CHAPTER 5

Evaluation

Suppose you are in charge of administering an enterprise database and are tasked with choosing a NLIDB system to provide a non-technical interface to the users of your database. You check the literature and find hundreds of research papers on the subject. You also check a few leaderboards and find tens of top-performing models. How do you decide on a system? What should you look for in choosing one from many choices? These are some of the questions that are answered in this chapter.

The chapter is organized as follows. The next section gives a general overview of the evaluation methodology for NLIDBs with a focus on text-to-SQL as well as the structure of the chapter. Datasets and benchmarks are presented in Section 5.2, and the two major evaluation methods, reference based and human centric evaluations, are discussed in Section 5.3 and 5.4 respectively. Finally, other evaluation metrics are reviewed in Section 5.5 and some of top-performing models and their characteristics are presented in Section 5.6.

5.1 METHODOLOGY OVERVIEW

As depicted in Figure 2.2, a NLIDB system consists of a few components including query understanding, query translation, interaction generation and structured query processing. Each of these components may be evaluated on its own or in combination with other components. The two common approaches that have emerged for evaluating NLIDB include (1) the performance of the system is measured in terms of that of the components that are responsible for mapping informal natural language questions to formal structured queries, and (2) the downstream task performance is measured, for example, whether an input question is answered and maybe the time and resources taken to obtain an answer.

A quantitative evaluation requires a workload or a benchmark based on which the performance of each system is measured and multiple systems can be compared. A *workload* refers to a set of questions or queries and the frequency or volume of each query, which indicates its importance. Clearly a benchmark has to be relevant as it is only a proxy for the actual user needs. For the same reason, multiple benchmarks are developed and some of them are reviewed in Section 5.2. The performance of database systems is traditionally measured in terms of throughput and price. Here throughput refers to the number of units of work or transactions completed per second, and price is the cost of the system, for example, five-year ownership cost in TPC benchmarks [Gray, 1992]. However, such performance metric is less relevant if the user question is not correctly mapped to a query. On this basis, the performance of NLIDB systems is measured in terms of the accuracy of the mappings as the primary criterion and the cost (e.g. running time, I/O and memory cost, etc.) as the secondary criterion. For example, two models can perform similarly in terms of the accuracy of the mappings but one can take less CPU time or less memory. Everything else the same, we prefer the model taking less resources.

Detecting or verifying the correctness of a generated query is not as simple as one may wish. It is generally impossible to fully automate the verification problem due to the *halting problem* which is undecidable [Davis, 2004]. Under this caveat, two evaluation methods are established: (1) reference-based evaluation, (2) human-centric evaluation. In a *reference-based evaluation*, there is already a reference solution or query that answers the question and one has to decide if a newly generated query is equivalent to the known reference. In a *human-centric evaluation*, a reference query may or may not be available and the verification is done by humans. Such human evaluators must be familiar with the query language to affirm whether a query answers a question. The two evaluation methods are discussed in more detail in Sections 5.3 and 5.4 respectively. Our focus in this chapter is on SQL, but the methods and challenges are broadly applicable to other query languages such as SPARQL, XQuery, etc.

5.2 DATASETS AND BENCHMARKS

Many datasets have been developed for evaluating natural language text to SQL systems. One of the early datasets is developed by Price [Hemphill et al., 1990] in the context of evaluating spoken language systems. With the surge of interest in NLIDBs, more datasets are published in the following years, covering more domains, more complex queries, etc. A subset of these datasets are reviewed here (see Section 5.2.1 for more general statistics about these datasets).

ATIS

This dataset, introduced by Price [Hemphill et al., 1990] and used in some follow-up works (e.g. Dahl et al. [1994], Iyer et al. [2017b]), is a collection of user questions on a flight database that consists of tables such as airline, airport, flight and fare. User questions over the database are manually annotated, resulting in over 900 SQL queries. The dataset contains many variations of the same query template, for example, with city names varied or the natural language questions slightly rephrased. One third of the queries use some form of nesting and a handful of them use group-by with no having clause.

GeoQuery

The dataset was developed by Zelle and Mooney [Zelle and Mooney, 1996] with queries in Prolog and the queries were changed to SQL in the follow-up works [Iyer et al., 2017b, Popescu et al., 2004]. This dataset is a collection of questions about US geography (e.g. what is the highest mountain in US?). The data is stored in 8 tables and the queries use a good mix of nesting and grouping. The average query length is less than that of ATIS because with fewer tables, there is less need for join.

Scholar

This dataset [Iyer et al., 2017b] includes a collection of 817 questions about academic publications (e.g. "What papers are written by authorX and authorY?" and "Who has the most publications on keyphraseX?"). There are multiple variations of each question, hence the number of unique query templates is much less. The SQL queries are generated automatically and are verified by users for correctness.

Academic

This dataset, first reported in the work of Li and Jagadish [Li and Jagadish, 2014c], consists of data from Microsoft Academic Search (MAS) stored in 15 tables (e.g. author, publication, journal, and keyword). The query set includes all logical queries that could be expressed on the search page of MAS and has a good mix of group-by, order-by and nesting.

Advising

This is a database of course information stored in 15 tables (e.g. offering, instructor, TA, and student) and an initial set of 208 questions either written manually or collected from the University of Michigan Facebook page [Finegan-Dollak et al., 2018]. A paraphrasing tool is used to generate multiple variations of each question.

IMDB and Yelp

Both datasets were collected by asking users in the organization of the authors to write English queries that they wanted to be answered on IMDB and Yelp [Yaghmazadeh et al., 2017c]. The users were given the type of data in each database (e.g. business and their cities and rating for Yelp and movies with their casts, directors and budget for IMDB) but they were not given the actual schema. The SQL queries were generated by the authors and were verified manually.

