Chapter 7
Interactivity

As depicted in Figure 2.2 of Chapter 2 earlier, Interaction Generation is a key component of an NLIDB, responsible for its interactivity, a key element of NLIDB impacting its effectiveness and usefulness. In this chapter, we introduce different dimensions of interactivity and discuss the corresponding techniques to support each dimension in details.

According to the Oxford Online Dictionary (2022), the word "Interactivity" has two definitions: one as "the process of two people or things working together and influencing each other" and the other as "the ability of a computer to respond to a user's input." Interactivity in an NLIDB is related to both definitions.

Obviously, the "ability of a computer to respond to a user's input" is crucial in NLIDBs. Interactivity leads to better communication between human and the underlying NLIDB by reducing user input effort, improving the effectiveness of the NLIDB in handling user questions, and increasing trust and understanding of the answer returned. Common interactivity techniques for improving human-machine communication include intelligent input assistant (such as type ahead, auto-complete, and auto-correction), active feedback solicitation (e.g., for disambiguation), and post-hoc result explanation.

Meanwhile, often the goal of an NLIDB user is not just to get the answer of one single specific question but to obtain meaningful, useful and actionable insights from the underlying data to help understand a problem or make decisions. This process of identifying insights from data is known as **exploratory data analysis** [130]. Exploratory data analysis is often a time-consuming process: besides the challenges in comprehending results returned, users often run into "blank canvas" syndrome (where to start) and "what's next" problem (where to explore next). Interactive techniques such as query suggestion, visualization recommendation and conversational interaction enables the machine to play a more active role in exploratory data analysis by "working together and influencing" the users and alleviating aforementioned challenges facing them.

In the rest of the chapter, we will introduce common interactivity techniques in NLIDBs and discuss their benefits and corresponding challenges. We first focus on interactivity at query time: Chapter 7.1 introduces methods for disambiguation including spell correction and interactive disambiguation; and Chapter 7.2 discusses query suggestions, including both dynamic query suggestion (also known as auto-completion) and other types of query suggestions. We then switch focus to interactivity after the initial query: Chapter 7.3 presents automatic data insights with a focus on visual data insights to help the user to better comprehend query results and assist further exploratory data analysis; Chapter 7.4 discusses explanation techniques for different aspects of NLIDBs. Finally, we present how different interactive techniques discussed in the chapter may be combined together to provide conversational NLIDBs: Chapter 7.5 summarizes work related to conversational NLIDBs under a unified discourse framework; and Chapter 7.6 describes multi-modal NLIDBs where user input is beyond natural language.

7.1 Disambiguation

Given a natural language question, an NLIDB may generate more than one interpretation of the question explicitly as the output of Query Understanding (Figure 2.2) or implicitly as part of an end-to-end model. In such cases, the NLIDB needs to decide which interpretation to use. **Disambiguation** refers to this process of distinguishing between possible interpretations and making the interpretation more certain. It is a longstanding research topic in natural language processing [28, 24, 357].

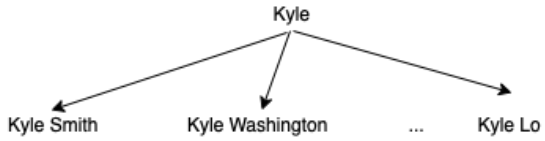


Fig. 7.1 An example of term disambiguation

At a high-level, disambiguation methods fall into two categories: (1) **interactive disambiguation** and (2) **automatic disambiguation**. For interactive disambiguation, explanations for each possible meaning of an ambiguous word/phrase are presented to a user, who then selects one from the list. For instance, given the question "*How many friends does Kyle have?*", as shown in Figure 7.1, the user is asked to select which student "*Kyle*" is referred to from the list of multiple students with the same first name "*Kyle*" but different last names. For automatic disambiguation, the top most candidate is automatically selected from possible candidates ranked based on certain automatically computed scores. For example, each possible student name listed in Figure 7.1 may be assigned a score based on its frequency in user questions, and the most frequently occurred one is selected automatically.

Most NLIDBs in the real-world apply a combination of automatic and interactive disambiguation methods with the goal of minimizing user input effort while maximizing the accuracy of the final results. For instance, for the earlier question, an NLIDB may apply an automatic disambiguation to select the most frequently chosen name (e.g., "*Kyle Smith*") and at the same time allow the user to examine and select alternative interpretations.

7.1.1 Ambiguity

Within the context of NLIDB, we are mainly concerned with two types of ambiguity [141]: (1) **lexical ambiguity**, where a single word/term has more than one interpretation; (2) **structural ambiguity**, also known as **syntactic ambiguity**, where the input question has more than one possible meaning due to the overall structure of the question.

7.1.1.1 Lexical ambiguity

Lexical ambiguity arises in NLIDBs when the context (i.e., the input question itself, the underlying database and any available external knowledge) is insufficient to determine the mapping of a single word/term to a single specific database artifact (also known as **meta data**) or a single data value.

One common cause of lexical ambiguity is **synonyms**, when multiple words or terms express the same or similar meaning. A term specified in the input question may be the synonym or near synonym for multiple

meta data or database values. For instance, "paper" may map into "publication" and "article"; "amount" may map into "quantity" and "volume." Another common cause of lexical ambiguity is **hypernym**, when a word refers to a group of other words. For instance, "people" is a hypernym of "author" and "editor"; "organization" is a hypernym of "company" and "institute." When an input question contain a term corresponding to the hypernym of a meta data or database value, then multiple mappings for the term are possible. In addition, words and term specified in the input question may be under-specified and hence can map into more than one database artifacts. As an example, "elementary school" may map into "*grade*=K", "*grade*=1", ..., "*grade*=5." Furthermore, users may incorrectly spell words in an input question. A misspelled word may map into more than one meta data / data value of the underlying database. For example, the misspelled word "operaton" is equally likely to be mapped into "operator" and "operation." Finally, stemming and lemmatization are often used for pre-processing questions to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form [210]. For instance, both "writers" and "wrote" can be all reduced to their base form "write" via lemmatization. This step, while important in producing meaning mappings, could sometimes lead to ambiguity. For instance, "operate", "operation", "operator", and "operators" all correspond to "oper" based on stemming [253]. As a result, "operators" in an input question may be mapped into both "operator" and "operation", leading to ambiguity.

7.1.1.2 Structural ambiguity

Structural ambiguity is usually the result of poor sentence construction. As a result, a sentence can have two or more meanings. For example, the question "*Who are friends of Kyle in Grade 1?*" has two meanings: one is to ask for the 1st graders who are friends with someone named Kyle; the other asking for the friends of a 1st grader named Kyle. Such ambiguity can result in more than one possible parse tree, as illustrated in Figure 7.2.

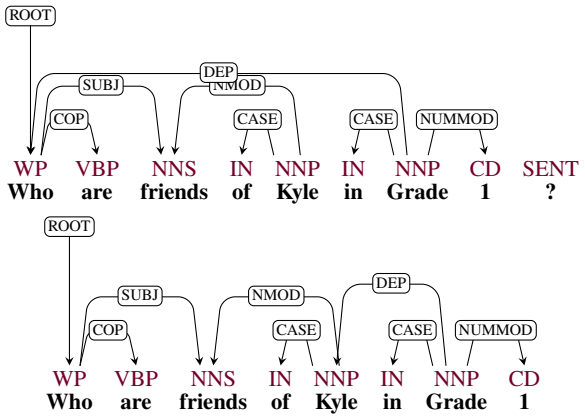


Fig. 7.2: Example Question with Attachment Ambiguity

Two common kinds of structural ambiguity include **attachment ambiguity** and **coordination ambiguity** [161]. Attachment ambiguity arises when a particular constituent can be attached to more than one place in the parse tree. The sentence in Figure 7.2 is an example of attachment ambiguity, where prepositional phrase *in Grade 1* can be attached to two different places. Adverbial phrases are also subject to this kind

of ambiguity. For instance, in question "*Who are friends of Kyle graduating next year*?" the gerundive-VP *graduating next year* can be interpreted to modify *Kyle*, asking about friends of a student named Kyle who is graduating next year, or it can be interpreted to modify *friends*, asking about the soon-to-graduate friends of a student named Kyle.

Coordination ambiguity arises when two phrases are conjoined by a conjunction such as *and*. The first question below has only one possible interpretation and is unambiguous. However, the second question has not only attachment ambiguity as discussed earlier but also coordination ambiguity, where the phrase *Kyle and Kim in the first grade* can be interpreted as *[Kyle and Kim][in the first grade]*, referring to a first-grader named Kyle and another first-grader named Kim, or *[Kyle] and [Kim in the first grade]*, referring to a student named Kyle and a first-grader named Kim.

1. Who are friends of Kyle and Kim?
2. Who are friends of Kyle and Kim in the first grade?

In addition, grammatical errors may also lead to structural ambiguity. For instance, when "*and*" is omitted in "*Find all products made by machines (and) sold in 2020*", the sentence has two meanings: to find all machine-made products sold in 2020 or to find all products that are made by machines that were sold in 2020.

7.1.2 Spell Correction

Misspelling, a major cause for ambiguity introduced earlier, is prevalent in real-world applications. According to an earlier study, over 10% of queries sent to search engines are misspelled [74]. In addition, adversarial misspellings can compromise the effectiveness of neural models [6], which are increasingly popular for NLIDBs. For instance, [225] shows that misspelling a single word in the input from "*film*" to "*films*" changes the prediction of a sentiment analyzer from negative to positive. Spell correction is an important defense against such attacks [255].

Techniques for spell correction falls into two categories: **automatic spell correction** and **interactive spell correction**. Both approaches automatically identify possible misspelled words, generate possible corrections as candidates, and then rank the candidates based on their likelihood to be the right correction, as illustrated in Figure 7.3. Automatic spell correction returns the top ranked candidate as the correction, while interactive spell correction presents the possible candidates for the user to select from.

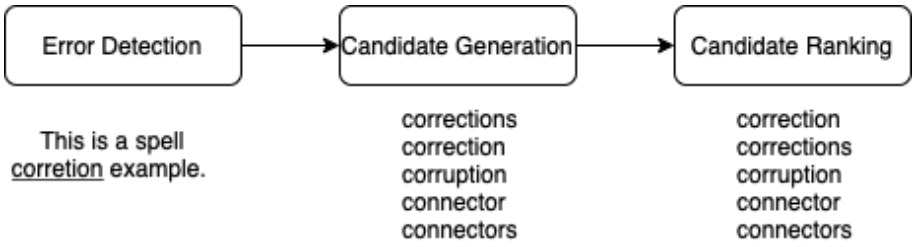


Fig. 7.3: Spell Correction: Overview

Given a dictionary of correct words, we can categorize spelling errors into the following two categories [177]: **nonword errors**, where the incorrect word form is not in the dictionary of correct words, such

as "*correction*" to "*corection*," and **real-word errors**, where the word form of the misspelled word is also in the dictionary of correct words. There are two kinds of real-word errors: **typographical errors** such as "*correction*" to "*corrector*"; and **cognitive errors** such as "*too*" to "*two*."

7.1.2.1 Error Detection

The most essential step of spell correction is **error detection**, the task of identifying any misspelled word from the input text. Detecting nonword errors is relatively straightforward and mainly involves a dictionary lookup, where the dictionary is constructed by taking domain vocabulary into account. Detecting real-word errors requires more sophisticated models that take context into consideration. About 25% to 40% of the errors are real-word errors [177]. Therefore, it is important to handle both nonword and real-word errors.

7.1.2.2 Candidate Generation

This step selects correction candidates for each detected spelling error from a dictionary of correct words .

For nonword errors, the selection is usually based on identifying words that are similar to the identified spelling error. The similarity of two strings, known as **string similarity**, is measured by one or more specific distance functions. Some distance functions such as Damerau–Levenshtein edit distance [79] captures **sequence similarity**: how characters and their orders in the strings are similar to each other. Others such as Soundex [8] and Chinese Soundex [195] capture **phonetic similarity**: how the sounds of the strings are similar to each other.

Two strings are deemed as similar when their similarity, computed based on such distance foundations is above a predetermined threshold. Limiting the threshold can reduce the search space and speed up candidate generation. For instance, [168] restricts the candidate list to words that differ from the misspelled words with just one edit operation, which can be any of insertion, deletion, or replacement of letters. Furthermore, recent work (such as [78]) leverage neural networks to embed similarity functions such as the edit distance into the Euclidean distance for fast approximate similarity search.

For real-word errors, each candidate list includes the original word itself and also candidate words that are similar in spelling or sound.

7.1.2.3 Candidate Ranking

This step ranks the correction candidates based on their probability of being the best correction for each given error. One popular approach is based on Shannon's noisy-channel model [284] (Figure 7.4). The basic intuition is to treat the misspelled word as if it is generated by passing a correctly spelled word through a noisy communication channel. Based on this model, for each misspelled word s, we want to find the correctly spelled word w prior to passing through the noisy channel. In other words, we want to find a word w from Vocabulary V such that it maximizes the probability $P(w|s)$. Using Bayes rule, we have

$$P(w|s) = \frac{P(s|w)P(w)}{P(s)}.$$

The denominator can be dropped since it is the same for all candidate words. Hence we are seeking $\hat{w} = \arg\max_{w \in V} P(s|w)P(w)$. Sometimes the two terms $P(s|w)$ and $P(w)$ are estimated differently and can have different ranges, and one may trust one term more in a specific setting. A weight λ, possibly learned

on a development set, may be added to the formulation.

$$\hat{w} = \arg\max_{w \in V} P(s|w)P(w)^{\lambda}.$$

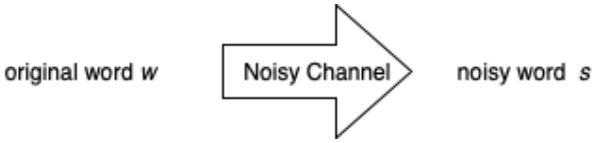


Fig. 7.4 Noise Channel for Spell Correction

Earlier spell correction models rely on patterns and manually-derived features [234]. The most successful recent models are based on neural architectures, particularly transformer-based architecture [31]. These models often leverage data augmentation based on artificial data generation to further boost their performance and overcome their data hungriness. A number of open source off-the-shelf spell correctors are readily available, such as Enchant [311], GNU Aspell [12], JamSpell [241], and NeuSpell [156].

