ezNL2SQL: A System for Network Devices Management with a Natural Language Interface for Databases

Jasmina Bogojeska[1]
IBM Research - Zurich
Switzerland
jbo@zurich.ibm.com

David Lanyi[1, 2]
IBM Research - Zurich
Switzerland
david.lanyi@continental.com

Mirela Botezatu[1, 3]
IBM Research - Zurich
Switzerland
mirela.botezatu@gmail.com

Dorothea Wiesmann[3]
IBM Research - Zurich
Switzerland
wiesmanndr@google.com

***Abstract*—To enhance and accelerate the work of network management experts of large IT environments, we present ezNL2SQL. It is a fast and flexible system that provides a natural language interface to a database in a domain-specific setting, where column and table names do not correspond to their underlying entities and a large labelled corpus is not available. Our system comprises three main components: linguistic index, natural language query analyzer and SQL generator, and an easy-to-use UI. Given an input question and a target database, ezNL2SQL generates a valid SQL statement for querying data that best satisfies the information request specified by the question. The result is presented to the user in the most suitable visual format. We demonstrate the usefulness and efficiency of the system in a user study with network management experts and a complex network security inventory database.**

I. Introduction

A large amount of valuable information pertaining to the network devices in large IT environments is stored in relational databases accessible via SQL queries. Writing such queries, however, is challenging and time consuming not only for users without the necessary technical background but also for users unfamiliar with the database schema. Providing an intuitive, easy-to-use interface for querying the information relevant for the network devices in a target IT environment would significantly enhance the management and maintenance of these devices. A natural language interface to a database (NLIDB) is an easy-to-use and intuitive way for users to access the information stored in relational databases as people use natural language every day to communicate among each other.

The main focus of this paper is to describe a natural language interface to a database (DB) for management of network devices where the table and column names do not correspond to the entities they represent and there is no large amount of labeled data available. Related approaches are not readily applicable to this setting, even though it often occurs in practice. To achieve this, we design and implement ezNL2SQL, a system composed of a static component that projects the target database model into a linguistic space, and a dynamic component that analyzes natural language queries, produces corresponding SQL statements, and retrieves the result. The UI then displays the result in the most suitable format (e.g., a certain type of chart or a table). The main contributions of this work are the following:

1) *Mapping Language.* We devise a declarative language to define associations between database objects and natural language mentions;
2) *Natural Language Query Analysis.* We develop an algorithm for discovering and disambiguating natural language mentions of database objects, filter conditions and functional operations in questions;
3) *Query Generation.* We render valid SQL statements based on an abstract query model;
4) *User Study.* We evaluate the usefulness and performance of our system in a user study with field experts using a network security inventory database.

When storing information from a certain target domain in a relational DB that needs to be accessed by users with different backgrounds, one should, in parallel, be able to also provide a natural language interface that enables intuitive information access for all the interested users. Thus, the goal of our work is to provide an easy-to-apply, general approach that enables access to the domain information stored in a relational DB using domain-specific vocabulary. To apply our approach one needs to provide a mapping that associates the database objects with their corresponding natural language entities in the target domain which can easily be achieved by the DB experts who create or maintain the target DB. Such mapping can later be expanded using synonyms or word embeddings.

The remainder of this paper is organized as follows. Section II describes the system architecture, Section III, IV and V detail the internal components and Section V— discusses results from the user study. We give details on the deployment in Section VII, review related work in Section VIII and conclude in Section IX.

II. System Overview

The ezNL2SQL system has three components:

1) The *static component* – which consists of an annotated database obtained by mapping the objects in the target database model to a linguistic space using the DB mapping language we develop for this purpose;
2) The *dynamic component* – a system for processing questions, generating SQL queries and retrieving data.