Fiben

The database in Sen et al. [2020], referred to as Fiben, models information about public companies, their offices, financial metrics, and transactions over holdings and securities. The schema is based on the union of two standard finance ontologies: (1) Financial Industry Business Ontology (FIBO) [1] and (2) Financial Report Ontology (FRO) [2]. Typical queries are about the financial health and performance of public companies (e.g. "Tell me the last traded value of Alphabet" and "Find all investors not holding MSFT stocks"). The benchmark includes 300 natural language questions and 237 distinct target SQL

[1] https://spec.edmcouncil.org/fibo

[2] http://www.xbrlsite.com/2015/fro

queries. More than 50% of the SQL queries are nested, which are further broken down to correlated sunqueries and uncorrelated subqueries and those with and without aggregation functions in the subqueries. Nested queries with an uncorrelated structure are generally simpler since the subqueries are independent of the outer query they are nested in, and those subqueries may be evaluated once for all iterations of the outer query.

WikiSQL

This work [Zhong et al., 2017] is a departure from the earlier work in that the queries are not formulated on a single database with a known schema. Instead the dataset uses a collection of more than 24000 tables from Wikipedia. Each query targets a single table and is constructed by crowd sourcing, where each worker is given a table and a question and is asked to paraphrase it. The set of questions that are given to workers are generated using some templates with corresponding SQL queries. Finally the paraphrases with less variations are discarded.

SPIDER

A key design goal in Spider [Yu et al., 2018c] is the generalizability to new databases and unseen schemes. Most of the works prior to Spider focus on a single database schema, which is available during training and is the only database used in testing. Since queries written on a single database schema usually follow a limited number of templates and many of those templates are seen during the training, a text-to-SQL task often reduces to selecting a template followed by filling the slots.

The Spider benchmark consists of 200 database schemas collected from different sources by a group of CS students at Yale University. It includes 70 database schemas collected from database college courses in different schools and various textbooks and online SQL tutorials. Another 40 database schemas are selected from DatabaseAnswers [3], which contains a large collection of industry-specific data models in many different domains. The remaining 90 schemas are constructed based on WikiSQL dataset after selecting 500 tables, grouping related tables into roughly 90 domains, and manually linking related tables in each domain using appropriate foreign keys. Of the 200 databases, only 160 are published and the remaining 40 databases and their queries are kept private to avoid overfitting. The developers can submit their models to be tested on the hidden test set.

The authors ask annotators to generate for each database schema 20-50 natural language questions and their SQL expressions. The annotators are explicitly asked to cover as many SQL components as possible including GROUP BY, HAVING, ORDER BY, LIMIT, UNION, INTERSECT, EXCEPT, EXISTS, IN, and LIKE. Some constraints are also placed on the query generation to make the learning process easier: (1) if multiple equivalent query templates can be generated for a question, the annotators are pushed to always select the same template; (2) questions cannot include ambiguous

[3] http://www.databaseanswers.org/data_models/index.htm

Type	Query

Domain Knowledge	List account id who chooses weekly issue issuance statement.
	SELECT account_id FROM account WHERE frequency=’TOPLATEK TYDNE’
	How many accounts are eligible for loans in New York City.
	SELECT COUNT(*) FROM account WHERE type=’OWNER’ AND city=’NY’

Numeric Computation	What percentage of posts are made by an elder user.
	SELECT CAST(SUM(IIF(T2.age>65, 1, 0)) AS REAL)*100/COUNT(T1.id)
	FROM posts AS T1 INNER JOIN users AS T2 ON T1.ownerUserId=T2.id

Table 5.1: Examples of reasoning type queries from BIRD [Li et al., 2023b]

terms and common sense knowledge outside databases; and (3) to increase the diversity of questions, the annotators are asked to paraphrase some questions and include those paraphrases as well.

BIRD

BIRD [Li et al., 2023b] is a more recent dataset (released in 2023) with a focus on database value comprehension to write the queries. Unlike some of the other benchmarks, the dataset includes not only the schema of the tables but also their content. The size of the released database content is 33.4 GB, the largest we have seen in text-to-SQL datasets. Each table on average has 549k rows. The dataset includes a total of 95 databases (69 in the train set, 11 in dev set and the remaining 15 in the test set).

Two types of queries are included in the dataset: (1) fundamental type, and (2) reasoning type. The former includes queries that can be often answered using the schema of the tables. This is also the common type of queries found in other datasets. The latter includes queries that require domain knowledge, numeric reasoning and synonym lists to formulate the queries (see Table 5.1 for examples). 70% of the queries in this class require database cell values to resolve possible ambiguities.

Two performance metrics are used to evaluate models: (1) Execution accuracy, and (2) Valid efficiency score. The latter measures the efficiency of model-generated queries, in terms of the running time, to that of the gold queries. This score is zero for queries that do not return a correct result.

5.2.1 BENCHMARKS STATISTICS AND QUERY COMPOSITION

Table 5.2 gives some statistics about text to SQL datasets reviewed here including the number of tables, the number of questions and SQL queries as well as the length of questions and queries in characters. WikiSQL is the largest dataset in terms of the number of tables and queries; however, the queries in this dataset are simpler. As shown in Table 5.3, many SQL components are not present in WikiSQL queries and each query is using a single table. Fiben, Spider and BIRD include the largest number of variations in terms of the query structure. For example, 37% of queries in Fiben include at least a GROUP BY clause and 26% of queries include at least a HAVING clause. 4% of queries in BIRD make references to NULL values (e.g., “IS NULL” or “IS NOT NULL”). Spider and Advising have the largest number of queries with negation (e.g. using terms such as “NOT IN”, “NOT EXISTS” and “EXCEPT”). The datasets discussed here are all available for download [4][5] [6].

