NLIDB-Bench: A Benchmark for Evaluating Natural Language Interfaces to Relational Databases

Kanchan Chowdhury
Arizona State University
Tempe, USA
kchowdh1@asu.edu

Vamsi Meduri
Arizona State University
Tempe, USA
vmeduri@asu.edu

Mohamed Sarwat
Arizona State University
Tempe, USA
msarwat@asu.edu

***Abstract*—The task of building a natural language interface to a database (*aka.* NLIDB) has recently gained significant attention from both the database and Natural Language Processing (NLP) communities. There have been many research works that study the problem of automatically generating SQL queries from natural language questions (*abbr.* NL-2-SQL). In this paper, we introduce NLIDB-Bench, which is a unified benchmark framework that researchers and practitioners can use to evaluate, compare and contrast various NL-2-SQL approaches. We formalize a comprehensive set of benchmark metrics in the proposed framework to evaluate the effectiveness, efficiency and scalability of state-of-the-art NL-2-SQL approaches. Furthermore, we provide an extensive experimental evaluation of four popular NL-2-SQL approaches using NLIDB-Bench. We compare such approaches on four existing datasets. We also evaluate such approaches using our newly developed dataset that consists of a large repository of diverse NL-2-SQL questions, which are extracted from real user interactions with the popular Yelp geolocation database. Based on the experiments, the studied neural network based techniques, despite their need for a large training dataset, show more promising performance in translating natural language questions to SQL queries compared to syntax/grammar-based and semantic parsing techniques. The experiments also show that the accuracy of the studied NL-2-SQL approaches significantly varies based on the semantics and structure of the underlying database. At the end, we present some open problems and propose future research directions to expand the field of natural language interfaces to relational databases.**

I. Introduction

Natural Language Interfaces to Databases, shortly known as NLIDBs, allow SQL-agnostic users who lack database expertise to issue queries to the database in natural language such as English. Given such a natural language query, an NLIDB system automatically generates the corresponding database query [1] and passes it to the database for execution. Because of the inherent complexity in translating natural language questions to SQL queries (*aka.* NL-2-SQL), the task of building an efficient NLIDB is still an open research problem. We also call these NLIDB systems NL-2-SQL approaches. In light of several recent works which attempt to solve this problem [2], [3], [4], we propose a comprehensive benchmark, $NLIDB\text{-}Bench$, that evaluates the effectiveness and scalability of the state-of-the-art NLIDB systems.

A. Motivation

We live in an era where conversational agents are gaining more prominence over static interfaces requiring specialized language expertise to query and retrieve data. Virtual agents with conversational ability such as Amazon Alexa and Google Assistant have become increasingly popular. While chatbots are used to give automatic response to user queries written in natural language, Amazon Alexa and Google Assistant allow users to communicate through natural language speech. In both the cases, these technologies need to understand the natural language query provided by the user and fetch the query answer from their data storage system. In order to use such technologies on systems where data is stored in relational databases, these automated question answering systems should be ale to translate natural language questions to SQL queries, which emphasizes the need for NLIDBs.

However, the complex structure of SQL queries coupled with the inherent ambiguity of natural language (NL) questions makes building an NLIDB system a challenging task. The difficulty in interpreting the correct semantic meaning of NL questions often results in the wrong synthesis of SQL queries. A small difference in the structure of a NL question might result in a huge difference in the semantic meaning of the question. Let us consider two NL queries for the relation *business* in Figure 1: (1) *What is number of businesses of all ratings in Nevada* and (2) *What is the total number of businesses in Nevada for each rating*. Although these two NL queries are almost similar, the formation of SQL query for the second one requires a GROUP BY operation whereas the SQL query corresponding to the first one does not have any GROUP BY operation. Besides this kind of syntactic and semantic ambiguities of NL questions, use of phrases and system specific terms makes the interpretation of NL questions much more difficult. The semantic meaning of a word varies based on the context in which it is used.

Having said that, designing an efficient NLIDB has become an open and a challenging research problem, which has sparked a significant amount of interest among AI and database researchers. As a consequence, a number of NLIDBs have been built recently, which include but are not limited to SQLizer [5], PRECISE [6]. Although many of these systems exhibit good initial steps to translate NL questions to SQL queries (i.e., NL-2-SQL), the approaches used to empirically evaluate such ideas are not thorough enough. This motivates the need to build a unified benchmark that can be utilized to comprehensively evaluate NL-2-SQL approaches.

business_id	name	city	state	rating	review_count
1	Texas de Brazil	Dallas	Texas	4.0	1008
2	Yeti Helpers	Henderson	Nevada	4.5	11
3	Best Buy	Madison	Wisconsin	2.5	24
4	Jasmine	Las Vegas	Nevada	3.5	274
5	Base Exchange	Nellis AFB	Nevada	3.0	31

(a) Part of the business relation on Yelp Database

NL Question: What are the names and average ratings of each business in Nevada with at least 31 reviews?

Ground Truth SQL: SELECT name, rating FROM business WHERE state = 'Nevada' AND review_count >= 31

SQL1: SELECT name, rating FROM business WHERE review_count >= 31 AND state = 'Nevada' ✓

SQL2: SELECT rating FROM business WHERE state = 'Nevada' AND review_count > 31 ✗

SQL3: SELECT rating, city FROM business WHERE state = 'Nevada' AND review_count > 31 ✗

SQL4: SELECT city FROM business WHERE state = 'Nevada' AND review_count >= 31 ✗

(b) An example NL question with ground truth SQL query and 4 SQL queries translated by NLIDBs

Ground Truth SQL Answer

name	rating
Jasmine	3.5
Base Exchange	3

SQL1 Answer

name	rating
Jasmine	3.5
Base Exchange	3

SQL2 Answer

rating
3

SQL3 Answer

rating	city
3.5	Las Vegas

SQL4 Answer

city
Las Vegas
Nellis AFB

(c) Output Relations for SQL Queries

Fig. 1: A sample of the Business relation in Yelp, a sample natural language question with the ground truth SQL query, and 4 queries translated by various NL-2-SQL approaches as well as the output relations for such queries.

B. Limitations of Existing Work

Existing surveys and benchmarks to study the effectiveness of NL-2-SQL approaches have the following limitations.

Dated Surveys: Existing surveys such as [7] and [8] compare and contrast various natural language interfaces to databases until the year 1989 and 1995, respectively. Such surveys also compare NLIDBs to formal query languages, form-based interfaces, and graphical interfaces. More recent surveys such as [9] and [10] also give an overview of natural language interfaces to databases. They also discuss various components of the existing NLIDB systems and the system architecture of various approaches. However, such surveys: (1) are outdated, and recent approaches are not compared in these surveys. and/or (2) only provide a literature overview of the NL-2-SQL approaches but not provide an experimental benchmark to empirically evaluate those approaches.

Incomprehensive study: Besides surveys, a benchmark named Spider [11] evaluates NL-2-SQL approaches. It does not provide a detailed evaluation of the wide variety of existing NL-2-SQL generation algorithms. Spider [11] can only be used to evaluate neural network based approaches where neural network models proposed by an algorithm are trained using a training dataset after which the trained models are evaluated on a test dataset. Syntax or semantic parsing based approaches such as SQLizer [5], PRECISE [6], ATHENA [12], and NaLIR [13] cannot be evaluated using the Spider benchmark. Another limitation of Spider is that it evaluates the algorithms based only on a limited set of evaluation metrics. For instance, the translated queries should be fully accurate, and partially accurate NL-2-SQL translation does not get any partial credit according to those metrics.

Insufficient datasets: Existing benchmarks lack in the diversity of datasets used for evaluation of NL-2-SQL approaches. Spider evaluates neural network based approaches using the Spider dataset alone, whereas some NL-2-SQL approaches reported their evaluation results on a different set of datasets. Our empirical evaluation shows that the accuracy of existing NL-2-SQL approaches varies significantly depending on the underlying dataset. Different datasets are extracted from various application domains, which may significantly affect the semantics of NL-2-SQL translation.

Lack of scalability evaluation: Scalability of an NL-2-SQL approach is a crucial factor in determining whether or not it is useful in building a large-scale real world system. For instance, the time required to translate a NL question into corresponding SQL query decides its suitability for a real time application. Another important scalability factor is the time consumed by an NL-2-SQL technique for preprocessing (training a deep learning model for instance) before it can synthesize SQL query. The available benchmark as well as existing experimental analysis presented in recent papers almost ignore the scalability of such techniques.

Inspired by these limitations of the existing surveys and benchmarks, we propose our benchmark, NLIDB-Bench, that provides an exhaustive comparison of all the recent state-of-the-art algorithms for generating SQL queries from NL questions using five different datasets and a number of evaluation metrics mitigating the lack of scalability evaluation.

