Chapter 4
Text to Data

This chapter covers text-to-data translation. It begins by providing a historical perspective, briefly covering early relevant work on semantic parsing and meaning representation before moving to newer approaches that utilize word embedding, semantic compositionality, knowledge graph, and large language models. Three key topics are covered in-depth: (1) converting sentences to structured form, (2) neural semantic parsing, and (3) text-to-SQL models, systems and benchmarks.

4.1 Introduction

Natural language interfaces to databases include two essential parts: (1) natural language understanding (NLU), also known as text-to-data and (2) natural language generation (NLG), or data-to-text. These two components correspond to the arrows connected to the user in Figure 2.2. The former (NLU) is primarily used to transform a user's request into a semantic representation or into executable code. This transformation can happen in the form of either an input question or a follow up user input during an interaction with the system. The latter (NLG) is not available in all NLIDB systems but, when used, it provides a natural way to display the results of the interaction in natural language form.

In this chapter we will be addressing the components of the text-to-data part of the pipeline. This includes meaning representation, semantic parsing, text to SQL translation, and related topics. Chapter 6 will cover data-to-text generation.

Let's start with a simple example. Suppose that the user is interested in the salaries of coaches in the Big Ten (American) football conference, and that such information is available in a relational database. The user can ask a question in natural language, e.g., "*Who is the highest paid football coach in the Big Ten and how much is he paid?*" instead of having to write the question as a SQL query, such as the one shown here.

```
SELECT first_name, last_name, MAX(salary)
FROM football_coach
WHERE conference LIKE 'Big Ten'
GROUP BY first_name, last_name
LIMIT 1
```

Before such a natural language question can be executed, it must be converted to a database query format, usually SQL. This automatic conversion process traditionally includes multiple components, corresponding to the Query Understanding and Query Translation boxes in Figure 2.2.

Specifically, the components for Query Understanding include:

1. Syntactic parsing – understanding the grammatical structure of the sentence or query.
2. Semantic parsing – understanding the meaning of the sentence or query.

The components for Query Translation include:

1. Mapping questions to the database schema – representing elements of the question as entities, relations, and sets, as well as functions that operate on such entities and relations.
2. Grounding symbols to data values – for example, determining that "*Big Ten*" refers to the name of a sports conference.
3. Reasoning about symbols in an abstract level – for example, answering questions that involve arithmetic or logical reasoning.

One way to think about the understanding process is that we represent the world in the form of entities and relations, as well as functions that operate on these entities. In this approach, nouns (e.g., "*student*") in the natural language sentence correspond to entities (e.g., a specific student), sets of entities (e.g., all students), or properties of entities (e.g., a given student's age), adjectives (e.g., "*tall*") correspond to properties, and verbs (e.g., "*takes*") correspond to functions or relations. Through semantic compositionality, the meanings of the individual words in the sentence or query are then combined into a meaning representation of the entire sentence or query.

4.1.1 Natural Language Understanding and Natural Language Generation

Communication typically includes at least two major roles: a speaker and a listener (Figure 4.1). The speaker communicates his or her intention, e.g., goals and beliefs, by converting them to natural language through a sequence of steps. This process covers tactical generation, followed by synthesis (e.g., text or speech). The listener, in turn, is involved in perception (either reading or hearing a natural language sentence uttered by the speaker), then performs syntactic, semantic, and pragmatic interpretation, and finally performs the act of incorporating the newly communicated information through internalization or understanding. Typically, the speaker and listener also share some context through grounding. For example, in a dialogue between a user and his or her bank, the names of the user and the bank, as well as the account number may be part of this shared context, even before the dialogue has begun.

Fig. 4.1: Natural Language Understanding (U) and Natural Language Generation (G).

In the classic Natural Language Processing pipeline, an input sentence or paragraph is first analyzed syntactically (Figure 4.2), e.g., by labeling each word with a part of speech tag, such as a noun (N), an

adjective (J), a preposition (P), a verb (V), etc., and then building a syntactic parse tree (Figure 4.3) of the sentence. Consequently (as in Figure 4.4), the sentence is converted to a **meaning representation** (MR). This MR is then augmented using grounding, commonsense knowledge, and reasoning, and can then be used as new knowledge to be injected into the system.

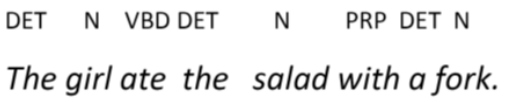


Fig. 4.2 Representing a sentence using part of speech tags.

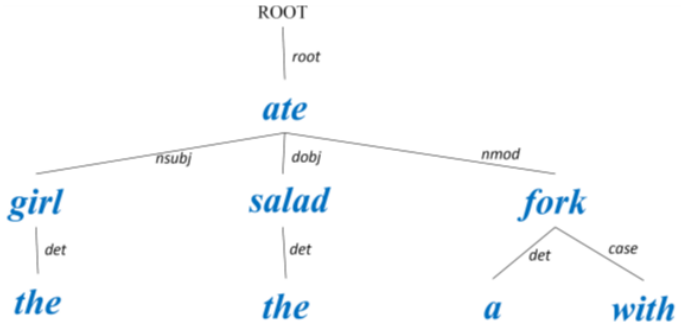


Fig. 4.3 Representing the sentence using a syntactic dependency tree. In this representation, the root of the parse tree is connected to the main verb of the sentence. Each word of the sentence is represented as a node in the tree. The word in a child node modifies the word in the parent node (e.g., "*the*" modifies "*girl*")

Fig. 4.4 Logical reasoning example. Here is the gist of the story in the example: The girl is hungry. She eats a salad. Food makes you not hungry. Salad is food. As a result, later on the girl is no longer hungry.

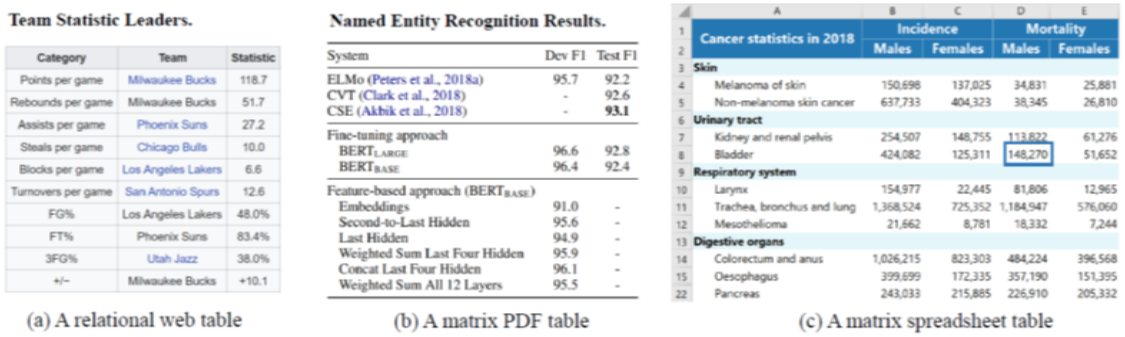
Facts:	Girl (g_1)
	Salad (s_1)
	Fork (f_1)
	EatingEvent $(e_1) \wedge$ Eater $(e_1, g_1) \wedge$ Eaten $(e_1, s_1) \wedge$
	$\wedge$ Instrument $(e_1, f_1) \wedge$ Time $(e_1$,"9am")
World knowledge:	$\forall x$: Salad $(x) \Rightarrow$ Food (x)
Inference:	$\forall z, t_0, t_1, y, e$: Hungry $(z, t_0) \wedge$ EatingEvent$(e) \wedge$ Eater $(e, z) \wedge$
	$\wedge$ Eaten $(e, y) \wedge$ Time $(e, t_0) \wedge$ Food $(y) \wedge$
	$\wedge$ Precedes $(t_0, t_1) \Rightarrow \neg$ Hungry (z, t_1)
Conclusion:	$\neg$ Hungry $(g_1$,now)

In a NLIDB, communication usually takes place between a human who asks questions and a database in which useful knowledge is stored in tabular or graph form. In order to understand the user, the system performs semantic parsing and meaning representation (e.g., in SQL format). Additionally, the database has to be represented in a format suitable for processing. In addition, replying to the user may involve table to text generation which we cover in Chapter 6. Finally, the communication can also be more interactive beyond text-to-data and data-to-text. We cover more topics related to interactivity in Chapter 7.

4.1.2 Historical Overview

Providing natural language interfaces to databases have been viewed by many as the holy grail for removing barrier for casual users to access databases. A major development took place in 2008 when the WebTables dataset was created [34, 35]. This collection of 154 million high-quality tables was extracted from 14 billion HTML tables and was, at the time, by several orders of magnitude, the largest collection of database tables available.

We should note that multiple types of tables exist. Three of the main types of tables (**relational tables**, matrix PDF, and matrix spreadsheet) from [93] are shown in Figure 4.5. Most of the work described in this chapter is about flat (non-hierarchical) relational tables.



Team Statistic Leaders.

Category	Team	Statistic
Points per game	Milwaukee Bucks	118.7
Rebounds per game	Milwaukee Bucks	51.7
Assists per game	Phoenix Suns	27.2
Steals per game	Chicago Bulls	10.0
Blocks per game	Los Angeles Lakers	6.6
Turnovers per game	San Antonio Spurs	12.6
FG%	Los Angeles Lakers	48.0%
FT%	Phoenix Suns	83.4%
3FG%	Utah Jazz	38.0%
+/−	Milwaukee Bucks	+10.1

(a) A relational web table

Named Entity Recognition Results.

System	Dev F1	Test F1
ELMo (Peters et al., 2018a)	95.7	92.2
CVT (Clark et al., 2018)	-	92.6
CSE (Akbik et al., 2018)	-	**93.1**
Fine-tuning approach		
$BERT_{LARGE}$	96.6	92.8
$BERT_{BASE}$	96.4	92.4
Feature-based approach ($BERT_{BASE}$)		
Embeddings	91.0	-
Second-to-Last Hidden	95.6	-
Last Hidden	94.9	-
Weighted Sum Last Four Hidden	95.9	-
Concat Last Four Hidden	96.1	-
Weighted Sum All 12 Layers	95.5	-

(b) A matrix PDF table

	A	B	C	D	E
1	Cancer statistics in 2018	Incidence		Mortality	
2		Males	Females	Males	Females
3	**Skin**				
4	Melanoma of skin	150,698	137,025	34,831	25,881
5	Non-melanoma skin cancer	637,733	404,323	38,345	26,810
6	**Urinary tract**				
7	Kidney and renal pelvis	254,507	148,755	113,822	61,276
8	Bladder	424,082	125,311	148,270	51,652
9	**Respiratory system**				
10	Larynx	154,977	22,445	81,806	12,965
11	Trachea, bronchus and lung	1,368,524	725,352	1,184,947	576,060
12	Mesothelioma	21,662	8,781	18,332	7,244
13	**Digestive organs**				
14	Colorectum and anus	1,026,215	823,303	484,224	396,568
15	Oesophagus	399,699	172,335	357,190	151,395
22	Pancreas	243,033	215,885	226,910	205,332

(c) A matrix spreadsheet table

Fig. 4.5: Three types of tables (relational, matrix PDF, and matrix spreadsheet). (Image from [93]).

Historically, two different communities have been making contributions to NLIDB: the NLP community (e.g., [380]) and the database community (e.g., [189]). As a sample from the literature, our first example (Figure 4.6) shows how semantic parsing is used to convert natural language queries to logical forms using a meaning representation based on **lambda calculus**. Our second example (Figure 4.7) is from the NaLIR system [189] that uses the Stanford dependency parser and a representation of MAS (Microsoft Academic Search) database. We note here that both of these examples are for domain-specific tables, and not general domain datasets.

What states border Texas
$\lambda x.state(x) \wedge borders(x, texas)$

What is the largest state
$\arg\max(\lambda x.state(x), \lambda x.size(x))$

What states border the state that borders the most states
$\lambda x.state(x) \wedge borders(x, \arg\max(\lambda y.state(y), \lambda y.count(\lambda z.state(z) \wedge borders(y, z))))$

Fig. 4.6 GeoQuery example. (Image from [380]).

The earlier book by Li and Rafiei [198], two of the co-authors of this current book, presented a thorough introduction of this area. It was followed more recently by two excellent new materials. One of them is an

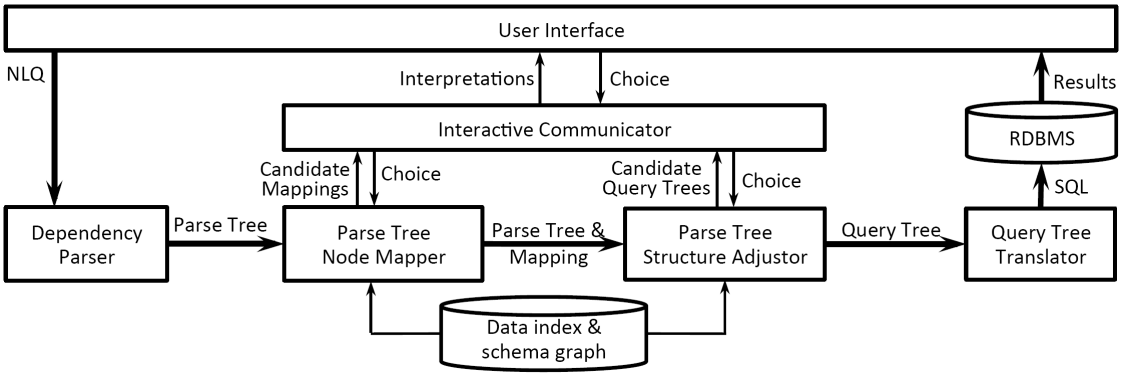


Fig. 4.7: Pipeline from Li and Jagadish 2014. (Image from [189]).

accessible introductory survey [262] and the other one [240] is a great tutorial. They split the papers on text to SQL into three categories. The first category is entity-based, where query entities and the relationships between those entities are recognized before an internal representation (using abstract syntax trees or other intermediate representations). The second category includes the "early" (pre-neural) systems, such as NaLIR [187], Athena [272], which uses an ontology, and Athena++ [279] while the papers in the third category are based on deep learning and operate in a single pipeline, e.g., HydraNet [209], RAT-SQL [326], SQLNet [352], TypeSQL [365], SyntaxSQLNet [367], and Seq2SQL [388].

Two other recent tutorials, [164] and [165], list a number of challenges, namely language ambiguity, reference resolution and schema linking, inference, the vocabulary gap, typos, the complex structure of SQL statements, and the structure of the database. A recent survey by the same authors [166] also provides a timeline (Figure 4.8) of the developments in this area.

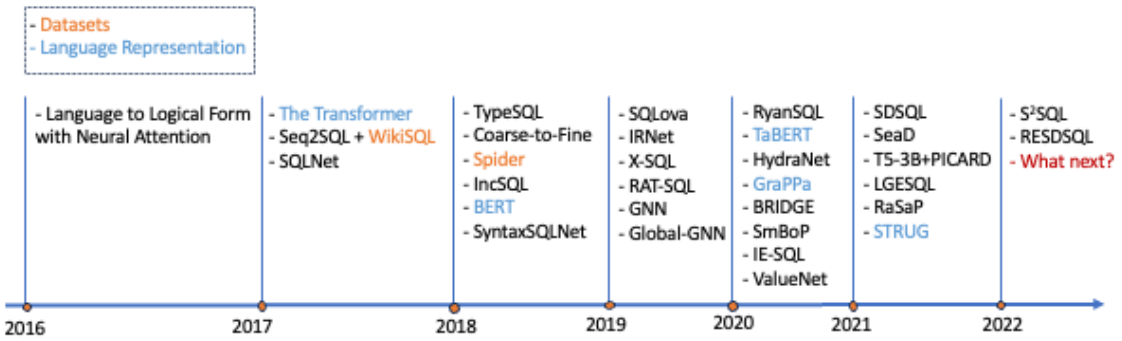


Fig. 4.8: A timeline of developments in NLIDB (Timeline from [166]).

4.1.3 Semantic Parsing

Semantic parsing is one of the key components of a text-to-data system. It has been around, in different forms, for decades. Earlier, pre-neural semantic parsing systems suffer from a number of issues. They were domain-dependent and not scalable or robust. Furthermore, the lack of large amounts of training data precluded the use of large scale supervised learning methods.

Neural semantic parsing is part of the deep learning revolution in NLP that started in 2013–14. Neural methods became predominant in semantic parsing around 2016 but for a while, these parsers were limited to simple queries in a single domain, e.g., geographical databases. One of the key papers that showed the promise of neural methods across domains was Dong and Lapata [94]. The authors use a multi-layer sequence to sequence, LSTM-based architecture (Figure 4.9). The input, in the form of word embeddings, is shown at the bottom of the figure.

