Chapter 4
Developing Conversational Natural Language Interface to a Database

Abstract In this Chapter we focus on a problem of a natural language access to a database, well-known and highly desired to be solved. We start with the modern approaches based on deep learning and analyze lessons learned from unusable database access systems. This chapter can serve as a brief introduction to neural networks for learning logic representations. Then a number of hybrid approaches are presented and their strong points are analyzed. Finally, we describe our approach that relies on parsing, thesaurus and disambiguation via chatbot communication mode. The conclusion is that a reliable and flexible database access via NL needs to employ a broad spectrum of linguistic, knowledge representation and learning techniques. We conclude this chapter by surveying the general technology trends related to NL2SQL, observing how AI and ML are seeping into virtually everything and represent a major battleground for technology providers.

4.1 Introduction

With the rapid proliferation of information in modern data-intense world, many specialists across a variety of professions need to query data stored in various relational databases. While relational algebra and its implementations in modern querying languages, such as SQL, support a broad spectrum of querying mechanisms, it is frequently hard for people other than software developers to design queries in these languages. Natural language (NL) has been an impossible dream of query interface designers, believed to be unreliable, except in limited specific circumstances. A particular case, NL interface to databases is considered as the goal for a database query interface; a number of interfaces to databases (NL2SQL) have been built towards this goal (Androutsopoulos et al. 1995; Agrawal et al. 2002; Galitsky 2005; Li et al. 2006; Bergamaschi et al. 2013).

NL2SQL have many advantages over popular query interfaces such as structured keyword-based search, form-based request interface, and visual query builder. A typical NL2SQL would enable naive users to specify complex queries without extensive training by database experts. On the other hand, single level keywords are insufficient to convey complex query logic, form-based interfaces can be used only

B. Galitsky, *Developing Enterprise Chatbots*,
https://doi.org/10.1007/978-3-030-04299-8_4

for a limited set of query types and where queries are predictable. For a novice user to employ a visual query builder, some training and solid knowledge of a database schema is required. Conversely, using an NL2SQL system, even naive users are expected to be able to accomplish logically complex query tasks, in which the target SQL statements include comparison predicates, conjunctions, quantifications, multi-level aggregations, nestings, and various types of joins, among other things.

Although NL2SQL is strongly desirable, it has not been extensively deployed yet. Microsoft has been distributing English Query product that has never become popular because of low robustness and substantial efforts in using the provided tools to build a rule-based NL model and mapping for a database. Oracle never had its database accessed via NL by a broad audience. The main reason is that it is rather hard to "understand" an NL query in a broad sense and to map a user phrase into given database field in particular (Galitsky and Grudin 2001).

A relationship between databases and chatbots is that of a mutual aid. A database is an important source of data for a chatbot. At the same time, a chatbot is a tool that facilitates error rectification in language understanding required to query databases. As a natural language query to a database is being interpreted, ambiguities arise and need to be resolved by asking a user which database table and fields she meant with her phrase.

The goal of this chapter is to explore what works and what does not work for NL2SQL. We will start with the most recent approach based on learning of a formal sequence (NL) to sequence(SQL) encoder via a neural network (Goldberg 2015). After that we consider classical approaches of 2000s based on token mappings of query words into names of tables, columns and their values. We then go deeper into the anatomy of how NL represents logic forms in general and SQL in particular, and focus on linguistic correlates of SQL. Having analyzed the shortcomings of these approaches, we formulate the one most plausible in industrial settings and dive into the steps of building SQL from text.

4.1.1 History

There is a series of ups and downs in attempts to access databases in NL. Natural language query interfaces have been attempted for decades, because of their great desirability, particularly for non-expert database users. However, it is challenging for database systems to interpret (or understand) the semantics of natural language queries. Early interactive NL2SQLs (Kupper et al. 1993) mainly focus on generating cooperative responses from query results (over-answering). Li et al. (2005) takes a step further, generating suggestions for the user to reformulate his query when it is beyond the semantic coverage. This strategy greatly reduces the user's burden in query reformulation. However, the fact that the input query is within the coverage of a prepared semantic model does not necessary mean it will be processed correctly. As new NLP techniques arise, there are new attempts to apply them to NL2SQL, not necessarily advancing a state-of-the-art but enabling the domain with new ideas and intuition of what has worked and what has not.

Early NL2SQL systems depended on hand crafted semantic grammars tailored to each individual database, which are hard to transport to other databases. Conversely, the system to be presented in the end of this chapter targets a generic query language such as SQL, as the translation goal, in an arbitrary domain with unrestricted vocabulary, as long as words in a query correspond to field names and values. We intend to build a database-independent system that learns the database structure online and prepares NL2SQL interface automatically.

Popescu et al. (2004) suggested to focus on a limited set of queries (semantically tractable) with unambiguous mapping into relations, attributes and values, and employ statistical semantic parsing. (Galitsky and Usikov 2008; Quirk et al. 2015) proposed a framework of natural language programming beyond NL2SQL, where a compiler inputs an NL description of a problem and forms a code according to this description. Galitsky et al. (2011) defined sentence generalization and generalization diagrams via a special case of least general generalization as applied to linguistic parse trees, which is an alternative way for query formation from NL expressions. Li and Jagadish (2016) proposed an NL2SQL comprising three main components: a first component that transforms a natural language query to a query tree, a second component that verifies the transformation interactively with the user, and a third component that translates the query tree into a SQL statement.

In most implementation, each SQL statement is composed from the ground up. When a query log, which contains natural language queries and their corresponding SQL statements, is available, an NL2SQL system can benefit from reusing previous SQL statements. When the new query is in the query log, NL2SQL system can directly reuse the existing SQL statement. When the new query is dissimilar to any previous queries, it can be composed from the ground up. It is promising to achieve somewhere in between, finding similar queries in the query log, reusing some of the SQL fragments, and completing the remaining parts.

A number of recent approaches have given up on feature engineering and attempt to learn NL2SQL as a sequence of symbols, via logic forms or directly (Zhong et al. 2017).

We conclude this section with a list of industrial NL2SQL Systems:

1. DataRPM, datarpm.com/product
2. Quepy (Python framework), quepy.machinalis.com
3. Oracle ATG (2010 commerce acquisition), docs.oracle.com/cd/E23507_01/Search.20073/ATGSearchQueryRef/html/s0202naturallanguagequeries01.html
4. Microsoft PowerBI, https://powerbi.microsoft.com/en-us/blog/power-bi-q-and-a-natural-language-search-over-data/
5. Wolfram natural language understanding, www.wolfram.com/natural-language-understanding/
6. Kueri allows users to navigate, explore, and present their Salesforce data with using a Google style as-you-type auto-complete suggestions. This platform is a complete library you can download and use commercially for free. The platform was developed especially for developers who would like to offer end-users the ability to interact with data using Natural Language. UX includes as-you-type smart suggestions.

4.2 Statistical and Deep Learning in NL2SQL Systems

Reinforcement Learning approach (Zhong et al. 2017) propose Seq2SQL, a deep neural network for translating natural language questions to corresponding SQL queries. This approaches takes advantage of the structure of SQL queries to significantly reduce the output space of generated queries. The rewards from in-the-loop query execution over the database support learning a policy to generate unordered parts of the query, which are shown to be less suitable for optimization via cross entropy loss.

Zhong et al. (2017) published WikiSQL, a dataset of relatively simple 80 k hand-annotated examples of questions and SQL queries distributed across 24 k tables from Wikipedia. Labeling was performed by Amazon Mechanical Turk. Each query targets a single table, and usually has a single constraint. This dataset is required to train the model and is an order of magnitude larger than comparable datasets. Attentional sequence to sequence models were considered as a baseline and delivered execution accuracy of 36% and logical form accuracy of 23%.

By applying policy-based reinforcement learning with a query execution environment to WikiSQL, Seq2SQL model significantly outperforms the baseline and gives corresponding accuracies of to 59.4% and 48.3%. Hence in spite of the huge dataset required for the accuracy, it is still fairly low. The training is very domain-dependent since the system does not differentiate between the words related to logical operations vs the words which are domain atoms. Therefore, for a real customer deployment, an extensive collection of a training set would be required, which is not very plausible. Hence we believe NL2SQL problem cannot do without an extensive feature engineering that makes it domain-independent and applicable to an arbitrary database.

One can apply RNN models for parsing natural language queries to generate SQL queries, and refine it using existing database approaches. For instance, heuristic rules could be applied to correct grammar errors in the generated SQL queries. The challenge is that a large amount of (labeled) training samples is required to train the model. One possible solution is to train a baseline model with a small dataset, and gradually refining it with user feedback. For instance, users could help correct the generated SQL query, and this feedback essentially serves as labeled data for subsequent training.

The approaches purely based on deep learning models may not be very effective. If the training dataset is not comprehensive enough to include all query patterns (some predicates could be missing), then a better approach would be to combine database solutions and deep learning.

Converting NL to SQL can be viewed from a more general framework of building a logical form representation of text, given a vast set of pairs. Semantic parsing aims at mapping natural language to machine interpretable meaning representations (Berant et al. 2002). Traditional approaches rely on high-quality lexicons, manually-built templates, and linguistic features which are either domain or representation-specific. Deep learning attention-enhanced encoder-decoder model encodes input utterances into vector representations, and generate their logical forms by conditioning the output sequences or trees on the encoding vectors. Dong and Lapata (2016) show that for a number of datasets neural attention approach performs competitively well without using hand-engineered features and is easy to adapt across domains and meaning representations. Obviously, for practical application a fairly extensive training dataset with exhaustive combination of *NL expressions – logic forms* pars would be required.

Although long short-term memory and other neural network models achieve similar or better performance across datasets and meaning representations, without relying on hand-engineered domain- or representation-specific features, they cannot be incrementally improved for a given industrial domain unless hundred or thousand-times larger training datasets (compared to the available ones) are obtained.

4.2.1 NL2SQL as Sequence Encoder

Semantic parsing aims at mapping natural language to machine interpretable meaning representations. Traditional approaches rely on high-quality lexicons, manually-built templates, and linguistic features which are either domain or representation-specific. There is a possibility of neural network based encoder-decoder model to perform semantic parsing. Utterances can be subject to vector representations, and their logical forms can be obtained by conditioning the output sequences or trees on the encoding input vectors.

It is possible to apply a machine learning approach to such a complex problem as semantic parsing because a number of corpora containing utterances annotated with formal meaning representations are available.

4.2.1.1 Sequence-to-Sequence Model

Encoder-decoder architectures based on recurrent neural networks allows for bridging the gap between NL and logical form with minimal domain knowledge. The general encoder-decoder paradigm can be applied to the semantic parsing task. Such model can learn from NL descriptions paired with meaning representations; it encodes sentences and decodes logical forms using recurrent neural networks with long short-term memory (LSTM) units.

This model regards both input q and output a as sequences. The encoder and decoder are two different L-layer recurrent neural networks with long short-term memory (LSTM) units which recursively process tokens one by one (Fig. 4.1).

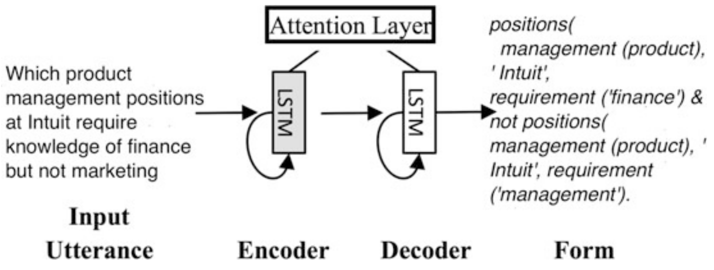

Attention Layer
Which product management positions at Intuit require knowledge of finance but not marketing
LSTM
LSTM
positions(management (product), ' Intuit', requirement ('finance') & not positions(management (product), ' Intuit', requirement ('management').
Input Utterance
Encoder
Decoder
Form

Fig. 4.1 Input utterances and their logical forms are encoded and decoded with neural networks. An attention layer is used to learn soft alignments

Dong and Lapata (2016) build a model which maps natural language input $q = x_1 \cdots x_{|q|}$ to a logical form representation of its meaning $a = y_1 \cdots y_{|a|}$. The conditional probability $p(a|q)$ is defined as:

$$p(a|q) = \prod_{t=1}^{|a|} p(\mathrm{y_t}|\mathrm{y_{<t}}, q)$$

where $y_{<t} = y_1 \cdots y_{t-1}$. The method consists from an encoder which encodes NL input q into a vector representation and a decoder which learns to generate $y_1, \cdots, y_{|a|}$ conditioned on the encoding vector.