C. Contributions

The goal of this paper is to introduce NLIDB-Bench, which is a unified benchmark framework that researchers and practitioners can use to comprehensively evaluate, compare and contrast various NL-2-SQL approaches. The main contributions of NLIDB-Bench are summarized as follows:

- The generic design and implementation of NLIDB-Bench allows it to work for myriad categories of NL-2-SQL approaches including syntax/grammar based, semantic parsing based, and neural network based approaches. This is in contrast to the Spider benchmark, which is inherently designed to evaluate only neural network based approaches.
- NLIDB employs a comprehensive set of benchmark metrics that evaluate the accuracy of translating NL questions to SQL queries. As opposed to Spider and existing experimental evaluations, NLIDB-Bench not only tests the exact

(binary) matching of ground truth SQL queries or execution results. Yet, it also introduces two main metrics, i.e., Query Similarity and Execution Result Affinity (see Section III-B2 & III-B4), which measure the degree to which a certain NL-2-SQL approach gets close to the ground truth.
- To the authors' knowledge, NLIDB-Bench is the first benchmark that evaluates the scalability of NL-2-SQL approaches. To achieve that, the benchmark introduces two generic evaluation metrics that fits all evaluated NL-2-SQL categories: (1) Preprocessing time, which calculates the total time an NL-2-SQL technique takes to preprocess the database and build necessary models offline. (2) NL-2-SQL translation time: which calculates the time a given technique takes to translate NL question to SQL query online.
- An extensive experimental evaluation of four NL-2-SQL approaches using NLIDB-Bench. We compare such approaches on four existing datasets. We also evaluate such approaches using our newly developed NL-2-SQL dataset that consists of a large repository of diverse NL/SQL questions, which are extracted from real user interacions with the popular Yelp database.

II. NL-2-SQL: BACKGROUND

NL-2-SQL approaches take as input metadata information of a database and a natural language (NL) question and outputs the corresponding SQL query of the NL question. In case of Figure 1, the inputs to an NLIDB are the relation shown in Figure 1a and the NL question shown at the beginning of Figure 1b. The goal of the NLIDB is to synthesize either ground truth SQL or SQL1 in Figure 1b. In this section, we discuss state-of-the-art approaches that convert NL questions into SQL queries. We classify the state-of-the-art NL-2-SQL approaches based on their adopted methodologies. Primarily, all the existing approaches of translating SQL queries from NL questions can be classified into four basic categories - keyword based approaches, syntax/grammar based approaches, semantic parsing based approaches, and neural network based approaches.

A. Keyword Based Approaches

Keyword based approaches do not generate SQL queries for fully structured NL queries. Instead, they work on a bag of keywords from an NL query. These approaches generate an inverted index on the metadata obtained from the database schema which helps in finding the appropriate tables containing the information in the NL query. Thus, they map the keywords from the natural language query into table or column names using the inverted index. Once the inverted indexes are built, primary and foreign key relationships among various tables are used to construct the database query.

Aqqu [14] executes NL queries upon a knowledge base. It finds those entities in the knowledge base that match to the keywords in the NL query. Subsequently it applies template matching and relation matching to convert the NL question into a SPARQL query. SODA [15] generates SQL queries for business users using a graph pattern matching algorithm. Another approach in this category includes Precis [16].

B. Syntax / Grammar Based Approaches

Syntax/grammar based approaches mainly focus on the syntactic structure of the NL question and the SQL query. In most of the cases, an NL query is parsed based on its syntactic structure, and a parse (syntax) tree is generated. The parse tree nodes are mapped to SQL nodes for the purpose of generating a SQL query. The syntax structure captured by the parse tree plays a significant role in generating the corresponding SQL counterpart for the NL query.

NaLIR [13] represents the dependency relationship between NL tokens with a parse tree after parsing the NL question. Then, the parse tree is converted to a query tree by replacing the NL tokens with the database tokens. Later, the query tree is used for synthesizing the SQL query. Another approach in this category is [17] which focuses on the syntactic dependencies between NL query and metadata from the database schema. Typed dependency relation is used to represent textual relationship of the NL question with the schema. Later, this approach was extended to improve the accuracy of a generative parser [18] introducing a preference re-ranking technique [19], [20] applied on the set of possible queries. Another approach from [21] also falls under the syntax parsing category. A major limitation of all these approaches is that they can only construct SQL queries consisting of SELECT, FROM, and WHERE clauses.

C. Semantic Parsing Based Approaches

Semantic parsing based approaches exploit the semantic meaning of an NL question and find a semantic interpretation of the same. Later, that semantic interpretation is mapped to SQL query by the use of grammars or rules. Sometimes, semantic grammars are designed for a domain-specific system, and the developed system cannot be applied to other domains. In spite of this, recent approaches tried to make their system domain independent.

PRECISE [6] finds valid semantic interpretations of all the tokens after tokenizing the NL question. If at least one valid semantic interpretation for a question is possible, SQL queries are generated from such interpretations. [22] reports the effectiveness of using a statistical parser as a pluggable parser in PRECISE [6]. Another important approach in this category is SQLizer [5] which uses a sketch repairing mechanism for translating NL questions to SQL queries. First, a semantic parser generates a query sketch for the NL query. A confidence score below a certain threshold sends the sketch to a sketch repairing algorithm. Once the confidence score for a sketch exceeds the threshold, the placeholders in the sketch are filled up to generate the SQL query. GUSP [23] proposes an unsupervised semantic parsing approach to map NL questions to SQL queries. [24] shows a feasibility study on automatically acquiring the required semantic and linguistic resources for the NL components using the database metadata and data content, a domain-specific ontology and a corpus of associated text documents.

D. Neural Network Based Approaches

They are also known as neural machine translation approaches. The basic intuition behind these approaches is to build one or more deep neural network models that can be trained offline with pairs of NL questions and their SQL counterparts. Later, the system is evaluated on a test set of NL and SQL pairs.

Among other approaches in this category, Seq2SQL [2] uses reinforcement learning to map NL questions to SQL queries. [25] proposes a user feedback based approach to learn neural semantic models. SQLNet [3] proposes a sketch based solution where a fixed sketch for all the SQL queries is defined, and neural network models are used to complete the sketch. TypeSQL [1] extends SQLNet [3] by considering types of NL tokens which include but are not limited to dates, years, columns, person names, place names. SyntaxSQLNet [26] uses a SQL specific syntax tree based decoder instead of SQL query sketch. [27] proposes a bidirectional attention based neural network model while [28] proposes an encoder-decoder framework to translate NL questions to SQL queries. DialSQL [4] proposes a dialogue based approach for structured query generation, and [29] proposes a syntax and table aware SQL generation approach. Other approaches [30], [31] propose a syntactic structural kernel in order to map NL questions to SQL queries.

III. NLIDB-Bench Overview

The purpose of our benchmark is to categorize the existing state-of-the-art SQL generation approaches based on their adopted algorithms and to show a comparative analysis of each category. In this section, we start with shortly describing the adopted methodologies by four different NL-2-SQL approaches. After that, we explain various evaluation metrics that are helpful to evaluate the performance of existing NL-2-SQL approaches. Next, we describe some datasets that we use for our experiments. At the end of this section, we shortly describe how our benchmark works.

A. Compared NL-2-SQL Approaches

NLIDB-Bench evaluates four promising state-of-the-art NL-2-SQL approaches. In this section, we briefly describe adopted methodologies of these approaches. Approaches that we evaluate using NLIDB-Bench include NaLIR [13], PRECISE [6], TypeSQL [1], and SyntaxSQLNet [26].

It should be noted that not all SQL features are supported by all SQL generation approaches. These SQL features include SELECT, WHERE, GROUP BY, HAVING, and ORDER BY clauses, aggregation operation, join operation, and nested queries. Again, aggregate operations include COUNT, SUM, AVG, MIN, and MAX. Precise supports only COUNT while other approaches support all aggregate operations. Table I shows a comparison of SQL features supported by four approaches evaluated using NLIDB-Bench.

Before illustrating four NL-2-SQL approaches evaluated by NLIDB-Bench, we want to highlight some factors that play an important role in translating NL questions to SQL queries.