1 Equal contribution.

2 Author's new affiliation is Continental ADAS AI, Hungary.

3 Author's new affiliation is Google Zurich, Brandschenkestrasse 110, 8002 Zürich

978-3-903176-32-4

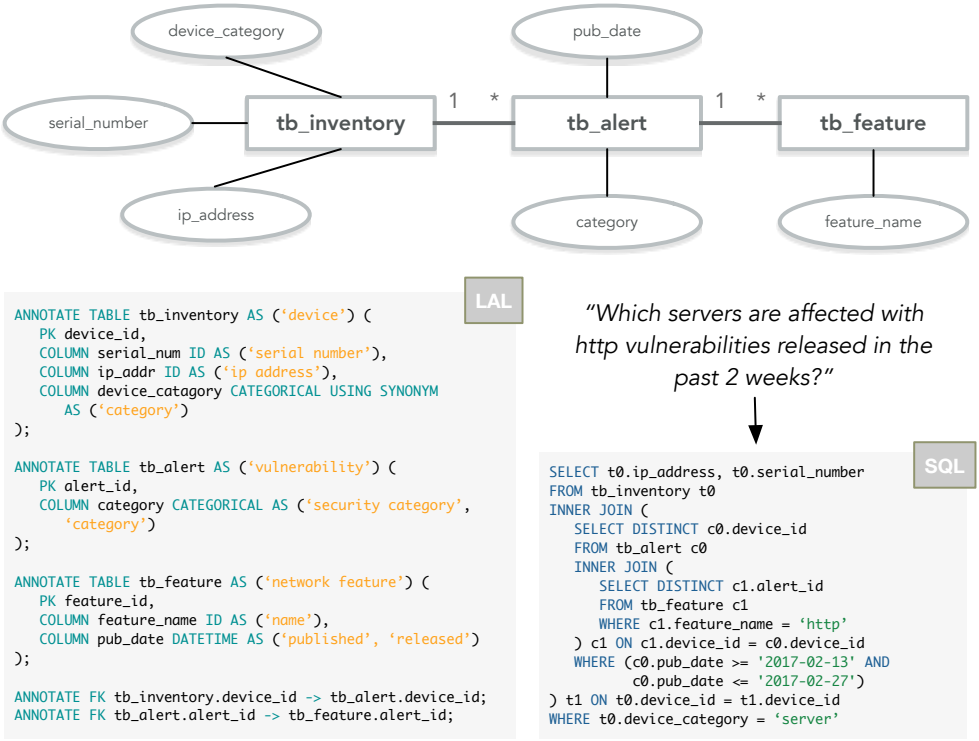

Fig. 1: DB schema with linguistic annotation used for answering natural language questions

This component comprises several subcomponents, namely: a DB object discovery, a DB filter assignment, an aggregation assignment, and a disambiguation component described in detail in Section 4, and the SQL query generation component described in Section 5;

3) The *UI component* – a user-friendly interface for posting questions and retrieving answers from the system.

The system is depicted in Figure 2. The linguistic annotations for the database are stored in an annotation file, which is parsed to produce what we call a serialized annotated database object. Serialization is used to persist the annotated database object in the file system and thus provide an in-memory, lightweight, annotated DB object. This object contains the semantic graph of the database – a graph whose nodes are database tables and whose edges are foreign key relationships between the tables. In addition, it contains an inverted index, which stores the mapping from each database entity to its set of corresponding aliases. Aliases can be *explicit* – the synonyms for the database entity name stored in the annotation file – or *implicit* – the values for a categorical column contained in the database. Most of the *dynamic* part of our system is contained in the *resolving* component, which contains all the logic for translating an input question into an SQL statement. Note that this system is able to provide an NLIDB for an arbitrary annotated database.

III. Linguistic Annotation

An integral step in processing natural language questions is to identify mentions of database-related objects in the question text. Our goal is to identify mentions of (1) tables, (2) columns, (3) values of columns, (4) functional operations and (5) foreign key relationship between tables.

A. Annotation Model

The ezNL2SQL system requires a target database annotated with natural language terms, which we denote as *linguistic annotation*. It comprises a set of clauses that associate elements of the DB schema with one or more representative natural language *aliases*. For example, an annotation could associate a table `tbl_alert` with the nouns "vulnerability", "security notice" and "alert", and its column `pub_date` with "released" and "published". The annotation provides an abstraction of the DB schema in natural language semantics. In our view, this simplification is reasonable as it is often cumbersome to specify a full ontology without relevant expertise and an ontology on its own does not account for the diversity of language use in questions. In addition, the linguistic annotation model yields a simple way to construct an index for detecting database-relevant mentions in question texts.

In the remainder of this section we introduce the Linguistic Annotation Language (LAL) – an easy-to-use SQL-like formal language for linguistic database annotation.

1. General Syntax and Grammar. The annotation syntax is designed to be simple and readable – similar to the widely used SQL Data Definition Language (DDL) to enable easy adoption by DB experts. The formal grammar of the language is specified in Grammar 1.