Dataset	#Tables	#Q	#SQL	len(Q)	len(SQL)
ATIS	25	5,280	1,134	69	1,027
GeoQuery	8	877	259	43	167
Scholar	12	817	284	42	367
Academic	15	196	189	82	361
Advising	18	4,387	214	61	515
IMDB	16	131	99	59	250
Yelp	7	128	122	67	295
Restaurants	3	378	23	56	314
Fiben	152	300	300	68	1,176
WikiSQL	26,530	80,655	51,159	58	126
Spider	875	9,693	5,183	66	197
BIRD	520	10,962	10,962	92	179

Table 5.2: Text-to-SQL datasets statistics (len refers to the length of questions and SQL queries in characters)

Dataset	tables/Q	nested	group	having	order	null	setOp	neg
ATIS	8.2	28%	<1%	0	0	4%	0	0
GeoQuery	2.5	64%	11%	3%	8%	0	0	3%
Scholar	3.4	2%	35%	7%	26%	0	0	0
Academic	3.5	4%	21%	10%	12%	0	0	0
Advising	3.5	16%	3%	0	7%	0	0	6%
IMDB	2.8	1%	6%	0	10%	0	0	0
Yelp	2.5	0	14%	3%	10%	0	0	0
Restaurants	2.4	17%	0	0	0	0	0	0
Fiben	5.7	57%	37%	26%	8%	0	2%	5%
WikiSQL	1.0	0	0	0	0	0	<1%	0
Spider	2.9	15%	24%	6%	22%	0	6%	6%
BIRD	2.4	8%	10%	1%	18%	4%	<1%	<1%

Table 5.3: Composition of SQL queries in text-to-SQL datasets

[4] https://github.com/jkkummerfeld/text2sql-data
[5] https://github.com/IBM/fiben-benchmark
[6] https://bird-bench.github.io

5.2.2 OTHER BENCHMARKS

The quest continues for a benchmark that is more comprehensive or covers a richer set of questions, for example in terms of the naturalness of questions or the complexity of SQL queries. For instance, Gkini et al. [2021] build their benchmark on datasets from IMDB, MAS, and YELP, following Yaghmazadeh et al. [2017b], but developing their own collection of queries [7]. Their queries cover a set of categories that are similar to those discussed in Section 3.2.4 and includes (1) join queries with terms matching both data and metadata, (2) aggregation and grouping, (3) queries with different comparison operators, conjunction and disjunction, (4) nested queries, (5) queries with synonyms, stemming and misspellings, and (5) queries with negation.

SParC [Yu et al., 2019c] is another dataset for evaluating multi-turn semantic parsing of text-to-SQL. The dataset, developed based on Spider, is context dependent and replaces each question in Spider with a sequence of questions. Each question in the sequence is usually related to or builds on top of the previous ones for the overall goal of answering the initial question. For example, the question "`Who is the head of the department with the least number of employees?`" may be replaced with the following three questions:

```
Q1. How many employees does each department have?
Q2. Which department has the least number of employees?
Q3. Who is the head of this department?
Q3. What is the name and position of this person?
```

CoSQL [Yu et al., 2019b] is yet another dataset developed based on Spider for evaluating conversational text-to-SQL systems. The interaction with the user is in the form of a dialog, with each step either formulating a question or resolving an ambiguty. Both SParC and CoSQL are available for download [8].

5.3 REFERENCE-BASED EVALUATION

The performance of a text-to-SQL system may be measured in comparison to a reference system. One such reference is a gold standard set of SQL queries that are known to be correct. All datasets discussed in the previous section provide at least one SQL query to each question in the dataset, and those queries are assumed to be correct. There are two issues that need to be addressed in a reference-based evaluation: (1) generating a reference, and (2) evaluating a candidate answer based on a reference.

Generating a reference

Generating a reference SQL query often requires users who are skilled in SQL. In some cases (e.g. SPIDER), users are given a database schema and are asked to come up with both questions and an SQL translation of each question. In other cases, questions are

[7] We could not find the query set online at the time of this writing.
[8] `https://yale-lily.github.io/sparc`,`https://yale-lily.github.io/cosql`

generated by the actual users who do not know the schema or are taken from a query log (e.g. IMDB, Yelp and Advising). Those questions are mapped to SQL by users who know the database schema and are proficient in SQL. WikiSQL works in a reverse order and generates SQL queries first. Given a table, random SQL queries that return a non-empty result set are generated along with a crude description of each query following some rules and templates. Those descriptions are later passed to crowd workers for better phrasing and different expressions. A general problem in generating a reference is that a natural language question can be subject to more than one interpretations, whereas SQL queries are based on logic (as discussed in Section 3.2) and exact, and an SQL query is likely to cover only one interpretation of the question. Assuming each question has a single interpretation, defined by a reference query, next we discuss how candidate answers can be evaluated.

Candidate answer evaluation based on a reference

Given a *reference query* that is known to be correct and a candidate query that needs to be evaluated, a reference-based evaluation must establish some form of equivalence between the two queries.

Definition 5.1 Two queries q_1 and q_2 are *equivalent*, denoted as $q_1 \equiv q_2$, if for every database instance I, their answers are identical, i.e. $q_1(I) = q_2(I)$.

Definition 5.2 A query q_1 is *contained* in a query q_2, denoted as $q_1 \subseteq q_2$, if for every database instance I, the answer of q_1 is contained in the answer of q_2.

Query equivalence is closely related to query containment in that

$$q_1 \equiv q_2 \iff q_1 \subseteq q_2 \; and \; q_2 \subseteq q_1.$$

It is a well-known result that query equivalence for first-order based query languages (which includes relational algebra, relational calculus and SQL) are undecidable. A proof may be given based on the undecidability of first order logic, which states that checking the validity in first-order logic on the class of all finite models is undecidable. In other words, we have no way of checking for a query such as

```
SELECT * FROM r WHERE c
```

if it is equivalent to

```
SELECT * FROM r WHERE false
```

on all possible instances when c is a general predicate. The problem becomes decidable but NP-complete if c is restricted to conjunctions of simple equality conditions constructed using column names and constants, known as *conjunctive queries*. Hence for a general query, which is not necessarily conjunctive, any reference-based evaluation is only a proxy measure which cannot be perfect, and a candidate query that is equivalent but not identical to a reference can be falsely deemed incorrect.

We next present some commonly-used performance metrics for evaluating a model generated query. These metrics are general, applicable to all ranges of queries, but they are also susceptible to errors. We then study semantic equivalence as a stronger form of equivalence between two queries with the caveat that it can only be performed on conjunctive queries.

