

only considering the databases which have more than 5 tables, where Spider only covers 49.6% of attributes. Corpus OTTA, in contrast, covers 54.4% of all attributes. Furthermore, the divide becomes more apparent when we consider databases with larger amounts of tables. For instance, for the FORMULA-1 database, our corpus covers 44.2% of all attributes, in contrast to Spider, where only 22.1% of attributes are covered. LC-QuaD 2.0 covers 1,310 out of 7,005 properties[4] (i.e. attributes in SQL), which corresponds to 18.7%. This is an extensive coverage, considering the high amount of properties.

The *table coverage* shows a similar picture: our approach covers 94.9% of all tables in the databases, whereas Spider covers 91.7%. This number drops down to 87% when considering only databases with more than 5 tables. Again, this effect is most pronounced for the FORMULA-1 database, where we cover 92% of the tables, whereas Spider only covers 69.2%. This shows that our method better scales to larger databases, which is relevant for real-world applications, where databases with a vast number of tables exist. LC-QuaD 2.0 covers around 1.9% of approx. 160k classes, which makes comparison hard, as it is impossible to cover this vast amount of classes with

30k queries.

Query Complexity. In order to compare the complexity of the queries, we examine the number of occurrences of different components in the queries (see Table 3).

We first observe that our corpus OTTA does not contain any queries with *Order By* operators or nested queries - however, they could be easily added to the grammar to fill this gap. Furthermore, Spider contains more aggregation operations (in particular *Min, Max, Count, Average*, and *Sum*). Again, this could be easily adapted in our corpus by sampling more trees that contain these aggregations. On the other hand, our corpus stands out in the number of *joins* per query: on average OTTA has 1.19 join operations per query in contrast to Spider, which has 0.537 joins per query. In fact, about 40% of the queries in Spider contain joins, whereas OTTA is composed of 54% of queries, which contain at least one join operation. Furthermore, around 37% of our queries contain two joins in contrast to 9% in Spider. On the other hand, *LC-QuaD 2.0* contains an average of 2 hops (equivalent to two joins in relational databases) per query, which lies in the nature of graph database queries that are optimized for handling queries that range over multiple triple patterns. However, *LC-QuaD 2.0* lacks complexity when considering more complex components (e.g., Group By, Set-Operation, etc.). In addition to the operations in relational

---

[4] For the number of classes and properties in Wikidata, we consulted: `https://tools.wmflabs.org/sqid`

algebra, the OTs also support *Boolean* questions (i.e., yes/no questions), which make 16.1% of our corpus compared to 8.9% in LC-QuaD 2.0.

Question Complexity. The lexical complexity of the NL questions is measured in terms of *mean-segmental token-type-ratio (MSTTR)* (Covington and McFall, 2010), which computes the number of different token types in relation to all tokens in a corpus. The MSTTR is computed over text segments of equal length, in order to avoid biases due to different lengths within the corpora. First, note that the average length of the questions in all three corpora is approximately the same, between 10.6-13.6 tokens on average. Table 2 shows that our corpus contains a much higher lexical complexity of the questions than Spider (0.67 instead of 0.52). Thus, our approach seems to avoid trivial or monotonous questions, which also matches with our impression from manual inspection. On the other hand, the lexical complexity is higher in *LC-QuaD 2.0*, which is due to the open domain nature of the dataset.

Examples. In Table 4, we show examples of questions from OTTA compared to questions from Spider. The examples show that the quality of the questions is similar. The *easy* questions in both datasets are often only simple filtering questions on one table. *Medium* complexity questions include join operations and filters. *Hard* questions in both datasets include join operations and aggregation operations such as finding the maximum or computing the average. The biggest difference is in the *Extra* complexity. There Spider focuses more on subqueries in the where clause. OTTA, on the other hand, focuses more on larger join paths, which are typical for real-world database queries as well as group-by operations and aggregations.