The first $|q|$ time steps belong to the encoder, while the following $|a|$ time steps belong to the decoder. Let $\mathbf{h}^l_t \in \mathrm{R}^n$ denote the hidden vector at time step t and layer l. $\mathbf{h}^l_t$ is then computed by:

$$\mathbf{h}^l_t = \mathrm{LSTM}\left(\mathrm{h}^l_{t-1}, \mathrm{h}^{l-1}_t\right)$$

where LSTM refers to the LSTM function being used. LSTM memory cell is depicted in Fig. 4.2. The architecture described in Zaremba et al. (2015) is fairly popular, For the encoder, $\mathbf{h}^0_t = \mathbf{W}_q\mathbf{e}(x_t)$ is the word vector of the current input token, with $\mathbf{W}_q \in \mathrm{R}^{n \times |Vq|}$ being a parameter matrix, and $\mathbf{e}(\cdot)$ the index of the corresponding token. For the decoder, $\mathbf{h}^0_t = \mathbf{W}_a\mathbf{e}(y_{t-1})$ is the word vector of the previous predicted word, where $\mathbf{W}_a \in \mathrm{R}^{n \times |Va|}$. Notice that the encoder and decoder have different LSTM parameters.

Once the tokens of the input sequence $x_1, \cdots, x_{|q|}$ are encoded into vectors, they are used to initialize the hidden states of the first time step in the decoder. Next, the hidden vector of the topmost LSTM $\mathbf{h}^L_t$ in the decoder is used to predict the t-th output token as:

$$p(y_t|y_{<t}, q) = \mathrm{softmax}\left(\mathbf{W}_o\mathbf{h}^L_t\right)^{\mathrm{T}}\mathbf{e}(y_t) \qquad (4.1)$$

where $\mathbf{W}_o \in \mathrm{R}^{|Va| \times n}$ is a parameter matrix, and $\mathbf{e}(y_t) \in \{0, 1\}^{|Va|}$ a one-hot vector for computing y_t's probability from the predicted distribution.

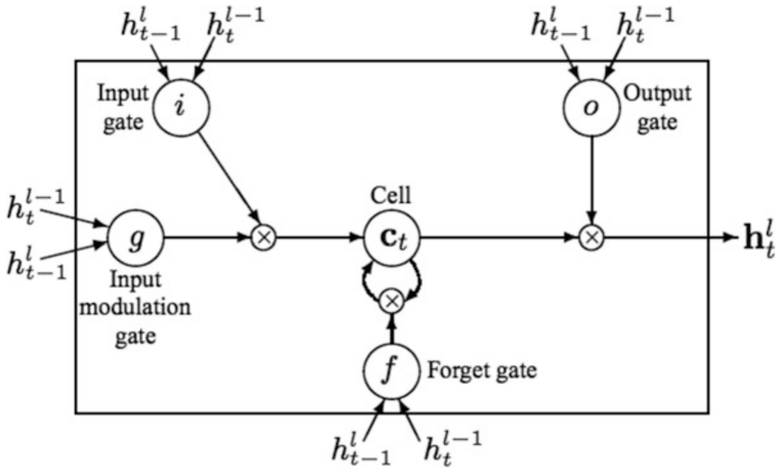


Fig. 4.2 A graphical representation of LSTM memory cells

We augment every sequence with a "start-of-sequence" <*s*> and "end-of-sequence" </*s*> token. The generation process terminates once </*s*> is predicted. The conditional probability of generating the whole sequence $p(a|q)$ is then obtained.

4.2.1.2 Sequence-to-Tree Model

The SEQ2SEQ model has a potential drawback in that it ignores the hierarchical structure of logical forms. As a result, it needs to memorize various pieces of auxiliary information (e.g., bracket pairs) to generate well-formed output. In the following we present a hierarchical tree decoder that better represents the compositional nature of meaning representations. A schematic description of the model is shown in Fig. 4.3.

The present model shares the same encoder with the sequence-to-sequence model described, learning to encode input q as vectors. However, tree decoder is fundamentally different as it generates logical forms in a top-down manner. In order to represent tree structure, a "nonterminal" <*n*> token is defined to indicate a subtree. As shown in Fig. 4.4, the logical form "*lambda $0 e (and (>(account balance $0) 1600:ti) (withdraw from $0 saving:ci))*" is converted into a tree by replacing tokens between pairs of brackets with nonterminals. Special tokens <*s*> and < *(*> denote the beginning of a sequence and nonterminal sequence, respectively. It is not shown in Fig. 4.4. Token </*s*> represents the end of sequence.

After encoding input q, the hierarchical tree decoder uses recurrent neural networks to generate tokens at depth 1 of the subtree corresponding to parts of logical form a. If the predicted token is <*n*>, the sequence is decoded by conditioning on the nonterminal's hidden vector. This process terminates when no more nonterminals

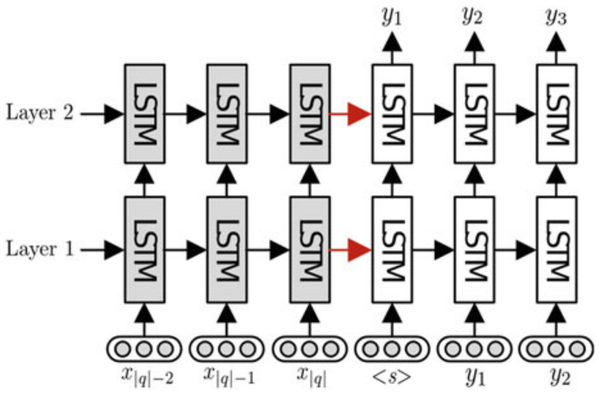


Fig. 4.3 Sequence-to-sequence model with two-layer recurrent neural networks

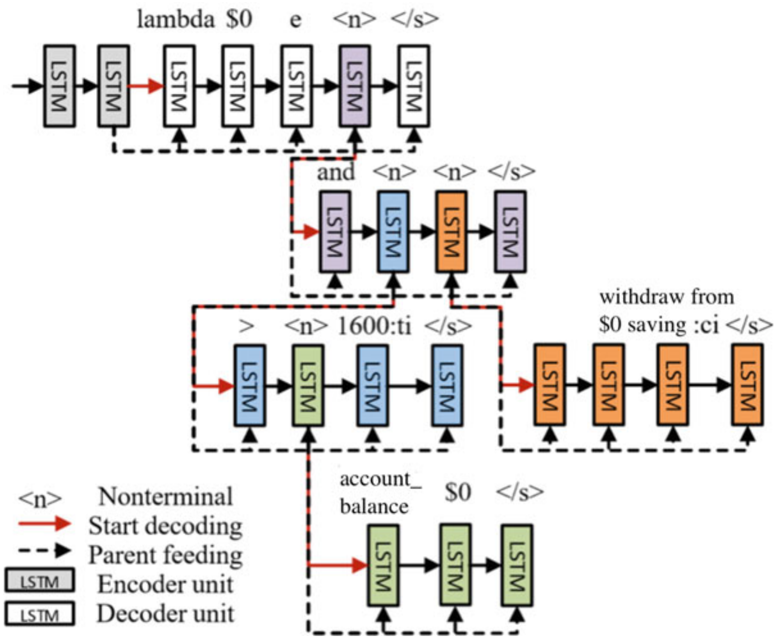


Fig. 4.4 Sequence-to-tree model with a hierarchical tree decoder

are emitted. In other words, a sequence decoder is used to hierarchically generate the tree structure.

In contrast to the sequence decoder described in Sect. 4.2.1.1, the current hidden state does not only depend on its previous time step. In order to better utilize the parent nonterminal's information, we introduce a *parent-feeding* connection where the hidden vector of the parent nonterminal is concatenated with the inputs and fed into LSTM.

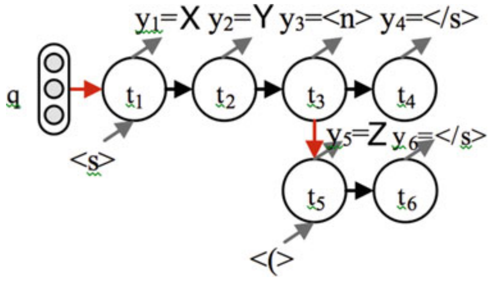


Fig. 4.5 A sequence to tree decoding example for the logical form "*X Y (Z)*"

As an example, Fig. 4.5 shows the decoding tree corresponding to the logical form "*X Y (Z)*", where $y_1 \cdots y_6$ are predicted tokens, and $t_1 \cdots t_6$ denote different time steps. Span "*(C)*" corresponds to a subtree. Decoding in this example has two steps: once input q has been encoded, we first generate $y_1 \cdots y_4$ at depth 1 until token *</s>* is predicted; next, y_5, y_6 sequence is generated by conditioning on nonterminal t_3's hidden vectors. The probability $p(a|q)$ is the product of these two sequence decoding steps:

$$p(a|q) = p(y_1 y_2 y_3 y_4|q) p(y_5 y_6|y_{\leq 3}, q)$$

where Eq. (4.1) is used for the prediction of each output token.

4.2.1.3 Attention Mechanism and Model Training

As shown in Eq. (4.1), the hidden vectors of the input sequence are not directly used in the decoding process. However, it makes intuitively sense to consider relevant information from the input to better predict the current token. Following this idea, various techniques have been proposed to integrate encoder-side information (in the form of a context vector) at each time step of the decoder (Bahdanau et al. 2015).

In order to find relevant encoder-side context for the current hidden state $\mathbf{h}^L{}_t$ of decoder, its attention score is computed with the k-th hidden state in the encoder as:

$$s_k^t = \frac{\exp\left\{\mathbf{h}_k^L \cdot \mathbf{h}_t^L\right\}}{\sum_{j=1}^{|q|} \exp\left\{\mathbf{h}_j^L \cdot \mathbf{h}_t^L\right\}}$$

where $\mathbf{h}_1^L, \cdots, \mathbf{h}_{|q|}^L$ are the top-layer hidden vectors of the encoder (Fig. 4.6). Then, the context vector is the weighted sum of the hidden vectors in the encoder:

$$\mathbf{c}^t = \sum_{k=1}^{|q|} s_k^t \mathbf{h}_k^L$$

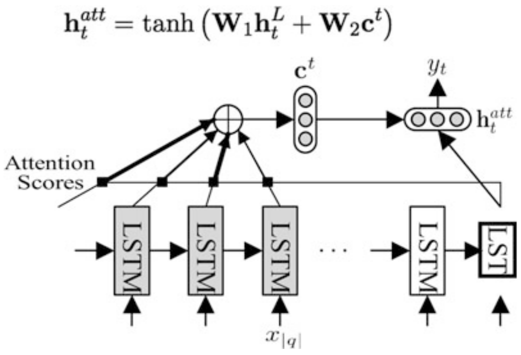

$\mathbf{h}_t^{att} = \tanh\left(\mathbf{W}_1\mathbf{h}_t^L + \mathbf{W}_2\mathbf{c}^t\right)$
$\mathbf{c}^t$
y_t
h_t^{att}
Attention Scores
LSTM
LSTM
LSTM
...
LSTM
LST
$x_{|q|}$

Fig. 4.6 Attention scores are computed by the current hidden vector and all the hidden vectors of encoder. Then, the encoder-side context vector $\mathbf{c}^t$ is obtained in the form of a weighted sum, which is further used to predict y_t

Employing (4.1), this context vector is further used. It acts as a summary of the encoder to compute the probability of generating y_t as:

$\mathbf{h}_t^{att} = \tanh\left(\mathbf{W}_1\mathbf{h}_t^L + \mathbf{W}_2\mathbf{c}^t\right)$

$p(y_t|y_{<t}, q) = \mathrm{softmax}\left(\mathbf{W}_o\mathbf{h}_t^{att}\right)^{\mathrm{T}}\mathbf{e}(y_t)$

where $\mathbf{W}o \in \mathrm{R}|Va| \times n$ and $\mathbf{W}1, \mathbf{W}2 \in \mathrm{R}n{\times}n$ are three parameter matrices, and $\mathbf{e}(y_t)$ is a one-hot vector used to obtain the probability of y_t.

The goal here is to maximize the likelihood of the generated logical forms given NL utterances as input. So the objective function is:

$$\text{minimize} - \sum_{(q,a)\in\mathcal{D}} \log p(a|q)$$

where D is the set of all natural language-logical form training pairs, and $p(a|q)$ is computed as shown in Eq. (4.1).

