NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Translating Natural Language to SQL using Deep Learning

Georgios Gr. Katsogiannis-Meimarakis

Supervisors: **Georgia Koutrika,** Research Director, ATHENA Research Center
Ioannis Ioannidis, Professor, NKUA

ATHENS

JUNE 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μετάφραση Φυσικής Γλώσσας σε SQL με Βαθιά Μάθηση

Γεώργιος Γρ. Κατσογιάννης-Μεϊμαράκης

Επιβλέποντες: **Γεωργία Κούτρικα,** Διευθύντρια Ερευνών, Ερευνητικό Κέντρο ΑΘΗΝΑ
Ιωάννης Ιωαννίδης, Καθηγητής, ΕΚΠΑ

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2020

BSc THESIS

Translating Natural Language to SQL using Deep Learning

Georgios Gr. Katsogiannis-Meimarakis

S.N.: 1115201400065

SUPERVISORS: **Georgia Koutrika,** Research Director, ATHENA Research Center
Ioannis Ioannidis, Professor, NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μετάφραση Φυσικής Γλώσσας σε SQL με Βαθιά Μάθηση

Γεώργιος Γρ. Κατσογιάννης-Μεϊμαράκης

Α.Μ.: 1115201400065

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γεωργία Κούτρικα,** Διευθύντρια Ερευνών, Ερευνητικό Κέντρο ΑΘΗΝΑ
Ιωάννης Ιωαννίδης, Καθηγητής, ΕΚΠΑ

ABSTRACT

Databases contain a vast amount of data, used to support a range of operations, from business operations, scientific experiments to activities in our everyday lives. However they are still inaccessible for non-technical users, without knowledge of Structured Query Language (SQL). Natural language interfaces to databases lift these obstacles for such users and they have recently bloomed. In this thesis, we will start by presenting the NL2SQL problem (translating Natural Language to Structured Query Language), its most important aspects and the anatomy of a NL2SQL system. We will compare some systems and see how each one of them chooses to tackle the problem. In the main part of this work, we will focus on the SQLNet system which uses deep learning methods to tackle the NL2SQL problem. We will also test our own implementation of the system, investigate possible improvements and test how well it works on various cases.

SUBJECT AREA: Deep Learning, Databases

KEYWORDS: Artificial Intelligence, Neural Networks, Machine Learning, Natural Language Processing, Natural Language Interfaces

ΠΕΡΙΛΗΨΗ

Οι βάσεις δεδομένων περιέχουν τεράστια ποσότητα δεδομένων, τα οποία χρησιμοποιούνται για την υποστήριξη ενός μεγάλου εύρους δραστηριοτήτων από επιχειρηματικές δραστηριότητες, επιστημονικά πειράματα μέχρι δραστηριότητες της καθημερινότητας μας. Παρά όλα αυτά παραμένουν μη προσβάσιμες για έναν χρήστη χωρίς γνώση Γλώσσας Δομημένων Ερωτημάτων (SQL). Οι διεπαφές φυσικής γλώσσας για βάσεις δεδομένων καταρίπτουν αυτά τα εμπόδια και τελευταία βρίσκονται σε άνοδο. Στα πλαίσια αυτής της πτυχιακής εργασίας, θα ξεκινήσουμε παρουσιάζοντας το πρόβλημα NL2SQL (μετάφραση φυσικής γλώσσας σε γλώσσα δομημένων ερωτημάτων), τα πιο σημαντικά του σημεία και την ανατομία ενός συστήματος NL2SQL. Θα συγκρίνουμε κάποια συστήματα και θα δούμε πως το καθένα από αυτά έχει επιλέξει να αντιμετωπίσει το πρόβλημα. Στο κύριο μέρος της εργασίας, θα εστιάσουμε στο SQLNet, ένα σύστημα το οποίο χρησιμοποιεί τεχνικές βαθιάς μάθησης για να αντιμετωπίσει το πρόβλημα NL2SQL. Επίσης, θα δοκιμάσουμε τη δική μας υλοποίηση του συστήματος, θα προσπαθήσουμε να εφαρμόσουμε κάποιες βελτιώσεις και θα ελέγξουμε πόσο καλά λειτουργεί σε διάφορες περιπτώσεις.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Βαθιά Μάθηση, Βάσεις Δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Τεχνητή Νοημοσύνη, Νευρωνικά Δίκτυα, Μηχανική Μάθηση, Επεξεργασία Φυσικής Γλώσσας, Διεπαφές Φυσικής Γλώσσας

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Georgia Koutrika, for her invaluable help and support during this very demanding thesis as well as her patience and encouragement when things weren't going as planned.

This work was supported by computational time granted from the National Infrastructures for Research and Technology S.A. (GRNET S.A.) in the National HPC facility - ARIS - under project ID gkatsog-nl2sql. The computational time was used for the training of all models presented in this thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This bachelor thesis was completed from September 2018 until June 2020 in Athens under the supervision of research director Georgia Koutrika of the Athena Research Center, who I would like to thank for her help and mentoring.

Θα ήθελα επίσης να πω ένα μεγάλο ευχαριστώ στην οικογένειά μου και στους φίλους μου, χωρίς τους οποίους δεν θα ήμουν αυτός που είμαι σήμερα και δεν θα είχα καταφέρει όσα έχω καταφέρει. Επίσης, ένα μεγάλο ευχαριστώ στου Άφαντους για όσα μου έχουν δώσει και συνεχίζουν να μου δίνουν.

1. INTRODUCTION

The ever-increasing demand for data-driven approaches in decision making, planning and business in general, has made the need for non-technical staff to use relational databases all the more relevant. As a result, the quest for a simple and meaningful way to access databases that does not require the knowledge of SQL is on the rise and has received a lot of attention from the scientific community.

If we consider today's wide-spread and mainstream use of web search engines and personal assistants, it becomes apparent how valuable it would be to create a system that provides access to a relational database in a similar way. Such a system would allow any person with elementary computer knowledge to access a relational database and retrieve answers and data by posing questions using natural language.

In the words of Ted Codd (1974): *"If we are to satisfy the needs of casual users of databases, we must break through the barriers that presently prevent these users from freely employing their native languages."* [3]

This thesis examines the problem of translating Natural Language (NL) to Structured Query Language (SQL) and investigates the efficiency of deep neural networks in this task. More specifically, in this chapter we will start by introducing the before-mentioned NL2SQL problem and we will explain its main aspects. We will then propose an abstract workflow/architecture, that covers most NL2SQL systems, in order to obtain a better idea of how most solutions approach the problem.

In chapter 2, we will go through the steps of the workflow and see how various NL2SQL systems chose to implement each step. This will help us get a better understanding of the ideas, methods and techniques that the scientific community has used to tackle the problem. We will also take a brief look at the databases/datasets available for testing NL2SQL systems. Among these systems, we will also examine SQLNet [18], a system based on deep learning (DL), on which we will focus for the remainder of this thesis.

Chapter 3 contains an overview and statistics of WikiSQL, a data set for training and testing NL2SQL systems, created by the authors of Seq2SQL [20]. SQLNet was also built to perform on WikiSQL, so it is better to understand the task first, before the system.

In chapter 4, we will go deeper into the SQLNet system, study its architecture and see exactly how it is built and how it works. An implementation of SQLNet has also been developed by me, for the means of this thesis, in order to better understand the system, but most importantly so that we can experiment on it. The implementation is based on the original paper as well as the authors' implementation which is available online[1].

Finally, in chapter 5, after making sure that this new implementation achieves the same scores as described in the SQLNet paper, we will be investigating ways to improve its performance. Based on the observation that a lot of words from the WikiSQL are "unknown" to the system, we will be trying to decrease the number of unknown words by trying different tokenization methods and by applying spell-checking. We will also be taking the system one step further, by testing it on a full relational database.

[1] https://github.com/xiaojunxu/SQLNet

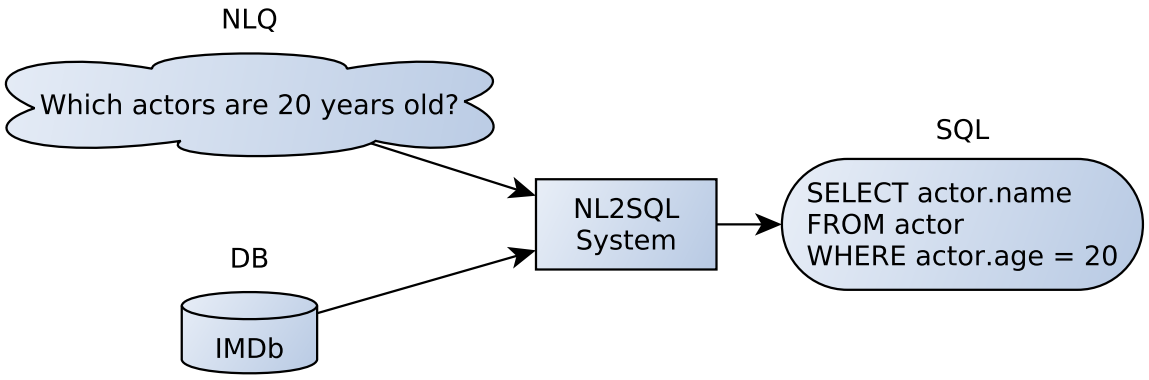

Figure 1.1: NL2SQL Example

1.1 The NL2SQL Problem

The NL2SQL problem can be simply described as follows:

Given a Natural Language Query (NLQ) on a Relational Database (RDB) with a specific schema, produce a SQL query equivalent to the NLQ which is valid for the said RDB.

An example of the NL2SQL problem can be seen in Figure 1.1.

Even though it can be defined this simply, we must not underestimate the complexity and the difficulty of the task. First and foremost any problem that involves NL is burdened by the complexity NL carries: it can be hard to understand, some words or phrases can have multiple interpretations, etc. Another notable point is that SQL can achieve great levels of complexity, so a complete solution must be able to produce many different and complex queries and understand the level of complexity that the user's NLQ requires with respect to the RDB's schema. The solution must also be independent of a single database or domain (e.g. movies, sports, etc.) and transferable to a new database with little to no adjustments and ideally without the need of human labour. Another aspect to keep in mind is that we are designing a system that is addressed to users with little or no knowledge of SQL and relational databases. This burdens our system in two ways: on one hand it might not be able to answer all NLQs, on the other hand it must be very cautious about the results it presents to the user. Now let us look deeper into these aspects.

Natural Language Complexity

Understanding NL remains an open problem for the scientific community to this day. NL can often be difficult to understand, even for a human, due to its flexibility and the absence of strict rules. Even in a conversation between two people, it is normal to come across misunderstandings, so a NL2SQL system must also be prepared to handle such cases. Let us now look into some causes of NL complexity.

References can confuse the receiver of the question, because the part of the sentence to which a word is referring is not always clear. For example, in the NLQ "Coaches of football teams *who* have won at least 3 Champions League titles" what is the word *who* referring to? It is clear we are looking for coaches, but who must have won the titles? The team or the coach? In real life, the receiver would ask for a clarification because this is not something he can resolve by himself. A NL2SQL system would have to do something

similar if it wanted to solve such misunderstandings. It would have to be able to identify ambiguous situations and offer an interface for the user to clarify what he is asking for.

Inferences are also very common in NL because humans assume that whoever is listening has some fundamental knowledge about the world and can infer some things without needing to hear them explicitly. Take for example the NLQ ”List all presidents after Barack Obama”. It might appear very simple to people who have some basic knowledge of U.S. politics, but for a computer it can be very confusing. First of all, it is implied that Barack Obama was himself a president at some point. Furthermore, it is never said that we are referring to presidents of the U.S. A NL2SQL system must have some kind of *knowledge base* from which it can understand that Barack Obama was a U.S. president, in order to handle this NLQ correctly.

Synonyms are another cause for misunderstandings. The user might choose a word in his NLQ that is stored in the database using a synonym. For example, the user asks ”Who is the *composer* of the Ode to Joy?” but the database has an entity named *songwriter*. In such cases a NL2SQL must be able to understand to what the user is referring, even though the exact word is not present in the database. We can also refer to this problem as a *vocabulary gap*, between the user’s vocabulary and the system’s or the database’s vocabulary.

Universality of Solution

Another important aspect is that the solution must work with every given database, no matter the schema nor the domain, with little to no fine tuning. If this fine tuning process is required, ideally, it must be performed automatically or by a user with limited technical knowledge.

For example, a NL2SQL system must be able to work with any type of domain (i.e. financial data, weather data, geospatial data, etc.) and shouldn’t expect to be taught new essential vocabulary. It should also be able to adapt to new databases that contain similar data but have very different schemas, causing the required SQL queries to be very different.

It is also worth saying that even though we are trying to enable people without any programming knowledge access DBs, we are at the same time assuming that they have adequate knowledge of the English language. Even though it might seem as a secondary goal, making a NL2SQL system that can handle any language can be proven a very difficult task [1] that should not be disregarded.

User Mistakes

The human factor is an aspect of the problem that must also be taken into account. Humans are prone to making mistakes that the system should be able to identify and handle.

Spelling mistakes and typos are a very common example. The system must employ spell-checking and correcting mechanisms, in order to be able to identify and correct such mistakes.

Syntactical and Grammatical mistakes are more challenging than spelling mistakes because they are both harder to identify and solve. Additionally, the way this kind of mistakes are handled is very dependant on the NLP techniques the system uses to process the NLQ. For example, if a system does not perform syntactic parsing, it most likely will

not identify any syntactical errors.

System Restrictions

Due to the complexity of the problem, it is possible to encounter instances in which a specific system is not able to produce a correct SQL query. This can be due to a either **NL2SQL system restriction** or a **database restriction**. In either case, the system should be able to identify the incapability to produce the expected result and inform the user accordingly.

In the first case, a system might not be designed to satisfy some edge use-case scenarios, because of the complexity and difficulty to do so. In the second case, the database might not be designed to answer a specific question asked by the user. There might be instances where the user might try to ask questions that cannot be answered, simply because the database lacks the data to do so. Foe example, it is not possible to tell which is the best selling phone if the database was not designed to keep track of sales.

Result Validation

Another problem that arises is that of validating whether the output the user gets is actually what he wanted. In contrast to using a search engine, our result is not a page with information and descriptions that he can read and criticise. A NL2SQL system gives SQL queries as its output or maybe the resulting data from executing that SQL query. Since most of the times the user will not be able to write and read SQL there is no way for him to be sure that the resulting SQL query represents his NLQ correctly.

In some cases, if the answer is blatantly different from what the user expects, it might be easy to recognise a mistake. For example if the user expects a date as a result, he would be unsettled by receiving a negative number or a name. However, in cases where the answer's type, value and range seems to match the user's expectations, how can he validate that this is what he was looking for?

1.2 NL2SQL System Workflow

Now that we have a grasp of the problem we are trying to solve, it would also be useful to have an idea of the structure of a system that tackles the problem. Even though many different approaches have been proposed, there are some key similarities between them. Based on these similarities we will now present an abstract workflow/architecture that can fit all of them, in order to better understand them. This architecture includes the most important components of a NL2SQL system and the workflow indicates the procedure followed from getting the user's NLQ up to returning a SQL query to them. This workflow can be seen in Figure 1.2.

Everything starts from the user giving a **NLQ** to the system through the **User Interface** (UI). The NLQ is then passed to the **Natural Language Processor** which transforms it into an **Intermediate Query Represenation** (IQR). The IQR is a representation or data structure created from the original NLQ that is easier to use and translate into SQL for the system. The **Query Interpreter** receives this representation and transforms it into a **SQL** query. Finally, the SQL is presented to the user through the **User Interface**.

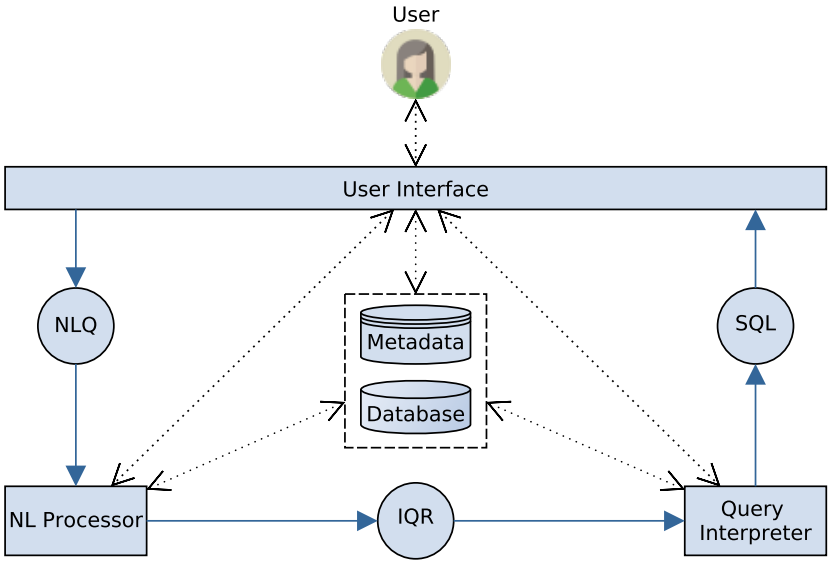

Figure 1.2: NL2SQL Workflow: Blue solid arrows represent the flow of the query and black dotted arrows represent interactions between components

The **Metadata** is another component that is present in the architecture but does not always have the same position in the workflow. It can interact with the UI, the NL Processor or the Query Interpreter and help them perform their tasks depending on the design choices of the system's creators. The same goes for the User Interface which can involve the user in the processes of the NL Processor or the Query Interpreter.

Let us now have a more in-depth look of each component.

User Inteface

The User Interface is the component that connects the user with the system. Its main task is to request input (NLQ) from the user and present the output of the system (SQL) back to the user. It is possible however to add a lot more functionality to the UI in almost every step of the workflow.

The UI can start helping even before the user has entered the NLQ by providing features such as auto-completion, suggestions, spell-checking, etc. It can also help the NL Processor and the Query Interpreter by asking the user to clarify any ambiguities they might encounter. Let us recall our previous example: "Coaches of football teams *who* have won at least 3 Champions League titles". In this case the UI could ask the user to clarify if the coach or the team must have won 3 titles. Lastly, when finaly presenting the output to the user, the UI can offer some features that help the user check if the result is what he actually wanted. For example it could provide an explanation in NL of the resulting SQL query, a visual representation, etc.

Metadata

The metadata used by the system is a very central component since it is useful in all steps of the workflow. When using the term metadata we are referring to all the data and information that the system can use to perform its task. One essential piece of metadata,

for example, is the schema of the database that the NLQ is addressed to. Without it, it would be impossible to construct a valid SQL query. There are, however, even more possibilities for going beyond that.