⟨*ldl*⟩ ::= (⟨*table_annotation*⟩ | ⟨*fk_annotation*⟩)

⟨*table_annotation*⟩ ::= 'ANNOTATE TABLE' ⟨*table*⟩ 'AS' ⟨*aliases*⟩ '(' ⟨*pk*⟩ { , (⟨*col_annotation*⟩ | ⟨*expr_annotation*⟩) } ');'

⟨*pk*⟩ ::= 'PK' ⟨*col*⟩

⟨*aliases*⟩ ::= '(' ⟨*alias*⟩ { , ⟨*alias*⟩ } ')'

⟨*alias*⟩ ::= 'FL'⟨*text*⟩'FL'

⟨*col_annotation*⟩ ::= 'COLUMN' ⟨*col*⟩ ⟨*type*⟩ 'AS' ⟨*aliases*⟩

⟨*expr_annotation*⟩ ::= 'EXPRESSION' ⟨*expr*⟩ ⟨*type*⟩ 'AS' ⟨*aliases*⟩

⟨*type*⟩ ::= ('NUMERIC' | 'DATETIME' | 'ID' | 'TEXT' | ⟨*categorical*⟩ | ⟨*boolean*⟩)

⟨*categorical*⟩ ::= 'CATEGORICAL' { 'USING SYNONYM' }

⟨*boolean*⟩ ::= 'BOOLEAN' ⟨*expr0*⟩ 'AS' ⟨*aliases*⟩ { ⟨*expr1*⟩ 'AS' ⟨*aliases*⟩ }

⟨*fk_annotation*⟩ ::= 'ANNOTATE FK' ⟨*table*⟩.⟨*col*⟩ '->' ⟨*table*⟩.⟨*col*⟩ { 'DOWNSTREAM AS' ⟨*aliases*⟩ } { 'UPSTREAM AS' ⟨*aliases*⟩ } ';'

Grammar 1: Formal grammar of the Linguistic Annotation Language.

2. Table and View Annotations. The annotation model contains clauses about tables and foreign key relationships. An annotated table assigns a set of NL aliases, most often nouns, to a certain table or view of the DB. It also specifies the primary key of the table, which is important when rendering a SQL query, e.g. to perform joins or count operations. The current syntax only supports simple primary keys, but it is easy to expand the language to accept compound primary keys as well. A table annotation may encapsulate one or more column- and column-expression annotations. A high-level view of the system is given in Fig. 2.

3. Column Annotations. Column annotations are defined inside the body of a table annotation. They assign one or more natural language aliases to a column and specify the column type. In contrast to DDL, the column type in LAL focuses on the semantics rather than the logical or physical organization of the data. The column type provides useful information when translating NL questions to SQL. There are six column types an annotation can specify:

1) `ID`: any value that is unique among the records and may be used to identify a specific record;
2) `CATEGORICAL`: any data that specifies a categorical classification for a record;
3) `NUMERIC, DATETIME, BOOLEAN`;
4) `TEXT`: everything else with no further assumptions on semantics.

Some categorical columns can also be annotated as *table synonym* when it is known that the category values can be used as a synonym for the entity represented by the table. For example, if the annotated table represents "devices", and there is a categorical column defining the device type such as "router", "switch" or "firewall", the values of that column are considered "devices" and will be treated as a mention of the table itself.

4. Expression Annotations. In addition to annotating columns of a table directly, LAL enables the annotation of more general SQL expressions. Anything that can be formulated in an unaggregated SELECT statement over table columns can also be annotated in LAL. Using the full capability of SQL in expression annotations (e.g. `CASE-WHEN` constructs and other functions), one can define very complex concepts or predicates about the target entity.

5. Foreign Key Annotations. It is important to have a model of the relationships between the annotated tables. In case a join operation is required to answer a question, the foreign key (FK) annotations provide information on how valid look-ups can be performed across the tables in the DB. FK annotations specify pairs of tables that are in a one-to-many relationship. With LAL, one can specify NL aliases for either or both of the upstream (many-to-one) and downstream (one-to-many) directions.

B. Natural Language Index

Once the DB annotation model is available, we build the inverted index of the linguistic aliases, where each index term is associated with one or more database objects via a mention type. Based on how the aliases might be used in questions, we index them assigning one or more of the following mention types (each pointing to their respective DB object):

1) **Table** mentions for all alias terms of table annotations;
2) **Column** mentions for all alias terms of column annotations;
3) **Foreign key** upstream and downstream mentions for alias terms of FK annotations;
4) **Boolean** mentions for value aliases of Boolean-typed columns. The index reference also contains information on which of the two possible states the alias refers to;
5) **Category** mentions for all unique values of categorical columns;
6) **Table synonym** mentions for all unique values of categorical columns marked with the `USING SYNONYM` directive;
7) **ID** mentions for all unique values of ID columns.