5.3.1 PERFORMANCE METRICS

Despite the challenges in establishing an equivalence between a model-generated query and a reference, we need some proxy measures of performance to evaluate the progress. We discuss three such measures developed in the literature.

Exact Match Accuracy

Each SQL query can be broken down into different clauses or components including `select` clause, `where` clause, `group by` clause and `order by` clause, and an evaluation can be conducted for each clause independent of other clauses. In particular, the output schema of a candidate query should match that of a reference, hence the columns in the `select` clause are expected to match. Since the ordering of the columns in the output does not affect the correctness, entries in the `select` clause can be treated as a set and an exact set matching between candidate and reference queries may be performed. For example, given the reference `Select airline, flno, avg(price)`, the candidate `select airline, avg(price), flno` match under an exact set matching whereas the candidate `select airline, flno, min(price)` does not.

If the `where` clause of a query is in the form of a conjunction of a set of atomic predicates, then both the reference and the candidate queries are expected to satisfy the same set of atomic predicates. For each query, those atomic predicates may be treated as a set and an exact set matching between the predicate set of queries may be performed. Similarly, the predicates in the `having` clause may also be treated as a set for evaluation purposes. The columns in the `group by` and `order by` clauses may also be treated as sets and compared using exact set matching.

It should be clear that each of those component evaluations only provides a conservative metric and cannot capture possible differences due to different naming of the columns, conditions that are not expressed as a conjunction of atomic conditions, different forms or levels of nesting, etc. For example, Queries Q1 and Q2, expressed over our tables in Figure 3.3, are equivalent but their `select` clauses will not match.

```
Q1.
select a, f, f.dep, av, c
from (select airline as a, flno as f, avg(price) as av, count(*) as c
      from bookings
      group BY airline, flno) t, flights f
where t.a=f.airline and t.flno=f.flno;
```

```
Q2.
```

```
select f.airline, f.flno, f.dep, avg(price), count(b.dep_date)
from flights f, bookings b
where f.airline=b.airline and f.flno=b.flno;
group by f.airline, f.flno, f.dep;
```

SPIDER, a pioneer in text-to-SQL benchmark, and some of the follow-up works perform their evaluations on the following clauses: *select*, *where*, *group by*, *order by*, and *SQL keywords*. The SQL keyword component includes all SQL keywords in a query, treated as a set, excluding table and column names and operators such as BETWEEN, LIKE. MAX, etc. The clause list does not include the `from` and `having` clauses. One may skip the `from` clause on the basis that the relation names are usually listed in other components to disambiguate the attribute names, but there is no good reason to skip the `having` clause in the evaluation.

With each component treated as a set, the performance of a candidate model is compared to the reference gold query on each component based on exact set matching and is reported in terms of precision, recall and F1 measure.

A challenge in component matching is that the same component can appear multiple times when the query consists of multiple subqueries. Those subqueries can be nested inside each other or be connected using operators such as `in`, `exists`, `union` and `except`. It is not clear how a component matching should be performed, for example, when there are multiple `select` or `where` clauses.

On the other hand, from an end user's perspective, one wants to know if a predicted query is equivalent to a gold query. It is clear that if a model correctly predicts all components of a reference query (as discussed), then we can call the model correct on that query under an *exact matching*. This is a strict measure though and only allows possible ordering differences within each clause (e.g. the ordering of attributes in the `from` clause or the ordering of predicates in the `where` clause). It is less likely to produce false positives at least on queries where a clear alignment between the query components can be established but many false negatives are expected. It should be noted that there cannot be false positives under an exact matching when there is a clear injective mapping between the components of a model generated query and the reference. However, an injective mapping may not be well-established, for example, when one query uses nesting and the other does not.

There are a few relaxation of exact set matching to better address cases where two queries are close but not identical. In particular, the Spider benchmark includes a category called *exact set matching without values*, which ignores the values in queries, allowing a predicted query to match the gold query even if the literals are not the same. This simplifies the problem when the values given in natural language queries are different than those stored in the database. For example, a predicted query may have the predicate `nationality = ‘‘Canadian’’` but the gold query can instead have `nationality = ‘‘Canada’’`. Another relaxation is to address cases where some but not all components of a predicted query match the gold query. Under an exact set matching, those queries are deemed incorrect with a score of zero even if many components of a large query are predicted correctly. *Partial component matching* [Hazoom et al., 2021] resolves this by evaluating the correctness of each query component separately and using the average performance of the components as the performance of a predicted query.

Execution Accuracy

As per the definition of equivalence, two queries are equivalent if their answer is the same on all possible instances. One way to test for the equivalence of a candidate query to a reference is to generate test database instances and check if both queries return the same results on those instances. Finding even one instance on which the results are different is sufficient to call the two queries not equivalent. Hence, running the queries on even a small subset of the instances can flag many candidates that are not equivalent to a reference query. However, one cannot test all possible instances due to the sheer number of such instances, and it may be difficult to tell with some certainty that two queries will produce the same result on all instances. One can only say that on the instances that are tested, two queries return the same result.

Compared to an exact matching of queries which can miss many generated candidates that are equivalent but not identical (subject to minor ordering differences), execution accuracy cannot miss a candidate query that is equivalent to a reference. However, many queries that are not equivalent can return the same result on a test instance and they all can be wrongly deemed as equivalent. A challenge in an execution-based evaluation is finding instances that cover as many edge cases as possible and are effective in detecting many candidate queries that are not equivalent to the reference.

Test suite accuracy

The term *execution accuracy*, as defined in text-to-SQL benchmarks [Yu et al., 2018e], refers to the accuracy on a single database instance. This means, a model-predicted query is tagged *correct* if it returns the same result as the gold query on the test database instance. Clearly two non-equivalent queries can produce the same result on a single database instance and be wrongly deemed as equivalent. Test suite accuracy for SQL queries is an attempt to reduce the number of such false positives by testing queries on a carefully chosen collection of database instances, also known as a test suite. The choice of database instances that form a test suite depends on the gold query that is being tested. Ideally we want a test suite that distinguishes any query that is not equivalent to the gold query. That means there must be at least one instance in the test suite that flags such queries. In practice, a good test suite for a reference query is expected to provide a high code coverage, resembling code coverage in software testing, differentiating queries that may be close but not equivalent.