Spell correction is a well-studied topic. For more details on existing work, we refer the readers to [31, 142] for recent work and [177] for earlier work.

7.1.3 Interactive Disambiguation

Spell correction and automatic type ahead can greatly reduce lexicon ambiguity in user input, especially those caused by misspelling. Modern syntactic parsers also can handle structural ambiguity [161]. However, additional ambiguities such as those caused by under-specification may remain. In such cases, soliciting the user input can help resolve the ambiguity.

One popular approach is first resolve the ambiguity automatically, generate a ranked list of possible interpretations, and then solicit user input. This **interactive disambiguation** approach surfaces certain disambiguation decisions automatically made and allow users to correct such decisions interactively.

As illustrated in Figure 7.5, to help a user to understand each interpretation, the system shows possible interpretations of an input question along with their corresponding natural language descriptions and execution results. Then the user can select the one deemed as correct to help the system disambiguate. The user input can also be given at a finer granularity. For instance, in DataTone [111], a user can view data ambiguities and use interactive widgets to correct the decisions on resolving lexical ambiguity (Figure 7.6(c)) and structural ambiguity (Figure 7.6(e)) (e.g., group by decisions). Similarly, Eviza [280], ArkLang [282], and DIY [232] expose system decisions as visual widgets via which the user can override or redefine such decisions.

User correction can also be provided via natural language. As depicted in Figure 7.7, a user can correct the system interpretation of an input question via natural language feedback [99]. As can be seen, for each user input, a natural language interpretation is generated with automatic disambiguation when applicable, along with the execution results. Based on the explanation and execution results, the user provides feedback in natural language, which is then used to correct the initial parse. [15] proposes a similar approach for query reformulation, where each input question is decomposed and rephrased into natural language fragments, each mapping to a portion of a generated SQL query; then the user can view the system interpretation and modify an input question by editing its corresponding natural language fragments.

Most existing interactive disambiguation solutions are tightly coupled with the underlying NLIDBs and hence not easily adaptable for different use cases. However, a parser-independent interactive approach for

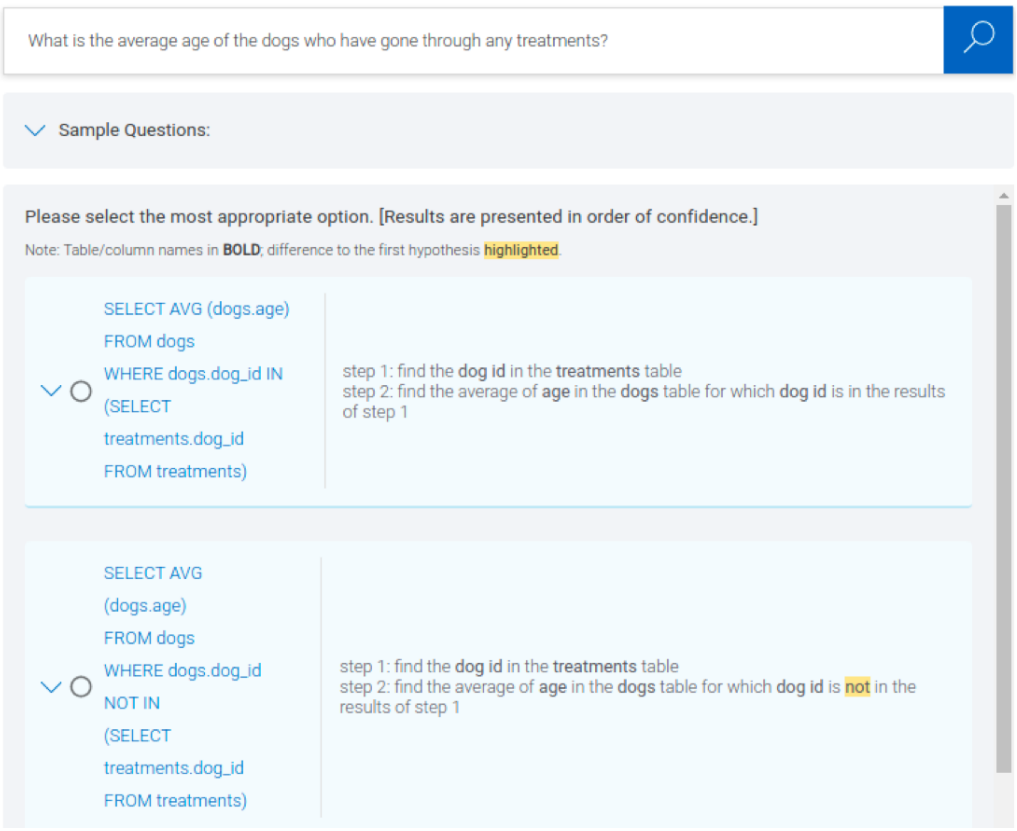


Fig. 7.5: UI displays possible interpretations for an input question; a user can then select the correct one based on each SQL statement and its natural language description (image source: [351])

disambiguation is also possible [197]. This approach assumes the underlying parsers are black boxes and can be deployed over different NLIDBs. As illustrated in Figure 7.8, this approach consists of three steps: (1) **Error Locator** employs an alignment method to identify uncertain tokens in the input questions; (2) **Question Generator** generates multi-choice questions in natural language; and (3) **NL Modifier** rewrites the input questions based on the users' choice. Question Generator often uses techniques on data to text as described earlier in Chapter 6.

An alternative approach to interactive disambiguation is **guided query reformulation** by providing feedback and guiding users to reformulate the input question into one that is unambiguous. For instance, as shown in Figure 7.9, NaLIX [199] provides feedback messages dynamically generated from errors identified for each input question along with suggestions on possible ways to revise the question. For ambiguities identified, NaLIX issues warnings to the user for their awareness of possible quality issues, and explicitly request the user's help to resolve the ambiguity if any. Such a system only returns results when it is fully confident about its handling of the input question. With the interactive feedback and guided query reformulation, the users learn to ask questions understandable by the underlying system overtime.

The interactive disambiguation approach allows users to help the system disambiguate with minimal effort. However, the user input is constrained by the decisions already made by the underlying NLIDB

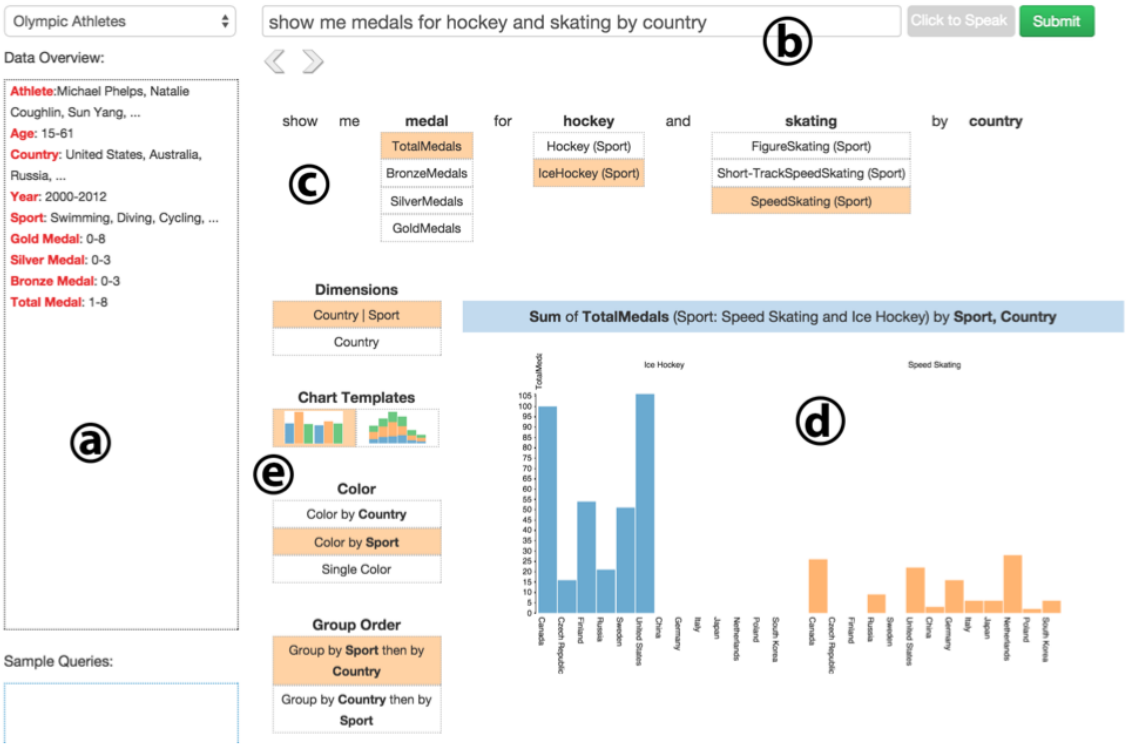


Fig. 7.6: DataTone UI with user ambiguity widgets to correct the system decisions on three data ambiguities: *medals*, *hockey*, and *skating* (image source: [111])

system and often limited to those for lexical ambiguities. In contrast, the guided query reformation approach gives users the full agency in help disambiguation but often requires more users efforts. For example, if none of the interpretations in Figure 7.5 is correct or the system fails to generate any interpretation at all, then the user needs to reformulate their input question without help from the system. On the other hand, the user needs to reformulate the input question in systems such as the ones illustrated in Figure 7.9 whenever there is ambiguity, which could incur extra iteration and additional cognitive load on users.

Systems such as Templar [16] leverage information embedded in query logs to improve disambiguation by acquiring domain knowledge. In a similar fashion, the history of interactive disambiguation not only can help disambiguate the current input question, but also can be leveraged to acquire new knowledge and better handle ambiguity encountered in the future, as demonstrated by DaNaLIX [196] and [99].

7.2 Query Suggestion

Questions submitted to an NLIDB can be formulated directly by users or selected by them from automatically generated queries suggested by the underlying system, also known as **query suggestions**. Query suggestion is an important mechanism to assist exploratory data analysis [338]. This functionality can help a user to

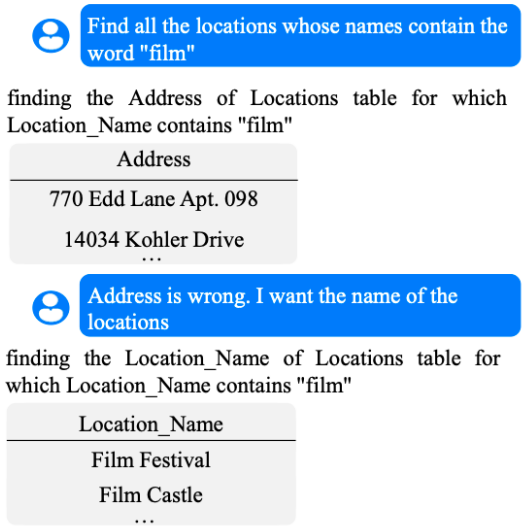


Fig. 7.7 User corrects system decisions via natural language (image source: [99])

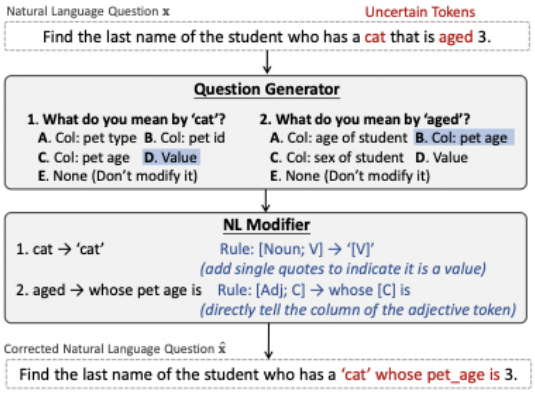


Fig. 7.8 Question Generator and NL Modifier: an example (image source: [197])

formulate a desired input question with less effort, reformulate the input question into a more effective one, or find alternative directions for exploration. Suggestions are typically provided according to the current user input question and users past behaviour.

7.2.1 Auto-completion

Query Auto-Completion (**QAC**), also known as **type-ahead** and **dynamic query suggestion**, can be viewed as a special form of query suggestion. Possible ways of completing a query are suggested while a user is formulating the query [36, 176]. QAC is widely used in modern search engines to improve user experience by reducing user efforts to enter a query, avoiding spelling errors, discovering relevant search terms, and

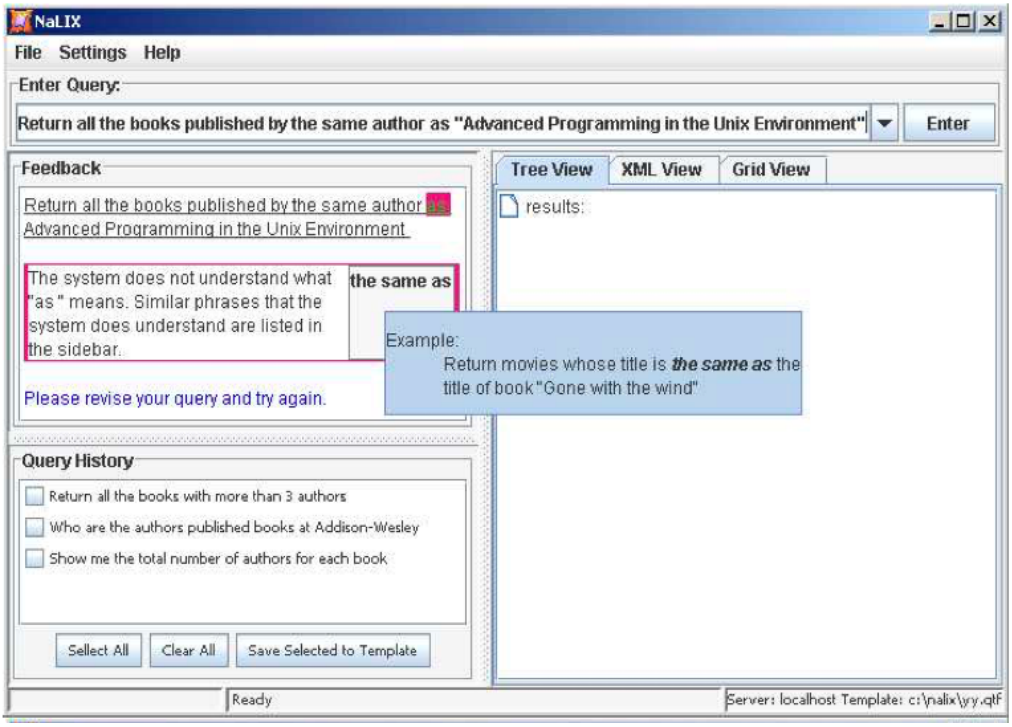


Fig. 7.9: NaLIX UI with feedback to guide user to reformulate the input question (image source: [199])

formulating a better query [36]. It is also commonly supported in commercial NLIDBs, as discussed earlier in Chapter 1.2.