Dropout operators are used between different LSTM layers and for the hidden layers before the softmax classifiers. This technique can substantially reduce overfitting, especially on datasets of small size. The dropout operator should be applied to the non-recurrent connections (Fig. 4.7). The dashed arrows indicate connections where dropout is applied, and the solid lines indicate connections where dropout is not applied.

The dropout operator corrupts the information carried by the cells, forcing them to perform their intermediate computations more robustly. At the same time, we do not want to erase all the information from the units. It is especially important that the units remember events that occurred many timesteps in the past. An information can flow from an event that occurred at time step t − 2 to the prediction in timestep t + 2 in our implementation of dropout. This information is distorted by the dropout operator L + 1 times, and this number is independent of the number of time steps traversed by the information. Standard dropout perturbs the recurrent connections,

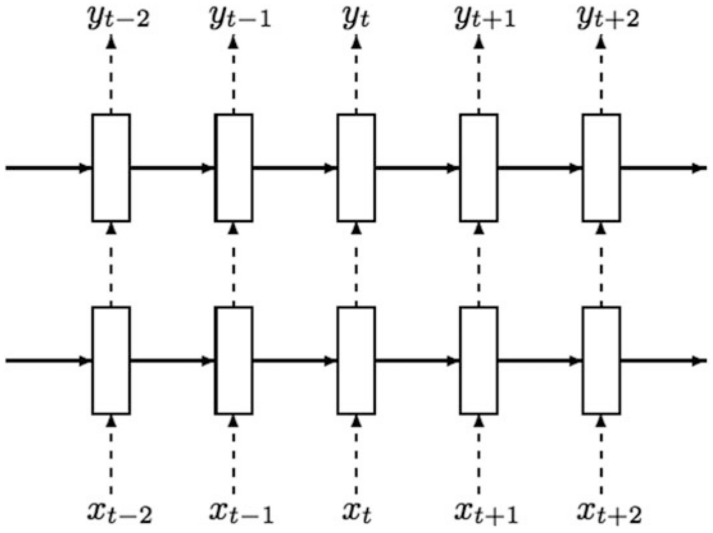


Fig. 4.7 Regularized multilayer RNN

which makes it difficult for the LSTM to learn to store information for long periods of time. By not using dropout on the recurrent connections, the LSTM can benefit from dropout regularization without sacrificing its valuable memorization ability.

The logical forms are predicted for an input utterance q by:

$$a\hat{} = \arg_{a'} \max p\left(a'|q\right)$$

where a' represents a candidate output. However, it is impractical to iterate over all possible results to obtain the optimal prediction. According to Eq. (4.1), we decompose the probability $p(a|q)$ so that we can use greedy search (or beam search) to generate tokens one by one.

Decoding algorithm takes a NL statement and produces a logic form. It includes the following steps:

- *Push the encoding result to a queue*
- *Decode until no more nonterminals*
  - *Call sequence decoder*
  - *Push new nonterminals to the queue*
- *Convert decoding tree to output sequence*

4.2.2 Limitations of Neural Network Based Approaches

Having presented the LSTM approach, we enumerate its limitations:

1. Lack of explainability and interpretability;
2. Necessity to obtain huge dataset;
3. Unable to perform incremental development;
4. Cannot compartmentalize a problem and solve each case separately;
5. Hard to reuse: need to be re-trained even for insignificant update in training set;
6. Hard to make discoveries in data, find correlations and causal links;
7. Hard to integrate with other types of decisions;
8. Computational complexity;
9. Requires a special platform;
10. Does not allow solving a problem once and forever. For example, it takes a single person significant mental efforts to build an NL2SQL. But once it is done, minimum efforts would be required. On the contrary, LSTM – based NL2SQL developed for one domains (such as banana-related queries) would require a totally different training set for queries in another (apple) domains, since nobody "explained" to the system what are words for SQL and what are domain specific word (as it is done for a rule-based system).

4.3 Advancing the State-of-the-Art of NL2SQL

4.3.1 Building NL2SQL via Multiword Mapping

We first address the problem that some words in an NL query may correspond to values, attributes and relations at the same time, so some constraint optimization needs to be applied to obtain a correct mapping. This mapping is a partial case to what is usually referred to as semantic parsing (Kate et al. 2005; Liang and Potts 2015). For some queries, this correct mapping is unique; Popescu et al. (2003) call them *semantically tractable* queries.

Many questions in natural language specify a set of attribute/value pairs as well as 'independently' standing values whose attributes are implicit (unknown).

A db-multiword is a set of word stems that matches a database element. For instance, multiword {*require, advance, rental*} and {*need, advance, rent, request*} match the database attribute *film.advance_rental_request*. Each db-multiword has a set of possible types (e.g. value multiword, attribute multiword) corresponding to the types of the database elements it matches. A syntactic marker (such as "this") is an element of a fixed set of database - independent multiwords that is used indirectly and whose semantic role to the interpretation of a question is limited. For a NL query to be mapped into SQL, we require that some set of db-multiwords exists such that every word in the query appears in exactly *one* db-multiword. We refer to any such db-multiword set as a complete db-multiword representation of query.

In order for a query to be interpreted in the context of the given database (without a need for clarification), at least one complete db-multiword representation must map to some set of database elements E as follows:

1. each db-multiword matches a unique database element in E;
2. each db-multiword for an attribute corresponds to a unique value word. This means that
   (a) the database attribute matching the attribute multiword and the database value matching the value word are compatible; and
   (b) the db-multiword for an attribute and the value token can be mapped into each other.
3. each db-multiword for relation corresponds to either an attribute multiword or a value multiword. This means that
   (a) the database relation matching the relation multiword and the database element matching the attribute or value multiword are compatible; and
   (b) the db-multiword for relation is mapped to the corresponding attribute or value token.

Otherwise, if these conditions do not hold, NL2SQL system needs to act in the chatbot mode.

Popescu et al. (2003) present an implementation of NL2SQL for what they call tractable NL queries, and prove the completeness and coverage statements.

The Tokenizer removes syntactic markers and produces a single db-multiword of this question: (*what, Java, process, Unix, system*, Fig. 4.8. By looking up the tokens in the lexicon (which also contains synonym information), the system retrieves the set of matching database elements for every word. In this case, *what, Java* and *Unix* are db-multiwords for values, system is an attribute token and process is a relation

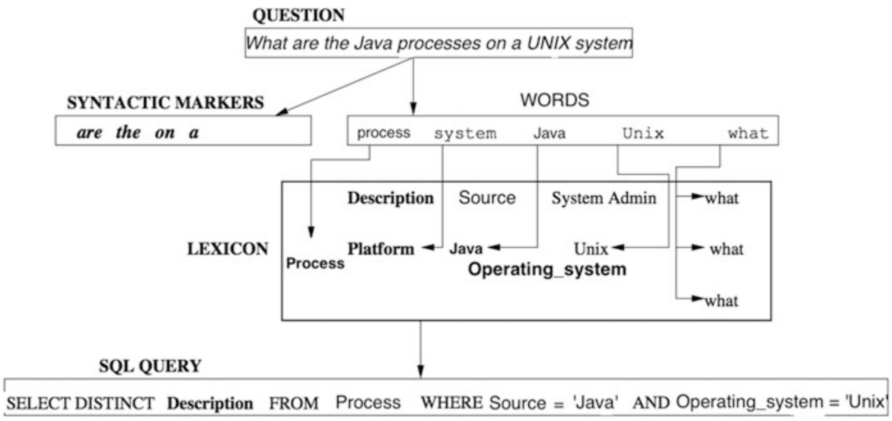

QUESTION
What are the Java processes on a UNIX system
SYNTACTIC MARKERS
are the on a
WORDS
process system Java Unix what
LEXICON
Description
Source
System Admin
what
Platform
Java
Unix
what
Process
Operating_system
what
SQL QUERY
SELECT DISTINCT Description FROM Process WHERE Source = 'Java' AND Operating_system = 'Unix'

Fig. 4.8 The transformation of the query '*What are the Java processes on a Unix system?*' to an SQL query, in the context of a database containing a single relation, process, with attributes *Description*, *Source* and *Operating_system*

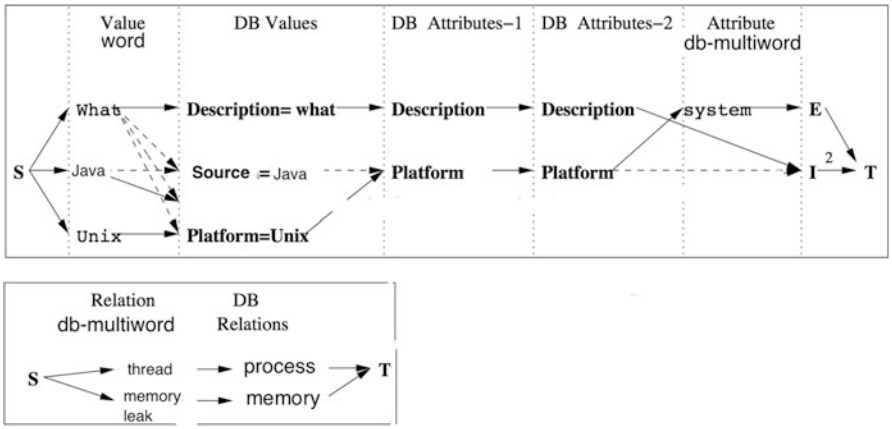


Fig. 4.9 The attribute-value graph for the query '*What are the Java processes on a Unix system?*' (on the top) and the relation graph for the query '*What are the Java processes on a Unix system with memory leaks?*'

word (see Fig. 4.1). The problem of finding a mapping from a complete tokenization of the query to a set of database elements such that the semantic constraints imposed by conditions (1–3) above are satisfied is reduced to a graph matching problem (Fig. 4.9, Galitsky et al. 2010).

After the Tokenizer builds the individual mappings into db-multiwords, the Matcher builds the attribute-value graph (Fig. 4.8). The leftmost column in this figure is a source node. The *Value word* column contains db-multiwords matching database values, which are found in the third column from the right. Some db-multiwords can be ambiguous as they match multiple attributes: for example, '*mem*' can be a value of attribute *description* and also a value of attribute *memory*. The edges go from each value word to each matching database value. The Matcher also connects each database value with its corresponding attribute which is then connected to its matching attribute words and also the node *I* for implicit attributes (*E* denote explicit attributes in the rightmost column). Hence the Matcher reduces NL interpretation problem to a graph (maximum-bipartite-matching) problem with the constraints demanding that all db-multiwords nodes for attributes and values need to occur in this match.

Finally, we present the chart for simple NL2SQL architecture (Fig. 4.10).

The limitations of this NL2SQL approach with the focus on resolving multiword mapping ambiguities are as follows:

- It does not provide a machinery to form individual clause, including an operation between a variable and a value
- It is not easy to integrate graph matching with thesaurus browsing
- it does not help establish assertions between the clauses, such as conjunction, disjunction or sub-query.

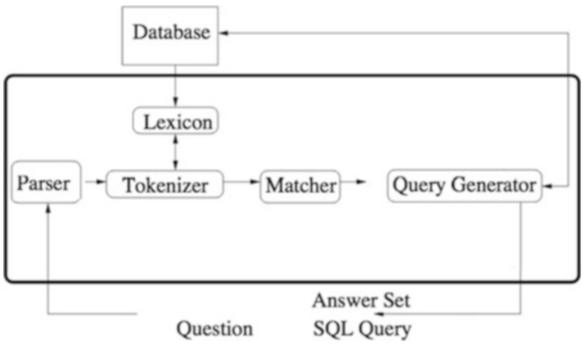


Fig. 4.10 Basic NL2SQL architecture

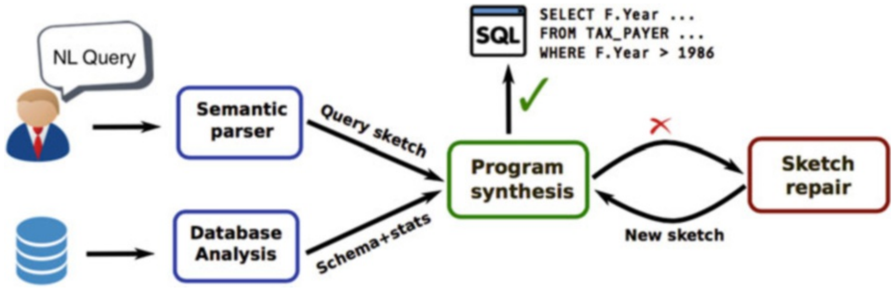


Fig. 4.11 Sketch repair – based approach to building better query representation

4.3.2 *Sketch-Based Approach*

Yaghmazadeh et al. (2017) also use semantic parsing to translate the user's English description into what they call a query sketch (skeleton). Since a query sketch only specifies the shape instead of the full SQL query content, the semantic parser does not need to know about the names of relations, attributes and values (database tables/ columns). Hence, the use of query allows a semantic parser to effectively translate the English description into a suitable formal representation without requiring any database-specific training (Fig. 4.11).