	SyntaxSQL	TypeSQL	Precise	NaLIR
SELECT Clause	Yes	Yes	Yes	Yes
Aggregate Operation	Yes	Yes	COUNT	Yes
JOIN Operation	Yes	No	Yes	Yes
WHERE Clause	Yes	Yes	Yes	Yes
GROUP BY Clause	Yes	No	No	Yes
HAVING Clause	Yes	No	No	No
ORDER BY Clause	Yes	No	No	No
Nested Queries	Yes	No	No	Yes

TABLE I: SQL features supported by various approaches

	SyntaxSQL	TypeSQL	Precise	NaLIR
Utilize DB Schema	Yes	Yes	Yes	Yes
Utilize DB Content	No	Yes	Yes	Yes
Question-SQL Pairs	Yes	Yes	No	No
Human Correction	No	No	No	Yes
Utilize Query Sketch	No	Yes	No	No

TABLE II: Factors utilized by various approaches

Those factors include but are not limited to utilizing database schema, utilizing database content, availability of pairs of questions and SQL queries, utilizing human in the loop, and utilizing query sketch. Utilizing database schema means taking into account the metadata information of the database such as table and column names when mapping NL questions to database queries. Utilizing database content or looking at values of various attributes in the database is another important factor that helps improve the SQL translation accuracy. A particular NL token might not match with any of the table or column names, but presence of that token as an attribute value in the data helps in mapping that token to corresponding database token. Availability of a large number of pairs of questions and SQL queries plays a significant role in neural network based approaches, because these approaches need to train neural network models with a large set of these pairs. Utilizing correction by human means if an approach finds any ambiguity while mapping NL tokens into database tokens, it interacts with the user to resolve the ambiguity. Some other approaches utilize a query sketch concept. A query sketch is a SQL query containing some holes in it, and those holes should be filled up with correct database elements. Table II shows which of these factors are utilized by which approaches. The methodologies of all four NL-2-SQL approaches that we evaluate using NLIDB-Bench are illustrated here.

1) NaLIR:

Among all the categories we explain in Section II, NaLIR [13] is one of the most promising approaches among syntax parsing based approaches for generating SQL queries. We experimentally evaluate this approach as a representative of syntax parsing based approaches. The most basic system architecture for syntax parsing based approaches is shown in 2. Interactive communication with the user during different steps of SQL generation is also an important part of NaLIR [13]. First, the NL question given by the user is parsed using a dependency parser, and a parse tree is generated. Stanford Parser [32] is used here as a dependency parser. The generated parse tree represents the dependency relationships between two words

in the NL question. Then, a mapper maps the parse tree nodes to appropriate SQL components. If the mapper fails to map a parse tree node to any SQL component, it seeks help from the user through the interactive communicator. On the other hand, if a parse tree node is mapped to multiple SQL components, all mappings are presented to the user and parse tree structure is readjusted based on the user choice. Once all the parse tree nodes are mapped to database components, the new tree is called query tree. In the last step, a query tree translator generates a SQL query exploiting the structure of the query tree. This approach can translate SQL queries consisting of GROUP BY clause along with SELECT, FROM, and WHERE clauses. It cannot form SQL queries having nested queries.

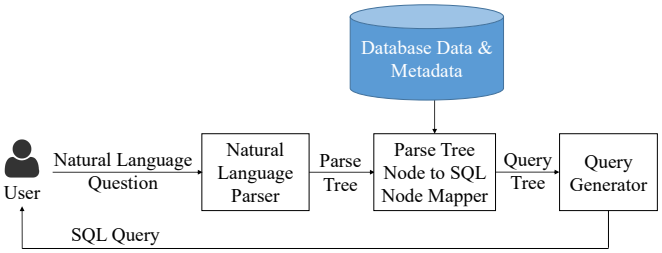


Fig. 2: Basic System Architecture of NaLIR

2) PRECISE:

PRECISE [6] is one of few significant approaches in the category of semantic parsing based approaches. PRECISE converts an NL question to a SQL query depending on the semantic tractability of the question. If an NL question is semantically tractable, PRECISE can map that question to corresponding SQL query. If the tokenization of a sentence is such that all tokens are distinct and there exists at least one value token contains a wh-value, that sentence is called semantically tractable question. Please refer to the original paper [6] for details explanation of semantically tractable question. SQL generation methodology adopted by PRECISE

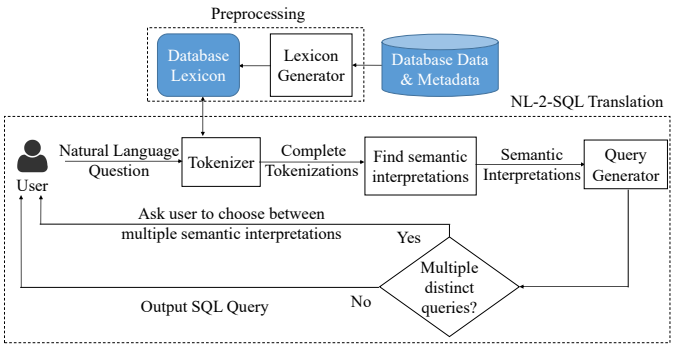


Fig. 3: Basic System Architecture of PRECISE

is shown in Figure 3. Given an NL question as input, tokenizer generates all possible complete tokenizations with the help of database lexicon. Once all complete tokenizations are generated, a matcher finds the semantic interpretation of natural language tokens. If any valid semantic interpretation is found and the question is semantically tractable, query generator takes semantic interpretations and forms a SQL query. An equivalence checker checks whether it is possible to find multiple distinct SQL queries for the same NL question and asks the user to choose among multiple semantic interpretations if multiple queries exist. This approach can translate SQL queries consisting of SELECT, FROM, and WHERE clauses.

3) TypeSQL:

Among all the approaches in the neural network based category, TypeSQL [1] is one of the few approaches that showed high efficiency in translating NL questions to SQL queries. TypeSQL considers NL-2-SQL synthesis problem as a sketch filling problem. It assumes a query sketch consisting of SELECT, FROM, and WHERE clauses. The goal is to predict the columns, relations, and conditions for SELECT, FROM, and WHERE clauses respectively. TypeSQL first preprocesses question inputs by type recognition. After tokenizing each question, it checks whether a token is INTEGER, FLOAT, DATE, and YEAR. It also searches for five types of entities: PERSON, PLACE, COUNTRY, ORGANIZATION, and SPORT on Freebase. Then, two bi-directional LSTMs are used to encode words in the question with their types and the column names separately. TypeSQL consists of three models: model for predicting column in the SELECT clause and columns in the conditions of WHERE clause, model for predicting aggregation operations, and model for predicting operators and values in the conditions of WHERE clause. These models are trained using a large set of question and SQL pairs. Later, these models are used to predict SQL queries for NL questions. Tthis approach can generate only SELECT and WHERE clauses of a SQL query.

4) SyntaxSQLNet:

Another promising approach in the neural network based category is SyntaxSQLNet [26]. Instead of query sketch, it uses SQL specific syntax tree based decoder with SQL path history. The decoder proposed by the model is a collection of recursive modules. In order to predict different SQL components, the decoding process consists of nine modules. Each module is responsible for predicting a specific component of SQL query. A fixed SQL grammar is followed to determine which module to invoke at each decoding step. In order to understand how SQL grammar and SQL path history help in invoking the right modules in various decoding steps, let us consider some simple examples. If the previously invoked module is a GROUP BY module and the current token is a column name, then the next module to call is the HAVING module. While most of the approaches in this category can predict only SELECT, FROM, and WHERE clauses, this approach can predict GROUP BY, HAVING, and ORDER BY clauses as well as nested queries. This approach also supports union, except and interaction operations. Figure 4 shows the basic system architecture of neural network based approaches that is applicable to TypeSQL, SyntaxSQLNet, and other neural network based approaches.

B. Evaluation Metrics

Various types of evaluation metrics are used to measure the performance of NL-2-SQL approaches. In this section, we

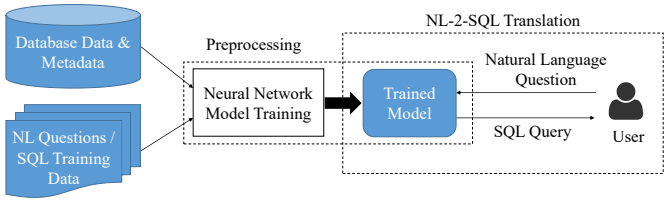


Fig. 4: Basic System Architecture of Neural Network Based Approaches

define all the evaluation metrics that we use in our experiments for the evaluation of NL-2-SQL approaches.

1) Exact Query Matching (EQM):

Exact query matching checks whether or not there is an exact string match between the predicted query and the ground truth query, but there is an ordering issue for the conditions in the WHERE clause that needs to be considered while calculating this evaluation metric. Let us consider ground truth SQL and SQL1 in Figure 1: These two queries are similar and generate same results once executed against a relational database management system (RDBMS), but the order of conditions in the WHERE clause is different. Exact query matching criteria should consider these two queries as similar queries. In order to calculate exact query matching values, we first convert both ground truth query and predicted query into an intermediate representation where each query is split into different clauses such as SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY. Each clause from the predicted query should match with the corresponding clause from the ground truth query. The WHERE clause is further split into different conditions. The predicted query should contain all conditions available in the ground truth query without having any extra condition. This way, we count the total number of predicted queries that have exact match with their corresponding ground truth queries. In Figure 1, only SQL1 is similar to ground truth SQL query according to exact query matching criteria.