The basic idea of QAC is the following: given a partially described input query (also referred to as **prefix**), identify a list of possible query completions, rank them based on a predefined criterion, and return some to the user. The aim is to return the user's intended query at the top position of a list of query completions.

As depicted in Figure 7.10, a basic QAC framework consists of two phases: (1) an offline phase that generates query completions for each specific prefix in advance based on query log and stores such associations between the prefix and its query completions in an efficient data structure, such as trie-based prefix-trees [145]; (2) a runtime phase that uses this data structure to efficiently look ups possible query completions by prefix matching and then re-ranks the query completions based on signals at query time, such as time, location and user behavior.

As shown in Figure 7.11, the lists of query completions change in real-time dynamically as the user types each character in the input query. In deployed web search engines, the number of completions returned for each query typically has a small fixed limit, e.g., ranging from 5 for Yandex [358], 8 for Bing [217], to 10 for Google [122], Baidu [14], and Yahoo [356].

The runtime phase of QAC mainly consists of three steps:

1. **Candidate retrieval**: retrieve possible query completions based on an input query;
2. **Basic ranking**: rank candidate query completions based on matching potential query completions with user query input;

3. **Re-ranking**: reorder possible query completions based on additional factors—such as query popularity, user information, and so on.

The goal of both basic ranking and re-ranking is to surface to the top the query completion that is most likely to be the user intended query.

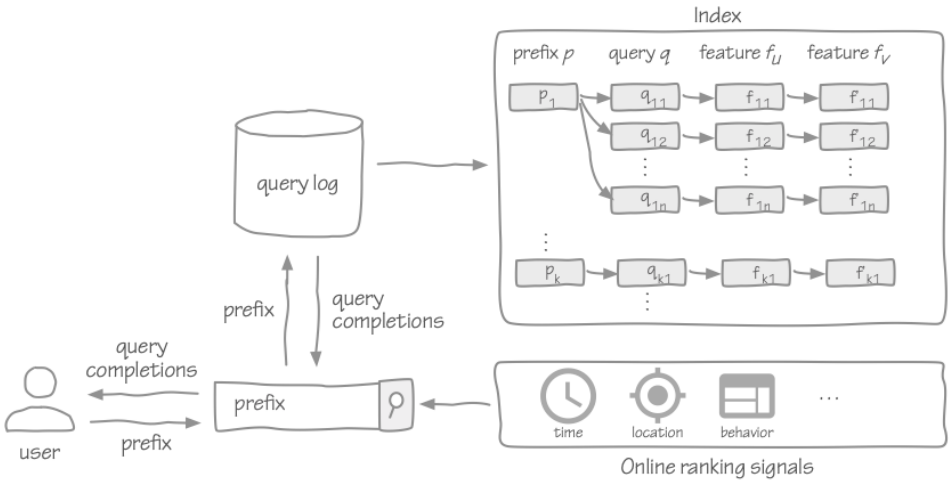
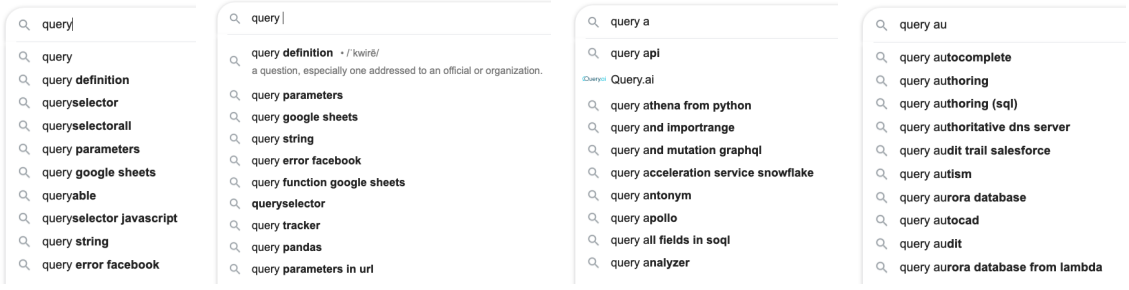


Fig. 7.10: A basic QAC framework (image source: [36])



(a) "query" (b) "query " (c) "query a" (d) "query au"

Fig. 7.11: Example query auto-completions with different inputs [122]

Existing candidate retrieval methods largely fall into the following five modes [176], depending on how a possible query completion T is considered a "match" to an input query Q,

Exact match	T matches Q when it is identical to Q; This is the most basic candidate retrieval method.
Prefix match	T is a "match" of Q when Q is a prefix of T; This is perhaps the most popular candidate retrieval method.

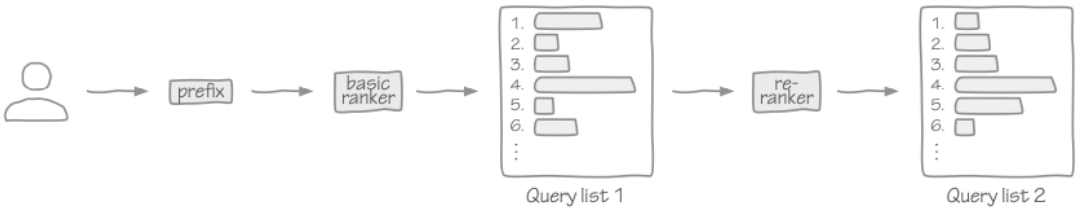


Fig. 7.12: General Flow for QAC Online Phase (image source: [36])

Multi-term prefix match	T is a "match" of Q when for each token t_{Q_i} of Q, there is a token t_{T_j} in T for which t_{Q_i} is a prefix. Further constraints can be enforced that the token t_Ts appears in the same order in T as their corresponding prefixes t_Qs in Q.
Pattern match	Pattern match is similar to multi-term prefix match, except that pattern match considers general substring match instead of just prefix match.
Relaxed pattern match	Tokens t_{Q_i} of Q and t_{T_j} in T are matched based on a string similarity or distance function with a specific limit on distance, for example, edit distance or Hamming distance.

As illustrated by Table 7.1, the matching criteria used by the above candidate retrieval methods are increasingly relaxed and also more computationally expensive when it comes to matching a user input against possible query completions. More restrictive candidate retrieval methods such as prefix match are more efficient in terms of build time, online latency and memory consumption. More relaxed candidate retrieval methods, such as relaxed pattern match, lead to more robustness against unintended errors in user input and essentially enable both query auto-completion and spell correction. The preferred candidate retrieval method to select is usually the one that provides the best balance between usefulness and efficiency, where usefulness is measured based on user feedback and efficiency by runtime latency.

Table 7.1: Matches against candidate query completion "*this is an example*" by different candidate retrieval methods

Input Entered	Exact	Prefix	Multi-term	Pattern match	Relaxed
this is an example	✓	✓	✓	✓	✓
this is an		✓	✓	✓	✓
th is an			✓	✓	✓
s ample				✓	✓
that is ample					✓

Based on how the score for re-ranking is computed, existing re-ranking approaches in query auto completion fall under two broad categories: heuristic-based and learning-based [36]. A heuristic-based approach computes a score for each candidate query completion using a heuristic function that considers different sources for each possible query completion, such as popularity, location, user demographic information, in a deterministic manner. In contrast, a learning-based approach learns re-ranking models with a learning algorithm and training data. The learning algorithm may rely on hand-crafted features, such as query history

and user reactions, such as click-through and query reformulation, or adopt deep neural networks such as Convolutional Neural Networks (CNN) to improve re-ranking by better considering the semantic coherence between the user input and suggested candidate query completions.

The aforementioned retrieve-and-rank approach for QAC discussed so far is efficient and effective when the user input already exists in the pre-computed data structure. However, user input queries are sometimes new and unseen, leading to challenges in QAC. For instance, 15% of queries submitted to Google search engine have never been seen before [103]. A retrieve-and-rank QAC approach cannot handle such **unseen queries** (**long-tail queries**) since they do not exist in the pre-computed data structure.

To handle such unseen/long-tail queries, a **generative** approach leveraging neural language models may work better. Earlier generative QAC work such as [244, 105] achieve high accuracy. However, such models are costly to deploy due to significant computational overhead at runtime, even with a more efficient candidate search algorithm [329]. More recent work has successfully improved the efficiency for production usage, by leveraging a more optimized neural language model for QAC [330, 163], achieving a better balance between accuracy and efficiency. In the case of NLIDBs, the chance for a user input query being a unseen query is potentially even greater due to a much smaller user base than web search engines and hence more limited query log to pre-compute possible query completions. Generative QAC is a promising direction towards addressing the long-tail query problem in NLIDB, with the additional challenge and opportunities to leverage the (semi-)structured information from the underlying database.

7.2.2 Query Suggestions Beyond Auto-Completion

Query suggestion as auto-completion allows a user to formulate queries with less effort. Query suggestion can also improve the user's original query and suggest new exploration paths, enabling a user to discover other questions of interest. Techniques for generating query suggestions mostly come from similar works in the context of Web search engine and query recommendation for database queries. They can be adapted to provide query suggestions in the context of NLIDB. Such techniques largely fall into the following two types [238].

7.2.2.1 Click-Through-based Query Suggestion

Click-Through-based query suggestion is a popular approach in search systems. The basic idea is to leverage query result click-through information across all or a subset of the users to identify common follow-up queries for users issuing similar search queries. Two queries are similar if they result in clicking on the same or similar URL. The query suggestions can be ranked based on (a) the similarity of the queries to the input query; and (b) the support, which measures how much attention the query results have received from users based on user clicks [13]. In the context of NLIDBs, we can view search queries that result in the same or similar structured query as similar and measure user attention to query results based on user interaction level such as time spent on examining the results.

7.2.2.2 Session-based Query Suggestion

A **query session** or **session** for short [154] refers to a series of interactions with the underlying search or query engine by the user toward addressing a single information need. Previous studies on search engines

have found that users often reformulate their original queries into other related queries in order to get better results. The basic idea of **session-based query suggestion** is to learn such query reformulations from past query sessions for relevant query suggestions to the new query session.

The key steps for conventional session-based query suggestion include:

1. **Session Identification**. This step identifies the boundary of each individual query session from the submission of an initial query through the submission of the final query ending the session. Each query session consists of a sequence of queries submitted and other user interactions during the session along with additional context information such as timestamp, meta data about the user and results returned for each query. For each query session, its **session Length** is the number of queries submitted by a single user during the session; its **session duration** is the time period from the start of the session to the end of it.
   The first step towards session identification is to identify unique users. For systems requiring users to log in, identifying unique users is a trivial task. For other systems requiring no user log in to use, unique users can be identified with a combination of Internet Protocol (IP) address of the user machine, browser cookies and temporal boundary.
   Based on the query log of each unique user, query sessions can be identified with one of the following common methods [154].

   Method 1: IP and cookie. This method defines a session as the period from the first interaction by the user with underlying system through the last interaction as recorded in the transaction log. It uses the user's IP address and the browser cookie to determine the initial query and all subsequent queries to establish the beginning and the end of a session. A change in either IP address or cookie always results in a new session.
   Method 2: IP, cookie, and temporal cutoff. This method also uses the user's IP address and the browser cookie to determine the initial query and all subsequent queries to establish the beginning and the end of a session. It uses a pre-defined temporal cutoff between interactions to determine a session boundary. More specifically, if the same user submits two queries that are more than a pre-defined temporal cutoff apart, then the two queries will be deemed to belong to two different sessions. The most commonly used temporal cutoff is 30 minutes [154, 132].
   Method 3: IP, cookie, and content change. This method also uses the user's IP address and the browser cookie to determine the initial query and subsequent queries. It uses the changes in the content of the user queries to determine a session boundary. One simple yet effective approach of detecting content change is based on query term overlapping: if a subsequent query by the same user contained no terms in common with the previous query, then it is deemed as the start of a new session [154].

2. **Session Aggregation** Once individual user sessions are identified, information from these sessions needs to be extracted and stored in a data structure. Exactly how this step is done depends on the method being used to generate query suggestions. Generally speaking, if query suggestions are based on the co-occurrence of queries within a session, then such pairs of queries are extracted from each session and identical pairs from different users are aggregated. If query suggestions are based on the query sequence, then the entire sessions are stored and identical sessions of different users are aggregated. For queries against databases, each session can be summarized to reflect tuples covered by that session [46, 45]. Such session summaries are then aggregated for generating query suggestions.
3. **Query Suggestion Generation** Given an input question with a query session, this step returns a ranked list of query suggestions based on past query sessions and other factors (for instance, user profiles). The process of query suggestion generation is similar to candidate generation for auto-completion described earlier in Chapter 7.2.1: first *retrieving candidates*, followed by *ranking* and then *re-ranking candidates*.

The candidate retrieval step can be based on a pair-wise string similarity function such as those for query based on pattern match or relaxed pattern match. This step may also use more complex models capturing query sequence in the current session and its resemblance to historical query sequence models built from the query log [132]. Ranking and re-ranking steps can be based on trained machine learning models using features of initial and rewritten query pairs such as token count [160, 273], or other query-independent factors such as user profiles [45, 46].