One possible design choice is to keep some general knowledge stored in the metadata, where the system can look up words and phrases it does not understand. This can happen by using a lexicon or word embeddings, for example. Some systems might create word indices from the database's contents to check if a word from the NLQ exists in the database and even relate that word to a specific entity. Another idea is to store templates of SQL queries in the metadata. This way the Query Interpreter can have some starting point when composing a SQL query instead of starting from scratch.

In any case, the metadata used by the system depends on the specific solution.

Natural Language Processor

When a user types a NLQ the system receives a simple string of text, which by itself is not very helpful since the operations that can be performed on it are very limited. The NL Processor is the first thing the NLQ meets when it is given to a NL2SQL system and is responsible for converting this string into a structure that is more practical and easy to handle for the system. The output of this component is the IQR, a representation of the initial string that carries a lot more info about how the system understands the NLQ and is easier for the system to transform into a SQL query.

This part usually contains techniques such as text tokenization, various forms of NL parsing such as syntactic or dependency parsing, word embeddings and other NLP methods. Some systems choose to include the user by asking him to clarify any ambiguities. It is also common to take advantage of the metadata, most often by using a knowledge base including lexicons to better understand the user's input, or even use the database's content (e.g. to check if a word from the NLQ appears in the DB).

Intermediate Query Representation

The Intermediate Query Representation is the output of the NL processing step. It is a representation of the NLQ which is ready to be given to the system's Query Interpreter for further processing.

These representations can range between many different data structures and the choice for each system largely depends on the system's algorithms. Some common choices are parse trees, program sketches, ontology languages and even sequences of words. Some systems might produce more than just one representation, if they are unsure which is the best-fitting for the NLQ. At a later point in the workflow they might decide to discard some or they might even present all of them to the user.

It should also be noted that in some systems the NL processing and the query interpretation steps are so close to each other that it might not be relevant to talk about an IQR.

Query Interpreter

The Query Interpreter receives the IQR, i.e. the result of NL processing on the NLQ and tries to interpret a SQL query based on it. To achieve this goal, the query interpreter

may have mechanisms to evaluate the probability of an interpretation matching the user's intent, interpretation generators, mechanisms that locate mistakes in interpretations and fix them, mechanisms that try to find the best way to use the words of the NLQ, neural networks and more.

As in the NL processor, the user's feedback along with the metadata can be requested to aid the interpretation. In this step, some common types of metadata used is the schema of the database and the data stored in it.

2. BACKGROUND AND RELATED WORK

In this chapter we will examine some NL2SQL systems and compare how they chose to implement the parts of the workflow described in the previous chapter. Specifically, we will be seeing the following systems: **SQLizer** [19], **NaLIR** [8], **ATHENA** [16], **Analyza** [4], **Seq2SQL** [20] and of course **SQLNet** [18].

Note that not all of these systems are strictly NL2SQL systems; some of them have slightly different goals and/or purposes. Specifically, Analyza is built for *"Exploring data with conversation"* and goes a bit beyond the NL2SQL task by proposing some very interesting user interaction capabilities. Also, SQLNet and Seq2SQL handle a simpler version of the problem called WikiSQL [20], where the NLQ of the user is directed to a single table rather than a complete database. However both systems present interesting ideas that could be applied to the general NL2SQL problem as well.

At the end of this chapter we will also discuss how a NL2SQL system can be evaluated, as well as the available data sets for evaluating and training a NL2SQL system.

2.1 NL2SQL Systems Comparison

In the following sections we will examine each part of the workflow and compare the different solutions proposed by each system. Namely, we will be seeing how each system implements the **Natural Language Processor**, **Intermediate Query Representation**, **Query Interpreter**, **User Interface** and **Metadata**. Tables 2.1 and 2.2 offer an overview of all system and all the parts of the workflow.

2.1.1 Natural Language Processor

NL Parsers

The most common way of processing NL is with a natural language parser. A NL parser is a system that converts an utterance given by the user into a data structure that represents the grammatical, syntactic, or other kind of structure of that utterance. The output of the parser can therefore be easier to handle for a computer and represents the meaning of the NLQ better than a string of text.

NL parsers are usually trained on hand-parsed sentences, because it is much easier and efficient to let a model learn how parses are created from examples, rather than try to design a set of rules (e.g. a Grammar) or a deterministic algorithm that can be applied to

Table 2.1: System Components Comparison

System	NL Processor	IQR	Query Interpreter
SQLizer	Semantic Parser	Query Sketch	Sketch Completion & Refinement
NaLIR	Dependency Parser	Parse/Query Tree	Parse Tree Mapper & Adjustor
ATHENA	”NLQ Engine”	Interpretation Tree	Ontology-driven Interpretations Generator
Analyza	Semantic Parser	Semantic Parse	Confidence Score & User Disambiguation
SQLNet	Tokenization	Word Embeddings	Neural Networks
Seq2SQL	Tokenization	Word Embeddings	Neural Networks

Table 2.2: System Components Comparison

System	User Interface	Metadata
SQLizer	No	Schema & Stats
NaLIR	Disambiguation	Schema & Data Index
ATHENA	Secondary options	Ontology & Data Index
Analyza	Disambiguation, Suggestions, Secondary Options	Schema & Knowledge Base
SQLNet	No	Column Names & Embeddings
Seq2SQL	No	Column Names & Embeddings

all possible inputs [9]. The use of neural networks is also possible and quite promising [5]. This also means that parsers are not perfect, even though they can achieve high accuracy.

It is common to use preexisting solutions or frameworks when implementing a NL2SQL system rather than building a new one from scratch. Specifically, SQLizer uses a custom semantic parser while taking advantage of the Stanford CoreNLP's pre-trained models. Analyza also uses a preexisting framework [9] based on the same ideas. NaLIR uses the Stanford Dependency Parser [12], a well known and highly used dependency parser.

ATHENA's "NLQ Engine"

ATHENA implements a *NLQ Engine*, as its authors call it. This is a complex component of three parts, that when given an NLQ as input, returns an Ontology Language Query. Briefly, these parts are:

1. The *Ontology Evidence Annotator*, which creates "Evidence Sets" and "Relationship Constraints" based on the user's NLQ
2. The *Ontology-driven Interpretations Generator*, which creates a number of "Interpretation Trees", which basically are the system's interpretations of the query. To determine which of these interpretations to keep, ATHENA ranks them based on the likelihood of being correct, with respect to the NLQ
3. The *Ontology Query Builder*, which transforms the highest ranking interpretation into an Ontology Language Query.

Essentially only the first part corresponds to what we have chosen to refer to as a "NL Processor" and produces an IQR (Evidence Sets & Relationship Constraints) for the next two components to interpret. The second and third parts function more like a "Query Interpreter". So in this section we will describe the first part and leave the other two for the Query Interpreter section.

The way ATHENA processes the NLQ through its Ontology Evidence Annotator, is by trying to understand to what each word of the NLQ is referring to, as well as the possible dependencies between them. To achieve this, it checks each word individually, trying to relate them to an ontology element (while also checking its synonyms with a dictionary) or to value stored in the database by looking them up in a *Translation Index*.

The output of this process is an *Evidence Set* and a set of *Relationship Constraints* which describe the candidate mappings of each word and the relationships between them.

Based on these two structures, the Interpretations Generator will try to generate the best interpretation which will be represented as an Interpretation Tree.

Text Tokenization

Seq2SQL and SQLNet present a different and simpler approach to handling NL, simply by tokenizing the NLQ into a sequence of words (tokens) and using word embeddings to represent them as vectors.

Text tokenization is the process of transforming a string of text into a sequence of separate words. When we receive the NLQ as a string from the user there is not much we can do with it, but extracting every word of the sentence allows us to use each word separately (e.g. with word embeddings). Creating a text tokenizer is relatively simple, there are however some edge cases that require careful handling. For this reason it is better to use one of the already available frameworks.

The output of the tokenization process is a sequence of words. Each word is then assigned a word embedding vector, so the final output of the NL processor is a sequence of word embeddings. Word embeddings will be further discussed in section 2.1.2.

2.1.2 Intermediate Query Representation

The intermediate representation of the query is a choice with multiple possibilities and variations even between systems that process the NLQ similarly or even in the same way. For example, both SQLizer and Analyza use semantic parsing to process the NLQ but their IQRs are very different; SQLizer uses query sketches whereas Analyza uses a data structure called a semantic parse. Let us now look into some IQR choices.

Query Sketches

SQLizer uses query sketches to represent the query. A sketch is the outline of an incomplete SQL query, with ”holes” (e.g. missing the content of the WHERE clause) that need to be completed for it to become valid. Sketches can have hints for each ”hole” that give some direction as to how it needs to be filled.

Parse/Query Trees

Analyza uses a data structure called a *semantic parse* which contains detailed information such as metrics, dimensions and terms left unused from the NLQ.

NaLIR, which uses a similar dependency parser, uses a parse tree which displays the dependencies between the words/phrases of the NLQ and builds on it by adding, removing and enriching its nodes (with the user’s guidance) until it can be transformed into a valid SQL query.

Word Embeddings

Word Embeddings is a technique for mapping a set of words to a set of numeric vectors. This enables the transformation of text into numbers, which is very helpful when using neural networks, since they can only accept numbers as input. There are multiple methods of creating this mapping but both Seq2SQL and SQLNet use the Global Vector (**GloVe**) [14] model. This particular model is trained on a big corpus of text (e.g. several pages of Wikipedia) and the vectors that are created reflect the probability of two words appearing in the same context (e.g. the same page). This means that similar words have vectors with small distance between them and the vectors of unrelated words have a larger distance. Using this type of embeddings not only allows us to transform text into vectors, but the vectors carry knowledge about the words they represent as well.

2.1.3 Query Interpreter

Interpretation Generation & Ranking

Although the techniques used by ATHENA, SQLizer, NaLIR and Analyza might seem completely different, their similarity becomes evident when looking at the bigger picture. Their data structures and algorithms might differ but all these systems follow a philosophy of generating interpretations, testing them and ranking them based on their validity, their similarity to the NLQ and other more specific criteria.

ATHENA's *Ontology-driven Interpretations Generator* picks one evidence for each word of the NLQ from the evidence set and tries to find the correct connections between them, thus producing possible interpretations of the NLQ. As these interpretations are produced, they are also ranked based on the likelihood of correctly capturing the users intent and the highest ranked one is finally presented to the user.

SQLizer's methodology has two steps, a *Type-directed Sketch Completion* followed by *Sketch Refinement using Repair*. First, based on the sketch generated by the parser, it tries to find expressions that complete it which are valid with respect to the schema of the database and assigns a confidence score to each one of them. The confidence score is an estimation of the probability that the way the sketch is completed matches the user's intent and is calculated based on the NL hints given by the parser, the names of the schema elements, the foreign and primary keys and the database's contents. After generating said completions, the system searches for possible faults in the completions and tries to repair them. This is again done by finding expressions that don't pass our confidence score threshold even though all their strict sub-expressions are above the threshold and applying some predefined repair techniques.

NaLIR's take on the matter uses a *Parse Tree Node Mapper* followed by a *Parse Tree Structure Adjustor*. The first generates mappings to SQL components of every node of the parse tree produced by the parser based on a similarity function. Each mapping exceeding a predefined threshold is considered a candidate mapping for the node. Then, the latter, generates more trees by creating all the possible adjustments to the parse tree so that it becomes syntactically valid and ranks them based on the probability of being correct. In both steps, despite the system's ranking, the final choice of which tree will be kept is made by the user and the ranking only serves in choosing which options to present to the user.

Neural Networks

Seq2SQL and SQLNet, employ a very different technique compared to all the other systems. They use neural networks and deep learning techniques in order to produce a SQL query that is most probable to match the user's input.

For example one neural network could be responsible for predicting the probability of which aggregation function between SUM, MAX, etc. should be used. Another network, when given the sequence of words from the user's input, could predict the sequence of words in the WHERE clause.

We will be seeing this case more in-depth in chapter 4, where we will be examining the SQLNet system.

2.1.4 User Interface

On the subject of User Interface capabilities and how to include the user in the process there are many different approaches and some interesting ideas. At one end of the spectrum, SQLNet, Seq2SQL and SQLizer are examples of systems not having any interaction at all with the user, besides asking for a NLQ and presenting a SQL query. It should be noted that this happens mostly because of the complexity of the first two systems which both use deep neural networks. In theses cases it is clearly much more challenging to include the user in the process. It could also be attributed to the simplicity of the WikiSQL problem, since there aren't any complex schemas or multiple tables to choose between.

ATHENA goes a small step further, not by requiring any further interaction by the user, but by accompanying the SQL query that it believes is the best interpretation of the NLQ, with some other high-ranking interpretations as alternatives, in case the user was looking for something different. This however might not be very effective if the user can't read SQL.

NaLIR goes way beyond, by not only including the user in the transformation, but by essentially relying on the user to clear any ambiguities both when processing the NL and when interpreting the query until the desired result is achieved. It could be argued that this kind of excessive involvement might render the system unfriendly to the user and decrease productivity by making the whole process too slow and tiring.

Analyza is a very good example on how the system can interact with the user. Some very interesting features it presents are the following:

- Suggesting queries that we think might be useful based on what the database contains or what the user has searched so far
- Auto-completing user's queries as they are being written so that the are well suited to our database's schema and data
- Providing alternative choices in case our highest-ranking prediction is not what the user is looking for
- Translating our output queries back to NL, in order to explain our interpretation to the user and try to validate the results
- Giving the user a comprehensible overview or a visualisation of the data in order to eliminate any confusion on what can be accomplished by our database and minimise user mistakes

2.1.5 Metadata

The amount and types of data kept by each system can range from the bare essentials (e.g. the schema of the database), up to some very sophisticated architectures that include knowledge that could even be unrelated to the current database in use.

On one side, SQLNet and Seq2SQL have the most simple approach. They only use the table column names of the table on which the NLQ is directed and rely on the knowledge of their word embeddings, described in section 2.1.2.

SQLizer also has a relatively simple approach by using the database's schema and various statistics about its contents to make sure it is forming valid queries and to assess the likelihood of the query being the one it is looking for. For example, a query with a WHERE clause of the form *WHERE x == c* will get a low likelihood if the value *c* doesn't exist in the database. NaLIR also takes advantage of the data in a similar way by using a *Data Index*, i.e. a data structure that helps the system check if a value appears in its contents.

ATHENA also uses an interesting data index for looking up keywords that also supports *variant generators* that helps group similar names that refer to the same real-world entity (e.g. "George Katsogiannis", "G.Katsogiannis" and "Giorgos Katsogiannis" refer to the same person and must be handled in the same way). What distinguishes it from other systems is that it does not rely on the schema of the data base but works with ontologies and mapping functions relating them to the DB that need to be provided by the designer of the DB. This adds an extra level of complexity to making the system adaptable to different databases but at the same time provides physical independence from the DB.

Analyza has an extended metadata store and goes way beyond than simply using the schema of the database. It enriches it by adding a lot of extra information describing the tables and the data by a curation process that requires a human curator and with the aid of a knowledge graph. Additionally, it keeps track of every interaction with the user and tries to learn phrases and words that are classified as unknown when first encountered as well as identify requests for data that is not available in the database and respond accordingly.

2.2 Evaluation of Systems

Another very important question that must be considered is how can we decide which system is better at this complex task. Based on the problem and its challenges, it becomes clear that evaluating a NL2SQL system comes down to the following key aspects:

- The **effectiveness and accuracy** of the system at synthesising valid SQL queries that correspond to the user's intentions
- How much **time** it takes to synthesise such a query
- Its ability to work equally well on **different databases** and how much aid it requires by the user to do so

Depending on the structure of each system and more importantly the level of interaction needed by the user, there are different ways of evaluating its quality. Systems that depend on the user's feedback, like NaLIR, should be evaluated by testing the speed and ease with which an inexperienced user manages to form an accurate query. For systems such

as SQLizer, where no user feedback is needed, evaluating their performance can be as simple as calculating the percentage of correct translations on a certain data set.

Because of these functional differences it is very difficult to have a precise comparison between all the systems we presented. Even though this is inevitable due to the problem's nature and the various levels of user inclusion, a common axis that would enable as to clearly and consistently evaluate NL2SQL systems would be beneficial to the efforts of tackling the problem.

Another problem in comparing NL2SQL systems is the lack of available databases as well as NLQs with their corresponding SQL queries, to test the systems on. This poses a very big barrier to testing the performance of systems, especially when it comes to testing their adaptability to new databases.

2.3 Available Data Sets/Databases

The lack of data sets, as mentioned before, has pushed researchers to use already-available databases, often adjusting them by hand to better suit their needs, while simultaneously generating NLQs by themselves or in some cases by crowd-sourcing them. In this section we will provide a brief overview and comparison of the data sets used by the systems presented. It should be noted that just as the aforementioned systems have some fundamental differences between them, so do the data sets.

Microsoft Academic Search (MAS)

The MAS data set was developed by the authors of NaLIR [8] based on Microsoft's Academic database[1]. The full database is not made completely available by Microsoft but can be accessed in some depth by its API. The data set created for [8] contains 196 questions and has been made public by its authors. The schema of the database consists of 17 tables and 53 collumns and takes up 3.2 GB of space. It has also been used for SQLizer [19] and ATHENA [16].

Internet Movie Database (IMDb)

The IMDb data set used by SQLizer [19] is based on the online movie database IMDb. It contains 128 questions and has been made available by its authors. The schema of the database consists of 16 tables and 65 collumns and takes up 2 GB of space.

Yelp data set

The Yelp data set used by SQLizer [19] is based on the the business review website Yelp. It contains 128 questions and has been made available by its authors.

The schema of the database consists of 16 tables and 65 columns and takes up 2 GB of space.

[1] https://academic.microsoft.com

WikiSQL data set

A crowd-sourced data set was developed for the WikiSQL task which was proposed in [20]. It contains NLQs and their equivalent SQL queries, each based on a single table taken from Wikipedia. The data set is available for download on Github[2]. It contains 87726 question pairs based on 26375 table schemas.

Besides not being a full relational database, another difference to all the other data sets mentioned is that it is not based on one specific domain, instead every table contains data from different domains (e.g. sports, geography, etc). We will take a closer look on this data set in chapter 3.

FIN data set

The FIN data set contains financial data and was created by IBM and used for the ATHENA system[16]. It consists of 108 question that were produced by IBM employees. Given that ATHENA works on ontologies rather than RDBs, the data set is described by an ontology that contains 75 concepts, 289 properties and 95 relations but it is also stored on a RDB with a normalized schema.

GeoQuery data set

The GeoQuery workload contains geographical data and has been used to evaluate the ATHENA system [16] among others. It was first used in [17] and was made public by its authors. It consists of 250 questions over 800 facts implemented in Prolog.