If g denotes a gold query and q denotes a model-generated query, a database instance t distinguishes the two queries if $[[g]]_t \neq [[q]]_t$, meaning that the query results are not the same. A test suite T distinguishes the two queries if for at least one instance $t \in T$, we have $[[g]]_t \neq [[q]]_t$. A problem is that one has to construct a test suite without having model-generated queries and without knowing potential false positive queries, i.e. queries that produce the same result as the gold query but are not equivalent to the gold query.

Based on the assumption that two queries are likely to produce the same result if their surface forms are similar, one may generate a set of *neighbour queries* as those that are close to the gold query in surface form but are likely different semantically [Zhong et al., 2020b]. For example, changing the predicate “age > 30” in a query to “age > 28” is likely to change the query semantics but may produce the same result on some database instances. Hence constructing a test suite may involve finding or generating such neighbour queries and ensuring that the test suite includes enough different instances to distinguish between as many neighbour queries as possible.

The generation of database instances is done through a testing technique, referred to as fuzzing, where database instances are generated by uniformly sampling from the domain of each column while ensuring all domain, primary and foreign key constraints are satisfied. Any database instance that can distinguish at least one neighbour query from the gold query is a candidate to be included in the test suite. The goal is to distinguish as many neighbour queries as possible while keeping the test suite size small. The selection of database instances can be done in a greedy fashion where each new instance to be added must distinguish at least one neighbour query that cannot be distinguished by those instances that are already in the test suite.

Test suite accuracy is applied to the development set of Spider with 1034 queries over 20 different database schemas. For each gold query, on average 94 neighbour queries are generated. It is shown that checking a single database instance per schema fails to distinguish 5% of the neighbour queries whereas this number drops to less than 1% using a test suite with 600 database instances per schema.

5.3.2 SEMANTIC EQUIVALENCE

Despite the NP-completeness result for checking the equivalence of conjunctive queries, an equivalence checking for conjunctive queries can be done in many cases since queries are not generally expected to be long. Consider the following two queries on the flight schema discussed in Cahpter 3.

```
Q1.
  select distinct passenger
  from Bookings b, Flights f1, Flights f2
  where b.airline=f1.airline and b.flno=f1.flno and
        f1.origin=f2.origin and f1.destination='Toronto';

Q2.
  select distinct passenger
  from Bookings b, Flights f
  where b.airline=f.airline and b.flno=f.flno and
         f.destination='Toronto';
```

Both $Q1$ and $Q2$ are conjunctive queries with only conjunctions in the where clause, and suppose we want to find out if they are equivalent. One way to check for a possible equivalence between two conjunctive queries is to check for an equivalence between their tableaus. In a *tableau* notation, each query becomes a database with both variables and constants appearing in rows. More specifically, each tableau consists of a summary tuple, representing the query output, and a set of rows in the tableau body, each representing a relation in the from clause. A row can have both variables and constants. The variables that appear in the summary tuple are often referred to as distinguished variables, and they are treated a bit different than other variables.

	passenger	airline	flno	origin	destination
Bookings b	x	m_1	m_2		
Flights f1		m_3	m_4	m_5	m_6
Flights f2		m_7	m_8	m_9	m_{10}
Summary	x				

Table 5.4: A tableau with three relations

Consider the tableau shown in Table 5.4, which represents $Q1$ without the conditions in the where clause. To simplify our tableau notation, we only show the columns that are referenced in our queries (ignoring columns such as price and dist which are not referenced). Each condition in the where clause constrains a variable to take the same value as another variable or a constant. For example, the predicate *b.airline=f1.airline* enforces that the variables m_1 and m_3 in the query tableau must have the same values. Through a renaming of variables or replacing them with constants, we can obtain a tableau that represents all conditions in the where clause. Tables 5.5 and 5.6 show such tableaus for $Q1$ and $Q2$ respectively. One now has to detect if the two tableaus are equivalent. A guiding principle here is a classic Homomorphism Theorem.

	passenger	airline	flno	origin	destination
Bookings b	x	m_1	m_2		
Flights f1		m_1	m_2	m_5	'Toronto'
Flights f2		m_7	m_8	m_5	m_{10}
Summary	x				

Table 5.5: $Q1$ tableau

Theorem 5.3 Homomorphism Theorem for Queries. *Let q=(T,u) be a query with tableau T and variables and constants u, and q'=(T',u') be a query with tableau T' and variables and constants u', both over the same schema. q is contained in q', denoted as* $q \subseteq q'$*, if there is a homomorphism from (T',u') to (T,u).*

	passenger	airline	flno	origin	destination
Bookings b	x	n_1	n_2		
Flights f		n_1	n_2	n_5	'Toronto'
Summary	x				

Table 5.6: $Q2$ tableau

For an equivalence between $Q1$ and $Q2$, we need to show $Q1 \subseteq Q2$ and $Q2 \subseteq Q1$ holds between their tableaus. In particular, for $Q1 \subseteq Q2$, we need to find a substitution that maps every row in the tableau of $Q2$ to a row in the tableau of $Q1$, with each distinguished variable mapped to the same variable and each non-distinguished variable mapped to a variable or a constant. Here is one such mapping:

$$x \mapsto x, \;\; n_1 \mapsto m_1, \;\; n_2 \mapsto m_2, \;\; n_5 \mapsto m_5.$$

Similarly for $Q2 \subseteq Q1$, we need a substitution that maps every row in the tableau of $Q1$ to a row in the tableau of $Q2$. Here is one:

$$x \mapsto x, \;\; m_1 \mapsto n_1, \;\; m_2 \mapsto n_2, \;\; m_5 \mapsto n_5, \;\; m_7 \mapsto n_1, \;\; m_8 \mapsto n_2, \;\; m_{10} \mapsto `Toronto'.$$

This establishes the equivalence between the two tableaus in Tables 5.5 and 5.6 and consequently the equivalence between queries $Q1$ and $Q2$.