One major limitation of the conventional retrieval-based query approach described so far is that it handles rare and unseen queries poorly. As such, various methods have been proposed to address this limitation. One approach is to identify orthogonal query suggestions to better handle long-tail queries [319]. The idea is to identify query pairs with non-zero result overlapping but low term overlap, so that the system can propose related queries that are syntactically different from the original query but semantically similar. A common approach is to generate query suggestions by leveraging query logs and external resources. Earlier work [305] proposes to generate query suggestions based on query templates. The basic idea is that for queries sharing the same query template, their related queries also share similar structures. As a simple example, if we observe a reformulation of ⟨*city*⟩ *surfing* to ⟨*city*⟩ *beach* in a query log, then for query "*Malaita surfing*", we can suggest "*Malaita beach*", even though neither queries were previously specified. More recent methods leverage sophisticated generative models to capture user interactions and query sessions. For example, a sequence-to-sequence model can be extended with query-aware attention to capture the structure of the session context [85]. As another example, Feedback Memory Network (FMN) models user interactions in the form of browsing and click action with a search engine as the attention over the top-ranked documents, along with the query sequence, to better capture the underlying information needs [346].

7.3 Automatic Data Insights

Besides suggestion at query time, interactivity of an NLIDB is also important when returning results to users to help them better understand the results and to facilitate exploratory data analysis. Exploratory data analysis is an iterative process of asking and answering questions about data through interacting with the underlying databases [316]. An important goal of NLIDBs is to facilitate exploratory data analysis and help users to obtain meaningful, useful and actionable insights for problem solving or decision making. Not surprisingly, real-world NLIDBs often return results along with automatic data insights (**auto-insights**). **Visual insights** in the form of data visualization to help reveal possible insights in the data. **Textual insights** present noteworthy facts based on the results returned. Visual insights and textual insights are often presented together, with textual insights highlighting the noteworthy observations one can make from a visualization.

7.3.1 Categorization

Figure 7.13 illustrates a combination of visual and textual insights provided by Narrative for Power BI: the data visualization surfaces details related to *Total Revenue* and *Gross Margin %*, while the textual insights describe notable facts about the data, such as the trends of the two attributes (e.g., "*Gross Margin % rising*") and how they compare to each other (e.g., "*moved in opposite directions from Jan to Dec*").

According to [182], common auto-insight methods fall into the following 12 types, in the order of their popularity:

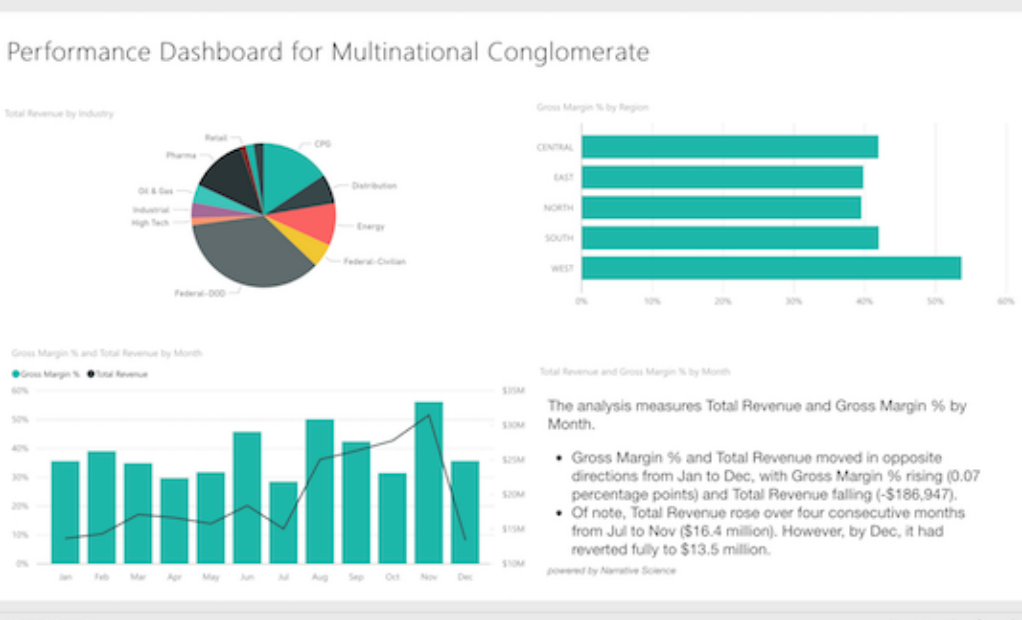


Fig. 7.13: Visual and textual insights in Microsoft Power BI. (image source: [216]

Outliers	Auto-insights revealing a value or data point that differs substantially from the rest of the data. For example, "*Only Kevin Chen has more than 20 friends*" describes a fact about an outlier.
Value/Derived Value	Auto-insights about a single prominent value of a variable or a value derived from multiple values of one or more variables. For a categorical variable, the value can be derived based on the number of unique categories and the proportion of categories (e.g., "*Total Revenue by region*" in Figure 7.13); for a numerical variable, the value can be derived based on the average of the variable (e.g., "*A highschooler has 6.5 friends on average*".
Association	Auto-insights on the quantitative relationship between two numerical variables, including linear relationship using Pearson correlation or non-linear relationship using more sophisticated measures such as Spearman correlation. For example, "*Weight and height show a moderate but statistically significant positive linear relationship with each other*" describes an insight on the association between *weight* and *height*, including details such as strength ("*moderate*"), direction ("*positive*"), statistical significance ("*statistically significant*"), and the type of correlation ("*linear relationship*").
Difference	Auto-insights on a quantitative comparison between two values/derived values, or distributions of two attributes. Figure 7.14 illustrates three different kinds of auto-insights for difference: (1) *one to one*: comparing one individual value against another; (2) *one to multiple*: comparing one individual value against a derived value from multiple values; and (3) *multiple to multiple*: comparing distributions of two attributes.

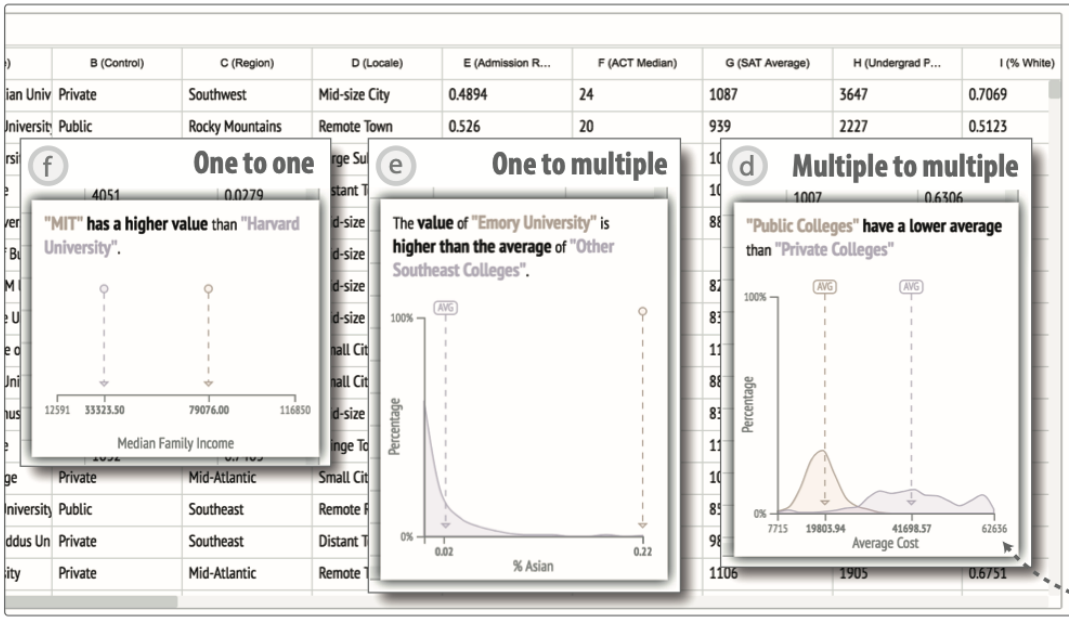


Fig. 7.14: Example auto-insights based on difference (image source: [181])

Trend	Auto-insights about temporal trends, including upward and downward trends, steady trends, and periodicity (such as seasonality). Figure 7.13 illustrates example auto-insights for trend.
Distribution	Auto-insights about the distribution of a variable, such as the range of a numerical variable, and other measures of distribution including normality, uniformity, dispersion, skewness, and heavytailedness to identify data with noteworthy distributions.
Extreme	Auto-insights about the minimum and maximum values for an attribute. Statements such as "*John Doe has the least number of friends*" and "*Kevin Chen has the most number of friends*" both describe extreme: the former about the minimum value and the latter about the maximum.
Visual Motifs	Auto-insights about unique patterns of the data that do not fall into other auto-insight types. For instance, Figure 7.15 illustrates a few special patterns in scatterplots identified using scagnostics measures [341].
Cluster	Auto-insights about clusters in a scatterplot using methods such as K-means and DBSCAN.
Metadata	Auto-insights about metadata information about a dataset, such as the meanings of labels, data quality issues such as missing values or errors, and appearance or disappearance of dimensions [58]. For example, the fact "*John Doe has no friend. Can it be right?*" was about potential data quality issue.
Rank	Auto-insights obtained by sorting categories using a numerical variable and showing the breakdown of selected data attributes [333]. For instance, "*Pickup truck, compact SUV, and sedan are the top 3 categories by sales in the year of 2020*" is a fact obtained by ranking different types of cars by their sales.

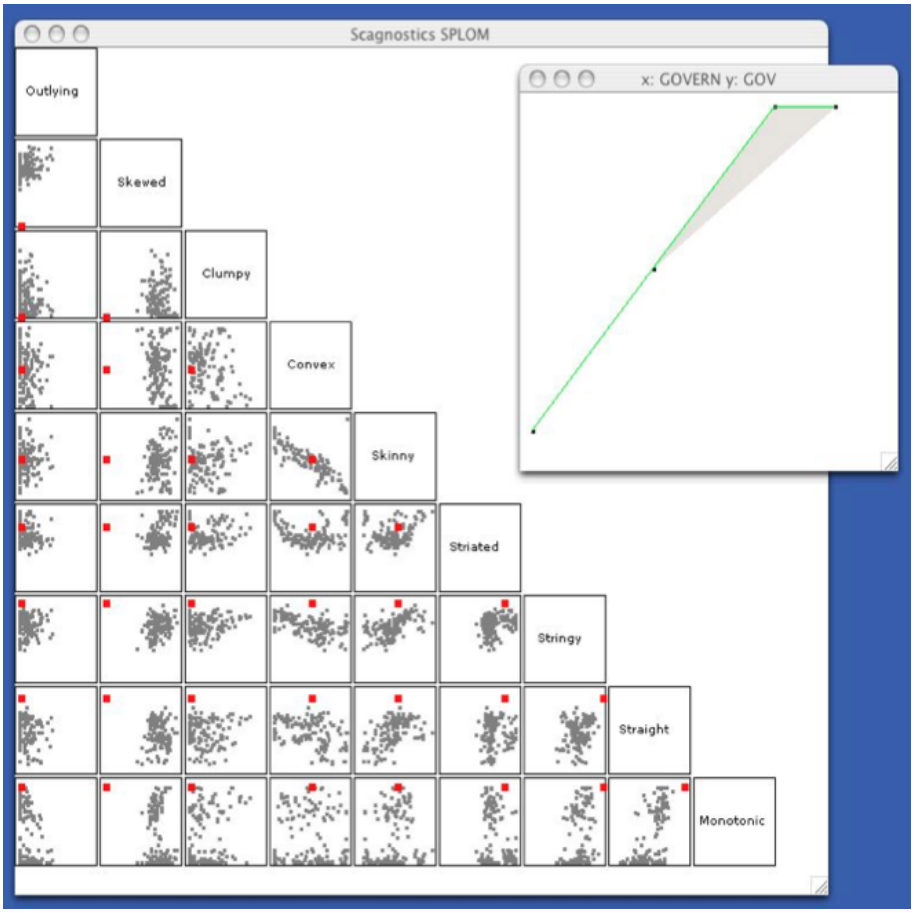
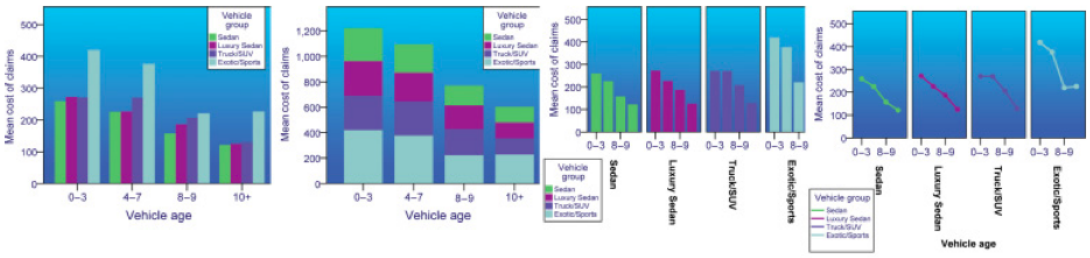


Fig. 7.15: Example auto-insights based on visual motif (image source: [341])

Compound Fact — Auto-insights obtained by combining two or more auto-insights. For example, the fact "*Gross Margin % and Total Revenue moved in opposite directions from Jan to Dec*" is a compound fact obtained by comparing the differences between the trends of two attributes. [359] presents a detailed taxonomy of compound facts and how different visualizations (e.g., the ones shown in Figure 7.16) impact users' comprehension of the intended auto-insights based on a large scale crowd-sourced user study.

The different types of auto-insights discussed above can be generated as visual insights or textual insights. We refer the readers to Chapter 6 for text generation from data for textual insights. We now discuss visual insights and multimodal insights with mixture of visual and textual insights.