6 Baseline Systems

Baseline model. As baseline model for OTs from NL questions, we follow the *Syntactic Neural Model for Code Generation* by (Yin and Neubig, 2017), which we refer to as *Grammar-RNN*[5]. This model is based on an encoder-decoder architecture that learns to generate a sequence of production rules of an arbitrary grammar, which in turn produces the query for a given question. For a more detailed discussion on this architecture, we refer

---

[5]The IR-Net (Guo et al., 2019) is also based on the Grammar-RNN. During the time of writing this paper, IR-Net was ranked second on the Spider leader board.

the reader to (Yin and Neubig, 2017). In our case, it learns to generate the rules defined in Figure 2 for a given question in natural language. Based on the generated list of rules, an OT is created.

We train the model in two phases - a *pre-training phase* and a *supervised phase*. In the pre-training phase, we train a grammar-autoencoder on large amounts of randomly sampled OTs. In the supervised phase, we replace the grammar-encoder by a text encoder and train on the labelled dataset, i.e., the samples with NL question and corresponding OT.

Encoder. For the NL question, we use a standard Gated-Recurrent Unit (GRU) (Chung et al., 2014) to encode the question. If w_i denotes the representation of the i-th token in the question, then the encoder produces a corresponding hidden state h_i^E. Let $H^E \in \mathbb{R}^{N \times h}$ denote the concatenation of all hidden states produced by the GRU for one question, where N is the number of tokens and h the size of the hidden state.

Decoder. The decoder learns to generate a sequence of production rules with which a tree y is generated for a given encoding x of the NL question. The generation process is formalized as:

$$p(y \mid x) = \prod_{t=1}^{T} p(a_t \mid x, a_{<t}, a_{p_t}) \qquad (1)$$

a_t is the action taken at time t, $a_{<t}$ are the actions taken before time t, a_{p_t} are the parent actions taken, and x is the encoded input question. There are two different types of rules that the model applies during decoding: 1) If the current rule generates a non-terminal symbol, then *ApplyRule[r]* is executed, which applies a production rule to the current tree. 2) If the next symbol is a terminal, then *GenToken[v]* is applied, which selects the token from a vocabulary. In our case, we have different types of tokens to be generated: table-names, attribute-names and filter operations. Similar to *Grammar-RNN*, we implement the decoder using a recurrent neural network, where the internal state is given by:

$$h_t = GRU([a_{t-1} : c_t : a_{p_t} : n_{f_t}], \widetilde{h}_{t-1}) \qquad (2)$$

n_{f_t} is the embedding of the current node type (e.g. average, union, ...), c_t is a context vector that is computed by applying soft-attention over the input hidden states H^E, and h_{t-1} is the hidden vector of the last state. In contrast to (Yin and Neubig,

Hardness	Spider	OTTA
easy	Find the number of albums.	Where were the invoices with the total sum of 1.99 or smaller issued?
	What is the average unit price of all the tracks?	What are the unit prices of tracks composed by Alfred Ellis/James Brown?
	Find all the customer information in state NY.	To which country belongs the 89503 postal code?
medium	Count the number of tracks that are part of the rock genre.	What is the average length of the tracks in the Grunge playlist?
	Please show the employee first names and ids of employees who serve at least 10 customers.	When did we sell tracks larger than 8675345 bytes?
	Find the name of the artist who made the album "Balls to the Wall".	To which postal codes did we sell a track named Headspace?
hard	What is the average duration in milliseconds of tracks that belong to Latin or Pop genre?	How many different playlists with a track that is bigger than 7045314 bytes do exist?
	What are the names of artists who have not released any albums?	What is the album title having the track with the lowest length in milliseconds in the genre name Sci Fi & Fantasy?
	What are the last names of customers without invoice totals exceeding 20?	What are the genres from artists not named Scholars Baroque Ensemble?
extra	What is the name of the media type that is least common across all tracks?	Whats the total unit price sold to customers with the email hholy@gmail.com and Argentina as billing country?
	Count the number of artists who have not released an album.	How many different genres do the tracks have, which were bought by customers who live in France?
	What are the album titles for albums containing both Reggae and Rock genre tracks?	Which customers made at least 35 purchases, excluding titles from the Chico Science & Nacao Zumbi album?

Table 4: Example questions from OTTA and Spider. We grouped the examples by the hardness scores. The examples are for the Chinook domain, which is an online music store database.