Other Available data sets

Besides the data sets used by the systems we have discussed, there are still some worth mentioning.

WikiTableQuestions is a data set very similar to WikiSQL, which features questions on single tables on a variety of domains, similarly to WikiSQL. Their main difference is that WikiTableQuestions doesn't expect SQL queries as an output to its questions but rather a simple answer. It was developed for [13] and has been made available by its authors.

The **SCHOLAR data set** was released with [6] and contains 816 NLQ accompanied with SQL, over an academic relational database.

[2]https://github.com/MetaMind/wikisql

3. THE WIKISQL TASK & DATASET

Before getting into further detail about the SQLNet system, it is necessary to first explain the task which it was made to deal with.

The WikiSQL task and dataset was proposed by the authors of Seq2SQL [20]. It consists of a large number of tables taken from various pages of Wikipedia. The data provided by its creators include the tables' contents and the names of each table's columns. In some cases some additional information is provided, such as the name of the Wikipedia page from which the table is taken or the caption of the table. It also contains NLQs and their equivalent SQL queries, each based on a single table. This is a very important difference to most available data sets, since it doesn't use complete databases but single tables for each query. This means that there is no underlying schema to take into account and that the SQL queries that it contains are not that complex (e.g. there are no joins in any query).

Besides not being a full relational database, another difference is that it is not based on one specific domain, because the tables were taken from Wikipedia. Instead, every table contains data from different domains (e.g. sports, geography, train tracks from eastern Europe, etc). An example of a data sample can be seen in Figure 3.1.

It was created by crowd-sourcing the questions and the queries. For this reason a lot of its queries might be badly written, have mistakes or even be completely wrong. In other occasions, the tables might have been copied incorrectly, leading to incomprehensible NL and SQL queries.

The data set is available for download on Github and contains 80,654 question pairs based on 26,531 tables. The Github page also features a performance "leaderboard", which presents the best performances achieved on the WikiSQL task by various NL2SQL systems from the scientific community.

3.1 The WikiSQL Subsets

The data set is separated into 3 parts by its creators to ensure that all models that use it as a benchmark are using the same questions for the same purposes. This ensures that the leaderboard of the accuracies achieved by various systems is fair. However, it can be used as a whole or with different splits if one does not intend to submit his results in the leaderboard.

Country	Players	Standard	Minor	First title	Last title
England	35	119	28	1977	2013
Scotland	7	71	7	1987	2013
Wales	6	29	3	1974	2013
⋮	⋮	⋮	⋮	⋮	⋮

NLQ: "Which country has more than 5 players?"

Ground Truth SQL Query: SELECT Country WHERE Players > 5

Figure 3.1: A WikiSQL data sample consists of the NLQ and the table on which the question is asked, along with the equivalent SQL query.

Table 3.1: WikiSQL Sub-Sets

	Questions	Tables
Train Set	56,355	18,585
Validation Set	8,421	2,716
Test Set	15,878	5,230
Total	80,654	26,531

The subsets are the following:

1. Train set: To be used to train the model.
2. Validation set: To be used to check the model's performance during training and possibly affect the model's training, e.g. check for over-fitting and stop training when it is observed.
3. Test set: To be used only for testing and comparing performance of different models, i.e. the test set does not affect the model during its training in any way.

Each subset has its own unique tables so the same query or table is never seen by the model both in training and testing.

3.2 SQL Query Complexity

Because all questions are targeted to single tables, the resulting SQL queries are not that complex. This is evident even by the way the SQL queries are stored in the data set. Each SQL query is comprised of:

1. A **SELECT** on one of the table's columns
2. One or no aggregation function performed on that column, which will be one of {*None*, MAX, MIN, AVG, SUM, COUNT}
3. A series of conditions (or no conditions) each of which is comprised of:
   (a) The column of the table on which the condition is applied
   (b) The operation that is applied, which will either be =, > or <
   (c) The value against which the condition is applied

So essentially if we predict each of these parts we have predicted the SQL query. This is something to keep in mind because it is the idea behind the way SQLNet works.

3.3 Bad Questions

As mentioned before, there are some questions and tables in the WikiSQL data set that make little or no sense at all. Such questions usually have no knowledge to offer to our systems and may even create problems, since they force the systems to learn something wrong or incomprehensible.

Table 3.2: WikiSQL Table Incorrectly Copied from Wikipedia

! Late 1941	Late 1942	Sept. 1943	Late 1943	Late 1944	1978 Veteran membership
Bosnia and Herzegovina	20000	60000	89000	108000	100000
Croatia	7000	48000	78000	122000	150000
Kosovo	5000	6000	6000	7000	20000
Macedonia	1000	2000	10000	7000	66000
Montenegro	22000	6000	10000	24000	30000
Serbia (proper)	23000	8000	13000	22000	204000
Slovenia	2000	4000	6000	34000	38000
Vojvodina	1000	1000	3000	5000	40000

Table 3.3: Actual Wikipedia Table (Yugoslavian Partisan Army Composition by Region)

	Late 1941	Late 1942	Sept. 1943	Late 1943	Late 1944
Bosnia and Herzegovina	20000	60000	89000	108000	100000
Croatia	7000	48000	78000	122000	150000
Kosovo	5000	6000	6000	7000	20000
Macedonia	1000	2000	10000	7000	66000
Montenegro	22000	6000	10000	24000	30000
Serbia (proper)	23000	8000	13000	22000	204000
Slovenia	2000	4000	6000	34000	38000
Vojvodina	1000	1000	3000	5000	40000
Total	81,000	135,000	215,000	329,000	648,000

Table 3.4: WikiSQL Questions on the Incorrect Table

NLQ:	Name the most late 1943 with late 194 in slovenia
SQL:	**SELECT** max(late 1943) **WHERE** ! late 1941 = slovenia
NLQ:	What is the least september 1943 when late 1943 is 78000
SQL:	**SELECT** min(sept. 1943) **WHERE** late 1943 = 78000
NLQ:	What is the late 1941 when 1978 veteran membership is 20000
SQL:	**SELECT** ! late 1941 **WHERE** 1978 veteran membership = 20000

Take for example Table 3.2, which is clearly a product of some error. As it turns out this table was copied incorrectly from Wikipedia, using a wrong header for the first column and as a result it has been rendered unusable. The correct table, seen in Table 3.3, can be found on a Wikipedia and refers to the Yugoslav Partisans of WWII.

Besides this table being incomprehensible, the WikiSQL dataset actually contains questions posed on this table and even gives the expected SQL queries that should be produced from these questions. These questions can be seen in Table 3.4. Note how these NLQs are both incomprehensible and have spelling mistakes but still are accompanied by some questionable SQL queries that will probably confuse any NL2SQL system that is trained on them.

3.4 WikiSQL Statistics

Finally, we computed a set of statistics on the data set, to better understand its characteristics.

In Figure 3.2 we see how many times each aggregation function is found in the queries. Here we observe that the overwhelming majority of SQL queries don't use an aggregation function. However if a function is used, the number that of appearances of each function

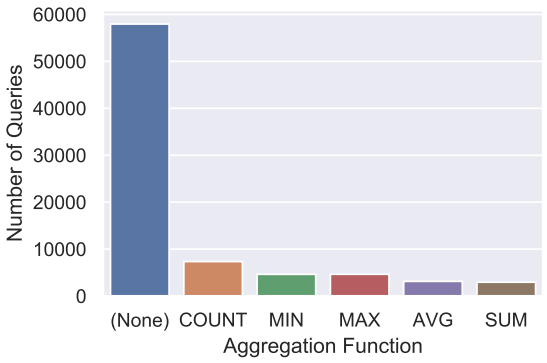

Figure 3.2: Aggregation functions in WikiSQL's queries

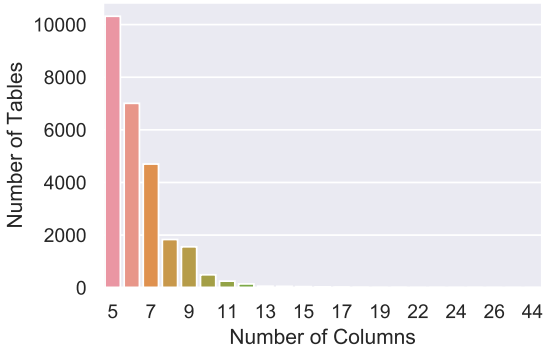

Figure 3.3: Number of columns in WikiSQL's tables

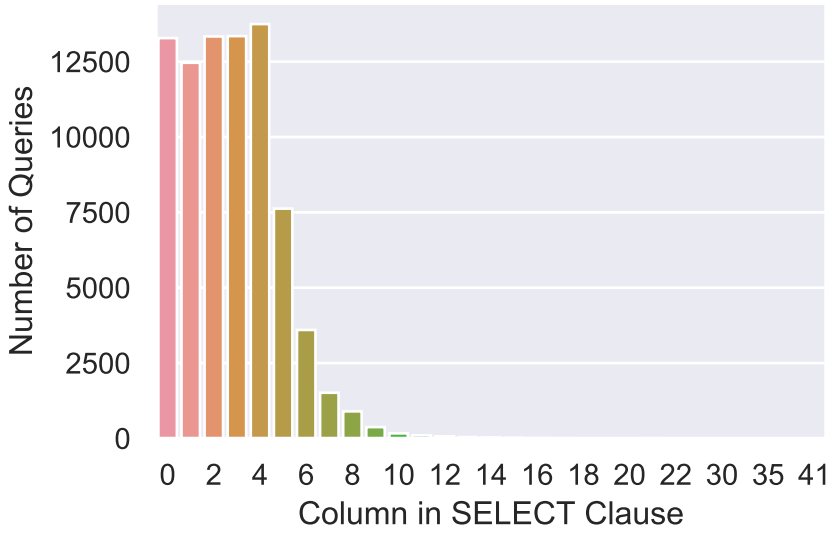

Figure 3.4: Count of columns in SELECT clause in WikiSQL's queries

are quite close.

In Figure 3.3 we can see that most tables have between 5 to 7 columns. Hence, we are dealing with tables with a small number of columns. Hence, this is another indication that WikiSQL is not a very complex dataset for evaluaitng NL2SQL systems.

Figure 3.4 shows the position of the columns that appear in the SELECT clause of the data set's queries (e.g. we see that the 5th column is most often selected). Considering that most tables have between 5 to 7 columns, we observe that table columns are uniformly selected, which is a good thing.

Figure 3.5 shows the number of conditions in the WHERE clause of the queries and Figure 3.6 shows how many times each operation appears in the conditions. Here we observe that the data set is not that balanced since most queries contain 1 condition and most conditions have the operation "=". It would be better for a model to learn on a more balanced data set.

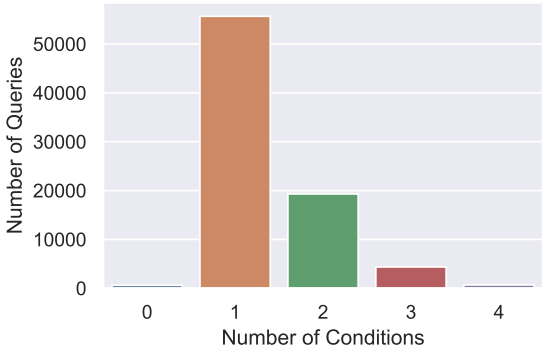

Figure 3.5: Number of conditions appearing in WikiSQL's queries

Figure 3.6: Number of operations appearing in WikiSQL's queries' conditions

3.5 Summary

To conclude, let us collect some key facts about the WikiSQL data set and outline its advantages and drawbacks.

- Crowd-sourced NLQs and SQL queries based on **single** tables copied from Wikipedia
- Published with the Seq2SQL paper [20]
- Available on Github
- A lot of systems have been created based on it by the scientific community

Advantages

- Very large number of NLQs along with the equivalent SQL queries
- Very large number of different tables
- Many different topics between tables

Drawbacks

- Some tables have been incorrectly copied
- Some NLQs are incomprehensible
- For some SQL queries it is very difficult to understand why they correspond to their equivalent NLQ
- Relatively simple SQL queries (no FROM, JOIN, GROUP BY, SORT, etc.)
- It does not appear to be very well balanced, especially when it comes to the conditions of the SQL queries

4. SQLNET

SQLNet [18] is a NL2SQL system that was developed to tackle the WikiSQL task [20], described in the previous chapter.

For the purposes of this thesis, an implementation of SQLNet was developed based on the published paper and the authors' implementation. This implementation was developed using the Tensorflow and Keras deep learning libraries for the Python programming language. The original implementation was created using the PyTorch deep learning library for Python and is available on GitHub[1]. The two implementations have very minor differences, mainly due to the different functionality of the libraries used. Some improvements were applied after implementing the original system, but these will be discussed in chapter 5.

This chapter will provide a description and analysis of SQLNet and its components along with remarks and observations made while implementing it. We will start by presenting the *SQL Sketch*, the idea on which SQLNet is based on. Next we will go through a generalized dataflow that will give us an idea of the processes and steps that take place in SQLNet from the moment that a NLQ is given from the user until the point where a prediction is produced. After we have a general understanding of this process, we will examine every neural component of the system separately and examine similarities and differences between them.

4.1 SQL Sketch

The most important aspect of SQLNet is that it is based on a "SQL Sketch". This sketch is essentially a grammar that can produce all possible SQL queries found in WikiSQL, as simple as it might seem. The sketch can be seen in Figure 4.1. This builds on the fact we mentioned in section 3.2, i.e. there are only a few specific sub-parts, that have to be predicted in order to create the desired SQL query.

Effectively, it means that every query is produced by filling the gaps of this specific sketch. Every query has a column to be selected, an aggregation function ((None), MAX, MIN, COUNT, etc.) to be applied to the results and zero or more conditions. Each condition can be specified by a column, an operation ($=, >, <$) and a value. As we will see in the following chapters, we are going to have a predictor component (a neural network) for each of these specific tasks.

The use of this specific sketch also means that the capabilities of SQLNet are restricted. It cannot predict any queries that don't follow this grammar. For example, queries with JOINs, GROUP BYs, etc are out of this system's reach.

[1] https://github.com/xiaojunxu/SQLNet

SELECT <*AGG*> <*COLUMN*>
(**WHERE** <*COLUMN*> <*OP*> <*VALUE*>
(**AND** <*COLUMN*> <*OP*> <*VALUE*>)$*$)?

Figure 4.1: SQLNet's Query Sketch

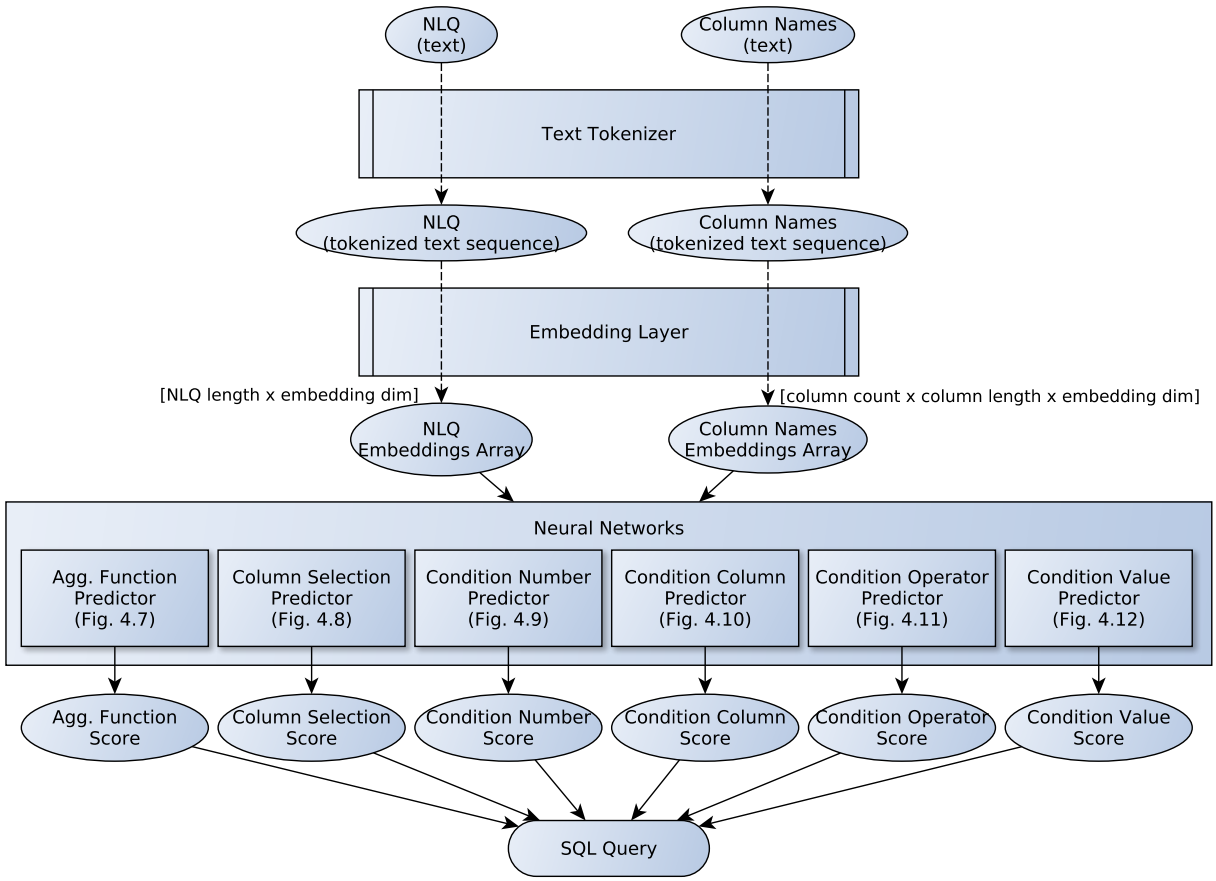

Figure 4.2: Dataflow in SQLNet

4.2 Dataflow in SQLNet

In this section we will give a brief explanation of how data flows in the SQLNet system; how everything starts from the user's input, how it is processed and how we reach to the final output.

Everything starts by giving the system a NLQ and a table on which it is being asked. Before being fed to the networks, the data is first tokenized and then transformed into numerical values, in the form of word embeddings. The arrays are then given to the networks and each one of them makes a prediction for a specific part of the SQL query. Finally, by joining the most probable predictions we can construct the entire predicted SQL query. An overview of the dataflow can be seen in Figure 4.2.