(a) Crossed Bar (b) Stacked Bar (c) Nested Bar (d) Nested Line

Fig. 7.16: Example composite visualization for auto-insights on compound fact (image source: [359])

7.3.2 Visualization recommendation

Visualization recommendation automatically generates visual insights. It facilitates exploratory data analysis by proactively suggesting effective visual encoding to enable better understanding of the returned results or recommending potentially interesting visualizations to enable users to decide the next steps to take during the exploration. Users can then interact with such visuals to further analyze the data or embed them into BI reports.

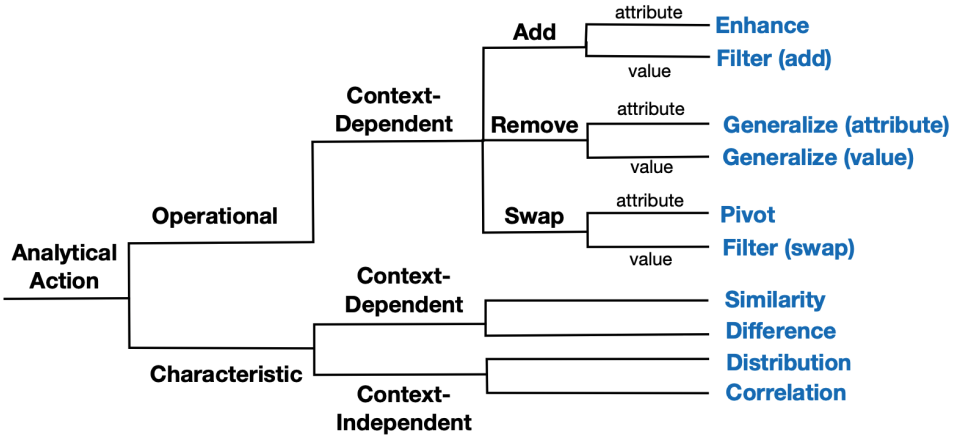


Fig. 7.17: A taxonomy of common analytical actions used in recommending visualizations for visual analysis. The analytical actions are indicated in blue (image source: [184])

Given the vast design space of possible visual insights from data described earlier, it is important to follow a principled approach for effective visualization recommendation.

First, one needs to understand the vast design space of visual recommendations. Figure 7.17 depicts a taxonomy of common analytical action-based recommendation categories in visual analysis [184]. The

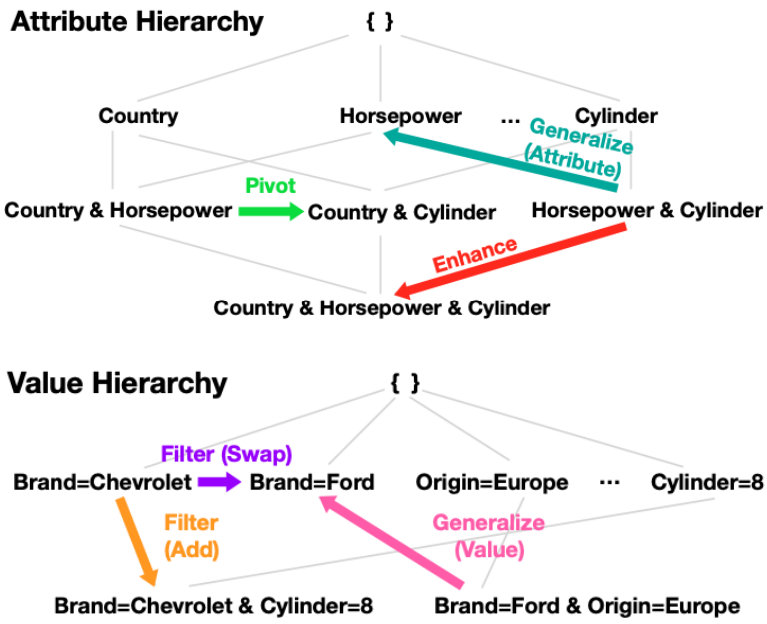


Fig. 7.18: Example operational actions through the attributes and value hierarchies (image source: [184])

operational category describes possible analytical actions through the visualization space. The characteristic category describes possible actions to surface certain characteristic patterns in the data. The taxonomy is further broken down into two categories: context-dependent and context-independent. If actions must depend on the current view (i.e., the selected attributes, values, and visual encoding), they are context-dependent; otherwise, they are context-independent. Figure 7.18 illustrates a few possible operational actions through the attributes and value hierarchies of a database about car sales.

The usefulness of different recommendation categories varies by the underlying data and user expertise. For designing practical visual recommendations, one may start with following findings obtained via prior user studies such as [184] and [359]. For instance, studies by [184] found that `correlation` was more frequently used for datasets with large numbers of measures; while `distribution` was more frequently used for datasets with fewer measures but more dimensions. However, the applicability of these findings vary for use cases with different dataset properties (e.g., higher-dimensional), different problem domains, and varying user expertise. As such, additional use studies may be needed to understand the efficacy of different recommendation categories for the specific use cases of importance by better taking possible common use cases into account.

7.4 Explanation

Prior user studies (e.g., [296, 331]) have found a strong need for surfacing better explanations in NLIDBs. Such findings are consistent with recent emerging trend on explainalbity for natural language processing

in general [81, 82]. [91] highlights the motivations behinds the needs for explanations about AI models: facilitating understanding of model design at a high-level or inner works, presenting details about the data over which a model is built, surfacing ethical considerations baked in to the model, highlighting expectation mismatches, and explanations in service of business actionability. Within the context of NLIDBs, the main drives for explanations are to aid users in gaining a deeper understanding of the inner workings of the NDLIBs; this includes comprehending the model design at a high level and identifying potential expectation mismatches. Specifically, explanations in an NLIDB focus on helping its users to understand the following aspects of the system:

Linguistic Capabilities

In the foreseeable future, NLIDBs will still have limitations in their language understanding capabilities. As such, to effectively leverage an NLIDB, a user needs to have a rough idea of what the system can and cannot understand. The system can help the user acquire such knowledge via explicit documentation (e.g., list the grammar corresponding to the controlled natural language supported by the system in Figure 1.3) as well as explaining its interpretation of individual questions via visual highlighting, visual widgets (e.g., Figure 7.6) or natural language description of its query interpretations.

System Decisions

Given a natural language question, a NDLIB needs to make many decisions in query understanding and translation in order to determine the corresponding structure query: from mapping the lexicons into database artifacts to determining one single interpretation of the question among all possible ones. The system can expose such decisions to help the user better understand and trust the results returned. Furthermore, as discussed earlier in Chapter 7.1.3, the user may also override system decisions and hence obtain more agency via direct UI manipulation or natural language instructions.

Query Results

After an NLIDB successfully translates and executes a given user question, the user may benefit from further assistance to comprehend the results returned and to decide the next steps. Explaining answers and missing answers for database queries is an active research area in the past decades.

One popular type of methods towards explaining database answers is via **data provenance** [33]. **Data provenance**, also referred to as "**lineage**" or "**pedigree**," is the description of the origins of a piece of data and the process by which it arrived in a database. *Where-provenance* describes where a specific piece of output comes from in the input; *why-provenance* shows input that explains why an answer is produced; *How-provenance* and *why-not provenance* explains why an answer is missing.

Another common type is **intervention-based** methods [215, 268, 269]. Such methods are motivated by **causality** [129]. **Causality** refers to the actual causes of answers or non-answers to their queries. Causality needs to be established through a contingency set. Another problem accompanied with causality is **responsibility**, which measures the degree of causality for each actual cause to the answers or non-answers. The basic idea of intervention-based methods is to identify changes to the database that would significantly change the query answer. The main challenge for intervention-based methods is their computing cost. Both causality and responsibility are more difficult to compute than the lineage.

Both provenance- and intervention-based methods focus on understanding how input data impacts the query answers. In contrast, **query-based** methods focus on understanding how a query affects the query answers or missing answers [4, 43, 60, 119]. In this approach, queries are broken down into parts and these individual parts are viewed as potential causes for the query results (existing or missing answers).

In recent years, other types of methods for explanations such as **statistics or observation-based**, **summarization-based** and **model-based** are also emerging. We refer the reader to [120] for a more comprehensive review of this topic.

Data Visualization

Data visualization is an important part of helping users to better understand query results [136]. As suggested by prior work [125, 136], when supporting novice users in learning how to use and interpret visualizations, the system also needs to consider explaining both what is displayed and why these visual mapping are chosen (by providing reasons, advantages and disadvantages).

Rationale Behind Query Suggestions

As discussed in Chapter 7.2.2, query suggestion is important in assisting exploratory data analysis by enabling a user to discover other questions potentially of interest. When presenting query suggestions, surfacing the rationale behind them can help a user to better decide which suggestion(s) to follow and foster trust.

Figure 7.19 illustrates a user interface that provides explanations for several key aspects of the underlying NLIDB. First, both the Annotated Question View and the Detect Entities View highlighting important tokens in an question, implicitly explaining the system's linguistic capabilities. In addition, the Detect Entities View illustrates the mappings between the important tokens to the underlying data artifacts, explaining decisions made in interpreting the input question. Furthermore, the interactive drop-down list in this view enables a user to override the system decisions as needed. The last but not the least, the Sample Data View, the Explaner View, and the Answer on the Data use an example database to explain the structured query translated from the input question step by step. Such example-based explanations help a user, particular one with limited knowledge of query languages, to better assess the correctness of the structured query.

7.5 Conversational Natural Language Interfaces to Databases

So far, we have discussed interactivity at both query time and when the result is returned. While research on natural language interfaces to databases often assumes that each query is specified independently of one another, in practice, search is rarely a single-step process. Users often pose questions in an interactive fashion with a sequence of interrelated questions [153]. Each followup question is often only partially specified and could refer to any point of recent questions. By allowing questions to be asked sequentially, users also can explore the database in a more flexible manner, which reduces their cognitive burden and increases their system involvement. The underlying NLIDB system may also provide an explanation of its answers and ask users questions to solicit input. Supporting such an interactive iterative search process requires the underlying NLIDB systems to be capable of processing conversational requests to access information in databases as well as generating responses and initiating questions.

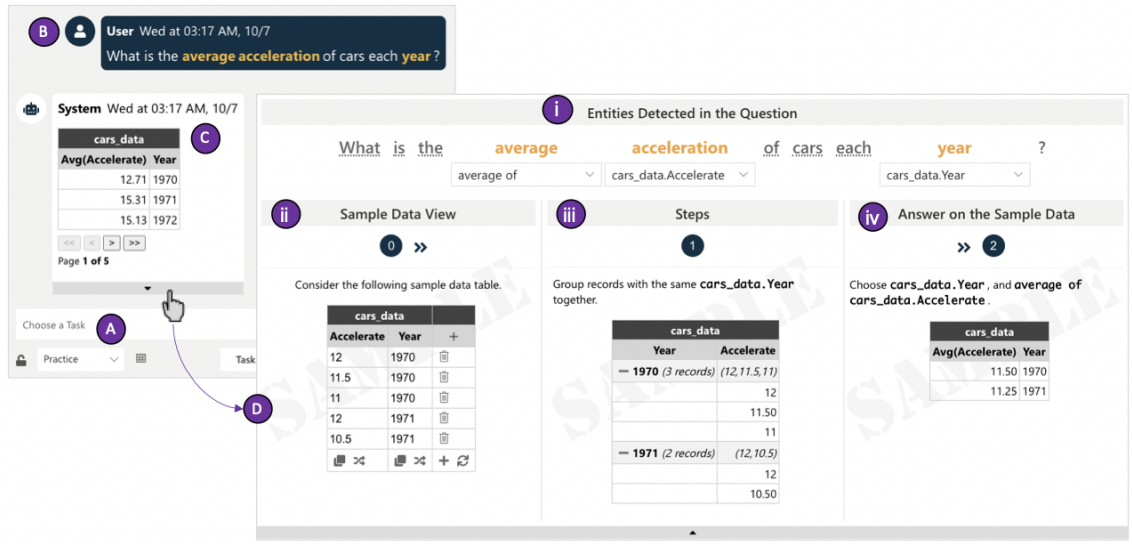


Fig. 7.19: The DIY technique implemented in a QA shell. (A) Query input, (B) Annotated Question View shows the question with important tokens highlighted, (C) Answer on Production Database View shows the query result on the production database (DB), and (D) Debug View. (i) Detect Entities View shows the mappings between the question and the query, (ii) Sample Data View shows a small-but-relevant subset (sample testing DB) of the production DB, (iii) Explainer View provides step-by-step explanations of the query, and (iv) Answer on Sample Data View shows the query result on the sample testing DB (image source: [232])

7.5.1 Discourse Structure

Figure 7.20 illustrates an example conversation session with a database. As can be seen, the user asks multiple questions to get the final answer; during the session, the NLIDB returns results along with their natural language summaries based on prior queries as well as requests user input for disambiguation. To design and build effective conversational NLIDBs, we first need to understand the basic structure of such a conversation session, known as **discourse structure**. Discourse structure in a conversational NLIDB consists of both the utterances from the NLIDB system and its users and the thematic relationships among the utterances [3, 21, 42]

7.5.1.1 Individual Utterances

Individual utterances in a conversational query session can be classified according to their intended affects (formally known as **speech act**s) into the following categories:

Question — A question is essentially a request for information. The questions are often issued by users to retrieve information from the underlying database. The system may also asks questions and request user input to help disambiguate the user query. For instance, in Figure 7.20, most of the questions such as “What are the names of all the dorms?” are issued by the user, while

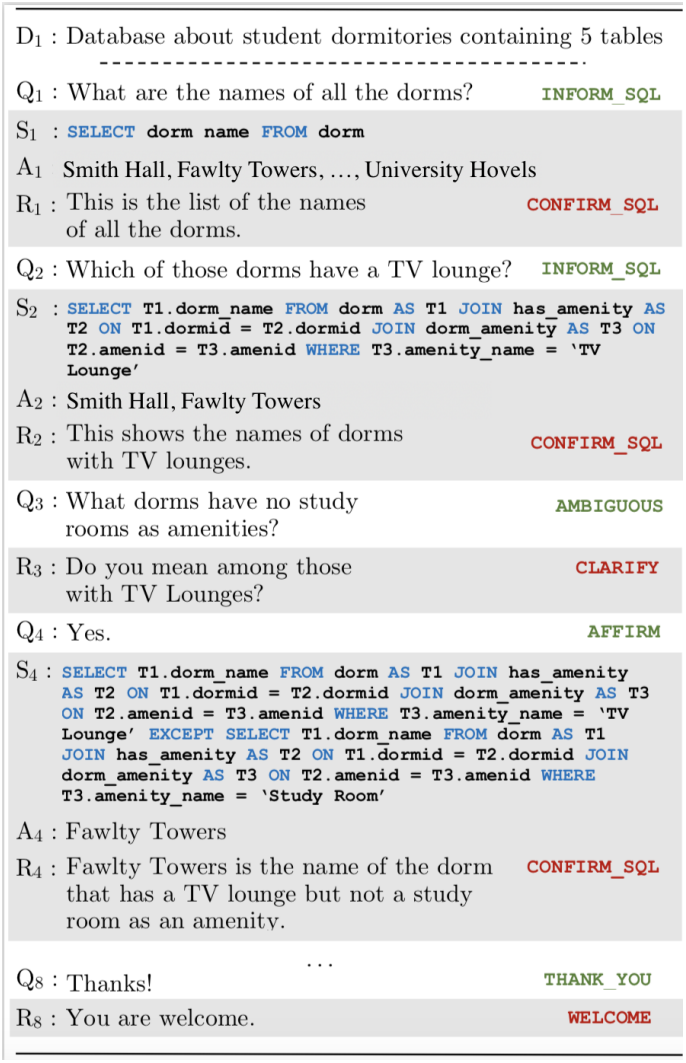


Fig. 7.20 Example Conversation with a Database (image source [369])

the system also asks questions such as "Do you mean among those with TV Lounge?" for user input to help query disambiguation.