The index is used to find DB-related mentions in the input question. Note that for categorical and ID-type annotations we also add the distinct values of the respective columns to the index. Note that to prevent bad matches and exploding complexity, too short or too long terms and stopwords are not indexed. In addition, we make sure that no index term from a categorical or ID column masks out a more generic linguistic alias, such as that of a table.

To enrich the human-authored annotation model with more diverse terms, we use WordNet to search for synonymous formulations of specified linguistic aliases. For mention types 1-4, we extend the index with new unspecified synonyms as returned by a WordNet lookup. One can further enrich this by using word embeddings.

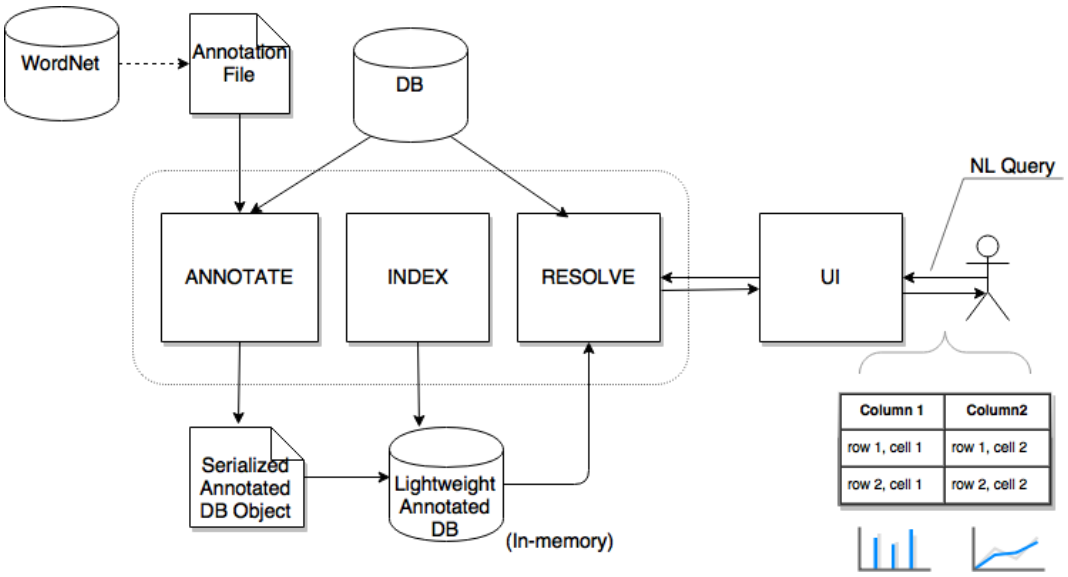

Fig. 2: High-level view of the system architecture

IV. Natural Language Query Analysis

In this section, we describe our question analysis method. First, it discovers all database objects mentioned in the question, it then performs disambiguation, and finally discovers and resolves the filters and aggregation functions.

A. Database Objects Discovery

To find all database objects mentioned in the question, which we will refer to as *matched mentions*, we implement the following three steps:

1) Find all tables and table synonym matched mentions by first generating all n-grams that end with a noun from the question text and then retrieving the longest match from the tables and table synonyms in the inverted index;
2) Discover the remaining matched mention types in a greedy fashion by extracting all n-grams from the question text and retrieving all possible exact matches from the inverted index;
3) Fuzzy matching – find approximate matches in the inverted index.

There are columns in the database that can contain very long values and therefore are difficult to match exactly. To address this, we devised a fuzzy-matching algorithm (described in Algorithm 1) that is able to match incomplete mentions of the values of such DB columns while ignoring the order of the words.

Algorithm 1 Fuzzy Matching Algorithm

Input: Sequence of words (stemmed) corresponding to a given question.

1) Find the longest list of words which are in the inverted index of a column we want to fuzzy match.
2) Intersect the lists corresponding to these words and then store the pairs *(column name, value)* as candidate fuzzy matches for the sequence of words.
3) If the set resulting from the intersection in step (2) is empty, the sequence is repeatedly restricted first forward and then backward until candidate matches are obtained.

Output: All the matched subsequences in the question together with the corresponding list of pairs *(column name, value)*.

B. Database Filters Assignment

The main task here is to recognize all different types of filters and assign them to the correct database entity mentions discovered in the target question. We distinguish four types of filters and handle each of them separately: i) numeric filters, ii) datetime filters, iii) expression filters, and iv) implicit filters.

1. Numeric Filters. Numeric filters can be simple numbers that denote a filter for a certain database column, e.g., for the question "Show me all alerts with priority 3.", 3 is a filter for the column *Alerts*. Numeric filters can also be specified as *quantifier phrase modifiers* (QP) like for example "at least 3" or "not more than 3".