Input Data & Preprocessing

The user's NLQ is taken as is, but the table on which the question is addressed is not taken in its entirety. From the table, we choose to only keep the column names and discard the data of its contents. So the input used by SQLNet consists of two parts:

1. The user's NLQ
2. The column names of the table on which the question is being asked

NLQ:
→ ”What’s the smallest number of players?”
Tokenized NLQ Sequence:
→ [<BEG>, ”what”, ”’s”, ”the”, ”smallest”, ”number”, ”of”, ”players”, ”?”, <END>]
NLQ Embeddings Array:
→ An array of shape: $nlq\_length \times embeddings\_dimension$

Column Names:
→ [”Country”, ”Players”, ”Standard”, ”Minor”, ”First title”, ”Last title”]
Tokenized Column Names Sequence:
→ [[”country”], [”players”], [”standard”], [”minor”], [”first”, ”title”], [”last”, ”title”]]
Column Embeddings Array:
→ An array of shape: $column\_count \times max\_column\_length \times embeddings\_dimension$

Figure 4.3: An input preprocessing example

So both our inputs are text and more specifically the NLQ is one single string of text and the column names are multiple strings in a specific order. However, before any data can be fed into the model it needs to be processed and most importantly it must be turned into numbers. It is necessary to remember that even though our data is in text format, neural networks only deal with numbers.

To achieve this transformation first we will use two NLP techniques: text tokenization and word embeddings. We will now explain these processes with more detail. An example of the entire preprocessing procedure can be seen in figure 4.3, for NLQ and table column inputs.

Text Tokenization

The first step is the tokenization of the query and the column names. This means separating each string we have into sequences of single words. This is a relatively simple task and there are many tokenizers available that can perform this task. SQLNet uses the Stanford CoreNLP tokenizer [11].

Something that needs to be noted here is that by tokenizing a string that contains a NLQ, we are creating a sequence of words. So our input is actually a sequence and not just a set of words, because the order of the words in a sentence is important. This is something very important to consider when choosing the right neural network architecture to treat our data. Equivalently, by tokenizing a series of strings, each of which is the name of a column, we are creating a series of series of words. So our second input, is a double nested series. The order of the columns is important as well because it will later help us distinguish them.

Embedding Layer

The embedding layer transforms each word that it is given to a numeric vector, so that the vector can then be fed to the neural network.

There are many embedding techniques for turning words into vectors, however SQLNet

uses the Global Vectors (GloVe) [14] method. In fact, the creators of GloVe have released a pre-trained GloVe embedding set, which we will be using. This set has been trained on 42 billion tokens and has a vocabulary size of 1.9 million words and a corresponding vector for each word with a dimension of 300. It is available on Stanford's website[2] under the name *"Common Crawl (42B tokens, 1.9M vocab, ...)"*.

So the job of our embedding layer will be relatively simple. Since all the vectors are already trained, it only needs to assign each word to the corresponding vector according to the pre-trained set we have. If we encounter a word which is not included in the pre-trained set of vectors, we assign a zero vector to it.

After the embedding array has been created, the preprocessing is completed and we can feed this array to each component's neural network.

Working with Batches of Data

Another aspect we must keep in mind is that training a neural network, or in our case multiple networks, is a costly procedure that takes time and processing power. To reduce this time, it is a common technique to feed the network multiple data samples at once. Such a group of samples is called a "batch" of data.

When working in batches the shape of our arrays changes by adding an extra dimension. So for example the NLQ embedding array will be of shape:

$$batch\_size \times max\_nlq\_length \times embeddings\_dimension$$

and the table column embedding array will be of shape:

$$batch\_size \times max\_column\_count \times max\_column\_length \times embeddings\_dimension$$

Note that a batch can have NLQs of many different lengths but an array can't have rows of different lengths. To make up for this we use the max length in the batch as the array's dimension and add "dummy" values in vectors that are smaller. This technique is called *padding*.

Neural Networks

After having processed the inputs into embedding matrices, they can be passed to the neural query translation components. SQLNet has 6 separate neural networks, each one of them receives the same input and is responsible for completing a different part of the SQL query. These neural networks are the main and most complicated part of the system. Each network will be explained in section 4.3.

Output

Finally, each of SQLNet's neural networks produces a score output, which is its prediction on the part of the sketch it was designed to complete. We will see every component and its output in the following sections.

[2] https://nlp.stanford.edu/projects/glove/

4.3 Neural Query Translation Components

SQLNet can be divided into 6 separate neural query translation components that produce 6 different outputs that complete different parts of the SQL query. Each component is an independent neural network responsible for a specific task. More precisely these components are responsible for predicting the following parts of the query sketch:

1. Aggregation Function *(Subsection 4.3.2)*
2. Column Selection *(Subsection 4.3.3)*
3. Number of Conditions (Subsection *4.3.4)*
4. Column in each Condition *(Subsection 4.3.5)*
5. Operation in each Condition *(Subsection 4.3.6)*
6. Value String in each Condition *(Subsection 4.3.7)*

4.3.1 Common Procedures

Before presenting every neural network separately, it would be preferable to explain some procedures that occur in almost all of them.

Encoding the Embeddings

The first thing that happens in each component's network when receiving the embedding array is an encoding process. To calculate the encodings E_Q, E_{col} of the NLQ and the column names, we feed each embedding array into two layers of bidirectional LSTMs. This procedure can be seen in Figure 4.4.

Notice that because the column embeddings have an extra dimension, we choose to keep only the last time step for each sample in the *column length* dimension. This transformation ensures that both inputs have the same number of dimensions.

Each component has separate weights for its LSTMs for the NLQ and the table columns and the the weights are not shared between components.

Column Attention

After having calculated the encodings, most of the components employ a technique named Column Attention. The creators of SQLNet demonstrate that this technique produces a noticeable increase in accuracy. It is possible to create all the networks we will display later without column attention, however since its use is clearly beneficial, we will be using it in all our networks.

Column attention is a technique introduced in SQLNet, that tries to give more meaning to the words of the NLQ, or to the encodings produced from the NLQ to be precise, by adding information from the encodings produced from the table columns encodings. In most cases there are some words in the NLQ that are more relevant in predicting a part of

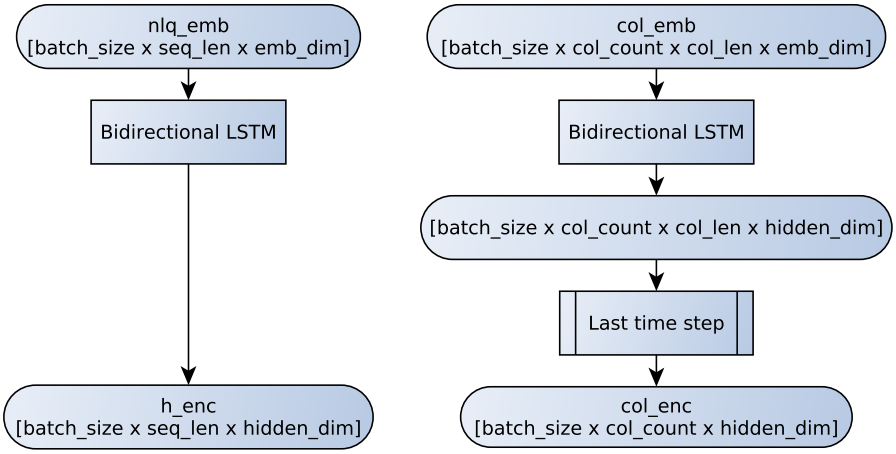

Figure 4.4: Producing encodings from embeddings

the SQL. For example, in the example of Figure 4.3, the word ”players” is very important, because it indicates we will select the column ”players” of the table. What column attention does is give more emphasis to key words in the NLQ by taking the column names into consideration.

So instead of calculating E_Q for NLQ we calculate $E_{Q|col}$ using the column encodings as follows:

$$E_{Q|col} = E_Q w$$

Where w are the weights of the column attention and are calculated as:

$$w = softmax(v)$$
$$v_i = (E_{col})^T W E_Q^i \quad \forall i \in \{1, \ldots, L\}$$

Where L is the length of the NLQ and W is a trainable matrix.

An overview of the column attention mechanism can be seen in figure 4.5.

Combining Outputs

Since we have two separate inputs (i.e. NLQ and column names) , but a single output for each neural query translation component, they have to be combined in order to produce a result. This happens at the end of each network after processing each input. The process consists of adding the outputs of each process, applying the tanh activation function on the result and feeding it to a linear layer. Finally, a softmax activation function is applied to produce the final prediction probabilities. An overview of this procedure can be seen in Figure 4.6 and can also be described as:

$$P(col|Q) = softmax(u_a^T tanh(u_c^T E_{col} + u_q^T E_Q))$$

Where E_{col}, E_Q are the encodings of the column names and the NLQ and u_c, u_q, u_a are the trainable matrices of the linear layers.

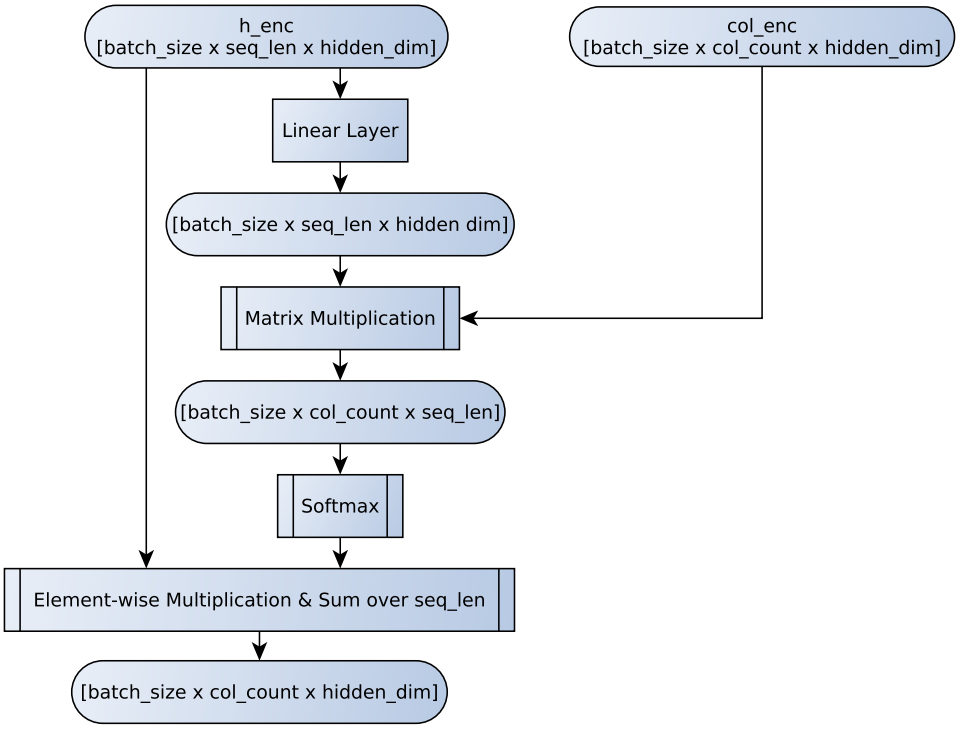

Figure 4.5: Column Attention

Note that this is only the general idea; some components have a slightly different behavior. For example the Condition Column Predictor, uses the sigmoid activation function and the condition string predictor uses the ReLU function instead of the softmax function. The Aggregation function and the condition number predictors don't add the column encondings to the final result so in their case the same procedure happens using only E_Q.

4.3.2 Aggregation Function Predictor

The aggregation function predictor predicts the probabilities of each of the 6 possible functions to be applied, for each column of the table. This means, for example that different probabilities are calculated for applying the MAX function on the first and second columns.

The output it produces has a shape of $batch\_size \times column\_count \times 6$, since there are only 6 possible functions to choose between. The possible outcomes are: *None*, COUNT, MIN, MAX, AVG and SUM. It is possible however to re-train this network with any number of functions we choose, but this would obviously require a data set that uses these extra functions.

An overview of its architecture can be seen in Figure 4.7.

4.3.3 Column Selection Predictor

The column selection predictor predicts which column of the table will be in the SELECT slot of the sketch. An overview of its structure can be seen in Figure 4.8.

The output of the column selection predictor is a $batch\_size \times max\_column\_length$ array that shows the predicted probability of each column of the table to be selected in the SQL query.

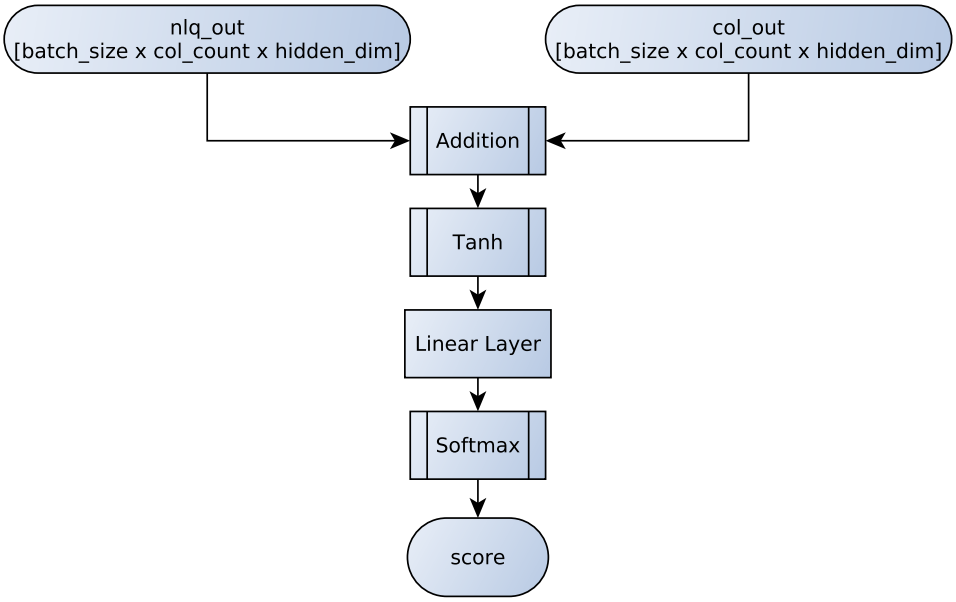

Figure 4.6: Combining Outputs

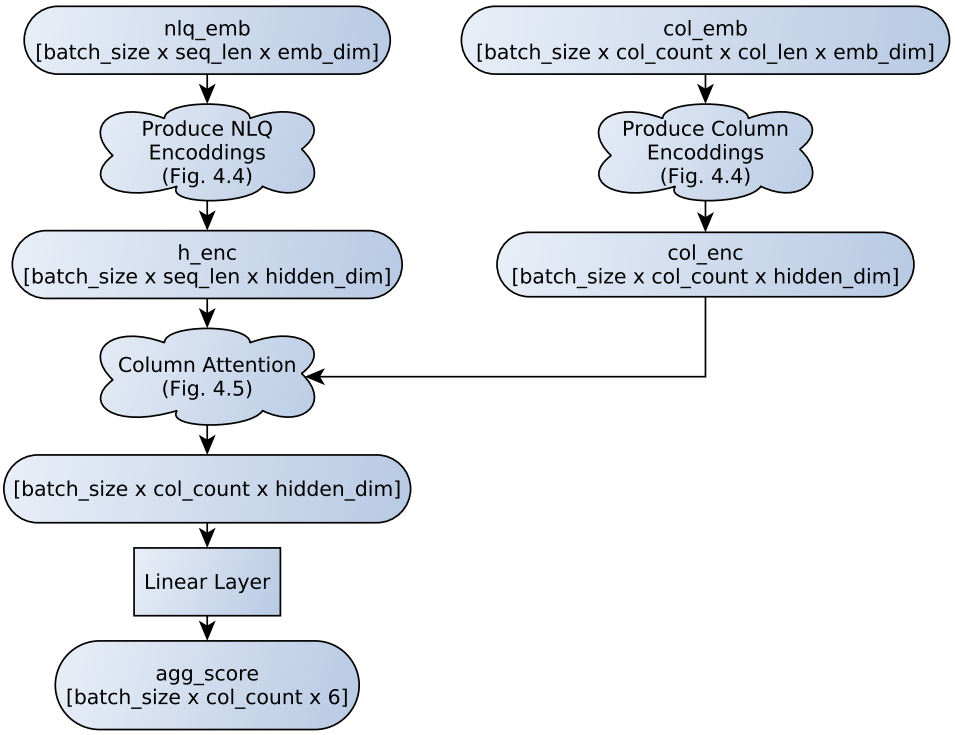

Figure 4.7: Aggregation Function Predictor with Column Attention

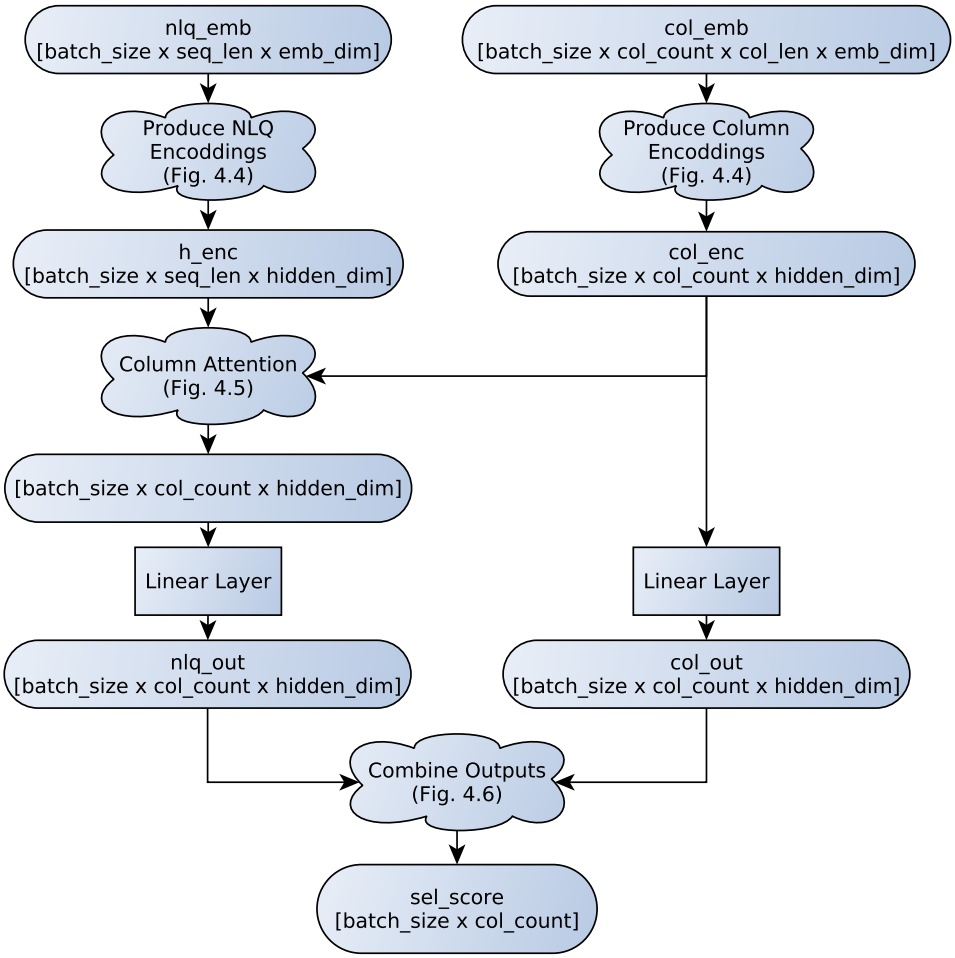

Figure 4.8: Column Selection Predictor with Column Attention

4.3.4 Condition Number Predictor

The condition number predictor is tasked with predicting how many conditions each query will have. Since no query in the WikiSQL data set has more than 4 queries, the creators of SQLNet decided to keep the prediction range between 0 and 4. This however is something that can be easily changed.