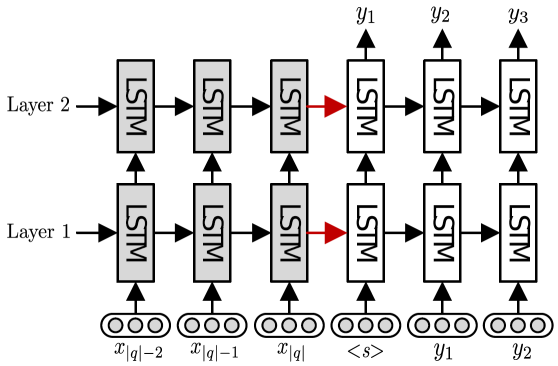


Fig. 4.9 Sequence to sequence for semantic parsing. (Image from [94]).

4.2 Meaning Representations

How to represent meaning in a format suitable for reasoning is a fundamental issue in semantic parsing. Meaning (semantic) representations (MR), unlike syntactic representations such as parse trees, need to be robust to various forms of paraphrasing. For example, the two sentences "*The cat found the fish*" and "*The fish was found by the cat*" should ideally have identical or, at least, very similar semantic representations. One such representation is First Order Logic (FOL): $\exists(x)\exists(y) : Cat(x) \wedge Fish(y) \wedge Found(x, y)$. Such semantic representation can be rendered as text sentences in multiple ways.

4.2.1 First Order Logic

First Order Logic (FOL), or **Predicate Logic** is a classic formalism used to represent meaning. It is particularly appropriate for relational databases since they are organized around the notion of predicates (e.g., relations and properties) and entities (e.g., objects).

The FOL notation is used to represent objects, relations, and properties. For example, a symbolic object can be used to represent the person Alice. A relation can represent the fact that Bob is Alice's brother: "`Brother(Bob, Alice)`". Another relation can represent the fact that both Alice and Bob speak French: "`Speaks(Alice, French)` $\wedge$ `Speaks(Bob, French)`". Furthermore, a property of an object can be used to represent Alice's age, e.g., "`AgeOf(Alice)`".

FOL also includes the use of two quantifiers. The existential quantifier, marked with the symbol $\exists$, describes the existence of an object, e.g., "$\exists$`x Cat(x)`" means that there exists an object x and that this object represents a cat. The universal quantifier, $\forall$ is used as a shorthand for a (potentially large) set of facts. For example, "$\forall$`x Player(x)` $\Rightarrow$ `Speaks(x, French)`" means that "all players speak French".

FOL is used for symbolic reasoning, e.g., for traditional forms such as syllogisms. Figure 4.10 illustrates this concept with an example, namely that if a statement α is true and if α implies another statement β, then it follows that if α is true, then β must also be true. The example on the right makes use of substitution, whereby in order to combine the two statements at the top, the unbound variable x has to be "bound to" (associated with) the constant Martin.

$$\frac{\alpha \qquad \alpha \Rightarrow \beta}{\beta} \qquad\qquad \frac{Cat(Martin) \qquad \forall x : Cat(x) \Rightarrow EatsFish(x)}{EatsFish(Martin)}$$

Fig. 4.10: Syllogism example.

4.2.2 Lambda Calculus

Lambda calculus was introduced by Alonzo Church [65] as a general model of computation. It is based on expressions, which can be names, functions, or applications. In lambda calculus, names are local to definitions. For example, in the expression λx:x , we consider x to be "bound" since it is introduced by λx. Other names, not preceded by λ, are called free variables. For example, in the expression λx:x+y, the variable y is considered to be "free".

Lambda calculus expressions can be typed. They can use a single type (e.g., truth value t or entity e), as well as complex types such as <e,t>. This latter type corresponds to an object that converts an input of type e to an output of type t, which can be written as < <e,t>, e>, and which describes an object that takes as input a (second) object of type <e,t> and returns a (third) object of type t. An example is shown in Figure 4.11.

Fig. 4.11 Entities (e) and truth values (t) in lambda calculus. The complex type of the verb "jumps" is <e,t> which means that "jumps" can be combined with a subject entity (e.g., Alice, as in "Alice jumps") to generate an object of type t (Alice jumps).

Alice	e
Alice jumps	t
jumps	<e,t>

Lambda calculus can be used to represent all of First-Order Logic. It allows for compositionality and higher order functions. The term "lambda function" is used to describe an unnamed function with an input parameter or parameters, as well as a body. Such a function can then be "applied" to the input parameter by executing the expression in the body on that input parameter. Figure 4.13 shows two examples of lambda functions. The first one adds 1 to its (single) argument. The second one computes the sum of its two arguments.

$$inc(x) = \lambda x.x + 1$$
$$inc(4) = (\lambda x.x + 1)(4) = 5$$

$$add(x, y) = \lambda x, \lambda y.(x + y)$$
$$add(3, 4) = (\lambda x, \lambda y.(x + y))(3)(4)$$
$$= \lambda y.(3 + 4) = 3 + 4 = 7$$

Fig. 4.13: Examples of lambda functions.

Some additional examples of lambda expressions are shown in Figure 4.14. The first example describes an intransitive verb (a verb without a direct object). The second one is for a transitive verb (one that takes a direct object) while the last one shows an example of a reflexive verb (in which the subject and object are the same entity, i.e., Mary likes herself).

Fig. 4.14 Sample lambda expressions for verbs.

$$[\lambda x.sleeps(x)](Mary) = sleeps(Mary)$$
$$[\lambda x.likes(Mary, x)](John) = likes(Mary, John)$$
$$[\lambda x.likes(Mary, x)](Mary) = likes(Mary, Mary)$$

Lambda calculus can be used as an intermediate representation for a variety of NLP tasks, including semantic parsing, inference, and question answering. One such example in question answering is shown in Figure 4.15. Then Figure 4.16 shows how FOL can represent a command in an interactive environment.

Fig. 4.15 Using lambda calculus to answer a factual question.

Predicate	Who is the president of China
Lambda expression	$\lambda(x) : Person(x) \wedge President(x, China)$
Answer	Xi

$$a.pre(a, x.chair(x)) \wedge move(a) \wedge len(a, 3) \wedge dir(a, forward) \wedge past(a, y.sofa(y))$$

Fig. 4.16: FOL representation for the command "at the chair, move forward three steps past the sofa." (Example from [11]).

Some well-cited references on constructing lambda expressions are [10] and [11], along with the matching tutorial [9].

4.2.3 Abstract Meaning Representation

Abstract Meaning Representation (AMR) [172] is a formalism that uses directed acyclic graphs with labels on the edges and on the leaf nodes in order to represent the meaning of a sentence as a set of connected entities, properties, and events, in a neo-Davidsonian [83, 140, 245] way. AMR graphs are designed to provide a representation that goes beyond syntax. AMR is based on earlier representations of feature structures, e.g., [290] and PENMAN input representations [213]. An example of an AMR tree is shown in Figure 4.17.

```
\begin{verbatim}
(s / say-01
  :ARG0 (g / organization
          :name (n / name
                   :op1 ``UN''))
  :ARG1 (f / flee-01
          :ARG0 (p / person
                   :quant (a / about
                             :op1 14000))
          :ARG1 (h / home :poss p)
          :time (w / weekend)
          :time (a2 / after
                   :op1 (w2 / warn-01
                           :ARG1 (t / tsunami)
                           :location (l / local))))
  :medium (s2 / site
            :poss g
            :mod (w3 / web)))
```

Fig. 4.17: Abstract Meaning Representation of the sentence "About 14,000 people fled their homes at the weekend after a local tsunami warning was issued, the UN said on its Web site" (Example from the AMR web site, https://amr.isi.edu).

In AMR, an entity is represented as an existentially quantified variable, in the form of a word node, e.g., "a/animal". Similarly, a notation like "(e / eat-01 :location (b / backyard))" represents the fact that an eating event (e) takes place in the backyard (b). Events are represented based on PropBank framesets (e.g., "eat-01"). AMR uses about 100 relations (e.g., "distance-quantity") to express other components of the meaning representation of a sentences.

AMR graphs can be re-entrant. For example, Figure 4.18 shows that the node "Lassie" indirectly descends from both the "e / eat-01" node and the "f /find-01" node.

AMR has evolved over the years and a number of extensions have been introduced. For example, it can be extended to multisentential text. O'Gorman et al. [237] introduce the Multi-Sentence AMR corpus (MS-AMR) in 2018. A more recent paper on this topic is by Naseem et al. 2021 [233]. Another newer proposal to extend AMR is Universal Meaning Representation (UMR) [320] which extends AMR to other languages, particularly morphologically complex, low-resource languages. UMR also extends AMR with features that are critical to semantic interpretation and document-level representation that captures linguistic phenomena that potentially go beyond sentence boundaries. More recently, another paper has appeared [212] that introduces the BabelNet Meaning Representation (BMR), as an alternative to AMR representation. Other useful references include two PhD dissertations: [80] and [106], which cover more on AMR, including both

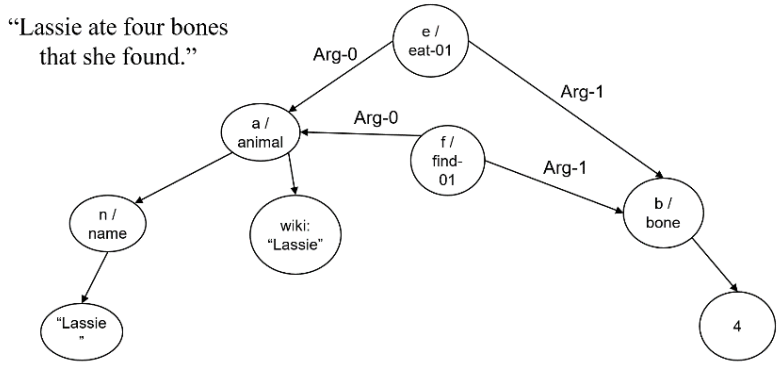


Fig. 4.18 AMR representation of the sentence "*Lassie ate four bones that she found*." In this example, there are two (parallel) paths from "*Lassie*" to "*Bones*"

parsing and generation. In addition, a recent tutorial on the topic of MR provides details on their design, use cases, modeling, and real-world use cases [107].

4.2.4 Word Embeddings

How to represent linguistic information has been a key question in NLP for decades. Most early methods used symbolic representations. More recently, a switch to distributed representations has occurred. One way to represent a word in a neural network is by using one-hot vectors (Figure 4.19) of size v, where v is the size of the vocabulary. In one-hot representations, all values of the vector are equal to zero except for a single non-zero value that appears in the i-th position of the vector and which corresponds to the i-th word in the vocabulary.

Fig. 4.19 One-hot vector representations for the words "*cat*" and "*kitten*". Note that the dot product of the two vectors is zero.

Cat = [0 0 0 0 0 0 **1** 0 0 0 0 0 0 ... 0]
Kitten = [0 0 0 0 0 0 0 0 0 0 **1** 0 0 ... 0]

A fundamental problem with one-hot representations is that they are unable to represent word similarity. For example, the vector distance between the pair of words ("*cat*","*dog*") is the same as the distance in ("*cat*","*car*"), even though the words in the first pair are in reality semantically more related to each other than the words in the second pair are.

Modern meaning representations are instead based on **word embeddings**. These are dense, low-dimensional (e.g., 300-dimensional) vectors, automatically learned from existing natural language text. Word embeddings are also known as distributed representations (Figure 4.20) and, by design, capture word similarity well. In the example, the distance between the vectors for "*cat*" and "*dog*" will be higher than the distance between the vectors for "*cat*" and "*car*".

A sample 300-dimensional GloVe [248] word embedding for the word "*pharmaceuticals*" is shown in Figure 4.21. Figure 4.22 shows how compositionality can be achieved by summing embeddings. The black arrow corresponds to the relation product-manufacturer, whereas the yellow arrow represents the type of product.

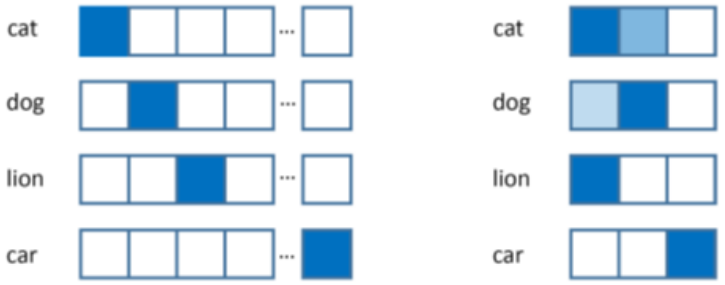


Fig. 4.20 One-hot (left) vs. distributed representation (right) of the words cat, dog, lion, and car.

```
[-0.43337 0.10411 -0.49926 -0.06442 0.39005 0.65823 -1.3033 -0.39019 -0.41713 -0.038667 -0.37677 0.35303
-0.16425 -0.46671 -0.24358 0.10142 0.54664 0.065344 0.022733 -0.10422 -0.52839 0.20096 -0.23819 0.28856
0.098377 -0.12174 -0.035851 -0.92717 0.28408 0.22439 0.027598 0.47919 -0.51142 -0.062696 0.13002 0.20366
-0.050135 -0.4593 0.63779 -0.56294 -0.40989 -0.60342 0.8151 0.024929 -0.0096054 0.55748 0.35156 0.14476
-0.36327 -0.34856 0.1787 0.76398 0.27803 0.11999 0.42814 0.17844 0.092123 0.058811 -0.41114 -0.12101
-0.34001 -0.49946 -0.073549 0.15371 0.18034 0.34176 0.072738 -0.23442 -0.023682 -0.2499 -0.17334 -0.15008
-0.14599 0.51706 0.52797 -0.075328 0.13018 -0.069498 -0.15378 -0.31615 0.59434 -0.91396 -0.12803 0.32963
0.70337 -0.095882 -0.37066 0.16993 -0.62115 -0.76234 0.49005 -0.026823 -0.35171 -0.070227 0.19778 0.25563
-1.4504 -0.47122 -0.10107 -0.18279 -0.31553 0.090524 0.1975 0.073745 0.34809 -0.26728 -0.04808 -0.18467
0.18147 0.37255 0.26197 -0.0046708 0.51 -0.99408 -0.1942 -0.82518 0.59211 0.31112 0.3472 -0.066567 0.65975
-0.52254 -0.48302 0.30366 -0.35524 0.0022488 -0.89521 -0.096487 -0.36811 -0.1139 -0.039127 0.03701 0.18691
-0.28874 0.19926 -0.71229 0.21108 0.1768 0.27541 -0.72828 -0.74097 0.15007 -0.46696 0.52759 0.69806 0.28434
1.2781 0.033105 0.2153 -0.59069 -0.18089 -0.28775 -0.30792 -0.32764 -0.20838 -0.49774 0.2604 -0.26116
-0.29203 -0.18311 0.016024 0.26013 -0.4441 0.11857 0.61598 -0.25685 -0.49715 0.58277 -0.093157 -0.078187
-0.18587 -0.021307 0.52742 0.75704 0.091185 -0.41006 -0.26896 0.43715 0.13183 -0.49085  -0.84639 -0.22379
-0.094786 0.35858 -1.16 -0.019064 -0.29052 0.21588 0.026218 0.22063 -0.64061 0.89117 -0.14541 -0.47563
0.77044 -0.45668 0.49585 0.45303 -0.24904 0.24502 0.42608 0.0077214 -0.55742 0.17449 -0.2142 0.26996
-0.26239 0.18933 -0.66798 -0.004951 0.062785 0.45616 -0.77372 0.29266 -0.76515 0.2079 -0.52916 -0.13621
-0.60588 -0.049171 -0.21234 -0.071004 0.092045 0.87973 -1.0929 -0.028515 -0.28424 0.26105 0.2524 0.35996
-0.74328 -0.27066 -0.046789 0.081494 0.70996 0.18443 -0.10652 -0.32352 -0.026315 0.079707 0.14203 -0.21906
0.49254 -0.29718 0.25746 -1.108 -0.35566 -0.14188-0.35727 -0.34243 -0.37111 1.0034 -0.13679 0.92309
-0.16716 -0.36536 0.019904 -0.33768 0.18646 0.4391 0.045023 -0.25726 -0.14073 0.77022 -0.18926 -0.27791
0.2978 -0.78997 0.052217 0.72518 0.0089245 0.1029 1.0213 -0.70057 0.31295 0.29313 -0.53556 0.75132 0.29351
-0.70482 -0.61882 -0.33732 0.60293 -0.33575 -0.12536 0.27659 -0.20361 0.12683 0.10469 -0.47956 0.187
0.38118 0.16238 -0.0484 0.43112 0.0089624 0.0051162 -0.67922 0.1709 -0.020472]
```

Fig. 4.21: Sample word embedding for the word "*pharmaceuticals*" from the glove300d dataset citepennington-etal-2014-glove.