5.4 HUMAN-CENTRIC EVALUATION

Human evaluation is typically considered the most important evaluation in many IR and NLG systems. For example, search engines such as Google hire many users to evaluate their new products and features before they are pushed to the production [Google, 2022]. This is because calling a generated or retrieved response *correct* or *relevant* is often subjective. Also a human evaluation can replicate how these systems are used in more realistic settings. At the same time, human evaluation in many cases is challenging due to the cost and the resources that are needed. While crowd sourcing platforms such as Amazon Mechanical Turk have been a driving force in reducing this cost by breaking down the work to smaller tasks and distributing the tasks to a globally distributed workforce that is usually cheap, quality control has been a major challenge in using those platforms.

5.4.1 EXPRESSING AND PERFORMING THE TASKS

Since NLIDB systems usually take the description of tasks in English, humans may be employed in describing the tasks. Humans may also be involved in interacting with the system and resolving query interpretations, as extensively discussed in Chapter 7. A human evaluation is usually the best mean to evaluate the effectiveness of the interactions and the effort required for users to obtain an answer to their questions. The *effort* may be measured in terms of the time required to resolve possible ambiguities in a question and sometimes the initial time for users to be acquainted with the NLIDB system and its functions. The skill level of the users may also be taken into account.

5.4.2 QUALITY OF GENERATED QUERIES

As a general evaluation scheme for NLIDB systems, humans may be asked to review the generated queries and assess its quality either overall or along some dimensions (e.g. correctness, readability, etc.). However, unlike general NLG system where the generated text is in natural language and can be evaluated by human, for example, in terms of accuracy of translation or fluency, evaluating queries in SQL and other formal languages can be more challenging. Users must know well not only the schema and the query language but also the intention of the question. When the queries get long or more complex, it is not easy even for experts proficient in SQL to detect if the query is correct. The authors of this book experience this when evaluating queries that are submitted by students in a database course for correctness. Users may run the queries on some database instances to help them detect if the query is correct, and as discussed for the case of execution accuracy, this process is also error-prone since not all database instances can be checked.

5.4.3 PERFORMANCE IN DOWN-STREAM TASKS

The performance of NLIDB systems should ideally be evaluated by the same users these systems are aiming to serve, i.e. people who have questions or need analytics from the database and they cannot express their needs in SQL. In such settings, the performance metrics can be the number of tasks or transactions completed in a time unit, the number of users who enjoy using the system, inter-user agreement, the complexity of the tasks performed, etc. We are not aware of any NLIDB system that is evaluated in such down-stream tasks and by the real users of these systems. The evaluation instead is performed using turn-based metrics where the system is evaluated in each step assuming other steps or components are correctly completed. For example, one may assume a reference query is available and is correct, the task is sufficiently expressed, the query can be answered using the database, etc. Each of these components may not be present or may introduce errors. Hence, an evaluation with these assumptions cannot measure how well the system will perform in real settings.

5.5 OTHER PERFORMANCE METRICS

The aforementioned evaluation metrics discussed so far are all focused on the correctness of the generated SQL queries. Correctness is only one metric, even though it is usually the most important one. However, the performance of a model may also be measured in terms of other metrics such as resource consumption, robustness to error and variations in input, etc. For example, one may prefer a model that requires much less resources even if it is not the top-performing model. In this section, we discuss some of these other performance metrics.

5.5.1 RESOURCE CONSUMPTION

Given two systems that are comparable in terms of the quality of their results (e.g. the accuracy of the generated SQL queries), we generally prefer a system that is more efficient in terms of the resources that are needed. While resource consumption has not been the focus of many text-to-sql benchmarks, there has been some attempts to measure (a) the running time and the memory footprint of text-to-sql systems and (b) the time and the memory usage of running the generated SQL queries [Gkini et al., 2021].

Resource consumption can be important for system deployment as it directly reflects the associated cost for providing lay people access to a database. On the other hand, measuring these resources is only meaningful if the system being evaluated achieves an acceptable level of accuracy. Otherwise, there is simply no value in using less resources when the generated queries are wrong. A caveat is that an acceptable accuracy on some benchmark may not translate to an acceptable accuracy in a real setting. Hence, measuring resource consumption may be applicable in more controlled settings where a limited functionality is sought and those functions can be easily tested.

5.5.2 QUERY HARDNESS

As queries can vary in hardness, the performance is also expected to vary. We discussed a few SQL query dimensions in Chapter 3. Each dimension adds a level of complexity to queries, generally making it more challenging for a model to get all components right. One reason for this is that the uncertainty or error usually propagates exponentially with the number of components. The space of possible queries also increases extensively with the number of query components, and this has implications on both the size of the training data as well as the running time. Spider defines four levels of query harness based on the number of query components: easy, medium, hard and extra hard. For example, a query joining two tables and using group by is considered a medium query, whereas a query joining three tables, with group by and having clauses is considered a hard query. An example of an extra hard query is one that uses an aggregation function in the select clause, a nested form of negation in the body (e.g. using "NOT IN"), and the subquery joins at least two tables and has some additional conditions in its where clause. Table 5.7 shows the exact matching accuracy of a few models for both question and query splits (see Section 5.5.4 for more details on question vs query split) and different levels of query hardness [Yu et al., 2018e].

5.5.3 ROBUSTNESS TO NOISE AND AMBIGUITY

Text-to-SQL systems typically ground the terms in questions by mapping them to tokens in the database, and they often rely on lexical matching between question words and database tokens that may appear in attribute names, attribute values, and table names.