The output array has a shape of $batch\_size \times 5$, which represents the probabilities for each query to have 0, 1, 2, 3 or 4 queries.

The architecture of this component can be seen in Figure 4.9. Notice how this component has only one branch, i.e. only one input. It does not use the column names, but predicts the number of conditions base solely on the user's NLQ.

4.3.5 Condition Column Predictor

The condition column predictor predicts the probability for each column of the table to be present in the conditions of the query. The output array's size is $batch\_size \times col\_count$.

The way it works is that the condition number predictor makes a prediction on the number k of conditions that will be present in the query and the k columns with the highest probabilities are used.

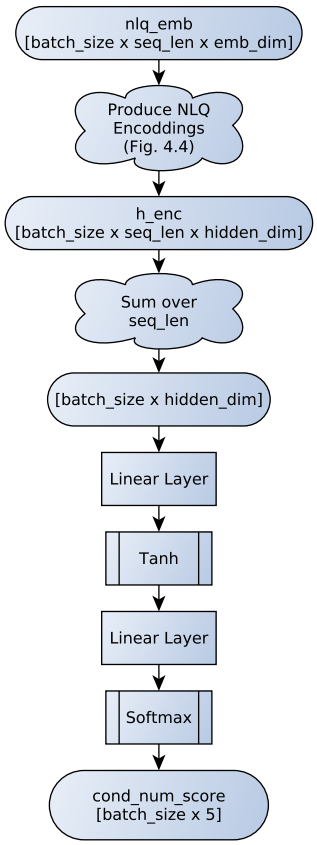

Figure 4.9: Condition Number Predictor

An overview of this component can be seen in Figure 4.10. Notice how it is nearly identical to the column selection predictor.

4.3.6 Condition Operation Predictor

This component predicts which operation would be applied to each column of the table if it were to be included in the conditions of the query. The output array's size is $batch\_size \times col\_count \times 3$, so we get the probability of each operation ($=, >, <$), for each column.

An overview of the component is presented in Figure 4.11. Again, notice how this component is nearly identical to the condition column predictor and the column selection predictor as well.

4.3.7 Condition Value Predictor

The condition value predictor is a Sequence to Sequence (seq2seq) network that predicts the value of the condition (i.e. the value slot in Figure 4.1) for each column. This prediction is made for all columns, no matter if they will be in fact chosen to appear in the conditions. The value is always a part of the user's NLQ, so its task is to select a sequence of words from the NLQ. There are some differences compared to the previous networks that we will explain in this section. The most important aspects to keep in mind are that the condition value predictor:

1. Predicts a sequence of $cond\_len$ length

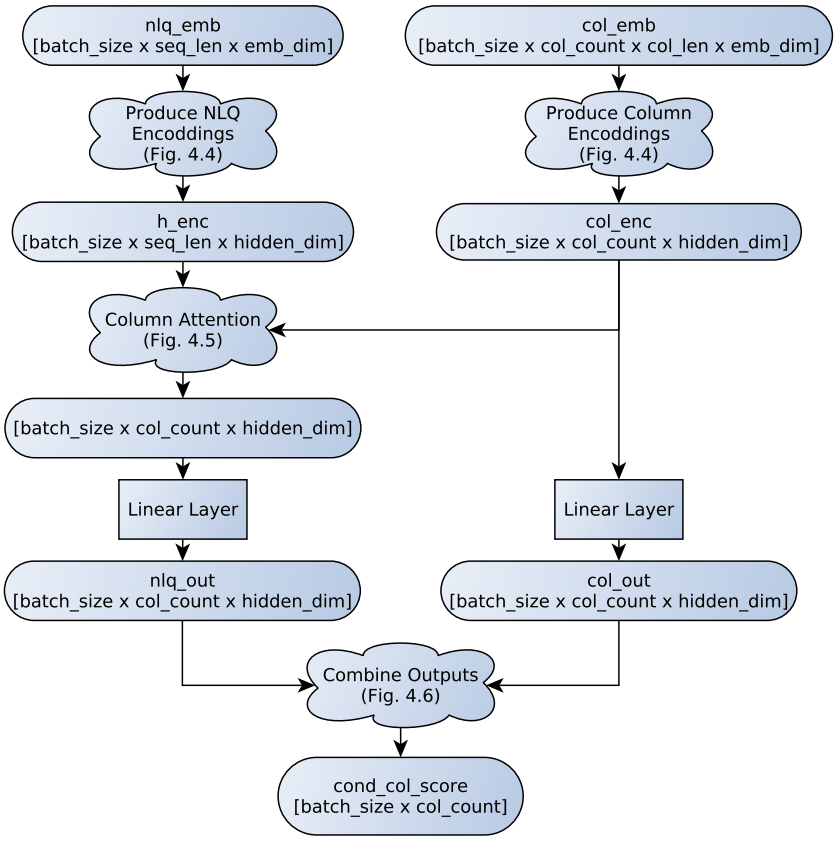

Figure 4.10: Condition Column Predictor

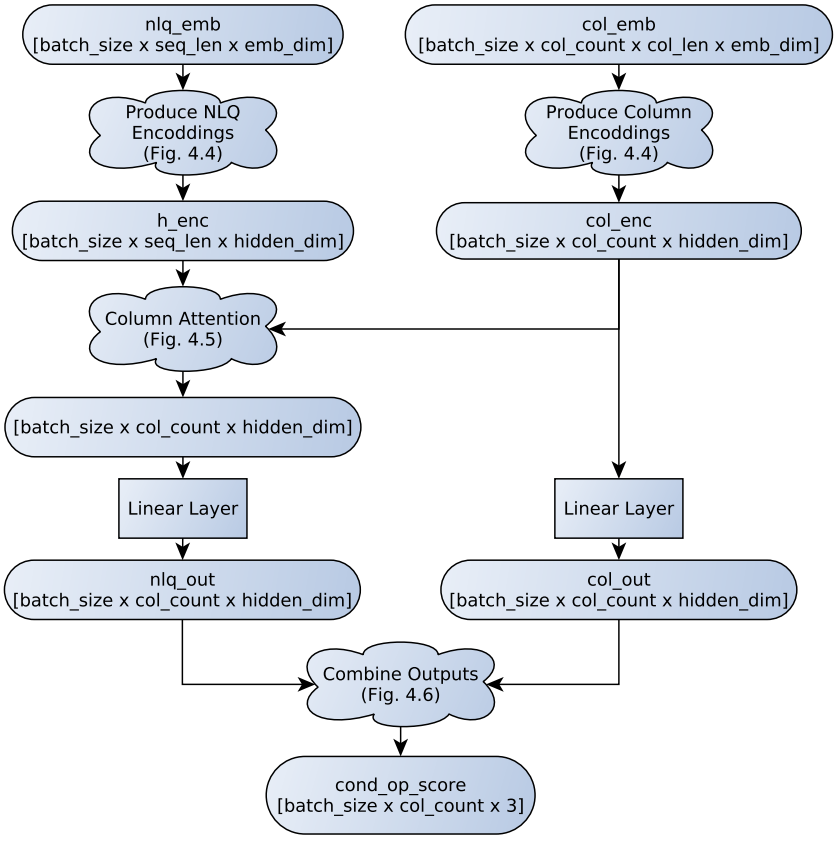

Figure 4.11: Condition Operation Predictor

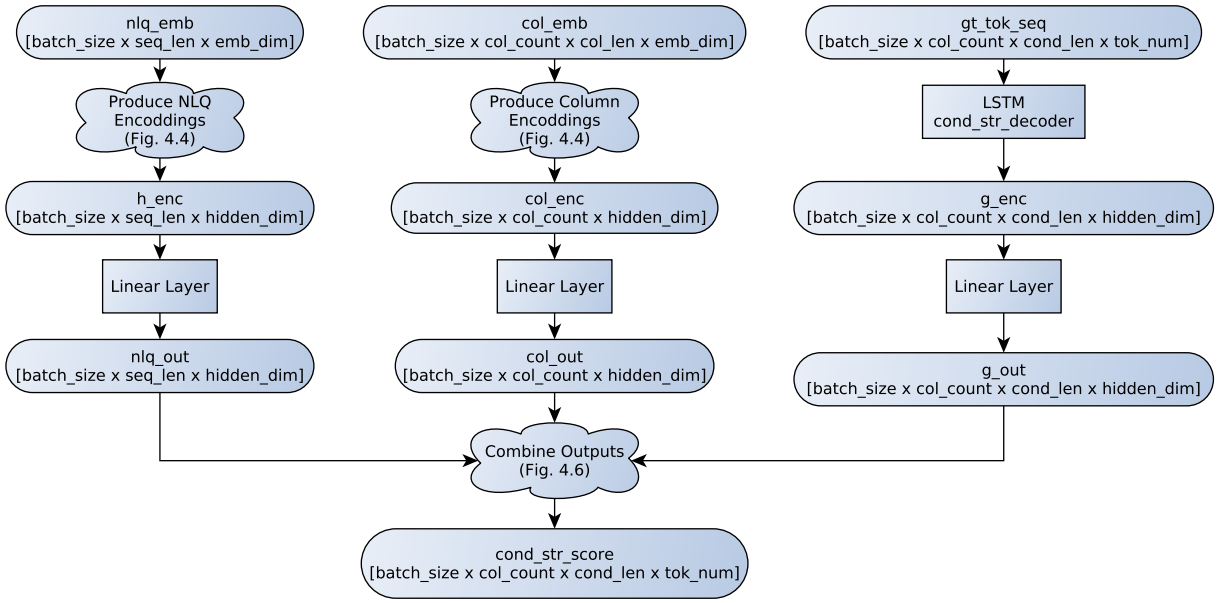

Figure 4.12: Condition String Predictor

2. Uses its last output as its next input
3. Operates differently during training and inference

An overview of the predictor's structure can be seen in Figure 4.12. A noticeable difference is that this component has an extra branch besides the NLQ and the table header branches. The third branch of the component employs a seq2seq type network, which changes the behavior of the whole component. The input of the new branch has a shape of $batch\_size \times col\_count \times cond\_len \times tok\_num$, where tok_num is the number of tokens in the NLQ (including a *"begin"* token at the start and a *"end"* token at the end). This matrix indicates which token has been assigned in which *position of the condition value* for each *NLQ in the batch*, for each *column in the table* corresponding to the NLQ.

For inference, the new branch operates as follows: First it receives an initial input of $cond\_len = 1$, containing only begin token ("<BEG>"). This notifies the component that the condition value must begin and as a result, the component predicts the first token (word) of the condition (for each column of each NLQ). Moving on, the hidden state of the LSTM is kept and the first output is now used as input. This can be seen as asking the component "If the first word is x what will be the second word of the condition value?". The new output is again used as the next input until the end token ("<END>") is predicted which signals the end of the condition string. Note that during this procedure, the inputs of the NLQ branch and the column names branch do not change.

During training, the aforementioned process must be simplified for two reasons. First, because the expected result is already known and there is no need to use the predictions as input. Second, and most important, because the predictions might be incorrect and training on incorrect input would hurt the model's performance. Instead of taking multiple inputs of $cond\_len = 1$ and using each output as the next input, all the inputs are given as one sequence. More specifically, the input of this branch is a sequence containing the begin token followed by the tokens of the (ground truth) condition value. Similarly, the correct output of the condition string predictor is a sequence of the tokens of the condition value followed by the end token.

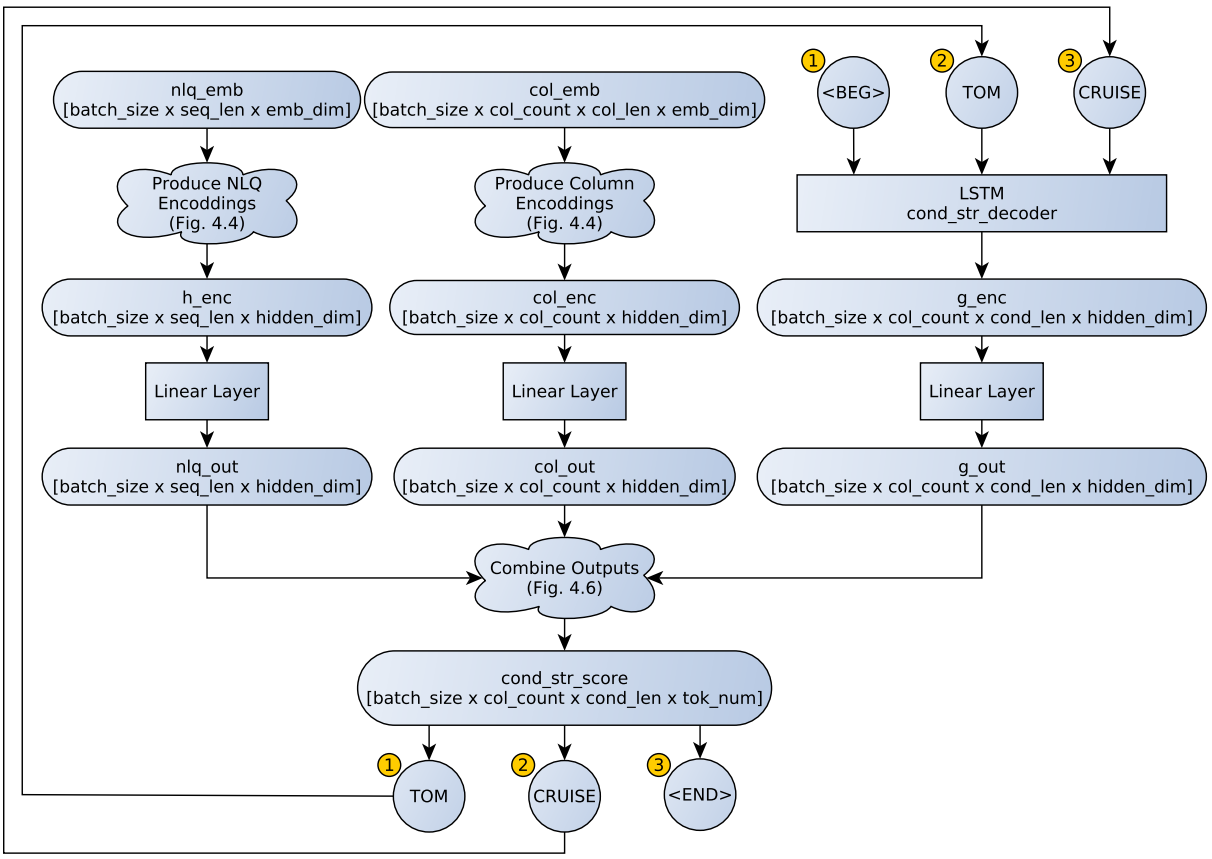

Figure 4.13: Condition String Predictor inference example

To better illustrate this, consider the following example NLQ: *"How old is Tom Cruise?"*, posed on a table containing among others the names and ages of various actors. The expected SQL query would be something like **SELECT age WHERE name = Tom Cruise**. Therefore, the condition value predictor would have to predict the sequence **["Tom", "Cruise", "<END>"]**.

During inference (i.e. when the training has been done and we don't have an expected output for our input), we only provide a *Begin* (<BEG>) token to the condition value decoder LSTM and as an output the condition value predictor returns the first predicted token of the condition value. In the current example that token would have to be *"Tom"*. The LSTM would keep its hidden state produced from the first token input (so that it has some knowledge about what it has seen before) and we would then feed it its previous output token ("Tom"). The model would take the new input and produce the next token of the string, which in this example would be *"Cruise"*. We then repeat the same process but this time, we would get the *End* (<END>) token signaling that this is the end of the condition value. An illustration of this process can be seen in Figure 4.13.

During training (i.e. when the model has an expected output for every input), there is no need to follow the same process because we already have access to all the tokens. Instead, we can give them to the decoder all at the same time as a sequence (i.e. ["<BEG>", "Tom", "Cruise"]). An illustration of the training process of the condition string predictor can be seen in figure 4.14.

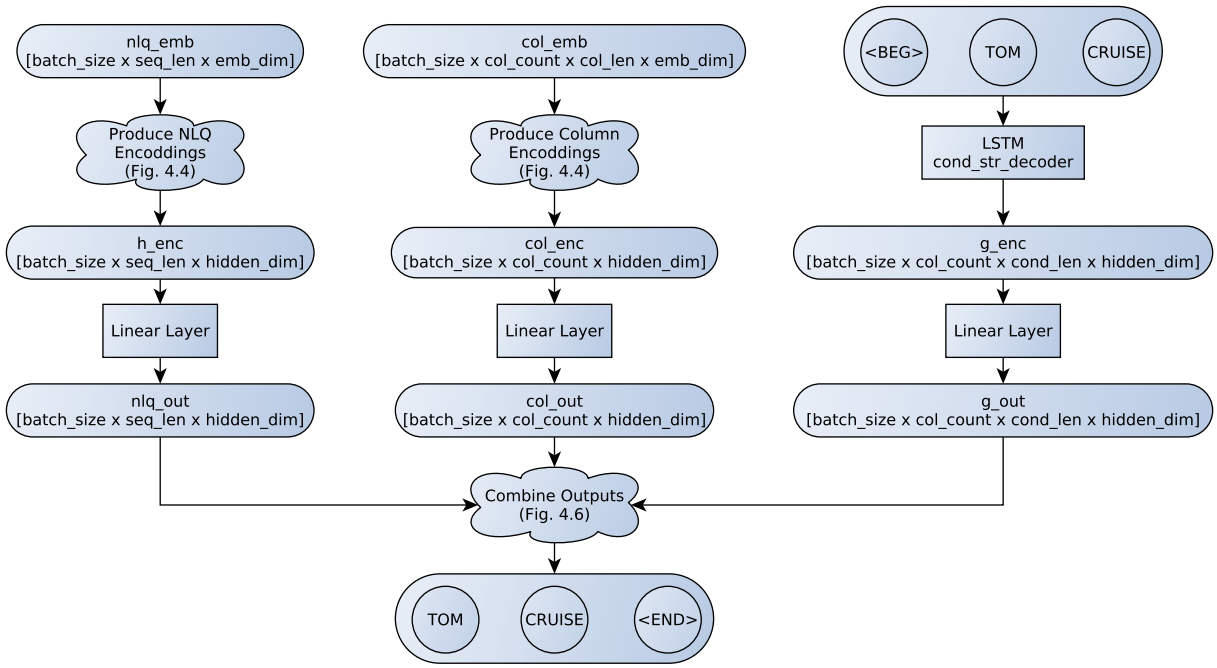

Figure 4.14: Condition String Predictor training example

4.4 Programming Details

Having described the system, it is also important to note some programming decisions made during its implementation. These decisions are made to avoid unnecessary computations that could reduce the system's speed.