Accuracy, Precision, Recall and F1 Score: Let us assume that N_{total} is the total number of queries in the test dataset, $Correct_{EQM}$ is the number of queries for which intermediate representations of both predicted queries and ground truth queries are similar, $Wrong_{EQM}$ is the number of queries for which the translations are wrong, and $N_{not\text{-}translated}$ is the total number of NL questions for which an approach cannot generate any SQL query. Accuracy on exact query matching, $Accuracy_{EQM}$ is defined as:

$$Accuracy_{EQM} = \frac{Correct_{EQM}}{N_{total}}$$

There are some SQL generation approaches which try to translate each NL question into SQL query regardless of the correctness of the translated query. Again, some other approaches cannot convert all NL questions into SQL queries, but most of the translated queries by these approaches are correct. We can represent these behaviors of different approaches using precision, recall, and F1 score. In order to calculate precision, recall, and F1 score on exact query matching, we check the correctness of all translated queries using the criteria described in section III-B1.

Precision is the ratio of number of queries for which an algorithm generates correct SQL queries and the number of queries for which it attempts to generate SQL queries. Precision helps us to represent the percentage of correctness of only translated queries by a particular approach. Based on the assumptions defined in this section, precision can be represented with the below formula.

$$Precision_{EQM} = \frac{Correct_{EQM}}{Correct_{EQM} + Wrong_{EQM}}$$

On the other hand, recall is the ratio of number of queries for which an algorithm generates correct SQL queries to the total number of queries for which it either generates correct SQL queries or does not generate any SQL query. The high value of recall for an approach means that the approach generates SQL queries for almost all NL questions regardless of the correctness of the generated SQL query. We represent recall with the following formula.

$$Recall_{EQM} = \frac{Correct_{EQM}}{Correct_{EQM} + N_{not-translated}}$$

Although precision and recall are helpful in showing different aspects of efficiency of a system, none of these evaluation metrics alone can represent the overall SQL translation efficiency. An approach having very high precision might have low recall value, and the reverse can also happen. In order to show the balance between precision and recall, we use F1 score. F1 score is a function of precision and recall representing an weighted average of precision and recall. It is the harmonic mean of precision and recall.

2) Query Similarity:

Query Similarity indicates the fraction of predicted query string that is similar to ground truth query string. While exact query matching assigns binary 0 or 1 to a predicted query depending on its exact similarity to the ground truth query, query similarity assigns partial score to each predicted wrong query. In order to calculate query similarity, we split the SQL into multiple units. Again, unit names vary depending on the components of SQL query. For example, column names in the SELECT clause, table names in the FROM clause, conditions in the WHERE clause, and so on are considered as units. At first, we set query similarity score to 0 for a SQL query. For each unit in the ground truth query string, we check whether that unit is available in the predicted query string. If it is available, then query similarity score is increased by 1. At last, we divide the summed query similarity score by total number of units in the ground truth query string. Once we calculate query similarity score for each query similarly, we take the average of query similarity scores of all queries.

For SQL3 in Figure 1, the ground truth SQL consists of 5 units: two column names in the SELECT clause, one table name in the FROM clause, and two conditions in the WHERE clause. SQL3 has three units that are similar to

ground truth SQL. So, query similarity score for SQL3 is 0.6 or 60%. In order to verify which SQL clause is the most difficult SQL component to translate among three basic SQL components (SELECT, FROM, and WHERE clauses), we propose three more evaluation metrics which are columns similarity, relations similarity, and qualifications similarity.

Columns Similarity, Relations Similarity, and Qualifications Similarity: We can calculate these three evaluation metrics using same algorithm that we also use to calculate query similarity. The only change we need is the value of units in the inner for loop. Instead of considering all units of SQL query, we consider only units of SELECT, FROM, and WHERE clauses in order to calculate columns, relations, and qualifications similarity respectively. For SQL3, columns similarity is 50% since it contains one out of two columns in the ground truth query. Also, relations similarity is 100% and qualifications similarity is 50% for SQL3.

3) Exact Execution Result Matching (ERM):

Exact execution result matching checks whether there is an exact string match between the execution results of predicted query and that of ground truth query. If the execution results of two queries match with each other, then these two queries are considered similar. Looking at output relations in Figure 1, we can say that only SQL2 has exact execution result match with ground truth query SQL1 which is also same in case of exact query matching criteria, but it may not be same always. We need exact execution result matching in order to overcome some of the limitations raised by exact query matching accuracy. Let us consider two new queries on the business relation shown in Figure 1: *SELECT COUNT(business_id) FROM business WHERE state = Nevada AND review_count ¿= 31* and *SELECT COUNT(*) FROM business WHERE state = Nevada AND review_count ¿= 31*. Since business_id is a primary key, these two queries will generate same results. Although these two queries are similar, exact query matching criteria will consider these two queries as different queries. Exact execution result matching helps to overcome this type of problem.

Accuracy, Precision, Recall and F1 Score: We calculate accuracy, precision, recall, and F1 score in terms of ERM following the similar equations we used for calculating these metrics in terms of EQM. The only exception is that here we check the correctness of predicted queries based on exact execution result matching criteria described earlier in this subsection in contrast to the exact query matching criteria.

Exact execution result matching also fails to represent the correctness of SQL queries in some cases. There are some problems with exact execution result matching for which we need to rely on exact query matching along with exact execution result matching. To explain this problem, again consider a new query from the business relation in Figure 1: *SELECT COUNT(business_id) FROM business WHERE rating ¡ 3.5*. Assume that the predicted query is *SELECT COUNT(business_id) FROM business WHERE rating ¿ 3.5*. Since number of businesses with rating greater than 3.5 is similar to number of businesses with rating less than 3.5, exact execution result matching criteria will mark the predicted query as correct while exact query matching accuracy solves this problem.

4) Execution Results Affinity (ERA):

We showed in section III-B1 and section III-B3 that none of the exact query matching criteria and exact execution result matching criteria can fully represent the correctness of all predicted queries. For that reason, we propose another evaluation metric, execution result affinity, which checks how close the execution result of a predicted query is to the execution result of the ground truth query instead of checking whether execution results are fully similar or not.

Accuracy, Precision, Recall and F1 Score: For execution results affinity, we calculate accuracy, precision, recall and F1 score values based on true positive, false positive, and false negative values.

True Positive indicates the fraction of ground truth query execution results overlapping with the execution results of predicted queries. In order to calculate true positives, we look for those rows and columns in the execution result of a ground truth query that can also be found once corresponding predicted SQL query is executed against the database. Algorithm for calculating true positive value for all queries in a dataset is shown in Algorithm 1. For SQL3 in Figure 1, the execution result of ground truth query contains two columns and two values for each column. The first column in the ground truth execution result name is not in the SQL3 execution result. So, column score for first column is 0. The second column rating is available, and its column score is 0.5 since it contains 1 out of 2 values in the ground truth execution result. A total column score of 0.5 for 2 columns results in a true positive value of 0.25 for SQL3. In order to calculate true positive value for a set of queries in the dataset, we take the average of true positive values for all queries.

```
Result: Value of true positive
true positive = 0;
query count = 0;
for each query in the dataset do
    query score = 0;
    column count = 0;
    for each column in ground truth execution result do
        if column is in predicted execution result then
            column score = number of matching rows/number of rows in
              ground truth;
        else
            column score = 0;
        end
        query score = query score + column score;
        column count = column count + 1;
    end
    query score = query score/column count;
    true positive = true positive + query score;
    query count = query count + 1;
end
true positive = true positive/query count;
```

Algorithm 1: Calculating True Positives for Execution Results Affinity

False Positive is opposite to true positive in the sense that it indicates the fraction of predicted query execution results that is not available in the execution results of ground truths. We calculate false positive for a query by considering those rows and columns in the execution result of the predicted query

that are not available in the execution result of the ground truth query. We can calculate false positive by doing a little bit modification in Algorithm 1 which is used to calculate true positive. In the inner for loop, instead of iterating the loop for every column in the ground truth query execution result, we iterate the loop for every column in the predicted query execution result. For each column in the predicted query execution result, if it is also available in the ground truth query execution result, we set the column score to number of extra rows in prediction result divided by the number of total rows in prediction result. Otherwise, we set the column score to 1. All other steps are unchanged. The execution result of SQL3 in Figure 1 contains two columns and one value for each column. The first column in the execution result rating is also in the ground truth execution result, and it does not contain any extra rows. So, column score for first column is 0. If it contains 3 rows and 1 of them is not in ground truth execution result, then column score would be 0.33. The second column city is not in ground truth execution result, and its column score is 1. False positive value for SQL3 is 0.5 since total column score is 1 and there are 2 columns in its execution result.

False Negative is the fraction of ground truth query execution results that are not available in the execution results of predicted queries. False negative looks for those rows and columns in the execution result of a ground truth query that can not be found in the execution result of corresponding predicted SQL query. Like we did for false positives, we can calculate false negative by doing a little modification in Algorithm 1. In the inner for loop, if a column in the ground truth query execution result is available in the predicted query execution result, we divide the number of rows in ground truth result which are not in predicted result by the number of total rows in ground truth result and set the column score to the quotient. Otherwise, we set the column score to 1. All other steps in Algorithm 1 are unchanged. For SQL3 in Figure 1, the first column in the ground truth execution result name is not in the SQL3 execution result. So, column score for first column is 1. The second column rating is available, and its column score is 0.5 since it misses 1 row out of 2 rows in the ground truth execution result. Value of false negative for SQL3 is 0.75 since total column score is 1.5, and there are a total of 2 columns in the ground truth execution result.