Request — A request is a utterance asking for action. For instance, "*Order by year*" is a request from the user to the underlying system to sort the returned answers by the value of *year* attribute; "*Please wait while I am finding the answers*" is a request from the system to the user.

Answer — An answer is a response to a question with the requested information or to a request that performs the requested action. An answer can be multi-modal, for instance in the form of text, visualization, or mixture of both.

Explanation — An explanation is a statement accompanying an answer to make it more understandable. For example, in Figure 7.20, the responses followed by answers, R_1, R_2, and R_4 are all explanations to help users to better understand the answers.

Assertion An assertion is a statement that one states to be true. In the context of NLIDBs, an assertion is often a statement that the system is unable to find the requested information ("*The database does not contain this information*.") or perform the requested action ("*You do not have access to the requested information*").

Based on the framework proposed in [42], the semantics of each utterance is captured by its **Topic** and **Focus**. Topic expresses what each utterance is about, which in the context of NLIDBs usually corresponds to a type of entity; while Focus refers to a particular aspect of Topic and indicates the current focus of attention on a particular topic, which in the context of NLIDBS usually corresponds to some property of the entity. For instance, for Q1 "What are the names of all the dorms" in Figure 7.20, its Topic is "dorm" and Focus is the "name" of "dorm."

7.5.2 Discourse Transition

Discourse transition refers to how the semantics of the conversation is changed from one utterance to another as the interaction proceeds and how such changes reflect the process of the user information needs. One important characteristic of utterances in a conversational query session is that many of the utterances, particularly questions and requests are fragmentary, as such their semantics could only be fully understood by taking the context of some prior utterances into consideration. Discourse transition determines whether and how to use the context to interpret a utterance.

A typical model for the human information seeking process formulates it as the following four-phase iterative framework [133, 211, 291, 303]:

- Query formulation
- Action (running the query)
- Review of results
- Query reformulation, if necessary

This framework corresponds to two important types of discourse transitions: transitions between user questions or requests and transitions between results and the subsequent query reformulation.

7.5.2.1 User Question-to-Question Transition

The most essential type of discourse transition in a conversational NLIDB is the transition between questions or requests issued by a user. For simplicity, in the rest of this section, we refer to a question or request issued by a user as a **user question** and discourse transition among them as **user question-to-question transition**. Based on how a user question semantically relates to some previous user question in the same session, we can categorize each into one of the following categories [21, 42]:

Topic Extension The current question concerns a similar topic as that of a previous question, but with different focus or constraints, including:

Constraint Refinement (refinement) If the current user question asks for the same type of entity (i.e. the same topic) as some previous question, with different restricting conditions, asking, thus, for a subset, superset or disjoint set of the same class. This class of follow-up questions can be further divided into the following categories:

Addition The current question adds more restricting conditions over the ones specified in some previous question, typically resulting in a subset of the same class. For instance, most of the user responses shown in Figure 7.20 belong to this category and eventually result in a complex SQL query. As illustrated, addition questions allow a user to specify more complex queries in an incremental fashion.

Substitution The current question replaces some restricting conditions already specified in some previous question, often resulting in a disjoint set of the same class.

Q_5. Which dorms on North Campus have a kitchen?
Q_6. What about those on South Campus?

Q_6 replaces the original constraint "on North Campus" in Q_5 with "on South Campus," while keeping the other constraint "have a kitchen."

Removal The current question removes restricting conditions over the ones specified in some previous question, typically resulting in a superset of the same class.

Q_7 Which dorms have air conditioning?
Q_8 Return all dorms regardless of amenities.

The system needs to understand that $Q8$ essentially removes all constraints specified in Q_7.

Participant Shift (Theme-property) If the current question asks for the same property as the immediately preceding question but for another specific entity of the same type, also known as **theme-property** by Bertomeu et al. [21].

Q_9 What kind of furniture are provided for North Squad dorm?
Q_{10} What about South Squad?

Both Q_9 and Q_{10} inquire details of furnishing for dorms, but about two different specific dorms. The semantics of Q_{10} could also be fully interpreted based on its immediately preceding question Q_9.

Topic exploration (Theme-entity) If the current question asks for the same topic but with a different focus as some previous question. Such transitions are also known as **theme-entity** as defined in [21].

Q_{11} What kind of furniture are provided for the dorms?
Q_{12} What about the types of bathrooms?

Each query asks about different properties of dorms. Q_{11} asks about furnishing provided for the dorms while Q_{12} inquires about the types of bathroom (e.g., shared, private, or communal) in the dorms. Q_{12} could only be fully interpreted by inheriting the topic "dorm" from Q_{11}.

Topic shift If two consecutive questions ask about two different topics, then such a transition is referred to as topic shift.

Q_{13} What is the average size of the dorms?
Q_{14} Is there a kitchen?

Q_{14} How many dorms are there on the North campus?
Q_{15} What about gyms?

As can be seen, even with topic shift, a follow-up question may still need to be interpreted with the context of an earlier question. In the above example, to properly answer Q_{15}, the system needs to be aware of the implicit constraint "*on the North campus*."

Paraphrase If the question is a paraphrase of some previous question. In such a case, the system can return the previous results directly.

7.5.2.2 User Question-to-Answer Transition

A user may identify additional topic or focus to explore after getting an answer. As such, another important type of discourse transitions in a conversational NLIDB is the transition between answers and the subsequent questions. Following [21], we can categorize them into two categories:

Refinement If the focus of the current question is based on a subset of the entities given in the previous answer, then the current question is considered to be a refinement of the previous question(s). The conversation in Figure 7.20 includes examples of such transitions: when A_1 returns a list of dorms, Q_2 asks for a subset of them that has a TV lounge; then when A_2 returns the list of dorms with a TV lounge, Q_3 asks for a subset of such dorms without study rooms. Refinement transition allows a user to narrow down their search in an iterative fashion based on information obtained from prior answers.

Focus If the current question asks about a specific entity introduced in some previous answer, then this transition is referred to as focus (theme). For example, Q_{17} asks about an entity from the answer of Q_{16}.

Q_{16} Find all dorms with a laundry room.
A_{16} *Betsy Barbour*, *East Quandrangle*, *Fletcher Hall*, ...
Q_{17} What kind of furnishing is provided by Betsy Barbour?

7.5.2.3 Other Types of Transitions

Besides the above two types of dialog transitions initiated by users, dialog transitions in a conversational NLIDB may also include system requests to clarify ambiguous user questions, system responses with a summary or an explanation of returned results, and system notifications about unanswerable or unrelated questions, as well as user responses to help disambiguate questions and confirmation of the returned results. Based on the types of transitions supported by conversational NLIDBs, we can further divide them into **unidirectional** and **bidirectional** conversational NLIDBs.

7.5.3 Unidirectional Conversation

A basic conversational NLIDB needs to support the two most essential types of dialog transitions: user question-to-question transitions and user question-to-answer transitions. Since users initiate both types of dialog transitions, the conversations are **unidirectional**: users ask questions and the system responds with answers. Such a system allows users to construct questions sequentially and enables users to explore the data gradually with a less cognitive burden, which in turn increases user engagement when interacting with the NLIDB system. The discourse transitions involved are user question-to-question and question-to-answer transitions.

Figure 7.21 illustrates an example unidirectional conversation session. As can be seen, the user explicitly uses "their name" to refer to the name of the most recent customer in the second follow-up question. In the third question, the user omits the previously mentioned "their name" and refines the request by asking for the first five customers.

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. 7.21 Example Unidirectional Conversation with SQL Segments from Previous Questions Underlined in SParC semantic parsing in context task. (image source [374])

The main challenge to support such unidirectional conversations is that the underlying system has to effectively process and encode context information to synthesize correct SQL queries. As such, the corresponding task is also referred to as **semantic parsing in context** [374] or **sequential question answering** [153].

To advance research in this direction, a cross-domain Semantic Parsing in Context (SParC) [374] dataset has recently been introduced. It is built on the text-to-SQL Spider benchmark by decomposing complex questions about various databases into multi-turn simpler questions. It consists of 4,298 unidirectional conversations with coherent question sequences covering user interactions with 200 complex databases over 138 domains.

Semantic parsing in context requires mapping each user question into a corresponding executable program. In some cases, a user might ask some questions that cannot be transformed into any executable programs. In a sequential question answering (SQA) [153] task, as shown in Figure 7.22, given a question and a Wikipedia table, the task is to generate the resulting answer directly. SQUALL [289] finds that about 40% of the questions in SQA are unanswerable by SQL, including questions with non-deterministic answers (e.g., "*show me an example of . . .*"), questions where SQL queries would be insufficiently expressive (e.g., "what team has the most consecutive wins?"). Sequential question answering covers a wider range of user possible questions but is less interpretable than sequential semantic parsing.

As mentioned earlier, context history understanding is one of the most important components in these tasks. NaLIX [200], an early conversational NLIDB, captures conversational history as query context and updates the query context for each follow-up query based on discourse transition; the updated query

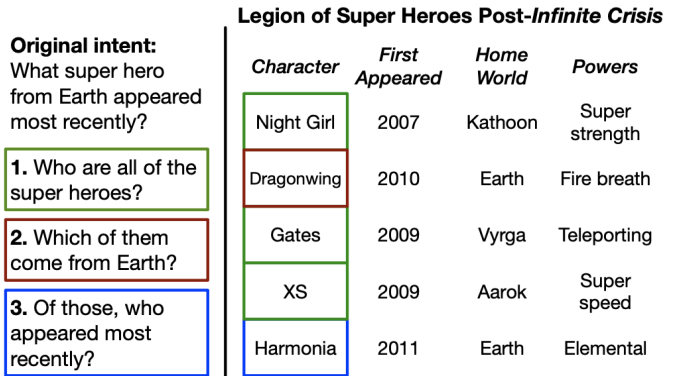


Fig. 7.22 An example question sequence in SQA sequential question answering task. (image source [153])

context is then used for query translation. [302] combines implicit and explicit modeling of references between utterances and incorporate interaction history by model maintaining an interaction-level encoder that updates after each turn, and copy sub-sequences of previously predicted queries during generation. To leverage the correlation between sequentially generated queries, EditSQL [383] proposes an editing-based approach for cross-domain context-dependent text-to-SQL generation tasks including SParC. It introduces an interaction encoder with turn attention and generates SQL for the current turn by editing the resulting SQL in the previous turn. Also, some works such as [153] frame the sequential problem as a reward-guided search-based process.

More recently, large pre-trained language models have demonstrated their surprising ability on encoding multi-turn context history, achieving state-of-the-art performance for various natural language processing tasks. Some studies [275, 349] have shown that, by simply concatenating all previous turns and the current turn as a single string and feeding it to sequence-to-sequence language models such as T5 [264], they can achieve the best performance on many context-dependent semantic parsing tasks. However, real-world dialogues can be lengthy. It can be impossible to input the full dialogue context to current large language models. Some works include recent dialog history spanning several turns. IC-DST [146] utilizes the previous program as a summary of the dialogue history and demonstrates state-of-the-art few-shot performance by using in-context learning with Codex [50], a large GPT-based language model, without training. Moreover, some pre-training methods [312, 370] have been proposed to further improve the representation of the multi-turn dynamics of the dialog in language models. With this approach, SCORE [370] [1] achieves SOTA on three of four conversational tasks (SPARC, COSQL, MWOZ, and SQA). Figure 7.23 shows an example.

An alternative approach to semantic parsing in context is dialogue rewriting. The goal of dialogue rewriting is to reduce a multi-turn dialogue task into a single-turn task via co-reference resolution and ellipsis complementing [100]. [192] presents DIR, a large-scale dialogue rewrite dataset extended from SParC and CoSQL. While dialogue rewriting allows straightforward adaptation of existing NLIDBs to support conversational NLQ, dialogue rewriting is a challenging problem on its own.