Tokenizing Before Training

The tokenization process can be costly and time-consuming, especially when faced with a large data set such as WikiSQL. Additionally, training for 200 epochs means that every question is seen be the model 200 times. This means that if tokenization was to happen at run-time, the data set would have to be tokenized 200 times. For these reasons it is preferable to tokenize all NLQs and column names of the data set before training the model.

This is done in two steps. First the data set is loaded and every NLQ and column name is tokenized. A new data set is then created using all the contents of the original, along with the tokenization. In this way, when the model loads the tokenized data set, it can use the tokenized inputs instead of having to tokenize every input when it needs to use it.

Creating the Embedding Layer

As mentioned earlier, the embedding layer stores the corresponding numerical vector for each word in its vocabulary. However, the pre-trained embedding set that is being used features a vocabulary of 1.9 million words. Loading all these vectors (approx. 5GB) on the memory whenever the model is run would be very costly and inefficient. Given that the WikiSQL data set has a vocabulary size of only about 50 thousand unique words in its

NLQs and table column names, there is no need to load all the vectors in the pre-trained embeddings.

Before creating the embedding layer, it would be beneficial to first find which words are needed and load only their embeddings, to save time and memory. This is achieved in two steps. First a word index is created, containing all the unique words of the data set that will be used by the model. In our case this means all the words found in the NLQs and column names of WikiSQL. Then, all the pre-trained vectors are loaded and a new matrix is created, containing only the vectors of the words in the word index.

The new embedding matrix has a shape of $[size(word\_index) \times embedding\_dim]$, where $embedding\_dim$ is the size of the embedding vectors and is selected during the creation of the embedding set. The pre-trained embedding set used by SQLNet has an $embedding\_dim$ equal to 300. The newly created embedding matrix is the used to initialise the embedding layer.

4.5 Training Details

At first we will be following the same training and network configurations as described in the paper, to test that our implementation is correct. In the following chapter, however, we will discuss how we can change these choices to improve the model presented in the paper.

We will be using the Adam optimizer [7] with a learning rate equal to 0.001. The hidden states of all layers (indicated as hidden_dim in all figures) will be 100. All LSTM layers have 2 layers depths (i.e. each time we refer to a LSTM we are actually stacking two of them) and a dropout rate of 30% is applied to all of them.

We will be using the same tokenization of the data set, which is produced using the Stanford CoreNLP Tokenizer [11]. We will however present some problems observed and some proposals to improve on them.

The model is trained for a total of 200 epochs; during the first 100 epochs the weights of the embeddings are frozen (i.e. they stay the same and are not trained), during the last 100 epochs we allow the embeddings to be trained as well. This is something that increases accuracy as shown by the authors of SQLNet.

As mentioned before, the training of the model is performed on ARIS and we would like to thank and acknowledge GRNET S.A. for granting us access to the Greek national high performance systems, ARIS.

5. EXPERIMENTS AND RESULTS

In this chapter, we will present several improvements we made on the original SQLNet and evaluate them experimentally. We will start by testing it on the same benchmark its authors tested it on, i.e. the WikiSQL data set and at the same time, attempt to evaluate it using the criteria set at the beggining of this work. Following that, we will investigate the coverage of the data set by our embeddings. As we show, more than 20% of the words found in WikiSQL do not have a word embedding representation. By applying spell-checking and different tokenization techniques we attempt to increase the coverage and also test how it affects the system's performance. Moving on, we will attempt to test SQLNet on a complete database. However, since it cannot create queries over multiple tables, we will be creating views covering multiple tables of the IMDb database and feeding them to the system as a single table. Finally, we will be displaying some results from the IMDb database as well as other interesting results from the WikiSQL data set while we try to provide some insight on the system's performance, what it can and cannot answer.

5.1 Evaluation

After having described the SQLNet system, let us recall the criteria set for evaluating a NL2SQ system as described in section 2.2. A NL2SQL system can be evaluated on three axes: (a) its effectiveness and accuracy at synthesising valid SQL queries that correspond to the user's intentions, (b) the time it needs to synthesise the query and (c) its ability to work equally well on different databases needing as little aid as possible from the user. We shall now evaluate SQLNet based on these axes.

Accuracy

What must first be examined is the new implementation's performance. To this end we can test our system on the same tests that the authors of the paper perform, i.e. test its accuracy on WikiSQL's validation and test sub-sets. We can measure accuracy in two ways:

1. Comparing the output SQL to the ground truth SQL (referred to as **query match accuracy** and also used in [18]). If the predicted query is the same as the ground truth (the order of the conditions does not matter), then it is considered correct. This metric depends on the following sub-metrics and a prediction is considered accurate when all the following are simultaneously accurate:
   (a) **Aggregation function accuracy**, which measures if the predicted aggregation function is the same as in the ground truth
   (b) **Column selection accuracy**, which measures if the predicted column in the SELECT clause is the same as in the ground truth
   (c) **Condition accuracy**, which measures if the entire condition clause in the prediction matches the ground truth, without taking the the order of the conditions into account (e.g. "WHERE age > 20 AND name = tom" matches "WHERE name = tom AND age > 20"). A predicted condition matches the ground truth if all the following are simultaneously accurate:

Table 5.1: New Implementation's Accuracy on WikiSQL (CA refers to using column attention and EMB refers to using trained embeddings)

Model	Test Set				Validation Set			
	Acc_{agg}	Acc_{sel}	Acc_{cond}	Acc_{qm}	Acc_{agg}	Acc_{sel}	Acc_{cond}	Acc_{qm}
Our Impl. (+CA)	88.7%	92.9%	69.1%	58.9%	88.6%	93.6%	70.4%	60.2%
Paper (+CA)	90.3%	90.4%	70.0%	-	90.1%	91.1%	72.1%	-
Our Impl. (+CA +EMB)	88.4%	93.3%	73.1%	62.0%	87.9%	93.6%	73.6%	62.3%
Paper (+CA +EMB)	90.3%	90.9%	71.9%	61.3%	90.1%	91.5%	74.1%	63.2%

   i. Number of conditions
   ii. Column in each condition
   iii. Operation in each condition
   iv. Value string in each condition

2. Comparing the results of executing both these queries (referred to as **execution accuracy** and also used in [20])

For the means of our evaluation we will only be using the *query match accuracy* and its sub-metrics. The reasons for this are two. First, because the complexity of the queries found in the WikiSQL data set is relatively low, as we mentioned in section 3.2. Second, because the SQLNet system produces predictions based on its *query sketch* which matches the structure of WikiSQL's queries very closely, as we mentioned in section 4.1. As a result, the possibility that SQLNet makes a prediction that is different from the ground truth but yields the correct results in a meaningful way (i.e. not by chance) are low.

In table 5.1 we can observe the query match accuracy (Acc_{qm}) of the model's predictions on both the test and validation subsets of WikiSQL, along with its aforementioned sub-metrics ($Acc_{agg}, Acc_{sel}, Acc_{cond}$). The upper half of the table compares our implementation's result to the results presented in [18], after being trained using column attention (CA) for 100 epochs. The lower half presents the same comparison after making the word embeddings trainable and training the models for an additional 100 epochs.

As it can be seen, the new implementation's accuracy is very close to the initial implementation's score, as it was anticipated. Even though some small differences can be observed, they can most likely be attributed to the randomness of the initialisation of the model's weights and to the randomness of the dropout in the LSTM layers.

Even though these results are a validation that the new implementation works correctly, they also point out that the system's effectiveness does not meet our standards. A query match accuracy of about 60% indicates that the model fails to produce a query that matches the user's intent almost half the time, which is not enough.

Time

When evaluating a SQLNet on the aspect of time efficiency we must take into account that, being a deep learning system, it works in two phases: training and inference.

Training is a costly task that requires iterating over the large data set that is WikiSQL for 200 epochs. A single epoch of training on the Greek supercomputer "ARIS" takes about 15 minutes. This time could be significantly improved by adding support for parallel computations and better use of the system's multiple graphics card. The training process might be costly, however it must take place only once and will produce a model that can

Table 5.2: Testing SQLNet's Adaptability on Different Categories of Tables

Category	#NLQs	#Tables	Acc_{agg}	Acc_{sel}	Acc_{cond}	Acc_{qm}
Movies	27	9	92.6%	100.0%	70.4%	63.0%
Music	37	11	78.4%	78.4%	51.4%	37.8%
Sports	37	13	89.2%	97.3%	62.2%	54.1%
Politics	23	8	73.9%	95.7%	65.2%	39.1%
Science	25	9	84.0%	92.0%	60.0%	44.0%
Economics	17	7	76.5%	82.4%	29.4%	23.5%
Technology	25	5	92.0%	80.0%	32.0%	24.0%

be used on any machine, such as an ordinary laptop, to make as many predictions as we like.

Inference is the phase during which the model has been trained and can make predictions on NLQs it has never seen before. The time needed to construct a query, on a laptop with an Intel i5 processor, is just a few seconds. This means that a user could run this system on his computer and create SQL queries from NLQs almost instantaneously, which in most cases is adequate of our expectations of time efficiency.

Database Adaptability

The third and final axis of our evaluation, is whether the system can adapt on new databases with as little help from the user as possible. Given that SQLNet does not work on full databases but on single tables, it cannot be fully evaluated on this axis. It would be possible, however, to examine how it adapts on different tables. To this end, it is to our benefit that the data set on which we are training the model is multi-disciplinary, since it was taken from Wikipedia. The system's adaptability could be tested by selecting tables from WikiSQL that fall into some basic categories and examining if the system performs equally on all of them.

Taking that into account, I have created 7 categories of tables based on the subject that they cover. The tables were hand-picked from the WikiSQL test set and all the question that were posed on these tables were used.

Each of the created categories was evaluated separately and the results can be seen in Table 5.2. Each row represents a different category and the accuracy scores that SQLNet was able to achieve. Looking at the *Query Match Accuracy* column, we notice that there is a big difference between the categories. The highest ranking category is ”Movies” with 63%, which is close to the model's usual performance, and the lowest ”Economics” with 23.5%. The model performs well, for its standards, in categories such as ”Movies” and ”Sports” but it does not achieve the same accuracy on the other 5 categories. In fact, the model produces an accuracy lower than 44% in all the 5 remaining categories. This difference could be an indicator that the model does not adapt well on different categories of tables.

This could be attributed to the unbalanced number of tables available for each category. While searching for tables to add to our categories, it was noticed that there were many more tables available for ”Movies” and ”Sports” in the data set, than for any other category. To conclude, even though a larger amount of NLQs in each category could allow us to be more certain, these results lead us to believe that SQLNet does not adapt that well on different categories.

Table 5.3: Words without an embedding for each tokenization

Tokenization	Unique Words	Unknown Words	
Baseline	51,114	11,181	21.8%
Spellchecking	51,114	8,197	16.0%
Spellchecking (ascii)	51,114	9,202	18.0%
Split ”/”, ”-”, ”.”, ”_”	48,869	10,847	22.2%
Split & Spellchecking (ascii)	48,869	8,731	17.8%

5.2 Trying to Maximize the Embedding Coverage of WikiSQL

While processing the inputs of our model it can be observed that about 20% of the words found in WikiSQL's NLQs and column names (i.e. the model's inputs) have no representation in Stanford's pre-trained GloVe embeddings that we are using. When our model comes across a word that has no embedding, it treats it as an unknown word and represents it with a zero vector to be fed to the network. This means that when given to our model, these words provide little to no knowledge at all and are most likely seen as noise, by the model, that makes the task of understanding the NLQ and producing a SQL query even more difficult.

Additionally, if the contents of the WikiSQL tables are taken into account as well, the percentage of words without an embedding representation rises up to about 50%. These however are words that our model never sees, because the model does not use the tables' contents.

It would be interesting to investigate why this happens and how this percentage could be improved.

Exploring Unknown Words

A first course of action would be to observe the unknown words, in order to understand why they might not be present in the pre-trained embeddings. With a quick look we can recognise some distinct categories of unknown words:

- Numbers (”138.063”, ”+1:35.553”, ”3x5402”)
- Dates (”15/02/1975”, ”24.11.07”)
- Different languages (Chinese, Arabic, etc.)
- Names (”petrovits”, ”kassianos”)
- Badly tokenized words (”goals/matches”, ”package-socket”, ”non-seating”)
- Spelling mistakes (”innhabitants”, ”episoede”)
- Symbols (scientific symbols, etc.)

Now we can begin to understand why there are so many unknown words. We obviously cannot expect the pre-trained GloVe to have seen every single number that can be formed, nor every date with every possible date format. The same goes for names, especially if

they are not English. Furthermore, these embeddings have been trained on an English text corpus so they are expected to not recognise other languages. Finally there are a lot of symbols and special Unicode characters that are reasonably unknown to our embeddings. In any case we must note that these categories of words are not always unknown; some similar examples of words can have an embedding (e.g. ”24:02”, ”115–118”, ”west/sudwest”, etc.).

There are however some things that we could improve. We could certainly split tokens like ”goals/matches” into two separate words that would definitely be known. We could also try to correct spelling mistakes.

Correcting Spelling Mistakes

To reduce the number of unknown words, we will first attempt to correct any spelling mistakes it is possible. This will not be done manually because it would undermine one of the main goals of the NL2SQL problem, which is that the system has to require little to no human involvement. Instead, a spell checking algorithm will be used. This algorithm was first proposed by Peter Norvig in a blog post [15] and has now been neatly implemented as a python library named *pyspellcheck*.

Now for every word that does not have an embedding, we will ask for the spell checker’s correction and check if the correction has an embedding. If the correction has an embedding, we will assign the correction’s embeddings to the word instead of the zero vector it would have been assigned as an unkown word. By applying this we notice the following behaviors:

- Some spelling mistakes are indeed being corrected
- Some foreign words are being corrected to unrelated English words (e.g. the French word ”montagne” which means mountain, was corrected to ”montage”)
- Some tokens containing numbers are being corrected to similar tokens (e.g. ”t214”, which does not have an embedding, was corrected to ”t2”, which has an embedding)
- Some symbols and words in Chinese and Arabic were corrected to random articles (e.g. the degree symbol was corrected to ”a”)

During this procedure we notice a lot of symbols and foreign words being corrected to English articles such as ”a”, ”of”, etc. This might cause the system to develop a false understanding of these words, which could have negative effects in its performance considering how often these articles are used. To prevent some unwanted corrections, we also try correcting only words that contain only ASCII characters. This means that words containing non-ASCII symbols, letters with accents as well as Chinese, Arabic etc. characters will be ignored.

This results in a reduction of unknown words, which can be seen in table 5.3. More specifically, line 2 shows the improvements by spell-checking all words and line 3 shows the result of spell-checking words with ascii characters only. As expected, correcting ascii words only, results in a smaller but probably more sensible reduction of unknown words. Nevertheless, even when correcting ascii words only, we manage to reduce the number of unknown words by about 2,000 words.

Improving Tokenization

Another thing we did is try to improve the tokenization quality. As we mentioned earlier, there are a lot of badly tokenized words that have not been split on characters like ”/”, ”-”, etc. resulting in unknown words such as ”goals/matches”, ”package-socket”, etc. At the same time, however, there are a lot of dates and numbers that have the same characters in them that we would not want to change. So the approach followed in this case will be to split all tokens that contain ”/”, ”-”, ”_”, ”.” or ”,” but only if these characters are not adjacent to a number.

Following this technique, the token ”package-socket” will be split into the three separate tokens: ”package”, ”-” and ”socket”. However, the token ”138.063” will not be split as it would not be any better to have the tokens ”138”, ”.” and ”063” instead of it.

This results in a decrease both in the number of total unique words but also the number of unknown words. The results can be seen in table 5.3. Specifically, line 4 shows the results of tokenizing using these ”split” rules and line 5 shows the result of applying spell-checking (on ascii words only) after using the new split-tokenization. What is interesting is that even though there are now less unknown words, the percentage of unknown words is higher than the original because the number of unique words has decresed as well. Additionally, applying spelling corrections on top of the split-tokenization reduces the number of unknown words by an extra 1,000 words. This means that in the split-tokenization there are less words to be corrected.

Results in Accuracy

It is evident that these modifications in the way we handle our input can increase the embedding coverage but how does this reflect in the final model performance? To test this we will train three new models, using the improved NL processing and test them on the same test we used for our implementation.

The results of each new model compared to our initial model can be seen in table 5.4. The table is separated in two parts, the top part being the model’s accuracy after training for 100 epochs and the bottom part after making the word embeddings trainable and training for an additional 100 epochs. The first line in both parts represents our initial model, presented in table 5.1, which uses the tokenization used in [18] (line 1 of table 5.3) while the following three lines show the accuracy achieved by the models trained using the improvements we discussed earlier. The marking +SP denotes a model using spellcheck on words with ascii characters (line 3 of table 5.3), the marking +TK denotes a model using the improved tokenization (line 4 of table 5.3) and the marking +TK +SP denotes a model using both improvements (line 5 of table 5.3).

Notice how the spell-checking model achieves a query match accuracy higher than the original model by almost 2% in the train set (61.6% as opposed to 58.9% and 63.9% as opposed to 62% when using trained embeddings). In contrast, the two other models, display an accuracy lower than the original model. It is interesting how applying spell-checking without changing the tokenization increases the accuracy but applying spell-checking to the improved tokenization has negative effects on the accuracy. This might be an indicator that the proposed changes in the tokenization are doing more harm than good.