2017), we apply attention based on (Luong et al., 2015), where $\widetilde{h}_{t-1} = tanh(W_c[h_{t-1} : c_t])$.

For the selection of the terms, we have four output matrices W_R, W_T, W_A, W_C, where W_R encodes the grammar rules (i.e. for the non-terminal symbols), and W_T, W_A, W_C encode the table names, attributes and comparison operations, respectively. Depending on the current frontier node, the next output is computed by:

$$a_t = argmax(softmax(W_R * h_t)) \quad (3)$$

Grammar Encoder. The tree encoder, which we use for the pre-training, is based on the same GRU architecture as the decoder. The hidden states for each rule are computed by:

$$h_t = GRU([a_{t-1} : a_{p_t} : n_{f_t}], h_{t-1}) \quad (4)$$

In contrast to the encoder, there is no context vector c_t. Moreover, h_{t-1} is the last hidden state computed by the GRU. The output of the encoder is a sequence of all states: $H^R \in \mathbb{R}^{R \times h}$, where R denotes the number of rules in the encoded tree.

Token Attention. A straight-forward method to include the explicit token alignment, which is created in the second annotation phase, is to force the attention mechanism to learn the alignment. For this, we add an extra loss function, which computes the binary cross entropy for each attention weight. More formally, let $\alpha_t = softmax(h_{t-1}H^E) \in \mathbb{R}^N$ be the attention weights computed for timestep

t (during the pre-training phase H^E is replaced by H^R). Then let $\alpha_t^{(i)}$ be the attention weight for the i-th token. For each token we add the loss

$$g_i * log(\alpha_t^{(i)}) + (1 - g_i) * log(1 - \alpha_t^{(i)}), \quad (5)$$

where $g_i \in [0, 1]$ denotes if the token is assigned to the current node or not.

7 Results

We now report the results of our model. The details of the experimental setup can be found in Appendix A. Each experiment is repeated five times with different random seeds. Table 5 shows the precision of the *Grammar-RNN* on the 5 datasets of OTTA. The *precision* is defined as the exact result set matching between the gold standard query and the generated query. Furthermore, the table shows the average precision for each query complexity category. The column "Weighted Avg." refers to the mean average precision over all queries irrespective of the query complexity category.

Precision. For all the databases, except FORMULA-1, the model achieves a precision between 45.1% and 47.5%. For FORMULA-1 the model only achieves a score of 26.3%. This could be explained by the fact that the FORMULA-1 database contains 93 different attributes, and our data only covers 42 of these attributes. Furthermore, each attribute appears only 17.1 times per query on average. In contrast, for the COLLEGE

904

database the attributes appear in 56 queries on average. Thus, it is harder for the model to learn attributes, which do not appear often in the training set. For most of the databases, the model cannot handle the *extra hard* questions, which often contain multiple joins, aggregations, and/or group by operators. Note that without the pre-training phase, the scores drop by a large margin. For instance, the scores for Moviedata drop below 30% precision.

	Easy	Medium	Hard	Extra Hard	Weighted Avg.
Moviedata	0.645	0.619	0.437	0.108	0.475
Chinook	0.610	0.442	0.396	0.482	0.473
College	0.525	0.739	0.294	0.077	0.468
Driving School	0.518	0.272	0.611	0.187	0.451
Formula 1	0.355	0.075	0.0	0.0	0.263

Table 5: Precision of queries against our 5 datasets according to query complexity. "Weighted Avg." refers to the mean average precision over all queries irrespective of the query complexity category.

Benefit from Token Assignments. We now evaluate whether the token assignments can help to train better models. Figure 3 displays the learning curves for the MOVIEDATA database *with and without* the token assignment. The model is trained with 20%, 40%, 60%, 80%, and 100% of the data. The results show that using the token assignment increases the scores by around 2%. In the case of 20% training data, the gain is even as high as 7%, thus showing that the model can benefit from the additional information that is provided in the token assignments.
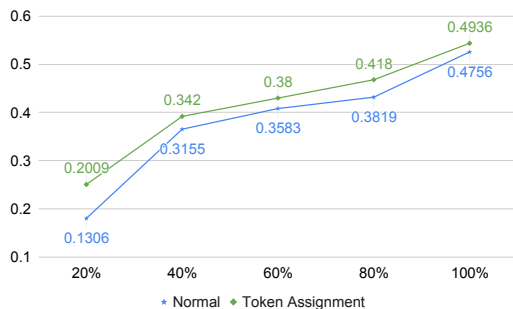

Figure 3: Learning curve for 20%, 40%, 60%, 80%, and 100% of the data on database *MOVIEDATA* as part of OTTA. We compare the scores for the training with and without token alignment.