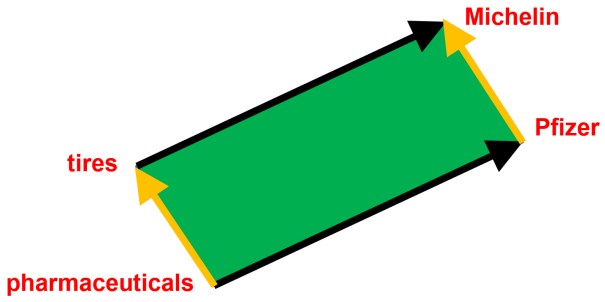


Fig. 4.22 Compositional embeddings.

Word embeddings are trained using unsupervised learning, based on word cooccurrence. They represent the probability of predicting a word given its context. For example, given the words "*bark*", "*bone*", "*puppy*", and "*veterinarian*", the word "*dog*" is very likely to appear nearby, compared to a semantically different word such as "*watermelon*". Word embeddings are appropriate for computing semantic similarity, as well as identifying nearest neighbors. Embeddings of individual words can further be composed into meaning representations for entire sentences. Even though they existed long earlier [202, 224, 317], word embeddings became very popular with the introduction of Word2Vec [218, 219] and GloVE [248]. These embeddings and many others can be easily downloaded from the Web and are routinely used in a wide range of NLP projects.

One main paradigm in modern NLP is the use of **pretrained language models** (PLM) which are trained on large quantities of textual data. One of the most salient such models is BERT [89] and it is used to compute contextual embeddings. This way, the neural network can distinguish between the use of the word "*mouse*" as an animal and its use as a computer accessory, since the two versions of "*mouse*" will be encoded differently. It is a large, pretrained neural network, used for hundreds of NLP tasks. It uses a number of transformer layers and takes its input as WordPiece [348] embeddings. It is trained on the Books and English Wikipedia text corpora using two objectives, masked language modeling (predicting a masked word in a sentence) and next sentence prediction. The base model of BERT includes 110M parameters, whereas the large one has 340M parameters. BERT can be finetuned with task-specific layers, e.g., to perform different NLP tasks. When it was first introduced, BERT achieved SOTA on a number of NLP tasks such as QNLI (Question-answering NLI) [325], semantic similarity, and linguistic acceptability. BERT has been also used as a foundation for more advanced and often significantly larger, pretrained language models, including RoBERTa [208] and table specific models such as TaPAS [139], TaBERT [363], and GraPPa [366], which we will discuss in Chapter 6 on text generation.

4.2.5 Semantic Compositionality

In order to represent the meaning of a full sentence, individual word embeddings have to be combined. The principle of compositionality in logic says that the meaning of a complex sentence can be computed recursively from the meaning and structure of the underlying components. One example of such sentence embeddings was introduced in [152]. This paper showed that Deep Averaging Networks (DAN) can achieve excellent performance on text classification tasks. DANs compute the average of the input word embeddings, then send these averages through the feedforward layers.

Natural language shows an intriguing mix of compositional and non-compositional properties. For example the meaning of the phrase "*red house*" is compositional; it can be obtained by combining the meanings of the two underlying words, since a red house is just a house that is painted red. However, an idiomatic expression such as "*red herring*" is non-compositional, because it is *not* about a red-colored herring. In another word, "*red house* = *red* + *house*" but "*red herring* ≠ *red* + *herring*".

Figure 4.23 shows how compositionality can be used to represent the meaning of phrases that appear in questions in NLIDB. For example, the meaning of the phrase "`count (Person)`" is composed of the meanings of the predicate "`count`" and the object "`Person`." Compositionality plays a major role in meaning representations and semantic parsing.

Phrase	Compositional Representation
How many people	`count (Person)`
Highest paid person	`argmax (Person, Salary)`
Total salaries	`sum (Salary)`
Salary gap	`max (Salary) { min (Salary)`

Fig. 4.23 Using compositionality to represent question phrases.

4.2.6 Knowledge Graphs and RDF

Knowledge graphs are another useful representation for entities and relationships. They can be used to represent knowledge about the world from Wikipedia, e.g., which songs were performed by a given artist, which rivers run in a given state, or which people ruled a specific country over the years. Knowledge graphs consist of vertices (entities) and edges (relations). Figure 4.24 shows a sample knowledge graph about the Houston Rockets basketball team and a number of objects related to it.

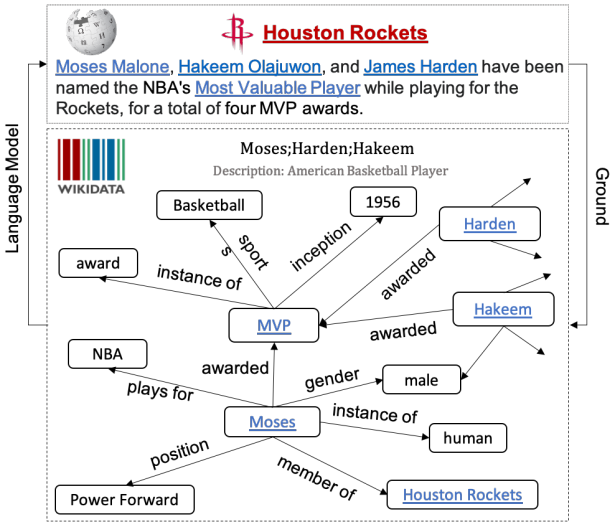


Fig. 4.24 A sample knowledge graph about the Houston Rockets basketball team. (Image from [55]).

RDF (Resource Description Framework) [75] is the language used to encode knowledge bases such as DBpedia and other datasets such as FOAF[1] and LinkedGeoData[2]. This representation is endorsed by the W3 consortium (`http://w3.org`). An RDF triple consists of a subject, a relationship (predicate), and an object. It is, however, not a natural language representation. An example from Yago[3] is shown in Figure 4.25.

```
@base <http://yago-knowledge.org/resource/> .
@prefix dbp: <http://dbpedia.org/ontology/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<yagoTheme_yagoLiteralFacts>
<Papua_New_Guinea><hasTLD>".pg" .
<Luxembourg><hasTLD>".lu" .
<Costa_Rica><hasTLD>".cr" .
<Japan><hasTLD>".jp" .
<Uruguay><hasTLD>".uy" .
<Tanzania><hasTLD>".tz" .
<Comoros><hasTLD>".km" .
<South_Korea><hasTLD>".kr" .
<Saint_Lucia><hasTLD>".lc" .
<Somalia><hasTLD>".so" .
<Philippines><hasTLD>".ph" .
<Saudi_Arabia><hasTLD>".sa" .
<Iceland><hasTLD>".is" .
<Switzerland><hasTLD>".ch" .
<Suriname><hasTLD>".sr" .
<Mali><hasTLD>".ml" .
<Guatemala><hasTLD>".gt" .
```

Fig. 4.25: A knowledge base in RDF.

```
#include <stdio.h>

int main()
{
  int n, reverse = 0;

  printf("Which number should I reverse\n");
  scanf("%d",&n);

  while (n != 0)
  {
    reverse = reverse * 10;
    reverse = reverse + n%10;
    n = n/10;
  }
  printf("Reversed number = %d\n", reverse);

  return 0;
}
```

Fig. 4.26: Unambiguous programming code in C.

4.2.7 Syntactic representations

Parsing computer languages (such as C) (Figure 4.26) is done by compilers which make use of the unambiguous structure of the computer code. Unlike computer languages, human language is (highly) ambiguous. Human language has no explicit "types" for words, no brackets around phrases, and in most cases, no explicit nesting markers. Additional problems arise from ambiguous words and phrases, the use of implied and commonsense information, etc. Syntactic parsing (both constituency parsing and dependency parsing) is used to address syntactic ambiguities in natural language text.

Constituency parsing builds syntactic trees that usually start with an S ("sentence") node as the root. Then, recursively, they encode the syntactic structure of the sentence, e.g., the S node can have a noun phrase (NP) and a verb phrase (VP) as its children. At the other end of the tree, the leaf (terminal) nodes correspond to the individual words in the sentence, while their parent nodes (the "pre-terminals") represent the part of speech tags of the leaf nodes (e.g., N for Noun or V for Verb). The pre-terminals and the other nodes above them are collectively called non-terminals. Constituency parsing was traditionally done using methods like the Collins parser [71] or the Charniak parser [44].

Dependency parsing builds a rooted, directed, spanning tree, without using non-terminals. Instead, each node, except the root node, corresponds to a word in the sentence. Each node can have zero or mode children. Each edge corresponds to a syntactic dependency, with the "head" of the dependency appearing as the parent node and each of its "modifiers" appearing as a child node. For example, the phrase "*red house*" is represented as a dependency edge with "red" as the modifier (dependent) node and "*house*" as the head node. The root node is typically the main verb of the sentence. Some of the classic methods for dependency parsing are MacDonald et al. [214] and Nivre et al., e.g., [235]. More recently, neural methods such as Chen and Manning [49] and Dozat and Manning [96], have overtaken this task.

A number of grammar formalisms have been proposed over the years, such as Inductive Logic Programming [379], SCFG [344], CCG + CKY [380], Constrained Optimization and Integer Linear Programming (ILP) [66], and dependency-based compositional semantics (DCS) combined with projective dependency parsing [201].

We should note here that these two traditional paradigms for syntactic parsing (constituency parsing and dependency parsing) are not directly used in modern NLP pipelines. Instead, they have now mostly been taken over by neural methods.

4.2.8 Combinatory Categorial Grammar

Combinatory Categorical Grammar (CCG) [298, 381]CCG is a mildly context-sensitive grammar formalism that can be used to represent accurately several interesting linguistic phenomena, especially from a lexical and syntactic perspective. The lexicon used in CCG is relatively small and easy to compose into complicated, nested constructions. Zettlemoyer and Collins (2009) give this example (Figure 4.27).

The lexicon describes a mix of syntactic and semantic rules. In the example above, the lexicon entry for "*flights*" indicates that the word is a Noun (N) but that it is also used to represent an object x of the semantic type "flight". The second lexicon entry for "*to*" indicates that the word "*to*" can be combined with a noun phrase (NP) to its right (e.g., with the noun phrase "*Boston*") and that the resulting span will be of type

[1] https://data.mendeley.com/datasets/zp23s23xpb/1

[2] http://linkedgeodata.org/

[3] https://resources.mpi-inf.mpg.de/yago-naga/yago3.1/yagoLiteralFacts.txt

$$\begin{array}{lcl} \text{flights} & := & N : \lambda x.flight(x) \\ \text{to} & := & (N\backslash N)/NP : \lambda y.\lambda f.\lambda x.f(x) \wedge to(x,y) \\ \text{boston} & := & NP : boston \end{array}$$

Fig. 4.27 CCG Example (flights to Boston).

(N\N), which can further be combined with a noun (N) and have the following semantic meaning: there is an object (e.g., "*flights*") as well as two locations (x and y) such that there is the object "*flights*", which connects these two locations (as in "*flights to Boston from Nashville*").

Figure 4.28 shows examples of both simple and complex syntactic types as employed in CCG. As can be seen, even basic part of speech categories such as adjectives, articles, and preposition can be defined compositionally from other types. For instance, an article can be defined as an object that, when combined with the noun on its right, returns an object of type NP (noun phrase).

Fig. 4.28 Sample lexical entries in Combinatory Categorial Grammar. Most syntactic categories are defined compositionally from other, simpler, syntactic categories.

Syntactic category	**CCG representation**
Nouns	N
Adjectives	N/N
Articles	NP/N
Prepositions	(NP\NP)/NP
Transitive verbs	S\NP
Intransitive verbs	(S\NP)/NP

CCG is further used for syntactic and semantic parsing, through the application of rules, e.g., combinator rules. Consider the example in Figure 4.29. At the syntactic level, it shows how the words "*to*" and "*Boston*" (with the following types, respectively: (N\N)/NP and NP) can be combined into the phrase "*to Boston*" (typed "N\N"), which can then in turn be combined with the word "*flights*" (of type N) to form the full phrase "flights to Boston" (typed "N" in this particular example, though it can be better represented as "NP" or noun phrase). At the semantic level, the end result is the lambda expression $\lambda x.flight(x) \wedge to(x, boston)$. This lambda expression can then later be combined with another N (e.g., "*Nashville*") to form the phrase "*flights to Boston from Nashville*".

$$\dfrac{\text{flights} \quad \dfrac{\text{to} \quad \text{boston}}{\begin{array}{c}(N\backslash N)/NP \\ \lambda y.\lambda f.\lambda x.f(x) \wedge to(x,y)\end{array} \quad \begin{array}{c}NP \\ boston\end{array}}>}{\begin{array}{c}N \\ \lambda x.flight(x)\end{array} \quad \begin{array}{c}(N\backslash N) \\ \lambda f.\lambda x.f(x) \wedge to(x, boston)\end{array}}<$$
$$\begin{array}{c}N \\ \lambda x.flight(x) \wedge to(x, boston)\end{array}$$

Fig. 4.29 CCG parse derivation. (Example from 381).

CCG can also be used to map syntactic types to semantic types (e.g., categories to lambda expressions). It can also perform other operations such as composition, type shifting, and coordination.

4.3 Converting sentences to structured form

Researchers have looked at automatic translation of natural language to database queries for decades. One of the first works on this topic is the Lunar Sciences Natural Language Information System [345] which provided an interface to a small database about lunar rocks.

4.3.1 Information Extraction

Information extraction is an important component of the text-to-data pipeline. It is the process of automatically converting unstructured text to structured tables that can be later processed through a database interface. In the example in Figures 4.30 and 4.31, plain text sentences, namely headlines about ratings changes of equities, get converted to a structured (database) format. Not all fields may appear in the input, or have values extracted by the system. Each record in the output matches one of the input sentences. The fields include the company name, its "`ticker`" (trading) symbol, the source of the rating change, the old and the new rating (if known), as well as the direction of the change (up, down, unchanged). The type of each field (or attribute) is shown in a different color.

RESEARCH ALERT-Wells Fargo cuts PPD Inc to market perform
China Southern Air Upgraded To Overweight From Neutral-HSBC
CITIGROUP RAISES INGERSOLL RAND <IR.N> TO HOLD FROM SELL
TCF Financial Corp Raised To Overweight From Neutral By JPMorgan
BAIRD CUTS KIOR INC <KIOR.O> TO UNDERPERFORM RATING
BRIEF-RESEARCH ALERT-Global Equities Research cuts LinkedIn to equal weight

Fig. 4.30: Transforming unstructured text to structured tables (input).

DATE/TIME	TICKER	COMPANY	SOURCE	OLD	NEW	CHANGE
		PPD Inc	Wells Fargo		market perform	↓
		China Southern Air	HSBC	Neutral	Overweight	↑
	IR.N	INGERSOLL RAND	CITIGROUP	SELL	HOLD	↑
		TCF Financial Corp	JPMorgan	Neutral	Overweight	↑
	KIOR.O	KIOR INC	BAIRD		UNDERPERFORM	↓
		LinkedIn	Global Equities Research		equal weight	↓

Fig. 4.31: Transforming unstructured text to structured tables (output).

4.3.2 GeoQuery

One of the most cited classic projects in an area of NLP adjacent to NLIDB is **GeoQuery** [379]. This line of work has spawned an outsized number of follow up contributions. The underlying database, GeoBase (sometimes referred to as GEO), includes 800 facts, represented in Prolog, about US geography. Examples from this database are shown in Figure 4.32. Then Figure 4.33 shows the main objects (e.g., country, state, river, etc.) used in GeoQuery. Figures 4.34 and 4.35 illustrate the predicates and meta-predicates, respectively, that are used in the paper.

Fig. 4.32 Sample queries in GeoQuery. (Example from [379]).

What is the capital of the state with the largest population? answer(C, (capital(S,C), largest(P, (state(S), population(S,P))))).
What are the major cities in Kansas? answer(C, (major(C), city(C), loc(C,S), equal(S,stateid(kansas)))).

Fig. 4.33 Basic objects in GeoQuery. (Example from [379]).

Type	Form	Example
country	countryid(Name)	countryid(usa)
city	cityid(Name, State)	cityid(austin,tx)
state	stateid(Name)	stateid(texas)
river	riverid(Name)	riverid(colorado)
place	placeid(Name)	placeid(pacific)

Fig. 4.34 Sample predicates in GeoQuery. (Example modified from [379]).

Form	Predicate
capital(C)	C is a capital (city).
city(C)	C is a city.
major(X)	X is major.
place(P)	P is a place.
river(R)	R is a river.
state(S)	S is a state.
capital(C)	C is a capital (city).
area(S,A)	The area of S is A.
capital(S,C)	The capital of S is C.
equal(V,C)	variable V is ground term C.
elevation(P,E)	The elevation of P is E.
high point(S,P)	The highest point of S is P.
higher(P1,P2)	P1's elevation is greater than P2's.
loc(X,Y)	X is located in Y.
low point(S,P)	The lowest point of S is P.
len(R,L)	The length of R is L.
next to(S1,S2)	S1 is next to S2.
size(X,Y)	The size of X is Y.