After calculating true positive, false positive, and false negative for all queries in the dataset, we use these values to calculate accuracy, precision, recall, and F1 score. For precision, recall, and F1 score, we use general formulas for calculating these three metrics based on true positive, false positive, and false negative. For calculating accuracy, we change the general formula as we do not have true negative in this scenario. If true positive, false positive, and false negative are represented by $\text{Positive}_{\text{true}}$, $\text{Positive}_{\text{false}}$, and $\text{Negative}_{\text{false}}$ respectively. Accuracy in terms of ERA is defined as:

$$Accuracy_{ERA} = \frac{Positive_{true}}{Positive_{true} + Positive_{false} + Negative_{false}}$$

5) Query Translation Time:
The Query translation time for an NL question indicates the time that is required to translate that NL question to its corresponding SQL query. The query translation time is averaged over all the NL questions and reported. Query translation time represents how fast NL questions can be translated to SQL queries. This evaluation metric is important because it shows the scalabilities of NL-2-SQL approaches. Query translation time also indicates whether organizations can apply an NL-2-SQL approach real-time.

6) Preprocessing Time:
Preprocessing time is the time that an NL-2-SQL approach takes before it becomes ready to translate NL questions to SQL queries. The preprocessing time varies depending on the type of preprocessing required by an NL-2-SQL approach. For example, some of the approaches use neural network model to translate NL questions to SQL queries. The main preprocessing time for these approaches involves the time required to train the model. These approaches also require database metadata, test and training data to be converted to JSON files of particular format. Another approach, PRECISE [6] requires the database metadata to be converted into a lexicon. Preparing the lexicon is the preprocessing for this approach. NaLIR [13] reads database metadata from two JSON files and does not require preprocessing.

C. Datasets

Various public datasets are available to evaluate NL-2-SQL approaches such as WikiSQL [33], Spider [34], GEO-Queries [35], and Yelp which have been used by different state-of-the-art algorithms. Since different approaches used different datasets and all datasets vary from each other depending on query complexity and SQL features, we evaluate approaches from various categories using all of the mentioned datasets. It helps to experiment how different approaches are performing on the same dataset. Our experimental results show that none of the approaches performs similarly on every dataset. In this section, we provide a short overview on all these datasets.

WikiSQL dataset was published by [2]. This dataset is the largest dataset among all available datasets. The size of this dataset is helpful to some approaches which use neural network models, because a large set of input output pairs is required to train a neural network model. In spite of this advantage, the problem with this dataset is that the queries of this dataset are not diverse in terms of SQL features. Each query in this dataset contains information from only one table because of which no join operation is required among different tables. The queries do not contain any GROUP BY, HAVING, or ORDER BY operations. This dataset also contains some column names such that it is not possible to estimate which type of information those columns contain by looking at those column names. This dataset provides database metadata using some JSON files, and it is not possible to generate some databases since some column names of these databases contain special characters or SQL keywords. To evaluate some of the

approaches, we require database files for which we changed some of the column names in the dataset to avoid special characters and SQL keywords and also to make them more meaningful.

Spider dataset was offered by [11]. Although this dataset is a bit smaller than the WikiSQL datset, it is more complex and has diverse SQL features. This dataset provides the database files along with JSON files containing database metadata. The SQL queries in this dataset contain join operation, nested queries, GROUP BY, HAVING, and ORDER BY clauses. The size of this dataset is such that it can also be used by neural network based approaches to train the models.

GEOQuery dataset is an American Geography database designed by [36]. The dataset contains Prolog logical form for NL questions. Later, it was converted to SQL by authors of [6]. This dataset has various versions, and number of queries are different in various versions. We use two versions of this dataset: GEO250 and GEO880. These two versions contain 250 and 880 SQL queries respectively. This dataset is also diverse in terms of SQL features.

Yelp dataset. Yelp dataset is a collection of NL questions upon the Yelp database which is built upon the Yelp website [37]. Yelp database contains diverse business information such as business names, categories, locations, user reviews and tips. This database consists of 7 tables. A $business$ table contains business names, addresses, average ratings, and the total number of reviews for each business. $Category$ and $neighborhood$ tables contain category names and neighborhood names of each business respectively. The number of checkins on different days are available in a table called $checkin$. Two more tables named $review$ and tip contain user ratings and feedbacks respectively. The last table $user$ contains names of all the users who reviewed or gave tips for improvement to any business. The first Yelp dataset was published by [5]. This dataset contains only 128 queries and it is also diverse in terms of SQL features. Since it is a very small dataset, we generate a new version of this dataset which contains 1000 queries. Our dataset is also complex and diverse in terms of SQL features. It has a wide variety of queries comprising nested queries and SQL clauses like WHERE, GROUP BY, HAVING, and ORDER BY. Three undergraduate students worked on generating this dataset as part of earning honors credit, and each of them wrote 200 queries. Again, 13 groups of undergraduate students worked on writing queries for Yelp database as part of earning bonus points in one of their courses related to database management system. Each group wrote 50 queries. After collecting all queries, we checked for accuracy and removed all duplicate and invalid questions. Table III reports the number of tables per database, number of foreign keys per database, and percentage of various SQL features in all datasets. Percentage of SQL features reported in Table III shows that our proposed Yelp dataset is the most diverse dataset among all datasets.

	WikiSQL	Spider	GEO250	GEO880	Yelp
# DBs	5230	20	1	1	1
# Tables/DB	1	5.1	8	8	7
# FKs/DB	0	3.2	8	8	7
WHERE (%)	99	47.5	75.6	72.9	85.4
JOIN (%)	0	39.8	7.6	11.3	50
GROUP BY (%)	0	30	1.2	5.9	25.2
HAVING (%)	0	7.8	0	0.8	10.6
ORDER BY (%)	0	23	31.2	31.3	19
Nested Query (%)	0	4.4	1.2	10.9	12.8

TABLE III: Summary of Datasets. # DBs, # Tables/DB, and # FKs/DB denote the total number of databases, Number of tables and foreign keys per database in every dataset. WHERE, JOIN, GROUP BY, etc... represent the percentage of queries with various SQL features.

D. Benchmark Implementation Details

In this section, we present a benchmark, NLIDB-Bench, that we use to evaluate different approaches of synthesizing SQL queries from NL questions. This benchmark is designed in an extensible way such that newly proposed approaches can be plugged in into our benchmark and can be evaluated upon all the datasets and evaluation metrics described in section III-B. Besides the datasets mentioned in the previous section, new datasets can also be inserted into our benchmark.

Our benchmark, NLIDB-Bench has mainly three modules: Preprocessing, Translating SQL, and Inserting new dataset. First of all, NLIDB-Bench has a configuration file Config.py which contains five required static information: database name, database connection information, questions file path, ground truth queries file path, and preprocessing output path. For the preprocessing module, the benchmark has a file Preprocess.py containing a class named Preprocess. For each approach, we run this file, and it does all the preprocessing required by that approach. During preprocessing, if it needs to save any file, that file should be saved under preprocessing output path directory mentioned in Config file. During the preprocessing, it can access the database name, database connection information, questions and corresponding ground truth SQLs from the Config file. If any new approach is to be plugged in into the benchmark, that approach should inherit the Preprocess class in such a way that it can do all the preprocessing once we run Preprocess.py.

The second part of our benchmark is to translate NL questions into SQL queries. NLIDB-Bench has a file Translation.py containing the class TranslateNLIDB. This class has two functions: loadData and translateQuery. Once loadData function is called for an approach, it should load all the preprocessed files along with other supporting files it needs to perform the translation. As an example, neural network based approaches should load the pre-trained models. Another function of TranslateNLIDB class, translateQuery, is called with the parameter NL question. This function should output the SQL query of the given question. Both loadData and translateQuery functions can access the database information from the Config file. The last part of our benchmark is inserting new

dataset into the benchmark. For this purpose, the benchmark has a class InsertNewDataset which receives three parameters in its constructor: dbPath, questionsPath, sqlQueryPath. First parameter, dbPath, is the path to the database which should be a .sql file. The second and third parameters correspond to the file paths of NL questions and the corresponding SQL queries respectively.

IV. Experimental Evaluation

We conduct our experiments on a machine with Intel(R) Core(TM) i7-4790 3.60GHz quad core CPU. The installed operating system is Ubuntu 16.04 LTS, and the installed memory is 32GB DDR3 RAM. Basically, the implementations are done using python 2.7, but Java implemented approaches are also tested by establishing a connection with python.