The procedure for associating numeric filters is given by:

- Discovering numeric filters in the question text using a Stanford CoreNLP pipeline [7];
- Associating each filter to the closest database entity mention;
- Converting the numeric filter to its corresponding SQL filter.

An important step in this procedure is the association of a filter to the correct entity mentioned in the question. We address this problem by assigning the filter to the closest entity mention. Thereby, closeness is quantified by first using the distance (in number of words) between the filter and the mentioned entity in the question and then, in case of ties, using the distance between them in the parse tree. We clarify

our approach with examples. When a numeric filter is in the immediate neigborhood of a numeric column mention with no filter assigned, the target filter is associated with the column. However, there are more complex cases. First, note that a numeric filter can also be associated with a table mention and not just with numeric columns. Consider the question "List all servers with more than 3 open tickets" along with its parse tree in Figure 3 and assume we have a table *Servers* and a table *Tickets*. Both table mentions "tickets" and "server" are at the same distance (in number of words) in the sentence from the numeric filter (the QP) "more than 3". In such cases, we take the closest mention with non-empty numeric filter, using the distance from the QP in the parse tree. More specifically, since the QP node "more than 3" in Figure 3 is closer to the NNS_2 node corresponding to "tickets" than to the NNS_1 node corresponding to "servers" it will be assigned to "tickets".

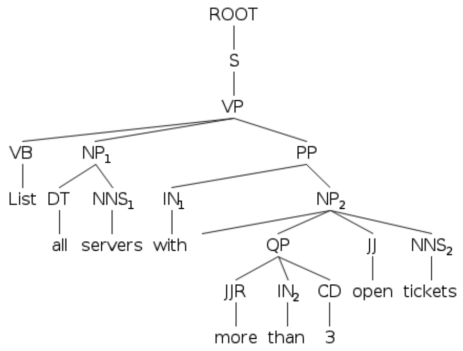

Fig. 3: Parse tree for the question: "List all servers with more than 3 open tickets".

In the third row of Table II, we provide example queries containing numeric filters with their respective SQL translations.

2. Datetime Filters. Temporal filters can be exact dates, such as "December 11, 2017", or temporal expressions such as "next Wednesday at 3pm" that need to be converted to exact dates. After such a conversion their resolution is similar to the one of numeric filters with the difference that they need to be associated to a datetime type of DB columns. We give an example query containing datetime filter and its respective SQL resolution in the second row of Table II

3. Expression Filters. Expression filters refer to DB-specific expressions such as "expired" or "end-of-support" which have individual definitions depending on their semantic in the database. For example, warranty is expired if the current time is smaller than the purchase date plus the warranty duration. Such expression filters are not associated with a specific DB column as the corresponding expression already contains the necessary columns. For example, for the question "Show me all devices which are out of support", the phrase "out of support" represents an expression `CASE WHEN END_OF_MAINTENANCE < DATE(CURRENT DATE) THEN '1' ELSE '0' END` that is specified in the DB annotation.

4. Implicit Filters. Implicit filters denote matches or fuzzy matches of categorical column values. For instance, a column "Warranty" might have values like "limited" or "lifetime". Then, in a question such as "Show me all lifetime devices.", the word "lifetime" is an implicit filter for the column "Warranty".

C. Aggregation Assignment

Questions often specify aggregation functions over columns, sometimes asking for the aggregated value as a result, and other times specifying certain conditions for the aggregated value. The aggregation assignment task consists of recognizing the aggregation functions and assigning them to the correct database entity mentions in the question. To achieve this, we take the following steps:

1) Identify mentions of aggregation functions in the question text;
2) Associate each aggregation function to the closest numeric or datetime DB entity mention;
3) Interpret any existing filter assignment for the matched mentions as a condition for the aggregated value.

Note that in step 1 above we use a fixed dictionary for the most common mentions of the functions that correspond to count, summarization, average, median, minimum or maximum. These mentions are also added to the inverted index.

D. Disambiguation of Overlapping Database Objects

Normally, in a DB schema each table relevant for the users represents a different entity in the target domain. Therefore, we assume that the phrases used to denote each table of interest are different. This is, however, not the case for the columns comprising the tables, the data in the tables or the specified expressions. For example, in the use case in our experiments *end-of-life* can be a phrase that is used to denote both a column and an expression in the same table. Furthermore, two columns in different tables can be referred to using identical phrases. Thus, we devise an approach that uses contextual information to disambiguate any ambiguous, overlapping mentions discovered in the question text. To disambiguate a set of overlapping matched mentions, our approach implements the following rules:

1) Prioritize matched mentions associated with the closest table matched mentions, tables of foreign key matched mentions or tables associated with unambiguous matched mentions (taken in the specified order of importance) discovered in the question text;
2) Prioritize longer matched mentions;
3) Prioritize matched mentions with filters or aggregation assigned.