Model	Easy	Medium	Hard	Extra Hard	All
Question Split					
Seq2Seq	24.3%	9.5%	6.3%	1.5%	11.2%
Seq2Seq+Attention Dong and Lapata [2016]	31.2%	13.9%	11.6%	3.3%	15.5%
Seq2Seq+Copying	30.0%	12.6%	10.0%	3.3%	14.8%
Iyer et al. [2017b]	18.6%	12.0%	9.0%	1.8%	9.8%
SQLNet [Xu et al., 2017]	36.2%	15.6%	7.9%	4.9%	17.4%
TypeSQL [Yu et al., 2018a]	48.6%	38.2%	18.1%	19.8%	34.3%
Query Split					
Seq2Seq	17.9%	2.7%	1.3%	0.6%	5.4%
Seq2Seq+Attention Dong and Lapata [2016]	17.9%	2.9%	1.8%	1.3%	5.7%
Seq2Seq+Copying	15.1%	3.4%	1.0%	1.3%	5.2%
Iyer et al. [2017b]	7.9%	2.1%	1.3%	1.1%	3.1%
SQLNet [Xu et al., 2017]	23.7%	5.9%	2.3%	0.3%	8.3%
TypeSQL [Yu et al., 2018a]	29.6%	6.1%	2.3%	0.3%	9.7%

Table 5.7: Exact match accuracy on queries with different levels of hardness [Yu et al., 2018e]

This reliance on lexical matching with database tokens makes these systems vulnerable to question variations due to misspelling and paraphrasing and possible ambiguities in mappings.

Term ambiguity

When a question word appears in multiple places in the database, selecting a correct mapping may not be straightforward. For example, *Paris* can be the city of a customer, the source or destination of a flight, and the name of a person. With two such terms in a question, the space of possible mappings is quadratic on the number of matches per term, and with three terms, the space becomes cubic. The robustness to term ambiguity may be measured by increasing the number of question words that appear multiple times and in multiple roles in the database. The more there are terms with multiple roles, the larger the search space is for grounding the words of a question.

Synonyms and misspellings

Consider the question "Which model of the car has the minimum horsepower?" from the Spider development set. The schema includes tables *car_name* and *cars_data*, both matching the word "car" in the question. The table *cars_data* has an attribute named "horsepower" and the table *car_name* includes an attribute named "model", both matching the exact same words in the question. One may have expressed the same question as "Which automobile model has the least force" or "which car has the least force." With one such paraphrasing of the question, the mapping of the question to database elements becomes more challenging.

In a study of robustness against synonym substitution in questions, Gan et al. [2021a] introduce Spider-Syn, where Spider questions are modified by human annotators by replacing some of the schema-matching question words with their synonyms. This change has a big impact on the performance of the models that work well on the Spider dataset. For example, the exact match accuracy of RAT-SQL+BERT [Wang et al., 2020c], a recent competitive model, drops from 69.7% on Spider to only 48.2% on Spider-Syn, both on the development sets. A similar drop in accuracy is expected in the presence of misspellings.

As a general approach to improve the robustness, adversarial training may be used where the model is trained on the data augmented with adversarial examples.

5.5.4 ADAPTABILITY TO UNSEEN DATABASES AND QUESTIONS

An evaluation must ideally represent how a model is expected to perform in real settings with unseen databases and questions or queries.

Adaptability to unseen questions

It has been a common practice to split questions to train and test sets, which is acceptable when questions are treated independent. However, in many cases the questions in train and test sets are not independent. In an interesting experiment, Finegan-Dollak et al. [2018] construct a template-based slot filling baseline where a model is trained to select for a given question a query template from the training set and words from the question that can fill the slots in the template. Clearly the baseline cannot generalize to any query template that is not seen in the training set. Dividing the questions into train and test sets in a few commonly used datasets in the literature (including Advising, ATIS, GeoQuery and Restaurants), the authors show that the simple baseline is competitive with some of the state-of-the-art baselines, achieving an exact match accuracy close to 100% in some datasets. When there are more questions than unique query templates, there is a good chance that a query template that matches a test question is already seen in the training set. As shown in Table 5.8 for different datasets, the ratio of questions to queries directly correlates with the performance of the template-based slot filling baseline. It is clear from this experiment that dividing questions into training and test sets cannot measure how well a model generalizes to unseen examples especially when many questions map to the same or very similar queries. The performance of the template-based baseline is not reported on WikiSQL and Spider, but based on the ratio of questions to queries for these two datasets, the accuracy is expected to be low.

An alternative is to divide queries and not questions. An exact matching of queries will not detect cases where the queries are different only due to variations in naming aliases, the ordering of column and table names, the ordering of predicates, etc. One may need to canonicalize the queries by alphabetically ordering column and table

Dataset	#Q	#queries	ratio	#patterns	accuracy
ATIS	5,280	1,134	4.7	751	46
GeoQuery	877	259	3.4	98	57
Academic	196	189	1.1	92	0
Advising	4,387	214	20.5	174	80
IMDB	131	99	1.3	52	0
Yelp	128	122	1.0	89	1
Restaurants	378	23	16.4	17	95
WikiSQL	80,655	51,159	1.6	488	?
Spider	9,693	5,183	1.9	?	?

Table 5.8: Dataset statistics and the exact match accuracy of the template-based slot filling baseline (as reported by Finegan-Dollak et al. [2018])

names and standardizing the aliases, capitalization, spacing, etc. Even with a canonicalization, two queries that are deemed different can show very similar patterns. For example, the SQL query templates for questions "What are direct flights from Toronto to Ottawa" and "What are direct flights from Edmonton to Montreal" are the same despite the lexical difference. Detecting this type of duplication may require annotating query words that are copied from questions to queries.

Adaptability to unseen databases

Early works in the area were developed for a single database where each dataset usually consist of a few closely related tables in a domain. This means the models were trained and tested on the same database and the database schema was not considered as a model parameter. All datasets discussed in Section 5.2 except WikiSQL and Spider also fall into this category.

In the WikiSQL task, a model takes as input a question and the schema of a single table that answers the question. In a typical train-and-test split, the model does not see the schema of the test database during training and is expected to generalize to unseen tables. However, with each question formulated on a single table from Wikipedia and with no join in the questions, the queries follow a simple structure and the query patterns cannot deviate much from those seen in the training. Hence a strong performance on this dataset cannot warrant generalizability to databases that have more than one table or the table structure is different than those in Wikipedia.