Once a query skeleton is generated, Yaghmazadeh et al. (2017) employ type-directed program synthesis to complete the sketch. NL2SQL system forms well-typed completions of the query skeleton with the aid of the underlying database schema. Since there are typically many well-typed terms, this approach assigns a confidence score to each possible completion of the sketch. The synthesis algorithm uses both the contents of the database as well as natural language hints embedded in the sketch when assigning confidence scores to SQL queries.

For the query *'Find the number of users who rented Titanic in 2016'* the semantic parser returns the sketch

SELECT count(?[users]) FROM??[film] WHERE? = *“Titanic 2016”.*

Here, ‘??’ represents an unknown table, and ? represent unknown columns. Where present, the words written in square brackets represent so-called “hints” for the corresponding placeholder.

Starting from the above sketch the system enumerates all well-typed completions of this sketch, together with a score for each completion candidate. In this case, there are many possible well-typed completions of this sketch; however, none of the those meet the confidence threshold. For instance, one of the reasons for it is that there is no entry called “*Titanic 2016*” in any of the database tables. We need to perform a fault localization to identify the root cause of not meeting confidence threshold. In this case, we determine that the likely root cause is the predicate? = *“Titanic 2016”* since there is no database entry matching *“Titanic 2016”* (The Cameron’s movie was done in 1997, and 2016 is a rental date, not movie creation date). The system repairs the sketch by splitting the *where* clause into two separate conjuncts:

SELECT count(?[users]) FROM??[film] WHERE? = “Titanic” AND ? = 2016”.

On the next step, the system tries to complete the refined sketch S but it again fails to find a high-confidence completion of the above representation. In this case, the problem is that there is no single database table that contains both the entry “*Titanic*” as well as the entry “2016”. We try to repair it by introducing a join. As a result, the new sketch now becomes:

SELECT count(users.id) FROM users JOIN? ON ? =? *[film] WHERE*? = *“Titanic” AND* ? = *“2016”.*

Finally, we arrive at the resultant query representation:

SELECT count(users.id) FROM users JOIN film ON users.rental_film_id = film.id

WHERE film.title = “Titanic” AND users.rental_date = “2016”.

4.3.3 Extended Relational Algebra to Handle Aggregation and Nested Query

To map aggregation operation references form NL to SQL, we need a special version of a relational algebra (Fig. 4.12, Yaghmazadeh et al. 2017). Here *c* are column names; *f* denotes an aggregate function, and *v* denotes a value. Relations, denoted as

Fig. 4.12 A version of relational algebra oriented towards representing database queries in NL

$$
\begin{array}{rcl}
T & := & \Pi_L(T) \mid \sigma_\phi(T) \mid T \,{}_c\!\bowtie_c T \mid t \\
L & := & L, L \mid c \mid f(c) \mid g(f(c), c) \\
E & := & T \mid c \mid v \\
\phi & := & \phi \ lop \ \phi \mid \neg\phi \mid c \ op \ E \\
op & := & \leq \mid < \mid = \mid > \mid \geq \\
lop & := & \wedge \mid \vee
\end{array}
$$

T in the grammar, include tables *t* stored in the database or views obtained by applying the following relational algebra operators:

1. projection (Π). Projection ΠL(T) takes a relation T and a column list L and returns a new relation that only contains the columns in L.
2. selection (σ). The selection operation σφ(T) yields a new relation that only contains rows satisfying φ in T.
3. join (◃▹)..The join operation $T_1\ {}_{c1}\bowtie_{c2}\ T_2$ composes two relations T_1, T_2 such that the result contains exactly those rows of $T_1 \times T_2$ satisfying $c_1 = c_2$, where c_1, c_2 are columns in T_1, T_2 respectively.

We assume that every column in the database has a unique name. Note that we can easily enforce this restriction in practice by appending the table name to each column name. Second, we only consider equi-joins because they are the most commonly used join operator, and it is easy to extend our techniques to other kinds of join operators (e.g., θ-join). Notice that the relational algebra allows nested queries. For instance, selections can occur within other selections and joins as well as inside predicates φ.

Unlike standard relational algebra, the relational algebra variant shown in Fig. 4.12 also allows aggregate functions as well as a group-by operator. For conciseness, aggregate functions

$f \in AggrFunc = \{max, min, avg, sum, count\}$ are specified as a subscript in the projection operation. In particular, *Πf(c)(T)* yields a single aggregate value obtained by applying f to column c of relation *T*. Similarly, group-by operations are also specified as a subscript in the projection operator. Specifically, *Πg(f(c_1), c_2)(T)* divides rows of T into groups g_1 based on values stored in column c_2 and, for each g_1, it yields the aggregate value $f(c_1)$.

The logical forms used for NL2SQL take the form of query skeletons, which are produced according to the grammar from Fig. 4.12. Intuitively, a query skeleton is a relational algebra term with missing table and column names. Query skeletons as the underlying logical form representation are used because it is extremely hard to accurately map NL queries to full SQL queries without training on a specific database. In other words, the use of query skeletons allows us to map English sentences to logical forms in a database-agnostic manner.

In Fig. 4.13 '*?h*' represents an unknown column with hint *h*, which is just a natural language description of the unknown. Similarly, *??h* represents an unknown table name with corresponding hint *h*. If there is no hint associated with a hole, we simply write *?* for columns and *??* for tables.

Fig. 4.13 Sketch Grammar for NL2SQL

$$\begin{array}{rcl}
\chi & := & \Pi_{\kappa}(\chi) \mid \sigma_{\psi}(\chi) \mid \chi_{?h}\bowtie_{?h}\chi \mid ??h \\
\kappa & := & \kappa, \kappa \mid ?h \mid f(?h) \mid g(f(?h), ?h) \\
\eta & := & \chi \mid ?h \mid v \\
\psi & := & \psi\ lop\ \psi \mid \neg\psi \mid ?h\ op\ \eta \\
op & := & \leq \mid < \mid = \mid > \mid \geq \\
lop & := & \wedge \mid \vee
\end{array}$$

id	first_name	num of films	film_id_fk
1	John	60	101
2	Jack	80	102
3	Jane	80	103
4	Mike	90	104
5	Peter	100	103
6	Alice	100	104
7	Julie	100	103

film_id	film_name	category_id
101	Name1	1001
102	Name2	1002
103	Name3	1001
104	Name4	1002

Fig. 4.14 *Customers* and *Films* tables to demonstrate aggregation

Fig. 4.15 Aggregated data

Film_category	Avg(num_of_films)
1001	85
1002	90

Given a query sketch generated by the semantic parser, this sketch needs to be completed by instantiating the placeholders with concrete table and column names defined in the database schema. The sketch completion procedure is type- directed and treats each database table as a record type.

$\{(c_1: \beta_1),\ldots(c_n: \beta_n)\}$,

where c_i is a column name and β_i is the type of the values stored in column c_i. The sketch completion algorithm need to select the best completion based on scoring, which takes into account semantic similarity between the hints in sketch and the names of tables and columns.

Let us consider the tables *Customers* and *films* tables from Fig. 4.14. Here,

$\Pi_{avg(num\_of\_films)}$ (Customers) = 87, and $\Pi_{g(avg(num\_of\_films),\ category\_id)}$(Customers $_{film\_id\_fk} \bowtie _{film\_id\_fk}$ films) gives the average number of films watched by customers, who currently rent the films of a given category (Fig. 4.15).

To provide an example of nested queries, suppose that a user wants to retrieve all film renting customers with the highest number of watched movies. We can express this query as:

$\Pi name\ (\sigma num\_of\_movies = \Pi_{max(num\_of\_movies)\ (customers)}\ (customers))$

For the tables from Fig. 4.14, this query yields a table with two rows, #5 and #6.

A limitation of this algebra-based approach is that a fairly complicated rule system is required; most sophisticated rules would cover rather infrequent cases. Even after a thorough coverage of various cases of mapping between words and table/column names, ambiguity still arises in a number of situations.

4.3.4 Interpreting NL Query via Parse Tree Transformation

Li and Jagadish (2016) proposed a way to correctly interpret complex natural language queries through a carefully limited interaction with the user. Their approach is inspired by how humans query each other, attempting to acquire certain knowledge. When humans communicate with one another in NL, the query-response cycle is not as rigid as in a traditional database system (Galitsky and Botros 2012). If a human formulates a query that the addressee does not understand, he will come back requesting clarification. The query author may do so by asking specific questions back, so that the question-asker understands the point of potential confusion. He may also do so by stating explicitly how she interpreted the query. Drawing inspiration from this natural human behavior, Li and Jagadish (2016) design the query mechanism to facilitate collaboration between the system and the user in processing NL queries. First, the system explains how it interprets a query, from each ambiguous word/phrase to the meaning of the whole sentence. These explanations enable the user to verify the answer and to be aware where the system misinterprets her query. Second, for each ambiguous part, multiple likely interpretations are given to the user to choose from. Since it is often easier for users to recognize an expression rather than to compose it, this query mechanism is capable of achieving satisfactory reliability without giving the user too much routine tasks.

We follow along the lines of this study and make clarification systematic; clarification request can be issued by a number of NL2SQL system components and layers. In our approach a data source can be SQL or noSQL database, unstructured data such as text and Q/A pairs, and transactional data such as a set of API calls.

4.3.4.1 Intermediate Representation Language

Due to the difficulties of directly translating a sentence into a general database query languages using a syntax - based approach, the intermediate representation systems were proposed. The idea is to map a sentence into a logical query language first, and then further translate this logical query language into a general database query language, such as SQL. In the process there can be more than one intermediate meaning representation language. A baseline architecture based on parse tree transformation is presented in Fig. 4.16.

Using predicate logic as the logical query language, an intermediate representation system could develop a semantic interpreter that maps the above sentence into the following logical query:

‘*Return users who watched more movies than Bob on Documentary after 2007’:*

countBob = count [rent(Bob , movie(movie_name, duration, rating, category, . . .), rental_date), rental_date>2007]

```
for(User user: users){
if count[user]>count[Bob]
}
```

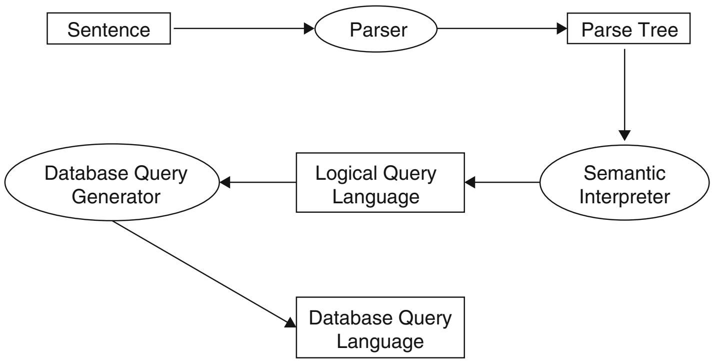


Fig. 4.16 A baseline architecture based on parse tree transformation for SQL interpretation pipeline

4.3.4.2 Mapping the Nodes of Query Parse Tree

A linguistic mapping approach to NL2SQL would be to classify each parse tree node as SQL command, reference to a table, field or value. Such approach identifies the nodes in the linguistic parse tree that can be mapped to SQL components and tokenizes them into different tokens. In the mapping process, some nodes may fail in mapping to any SQL component. In this case, our system generates a warning to the user, telling her that these nodes do not directly contribute in interpreting her query. Also, some nodes may have multiple mappings, which causes ambiguities in interpreting these nodes. For each such node, the parse tree node mapper outputs the best mapping to the parse tree structure adjustor by default and reports all candidate mappings to the interactive communicator.