Table I shows a few simple examples of disambiguation.

In the rare cases where ambiguous matched mentions remain after the disambiguation, they are all kept and if an attempt to generate the optimal SQL query fails, a feedback on the ambiguities is provided to the user, such that he/she can reformulate the question.

Question	Disambiguation
Show all alert descriptions published before 2013.	*published* refers to datetime columns in three different tables, and only the column that appears in the same table as *alert descriptions* is kept
List all devices that become end of life this year.	*end of life* refers to both a datetime column and an expression in the same table, and only the datetime column is kept because of the modifier *this year* assigned to it
Show the end of life date of device X.	*end of life* and *end of life date* are overlapping mentions, and only the *end of life date* column is kept because it is more specific (longer match)

TABLE I: Disambiguation examples.

V. SQL Query Generation

The final step of the NLI pipeline is to render a valid SQL query over the annotated database schema that satisfies the input given the intermediate question model. We now describe our approach that generates the SQL query.

A. Computing the Join Graph

Let the undirected, unweighted graph $G_{DB} = (V, E)$ represent the DB schema, where nodes $v \in V$ are annotated tables and edges $e = (v_i, v_j) \in E$, $v_i, v_j \in V$, are annotated one-to-many foreign key connections among the corresponding tables. The disambiguated set of DB entities yields information about all tables and foreign keys referred to in the input question. Then, let $V_T \subseteq V$ denote the set of mentioned tables and $M_{FK} \subseteq E$ the set of mentioned foreign keys discovered in the target question. The graph then comprises one or several connected components. To formulate a single query that answers the question, all mentioned tables need to be within one of the connected components and joined through a set of valid foreign key connections. If there is no direct foreign key relationship that would connect all of them, the goal is to construct the simplest possible subgraph within the component that comprises all tables mentioned in the question.

Let G_M be the mention graph – a subgraph of the target connected component G with a set of edges M_{FK}, all nodes of V_T and all nodes representing the endpoints of edges in M_{FK} that are not present in V_T. The goal now is to find a tree of minimal size in G that contains all nodes from the mention graph G_M. This is the *minimum Steiner tree problem* [4], a known NP-hard problem. To tackle this problem we apply a minimum spanning tree heuristic similar to the one introduced in BANKS [2]. More specifically, we use a multi-root BFS started from all nodes in G_M in parellel until they meet, not starting to traverse a new BFS level until all other searches have finished the preceding level for all nodes in G_M. Then, the solution is computed by constructing the resulting subtree.

B. Rendering the SQL Output

Analyzing the discovered DB entities, filter specifications, aggregations and the join graph, we obtain an intermediate model of the query that can be converted into standard SQL. To achieve this, we use a generic set of six rules:

1) The join graph contains the information on which **tables** to include in the query through which **foreign key connections**;
2) Mentions of DB columns or expressions over non-aggregated tables with no filter specification give the set of **columns or expressions to select** for the results;
3) The direction of foreign key connections suggests the result cardinality, which specifies where and how to use **nested queries**;
4) Mentions of aggregation functions over a column give the **aggregation level** to use for each queried table, and specify the aggregation itself;
5) For aggregated subqueries, mentions of columns with no filter specification yield the columns for the **group by** of the aggregation;
6) Matched filters for columns, expressions or aggregated values, and direct matches of categorical or ID values specify the **filter conditions** to apply.

The resulting query is rendered following the language grammar of SQL. All queries are formulated as a `SELECT` statement. The `FROM` clause is used for the *subject* table of the query, which is defined as the first directly or indirectly mentioned table in the question. Other tables appear in `JOIN` clauses based on the join graph computed for the actual query. When joining a table with a one-to-many relationship, it is embedded into a nested subquery with all corresponding `WHERE` conditions and aggregations applied, in order to keep the cardinality of the result on the subject table's level. As many-to-one joins do not affect the cardinality of the results they are not embedded into nested queries. Filter conditions are rendered as `WHERE` clauses at the corresponding level of the query, except when they are applied to an aggregated column. Such filter conditions are put into the `HAVING` clause.