[1] https://github.com/microsoft/SCoRE

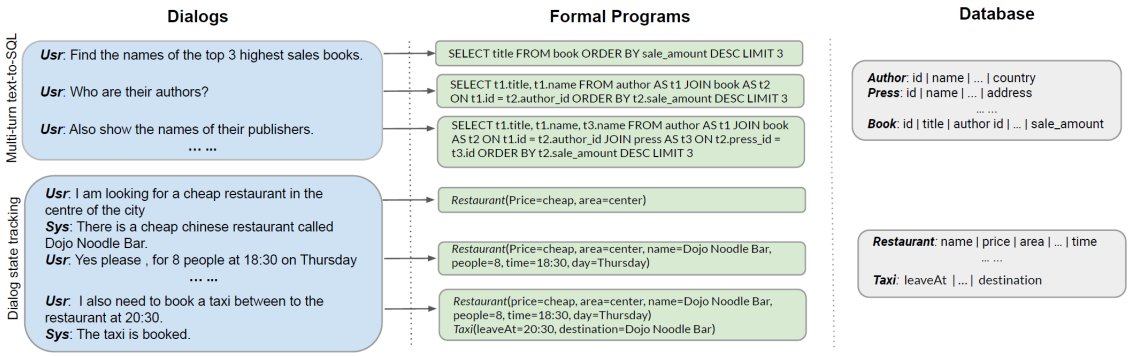


Fig. 7.23: An example from SCORE [370]. (Image from [370]).

7.5.4 Bidirectional Conversation

Most prior work in unidirectional conversational NLIDB assume that all user questions are reasonable and relevant to the underlying databases (e.g., can be mapped into SQL queries). In real-world scenarios, however, users might not be familiar with data stored in the databases, ask questions that maybe be unrelated to the databases, or do not know how to continue finding interesting data insights. As such, users tend to converse with a system to ask a series of questions that can often be under-specified. An ideal and robust NLIDB conversational system that can engage with users by forming its own responses has become increasingly necessary. It needs to first identify such a situation by searching and reasoning through its underlying databases and then appropriately interacting with users to resolve it.

Studies in building bidirectional conversational systems can be found under task-oriented dialogue settings [32]. The goal of task-oriented dialogue systems is to assist the user in performing various concrete tasks such as information access in databases, data exploration, and analysis for business insights, intelligent customer services, and smart home device control. Most of the current state-of-the-art task-oriented systems are powered by structured knowledge, including relational databases, tables, knowledge bases, web services, and applications, in order to complete some concrete tasks for users.

As shown in Figure 7.24, a classical pipeline framework for modeling goal-oriented dialogue systems includes four key components:

Language Understanding	It identifies users' intents and maps input utterances into some semantics frames (e.g., speech acts, slots).
Dialog State Tracking	It maps partial dialogues into predefined dialog states or other logic forms such as SQL.
Dialog Policy learning	This component, also known as Language Understanding decides the system actions or responses for Dialog Management, based on dialogue history and current dialogue states and outputs system actions (e.g., `inform(hotel_name="B&B")`). It can be learned from a human-human paired dialog corpus and using some reinforcement learning approaches.
Response Generation	It generates systems' natural language responses from structured meaning representations produced in the dialog management step.

To study task-oriented dialogue systems, [32] introduces the Multi-Domain Wizard-of-Oz (MultiWOZ) dataset, consisting of 10k human-human written conversations spanning over multiple domains and topics,

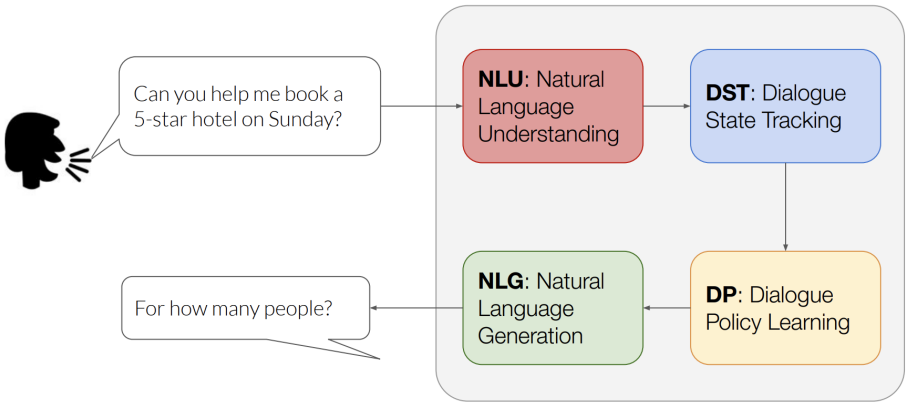


Fig. 7.24: An example of a modularized task-oriented dialogue system. (image source [108])

such as hotel booking or transportation planning. Many neural-based task-oriented dialogue systems [144, 247, 347] have been proposed based on the MultiWOZ dataset. Especially, instead of building different models for each module in the task-oriented dialogue systems, recent works including SimpleToD [144] and SOLOIST [247] try to parameterize classical modular dialog systems using a single Transformer-based auto-regressive language model such as GPT-2. These task-oriented dialogue systems rely on pre-defined slots and values for request processing and only operate on a small number of domains.

In contrast, bidirectional conversational NLIDB systems (which can still be seen as one type of task-oriented or information-seeking dialogue system) need to support general-purpose exploration and querying of an arbitrary database by users. To do so, these systems must possess the ability to (1) detect questions relevant to or answerable by the database, (2) ground user questions into executable queries or answers directly if possible, (3) return results or suggest insights and next steps to the user, and (4) handle unanswerable questions. General-purpose and scalable NLIDB systems would not be tied to databases in any specific domains. To drive the progress of building such NLIDB systems, [369] defines and introduces a cross-domain Conversational text-to-SQL (CoSQL) task, consisting of a Wizard-of-Oz collection of 3k dialogues querying 200 complex databases spanning 138 domains. Each dialogue in the CoSQL dataset simulates a real-world DB query scenario with a user exploring the database and a SQL expert searching answers with SQL, clarifying ambiguous requests, or otherwise informing of unanswerable user requests. CoSQL includes three different tasks: (1) SQL-grounded dialogue state tracking to map user utterances into SQL queries if possible given the interaction history; (2) natural language response generation based on an executed SQL and its results for user verification; and (3) user dialogue act prediction to detect and resolve ambiguous and unanswerable questions.

In an extension of CoSQL, [384] introduces a question intention classification component for building a robust bidirectional conversational NLIDB system. The task is to distinguish different unanswerable questions from answerable questions and then take appropriate actions. This component has to classify a user question about a database into several predefined categories (shown in Figure 7.25):

Improper to DB	A user might ask some questions that are improper to a database such as small talks. In this case, the NLIDB system can inform the user and return a tutorial instead of answering the questions.
Require external knowledge	A user might ask some questions that are relevant to the database but require extra information which cannot be found in the database. This could be

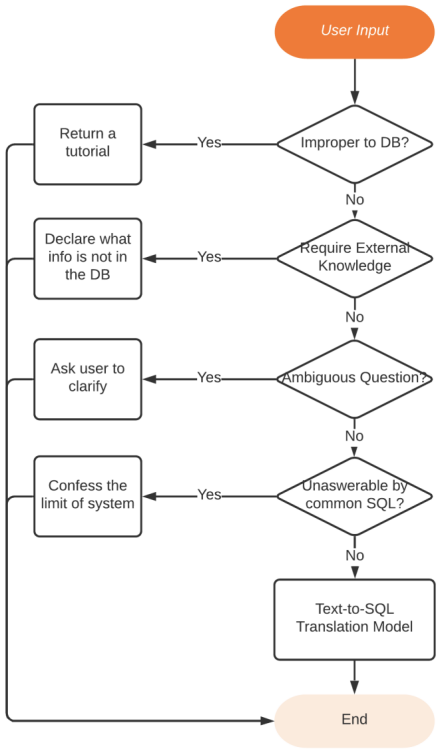


Fig. 7.25 A robust bidirectional conversational NLIDB system needs to distinguish answerable questions from other possible inputs and then triggers different actions for different types of unanswerable inputs. (image source [384])

common when the user is not familiar with data stored in the database. The NLIDB system needs to detect which parts cannot be answered and ask the user for more information.

Ambiguous — A user might ask ambiguous or under-specified questions. The NLIDB system needs to indicate the ambiguities and ask the user to clarify.

Unanswerable by common SQL — This type contains questions that are relevant to the database but are not answerable by SQL. The NLIDB system needs to find these parts and call a question-answering model instead of a text-to-SQL model to handle them.

Answerable — This type refers to regular user questions that are relevant to and answerable by the database.

This intention classification component helps the NLIDB system to know when to abstain and confirm potential failures with the user, which gains the trust between the user and the system. For example, by knowing when it may be wrong, the system can request human intervention and intent clarification, and incorporate and learn from human feedback, resolving language ambiguity and complexity.

A recent study [135] shows that LLMs such as ChatGPT with intuitive natural language prompts are able to achieve state-of-the-art performance on dialogue state tracking. Such general purpose models show great promise to enhance specialized NLIDBs to go beyond their pre-defined scope to better support conversational NLQ, for both unidirectional and bidirectional conversations.

7.6 Multi-Modal Conversational NLIDB

We have just discussed conversational NLIDBs where users interact with the underlying databases via natural language (typed or spoken). Natural language is effective in enabling users to describe objects and time periods that cannot be referred to directly and express quantification information. Natural language also allows more efficient interactions by permitting users' utterance to depend on context for their interpretation [70]. In a multi-modal NLIDB, users can also interact with the underlying databases via **direct manipulation**. Direct manipulation via a keyboard and mouse/touch screen is effective when one can easily point to the objects of interest (e.g., clicking on a data point) and apply actions made visible on the screen in a coherent fashion [143]. However, such interaction can be inefficient when the interface requires many steps to complete a task [70] or when mapping high-level concepts (such as *popularity*) into concrete data attributes or selecting data attributes from man choices [125]. A multi-modal NLIDB combines the power and flexibility of natural language along with the ease of use and support of exploration of direct manipulation. Such systems hold the promise of providing users an easier and more expressive means for obtaining insights from the underlying databases [70].

With the increasing demand to democratize data analysis along with the rapid growth in computation power and technology, many commercial systems (Chapter 1.2) support multi-modal NLIDBs in some form by allowing users to query the system using both natural language questions and UI actions. Not surprisingly, in the research community, multi-modal conversational NLIDBs, while still at its infancy, also start to receive increasing attention.

Most existing research focus on augmenting direct manipulation in visual analytics with natural language to overcome aforementioned limitations of direct manipulation[286]. In addition, natural language interfaces help in building a more inclusive technology, for example, by better supporting blind and low vision people [286]. For instance, Figure 7.26 displays an interactive UI with a day of patient measures for a time series database. As shown in Figure 7.27, direct manipulations over such a UI via mouse clicks allow a user to easily focus on specific events and obtain basic information about the events, while natural language questions allow the user to ask more complex questions that require multiple mouse clicks (e.g., Q1 and Q2) or are hard to ask via direct manipulations (e.g., "*How often does the patient exercise?*").

7.6.1 Conversational Transition Modeling

One way to support multi-modal conversations is to build systems based on pragmatics principles. The most influential conversational transitions model [315] (Figure 7.28) is built upon conversational centering theory [127], visualization reference model [39] and insights gained via user studies [315].

In this model, based on conversational centering theory, utterances are divided into constituent conversational segments, embedding relationships that may hold between segments to maintain coherence. A center refers to those entities serving to link that utterance (corresponding to visualization components) to other utterances in the conversation. Three types of transitions on how the users expect the visualization to change are: **Continue** refers to a transition that continues the context from the previous questions to the new question, while potentially adding new entities; **Retain** refers to a transition that retains the context from some previous questions in the new one without adding additional entities; and **Shift** refers to transition shifts or changes in context from the previous one. These three types of transitions directly correspond to the three types of user question-to-question transitions discussed earlier in Chapter 7.5.2, where *continue* corresponds to *topic extension*, *retain* to *topic exploration*, and *shift* to *topic shift*.

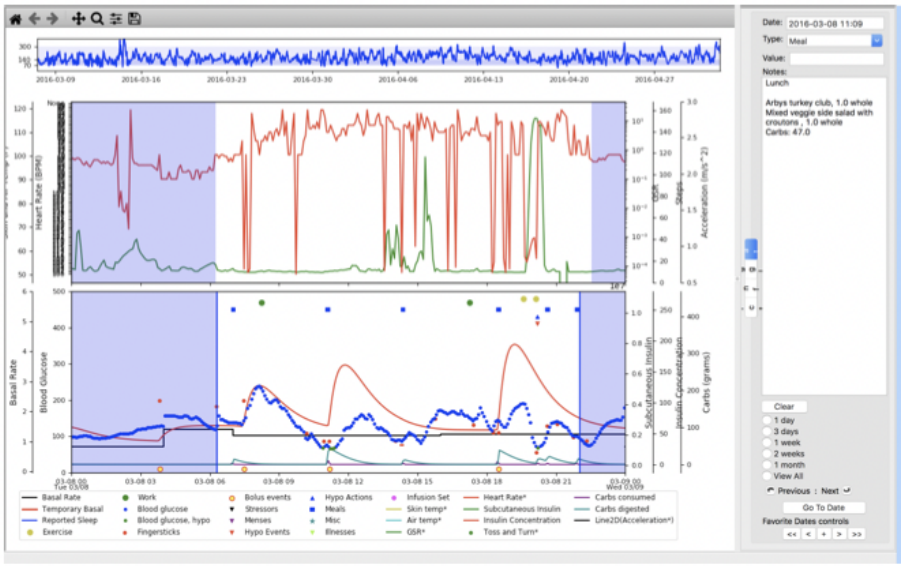


Fig. 7.26: Example GUI displaying a day of patient measures for a time series database (image source: [48])

Click on Exercise event at 9:29am.
$Click(e) \wedge e.type = \text{Exercise} \wedge e.time = \text{9:29am}$

Click on Miscellaneous event at 9:50am
$Click(e) \wedge e.type = \text{Misc} \wedge e.time = \text{9:50am}$

Q_1: What was she doing mid afternoon when her heart rate went up?

$$Answer(e) \wedge Behavior(e_1.value, \text{Up}) \wedge Around(e.time, e_1.time) \wedge e.type == DiscreteType \wedge e1.type == HeartRate \wedge e1.time == MidAfternoon()$$

Q_2: What time did that start?
$Answer(e(-1).time)$

Fig. 7.27 Example multi-modal conversation (image source: [48]).