Parse Tree Structure Adjustor After the node mapping (possibly with interactive communications with the user), we assume that each node is understood by our system. The next step is to correctly understand the tree structure from the database's perspective. However, this is not easy since the linguistic parse tree might be incorrect, out of the semantic coverage of our system or ambiguous from the database's perspective. In those cases, Li and Jagadish (2014) adjust the structure of the linguistic parse tree and generate candidate interpretations (query trees) for it. In particular, the structure of the parse tree is adjusted in two steps. In the first step, the nodes are reformulated in the parse tree to make it similar in structure to one of the stored parse trees. If there are multiple candidate valid parse trees for the query, the system chooses the best tree as default input for the second step and report top k of them to the interactive communicator. In the second step, the chosen or default parse tree is semantically processed and new tree nodes are inserted to make it more semantically plausible. After inserting these implicit nodes, the system obtains the exact tree interpretation for the query.

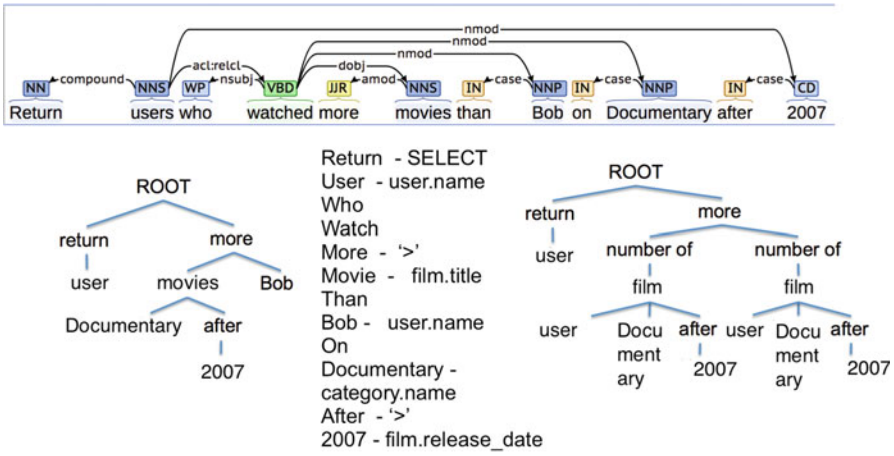


Fig. 4.17 Transformation steps for the query

Interactive Communicator In case the system possibly "misunderstands" the user, the interactive communicator explains how her query is processed, visualizing the semantically plausible tree. Interactive communications are organized in three steps, which verify the intermediate results in the parse tree node mapping, parse tree structure reformulation, and implicit node insertion, respectively. For each ambiguous part, a multiple choice selection panel is generated, in which each choice corresponds to a different interpretation. Each time a user changes a choice, the system immediately reprocesses all the ambiguities in later steps.

In Fig. 4.17 we show transformation steps for the query '*Return users who watched more movies than Bob on Documentary after 2007'*. In the first step, a parse tree T is obtained by Stanford NLP (on the top). In the second step, each query word is mapped into a database operator, field or value.

In the third step, the parse tree adjustor reformulates the structure of the parse tree T and generates a set of candidate parse trees. The interactive communicator explains each candidate parse trees for the user to choose from. For example, one candidate is explained as '*return the users whose movies on Documentary after 2007 is more than Bob's*.' In the fourth step, this candidate tree is fully instantiated in the parse tree structure adjustor by inserting implicit nodes (shown in the bottom-right of Fig. 4.17). The resultant selected query tree is explained to the user as '*return the users, where the number of films in Documentary released after 2007 is more the number of films rented by Bob in Documentary released after 2007*'.

The overall architecture with Clarification Requester is shown in Fig. 4.18. The system includes the query interpretation part, interactive communicator and query tree translator. The query interpretation part, which includes parse tree node mapper and structure adjustor, is responsible for interpreting an NL query and representing the interpretation as a query tree. The interactive communicator is responsible for communicating with the user to ensure that the interpretation process is correct. The

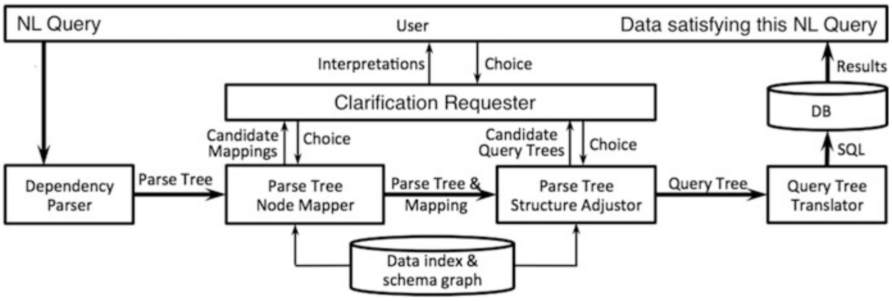


Fig. 4.18 Overall architecture of a NL2SQL based on parse tree transformation with Clarification Requester

query tree, possibly verified by the user, is translated into a SQL statement in the query tree translator and then evaluated against a DB.

4.4 Designing NL2SQL Based on Recursive Clause Building, Employing Thesauri and Implementing Via Chatbot

4.4.1 Selecting Deterministic Chatbot-Based Approach

An extensive corpus of work in NL2SQL showed that it is rather difficult to convert all user queries into SQL mostly due to ambiguity of database field names and a complex structure of practical database, in addition to query understanding difficulty. Also, it is hard for NL2SQL problem to communicate with the user which NL queries are acceptable and which are not. Even if 80% of user NL queries are properly translated to SQL, which is hard to achieve, the usability is questionable.

To address this problem, we propose to implement NL2SQL as a chatbot, so that the system can clarify every encountered ambiguity with the user right away. If a confidence score for a given NL2SQL component is low, the chatbot asks the user to confirm/clarify whether the interpretation of a query focus or a given clause is correct. For example, interpreting a phrase *movie actor name,* the chatbot requests user clarification if *name* refers to the actor last name, first name or film title.

The main highlights of the selected approach are as follows:

1. We extract a linguistic representation for a SQL clause in the form of *table. column* assignment;
2. We build the sequence of SQL clauses in the iterative way;
3. We rely on thesauri, possibly web mining (Chap. 8) and other cues to build a mapping from NL representation for a clause into table and column name;
4. We resolve all kinds of ambiguities in NL query interpretation as a clarification request via chatbot.

4.4.2 Interpreting Table.Field Clause

The input of *Table.Field* clause recognizer is a phrase that includes a reference to a table and/or its field. Each word may refer to a field of one table and a name of another table, only to a field, or only to a table, hence the query understanding problem is associated with rather high ambiguity.

The fundamental problem in NL2SQL is that interpreting NL is hard in general and understanding which words refer to which database field is ambiguous in nature (Galitsky 2003). People may use slang words, technical terms, and dialect-specific phrasing, none of which may be known to the NL2SQL system. Regretfully, even with appropriate choice of words, NL is inherently ambiguous. Even in human-to-human interaction, there are miscommunications.

One of the difficulties is substituting values for attributes of similar semantic types, such as *first* and *last name*. For example, it is hard to build the following mapping unless we know what first and last names are:

actor name John Doe ⇒ *actor.first_name* = … & *actor.last_name* = …

There is a need for transformations beyond mapping *phrase2table.field*, such as a lookup of English first names and knowledge that first and last name can be in a single field, can be in various formats and orders, or belong to distinct fields, like in the case of Sakila database (Oracle 2018).

When a user is saying '*film name*' the system can interpret it as a table with field = '*name*' when *film.name* does not exist. Although 'name' is a synonym of 'title', the phrase 'name' can be mapped into totally foreign table such as category.name instead of *actor.first_name*. If a phrase includes '*word1 word2*' it is usually ambiguous since *word2* can be *table 1.field* and also *table2.word2* can be a field (or a part of a field, as a single word) in another table. Hence we need a hypothesis management system that proceeds from most likely to least likely cases, but is deterministic so that the rule system can be extended.

We start with the rule that identify a single table name and make sure there are no other table names mentioned (Fig. 4.19). Also, we need to confirm that no field name is mentioned in the string to be mapped into a table name. Once a table is confirmed, we select its default field such as 'title' or any other field with the name of entity represented by this table.

If a pure table rule is not applicable, we proceed to the *table + its field* rule. The system identifies a table and its field together. We iterate through all table-field words and select the table-filed combination when a highest number of words are matched against the phrase. If we do not find a good match for table-filed set of keywords against the phrase, we proceed to matching a field only (the third step). At this step we use ontology so expand a list of keywords for a field with synonyms. Once we find a match for a field, we get a list of table this field can possibly belong to.

In the second step, we try to find words in the phrase correlated with this table name. In this step, for each of these tables we in turn obtain a list of their fields and

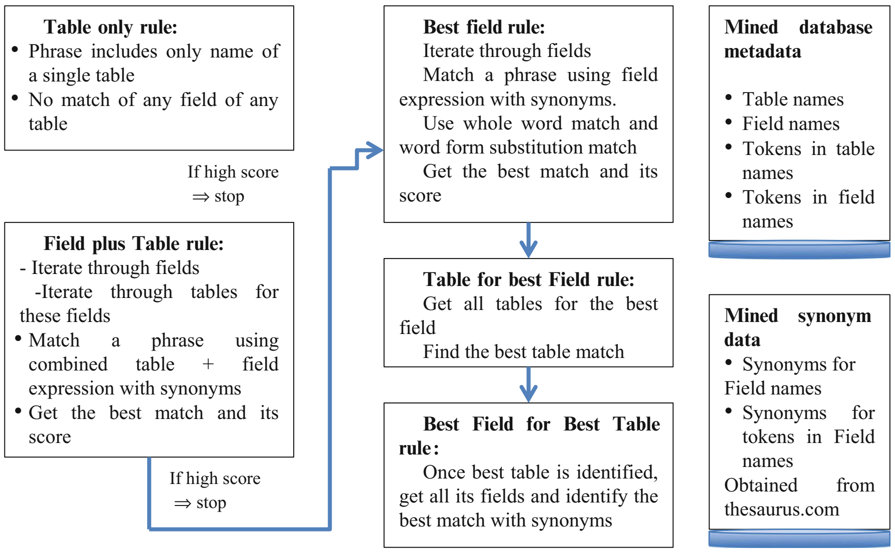


Table only rule:
• Phrase includes only name of a single table
• No match of any field of any table
If high score ⇒ stop
Field plus Table rule:
- Iterate through fields
-Iterate through tables for these fields
• Match a phrase using combined table + field expression with synonyms
• Get the best match and its score
If high score ⇒ stop
Best field rule:
Iterate through fields
Match a phrase using field expression with synonyms.
Use whole word match and word form substitution match
Get the best match and its score
Table for best Field rule:
Get all tables for the best field
Find the best table match
Best Field for Best Table rule:
Once best table is identified, get all its fields and identify the best match with synonyms
Mined database metadata
• Table names
• Field names
• Tokens in table names
• Tokens in field names
Mined synonym data
• Synonyms for Field names
• Synonyms for tokens in Field names
Obtained from thesaurus.com

Fig. 4.19 *Phrase2Table.Field* Unit

verify that the original field from the step one of this unit is identified, not another one. If the second step fails we stop on the first one, and if the verification of the third step fails, we stop on the second step. The higher is the number of steps, the higher is the confidence level.

4.4.3 Collecting Information on a Database and Thesaurus for NL2SQL

The NL2SQL system is designed to automatically adjust to an arbitrary database where table and column names are meaningful and interpretable. The following data structures are used by *Phrase2Table.Field* and other algorithms

- Set *fieldsForMatching*: A set of fields;
- Map *tablesFieldsForMatching* gives a list of fields for a table;
- Map *fieldsTableListsForMatching* gives a list of tables for a field;
- Map *fieldsTablesForMatching* gives a selected table for a field. For some fields such as entity name, there is just a single table for this entity.

Since terms in a user query can deviate from field names in a database, it is necessary to mine for synonyms offline from sources like thesaurus.com or use trained synonym models such as word2vec (Mikolov et al. 2015). Lists of synonyms or similarity function are then used in *phrase2table.field* component of Query understanding pipeline. The arrow in the right-middle shows communication with the *Phrase2Table.Field* Unit of Fig. 4.19.

4.4.4 Iterative Clause Formation

Once a focus clause is identified, we consider the remaining of the NL query as a *Where* clause (Figs. 4.21 and 4.22, Galitsky et al. 2013a, b). It is hard to determine boundaries of clauses; instead, we try to identify the assignment/comparison word (anchor) such as *is, equals, more, before, as,* which indicates the center of a phrase to be converted into SQL clause. Once we find the leftmost anchor we attempt to build the left side (attribute) and the right side (value).