VI. Experiments

To evaluate our system, we performed a user study where the participants posed natural language questions and verified the correctness of the answers returned by the system. The main strength of our system is that it can be applied in a setting where existing, related approaches are not applicable, namely a domain-specific setting where the table and column names do not correspond to the underlying entities and there is no large manually curated question-SQL pairs dataset. Thus, the main goal of the experiments is not to provide a comparative study but to demonstrate that our system can successfully be applied and used in practice in the aforementioned setting where existing approaches are not readily applicable.

A. User Study

The ezNL2SQL system was built for a real-world database that contains information such as type, support, vulnerabilities, and firmware for the network devices of many clients. The five participants in the study have different levels of expertise in network management (from basic to advanced) and are

all aware of the type of information that resides in the database. They pose different questions to the system to get the information of interest. Some queries require simple mapping to the entities of a single table to generate their corresponding SQL statement. Others contain ambiguities, require more complex join statements involving multiple tables, non-trivial filter assignments, fuzzy matching or resolving complex temporal expressions. Based on this, we group the queries in our user study into two types, namely easy and difficult. Example questions and generated SQL translations for both types are given in Table II.

Question	SQL Statement Generated	Type
How many routers are there?	`SELECT COUNT(DISTINCT t0_0.device_id) FROM tb_inventory t0_0 t0_0 WHERE (LOWER(t0_0.device_category)='router';`	E
How many of the field notice alerts are published after 2013?	`SELECT COUNT(DISTINCT t1_0.alert_id) FROM tb_fn_alerts t1_0 WHERE t1_0.first_publish_date>'2013-12-31';`	D
Show me switches with more than 3 security vulnerabilities.	`SELECT DISTINCT t0_0.hostname, t0_0.ip, t0_0.serial_num, SUM(t3_0.agg_0) FROM tb_inventory t0_0 INNER JOIN (SELECT DISTINCT c0.device_id, COUNT(DISTINCT c0.alert_id) AS agg_0 FROM tn_alerts c0 GROUP BY c0.device_id HAVING COUNT(DISTINCT c0.alert_id)>3) t3_0 ON t0_0.device_id=t3_0.device_id WHERE (LOWER(t0_0.device_category)='switch') GROUP BY t0_0.hostname, t0_0.ip, t0_0.serial_num;`	D

TABLE II: Example questions from our user study and their respective SQL translations generated by the ezNL2SQL system. In the "Type" column D stands for difficult and E for easy.

B. Results and Discussion

The ezNL2SQL system described in this paper was tested by five participants who posed 164 different questions, out of which 96 were difficult and the remaining 68 were easy. The users verified the system answers as correct for 151 of the questions. The results by question type are summarized in Table III. Analyzing the questions where our system failed to provide the correct answer, we observe that they either involve complex compositional aggregation or complex negation. While our system supports simple negation types (such as *excluding*, *not including*, *non-...*) that are considered as filters and assigned to the corresponding database entities, more complex negation types that require a deeper understanding of the question text semantics and addressing complex compositional aggregation are left for future work.

The response time of the query resolution is between 100 and 250 milliseconds both for easy and difficult questions, tested on an index containing more than 2 million terms, using an Intel[1] Core i5 2.60 GHz CPU laptop with 16 GB of RAM.

[1]Intel is a trademark of Intel Corporation in the U.S. and/or other countries.

Question Type	Correct Answers
Easy	95.38%
Difficult	88.37%

TABLE III: Percentage of correctly answered questions for the question types in the user study.

VII. Deployment

Our NLI system has been deployed as a RESTful Web Service, and it offers both a Web GUI and a CLI for interaction. It authenticates users and limits their accessibility based on their corresponding access levels. It serves questions on a database comprising the information pertaining to the networking devices of many different clients. It can be used by both the clients and the technical support teams for the networking equipment.

Our system processes NL questions reliably and fast, as shown by our experiments, and retrieves the requested data from the database. The user interface enables the user to post questions of interest and, dependent on the size and type of the data retrieved from the DB, to get the answer in the most suitable format. The Web GUI supports various data visualizations ranging from numbers and tables to different types of charts, where the user can also easily switch from the vizualization automatically chosen by the system to another one. Last but not least, the system provides feedback on all different mentions discovered in the target question along with information on their corresponding types. In this way, the user gets the information on how the system interprets the question. Moreover, the user can provide feedback in case that a natural language term has not been recognized by the system, which contributes to the enrichment of the natural language aliases for the corresponding DB entity.