8 Conclusion

In this paper, we introduced a fast annotation procedure to create NL queries and corresponding database queries (in our case, Operation Trees).

Our procedure more than triples the velocity of annotation in comparison to previous methods, while ensuring a larger variety of different types of queries and covering a larger part of the underlying databases. Furthermore, our procedure allows a fine-grained alignment of tokens to operations. We then used our new method to generate *OTTA*, a novel corpus for semantic parsing based on operation trees in combination with token assignments. Generating this corpus was more time- and cost-efficient than with previous approaches. Our statistical analysis showed that the corpus yields a higher coverage of attributes in the databases and more complex natural language questions than other existing methods. Furthermore, we implemented a baseline system for automatically generating OTs from NL queries. This baseline achieves scores of up to 48% precision, which are already reasonable while also leaving large potential for improvement in future research. Finally, we showed that the inclusion of the token alignment results in an increase of precision of up to 7%.

Based on these results, we will explore ways to leverage the token assignment to domain adaption and few-shot learning. We also plan to enhance the annotation process by automatically generating proposals for the NL questions and token assignments and letting the annotators only perform corrections. We hope that this increases annotation efficiency even more.

References

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

Fuat Basik, Benjamin Hättasch, Amir Ilkhechi, Arif Usta, Shekar Ramaswamy, Prasetya Utama, Nathaniel Weir, Carsten Binnig, and Ugur Cetintemel. 2018. Dbpal: A learned nl-interface

for databases. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1765–1768. ACM.

Hannah Bast and Elmar Haussmann. 2015. More accurate question answering on freebase. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 1431–1440. ACM.

Lukas Blunschi, Claudio Jossen, Donald Kossmann, Magdalini Mori, and Kurt Stockinger. 2012. Soda: Generating sql for business users. *Proceedings of the VLDB Endowment*, 5(10):932–943.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146.

Jianpeng Cheng, Siva Reddy, Vijay Saraswat, and Mirella Lapata. 2019. Learning an executable neural semantic parser. *Computational Linguistics*, 45(1):59–94.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

Michael A. Covington and Joe D. McFall. 2010. Cutting the gordian knot: The moving-average typetoken ratio (mattr). *Journal of Quantitative Linguistics*, 17(2):94–100.

Deborah A. Dahl, Madeleine Bates, Michael Brown, William Fisher, Kate Hunicke-Smith, David Pallett, Christine Pao, Alexander Rudnicky, and Elizabeth Shriberg. 1994. Expanding the scope of the atis task: The atis-3 corpus. In *Proceedings of the Workshop on Human Language Technology*, HLT '94, pages 43–48, Stroudsburg, PA, USA. Association for Computational Linguistics.

Danica Damljanovic, Milan Agatonovic, and Hamish Cunningham. 2010. Natural language interfaces to ontologies: Combining syntactic analysis and ontology-based lookup through the user interaction. In *Extended Semantic Web Conference*, pages 106–120. Springer.

Mohnish Dubey, Debayan Banerjee, Abdelrahman Abdelkawi, and Jens Lehmann. 2019. Lc-quad 2.0: A large dataset for complex question answering over wikidata and dbpedia. In *International Semantic Web Conference*, pages 69–78. Springer.

Sébastien Ferré. 2017. Sparklis: an expressive query builder for sparql endpoints with guidance in natural language. *Semantic Web*, 8(3):405–418.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018. Improving text-to-sql evaluation methodology. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 351–360, Melbourne, Victoria, Australia.