The Spider task has been a major step forward in evaluating models that aim to adapt to unseen databases. This is achieved in two parts. First, the dataset includes 200 databases, each consisting of a few tables in a domain, which may be linked using primary and foreign key relationships. The databases are in different domains and are collected from multiple sources. Second, while 160 databases and their questions are published to the community as the training and development sets, the remaining 40 databases and their questions are not published. This means the models have not seen the databases they are tested on. When the dataset was released in 2018, the exact match accuracy on the test set for many of the models did not exceed 0.11. As of July 2022, the leaderboard for Spider shows an accuracy in the the range of 0.72. Clearly the models today either are better at adapting to unseen databases or have been learning the patterns and the parameters of the test set.

5.6 TOP PERFORMING MODELS

At the time of this writing, Spider is considered the most complex and cross-domain dataset for evaluating text-to-SQL parsing, and many models have been competing to take a top spot in the benchmark leaderboard. We review some of the characteristics of these models sitting at the top of the leaderboard [9].

5.6.1 THE MODELS

Table 5.9 shows five top-performing models on the Spider test set in terms of execution accuracy, and Table 5.10 gives five top-performing models on both the development and test sets in terms of exact set match accuracy where each SQL query is decomposed into a few clauses and a set comparison is conducted on each clause (as discussed in Section 5.3.1). As expected, the execution accuracy is generally higher on the test set since the test is done on a single database instance and possible false positives are not accounted for.

Rank	Date	Model	Accuracy
1	Jun 4, 2022	RASAT + PICARD	75.5
2	May 8, 2022	T5-SR	75.2
3	Jul 14, 2021	T5-3B + PICARD [Scholak et al., 2021]	75.1
4	May 18, 2022	SeaD + SP	74.1
5	May 4, 2021	RATSQL + GAP + NatSQL [Gan et al., 2021b]	73.3

Table 5.9: Top-performing models and their execution accuracy on the test set of Spider (as of July 25, 2022)

There are some common characteristics of the reported models that seem to be essential in pushing their performance. We review two of them here.

5.6.2 LARGE LANGUAGE MODELS

Large pre-trained language models are being widely used in many language processing tasks as few-shot learners with great success [Brown et al., 2020a, Shaw et al., 2021]. Many of these models are replacing some of those complex models that are specialized for a single task and are costly to build and train. The same trend is seen on the task of text-to-SQL over the past few years, for example, with large pre-trained models with-

[9] https://yale-lily.github.io/spider

Rank	Date	Model	Accuracy	
			(dev)	(test)
1	May 23, 2022	G^3R + LGESQL + ELECTRA	77.2	72.6
2	May 8, 2022	T5-SR	77.2	72.4
3	Sep 1, 2021	S^2SQL + ELECTRA [Hui et al., 2022]	76.4	72.1
4	Jun 1, 2021	LGESQL + ELECTRA [Cao et al., 2021]	75.1	72.0
5	Jul 14, 2021	T5-3B + PICARD [Scholak et al., 2021]	75.5	71.9

Table 5.10: Top-performing models and their exact set match accuracy on both development and test sets of Spider (as of July 25, 2022)

out finetuning on text-to-SQL taking the lead on the Spider leaderboard, as shown in Tables 5.9 and 5.10. It has been recently shown that using the Davinci-Codex model with no text-to-SQL training (i.e. a zero-shot setting) on Spider development set is doing well with an execution accuracy of 0.67 [Rajkumar et al., 2022]. The same model beats T5-3B trained on the whole GeoQuery training set in a zero-shot setting. Under an n-shot setting where n random samples of the most frequent question-query templates are provided, Codex shows better adaptation to the input, outperforming a finetuned T5 on the Scholar dataset.

5.6.3 CONSTRAINING THE DECODER

A problem with using large pre-trained language models is that the output may not be consistent with the schema that is being queried or even may not be a syntactically correct SQL query. One may constrain the generation within the auto-regressive or semi-auto-regressive decoding to token sequences that correctly parse to SQL [Lin et al., 2020b, Rubin and Berant, 2021]. This makes them less compatible with generic pre-trained language model decoders. A better solution, adopted by PICARD [Scholak et al., 2021], is to incrementally parse the output and filter out invalid queries within the beam search. This is done by restricting the prediction at each step of the beam search to top-k tokens with the highest probability. Also any token sequence that fails the incremental parsing and/or violates a constraint is assigned a score of $-\infty$ to cut further searches.

Four modes of restrictions can be set in PICARD. One mode is *lexing* where the generated tokens pass a lexical filter to detect non-SQL keywords, column names that don't appear in tables, etc. At this mode, there is no restriction on the ordering of the tokens. The next level is *parsing without guards* where the output is checked for grammatical mistakes. The ordering of the tokens is important in this mode. For example, if a tuple variable is assigned to a table and the same tuple variable is used with a column name later, it is checked if the table representing the tuple variable has that column. The strongest mode is *parsing with guards* where all sorts of constraints are checked. For example, a variable must be in the scope before it is referenced; or not more than one table in the same scope can have the same column names. Finally, there is also a mode where no restriction is set on the output. PICARD improves the performance of all T5 models (i.e. T5-Base, T5-Large, T5-3B) on the Spider benchmark. As shown in Table 5.9, two of the top three models on the leaderboard are using PICARD.

5.7 FURTHER READING

For a review of different evaluation schemes in text generation, though not specific to NLIDB systems and text to SQL, the readers are referred to a recent survey [Celikyilmaz et al., 2020]. Our coverage of exact match accuracy, execution accuracy and test suite accuracy are based on Yu et al. [2018e] and Zhong et al. [2020a]. For more information on query equivalence, query containment, and tableau representation of queries, the readers can check the classic book covering the foundations of databases [Abiteboul et al., 1995] and some recent developments (e.g. Chu et al. [2017, 2018]). The subject of robustness for text-to-SQL models has been getting some attention lately, for example, with robustness against synonym substitution [Gan et al., 2021a] and adversarial table perturbation [Pi et al., 2022].