More interesting observations can be made from the three accuracy sub-metrics:

Table 5.4: Accuracy of modified NL processing models on WikiSQL (EMB: Trained Embeddings, SP: Spellchecking, TK: Improved Tokenization)

Model	Test Set				Validation Set			
	Acc_{agg}	Acc_{sel}	Acc_{cond}	Acc_{qm}	Acc_{agg}	Acc_{sel}	Acc_{cond}	Acc_{qm}
SQLNet	88.7%	92.9%	69.1%	58.9%	88.6%	93.6%	70.4%	60.2%
SQLNet (+SP)	88.6%	91.2%	73.4%	**61.6%**	87.8%	92.1%	74.0%	**61.7%**
SQLNet (+TK)	88.8%	91.9%	67.0%	56.4%	88.2%	92.7%	66.9%	56.3%
SQLNet (+TK +SP)	89.5%	91.9%	66.8%	56.4%	88.3%	92.7%	66.8%	56.5%
SQLNet (+EMB)	88.4%	93.3%	73.1%	62.0%	87.9%	93.6%	73.6%	62.3%
SQLNet (+EMB +SP)	88.8%	92.3%	75.3%	**63.9%**	88.3%	93.0%	75.6%	**63.7%**
SQLNet (+EMB +TK)	88.9%	92.9%	68.8%	58.3%	88.2%	93.8%	68.9%	58.3%
SQLNet (+EMB +TK +SP)	89.0%	92.6%	68.5%	58.2%	88.2%	93.9%	68.6%	58.0%

- The highest aggregation function accuracy is achieved by the model using both spell-checking and the improved tokenization, which stands out considering that it has the worst performance in all the other metrics
- No improved model manages to out-perform the original when it comes to column selection accuracy, not even the spell-checking model which has a higher query match accuracy
- The spell-checking model achieves the highest condition accuracy between all other models, which is probably why it also has the best performance in query match accuracy as well

5.3 Testing on Database Views

Another very interesting question is how our system performs when faced with a complete database and not just single tables. Since SQLNet is designed to only accept single tables as input, what we tried is to create views covering multiple tables of a database. Doing this, we are be able to pass a view of multiple tables as a single table, thus eliminating our system's incapability of taking multiple tables into account and performing joins. This is an attempt to test the system on an experiment that is somewhat closer to a complete database so as to get an idea of how it would perform.

Creating the Input

To achieve our goal we will be using the IMDb database, that was published along with [19], on which we have created the following views/tables:

$$\begin{aligned}
\text{actor_by_tv_series} &\leftarrow actor \bowtie cast \bowtie tv_series \\
\text{actor_by_movie} &\leftarrow actor \bowtie cast \bowtie movie \\
\text{actor_by_movie_by_genre} &\leftarrow actor \bowtie cast \bowtie movie \bowtie classification \bowtie genre \\
\text{movie_by_company} &\leftarrow movie \bowtie copyright \bowtie company \\
\text{movie_by_keyword} &\leftarrow movie \bowtie tags \bowtie keyword \\
\text{movie_by_genre} &\leftarrow movie \bowtie classification \bowtie genre
\end{aligned}$$

NLQ:
→ ”Find all movies by Netflix”
Column Names:
→ [”movie title”, ”movie release year”, ”movie title aka”, ”company name”, ”company country code”]

Figure 5.1: An input example from the IMDb view movie_by_company

An example of a table that is created by these views can be seen in table 5.5 and a complete example of the input that SQLNet receives is shown in Figure 5.1. Notice how the column names of the created view also carry the name of the table from which the column is taken (e.g. ”movie”, ”company”, etc).

After having created these tables based on the aforementioned views, we are able to pass them as input to SQLNet just as we have been doing with WikiSQL’s tables. Any SQL queries created will be based on the query sketch we described in section 4.1 and will be valid for execution on the equivalent view but not on the entire database.

Testing Performance on the Views

Moving on, the new views can now be tested, to see how the system reacts to NLQs directed to them. The NLQs given to the system for each view along with the results produced can be seen in appendix A. The queries used originate from THOR’s [2] query benchmark.

The questions are separated in two categories: (a) NLQs that can be (theoretically) translated into SQL queries by SQLNet and (b) NLQs that cannot be translated into SQL queries because of SQLNet’s restrictions. Such restrictions are also described at the end of this chapter. For the questions of category (a), a ground truth was added manually so that the accuracy of the predictions can be measured.

The goal of these NLQs is to try to utilise almost all the available attributes in as many parts of the query as possible (i.e. in the SELECT clause, the condition, etc.) and at the same time try to cover as many possible types of question a user could ask this specific database. Given a movie database, it is expected to encounter a lot of questions that contain names, titles, years and roles (character names). The difficulty of the NLQs varies from easy (e.g. ”Return all movies of Brad Pitt”) to difficult (e.g. ”Who acts as Olivia Pope in the series Scandal”). In the second question, the system needs to understand that the name given belongs to a character (role attribute) and not to an actor (name attribute). There are also some deliberate spelling mistakes included so as to examine how the system reacts to them.

What we notice in general is that the system’s predictions are not that good. Surely this

Table 5.5: An example of an IMDb view: movie_by_company

movie title	movie release_year	movie title_aka	movie budget	company name	company country_code
’Northwest Passage’	1940	-	$2,677,762	MGM	[us]
’I Know Where I’m Going!’	1945	-	£200,000	Archers, The	[gb]
⋮	⋮	⋮	⋮	⋮	⋮

Table 5.6: Accuracy on IMDb single tables and views (EMB: Trained Embeddings, SP: Spellchecking, TK: Improved Tokenization)

Model	**Tables** $(n = 20)$				**Views** $(n = 25)$			
	Acc_{agg}	Acc_{sel}	Acc_{cond}	Acc_{qm}	Acc_{agg}	Acc_{sel}	Acc_{cond}	Acc_{qm}
SQLNet	90.0%	55.0%	15.0%	0.0%	92.0%	44.0%	24.0%	4.0%
SQLNet (+SP)	90.0%	70.0%	20.0%	10.0%	92.0%	28.0%	12.0%	4.0%
SQLNet (+TK)	95.0%	50.0%	15.0%	5.0%	84.0%	52.0%	8.0%	4.0%
SQLNet (+TK +SP)	100.0%	45.0%	25.0%	10.0%	92.0%	48.0%	8.0%	4.0%
SQLNet (+EMB)	90.0%	55.0%	30.0%	5.0%	92.0%	40.0%	20.0%	4.0%
SQLNet (+EMB +SP)	85.0%	70.0%	25.0%	5.0%	92.0%	24.0%	12.0%	4.0%
SQLNet (+EMB +TK)	75.0%	70.0%	10.0%	0.0%	92.0%	32.0%	0.0%	0.0%
SQLNet (+EMB +TK +SP)	90.0%	60.0%	25.0%	10.0%	92.0%	28.0%	8.0%	4.0%

decrease in performance can also be attributed to the small number of examples, the use of a different database and the introduction of a new methodology (using views to counter the inability to use JOINs). Despite these factors, these results can surely provide an insight on aspects of the system that are in dire need of improvement.

Table 5.6 shows the achieved accuracy of queries on single tables of the database (left side) and on views created from multiple tables (right side). The aggregation function accuracy is the only metric which is close to the performance on the WikiSQL data set. The select column accuracy is lower than previously, ranging between 24% and 52% (on views) compared to the 90% accuracy achieved on WikiSQL. The biggest problem, however, is observed in the condition accuracy metric, which in some cases can be as low as 0% and in the best case reaches only 30%. This is clearly what drags down the query match accuracy as well.

What is most alarming, is that the performance on single tables of the IMDb database is not that higher than the performance on views. One would expect, that after having seen so many tables in the WikiSQL data set, our model would not struggle that much (max. 10% query match accuracy) with similar tables. This discovery resonates to the concern that the WikiSQL data set is not that well crafted (section 3.3) and that a model trained on it might not generalise that well in real-world situations.

Some simple questions work relatively well and their results are correct or almost correct We can see some inexplicably bad condition values and condition clauses in general and in a lot of cases the wrong column is being selected. By looking at the results it seems like the system is missing some information from the NLQ or that it is not understanding the tables that well. This might imply that the complicated condition predictors (i.e. number, column, operation and value predictors) fail to generalise and are in need of fine tuning or even training in new and different data sets.

Another thing we can notice is that our system does not understand that the phrase ”female actor”, ”male director”, etc. means that an extra condition like ”WHERE gender = female/male” must be added. This might happen because most NLQs in WikiSQL (from which it was trained) don’t have such ”hidden” conditions, i.e. in most cases the condition is more explicit and at the end of the NLQ. For example, the question ”Find all actors from Austin that are female” might have been easier to translate. Even when the condition is added at the end of the sentence (e.g. ”Find all actors from from Austin that are female”) the system struggles to use the term correctly. By inspecting the WikiSQL data set, it was noticed that it also lacks NLQs using a gender attribute. Even when attempting to recreate something similar, using a table from WikiSQL that contains information about political

candidates[1], the same mistakes were encountered (table 5.7).

Table 5.7: Examples of queries about gender

NLQ:	Show male candidates
SQL:	**SELECT** Votes **WHERE** Votes =
NLQ:	What are the names of male candidates?
SQL:	**SELECT** Candidate's Name **WHERE** Rank = male
NLQ:	Find all female candidates
SQL:	**SELECT** COUNT(Candidate's Name) **WHERE** Rank =

One thought on what could be going wrong is the main difference between these tables created by view and the WikiSQL tables. The column names of tables created from views also carry the name of their original table. For example, in *movie_by_company* the first columns are ”movie title”, ”movie release year”, ..., ”company name”, ”company country code”. This might make columns from the same table harder to distinguish because they all start with the same word. It might confuse the system or make it harder to understand which is more appropriate each time. However, this intuition was proven wrong; after removing the table names no significant change was observed.

5.4 Interesting Examples of Queries

In order to fully understand the system's capabilities it would be beneficial to examine it further than just a simple accuracy percentage. For this reason, this section will examine specific types and cases of questions and check how the system responds to them. To do this, we will use a table from the WikiSQL test set and produce our own NLQs on this table. We will be checking which words have embeddings and what results are being produced from our system. We will be using the SQLNet model trained with column attention for 200 epochs, with its embedding layer being trained for the 100 last epochs.

Spelling Mistakes

In a previous section we noticed and tried to correct the problem of misspelled words, but it would also be interesting to see what actually happens when misspelling occur. The underlined words are the ones we are spelling incorrectly on purpose and the words in red are the words that don't have an embedding. Notice that in some cases we might make a spelling mistake but the misspelled word might still have an embedding.

In the table above we have some NLQs posed on a table[2] with data of players from American football teams. As we see the first NLQ, without mistakes, gets transformed correctly. When we make a spelling mistake on a word that indicates the column that must be selected, there is a mistake in the column that gets selected. This is expected; the system does not recognise the word because it only receives a zero vector because this word has no embedding. When we make a spelling mistake in a word that has to be used as a value for a condition we notice something interesting. When a word doesn't have an embedding it gets ignored in the condition string (e.g. when we mistakenly write ”Maden” instead of ”Madden”) but when a word is misspelled but still has an embedding it gets

[1]Table ID: 2-12890300-4
[2]Table ID: 1-11677691-4

Table 5.8: Examples of Spelling Mistakes & Predicted SQL Queries, red color indicates words without embeddings and underlining indicates spelling mistakes

NLQ:	What's the position of Tre Madden?
SQL:	**SELECT** Position **WHERE** Player = tre madden
NLQ:	What's the possition of Tre Madden?
SQL:	**SELECT** School **WHERE** Player = tre madden
NLQ:	What's the position of Tre Maden?
SQL:	**SELECT** Position **WHERE** Player = tre
NLQ:	What's the position of Te Madden?
SQL:	**SELECT** Position **WHERE** Player = te madden
NLQ:	What's the position of Te Maden?
SQL:	**SELECT** Position **WHERE** Position = te
NLQ:	hat's the position of Tre Madden?
SQL:	**SELECT** Position **WHERE** Player = tre madden
NLQ:	Whqt's the position of Tre madden?
SQL:	**SELECT** Position **WHERE** Player = tre madden
NLQ:	Position of Tre Madden?
SQL:	**SELECT** Position **WHERE** Player = tre madden

included nevertheless (e.g. "When we write "Te" instead of "Tre"). This shows that the impact of word not having an embedding is worse than the impact of a misspelled word. One last thing we notice is that it doesn't matter if we misspell the introductory word of the question (i.e. "what"), nor if it doesn't have an embedding nor if its completely removed. In all three cases we get the same correct result.

Words with no Embeddings

We have already seen an interesting example where a word without an embedding is not used in the condition string when it should have, it would be interesting to investigate other similar cases. This time we will be using a table[3] that contains information on songs from Indian movies. Again, the words in red color don't have an embedding.

Table 5.9: Examples of Words without Embeddings & Predicted SQL Queries, red color indicates words without embeddings

NLQ:	In which songs is Ravindra Jain the music director?
SQL:	**SELECT** Song name **WHERE** Music director = ravindra jain
NLQ:	In which songs is Rajan Bawa the music director?
SQL:	**SELECT** Lyricist **WHERE** Music director = rajan bawa
NLQ:	In which songs is Usha Khanna the music director?
SQL:	**SELECT** Song name **WHERE** Music director = usha
NLQ:	In which songs is Jugal Kishore-Tilak Raj the music director?
SQL:	**SELECT** Lyricist **WHERE** Film name = jugal —raj

In these examples we see that a word without an embedding can be used in the condition string, in contrast to what happened in the previous example. We also notice, however, that it might cause other problems. For instance, in the second example the name "Rajan"

[3]Table ID: 1-11827596-2

gets used in the condition value but for some reason changing the name breaks the column selection and gives an incorrect result. In the third example only one of the two unknown words gets used in the condition string and in the final example we get a complete mess where both the selected column and the condition are wrong. As a general observation we can say that having words without embedding in our NLQ increases the chances of something going wrong.

Paraphrasing words

Another aspect that would be interesting to investigate is how well our system deals with different phrases that mean or imply the same thing. This time we will be using a table[4] that contains info on the national basketball team of Lithuania.

Table 5.10: Examples of Paraphrasing & Predicted SQL Queries

NLQ:	Which player has a height of 2.00?
SQL:	**SELECT** Player **WHERE** Height = 2.00
NLQ:	Which player is as tall as 2.00?
SQL:	**SELECT** Player **WHERE** Height = 2.00
NLQ:	Which player is 2.00 meters tall?
SQL:	**SELECT** Player **WHERE** Height = 2.00
NLQ:	Who has a height of 2.00?
SQL:	**SELECT** Player **WHERE** Height = 2.00

NLQ:	Which players play the guard position?
SQL:	**SELECT** Player **WHERE** Position = guard
NLQ:	Which players are guards?
SQL:	**SELECT** Player **WHERE** Position = guards
NLQ:	Show me the guards
SQL:	**SELECT** Current Club **WHERE** Player = guards **AND** No = guards
NLQ:	Show me the players that are guards
SQL:	**SELECT** Player **WHERE** Position = guards
NLQ:	Who are the guards?
SQL:	**SELECT** Current Club

Here we have a set of example where the system performs exceptionally and a set where it works incorrectly. We see that when asking about the height of the players in different ways it produces correct results. However when asked about which players are in the guard position, it doesn't do that well unless the word player in found in the NLQ. This might be due to the fact that it might be more common to ask about the height of a person in different manners but talking about the word ”guard” in the sense of the basketball position might be less common.

Something slightly unrelated, but very interesting, is that between the SQL queries asking about the guards, only the first would execute correctly. This is because the column *Position* of the table contains the word ”guard” and not ”guards”. This is something the system is not aware of and also is completely incapable of solving. As we mentioned before, the condition value is always a sequence of words from the NLQ so in this case, if the user's

[4] Table ID: 1-12962773-10

NLQ contains the wrong value, he is doomed to get a SQL query that might seem correct but does not return any results.

5.5 Queries that SQLNet can not Answer

We have demonstrated some queries that are very interesting in testing the system's limits to see where it performs well and where it doesn't. Now we will talk about some types of NLQs that are clearly out of the system's reach and we would not expect it to provide a correct SQL. These limitations, as we will see, are mostly due to the the simplicity of the *Sketch* that all queries produced by SQLNet are based on.

Open-Ended Questions

Because of the familiarity most users have with search engines, they might pose vague and open-ended questions such as ”Tell me about ...” or ”Find about ...”. A possible solution could be to include the option of using the asterisk (*) operator in the SELECT clause, among all the columns of the table. However such questions highlight the need to include data exploration capabilities in NL2SQL systems. Let's use the table of football players once again to demonstrate some examples.

Table 5.11: Examples of Open-Ended NLQs & Predicted SQL Queries

NLQ:	Tell me about Tre Madden
SQL:	**SELECT** Position **WHERE** Player = tre madden
NLQ:	Who is Tre Madden?
SQL:	**SELECT** Player **WHERE** Player = tre madden

Queries answered with a yes or a no

Questions that are answered with a yes or a no cannot be strictly answered because SQLNet only returns a SQL query and by extension the resulting data. Using that data to produce an affirmative or negative response demands an extra layer of processing which is currently beyond the scope of the system. The user could look through the resulting data and reach to a conclusion by himself but it would be nice to be able to have the system do that for him. We will demonstrate some examples from the football players table.

Table 5.12: Examples of NLQs Answered with Yes or No & Predicted SQL Queries

NLQ:	Are there any players from Springfield High School?
SQL:	**SELECT** Player **WHERE** School = springfield high school
NLQ:	Is Tre Madden a linebacker?
SQL:	**SELECT** Hometown **WHERE** Player = tre madden **AND** Position = linebacker

As we see, in the first case the user would get an indirect response to his question. In the second example however, the system would return something completely irrelevant that would be of no use to the user.

Comparing Table Entries

This is something that clearly cannot happen because of the limitations of the *Sketch* we are trying to complete. Such NLQs would require different structures of SQL queries, perhaps a nested query. Lets use the table of the Lithuanian basketball team once more to demonstrate some examples.