To find the boundary of an attribute, we iterate towards the beginning the NL query to the start of the current phrase. It is usually indicated by the prepositions *with* or *of*, connective *and*, or a Wh-word. The value part is noun and/or a number, possibly with an adjective. The same words mark the end of value part as the beginning of next attribute part.

Once the clause is built, we subject the remaining part of the NL query to the same clause identification algorithm, which starts with finding the anchor. If a structure of phrase follows Fig. 4.21, it is processed by the middle-left component *Clause builder from phrase* in the Fig. 4.20 chart. Otherwise, it there is no anchor word and it is hard to establish where the phrases for attribute and values are, we apply *the Clause builder by matching the phrase with indexed row* approach.

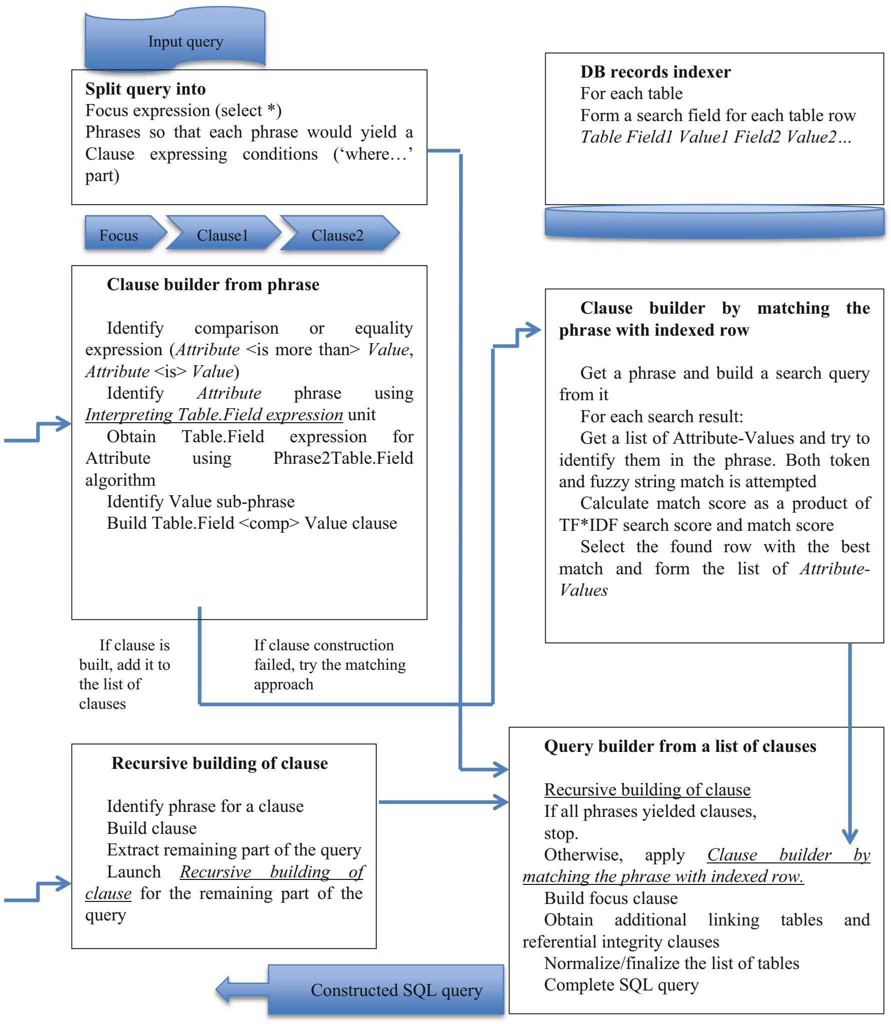
4.4.5 Clause Building by Matching the Phrase with Indexed Row

We also refer to this alternative approach to building SQL query clauses as NL2SQL via search engineering: it involves building a special index (not to confuse with database own index) and executing a search of a part of user NL query against it. At indexing time, we index each row of each table in the following format (top-right of Fig. 4.20):

Table field1 value1 field2 value2 ...

Associative tables and other ones which do not contain data for entities such as customer of address are not indexed for matching. The index includes the fields for search and for storing the values.

Once an NL query is obtained and *Clause builder from phrase* failed to build a clause from a phrase expected to contain a *Where* clause, the search expression is built from this phrase. This search expression includes the words which are expected



Input query

Split query into
Focus expression (select *)
Phrases so that each phrase would yield a Clause expressing conditions ('where…' part)

Focus → Clause1 → Clause2

DB records indexer
For each table
Form a search field for each table row
Table Field1 Value1 Field2 Value2…

Clause builder from phrase

Identify comparison or equality expression (*Attribute* <is more than> *Value*, *Attribute* <is> *Value*)
Identify *Attribute* phrase using *Interpreting Table.Field expression* unit
Obtain Table.Field expression for Attribute using Phrase2Table.Field algorithm
Identify Value sub-phrase
Build Table.Field <comp> Value clause

Clause builder by matching the phrase with indexed row

Get a phrase and build a search query from it
For each search result:
Get a list of Attribute-Values and try to identify them in the phrase. Both token and fuzzy string match is attempted
Calculate match score as a product of TF*IDF search score and match score
Select the found row with the best match and form the list of *Attribute-Values*

If clause is built, add it to the list of clauses

If clause construction failed, try the matching approach

Recursive building of clause

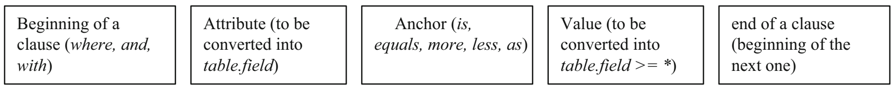
Identify phrase for a clause
Build clause
Extract remaining part of the query
Launch *Recursive building of clause* for the remaining part of the query

Query builder from a list of clauses

Recursive building of clause
If all phrases yielded clauses, stop.
Otherwise, apply *Clause builder by matching the phrase with indexed row.*
Build focus clause
Obtain additional linking tables and referential integrity clauses
Normalize/finalize the list of tables
Complete SQL query

Constructed SQL query

Fig. 4.20 A high-level view of NL2SQL system. Integration of *Phrase2Table.Field* and *Recursive building of clause* units is shown by step-arrows on the left



Beginning of a clause (*where, and, with*)
Attribute (to be converted into *table.field*)
Anchor (*is, equals, more, less, as*)
Value (to be converted into *table.field >= **)
end of a clause (beginning of the next one)

Fig. 4.21 A structure of a clause to be converted into *table.field [=/< / > / like] value*

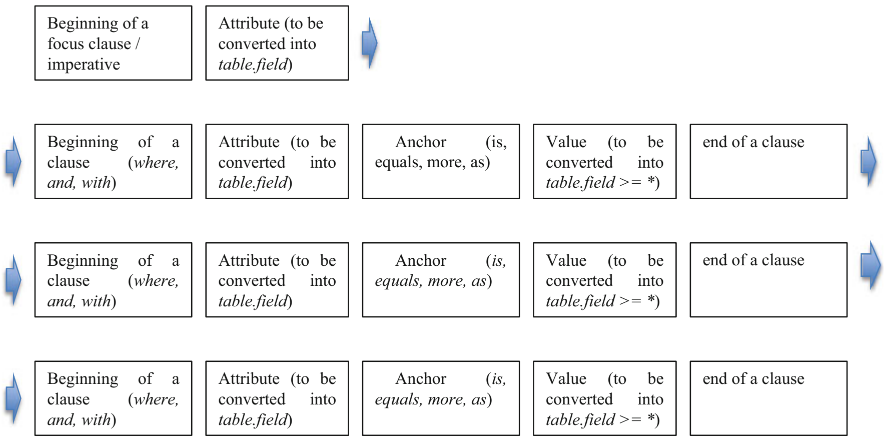

Beginning of a focus clause / imperative
Attribute (to be converted into *table.field*)
Beginning of a clause (*where, and, with*)
Attribute (to be converted into *table.field*)
Anchor (is, equals, more, as)
Value (to be converted into *table.field* >= *)
end of a clause
Beginning of a clause (*where, and, with*)
Attribute (to be converted into *table.field*)
Anchor (*is, equals, more, as*)
Value (to be converted into *table.field* >= *)
end of a clause
Beginning of a clause (*where, and, with*)
Attribute (to be converted into *table.field*)
Anchor (*is, equals, more, as*)
Value (to be converted into *table.field* >= *)
end of a clause

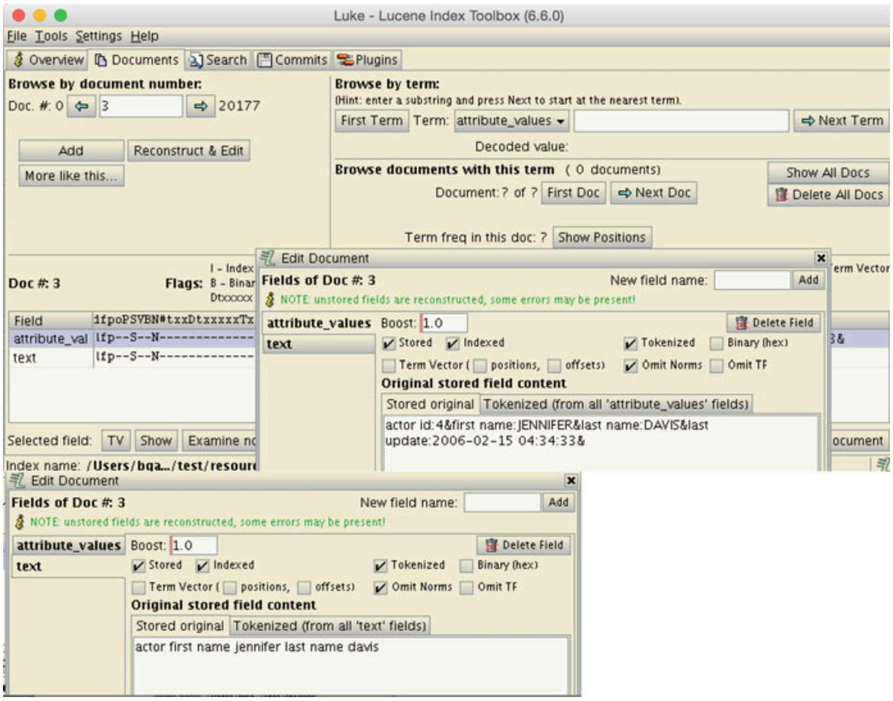
Fig. 4.22 User query as a sequence of clauses: some of them follow the template on the top and are processed by *Clause builder from phrase*, and some of them do not and are handled by *Clause builder by matching the phrase with indexed row*

to match tables, fields and values. Since we do not know what is what before search results are obtained, the query is formed as conjunction of words first and then as a disjunction of these words, if the conjunction query fails to give results. In a disjunction queries, not all keywords have to be matched: some of them are just used by the NL query author but do not exist in table data. To make search more precise, we also form span-AND queries from entities identified in the phrase to be converted, such as '*Bank of America'*.

Numbers need a special treatment. For a query of *equal* kind, finding an exact number would make SQL query formation precise and in fact substitutes an execution of a resultant query. Since all numbers are indexed for search as string tokens, real numbers need to be stored and searched with '.' substituted to avoid splitting string representation into two parts.

Once search results are obtained, we iterate through them to find the most likely record. Although the default TF*IDF relevance is usually right, we compute out own score based on the number of attribute-value pairs which occur in both the query and a candidate search result (Fig. 4.23). Our own score also takes into account individual values without attribute occurrence in both the query and the record. String-level similarity and multiword deviations between occurrences in the query and the record are also taken into account (whether some words in a multiword are missing or occur in a different form (such as plural for a noun or a tense for a verb).

Depending on the type of string for the value (numeric or string), we chose the operation '=' or 'like' when the *table.field* < *assignment* > *value* clause is built. Obviously, when this clause building method is employed we do not need to call the *phrase2Table.Field* component.



Browse by document number:
Doc. #: 0 3 20177

Browse by term:
(Hint: enter a substring and press Next to start at the nearest term).
Term: attribute_values
Decoded value:
Browse documents with this term (0 documents)
Document: ? of ?
Term freq in this doc: ?