GEO is based on Chill [378] which uses pairs of sentences and executable database queries as input. The language used to query is based on logic grammars (e.g., [77, 336]). The query interpreter executes

Form	Explanation
answer(V,Goal)	V is the variable of interest in Goal.
largest(V, Goal)	Goal produces only the solution that maximizes the size of V.
smallest(V,Goal)	Analogous to largest.
highest(V,Goal)	Like largest (with elevation).
lowest(V,Goal)	Analogous to highest.
longest(V,Goal)	Like largest (with length).
shortest(V,Goal)	Analogous to longest.
count(D,Goal,C)	C is count of unique bindings for D that satisfy Goal.
most(X,D,Goal)	Goal produces only the X that maximizes the count of D.
fewest(X,D,Goal)	Analogous to most.

Fig. 4.35 Meta predicates in GeoQuery. (Example modified from [379]).

queries on the database. Later, Tang and Mooney [306] use a shift reduce type parser with operators such as INTRODUCE, COREF_VARS, DROP_CONJ, LIFT_CONJ, and SHIFT. INTRODUCE inserts a predicate at the top of the stack in anticipation of an expression that starts with the current word in the input. This parser is then followed by an Inductive Logic Programming (ILP) method that uses both bottom-up and top-down search through the space of rules.

4.3.3 Semantic Parsing

Semantic parsing is used to convert natural language to a symbolic and/or logical form, i.e., to executable code for a specific application. Some obvious examples include airline reservations, geographical information systems, access to tables with sports results, etc.

Figure 4.36 shows how semantic attachments can be used to build a hierarchical meaning representation. In this example, the input is the natural language sentence "*Javier likes pizza*." This sentence is represented as a semantic tree with "`S: likes (Javier, pizza)`" as the root node. In this representation, "S" stands for "Sentence" (the syntactic category of the input sentence) whereas "`likes (Javier, pizza)`" gives the meaning of the input sentence. Recursively, the internal nodes have a similar syntactic+semantic structure. E.g., the "VP" (verb phrase) node has a lambda expression as its semantic representation.

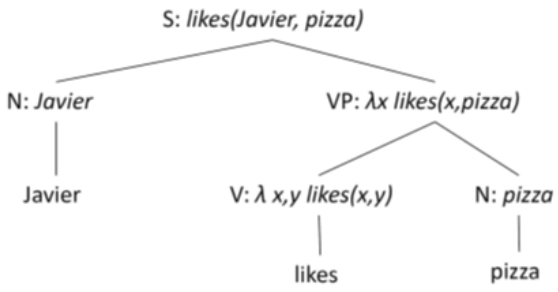


Fig. 4.36 Hybrid parsing with semantic attachments.

Figure 4.37 shows a context-free grammar that can be used to generate the parse tree in Figure 4.36.

```
S  -> NP VP   {VP.Sem(NP.Sem)}    t
VP -> V NP    {V.Sem(NP.Sem)}     <e,t>
NP -> N       {N.Sem}             e
V  -> likes   {λ x,y likes(x,y)   <e,<e,t>>
N  -> Javier  {Javier}            e
N  -> pizza   {pizza}             e
```

Fig. 4.37 Grammar with semantic attachments.

4.3.4 Semi-supervised semantic parsing

A number of other relevant papers from that time need to be mentioned. In [20], the authors describe a semantic parser on Freebase without using annotated logical forms. They use a simplified version of Liang's Lambda Dependency-Based Compositional Semantics (λ-DCS) framework. They map questions to answers using latent logical forms. First, they employ a coarse alignment using Freebase and a text corpus, then perform a bridging operation that makes adjacent predicates compatible (see Figure 4.38). For their work, they use the FREE917 dataset by Cai and Yates [37] that consists of 917 questions associated with 635 Freebase relations, and despite not using annotated logical representations, they outperform the Cai and Yates parser. They also introduce the WebQuestions dataset, which includes wh-questions about a single entity, as retrieved using the Google Suggest API.

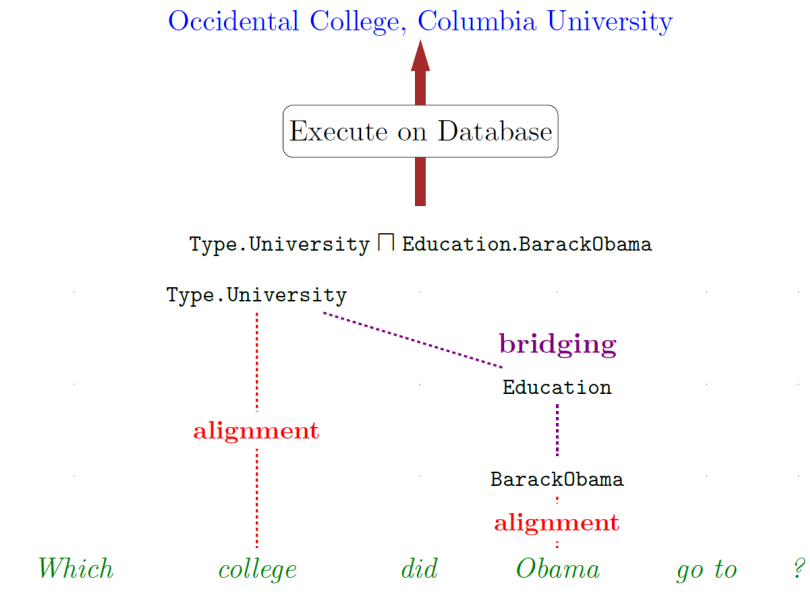


Fig. 4.38 Semantic parsing on Freebase. (Image from [20]).

An interesting research question is what happens if there are no logical form annotations (such as lambda expressions of CCG formulas) for training. Some of the main contributions in this area that happened in the mid-2010s is related to the WikiTableQuestions [4] (WTQ) dataset [246] and the work that follows up on it. The WTQ dataset includes 22,033 question/answer pairs from 2,108 open domain wikipedia tables. The Q/A pairs were obtained through crowdsourcing. The crowd workers were instructed to write questions that

[4] https://ppasupat.github.io/WikiTableQuestions/

require compositionality (e.g., include counts, differences, and comparisons). Figure 4.39 shows an example from this dataset.

Year	City	Country	Nations
1896	Athens	Greece	14
1900	Paris	France	24
1904	St. Louis	USA	12
...	...	...	...
2004	Athens	Greece	201
2008	Beijing	China	204
2012	London	UK	204

x = Greece held its last Summer Olympics in which year?

y = 2004

Fig. 4.39 WikiTableQuestions example (Image from [246]).

The paper [246] focuses on semantic parsing through complex question answering from tables. It introduces a parsing algorithm based on logical forms and strong typing constraints that is designed to deal with the combinatorial explosion in the number of possible parses, which in turn is caused by the need to generalize to unseen domains, schemas, and tables and the need for deeper compositionality.

The training triples, $(x_i; y_i; t_i)$, consist of a question, an answer, and a table, respectively. No logical forms are needed. The tables from the training and test datasets are disjoint. Figure 4.40 illustrates the prediction framework. In Step (1) the input table t is converted to a representation w. In Step (2), the parser reads the input question x, e.g., "*Greece held the last Summer Olympics in which year?*" as well as w and generates multiple parse candidates of the input question in the form of a latent variable Z_x. The following step (3) performs ranking and generates z. Then, in step (4) the expressions generated in z are executed on the table representations and the result is stored in the variable y. Finally, the (x, Z, z, t, y) 5-tuples are ranked.

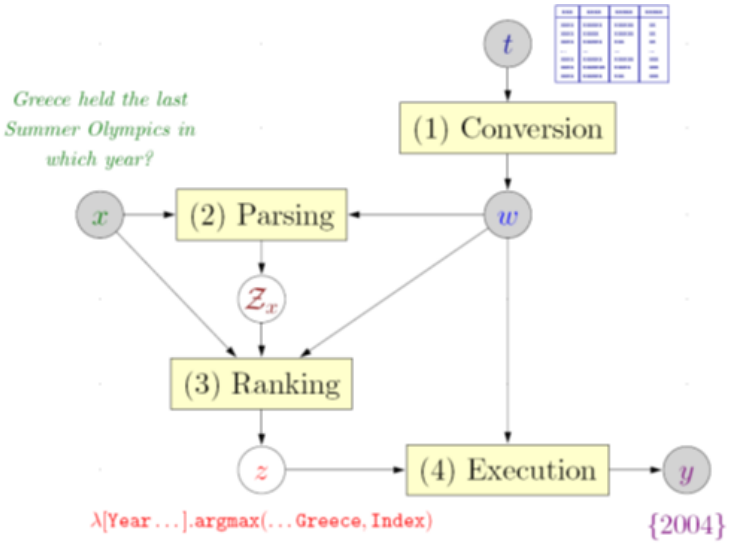


Fig. 4.40 WikiTableQuestions prediction framework (image from [246]).

A follow up paper [332] uses a simple grammar to convert utterances to simple ("clumsy") logical forms. Then crowd workers paraphrase the simple logical forms and the resulting dataset is then used to train a semantic parser over just a few hours (see Figure 4.41 for an illustration).

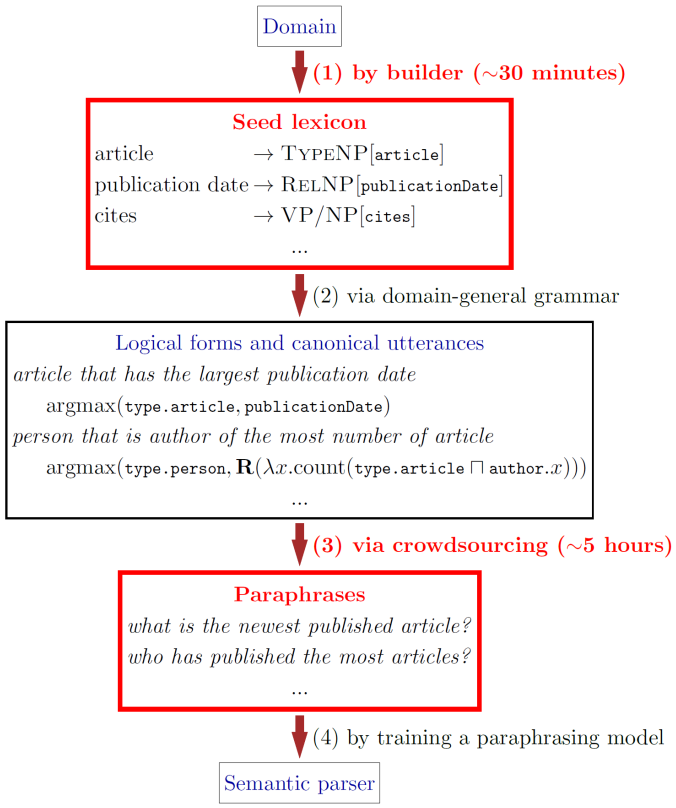


Fig. 4.41 Building a semantic parser overnight. (Image from [332]).

4.4 Neural Semantic Parsing

The bulk of the most recent work in natural language understanding has been done using neural networks for semantic parsing. This trend started around 2014 and has not abated since then.

4.4.1 Sequence to sequence methods

Text-to-text generation–translation of textual sequences to other textual sequences–is one of the fundamental tasks in NLP. This paradigm covers speech to text generation, machine translation, text summarization, and many other tasks including text to SQL translation. One of the main approaches to do text-to-text generation is through sequence to sequence models, such as those in the original seq2seq paper [304].

In such a model, a recurrent neural network (appropriately called the encoder) is used to extract some meaning from the input sequence. On reaching the end of the input (e.g., getting to an <EOS> symbol or equivalent), the network proceeds to "decode" the meaning extracted so far into a matching output sequence (e.g., in a different language or a different representation). This step is often done in an autoregressive manner, when each subsequent output symbol is generated based on both the input and the previously generated output symbols. The example in Figure 4.42 shows how a simple natural language query can be converted to a SQL statement. More recent methods additionally use attention and transformers.

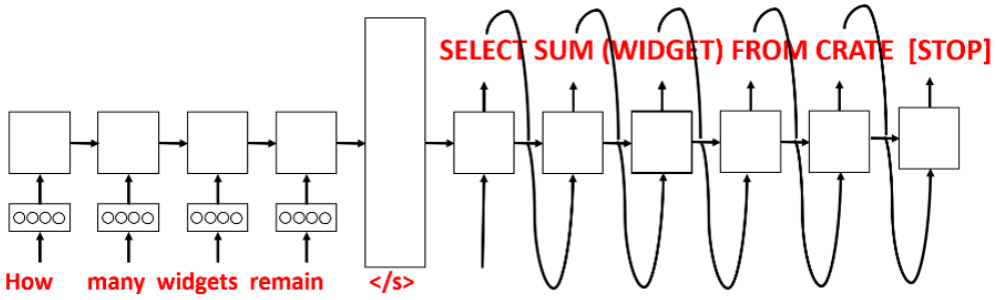


Fig. 4.42: Text to SQL with an encoder and a decoder.

One of the first modern papers related to text-to-SQL was Dong and Lapata [94] and it was followed by [388]. Some of the subsequent work has been sketch-based. One such paper is SQLNet[5] [352], in which each component has its own encoder. Column attention is used to emphasize the words in the query that match the table headers. Unlike WikiSQL, SQLNet does not use reinforcement learning.

Another relevant paper is SQLova[6] [147]. It uses the same sketch idea as SQLNet and processes all columns of the table at once. A related paper is HydraNet[7] [209]. It processes each column separately, then it passes on to BERT the following information for each column: ([CLS], NLQ, [SEP], $column\_type$, $table\_name$, $column\_name$, [SEP]). It then predicts the sketch slots using the output of BERT. The classification tasks include predicting whether a specific column appears in the SELECT or WHERE clause, predicting the aggregation function for a specific column, predicting the condition for the column, etc.

Some other relevant papers from that period are IncSQL [287], IRNet [128], and [362]. A recent survey [165] notes that Seq2SQL and SQLNet use word embeddings as input, whereas HydraNet, SQLova, IRNet, and RAT-SQL use transfer learning.

4.4.2 Applications of Neural Semantic Parsing

A number of papers on applications of neural semantic parsing have been published over the years. We will discuss briefly some of them here.

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

[6] https://github.com/naver/sqlova

[7] https://github.com/lyuqin/HydraNet-WikiSQL

Neural methods allow for semantic parsers to scale up to arbitrary domains. Instead of using lexicons, manual rules, and manual templates, Dong and Lapata [94] encode the logical forms (Figure 4.9) using an encoder-decoder model with attention. The underlying LSTM-based architecture allows for the generation of either trees (Figure 4.43) or sequences.

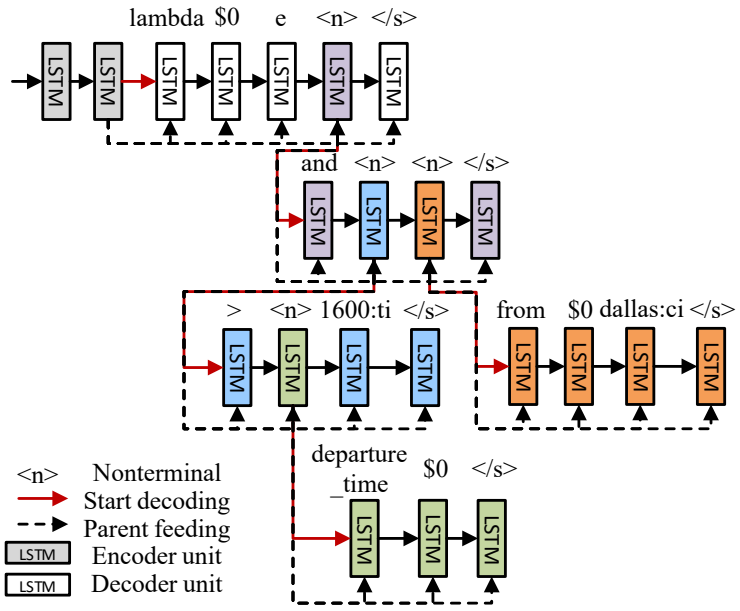


Fig. 4.43 Generating a logical form as a tree. (Image from [94]).

This work was evaluated on four datasets (Figure 4.44), Jobs, Geo, ATIS, and Ifttt. The first three datasets are reviewed in Chapter 5 and the Ifttt dataset includes if-this-then-that recipes from the IFTTT website [8]. The alignments produced by the attention mechanism are shown in Figure 4.45 for each of the four datasets. The colored boxes show how spans of the input sequence get aligned to spans of the matching output sequence.