Table 5.13: Examples of NLQs Comparing Table Entries & Predicted SQL Queries

NLQ:	Who is the tallest player?
SQL:	**SELECT** Player
NLQ:	Who is the tallest guard?
SQL:	**SELECT** Player **WHERE** Position = guard
NLQ:	Who is taller than the player with the number 13?
SQL:	**SELECT** Height **WHERE** No = 13

Using Multiple Constraints on the Same Attribute

Another restriction originating from the design of the system is the incapability for the same attribute (table column) to be included more than once in the condition clause. The condition column predictor makes a prediction about the most probable column to appear in the condition clause and the k highest ranking columns are chosen based on the condition number predictor's output. This does not leave room for the possibility of a column appearing more than once. As a result, the system cannot correctly translate NLQs such as the following questions posed on a table[5] describing a political composition over the years:

Table 5.14: Examples of NLQs using multiple constraints on the same attribute

NLQ:	Show all treasurers between 2004 and 2008
Expectation:	**SELECT** Treasurer **WHERE** Year > 2004 **AND** Year < 2008
Prediction:	**SELECT** Electoral College votes **WHERE** Year = 2008 **AND** State Senate = 2008
NLQ:	Who was the governor after 2004 and before 2008?
Expectation:	**SELECT** Governor **WHERE** Year > 2004 **AND** Year < 2008
Prediction:	**SELECT** U.S. Senator (Class III) **WHERE** Year > 2004 **AND** State Senate > 2004

[5] Table ID: 1-18052353-4

6. CONCLUSION

In this thesis we talked about the NL2SQL problem, we looked at the general architecture of a NL2SQL system, compared some interesting NL2SQL systems and focused on the SQLNet system, which tries to tackle the problem using Deep Learning techniques. To better understand SQLNet, we examined the WikiSQL data set which it was made for and trained with and underlined its relatively low complexity. After developing our own implementation of SQLNet, we first tested it on WikiSQL and got similar results as its creators. We then tried to improve its performance based on the remark that a big percentage of the words it handles have no vector representation in its word embedding set. We discovered that by performing a spellcheck on every word we don't have an embedding for, we can increase the model's accuracy on WikiSQL by 2%. Following this, we tried to test the model on complete databases (using views) and specific types of questions that would be interesting in real use-cases and noticed that the model does not respond that well to neither them. This is an indicator that even though SQLNet performs relatively well on the WikiSQL task, it still requires some improvements before it's ready to be used in bigger scale scenarios.

Future Work

While working on SQLNet and studying similar systems some ideas for improving the system came up, that seem promising but were not included in this thesis. Both were inspired by the gap in the embeddings we have talked about and aim to enrich the embeddings as well as the NL understanding of the system in general.

The first idea is to train a set of GloVe embeddings using the contents of the tables we are using to train the model. As other non-ML systems create word indices and dictionaries for word disambiguation using the contents of their databases, we could also add more info to our embeddings using the contents of our tables. More specifically, one possible approach would be to train a new set of word embeddings using each column of each table as a GloVe *context*. This means that these new embeddings would learn how often a word in a table is associated with the name of a column.

An other idea is to try to encapsulate a syntactic, dependency, etc. tree in the embeddings. There has been work [10] on the Structural Embedding of Syntactic Trees (SEST) for NL tasks using character or word embeddings, that has shown that it possible to embed such a structure in the embeddings and that it can improve the model's accuracy. Considering that most non-ML solutions of the NL2SQL problem use a syntactic or other type of tree, it would be very interesting to see if ML solutions can also benefit from such a structure. Besides giving very beneficial information on every word of the NLQ, these embeddings would also help understand unknown words.

ABBREVIATIONS - ACRONYMS

DB	Data Base
DL	Deep Learning
IQR	Intermediate Query Representation
LSTM	Long Short-Term Memory
ML	Machine Learning
NL	Natutal Language
NL2SQL	Natural Language to Structured Query Language
NLP	Natural Language Processing
NN	Neural Network
RDBMS	Relational Data Base Management System
RNN	Recurrent Neural Network
SEQ2SEQ	Sequence to Sequence
SQL	Structured Query Language
UI	User Interface

APPENDIX A. PREDICTIONS ON IMDB DATABASE

Table A.1: NLQs on IMDb tables

movie	
NLQ:	How much was the budget of ”Finding Nemo”
GT:	**SELECT** budget **WHERE** title = ”Finding Nemo”
PRED:	**SELECT** budget **WHERE** title = ”finding ””
NLQ:	Find all movies produced in 2015
GT:	**SELECT** title **WHERE** release year = 2015
PRED:	**SELECT** title **WHERE** title = 2015
director	
NLQ:	Find the naitonaliti of directro ”Woody Allen”
GT:	**SELECT** nationality **WHERE** name = ”Woody Allen”
PRED:	**SELECT** COUNT(birth city) **WHERE** name = ”woody allen”
NLQ:	Find all male directors from Greece
GT:	**SELECT** name **WHERE** nationality = Greece **AND** gender = male
PRED:	**SELECT** name **WHERE** nationality = greece

Table A.2: NLQs on IMDb tables

actor	
NLQ:	Find all actors from ”Los Angeles”
GT:	**SELECT** name **WHERE** birth city = ”Los Angeles”
PRED:	**SELECT** name **WHERE** birth city = ”los ”
NLQ:	Find all female actors from Austin
GT:	**SELECT** name **WHERE** birth city = Austin **AND** gender = female
PRED:	**SELECT** name **WHERE** birth city = austin
NLQ:	Find all actors from Austin that are female
GT:	**SELECT** name **WHERE** birth city = Austin **AND** gender = female
PRED:	**SELECT** name **WHERE** nationality = austin
NLQ:	Find all the actresses from Austin
GT:	**SELECT** name **WHERE** birth city = Austin **AND** gender = female
PRED:	**SELECT** name **WHERE** birth city = austin
NLQ:	Find all the women actors from Austin
GT:	**SELECT** name **WHERE** birth city = Austin **AND** gender = female
PRED:	**SELECT** name **WHERE** birth city = austin
NLQ:	What is the nationality of ”Kevin Spacey”
GT:	**SELECT** nationality **WHERE** name = ”Kevin Spacey”
PRED:	**SELECT** nationality **WHERE** name = ”kevin ””
NLQ:	Count all actors
GT:	**SELECT** COUNT(name)
PRED:	**SELECT** name
NLQ:	Count all male actors
GT:	**SELECT** COUNT(name) **WHERE** gender = male
PRED:	**SELECT** gender **WHERE** name = actors
NLQ:	How many female actors were born in ”New York City”
GT:	**SELECT** COUNT(name) **WHERE** birth city = ”New York City” **AND** gender = female
PRED:	**SELECT** COUNT(name) **WHERE** birth city = ”new york city”
NLQ:	Find all actors who were born in ”New York City” before 2000
GT:	**SELECT** name **WHERE** birth city = ”New York City” **AND** birth year < 2000
PRED:	**SELECT** name **WHERE** birth city = 2000new ”city” **AND** birth year < 2000
NLQ:	What is the birth place of actor ”Kevin Spacey”
GT:	**SELECT** birth city **WHERE** name = ”Kevin Spacey”
PRED:	**SELECT** birth city **WHERE** name = ”kevin ””
NLQ:	What is the sex of ”Angelina Jolie”
GT:	**SELECT** gender **WHERE** name = ”Angelina Jolie”
PRED:	**SELECT** gender **WHERE** name = ”angelina jolie”
NLQ:	Return the gendre of ”Angelina Jolie”
GT:	**SELECT** gender **WHERE** name = ”Angelina Jolie”
PRED:	**SELECT** birth year **WHERE** name = ”angelina jolie”
NLQ:	Return all Asian actors
GT:	**SELECT** name **WHERE** nationality = Asian
PRED:	**SELECT** birth city **WHERE** name = asian actors
NLQ:	Return all American actors
GT:	**SELECT** name **WHERE** nationality = American
PRED:	**SELECT** birth year **WHERE** gender = american
NLQ:	Where is ”Brad Pitt” from
GT:	**SELECT** nationality **WHERE** name = ”Brad Pitt”
PRED:	**SELECT** birth city **WHERE** name = ”brad pitt”

Table A.3: NLQs on IMDb views

actor_by_movie	
NLQ:	Find all moveis of ”Brad Pitt”
GT:	**SELECT** movie title **WHERE** actor name = ”Brad Pitt”
PRED:	**SELECT** cast role **WHERE** actor name = ”brad pitt
NLQ:	Find all movies of atcor ”Margot Robbie”
GT:	**SELECT** movie title **WHERE** actor name = ”Margot Robie”
PRED:	**SELECT** cast role **WHERE** movie title = ””margot robbie”
NLQ:	Find all roles in the movie ”My Cousin Vinny”
GT:	**SELECT** cast role **WHERE** movie title = ”My Cousin Vinny”
PRED:	**SELECT** cast role **WHERE** movie title = ”my cousin vinny”
NLQ:	What is the number of actors in the movie ”Grumpier Old Men”
GT:	**SELECT** COUNT(actor name) **WHERE** movie title = ”Grumpier Old Men”
PRED:	**SELECT** COUNT(cast role) **WHERE** movie title = ”old old men”
NLQ:	How many actors from China have acted in ”Rush Hour 3”
GT:	**SELECT** COUNT(actor name) **WHERE** actor nationality = China **AND** movie title = ”Rush Hour 3”
PRED:	**SELECT** COUNT(actor name) **WHERE** actor nationality = china
NLQ:	What is the number of movies in which Jennifer Aniston acted after 2010?
GT:	**SELECT** COUNT(movie title) **WHERE** actor name = Jennifer Aniston **AND** movie release year > 2010
PRED:	**SELECT** COUNT(movie title aka) **WHERE** actor name = jennifer **AND** movie release year > 2010
NLQ:	Find the actor who played Captain Miller in the movie ”Saving Private Ryan”
GT:	**SELECT** actor name **WHERE** cast role = Captain Miller **AND** movie title = ”Saving Private Ryan”
PRED:	**SELECT** actor birth year **WHERE** movie title aka = ”private ryan”
NLQ:	Who acted John Nash in the movie ”A Beautiful Mind”?
GT:	**SELECT** actor name **WHERE** cast role = John Nash **AND** movie title = ”A Beautiful Mind”
PRED:	**SELECT** actor name **WHERE** movie title = ”a beautiful mind”
NLQ:	Find all movies where ”Brad Pitt” act after 2002 and before 2010
GT:	**SELECT** movie title **WHERE** actor name = ”Brad Pitt” **AND** movie release year > 2002 **AND** movie release year < 2010
PRED:	**SELECT** actor birth city **WHERE** movie release year > 2010 **AND** actor name = brad pittpitt” **AND** actor birth year = 2010
NLQ:	Find all movies featuring ”Kate Winslet”
GT:	**SELECT** movie title **WHERE** actor name = ”Kate Winslet”
PRED:	**SELECT** cast role **WHERE** movie title = ”kate movies movies
NLQ:	Find all movies in which ”Robin Wright” appears
GT:	**SELECT** movie title **WHERE** actor name = ”Robin Wright”
PRED:	**SELECT** movie title aka **WHERE** actor name = ”robin wright”
NLQ:	Return all movies of ”Brad Prit”
GT:	**SELECT** movie title **WHERE** actor name = ”Brad Pitt”
PRED:	**SELECT** actor birth year **WHERE** movie title = brad brad
NLQ:	Find all females in the cast of ”Grumpier Old Men”
GT:	**SELECT** actor name **WHERE** movie title = ”Grumpier Old Men” **AND** actor gender = female
PRED:	**SELECT** movie title aka **WHERE** movie title = ”old old men”

Table A.4: NLQs on IMDb views

	actor_by_tv_series
NLQ:	Who acts as ”Olivia Pope” in the series Scandal
GT:	**SELECT** actor name **WHERE** cast role = ”Olivia Pope”
	AND tv series title = Scandal
PRED:	**SELECT** actor name **WHERE** tv series title = ”olivia pope””
	movie_by_company
NLQ:	Find all movies by Netflix
GT:	**SELECT** movie title **WHERE** company name = Netflix
PRED:	**SELECT** movie title aka **WHERE** company country code = movies
NLQ:	How many movies did Netflix produce?
GT:	**SELECT** COUNT(movie title) **WHERE** company name = Netflix
PRED:	**SELECT** COUNT(company country code)
NLQ:	Find all movies produced by Netflix after 2009
GT:	**SELECT** movie title **WHERE** company name = Netflix
	AND movie release year > 2009
PRED:	**SELECT** movie title aka **WHERE** movie release year > 2009
	AND company name = netflix
NLQ:	Find all movies produced by Netflix before 2015
GT:	**SELECT** movie title **WHERE** company name = Netflix
	AND movie release year < 2015
PRED:	**SELECT** movie title aka **WHERE** movie release year = 2015
	AND company country code = 2015
NLQ:	Find the movies produced by Neftlix
GT:	**SELECT** movie title **WHERE** company name = Netflix
PRED:	**SELECT** company country code **WHERE** company country code = movies
	movie_by_director
NLQ:	Find all movies directed by Steven Spielberg after 2006
GT:	**SELECT** movie title **WHERE** director name = Steven Spielberg
	AND movie release year > 2006
PRED:	**SELECT** director name **WHERE** director name = steven
	AND director birth year > steven
NLQ:	Who is the director of the movie ”Catch me if you can”?
GT:	**SELECT** director name **WHERE** movie title = ”Catch me if you can”
PRED:	**SELECT** director name **WHERE** movie title = catch me if you can
NLQ:	Who was the director of the movie Joy from 2015?
GT:	**SELECT** director name **WHERE** movie title = Joy
	AND movie release year = 2015
PRED:	**SELECT** director name **WHERE** director name = joy
	AND director nationality = joy
NLQ:	How many movies did Alfred Hitchcock direct?
GT:	**SELECT** COUNT(movie title) **WHERE** director name = Alfred Hitchcock
PRED:	**SELECT** COUNT(director gender) **WHERE** director name = alfred
NLQ:	Find all movies directed by ”Steven Spielberg” after 2006
GT:	**SELECT** COUNT(movie title) **WHERE** director name = ”Steven Spielberg”
	AND movie release year > 2006
PRED:	**SELECT** director name **WHERE** director name = ”steven ””
	AND director birth year > 2006
NLQ:	Find all films directed by ”Woody Allen”
GT:	**SELECT** COUNT(movie title) **WHERE** director name = ”Woody Allen”
PRED:	**SELECT** movie budget **WHERE** director name = ”woody allen”

Table A.5: NLQs on IMDb tables and views where a prediction is not possible

movie	
NLQ:	Find about ”star wars”
PRED:	**SELECT** title aka **WHERE** title = ”star wars”
NLQ:	What is ”star wars”
PRED:	**SELECT** title aka **WHERE** title = ”star wars”
NLQ:	Find about ”true crime”
PRED:	**SELECT** budget **WHERE** title = ”true crime”
NLQ:	Find the most expensive movie
PRED:	**SELECT** MAX(budget)
NLQ:	Find the decade with the most movies produced
PRED:	**SELECT** MAX(release year) **WHERE** budget = movies
director	
NLQ:	What is the most prolific director
PRED:	**SELECT** MAX(name)
actor	
NLQ:	Find about ”Margot Robbie”
PRED:	**SELECT** birth year **WHERE** name = ”margot robbie”
tv_series	
NLQ:	Which is the most long lasting tv series
PRED:	**SELECT** MAX(release year) **WHERE** num of episodes =
producer	
NLQ:	Find about ”Walt Disney”
PRED:	**SELECT** birth year **WHERE** name = ”walt disney”
NLQ:	Who is ”Walt Disney”
PRED:	**SELECT** name **WHERE** name = ”walt disney”
actor_by_tv_series	
NLQ:	Is ”Fred Mendes” presenting the News
PRED:	**SELECT** actor birth year **WHERE** actor name = ”fred mendes”
NLQ:	Is ”Mendes Fred” presenting the News
PRED:	**SELECT** actor birth year **WHERE** actor name = ”mendes fred””
actor_by_movie	
NLQ:	Is ”Sharp Douglas” playing in ”Power With-out Glory”
PRED:	**SELECT** MIN(actor birth year) **WHERE** actor name = ”sharp douglas
NLQ:	Is ”Tom Cruise” playing in ”American Made”
PRED:	**SELECT** actor birth year **WHERE** movie title = ”tom ””
NLQ:	Find all actors who acted in the same movie as Tom Hanks
PRED:	**SELECT** actor name **WHERE** movie title = tom in
NLQ:	Find the actors that have not played in the same movie with ”Brad Pitt”
PRED:	**SELECT** actor name **WHERE** actor name = ”brad pitt”
NLQ:	What is the latest movie by ”Jim Jarmusch”
PRED:	**SELECT** MAX(movie title aka) **WHERE** actor name = ”jim jarmusch”
NLQ:	List the movies by ”Woody Allen” sorted by cast size
PRED:	**SELECT** movie title aka **WHERE** actor name = woody woody allen”
NLQ:	Find all movies that star both Angelina Jolie and Brad Pitt
PRED:	**SELECT** cast role **WHERE** actor name = angelina jolie
movie_by_director	
NLQ:	Find the directors that have not directed more than 10 movies
PRED:	**SELECT** director name **WHERE** director name = not directed

BIBLIOGRAPHY

[1] Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. Natural language interfaces to databases - an introduction. *CoRR*, cmp-lg/9503016, 1995.

[2] Theofilos Belmpas, Orest Gkini, and Georgia Koutrika. Analysis of database search systems with thor. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2681–2684, Richland, SC, 2020. International Foundation for Autonomous Agents and Multiagent Systems.

[3] E.F. Codd. Seven steps to rendezvous with the casual user. *Data Base Management*, pages 179–199, 01 1974.

[4] Kedar Dhamdhere, Kevin S. McCurley, Ralfi Nahmias, Mukund Sundararajan, and Qiqi Yan. Analyza: Exploring data with conversation. In *Proceedings of the 22Nd International Conference on Intelligent User Interfaces*, IUI '17, pages 493–504, New York, NY, USA, 2017. ACM.

[5] Li Dong and Mirella Lapata. Language to logical form with neural attention. *CoRR*, abs/1601.01280, 2016.

[6] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. Learning a neural semantic parser from user feedback. *CoRR*, abs/1704.08760, 2017.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

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

[9] Percy Liang. Learning executable semantic parsers for natural language understanding. *Commun. ACM*, 59(9):68–76, August 2016.

[10] Rui Liu, Junjie Hu, Wei Wei, Zi Yang, and Eric Nyberg. Structural embedding of syntactic trees for machine comprehension, 2017.

[11] 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.

[12] M. Marneffe, B. Maccartney, and C. Manning. Generating typed dependency parses from phrase structure parses. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC-2006)*, Genoa, Italy, May 2006. European Language Resources Association (ELRA). ACL Anthology Identifier: L06-1260.

[13] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. *CoRR*, abs/1508.00305, 2015.

[14] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[15] Peter Norvig. How to write a spelling corrector, 2007. [Online; accessed 20-February-2020].

[16] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R. Mittal, and Fatma Özcan. Athena: An ontology-driven system for natural language querying over relational data stores. *Proc. VLDB Endow.*, 9(12):1209–1220, August 2016.

[17] Lappoon R. Tang and Raymond 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.

[18] Xiaojun Xu, Chang Liu, and Dawn Song. Sqlnet: Generating structured queries from natural language without reinforcement learning. *CoRR*, abs/1711.04436, 2017.

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

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