We evaluate all the compared approaches upon the datasets described in section III-C using the evaluation metrics described in section III-B. We select four state-of-the-art approaches from the categories described in section II. We did not evaluate any approach from the keyword-based category because it is a very naive approach and works only for NL questions represented as a bag of keywords. From the neural network based category, we evaluated SyntaxSQLNet [26] and TypeSQL [1]. Besides these approaches, we also evaluated PRECISE [6] from the semantic parsing category and NaLIR [13] from the syntax parsing category. Note that for NaLIR, we have shown the results without user interaction in order to be consistent with other compared approaches that do not involve a human-in-the-loop. For results on real human interaction, the readers may refer to the original paper [13]. We used github implementations by authors of these approaches and make necessary changes to fit the implementations into NLIDB-Bench.

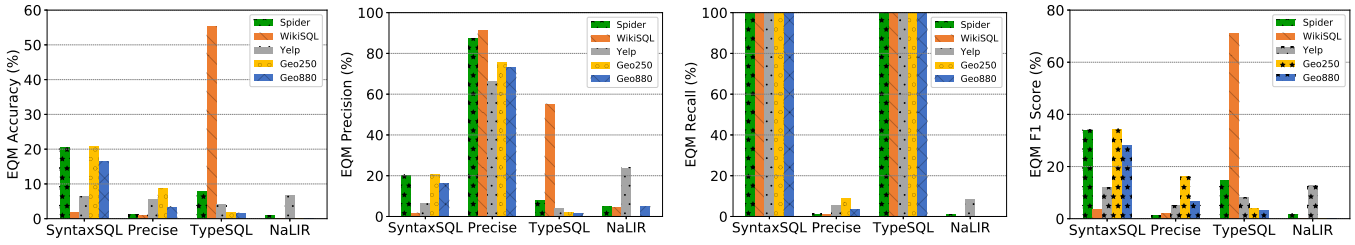
A. Evaluating EQM

Figure 5 shows the comparison of accuracy, precision, recall and F1-score on exact query matching for all the implemented approaches over all the datasets. Exact query matching values over different datasets indicate that although TypeSQL [1] yields a high matching quality over the WikiSQL dataset, SyntaxSQLNet [26] performs the best upon all the remaining datasets. Also, we can notice that there is not a single winning approach that performs the best over all the datasets. NaLIR performs poorly over all the datasets.

Precise stores a list of syntactic makers which include but are not limited to are, the, in, that, be, of, do, with, and by. It also manages a list of wh-tokens such as what, where, when, who, how, and how many. If a token in the NL question is not available in the syntactic makers list or wh-tokens list, Precise tries to map that token into a database token. If any token in an NL question cannot be mapped to any of the database tokens, Precise fails to synthesize a SQL query for that NL question. This is the most important drawback of Precise that limits its SQL translation efficiency. In order to understand this drawback, consider two NL queries: *What are the states that border Florida?* and *What are the names of the states that border Florida?*. Although Precise can successfully translate the first question into its corresponding SQL query, it fails to synthesize any candidate query for the second question. The second question contains an extra token: *name* which is neither a database token nor is it available in the wh-tokens list or syntactic makers list. Just because it fails to map the token *name* to any database token, it can not generate any SQL query for the second question. In case of NaLIR, if user interaction is turned on, it can solve this mapping problem sometimes though interaction with the user, but NaLIR without interaction also suffers from the same mapping problem. Another problem that degrades the performance of both NaLIR and Precise is their lack of ability to detect synonyms of database tokens. Sometimes, NL questions may contain synonyms of database tokens instead of the tokens themselves which often results in wrong mapping of the NL query tokens to database tokens in case of Precise and NaLIR. On the other hand, neural network based approaches (SyntaxSQLNet and TypeSQL) mitigates this problem using word embedding and paraphrase embedding which results in improved SQL translation efficiency. Another point that helps TypeSQL improve its translation accuracy is its ability to detect the types of NL tokens. After tokenizing the NL question, it checks the type of each token.

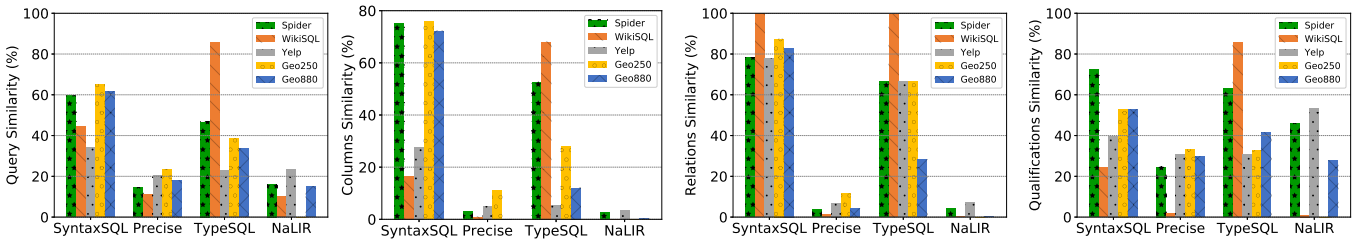
Among the two neural network based approaches, SyntaxSQLNet performs better than TypeSQL on all the datasets except for the WikiSQL dataset. While TypeSQL builds neural network models only to predict columns, aggregation operators, and mathematical operators, SyntaxSQLNet has nine separate models predicting all units of the SQL query. This is one reason for which SyntaxSQLNet surpasses TypeSQL in synthesizing SQL queries from NL questions. Another efficient part of SyntaxSQLNet is the use of SQL specific syntax tree based decoder which helps to invoke the right model during the different steps of SQL query synthesis.

Precision on exact query matching values are similar for both SyntaxSQLNet [26] and TypeSQL [1]. The reason is that these two approaches generate SQL queries for all NL questions regardless of the correctness of the queries. For the same reason, recall on exact query matching values are 100% for these two approaches. It can also be said that precision value of the approach Precise is very high compared to other approaches although the values of other metrics such as query matching accuracy, recall and f1 scores are not that high. The reason is that this approach can translate very few NL questions to SQL queries, but most of the translated SQL queries by this approach are correct. Precision and recall values of NaLIR are almost similar on all the datasets. Since there are very high differences between precision and recall values SyntaxSQLNet and TypeSQL, F1-score helps to find the balance between precision and recall values. F1-score shows that SyntaxSQLNet performs best on all the datasets except WikiSQL, on which TypeSQL is the best performer.



(a) Accuracy (b) Precision (c) Recall (d) F1 Score

Fig. 5: Studying the effect of various datasets and NL-2-SQL approaches on EQM



(a) Query Similarity (b) Columns Similarity (c) Relations Similarity (d) Qualifications Similarity

Fig. 6: Studying the effect of various datasets and NL-2-SQL approaches on Query Similarity

B. Analyzing Query Similarity

Figure 6 shows the query similarity, columns similarity, relations similarity, and qualifications similarity for all the implemented state-of-the-art approaches over all the five datasets. The higher query similarity values compared to the exact query matching values indicate that these SQL translation approaches synthesize various components of SQL query correctly most of the time. Still, exact query matching values are poor because failure to synthesize even a single component of a query evaluates to incorrect synthesis for the entire query using the exact matching metric. Evaluating similarity values separately for three basic SQL components (columns, relations, and qualifications) helps to figure out those units of a simple SQL query that are difficult to translate correctly. Columns, relations, and qualifications similarities reported in Figure 6 verify that most of the approaches perform good in picking correct relations in FROM clause but not so good in choosing correct columns in the SELECT clause.

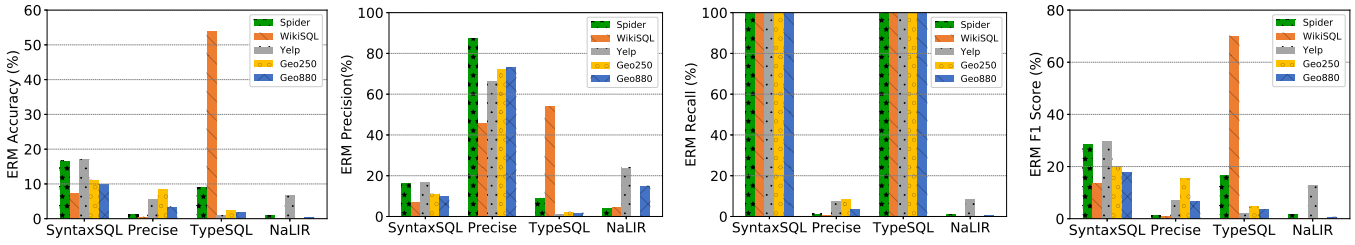
SyntaxSQL and TypeSQL have 100% relation-list similarity on the WikiSQL dataset because each database in the WikiSQL dataset has only one table. In spite of this simplicity, Precise and NaLIR have low relation-list similarities because this two approaches do not form any SQL query for most of the NL questions. The reason behind the low columns similarity can be explained with a simple example. Consider the yelp query: *What are the businesses in place X with rating more than 3.5?*. Here, the token businesses refer to the *business* table of the Yelp database. Among all columns of business table, the example question refers to the *name* column. Absence of *name* token in the NL question results in the selection of wrong columns in the SELECT clause.