Transitions in utterances directly impact visualization states, including the data attributes in play, transformations (e.g., computing derived attributes), filters, and the visual encoding of attributes, as defined in visualization reference model [39]. Users' intent expressed in utterances may impact any or all aspects of a visualization state. Some studies [315] have found that maintaining coherence in a visual encoding is important, as abrupt changes to the visual representation can be disruptive and confusing. However, the same studies also suggest that analytical intent may conflict with the goal to maintain visual encoding coherence; in such a case, analytical intent should take priority potentially at the cost of visual encoding coherence. For instance, Figure 7.29 shows an example analytical conversation: the results for the initial utterance is shown in a horizontal bar chart; then given the follow-up question, the system keeps the same visualization and only adds a new column and color encoding; however, when the next question completely shifts the topic and asks for correlation, a heatmap visualization is used to better support the analytic intent.

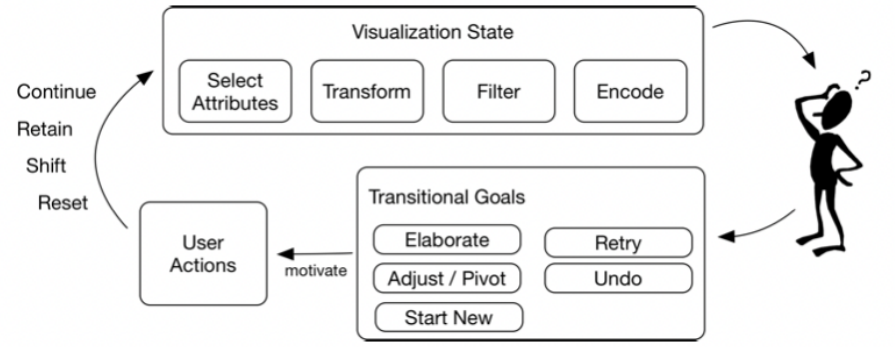
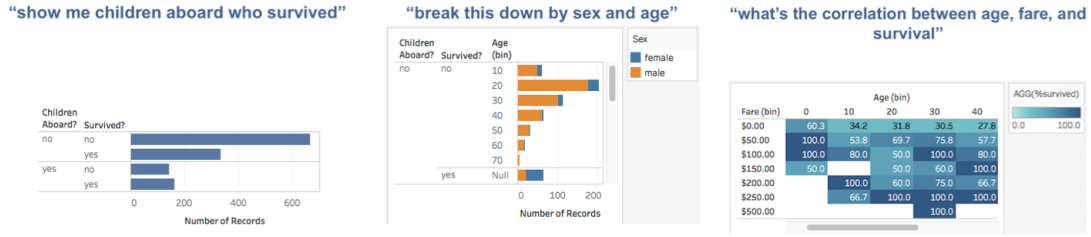


Fig. 7.28: Conversational Transition Model (image source: [315])



(a) initial utterance (b) implicit intent to retain context (c) implicit intent asking for a correlation

Fig. 7.29: Example Analytical Conversation (image source: [315])

As illustrate by Figure 7.28, after examining a visualization, a user may wish to transform an existing visualization to answer a new question, including: **elaborate** (add new data to the visualization), **adjust / pivot** (adapt aspects of the visualization), **start new** (create an altogether new visualization), **retry** (re-attempt a previous failed step), and **undo** (return to the prior state). Different transitional goals in turn drive user actions in the form of utterances and direct manipulations.

Both Evizeon [143] and Orko [297] rely on this model to decide how to transition a visualization state during an analytical conversation, but focusing on filter only for visualization state. Analyza [90] implicitly follows the same model as well. By retaining the context and answer of the previous question, it supports follow up questions with substitution of a single element of the previous question as well as questions about results shown in the answer. In addition, it provides a UI to enable users to explore similar queries. In all three systems, the interpretation of the questions relies on grammar-based parsers. The systems also gives feedback for questions it cannot fully interpret and suggest possible ways to reformulate the questions.

7.6.1.1 Template-Driven

Conversational NLIDBs can also be built based on a template-driven approach. In this approach, both the conversational flow and the interpretation of individual utterance can be based on templates, which can be either manually defined or automatically generated.

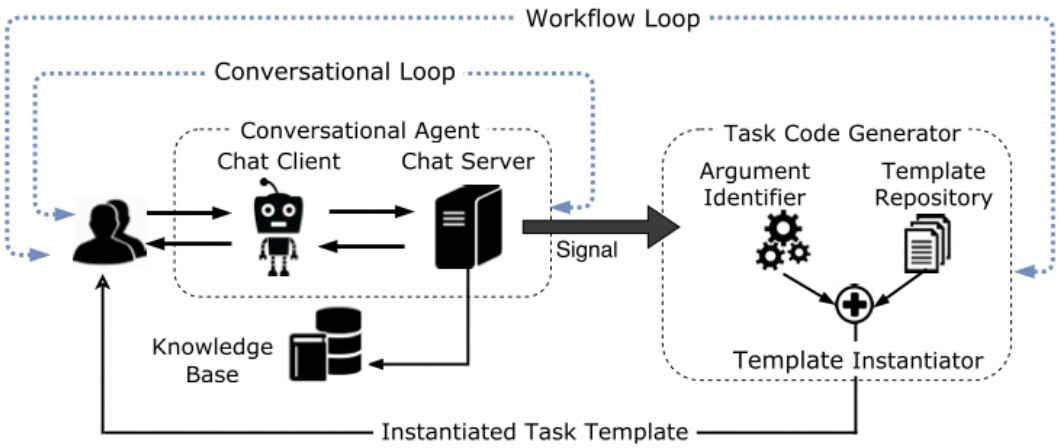


Fig. 7.30: Template-driven control flow (image source: [159])

Figure 7.30 shows the conversational data science workflow in Ava [159]. In this system, a controlled natural language in the form of ⟨ACTION VERB⟩ ⟨NOUN⟩ ⟨PREDICATE⟩ is used to interact with the underlying data. The conversational flow is driven by templated sequence of logical stages, each consisting of templatized tasks. Ava uses a state machine to represent the conversational context, including: (1) The current state of the conversation (e.g., stage of the workflow, dataset and model variables); (2) The allowable inputs to trigger a state transition (e.g., what can the user input to select a task or move to the next stage in the workflow); and (3) The corresponding actions that Ava must take based on the user's input (e.g., what code to execute when the user requests to split the data).

GeCoAgent [73] follows the same approach and defines a high-level finite state automaton to capture the high-level workflow as well as the individual tasks for genomic data extraction and analysis. The conversation flow is guided by the automata, where based on the exact state of the system and the state diagram, the system proactively solicits user inputs, infers the user's intent, and provides suggestions based on past executions. For intent understanding, GeCoAgent uses a model trained on synthetic data generated using templates. Data visualization is supported both for displaying data analysis results and for simple data exploration.

Template-driven approach requires an in-depth and comprehensive understanding of data workflow and common tasks involved. It is particularly suitable for building domain-specific conversational NLIDBs where such understanding is possible and the variety of questions to support is limited.

7.6.1.2 Supervised Learning

Another way to support multi-modal conversations is to train semantic parsers to take context, including both past utterances and direct UI manipulations, into account with supervised learning. This approach faces two major challenges: (1) a model architecture that can efficiently and effectively capture the multi-modal context; and (2) sufficient high-quality training data. For instance, a recent work [48] designs an LSTM-based encoder-decoder architecture to model context dependency through a copying mechanism and multiple levels of attention over input and previous output. In addition, it proposes a synthetic data generator

that simulates user-GUI interactions, with sentence templates defining the skeleton of each entry in order to maintain high-quality sentence structure and grammar. Techniques discussed earlier in Chapter 7.5 are also applicable.

7.6.2 Conversation via Query Suggestion

Conversational-style interactions with a multi-modal NLIDB is also possible via query suggestion. The basic idea is to guide users with question suggestions in natural language on what to ask next while supporting interactive data analysis using natural language.

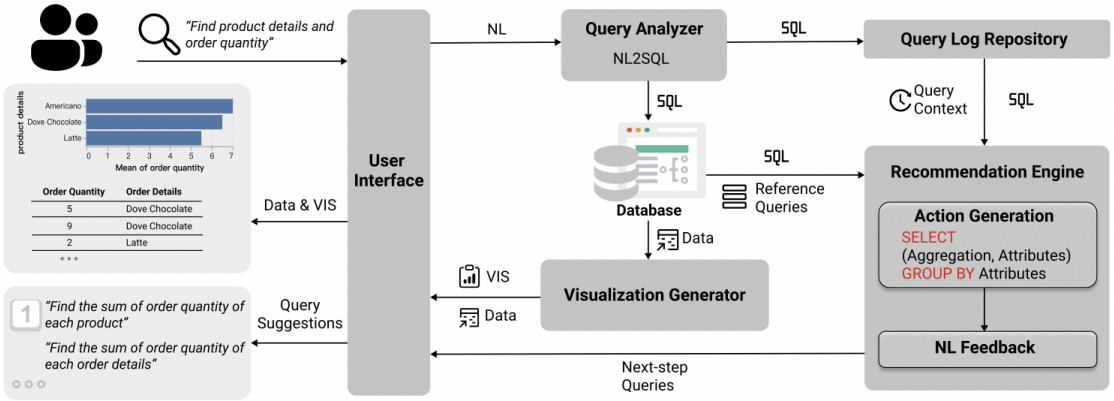


Fig. 7.31: Example Conversational NLIDB via Query Suggestion (image source: [331])

Figure 7.31 illustrates an example conversational NLIDB via query suggestion. As can be seen, such a system typically consists of two major components:

Query Parsing — This component functions the same as in a non-conversational NLIDB and handles the understanding and translation of individual utterances on their own.

Query Suggestions — This component, also known as Query Recommendation, suggests possible questions in natural language to the users. The suggestions can be follow-up queries to the current question with topic extension or exploration; they can also be questions shifted to new topics. As discussed earlier in Chapter 7.2.2, the suggestions can be automatically generated based on a combination of underlying data and user interaction history (e.g., in Snowy [296]) or based on a data-driven approach to select semantically relevant and context-aware queries from query log (e.g., in QRec-NLI [331]).

Supporting conversational NLIDBs with query suggestion has two advantages: first, the suggested questions can guide users in their analytic workflows and help them overcome the challenges of deciding what to ask next; second, the suggested questions may always be correctly interpreted, eliminating possible confusion caused by query parsing errors and serve as useful examples for users to better understand how to formulate queries that could be correctly handled.

Prior study [143] has found that while multi-modal conversational systems have made it easier for novice data analysts to construct visualizations, users can feel lost in the conversation regarding what questions can

be asked. Supporting conversations via query suggestion is an important step to overcome such a "what's next" challenge.

7.6.3 Discussions

Multi-modal conversational NLIDB is an active research area with two major open challenges. The first challenge is the richness of the design space. Designing a multi-modal conversational NLIDB requires taking user preferences in all modalities into consideration. User preferences can be data specific and task-specific. For example, prior crowdsourcing studies [281] find that for autocompletion, users prefer visualization widget-based autocompletion for numerical, geospatial, and temporal data while textual autocompletion for hierarchical and categorical data. Similarly, users may prefer different visualizations for the same data [359]. Much research is still required to better understand such preferences to inform the different design decisions for multi-modal conversational NLIDBs. The second challenge is the needs for personalization. Prior work [73, 134] found that different users have different communication preferences with regard to conversation styles and content. For instance, an earlier study [134] reports significant discrepancy in preferences among their participants when asking questions about comparisons and trends via a conversational interface: while most participants prefer data visualization, about 40% of the participants prefer not to see data visualization at all. Clearly, it is important to understand such communication preferences via user studies and model such preferences properly through personalization.

One direction to address the above challenges is to focus on building complementarity-based multimodal interaction [67] where individual modalities are used in a complementary fashion to enable a seamless user experience. The basic idea is to interweave visualizations, potentially multiple coordinated visualizations with pen, touch, and speech-based multimodal interaction for data exploration, while at the same time support natural language questions during visual data exploration. Recent work such as DataBreeze [295] and MIVA [62] demonstrate a great promise of enabling more fluid and natural interaction with data towards this direction.

7.7 Summary

Interactivity is important for the effectiveness and usefulness of NLIDBs. In this chapter, we have introduced two major categories of common interactive techniques in NLIDBs. The first category includes those improving the communication between human and the underlying NLIDBs, such as disambiguation and auto-completion; the other category focuses on those assisting users with exploratory data analysis such as automatic data insights and query suggestion. We have also summarized common explanation techniques to explain key aspects of NLIDBs. Finally, we have reviewed conversational NLIDBs, including the underlying discourse structure, different types of conversational NLIDBs, the associated open challenges, and the opportunities with LLMs.

7.8 Further Reading

Ambiguity in natural language is a major factor when designing interactivity for NLIDBs. An earlier book [141] provides a comprehensive review of common causes of ambiguity and classic approaches towards disambiguation. A recent survey [142] reviews more recent work on spell correction under a unified framework.

Visual modality is important in interactivity. We refer the readers to Shen et al. [286] for a comprehensive review of existing Visualization-oriented Natural Language Interfaces (V-NLI) and to Law et al. [182] and Lee et al. [184] for more references on automated data insights, particularly visual insights and visualization recommendation.

Explainability for natural language processing is an emerging topic in recent years. For the interested reader, systematic reviews of the state-of-the-art literature on explainability for natural language processing, including human-centered evaluations of explanations, can be found in recent publications [27, 81, 82, 293].

Finally, when designing conversational NLIDBs, it is important to consider the fact that different users have different communication preferences with regard to different conversation styles and content. We refer the readers to previous study [134] on such communication preferences.