Dataset	Length	Example
JOBS	9.80 22.90	*what microsoft jobs do not require a bscs?* answer(company(J,'microsoft'),job(J),not((req_deg(J,'bscs'))))
GEO	7.60 19.10	*what is the population of the state with the largest area?* (population:i (argmax $0 (state:t $0) (area:i $0)))
ATIS	11.10 28.10	*dallas to san francisco leaving after 4 in the afternoon please* (lambda $0 e (and (>(departure_time $0) 1600:ti) (from $0 dallas:ci) (to $0 san_francisco:ci)))
IFTTT	6.95 21.80	*Turn on heater when temperature drops below 58 degree* TRIGGER: Weather - Current_temperature_drops_below - ((Temperature (58)) (Degrees_in (f))) ACTION: WeMo_Insight_Switch - Turn_on - ((Which_switch? ("")))

Fig. 4.44: Evaluation on four datasets, Jobs, Geo, ATIS, and Ifttt (Image from [94]).

[8] http://www.ifttt.com

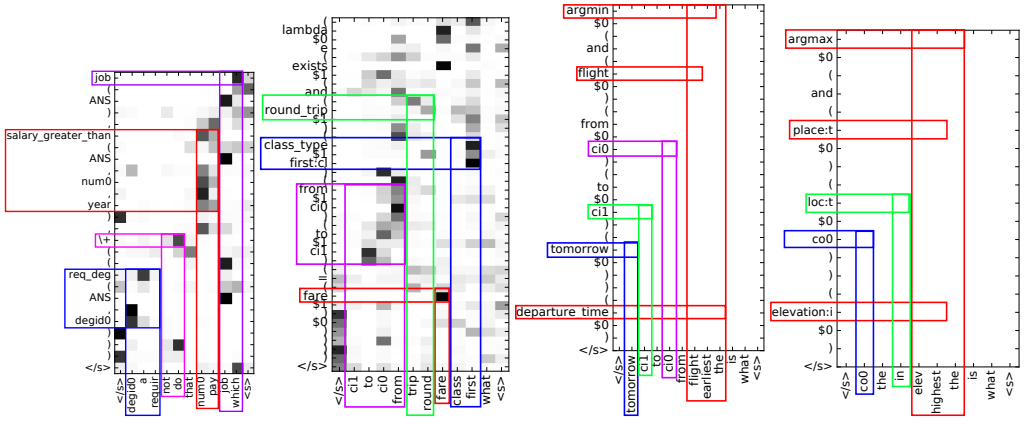
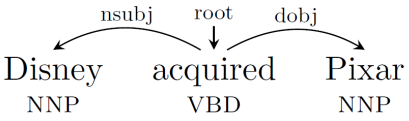


Fig. 4.45: Alignments produced by the attention mechanism (Image from [94]).

A follow up paper by Reddy et al. [267] looks into converting dependency structures to logical forms. Focusing on dependency structure instead of constituent structure allows for a simpler architecture. The results show that this method outperforms the earlier CCG-based representations on the Free917 dataset and achieves competitive results on the WebQuestions dataset.

The dependency tree is converted to a lambda expression (Figure 4.46). Figure 4.47 shows some lambda expressions for single words as well as a sentence parse. The operations used by this approach include binarization, e.g., "`(nsubj (dobj acquired Pixar) Disney)`", substitution, and composition.



(a) The dependency tree for *Disney acquired Pixar.*

$$(\text{nsubj } (\text{dobj acquired Pixar}) \text{ Disney})$$

(b) The s-expression for the dependency tree.

$$\lambda x.\, \exists yz.\, \text{acquired}(x_e) \wedge \text{Disney}(y_a) \wedge \text{Pixar}(z_a) \wedge \text{arg}_1(x_e, y_a) \wedge \text{arg}_2(x_e, z_a)$$

(c) The composed lambda-calculus expression.

Fig. 4.46 Dependency tree to lambda expression (Image from [267]).

Dong and Lapata later introduced coarse-to-fine decoding [95] using the same datasets as their 2016 paper (see Figure 4.48). The idea in this work is to first generate a **meaning sketch** during the coarse stage, then, during the fine stage, get the decoder to fill in the missing details in the meaning representation (Figure 4.49). Finally, they also test their method on the WikiSQL dataset [388].

Here we will also briefly mention the work by [301]. This paper introduces a context-dependent model for converting utterances to executable formal queries. The interaction with ATIS is shown in Figure 4.52. The annotated SQL queries are shown in Figure 4.53.

$$
\begin{array}{lll}
\text{acquired} & \Rightarrow & \lambda x.\, \text{acquired}(x_e) \\
\text{Disney} & \Rightarrow & \lambda y.\, \text{Disney}(y_a) \\
\text{Pixar} & \Rightarrow & \lambda z.\, \text{Pixar}(z_a)
\end{array}
\qquad
\begin{array}{l}
\lambda x.\, \exists yz.\, \text{acquired}(x_e) \wedge \text{Disney}(y_a) \\
\qquad \wedge \text{Pixar}(z_a) \wedge \text{arg}_1(x_e, y_a) \wedge \text{arg}_2(x_e, z_a)
\end{array}
$$

Fig. 4.47: Sample lambda expressions for single words and sample parse (Image from [267]).

Dataset	Length	Example
GEO	7.6	x : *which state has the most rivers running through it?*
	13.7	y : (argmax $0 (state:t $0) (count $1 (and (river:t $1) (loc:t $1 $0))))
	6.9	a : (argmax#1 state:t@1 (count#1 (and river:t@1 loc:t@2)))
ATIS	11.1	x : *all flights from dallas before 10am*
	21.1	y : (lambda $0 e (and (flight $0) (from $0 dallas:ci) (< (departure_time $0) 1000:ti)))
	9.2	a : (lambda#2 (and flight@1 from@2 (< departure_time@1 ?)))
DJANGO	14.4	x : *if length of bits is lesser than integer 3 or second element of bits is not equal to string 'as' ,*
	8.7	y : if len(bits) < 3 or bits[1] != 'as':
	8.0	a : if len (NAME) < NUMBER or NAME [NUMBER] != STRING :
WIKISQL	17.9	Table schema: ‖*Pianist*‖*Conductor*‖*Record Company*‖*Year of Recording*‖*Format*‖
	13.3	x : *What record company did conductor Mikhail Snitko record for after 1996?*
	13.0	y : `SELECT` *Record Company* `WHERE` (*Year of Recording* > *1996*) `AND` (*Conductor* = *Mikhail Snitko*)
	2.7	a : `WHERE` > `AND` =

Fig. 4.48: Natural language expressions x, meaning representations y, and meaning sketches a (Image from [95]).

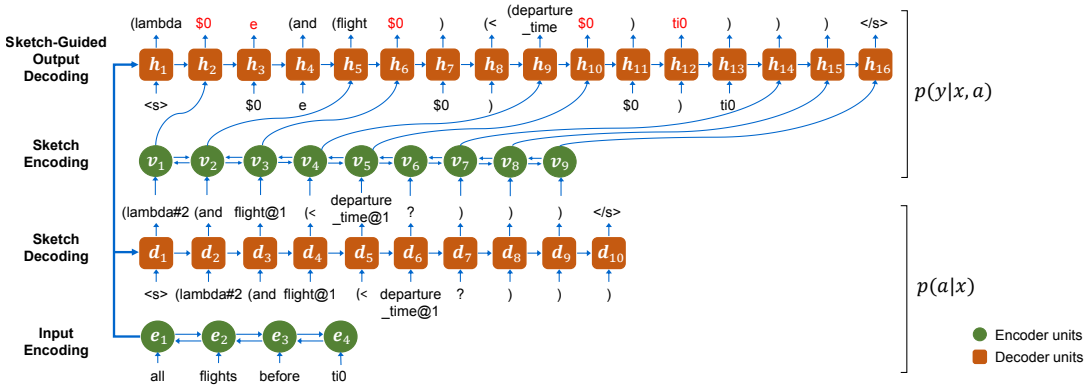


Fig. 4.49: Converting the natural language utterance x to a meaning sketch a (Image from [95]).

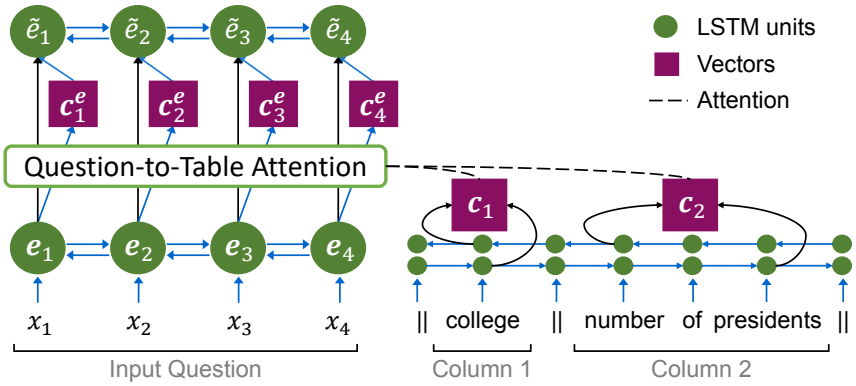


Fig. 4.50: The input encoder used for WikiSQL is shown on the left and the table column encoder is on the right (Image from [95]).

(lambda2 (and flight@1 from@2 (< departure time@1 ?)))
(lambda $0 e (and (flight $0) (from $0 dallas:ci) (< departure-time 0) 1000:ti

Fig. 4.51: Example of a meaning sketch example and low-level details (e.g., arguments and variable names) for the input "*all flights from dallas before 10am*" (Image slightly modified from [95]).

show me flights from seattle to boston next monday
[Table with 31 flights]
on american airlines
[Table with 5 flights]
which ones arrive at 7pm
[No flights returned]
show me delta flights
[Table with 5 flights]
. . .

Fig. 4.52 Sample interaction with ATIS (Image from [301]).

Useful Links

https://en.wikipedia.org/wiki/Semantic_parsing
http://nlpprogress.com/english/semantic_parsing.html
https://www.youtube.com/watch?v=liMJS5DrnlQ
https://www.cl.uni-heidelberg.de/statnlpgroup/nlmaps/
https://blog.lambdo.com/spegql-openai-scholars-final-project/
https://paperswithcode.com/task/semantic-parsing

$\bar{x}_1$: *show me flights from seattle to boston next monday*
$\bar{y}_1$:
```
(SELECT DISTINCT flight.flight_id FROM flight WHERE (flight.from_airport IN (SELECT
airport_service.airport_code FROM airport_service WHERE airport_service.city_code IN (SELECT
city.city_code FROM city WHERE city.city_name = 'SEATTLE'))) AND (flight.to_airport IN (SELECT
airport_service.airport_code FROM airport_service WHERE airport_service.city_code IN (SELECT
city.city_code FROM city WHERE city.city_name = 'BOSTON'))) AND (flight.flight_days IN (SELECT
days.days_code FROM days WHERE days.day_name IN (SELECT date_day.day_name FROM date_day WHERE
date_day.year = 1993 AND date_day.month_number = 2 AND date_day.day_number = 8))));
```
$\bar{x}_2$: *on american airlines*
$\bar{y}_2$:
```
(SELECT DISTINCT flight.flight_id FROM flight WHERE (flight.airline_code = 'AA') AND (flight.from_airport
IN (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city_code IN (SELECT
city.city_code FROM city WHERE city.city_name = 'SEATTLE'))) AND (flight.to_airport IN (SELECT air
port_service.airport_code FROM airport_service WHERE airport_service.city_code IN (SELECT city.city_code
FROM city WHERE city.city_name = 'BOSTON'))) AND (flight.flight_days IN (SELECT days.days_code FROM
days WHERE days.day_name IN (SELECT date_day.day_name FROM date_day WHERE date_day.year = 1993 AND
date_day.month_number = 2 AND date_day.day_number = 8))));
```
$\bar{x}_3$: *which ones arrive at 7pm*
$\bar{y}_3$:
```
(SELECT DISTINCT flight.flight_id FROM flight WHERE (flight.airline_code = 'AA') AND (flight.from_airport
IN (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city_code IN (SELECT
city.city_code FROM city WHERE city.city_name = 'SEATTLE'))) AND (flight.to_airport IN (SELECT air
port_service.airport_code FROM airport_service WHERE airport_service.city_code IN (SELECT city.city_code
FROM city WHERE city.city_name = 'BOSTON'))) AND (flight.flight_days IN (SELECT days.days_code FROM
days WHERE days.day_name IN (SELECT date_day.day_name FROM date_day WHERE date_day.year = 1993 AND
date_day.month_number = 2 AND date_day.day_number = 8))) AND (flight.arrival_time = 1900));
```
$\bar{x}_4$: *show me delta flights*
$\bar{y}_4$:
```
(SELECT DISTINCT flight.flight_id FROM flight WHERE (flight.airline_code = 'DL') AND (flight.from_airport
IN (SELECT airport_service.airport_code FROM airport_service WHERE airport_service.city_code IN (SELECT
city.city_code FROM city WHERE city.city_name = 'SEATTLE'))) AND (flight.to_airport IN (SELECT air
port_service.airport_code FROM airport_service WHERE airport_service.city_code IN (SELECT city.city_code
FROM city WHERE city.city_name = 'BOSTON'))) AND (flight.flight_days IN (SELECT days.days_code FROM
days WHERE days.day_name IN (SELECT date_day.day_name FROM date_day WHERE date_day.year = 1993 AND
date_day.month_number = 2 AND date_day.day_number = 8))));
```

Fig. 4.53: Annotated SQL queries (Image from [301]).

4.5 Text to SQL

Translating natural language queries to SQL in a robust manner poses numerous challenges. The underlying issues include open domain, elaborate database schemas, diverse language formulations, as well as recursion and compositionality.

Consider Figure 4.54. The user question is "*Which European countries have some players who won the Australian Open at least 3 times?*". Finding an answer to this question requires a JOIN of three tables: `Matches`, `Ranking`, and `Players`. The stages involved in the process include mapping entities to cell values (Figure 4.55), mapping question terms to column headers and table names (Figure 4.56 and Figure 4.57), and then dealing with aggregate queries (Figure 4.58).

An early paper, Neural Enquirer [361], introduced a neural architecture (Figure 4.59) to map natural language queries to tables. It performed end-to-end neural query translation and execution. Then, SQL-Net [352] uses column attention and a sketch-based approach to generate SQL as slot filling, without a need for reinforcement learning.

One factor hindering earlier work on the text-to-SQL task was the lack of large scale and diverse datasets for model training and evaluation. In recent years, multiple datasets were introduced to overcome this challenge and have been serving as the driving force along with more powerful neural models towards rapid progresses on this important yet challenging problem.

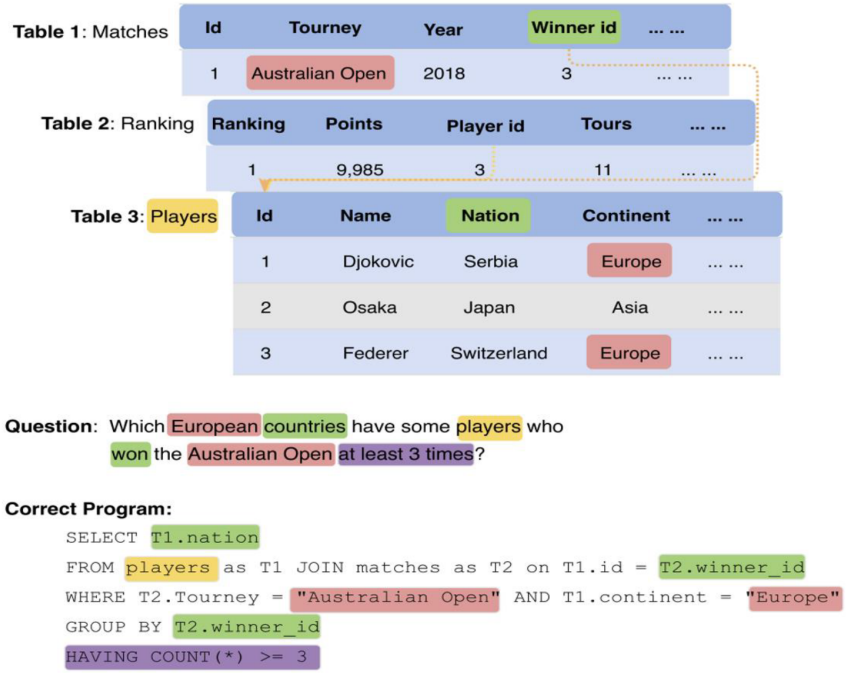


Fig. 4.54: Text to SQL example. The example uses different colors to map different words from the question to different parts of the database. Cell values are colored in pink, table names are in yellow, while table attributes are in green. The phrase "*at least 3 times*", colored in purple is mapped to the aggregate command "`HAVING COUNT(*) >= 3''.`

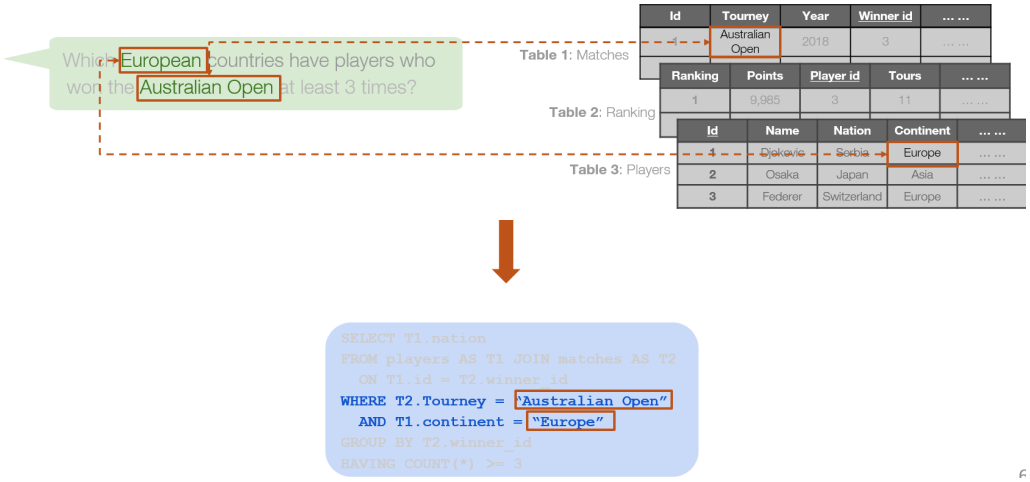


Fig. 4.55: Mapping entities to cell values.