Alessandra Giordani and Alessandro Moschitti. 2012. Automatic generation and reranking of sql-derived answers to nl questions. In *Proceedings of the Second International Conference on Trustworthy Eternal Systems via Evolving Software, Data and Knowledge*, EternalS'12, pages 59–76, Berlin, Heidelberg. Springer-Verlag.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-SQL in cross-domain database with intermediate representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535, Florence, Italy. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017a. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017b. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973, Vancouver, Canada. Association for Computational Linguistics.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Fei Li and HV Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84.

Haoyan Liu, Lei Fang, Qian Liu, Bei Chen, LOU Jian-Guang, and Zhoujun Li. 2019. Leveraging adjective-noun phrasing knowledge for comparison relation prediction in text-to-sql. 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)*, pages 3506–3511.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal. Association for Computational Linguistics.

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, IUI '03, pages 149–157, New York, NY, USA. ACM.

Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R Mittal, and Fatma Özcan. 2016. Athena: an ontology-driven system for natural language querying over relational data stores. *Proceedings of the VLDB Endowment*, 9(12):1209–1220.

Alkis Simitsis, Georgia Koutrika, and Yannis Ioannidis. 2008. Précis: from unstructured keywords as queries to structured databases as answers. *The VLDB JournalThe International Journal on Very Large Data Bases*, 17(1):117–149.

Dezhao Song, Frank Schilder, Charese Smiley, Chris Brew, Tom Zielund, Hiroko Bretz, Robert Martin, Chris Dale, John Duprey, Tim Miller, et al. 2015. Tr discover: A natural language interface for querying and analyzing interlinked datasets. In *International Semantic Web Conference*, pages 21–37. Springer.

Lappoon R. Tang and Raymond J. Mooney. 2000. Automated construction of database interfaces: Intergrating statistical and relational learning for semantic parsing. In *2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 133–141, Hong Kong, China. Association for Computational Linguistics.

Yushi Wang, Jonathan Berant, and Percy Liang. 2015. 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)*, pages 1332–1342, Beijing, China. Association for Computational Linguistics.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Sqlizer: Query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA):63:1–63:26.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. 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*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, AAAI'96, pages 1050–1055. AAAI Press.

Weiguo Zheng, Hong Cheng, Lei Zou, Jeffrey Xu Yu, and Kangfei Zhao. 2017. Natural language question/answering: Let users talk with the knowledge graph. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 217–226. ACM.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

A Experimental Setup

Preprocessing We use Spacy[6] to tokenize the NL questions in OTTA. In order to find the entities for the constraints, we employ simple string matching. With this, we find 96% of all entities. Thus, the generated OTs are executable, and we can compare the results of the generated OT to the results by the gold-standard OT from the corpus.

Model Configuration. For our model, we chose a hidden layer of $h = 256$ dimensions. We optimize using the *Adam* (Kingma and Ba, 2014) optimizer with the standard values. We let the model train using early stopping with a patience on the validation loss, where the validation set is the left-out fold in 5 fold cross-validation. For the word embeddings, we use the pre-trained *FastText* embeddings (Bojanowski et al., 2017), which are refined during the training phase.

B Query Complexity

The tables below show more details of the coverage (see Tables 6 and 7) and the average number of joins per query (see Table 8).

C Example of Tree Sampling

Figure 4 shows a randomly sampled tree. During Phase 1 of the annotation procedure, an annotator associated the tree with the question: *What is the average movie vote of different movies having an Oscar nominee with a cast character called Jesse and were nominated for an Oscar in the year 1991 or later?*. In the second phase of the annotation, the tokens of the questions were associated with the nodes of the tree. The tree is depicted from root to leaves, where the root node is the last operation, and the leave nodes are the *GetData*-nodes. Here we describe the tree sampling procedure in more detail with the tree in Figure 4 as an example.

1 The query type is selected. There are five different types: list, sum, count, average, and Boolean. In our example, the average was selected. This can be forced manually or randomly sampled.

2 The result type is selected, which, in this case, is *movie.vote_average*. This can also be set manually or be sampled at random. Based on the query type, only certain types of results

are allowed. More precisely, for average and sum operations, only numeric result types are allowed.