C. Studying ERM

Figure 7 shows the comparison of accuracy, precision, recall, and F1-score in terms of exact execution result matching for all the implemented approaches over all the five datasets. The summary of this graph is that the behavior of all the approaches on exact execution result matching is similar to that on exact query matching. The explanations behind precision, recall, and F1-score on EQM are also applicable to precision, recall, and F1-score on ERM. In the case of SyntaxSQLNet, the ERM values on WikiSQL dataset and Yelp dataset are better than the EQM values on the same datasets. On the other hand, ERM values of this approach drop for Geo250 and Geo880 datasets as compared to the EQM values. This behavior is an indication that these two exact matching evaluation metrics (EQM and ERM) are not good representatives of how good various NL-2-SQL approaches are.

D. Evaluating ERA

The exact execution results matching values mentioned in the previous section look for exact equality of predicted query results and ground truth query results. So, this evaluation metric does not indicate how close the predicted query results are to the ground truth query results. To bridge this gap, we have another evaluation metric which is execution results affinity. Figure 8 shows the comparison of accuracy, precision, recall, and F1-score on execution results affinity for all the implemented state-of-the-art approaches over all the five datasets. Execution result affinity values are higher than exact execution result matching values. Especially, execution result affinity values rise significantly on the Yelp dataset. Although neural network based approaches show 100% recall in terms



(a) Accuracy (b) Precision (c) Recall (d) F1 Score

Fig. 7: Studying the effect of various datasets and NL-2-SQL approaches on ERM

of EQM and ERM, recall values in terms of ERA drop significantly because all approaches have false negatives among the execution results. The excellent balance between precision and recall values in terms of ERA shows the significance of ERA in evaluating NL-2-SQL approaches.

The values of the execution results affinity show that SyntaxSQLNet [26] is the overall best performer for all five datasets while TypeSQL [1] is better than SyntaxSQLNet only on the WikiSQL dataset. There are explanations behind this behavior. Mapping NL tokens to correct database tokens is one of the most important steps of synthesizing a correct SQL query. Looking at the table names and column names of the WikiSQL dataset, we can say that table and column names of this dataset are not good enough to define type of information those tables and columns contain. This problem makes it very difficult for SyntaxSQLNet to map NL tokens to correct SQL tokens. On the other hand, TypeSQL performs very good for this dataset, because it looks for all values of each column in a table while helps it to map NL tokens to correct columns and tables. However, TypeSQL does not perform well on other datasets except the WikiSQL dataset. The reason is that WikiSQL dataset is very different from all other datasets which is reported in Table III. In Section III-C, we showed that each NL question in the WikiSQL dataset contains information only from one table because of which no join is required between various tables. TypeSQL forms only SELECT and WHERE clauses of SQL queries. The nature of WikiSQL dataset helps TypeSQL to avoid this problem while synthesizing SQL queries for other datasets require the formation of other clauses along with join operations and nested queries. This is why, TypeSQL performs very poorly over other datasets as compared to the WikiSQL dataset.

E. Scalability Evaluation

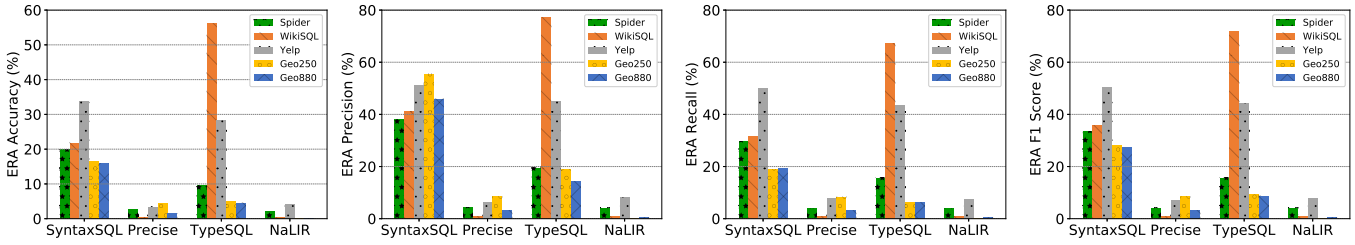
Besides the correctness of the translated SQL queries, scalability is another important factor to consider while translating NL questions to SQL queries. Query translation time is the most important evaluation metric to look at while evaluating the scalability of an NL to SQL translation system. An analysis of the comparison of query translation time for various state-of-the-art algorithms on each dataset is shown in Table IV. All the times are in milliseconds. All the translation times shown in Table IV show that TypeSQL is the fastest approach to convert NL questions to SQL queries. Another neural network based approach, SyntaxSQLNet is also very fast in SQL translation. Although a bit slower than neural network based approaches, Precise can finish the translation within a reasonable time. We observe that NaLIR takes a very long time to translate Yelp queries although it can finish the conversion for other datasets within a reasonable time. The preprocessing time differs among various approaches depending on the nature of the exact preprocessing task. In the case of neural network based approaches, the time required to train all neural network models influences scalability. For training purpose, these approaches need a large set of pairs of NL questions and SQL queries which is a hindrance to the scalability of these approaches because all databases do not have a large set of pairs of questions and SQL queries. We train the SyntaxSQLNet using the augmented training dataset of Spider and WikiSQL. We use the same trained models to test this approach over all test datasets. Yelp and GeoQuery datasets are not large enough to be used as training datasets. SyntaxSQLNet consists of nine modules, and each of these modules needs to be trained separately which results in a long training time for SyntaxSQLNet. We train TypeSQL using WikiSQL & Spider datasets. Preprocessing time for Precise includes lexicon preparation time. The lexicon preparation time for Spider dataset reported in the table is the summation of lexicon preparation time for all databases in the Spider dataset. It is also same for WikiSQL dataset. Time needed to generate lexicon for precise depends on the values of various attributes in the database. If the attribute values contain long text, lexicon preparation takes a lot of time. Lexicon preparation time for Yelp is very high because Yelp database contains user reviews and tips which contain very long text. Preprocessing time

	Spider	WikiSQL	Yelp	Geo250	Geo880
SyntaxSQL	67.6	54.29	73.68	84	84
Precise	99.97	168.52	110	69.59	69.59
TypeSQL	2.9	12.6	10.53	4	4
NaLIR	278.83	270	2438	104.65	154.13

TABLE IV: Query Translation Time in milliseconds

	Spider	WikiSQL	Yelp	Geo250	Geo880
SyntaxSQL	330337	330337	330337	330337	330337
Precise	14.25	77507	49174.7	13.12	13.12
TypeSQL	105556	54141	105556	105556	105556

TABLE V: Preprocessing Time in seconds



(a) Accuracy (b) Precision (c) Recall (d) F1 Score

Fig. 8: Studying the effect of various datasets and NL-2-SQL approaches on ERA

required by all approaches for each dataset is shown in Table V. In the case of neural network based approaches such as SyntaxSQLNet and TypeSQL, preprocessing time reported in the table indicates the training time while for Precise, reported time indicates the lexicon preparation time. NaLIR does not require any preprocessing.

V. Learned Lessons

The summary of the lessons learned from our extensive evaluation using NLIDB-Bench is given below:

- Despite the great advancement in syntax and semantic parsing technologies in the recent years and availability of many syntax and semantic parsing based algorithms for translating NL questions into SQL queries, the translation accuracy of such NL-2-SQL approaches are not very promising. On the contrary, neural network based approaches show more promising translation accuracy despite a late start in the development of NL-2-SQL approaches in this category. Along with their good translation accuracy, such neural network based approaches exhibit faster NL-2-SQL translation time.
- Although NL-2-SQL approaches in the neural network based category are performing generally better than the other categories, still none of the approaches is efficient enough in translating NL questions into SQL queries. Even the relatively high translation accuracy of TypeSQL over the WikiSQL dataset does not prove the effectiveness of TypeSQL. That is because TypeSQL still fails to show such accuracy over other datasets. The low translation accuracy of all the evaluated approaches proves that building efficient NLIDBs is still an open research problem.
- Even though some of the approaches try to form complex SQL queries consisting of nested queries along with `GROUP BY`, `HAVING`, and `ORDER BY` operations, none of these approaches is deemed successful in forming these type of complex queries. That opens the door for new research directions that aim at designing novel NL-2-SQL approaches.
- The large variation in SQL synthesis efficiency over different datasets shows that none of the approaches can consistently maintain a good NL-2-SQL translation accuracy. Moreover, the experimental evaluation proves that none of the studied approaches can show a good balance between precision and recall over all the datasets.