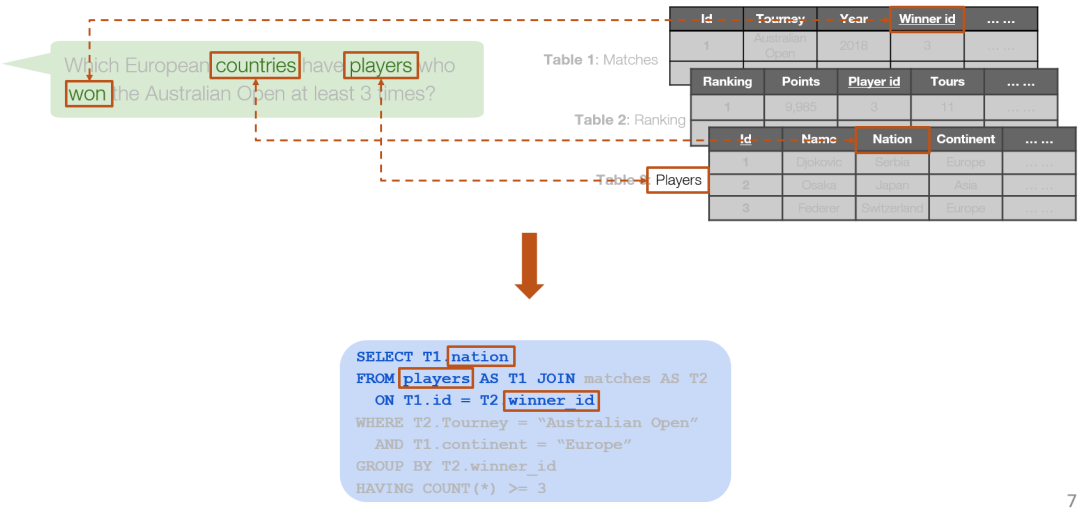


Fig. 4.56: Columns and table names 1.

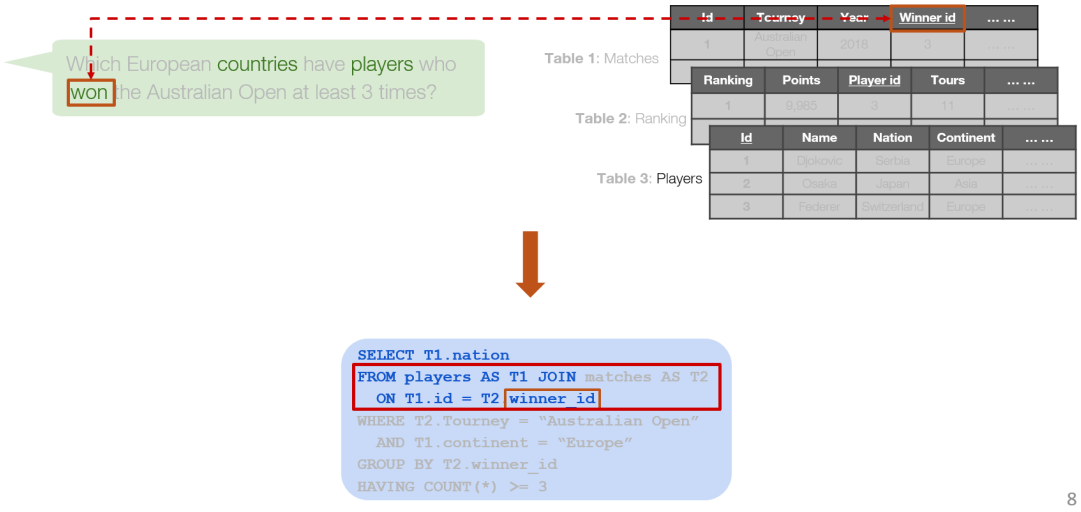


Fig. 4.57: Columns and table names 2.

4.5.1 WikiSQL

The WikiSQL[9] [388] dataset was created by crowdsourcing and using Wikipedia tables as a source of data. Given an automatically generated question at a time, the crowd workers were asked to paraphrase it. WikiSQL includes 80,654 examples from 24,241 schemas, all annotated with logical forms. A demo can be seen online [10] The system uses a deep network and refers to the SQL structure in order to prune the

[9] https://github.com/salesforce/WikiSQL

[10] https://techcrunch.com/wp-content/uploads/2017/08/unnamed2.gif

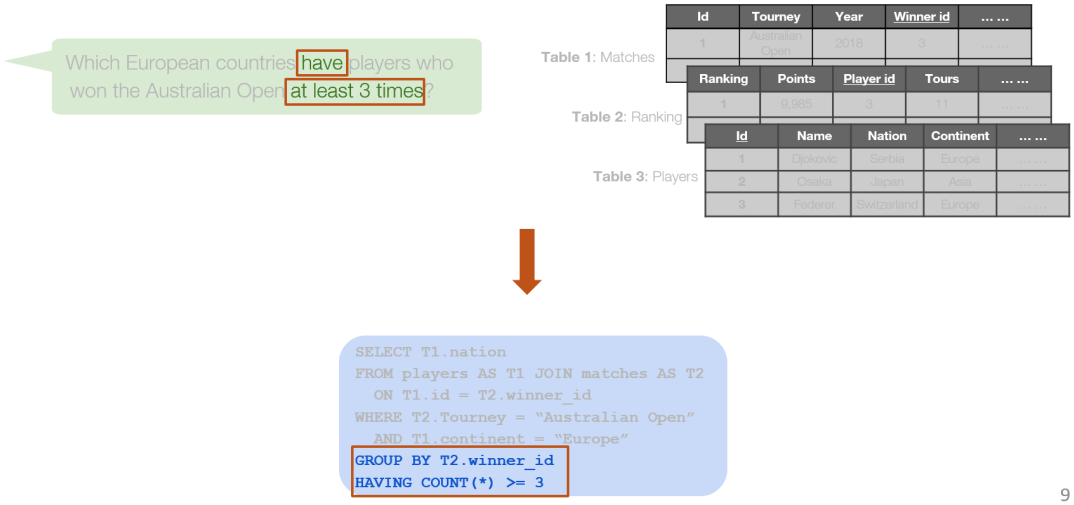


Fig. 4.58: Aggregate queries (e.g., using "`GROUP BY`").

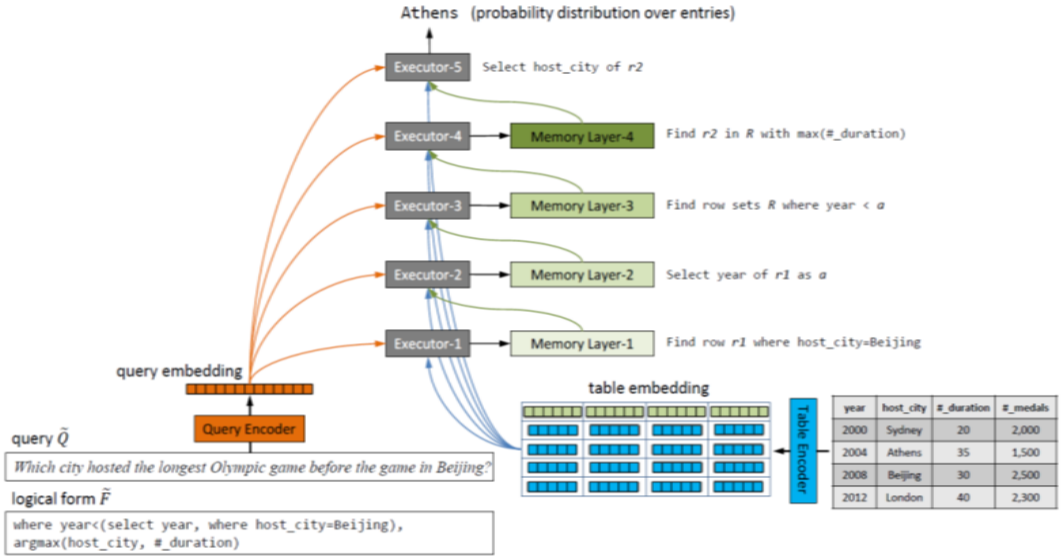


Fig. 4.59: Neural Enquirer (Image from [361]).

generated query space. For the aggregation operations, the network produces one of the following labels: COUNT, MAX, MIN, or NONE. The paper introduces a policy-based reinforcement learning method called Seq2SQL (Figure 4.60). Using policy-based reinforcement learning in addition to the standard cross-entropy loss makes it possible to optimize the (unordered) conditions of the query. Some sample outputs of Seq2SQL are shown in Figure 4.61.

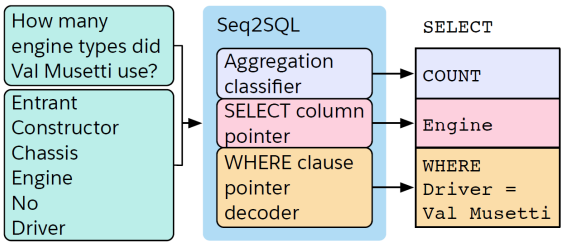


Fig. 4.60 WikiSQL text to SQL example (Image from [388]).

Q	when connecticut & villanova are the regular season winner how many tournament venues (city) are there?
P	SELECT COUNT tournament player (city) WHERE regular season winner city) = connecticut & villanova
S'	SELECT COUNT tournament venue (city) WHERE tournament winner = connecticut & villanova
S	SELECT COUNT tournament venue (city) WHERE regular season winner = connecticut & villanova
G	SELECT COUNT tournament venue (city) WHERE regular season winner = connecticut & villanova
Q	what are the aggregate scores of those races where the first leg results are 0-1?
P	SELECT aggregate WHERE 1st . = 0-1
S'	SELECT COUNT agg. score WHERE 1st leg = 0-1
S	SELECT agg. score WHERE 1st leg = 0-1
G	SELECT agg. score WHERE 1st leg = 0-1
Q	what is the race name of the 12th round trenton, new jersey race where a.j. foyt had the pole position?
P	SELECT race name WHERE location = 12th AND round position = a.j. foyt, new jersey AND
S'	SELECT race name WHERE rnd = 12 AND track = a.j. foyt AND pole position = a.j. foyt
S	SELECT race name WHERE rnd = 12 AND pole position = a.j. foyt
G	SELECT race name WHERE rnd = 12 AND pole position = a.j. foyt
Q	what city is on 89.9?
P	SELECT city WHERE frequency = 89.9
S'	SELECT city of license WHERE frequency = 89.9
S	SELECT city of license WHERE frequency = 89.9
G	SELECT city of license WHERE frequency = 89.9

Fig. 4.61: WikiSQL examples (Image from [388]).

4.5.2 Dataset splits

Finegan-Dollak et al. [104] looked into the issue of training and test splits for common datasets often containing very similar data points. The paper proposed a split that ensures that any related examples appear in the same fold (either just training, or just test). The authors also showed that with such splits, previously reported performance results for many datasets appear too optimistic and that the revised results on the new splits often show very low scores (see Section 5.5.4.1 for more details).

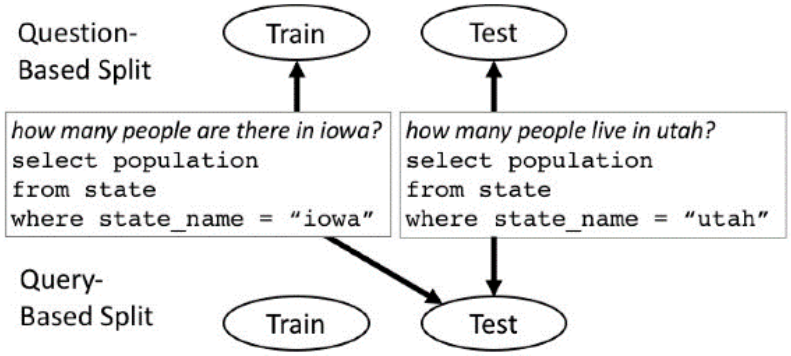


Fig. 4.62 Proper train/test split. (Image from [104]).

A result from this work is the University of Michigan text2sql-data corpus[11] that includes the properly split SQL datasets from [104], based on [76, 117, 150, 189, 252, 306, 353, 371, 379, 388].

4.5.3 Spider

The **Spider** dataset[12] (Figure 4.64), introduced by Yu et al. [372], is another influential open-domain text-to-sql dataset. Its focus is on multi-table joins (JOIN, UNION) as well as aggregate queries (e.g., GROUP BY, MAX, SUM) and nested queries. It includes fewer queries and tables than WikiSQL but it has been judged as having better quality and complexity. It includes single-turn questions and 140 databases in its training set and another 60 unrelated databases as part of the test set.

Building Spider took a large amount of effort over several months by 11 Yale students. The breakdown is as follows: Database creation (150 ph = person x hours), question and SQL annotation (500 ph), SQL review (150 ph), question reviewing and paraphrasing (150 ph), and final review and processing (another 150 ph).

The data points in Spider are labeled by difficulty, as determined by the number and type of SQL keywords. As the paper indicates, a query is counted as hard if it has "more than two SELECT columns, more than two WHERE conditions, and GROUP BY two columns, or contains EXCEPT or nested queries". To be labeled as extra hard, a query has to be even more complex. Some examples are shown in Figures 4.65, 4.66, 4.67, and 4.68. Statistics about Spider and WikiSQL are shown in Figure 4.69.

4.5.4 Extensions to Spider

A number of extensions to Spider have been created over the years. We will now introduce some of them briefly. **CoSQL**[13] [368] is an extension of Spider to a multi-turn, dialogue context. An example from the CoSQL dataset is shown in Figure 4.70. It includes 30,000+ turns plus 10,000+ annotated SQL queries, collected using a Wizard-of-Oz (WOZ) approach from 3,000+ dialogues about the Spider dataset of 200 databases covering 138 domains. The dialogues are intended to match a real world scenario that involves a

[11] https://github.com/jkkummerfeld/text2sql-data/
[12] https://yale-lily.github.io//spider
[13] https://yale-lily.github.io/cosql

```
"query-split": "test",
 "sentences": [
     {
         "question-split": "train",
         "text": "Is course number0 available to undergrads ?",
         "variables": {
             "department0": "",
             "number0": "519"
         }
     }
 ],
 "sql": [
     "SELECT DISTINCT COURSEalias0.ADVISORY_REQUIREMENT ,
                      COURSEalias0.ENFORCED_REQUIREMENT ,
                      COURSEalias0.NAME FROM COURSE AS COURSEalias0
      WHERE COURSEalias0.DEPARTMENT = \"department0\"
      AND COURSEalias0.NUMBER = number0 ;"
 ],
 "variables": [
     {
         "example": "EECS",
         "location": "sql-only",
         "name": "department0",
         "type": "department"
     },
     {
         "example": "595",
         "location": "both",
         "name": "number0",
         "type": "number"
     }
 ]
}
```


Fig. 4.63: Sample data point from the text2sql-data corpus (Image from [104]).

crowd worker and a SQL expert who can retrieve answers using SQL, possibly in the process having to clarify ambiguous questions or to inform the crowd worker that the query is unanswerable. If a query is answerable, the SQL expert describes the schema, the SQL query, and the execution results in a natural dialogue format. Compared to traditional task-oriented dialogue datasets, here the dialogue states are grounded in SQL. Furthermore, testing is done on new domains and hence systems that do well on the CoSQL dataset have to be domain-independent. CoSQL contains three challenges: SQL-grounded dialogue state tracking, response generation from query results, and user dialogue act prediction.