3 The join path is selected. In the first step, a path length is selected, which can be predefined or randomly sampled. In this case, the path length is set to five. Then, in a second step, a random path of the predefined length is selected. In the current example, the query path is: *movie, cast, person, oscar_nominee, oscar*. The path always starts with the result type of table.

4 The set operation is selected among *union*, *intersection*, or *difference*. In this example, there is no set operation. After the operation was selected, a subpath is chosen, on which the set operation is performed. For instance, if we wanted to know the movies where Brad Pitt and George Clooney worked together, then the subpath *movie, cast, person* is selected. Finally, two different filters are inserted, one for each actor.

5 The *group by* operation is selected. First, the operator is selected among *sum, average*, or *count*. Then, the *group by* attribute and the *aggregation* attribute are selected. In our example, there is no group by operation.

6 The aggregation operation is selected among the *min* and *max* operation. This is relevant for the questions of the type: Which movie has the highest rating. In this example, we have no aggregation operation.

7 The filters are selected. For this, we define the number of total filters and the maximal number of filters per table. In this case, we set the number of filters equal to 2, and the maximal number of filters per table to one. Then, the appropriate number of attributes is selected randomly alongside the path. In this case, the tables oscar and cast were selected. Then, an attribute is selected, followed by a comparison operator and a value, which is randomly sampled from the database. In our example, we have: oscar.year $\geq$ 1991 and cast.character = Jesse.

D Annotation Tool

The annotation process is performed in two phases: writing an NL question for a given OT, and as-

	TOTAL	CHINOOK	COLLEGE	DRIVING SCHOOL	FORMULA_1
SPIDER	0. 917 (0.87)	0.727	0.909	1	0.692
OUR DATASET	0.949	1	0.818	1	0.923

Table 6: Table Coverage, in % to total amount of existing tables. Our dataset shows better table coverage, except for one database (college), where the coverage differs by one table. The biggest improvement in coverage was achieved on the database formula_1, which is also the most complex database with the largest amount of tables. The number in braces indicates the average table coverage for the databases with more than 5 tables.

	TOTAL	CHINOOK	COLLEGE	DRIVING SCHOOL	FORMULA_1
SPIDER	0.621 (0.496)	0.354	0.383	0.730	0.221
OUR DATASET	0.544	0.584	0.384	0.756	0.442

Table 7: Attribute Coverage. Our method gives better attribute coverage in particular for larger datasets, for instance, FORMULA_1. The number in braces indicates the average attribute coverage for the databases with more than 5 tables.

	TOTAL	CHINOOK	COLLEGE	DRIVING SCHOOL	FORMULA_1
SPIDER	0.504	0.667	0.412	0.441	0.925
OUR DATASET	1.15	0.95	1.18	0.837	1.2

Table 8: Average number of joins per query.

signing tokens from the NL question to the nodes within the OT. We have built two user interfaces, one for each phase. Figure 5 shows screenshots of both tools.

Phase 1. In the first phase, the annotators are presented with an OT and the constraints. Their task is to write an appropriate question for the OT. For this, they can adapt the constraints, in case that they are nonsensical. Furthermore, the annotators can access the intermediate results for each node in the tree to better understand what the OT does. In cases where the OT cannot be annotated with an appropriate question, the OT can be skipped.

Phase 2. For the second phase (Figure 5 (b)), the annotators are presented with an OT and an NL question, which was written by another annotator in the previous phase. The task is to correct the question, and then assign the tokens of the NL to the nodes (i.e., operations) in the tree. For this task, the tool guides the annotator from node to node in the OT. Moreover, for each node, the annotator can choose the corresponding tokens. In the final step, the annotators can correct their token assignment using drag-and-drop features.

Guidelines. In order to have consistent annotations (especially in the second phase), we provided the annotators with extensive tutorial videos. On average, the annotators took 30 minutes to get used to the tool and start to work efficiently. For the first phase, we instructed the annotators to write an

appropriate question and gave examples, as well as examples of pitfalls.