Doc #: 3
Flags: I - Index B - Binar Dtxxxxx

Field	ɪfpoPSVBN#txxDtxxxxxTx
attribute_val	ɪfp--S--N------------
text	ɪfp--S--N------------

Selected field:
Index name: /Users/bqa.../test/resour

Fields of Doc #: 3 New field name:
NOTE: unstored fields are reconstructed, some errors may be present!
attribute_values Boost: 1.0
text
[✓] Stored [✓] Indexed [✓] Tokenized [ ] Binary (hex)
[ ] Term Vector ([ ] positions, [ ] offsets) [✓] Omit Norms [ ] Omit TF
Original stored field content
Stored original | Tokenized (from all 'attribute_values' fields)
actor id:4&first name:JENNIFER&last name:DAVIS&last update:2006-02-15 04:34:33&

Fields of Doc #: 3 New field name:
NOTE: unstored fields are reconstructed, some errors may be present!
attribute_values Boost: 1.0
text
[✓] Stored [✓] Indexed [✓] Tokenized [ ] Binary (hex)
[ ] Term Vector ([ ] positions, [ ] offsets) [✓] Omit Norms [ ] Omit TF
Original stored field content
Stored original | Tokenized (from all 'text' fields)
actor first name jennifer last name davis

Fig. 4.23 A view of the index for *matching the phrase with indexed row approach.* Each database record is stored for search (bottom-right) and data retrieval (in the middle)

4.4.6 Extracting Focus Clause

We refer to text which is converted into ‘select *’ statement as focus clause. We start with *Wh* word and then extract the phrase that follows it. This phrase must be the shortest one among those, which follow the *Wh* word. Noun, verb, prepositional and other kinds of phrases are acceptable. From this phrase, a clause will be built applying *phrase2table.field* component. This clause will not have an assignment but will possibly have a grouping term instead, such as ‘*give me the maximum temperature of water* . . . ‘.

4.5 Resolving Ambiguities in Query Interpretation via Chatbot

We have presented a rule-based architecture for query interpretation. Naturally, in many processing components, ambiguities arise, such as table name, field name or relationship. Instead of trying to find a most plausible representation of an NL query,

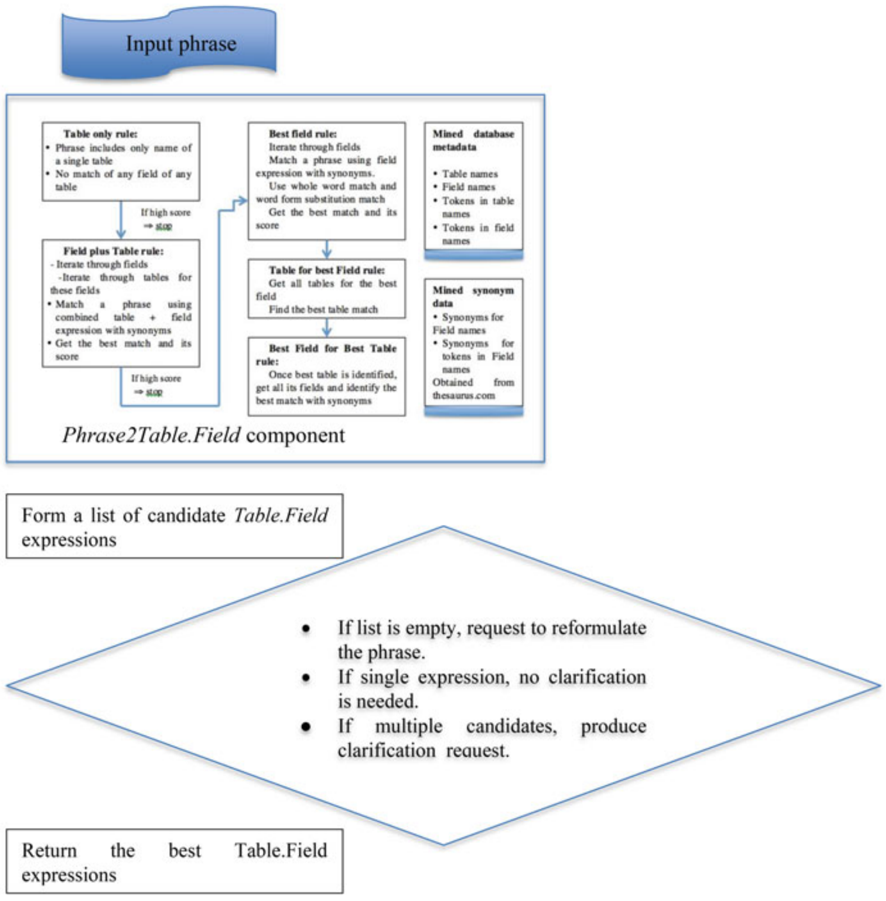

Input phrase
Table only rule:
• Phrase includes only name of a single table
• No match of any field of any table
If high score ⇒ stop
Field plus Table rule:
- Iterate through fields
-Iterate through tables for these fields
• Match a phrase using combined table + field expression with synonyms
• Get the best match and its score
If high score ⇒ stop
Best field rule:
Iterate through fields
Match a phrase using field expression with synonyms.
Use whole word match and word form substitution match
Get the best match and its score
Table for best Field rule:
Get all tables for the best field
Find the best table match
Best Field for Best Table rule:
Once best table is identified, get all its fields and identify the best match with synonyms
Mined database metadata
• Table names
• Field names
• Tokens in table names
• Tokens in field names
Mined synonym data
• Synonyms for Field names
• Synonyms for tokens in Field names
Obtained from thesaurus.com
Phrase2Table.Field component
Form a list of candidate *Table.Field* expressions
• If list is empty, request to reformulate the phrase.
• If single expression, no clarification is needed.
• If multiple candidates, produce clarification request.
Return the best Table.Field expressions

Fig. 4.24 Disambiguation of *Phrase2Table.Field* (Fig. 4.19) results via chatbot interaction

like most of NL2SQL systems do, we rely on the user to resolve ambiguities via clarification requests. Our NL2SQL system gives the user the query phrase being mapped into a table name, and enumerates possible tables, providing a clarification request.

A chart for a chatbot wrapper for *Phrase2Table.Field* component is shown in Fig. 4.24. When a single *Table.Field* is obtained, no disambiguation is necessary. To disambiguate a phrase, the wrapper asks the user which mapping is correct. For example, if a user is asking … *'when guys name is ..'* the system identify the token *name* and obtains a number of *Table.Field* candidates. Then the *Phrase2Table.Field* wrapper offers *actor.first_name / actor.last_name /staff.first_name / staff.last_name* | *customer.first_name / customer.last_name* options. Once the user selects the correct mapping option, it is set by the *Phrase2Table.Field* component.

4.6 A Sample Database Enabled with NL2SQL

To build an industrial-strength NL2SQL, we select a default database Sakila (Oracle 2018) that demonstrates a variety of MySQL capabilities. It is intended to provide a standard schema that can be used for examples in tutorials and samples, and also serves to highlight the latest features of MySQL such as Views, Stored Procedures, and Triggers. The Sakila sample database was designed as a replacement to the *world* sample database, which provides a set of tables containing information on the countries and cities of the world and is useful for basic queries, but lacks structures for testing MySQL-specific functionality and new features found in MySQL 5.

Notice that for NL2SQL we selected a fairly feature-rich database with a complicated structure of relations (Fig. 4.25). The database structure is much more complex than the ones used in academic studies to evaluate NL2SQL in Sects. 4.2, and 4.3.

These are the examples of NL query, logs for intermediate step, resultant SQL representation and query results:

Query: ‘*what is staff first name when her movie return date is after 2005-06-02 01:02:05*’
looking for table.field for ‘[staff, first, name]’
found table.field = staff.first_name
looking for table.field for ‘[movie, return, date]’
found table.field = rental.return_date
Results: Mike

```
SQL: select staff.first_name from staff, rental where
rental.return_date > ‘2005-06-02 01:02:05’ and rental.
staff_id = staff.staff_id
```

Query: ‘*what film title has actor’s first name as Christian and category Documentary*’
looking for table.field for ‘[film, title]’
found table.field = film.title
looking for table.field for ‘[actor, first, name]’
found table.field = actor.first_name
Results: ACADEMY DINOSAUR |
CUPBOARD SINNERS |
MOD SECRETARY |
PRINCESS GIANT |

```
SQL: select film.title from film_category, film_actor,
film, actor, category where actor.first_name like ‘%Chris-
tian%’ and category.name = ‘documentary’ and film_actor.
film_id = film.film_id and film_actor.actor_id = actor.
```

(continued)

`actor_id and film_actor.film_id = film.film_id and film_actor.actor_id = actor.actor_id and film_category.film_id = film.film_id and film_category.category_id = category.category_id and film_category.film_id = film.film_id and film_category.category_id = category.category_id.`

Query: *'What is actor fist name when movie category is Documentary and special features are Behind the Scenes'*
looking for table.field for '[actor]'
found table.field = actor.first_name
looking for table.field for '[movie, category]'
found by table ONLY = category.name
Results: PENELOPE |
CHRISTIAN |
LUCILLE |
SANDRA |
`SQL:` select actor.first_name from film_category, film_actor, film, actor, category where category.name like '%Documentary%' and film.special_features like '%behind%the%scenes%' and film_actor.film_id = film.film_id and film_actor.actor_id = actor.actor_id and film_actor.film_id = film.film_id and film_actor.actor_id = actor.actor_id and film_category.film_id = film.film_id and film_category.category_id = category.category_id and film_category.film_id = film.film_id and film_category.category_id = category.category_id.

Query: *'What is a film category when film title is Ace Goldfinger'*
looking for table.field for '[film, category]'
found table.field = category.name
looking for table.field for '[film, title]'
found table.field = film.title
Results:
Horror |
SQL: select category.name from film_category, film, category where film.title like '%ACE GOLDFINGER%' and film_category.film_id = film.film_id and film_category.category_id = category.category_id

Notice that we do not require the user to highlight the parameter values versus parameter names.

For the last, fourth example, the query could have been formulated as *'What is a category of film…'* but it would make it harder for NL2SQL system to determine the fields of the tables referred to by the words *category* and *film*.

In many cases, when a reference to a table name is not mentioned in an NL query, we attempt to identify it based on a column name. If multiple tables have this

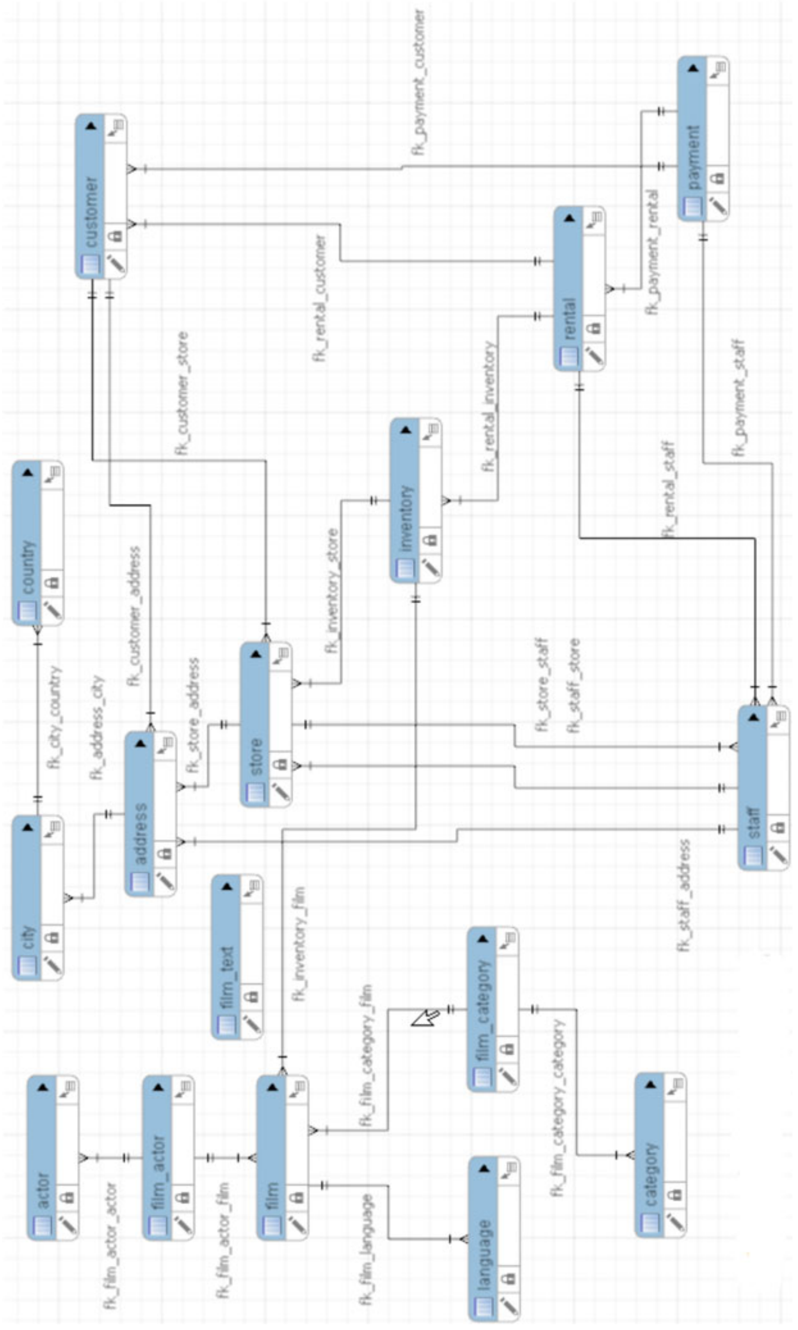


Fig. 4.25 The structure of Sakila database as visualized by MySQL Workbench

extracted column name, the chatbot mode is initiated and the user needs to pick up a single table from the list of ones with this column name.