EditSQL[14], [382] generates sequential SQL statements by editing the most recent statements. An example is shown in Figure 4.71. The editing component replaces "`id`" by "`name`" in the latter SQL query.

SpaRC[15] was introduced in [375]. It is a dataset for multi-turn, cross-domain semantic parsing in context. It includes 4,298 question sequences including 12,000+ questions, all annotated with SQL queries. Just like CoSQL, it is based on the original Spider dataset. An example illustrating the sequential aspect of SpaRC is included in Figure 4.72.

[14] https://github.com/ryanzhumich/editsql

[15] https://yale-lily.github.io/sparc

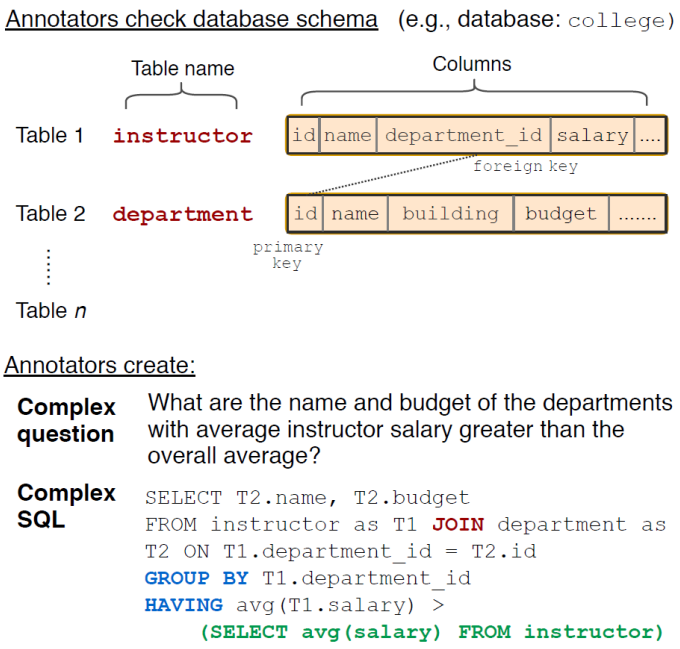


Fig. 4.64 Creation of the Spider dataset (Image from [372]).

```
SELECT COUNT(*)
FROM cars_data
WHERE cylinders > 4
```

Fig. 4.65: Easy Spider example: "*What is the number of cars with more than 4 cylinders?*". (Example from [372])

```
SELECT T2.name, COUNT(*)
FROM concert AS T1
JOIN stadium AS T2 ON T1.stadium_id = T2.stadium_id
GROUP BY T1.stadium_id
```

Fig. 4.66: Medium Spider example: "*For each stadium, how many concerts are there?*". (Example from [372])

```
SELECT T1.country_name
FROM countries AS T1
JOIN continents AS T2 ON T1.continent = T2.cont_id
JOIN car_makers AS T3 ON T1.country_id = T3.country
WHERE T2.continent = 'Europe'
GROUP BY T1.country_name
HAVING COUNT(*) >= 3
```

Fig. 4.67: Hard Spider example: "*Which countries in Europe have at least 3 car manufacturers?*". (Example from [372])

```
SELECT AVG(life_expectancy)
FROM country
WHERE name NOT IN
  (SELECT T1.name
   FROM country AS T1
   JOIN country_language AS T2 ON T1.code = T2.country_code
   WHERE T2.language = ''English''
     AND T2.is_official = ''T'')
```

Fig. 4.68: Very Hard Spider example: "*What is the average life expectancy in the countries where English is not the official language?*". (Example from [372])

	Spider			WikiSQL	
	# Q	# SQL	#DB	# Q	# Table
Train	8,695	4,730	140	56,355	17,984
Dev	1,034	564	20	8,421	1,621
Test	2,147	–	40	15,878	2,787

Fig. 4.69 Spider and WikiSQL statistics. (Image from [203]).

TypeSQL was introduced in [365]. It uses a sketch-based method to fill the slots. It makes use of two bidirectional LSTMs to encode the types and column names of the question words. The hidden states of the LSTMs predict the values in the sketch. An example from TypeSQL is included in Figure 4.73.

Finally, **SyntaxSQLNet** was introduced in [367]. It uses a SQL specific tree decoder for complex and cross-domain text-to-sql generation. An example is shown in Figure 4.74. It uses attention encoders for the SQL path history and table structure as well as a SQL specific syntax tree-based decoder. An example from SyntaxSQLNet is included in Figure 4.75. Some of the SQL grammar is shown in Figure 4.76.

4.5.5 Selective Recent Papers

RAT-SQL[16] was introduced in [327]. It focuses on domain generalization. It uses a relation-aware self-attention method to address, in a unified way, multiple tasks, including schema encoding, schema linking, and feature representation. It uses a question contextualized schema graph and predefined schema relations. The edges correspond to schema relations and are also used to link by name and value. A transformer trained for relation aware self attention identifies known relations. At the time of its publication, it had achieved SOTA on the Spider leaderboard. Examples are shown in Figure 4.77.

SQUALL [288] is a dataset that takes 11,726 questions from WikiTableQuestions and matches them with manually written SQL statements. The SQL statements are further aligned to the question fragments. Examples from SQUALL are shown in Figure 4.78.

BRIDGE[17] was introduced in [203]. The system converts a given question and schema into a sequence in which some of the fields are augmented with some of the cell values related to the question. At the time of the paper's publication, BRIDGE achieved SOTA on Spider and WikiSQL. An example is shown in Figure 4.79.

[16] https://github.com/Microsoft/rat-sql

[17] https://github.com/salesforce/TabularSemanticParsing

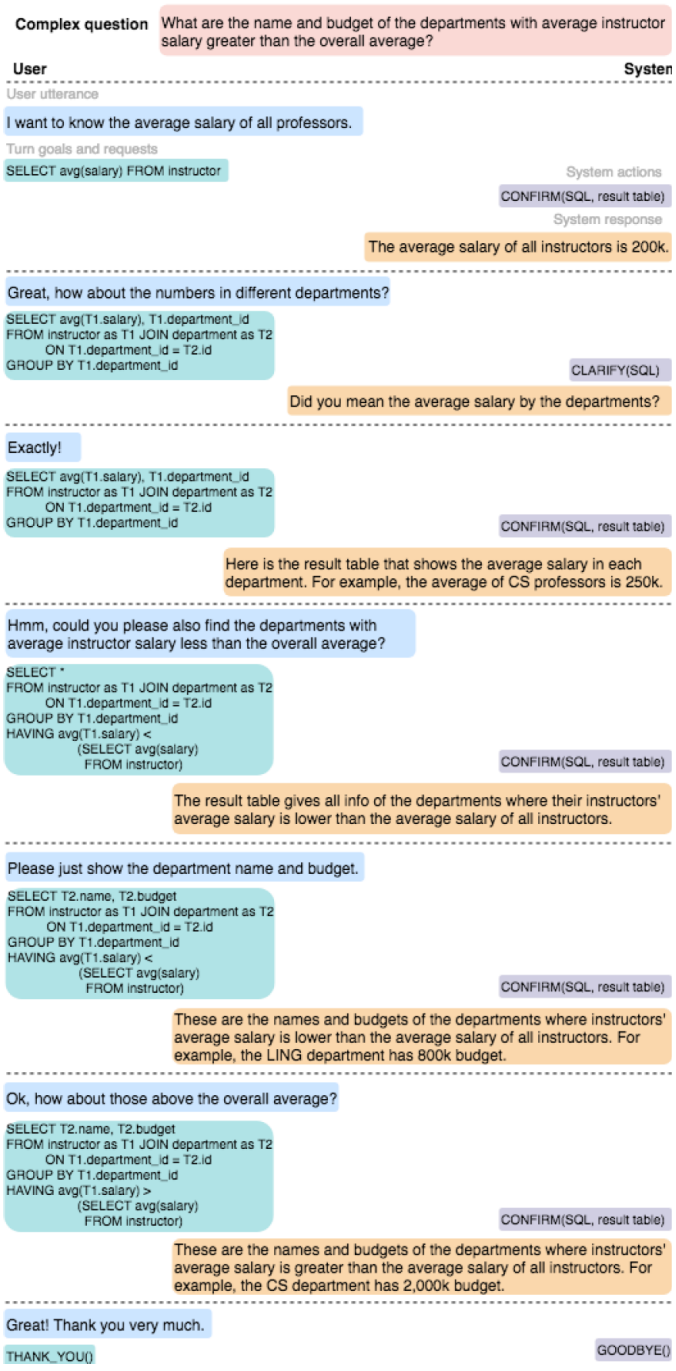


Fig. 4.70 Example from CoSQL. (Image from [368]).

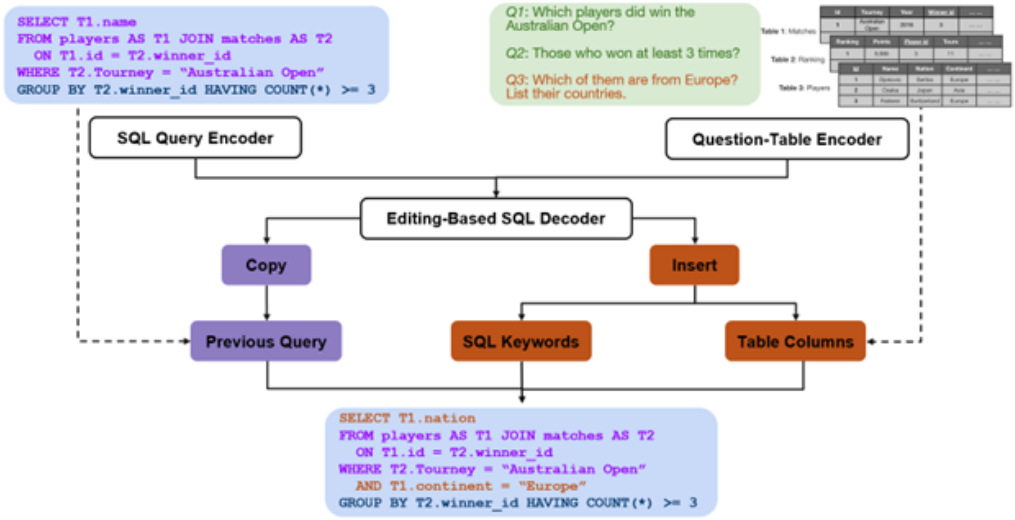


Fig. 4.71: EditSQL example (alt). (Image from [382]).

STRUG[18] is about structure-grounded pretraining. It was introduced in Deng et al. [87]. The authors describe a method for text-to-table alignment using a parallel corpus. The pretraining tasks that they take on are column grounding, value grounding, and column-value mapping.

PICARD was introduced in [275] to address the problem caused by the unconstrained output space available at each decoding step. The authors present a method that prevents the generation of invalid code through the use of incremental parsing. At each decoding step, PICARD rejects the inadmissible tokens. When evaluated on Spider and CoSQL, PICARD achieves SOTA performance. We will discuss this paper in more detail in Chapter 5.

Another recent paper [265] uses an approach that is very representative of the most recent trends in this field. More specifically, it employs a pretrained language model (in this case, Codex) in a zero-shot setting to achieve a slightly better test suite accuracy [19] than fine-tuned T5-base on the Spider dataset without any Spider training data or finetuning. Figures 4.81, 4.82, and 4.83 show examples of the inputs to Codex.

DIN-SQL is a recent work [254] that uses large language models in a few-shot setting and achieves a new SOTA performance on the Spider dataset. In particular, each text-to-SQL task is decomposed into four sub-tasks: (1) schema linking, (2) query classification and decomposition, (3) SQL generation and (4) self-correction. All sub-tasks are solved using the prompting technique of large language models and the output of each intermediate sub-task (e.g. schema linking) is used as an input in other tasks (e.g. SQL generation). Examples of the schema linking module and query classification and decomposition module are respectively shown in Figures 4.84 and 4.85. Before DIN-SQL, the SOTA on the holdout test set of Spider, in terms of test suite accuracy, was 79.9, which belonged to RESDSQL [190], a fine-tuned encoder-decoder model. DIN-SQL achieves a test suite accuracy of 85.3 using GPT-4 and a test suite accuracy of 78.2 using Codex Davinci with in-context learning using only a few demonstrations and with no pretraining or finetuning the Spider training set. While the use of large language models is a contributing factor in these

[18] https://aka.ms/strug

[19] See Section 5.3.1 for more details of the metric.

D_1 : Database about student dormitory containing 5 tables.

C_1 : **Find the first and last names of the students who are living in the dorms that have a TV Lounge as an amenity.**

Q_1 : How many dorms have a TV Lounge?

S_1 :
```
SELECT COUNT(*) FROM dorm AS T1 JOIN has_amenity
AS T2 ON T1.dormid = T2.dormid JOIN dorm_amenity
AS T3 ON T2.amenid = T3.amenid WHERE
T3.amenity_name = 'TV Lounge'
```

Q_2 : What is the total capacity of these dorms?

S_2 :
```
SELECT SUM(T1.student_capacity) FROM dorm AS T1
JOIN has_amenity AS T2 ON T1.dormid = T2.dormid
JOIN dorm_amenity AS T3 ON T2.amenid = T3.amenid
WHERE T3.amenity_name = 'TV Lounge'
```

Q_3 : How many students are living there?

S_3 :
```
SELECT COUNT(*) FROM student AS T1 JOIN lives_in
AS T2 ON T1.stuid = T2.stuid WHERE T2.dormid IN
(SELECT T3.dormid FROM has_amenity AS T3 JOIN
dorm_amenity AS T4 ON T3.amenid = T4.amenid WHERE
T4.amenity_name = 'TV Lounge')
```

Q_4 : Please show their first and last names.

S_4 :
```
SELECT T1.fname, T1.lname FROM student AS T1 JOIN
lives_in AS T2 ON T1.stuid = T2.stuid WHERE
T2.dormid IN (SELECT T3.dormid FROM has_amenity
AS T3 JOIN dorm_amenity AS T4 ON T3.amenid =
T4.amenid WHERE T4.amenity_name = 'TV Lounge')
```

D_2 : Database about shipping company containing 13 tables

C_2 : **Find the names of the first 5 customers.**

Q_1 : What is the customer id of the most recent customer?

S_1 :
```
SELECT customer_id FROM customers ORDER BY
date_became_customer DESC LIMIT 1
```

Q_2 : What is their name?

S_2 :
```
SELECT customer_name FROM customers ORDER BY
date_became_customer DESC LIMIT 1
```

Q_3 : How about for the first 5 customers?

S_3 :
```
SELECT customer_name FROM customers ORDER BY
date_became_customer LIMIT 5
```

Fig. 4.72 SpaRC example (Image from [375]).

results, decomposing the task also plays a major role, leading to significant improvement especially for hard and extra-hard queries

Note that the Spider leaderboard and the recent works including DIN-SQL have switched to test suite accuracy as of November 2020, and what used to be test suite accuracy is now simply referred to as the execution accuracy [20]

[20] This name switch has led to some confusion with some papers comparing their execution accuracy to the test suite accuracy reported on the leaderboad [205].

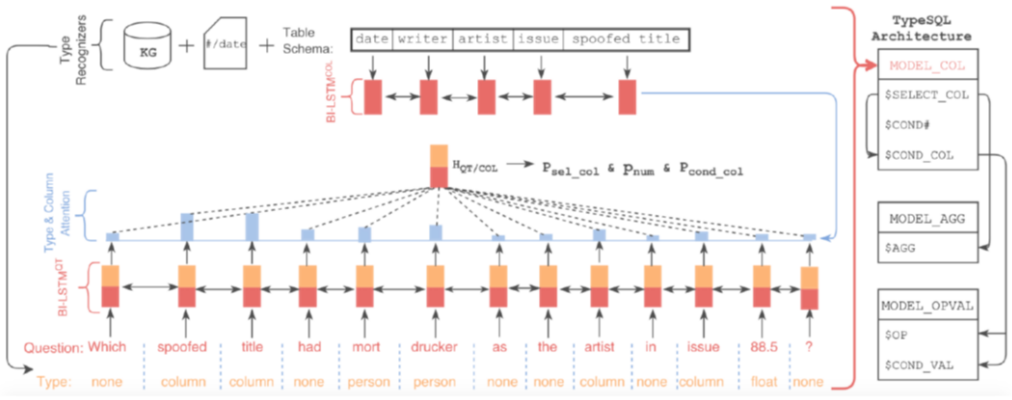


Fig. 4.73: TypeSQL (Image from [365]).