For the second phase, we introduced stricter guidelines, as we noticed that annotators had trouble with this step. Especially, the join operations were not clear to the annotators. Thus, we decided on the following rules:

- Table: If the table denotes an entity type (e.g., movie), the tokens that denote this entity type are to be assigned (e.g., "movies"). If the table is a bridge table, which denotes a relationship between entities (e.g., production_country), then the tokens that denote this relationship are to be assigned to the operation (e.g., "movies", "produced", "in").

- Joins: For the join operations, the same guidelines as for the bridge-tables are to be followed.

- Filter: For the filter constraints (e.g., "person_name= Tom Cruise") the tokens, which represent the constraint, are to be selected (e.g., "by", "Tom", "Cruise").

- Query type: For each query type (e.g., count, average, sum, ...), the tokens that correspond or trigger this question type are to be selected (e.g., "How", "many").

Annotators. We recruited 22 annotators, which have a basic understanding of database technologies. We paid each annotator 25$ per hour. Each
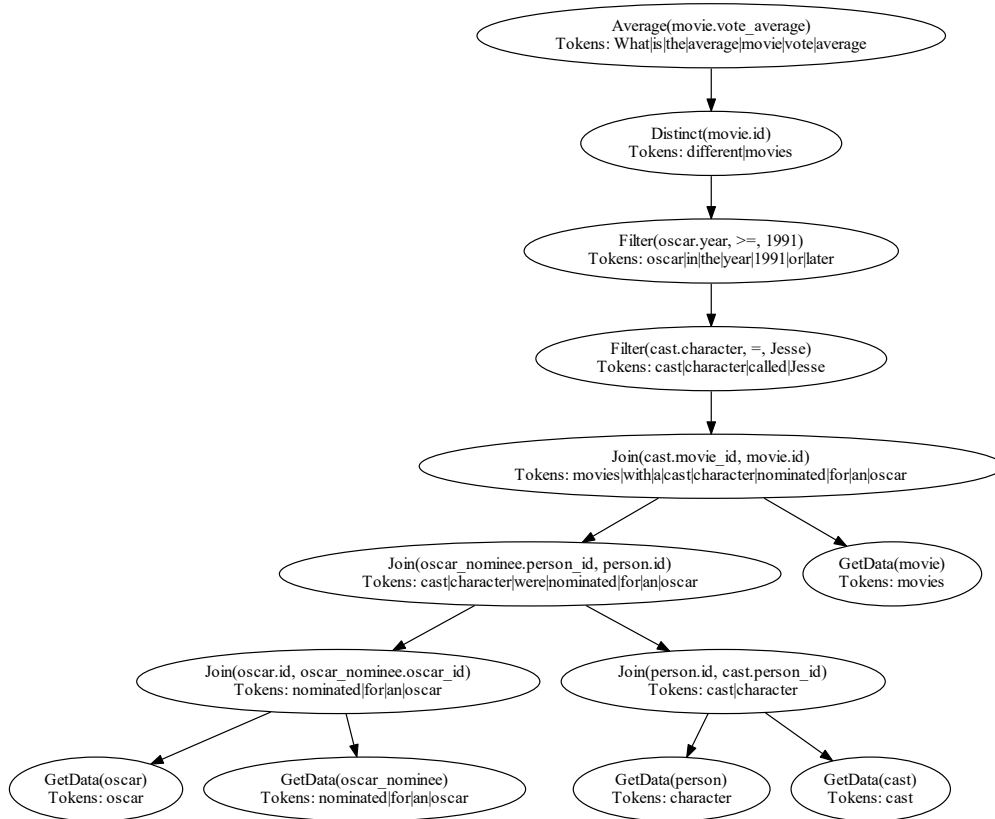
Figure 4: Example of a randomly sampled tree. The nodes denote the node type with their arguments. The *Tokens* are assigned during the second phase of the annotation process. This tree is the answer to the question: *What is the average movie vote of different movies having an Oscar nominee with a cast character called Jesse and were nominated for an Oscar in the year 1991 or later?*

annotator was given access to a set of instruction
videos as well as a user manual. Furthermore, the
annotators could pose questions in a forum.

(a)

(b)

Figure 5: The annotation tool. (a) The OT and the constraints are shown to the annotators. For each node, the annotators can inspect the result of the execution. The annotators write a question and (b) assign the tokens of the question to the operations.