VI. Conclusion and Future Work

Building an efficient NLIDB has been a hot research question for many years, and a number of state-of-the-art algorithms have recently been proposed to build NLIDBs. In this paper, we present $NLIDB\text{-}Bench$, a benchmark that evaluates existing state-of-the-art NL-2-SQL approaches over five different datasets. Our benchmark proposes some evaluation metrics that are very helpful in evaluating NL-2-SQL approaches. We also create a new Yelp dataset that is a diverse collection of NL questions and query pairs. While an existing benchmark can only evaluate neural network based approaches, NLIDB-Bench can be used to evaluate NL-2-SQL approaches from all categories. Evaluation of four state-of-the-art promising NL-2-SQL approaches from various categories through our benchmark NLIDB-Bench shows that neural network based approaches are more efficient than approaches from other categories although none of these approaches is efficient enough in perfectly synthesizing SQL queries from NL questions. Our benchmark show that there exists ample scope for future research in the field of building effective, efficient and scalable NLIDBs. In future, we will share the code of our benchmark with both the Database and NLP communities to foster more research in the NL-2-SQL area.

References

[1] T. Yu, Z. Li, Z. Zhang, R. Zhang, and D. Radev, "Typesql: Knowledge-based type-aware neural text-to-sql generation," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Association for Computational Linguistics, 2018, pp. 588–594. [Online]. Available: http://aclweb.org/anthology/N18-2093

[2] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," *CoRR*, vol. abs/1709.00103, 2017. [Online]. Available: http://arxiv.org/abs/1709.00103

[3] X. Xu, C. Liu, and D. Song, "Sqlnet: Generating structured queries from natural language without reinforcement learning," *CoRR*, vol. abs/1711.04436, 2017. [Online]. Available: http://arxiv.org/abs/1711.04436

[4] I. Gur, S. Yavuz, Y. Su, and X. Yan, "Dialsql: Dialogue based structured query generation," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2018, pp. 1339–1349. [Online]. Available: http://aclweb.org/anthology/P18-1124

[5] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, "Sqlizer: Query synthesis from natural language," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 63:1–63:26, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133887

[6] A.-M. Popescu, O. Etzioni, and H. Kautz, "Towards a theory of natural language interfaces to databases," in *Proceedings of the 8th International Conference on Intelligent User Interfaces*, ser. IUI '03. New York, NY, USA: ACM, 2003, pp. 149–157. [Online]. Available: http://doi.acm.org/10.1145/604045.604070

[7] A. Copestake and K. S. Jones, "Natural language interfaces to databases," *The Knowledge Engineering Review*, pp. 225–249, 1990.

[8] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, "Natural language interfaces to databases - an introduction," *CoRR*, vol. cmp-lg/9503016, 1995. [Online]. Available: http://arxiv.org/abs/cmp-lg/9503016

[9] M. N. Nihalani, D. S. Silakari, and D. M. Motwani, "Natural language interface for database : A brief review," 2011.

[10] E. U. Reshma and P. C. Remya, "A review of different approaches in natural language interfaces to databases," in *2017 International Conference on Intelligent Sustainable Systems (ICISS)*, Dec 2017, pp. 801–804.

[11] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2018, pp. 3911–3921. [Online]. Available: http://aclweb.org/anthology/D18-1425

[12] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. zcan, "Athena: An ontology-driven system for natural language querying over relational data stores," *Proceedings of the VLDB Endowment*, vol. 9, pp. 1209–1220, 08 2016.

[13] F. Li and H. V. Jagadish, "Constructing an interactive natural language interface for relational databases," *Proc. VLDB Endow.*, vol. 8, no. 1, pp. 73–84, Sep. 2014. [Online]. Available: http://dx.doi.org/10.14778/2735461.2735468

[14] H. Bast and E. Haussmann, "More accurate question answering on freebase," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, ser. CIKM '15. New York, NY, USA: ACM, 2015, pp. 1431–1440. [Online]. Available: http://doi.acm.org/10.1145/2806416.2806472

[15] L. Blunschi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger, "Soda: Generating sql for business users," *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 932–943, Jun. 2012. [Online]. Available: http://dx.doi.org/10.14778/2336664.2336667

[16] A. Simitsis, G. Koutrika, and Y. Ioannidis, "Précis: from unstructured keywords as queries to structured databases as answers," *The VLDB Journal*, vol. 17, no. 1, pp. 117–149, Jan 2008. [Online]. Available: https://doi.org/10.1007/s00778-007-0075-9

[17] A. Giordani and A. Moschitti, "Generating sql queries using natural language syntactic dependencies and metadata," in *Proceedings of the 17th International Conference on Applications of Natural Language Processing and Information Systems*, ser. NLDB'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 164–170. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31178-9_16

[18] ——, "Translating questions to sql queries with generative parsers discriminatively reranked," in *Proceedings of COLING 2012: Posters*. The COLING 2012 Organizing Committee, 2012, pp. 401–410. [Online]. Available: http://aclweb.org/anthology/C12-2040

[19] A. Moschitti, D. Pighin, and R. Basili, "Semantic role labeling via tree kernel joint inference," in *Proceedings of the Tenth Conference on Computational Natural Language Learning*, ser. CoNLL-X '06. Stroudsburg, PA, USA: Association for Computational Linguistics, 2006, pp. 61–68. [Online]. Available: http://dl.acm.org/citation.cfm?id=1596276.1596289

[20] A. Severyn and A. Moschitti, "Structural relationships for large-scale learning of answer re-ranking," in *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '12. New York, NY, USA: ACM, 2012, pp. 741–750. [Online]. Available: http://doi.acm.org/10.1145/2348283.2348383

[21] A. Vertsel and M. Rumiantsau, "Pragmatic approach to structured data querying via natural language interface," *CoRR*, vol. abs/1807.00791, 2018. [Online]. Available: http://arxiv.org/abs/1807.00791

[22] A.-M. Popescu, A. Armanasu, O. Etzioni, D. Ko, and A. Yates, "Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability," in *Proceedings of the 20th International Conference on Computational Linguistics*, ser. COLING '04. Stroudsburg, PA, USA: Association for Computational Linguistics, 2004. [Online]. Available: https://doi.org/10.3115/1220355.1220376

[23] H. Poon, "Grounded unsupervised semantic parsing," *ACL 2013 - 51st Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, vol. 1, pp. 933–943, 01 2013.

[24] C. Hallett and D. Hardcastle, "Towards a bootstrapping nlidb system," in *Proceedings of the 13th International Conference on Natural Language and Information Systems: Applications of Natural Language to Information Systems*, ser. NLDB '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 199–204. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69858-6_20

[25] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer, "Learning a neural semantic parser from user feedback," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2017, pp. 963–973. [Online]. Available: http://aclweb.org/anthology/P17-1089

[26] T. Yu, M. Yasunaga, K. Yang, R. Zhang, D. Wang, Z. Li, and D. Radev, "SyntaxSQLNet: Syntax tree networks for complex and cross-domain text-to-SQL task," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, Oct.-Nov. 2018, pp. 1653–1663. [Online]. Available: https://www.aclweb.org/anthology/D18-1193

[27] T. Guo and H. Gao, "Bidirectional attention for sql generation," *CoRR*, vol. abs/1801.00076, 2017.

[28] R. Cai, B. Xu, X. Yang, Z. Zhang, and Z. Li, "An encoder-decoder framework translating natural language to database queries," in *IJCAI*, 2018.

[29] Y. Sun, D. Tang, N. Duan, J. Ji, G. Cao, X. Feng, B. Qin, T. Liu, and M. Zhou, "Semantic parsing with syntax- and table-aware sql generation," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2018, pp. 361–372. [Online]. Available: http://aclweb.org/anthology/P18-1034

[30] A. Giordani and A. Moschitti, "Syntactic structural kernels for natural language interfaces to databases," in *Proceedings of the 2009th European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part I*, ser. ECMLPKDD'09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 391–406. [Online]. Available: https://doi.org/10.1007/978-3-642-04180-8_43

[31] ——, "Semantic mapping between natural language questions and sql queries via syntactic pairing," in *Proceedings of the 14th International Conference on Applications of Natural Language to Information Systems*, ser. NLDB'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 207–221. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12550-8_17

[32] 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'06)*. European Language Resources Association (ELRA), 2006. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2006/pdf/440_pdf.pdf

[33] "Wikisql data." [Online]. Available: https://github.com/MetaMind/wikisql

[34] "Spider data." [Online]. Available: https://yale-lily.github.io//spider

[35] "Geoquery data." [Online]. Available: http://www.cs.utexas.edu/users/ml/nldata/geoquery.html

[36] L. R. Tang and R. J. Mooney, "Using multiple clause constructors in inductive logic programming for semantic parsing," in *Proceedings of the 12th European Conference on Machine Learning*, ser. EMCL '01. London, UK, UK: Springer-Verlag, 2001, pp. 466–477. [Online]. Available: http://dl.acm.org/citation.cfm?id=645328.650015

[37] "Yelp dataset." [Online]. Available: https://www.yelp.com/dataset