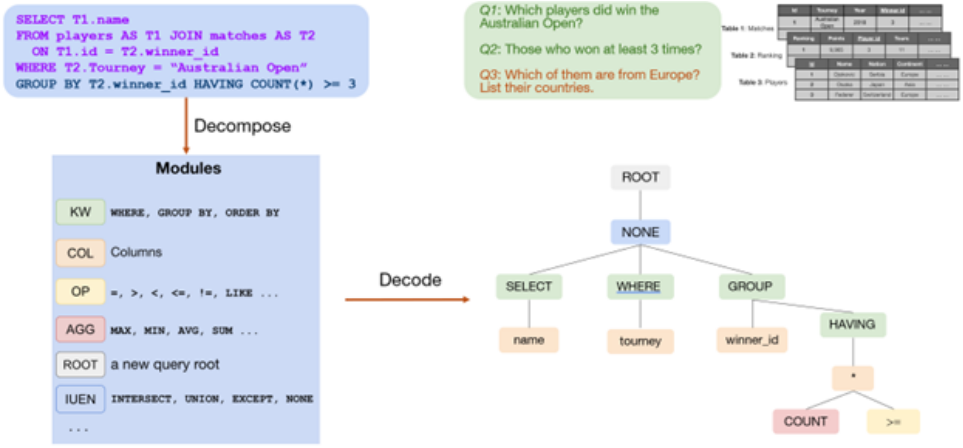


Fig. 4.74: SyntaxSQLNet pipeline. (Image from [367])

4.6 Summary

In this chapter we covered representations for text-to-data, semantic parsing, sequence to sequence translation, text to SQL, and other related topics. We will turn next to evaluation, data-to-text, and interactivity.

4.6.1 Further Reading

We refer the readers to [2] for an earlier survey of this topic. For more recent work, we refer the readers to a few recent tutorials: [86, 164, 165, 171, 261].

As of the writing of this book, a new paradigm has emerged in NLP. Through the use of prompts and in-context learning, multi-task learning (Big-Bench, T5, T0, Unified SKG), in conjunction with large language

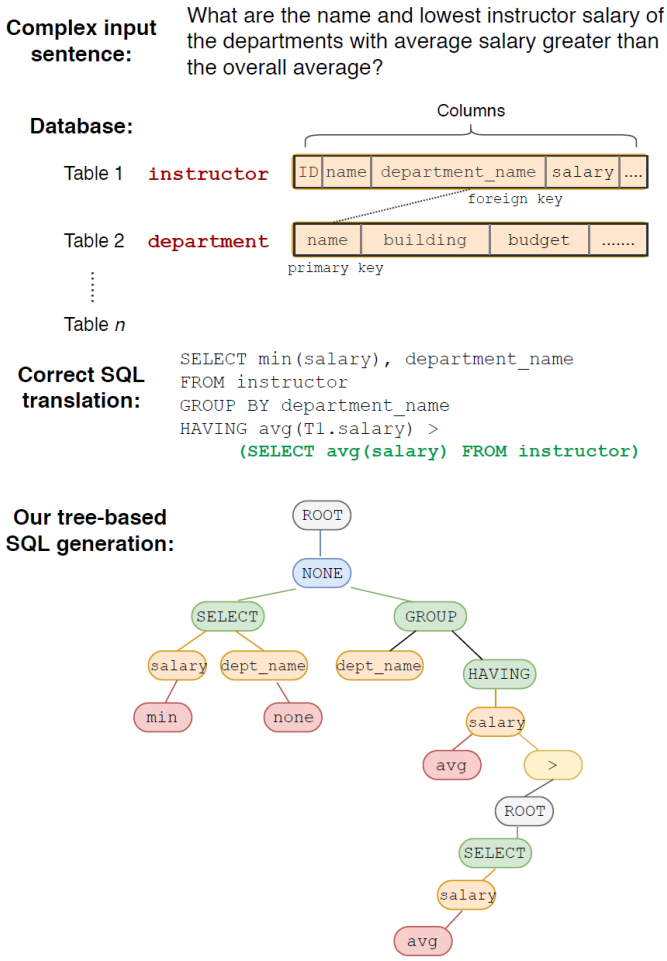


Fig. 4.75 Complex text to SQL generation (Image from [367]).

models, or through the use of models trained to generate executable code, it appears to be possible to build NLIDB systems without task-specific training or finetuning. We have covered some of such recent work and will be following closely the literature on these topics and consider writing a new chapter for a future update to this book. In the meantime, we refer the readers to a recent survey of deep learning methods for text-to-SQL: [167].

Additional Links

https://wp.sigmod.org/?p=2897
https://medium.com/mytake/literature-survey-natural-language-interfaces-for-data-bases-nlidb-6a86504f72c3
https://medium.com/visionwizard/text2sql-part-1-introduction-1327cf7f5f51
https://medium.com/visionwizard/text2sql-part-2-datasets-4ef61359e90e
https://medium.com/visionwizard/text2sql-part-3-baseline-models-85f14fba4122
https://medium.com/visionwizard/text2sql-part-4-state-of-the-art-models-cf81a377d4d2

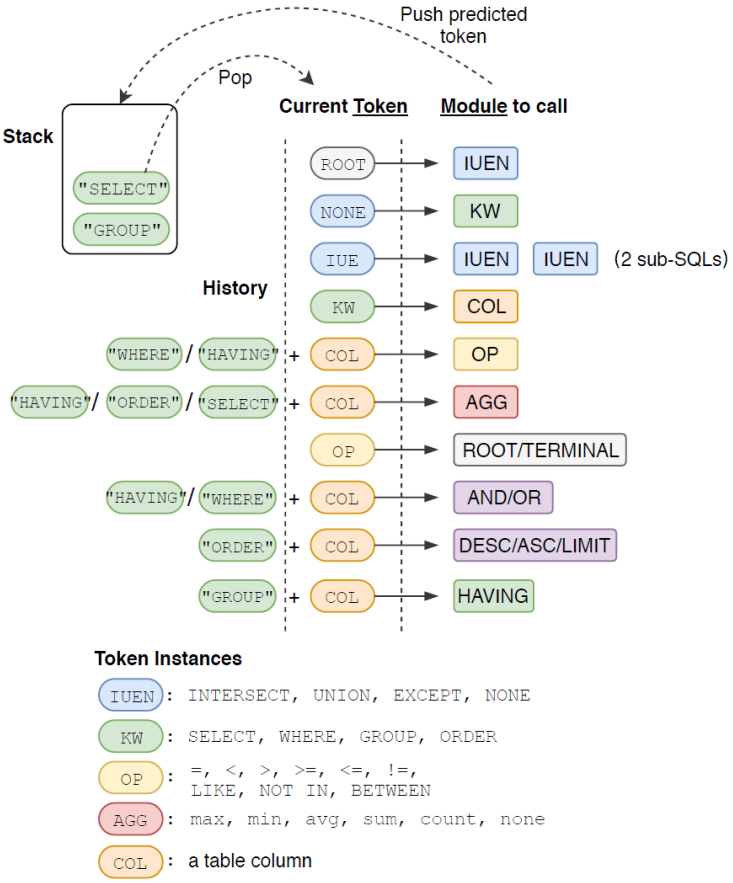


Fig. 4.76 SyntaxSQLNet: modules and grammar used in decoding (Image from [367]).

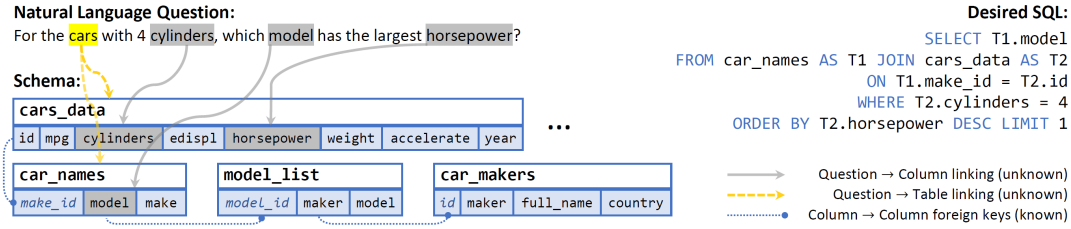


Fig. 4.77: RAT-SQL example (Image from [327]).

Table: Province of Alessandria

City (c1)	*Population (c2)*	*Area (km²) (c3)*	...
Alessandria	94191	203.97	...
Casale Monferrato	36039	86.32	...
Novi Ligure	28581	54.22	...
Tortona	27476	99.29	...
Acqui Terme	20426	33.42	...

Question: ① How many ② cities have ③ at least ④ 25,000 ⑤ people?

Target Logical Form:

```
SELECT ①count(②c1①) FROM w WHERE ⑤c2_number③>=④25000
```

Answer: 4

Table: Bulgaria at the 1988 Winter Olympics

Athlete (c1)	*Total Time (c2)*	*Total Rank (c3)*	...
Stefan Shalamanov	1:52.37	23	...
Borislav Dimitrachkov	1:50.81	19	...
Petar Popangelov	1:46.34	16	...

Question: ① Who has ② the highest ③ rank ?

Target Logical Form:

```
SELECT ①c1 FROM w ②ORDER BY ③c3_number ②LIMIT 1
```

Answer: Petar Popangelov

Fig. 4.78 Examples from SQUALL, showing the alignments of SQL queries to questions (Image from [288]).

https://medium.com/visionwizard/text2sql-part-5-final-insights-d00f85093d16
https://medium.com/@3502.stkabirdin/natural-language-to-sql-queries-d5ac1dda8404
https://www.slideshare.net/YunyaoLi/meaning-representations-for-natural-languages-design-models-and-

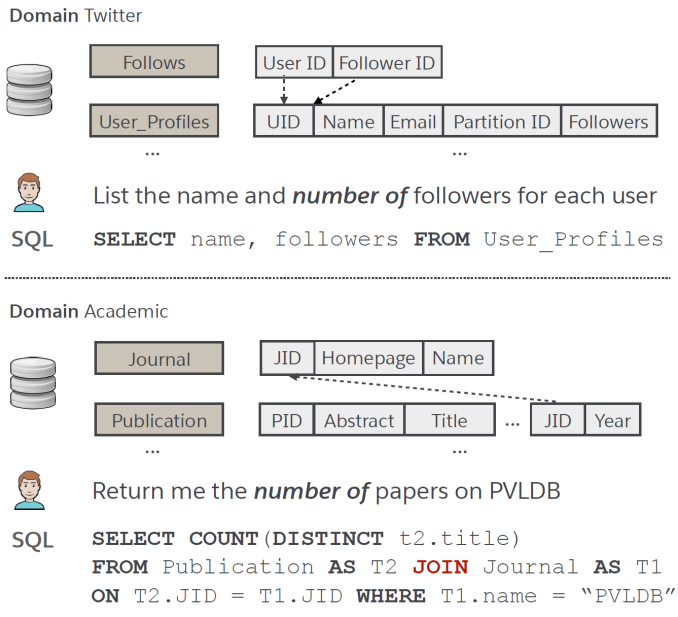


Fig. 4.79 Bridge (Image from [203]).

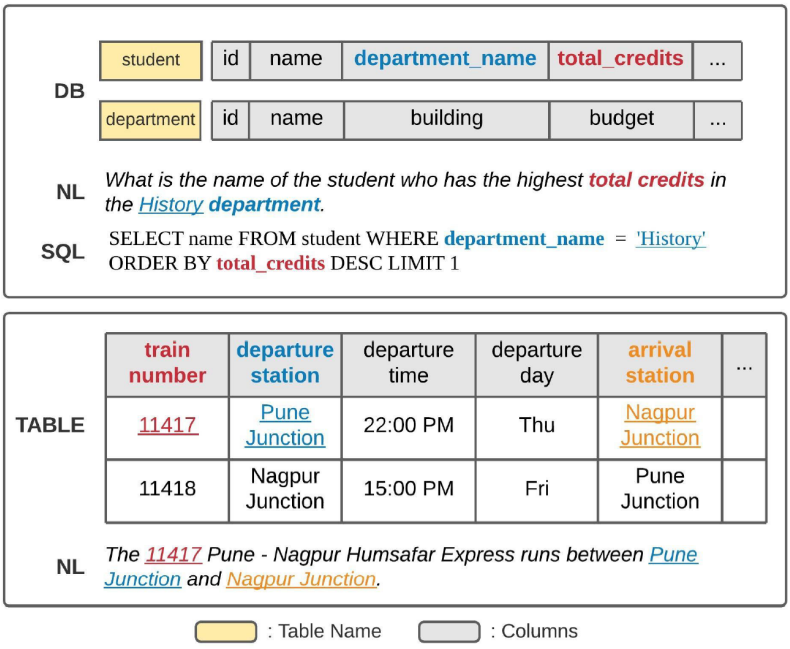


Fig. 4.80: Example from the STRUG paper [87] (Image from [87]).

```
What is Kyle's id? | network_1 | highschooler :
id, name ( Kyle ), grade | friend :
student_id, friend_id | likes :
student_id, liked_id
```

Fig. 4.81: Sample Codex prompt 1 (Image from [265]).

```
-- Using valid SQLite, answer the following questions.

-- What is Kyle's id?
SELECT
```

Fig. 4.82: Sample Codex prompt 2 (Image from [265]).

```
### SQLite SQL tables, with their properties:
#
# Highschooler(ID, name, grade)
# Friend(student_id, friend_id)
# Likes(student_id, liked_id)
#
### What is Kyle's id?
SELECT
```

Fig. 4.83: Sample Codex prompt 3 (Image from [265]).

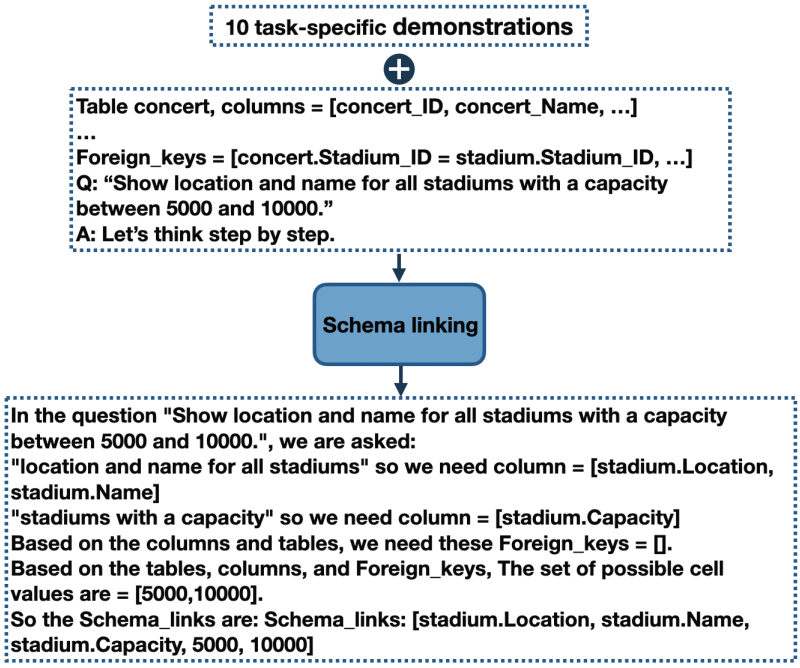


Fig. 4.84: Sample schema linking prompt (Image from [254]).

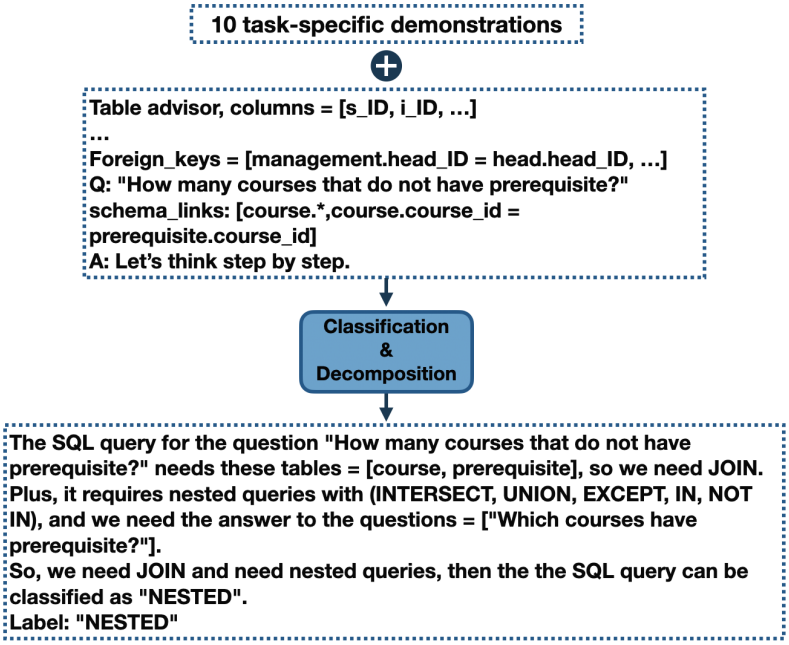


Fig. 4.85: Sample query classification and decomposition prompt (Image from [254]).