VIII. Related Work

The motivation for our work originates from PRECISE [9, 10]. It relies on the assumption that the names for tables and columns in the DB are NL names of their corresponding entities and uses a graph-matching algorithm to find a mapping between tokens and the database entities – subject to a set of constraints. Our system, however, introduces a database annotation language that can be used for every database and allows a more flexible matching using a combination of different rule-based and fuzzy-matching approaches. There are approaches like [8, 14, 3] that assume a semantic graph describing the database is available, or that a domain-specific onthology is available [11], or use a high-quality grammar [5] to translate NL queries to SQL. Moreover, the authors in [6, 5] develop systems that rely on user feedback. Finally, there are also learning-based approaches for devising natural language interfaces to databases, such as the ones presented in [12, 15, 1, 13]. One disadvantage of such models is, however, that they require a large labelled corpus of question-query pairs for each database and achieve accuracies insufficient for their application in practice (around 60%). Note that existing work is not readily applicable in the setting where DB column and

table names do not correspond to their underlying entities and a large labelled corpus is not available.

IX. Conclusions and Future Work

In this paper, we introduce a novel approach that aims to assist and enhancing the management of network devices by automatically translates a natural language question into an SQL query that can be directly run against an existing database. The approach has been developed and tested in a challenging domain-specific setting that often occurs in practice: where DB column and table names do not correspond to the underlying entities and a large labelled corpus is not available. While it is hard to apply existing work in this setting, our ezNL2SQL features an easy-to-use DB annotation language that enables its application to an arbitrary database. We demonstrate the usefulness of our approach in a user study with network management experts and a complex network security inventory database, where it achieves 90% accuracy in NL question to SQL translation.

A future enhancement of the system is to integrate support for follow-up questions that ask for further details regarding previously asked (complete, non-follow-up) questions. This can be achieved by using a two-step approach. The first step needs to correctly identify the follow-up questions given a previously asked complete question. As follow-up questions are incomplete, in the second step, a new question that reflects the information requested in the follow-up question needs to be constructed using the information from the previous complete and the target follow-up question. This newly constructed question can then be answered using our ezNL2SQL system. We believe such support for follow-up questions adds value to our system by contributing to a better user experience.

The ezNL2SQL has been developed for English; however our system can easily be ported to other languages such as French, Italian, or Spanish, especially as these languages are already supported by the Stanford CoreNLP pipeline [7].

X. Acknowledgements

The authors would like to gratefully acknowledge Robert Labori for supporting this research, and Biliana Bojilova, Slavcho Mitov, Maxim Mavrudiev, Zhivko Dimov and Kaloyana Vasileva-Mitova for the system evaluation and their valuable feedback and discussions during the system development.

References

[1] Fuat Basik, Benjamin Hättasch, Amir Ilkhechi, Arif Usta, Shekar Ramaswamy, Prasetya Utama, Nathaniel Weir, Carsten Binnig, and Ugur Cetintemel. DBPal: A learned NL-interface for databases. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. ACM, 2018.

[2] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabati, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proceedings of the 18th International Conference on Data Engineering*, pages 431–440. IEEE, 2002.

[3] Catalina Hallett, Donia Scott, and Richard Power. Composing questions through conceptual authoring. *Computational Linguistics*, 33(1):105–133, 2007.

[4] FK Hwang and Dana S Richards. Steiner tree problems. *Networks*, 22(1):55–89, 1992.

[5] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. Learning a neural semantic parser from user feedback. In *ACL*, 2017.

[6] Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.*, 8(1):73–84, September 2014.

[7] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[8] Frank Meng and Wesley W. Chu. Database query formation from natural language using semantic modeling and statistical keyword meaning disambiguation. Technical report, 1999.

[9] Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *Proceedings of the 20th International Conference on Computational Linguistics*, COLING '04, Stroudsburg, PA, USA, 2004. Association for Computational Linguistics.

[10] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 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, 2003. ACM.

[11] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar F. Minhas, Ashish R. Mittal, and Fathma Özcan. ATHENA: an ontology-driven system for natural language querying over relational data stores. In *Proceedings of the VLDB Endowment*, 2016.

[12] L. R. Tang and R. J. Mooney. Using multiple clause constructors in inductive logic programming for semantic parsing. In *Proceedings of the 12th European Conference on Machine Learning*, pages 466–477, Freiburg, Germany, 2001.

[13] Semih Yavuz, Izzeddin Gur, Yu Su, and Xifeng Yan. What It Takes to Achieve 100In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[14] Guogen Zhang, Wesley W Chu, Frank Meng, and Gladys Kong. Query formulation from high-level concepts for relational databases. In *Proceedings User Interfaces to Data Intensive Systems, 1999*, pages 64–74. IEEE, 1999.

[15] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv*, 2017.