4.7 Conclusions

There are a number of issues with usability of NL2SQL systems (Nihalani et al. 2011). When NL2SQL system fails, it is frequently the case that the system does not provide any explanation of what causes the system to fail. Some users may try to rephrase the NL query or just move on to another query. Most of the time, it is up to the users to determine of the causes the errors.

Also, customers may have false expectations, be misled by an NL2SQL system's ability to understand NL. Customers may assume that the system is intelligent and overestimate its results. Instead of asking precise questions in database terms, they may be tempted to ask questions that involve complex ideas, certain judgments, reasoning capabilities, etc., which an NL2SQL system is not designed to properly handle.

Each NL2SQL is limited to its coverage of acceptable NL expressions. Currently, all NL2SQL systems can only handle some subsets of NL and it is not easy to define these subsets. Some NL2SQL systems cannot even answer certain questions which belong to their own subsets. This is not the case in a formal language. The formal language coverage is obvious and any statements that follow the defined syntactic rules are guaranteed to give the corresponding answer.

Despite these limitations, the NL2SQL chatbot that leverages the NL understanding pipeline, interactivity in resolving ambiguities and domain-specific thesauri, is an efficient and effective tool accessing data in a database by a broad audience of users. As amount of data collected by various organization grows, NL2SQL becomes a must big data technology.

We now proceed to surveying general technology trends related to NL2SQL. AI and ML are seeping into virtually everything and represent a major battleground for technology providers over the next 5 years. Also, there is a blending the digital and physical worlds which creates an immersive, digitally enhanced environment. A connections between an expanding set of people and businesses, as well as devices, content and services to deliver digital business outcomes will be exploited. We enumerate the recent technology trends, following (Cearley 2017).

Conversational Platforms They will shift in how humans interact with the digital world. The routine activity of translating intent shifts from user to computer. The platform takes a question or command from the user and then responds by executing some function, presenting some content or asking for additional input. Over the next few years, conversational interfaces will become a primary design goal for user interaction and be delivered in dedicated hardware, core OS features, platforms and applications.

Upgrading Organizational Structure for Advanced Technologies Over the next few years, creating software that learn, adapt and performs independently and autonomously is a major competitive battle ground for vendors. Relying on AI-based decision making, upgraded business models and ecosystems, and enhanced customer experience will yield the payoff for digital technologies over the next decade. As AI and ML technologies are rapidly developing, companies will need to devote significant attention to skills, processes and tools to successfully exploit these technologies. The investment focus areas will include data collection, preparation, cleaning, integration, as well as efforts into the algorithm and training methodologies and model creation. Multiple specialists including data scientists, developers and business process owners will need to form teams together.

Embedding AI into Apps Over next couple of years, some AI features will be embedded into virtually every app, application and service. Some of these apps will be obvious intelligent apps that rely on AI and ML 100%, whereas other apps will be unpretentious users of AI that provide intelligence behind the scenes. a new intelligent intermediary layer will be created between people and systems so that not only new interaction modes will appear but also the nature of work and the structure of the workplace will be transformed.

Intelligent apps augment human activity; they are not simply a way to substitute humans. Also, analytics for augmented reality is a particularly strategic growing area which uses machine learning to automate data preparation, insight discovery and insight sharing for a broad range of business users, operational workers and citizen data scientists. Enterprise resource planning is another area where AI is expected to facilitate the next break-through. Packaged software and service providers need to invent ways to use AI to extend business for advanced analytics, intelligent processes and enhanced user experiences.

Intelligent Things and Everything These are physical things that go beyond the execution of rigid programming models to exploit AI to deliver advanced behaviors and interact more naturally with their surroundings and with people. AI stimulates the development of new intelligent things (including, but not limited to autonomous vehicles, robots and drones) and enables improved capability to many existing things such as Internet of Things connected consumer and industrial systems (Galitsky and Parnis 2019).

The use of autonomous vehicles in constrained, controlled settings is intensively growing area of intelligent things. Autonomous vehicles will likely be employed in a limited, well-defined and controlled roadways over next 5 years, but general use of autonomous cars will likely require a driver to anticipate technology failures. As semi-autonomous scenarios requiring a driver dominate, car producers will test the technology more thoroughly, and meanwhile legal and regulatory issues will be resolved.

The Internet of Everything generalizes computer-to-computer communications for the Internet of Things to a more complex system that also encompasses people, robots and machines. Internet of Everything connects people, data, process and things (Chambers (2014), taking the way we do business, transforming communication, job creation, education and healthcare across the globe to the next level. Over next few years, more than 70% of earth population will be connected with more than 50 billion things. With

Internet of Everything systems people will be better served in education, healthcare and other domains to improve their lives and have better experiences.

For a present overview of IoE, the Internet of things (IoT) is about connecting objects to the network and enabling them to collect and share data" (Munro 2017). As presently conceived, "Humans will often be the integral parts of the IoT system" (Stankovic 2014). Internet of Everything, Internet of battlefields, Internet of the medical arena and other domains will manifest themselves as heterogeneous and potentially self-organizing complex-systems that define human processes, requiring interoperability, just-in-time human interactions, and the orchestration of local-adaptation functionalities in order to achieve human objectives and goals (Suri et al., 2016).

Digital Fingerprints They refer to the digital representation of a real-world entity or system. Digital fingerprints in the IoT context projects are believed to be employed over the next few years; properly designed digital fingerprints of entities have the potential to significantly improve enterprise decision-making. These digital fingerprints are linked to their real-world counterparts and are used to understand the state of an entity, a thing or a system, respond to changes, improve operations and add value. Organizations will implement digital fingerprints simply at first, then evolve them over time, improving their ability to collect and visualize the right data, apply the right analytics and rules, and respond effectively to business objectives. Various professional from civil engineering to healthcare will all benefit from this paradigm of the integrated digital twin world.

References

Agrawal S, Chaudhuri S, Das G (2002) Dbxplorer: a system for keyword-based search over relational databases. In: ICDE, pp 5–16

Androutsopoulos I, Ritchie GD, Thanisch P (1995) Natural language interfaces to databases – an introduction. Nat Lang Eng 1(1):29–81

Bahdanau D, Cho K, Bengio Y (2015) Neural machine translation by jointly learning to align and translate. In: Proceedings of the ICLR, San Diego, California

Berant J, Chou A, Frostig R, Liang P (2013) Semantic parsing on freebase from question-answer pairs. In: EMNLP, pp 1533–1544

Bergamaschi S, Guerra F, Interlandi M, Lado RT, Velegrakis Y (2013) Quest: a keyword search system for relational data based on semantic and machine learning techniques. PVLDB 6 (12):1222–1225

Cearley DW (2017) Assess the potential impact of technology trends. https://www.gartner.com/doc/3823219?ref=AnalystProfile&srcId=1-4554397745

Chambers J (2014). Are you ready for the internet of everything? World Economic Forum, from https://www.weforum.org/agenda/2014/01/are-you-ready-for-the-internet-of-everything/ (15 Jan 2014)

Dong L, Lapata M (2016) Language to logical form with neural attention. ACL

Galitsky B (2003) Natural language question answering system: technique of semantic headers. Advanced Knowledge International, Australia

Galitsky B (2005) Natural language front-end for a database. Encyclopedia of database technologies and applications, p 5. IGI Global Pennsylvania USA

Galitsky B, S Botros (2012) Searching for associated events in log data. US Patent 8,306,967

Galitsky B, M Grudin (2001) System, method, and computer program product for responding to natural language queries. US Patent App. 09/756,722

Galitsky B and Parnis A (2019) Accessing Validity of Argumentation of Agents of the Internet of Everything. In Lawless, W.F., Mittu, R., Sofge, D., and ·Russell, S., Artificial Intelligence for the Internet of Everything. Elsevier, Amsterdam

Galitsky B, D Usikov (2008) Programming Spatial Algorithms in Natural Language. AAAI Workshop Technical Report WS-08-11.–Palo Alto, pp 16–24

Galitsky B, Dobrocsi G, De La Rosa JL, Kuznetsov SO (2010) From generalization of syntactic parse trees to conceptual graphs. In: International conference on conceptual structures, pp 185–190

Galitsky B, De La Rosa JL, Dobrocsi G (2011) Mapping syntactic to semantic generalizations of linguistic parse trees. In: Proceedings of the twenty-fourth international Florida artificial intelligence research society conference

Galitsky B, D Ilvovsky, F Strok, SO Kuznetsov (2013a) Improving text retrieval efficiency with pattern structures on parse thickets. In: Proceedings of FCAIR, pp 6– 21

Galitsky B, Kuznetsov SO, Usikov D (2013b) Parse thicket representation for multi-sentence search. In: International conference on conceptual structures, pp 153–172

Goldberg Y (2015) A primer on neural network models for natural language processing. CoRR, abs/1510.00726

Kate RJ, Wong YW, Mooney RJ (2005) Learning to transform natural to formal languages. In: AAAI, pp 1062–1068

Kupper D, Strobel M, Rosner D (1993) Nauda – a cooperative, natural language interface to relational databases. In: SIGMOD conference, pp 529–533

Li FH, Jagadish V (2014) Nalir: an interactive natural language interface for querying relational databases. In: VLDB, pp 709–712

Li F, Jagadish HV (2016) Understanding natural language queries over relational databases. SIGMOD Record 45:6–13

Li Y, Yang H, Jagadish HV (2005) Nalix: an interactive natural language interface for querying xml. In: SIGMOD conference, pp 900–902

Li Y, Yang H, Jagadish HV (2006) Constructing a generic natural language interface for an XML database. In: EDBT, pp 737–754

Liang P, Potts C (2015) Bringing machine learning and compositional semantics together. Ann Rev Linguis 1(1):355–376

Mikolov T, Chen K, Corrado GS, Dean J (2015). Computing numeric representations of words in a high-dimensional space. US Patent 9,037,464, Google, Inc

Munro K. (2017, 5/23), How to beat security threats to 'internet of things'. From http://www.bbc.com/news/av/technology-39926126/how-to-beat-security-threats-to-internet-of-things

Nihalani MN, Silakari DS, Motwani DM (2011) Natural language Interface for database: a brief review. Int J Comput Sci Issues 8(2)

Popescu A-M, Etzioni O, Kautz HA (2003) Towards a theory of natural language interfaces to databases. In: IUI, pp 149–157

Popescu A-M, Armanasu A, Etzioni O, Ko D, Yates A (2004) Modern natural language interfaces to databases: composing statistical parsing with semantic tractability. In: COLING

Quirk C, Mooney R, Galley M (2015) Language to code: learning semantic parsers for if-this-then-that recipes. In: ACL, pp 878–888

Stankovic JA (2014) Research directions for the internet of things. IEEE Internet Things J 1(1):3–9

Suri N, Tortonesi M, Michaelis J, Budulas P, Benincasa G, Russell S, Winkler R (2016) Analyzing the applicability of internet of things to the battlefield environment. In: Military communications and information systems (ICMCIS), 2016 international conference on. IEEE, pp 1–8

Yaghmazadeh N, Wang Y, Dillig I and Dillig T (2017) SQLizer: Query synthesis from natural language. Proceedings of the ACM on Programming Languages, 1:63:1–63:26

Zaremba W, Sutskever I, Vinyals O (2015) Recurrent neural network regularization. In: Proceedings of the ICLR, San Diego, California

Zhong V, Xiong C, Socher R (2017) Seq2SQL: generating structured queries from natural language using reinforcement learning. https://arxiv.org/pdf/1709.00103.pdf