Chapter 2
Building an NLIDB: The Basics

The main purpose of this chapter is to help readers to form a high-level understanding of NLIDBs and to better understand and leverage the techniques and approaches to be introduced in the rest of the book. We first describe an example database and example input questions against the database in Chapter 2.1. We then introduce the common architecture of a baseline NLIDB before introducing terminologies to be used throughout the rest of the book (Chapter 2.2). Finally, we describe a basic step-by-step process to construct such a baseline NLIDB for the example database to handle the example question in Chapter 2.3.

2.1 Example Database

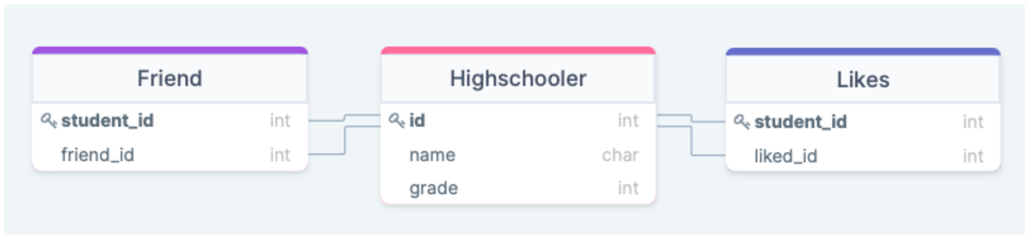


Fig. 2.1: Example Database

Consider a database with schema illustrated in Figure 2.1. This simple database includes only three tables, each with two to three columns. Even for such a simple database, one can ask wide varieties of questions about individual high school students or groups of them as well as their relationships with each other. Table 2.1[1] lists a few possible input questions against the database and the corresponding SQL queries that can retrieve the correct output from the database. As can be seen, although these questions are of similar length, the complexity of the corresponding SQL statements varies greatly. In the rest of the chapter, we describe the steps of building a basic NLIDB that takes such questions as input and translates them into SQL

[1] The database itself and most of the example input questions are from the SPIDER dataset [373].

queries over the database. We will also discuss key challenges in building such an NLIDB with additional examples.

Table 2.1: Example input questions and corresponding SQL queries.

ID	Input Question	SQL Query
Q1	Show the name and grade of each high schooler	`SELECT name, grade` `FROM Highschooler`
Q2	What are the names of all high schoolers in grade 10?	`SELECT name` `FROM Highschooler` `WHERE grade = 10`
Q3	How many friends does Kyle have?	`SELECT count(*)` `FROM Highschooler AS T1` `JOIN Friend AS T2` `ON T1.student_id = T2.id` `WHERE T1.name = "Kyle"`
Q4	What is the name of the high schooler who has the most friends?	`SELECT T1.name` `FROM Highschooler AS T1` `JOIN Friend AS T2` `ON T1.student_id = T2.id` `GROUP BY T2.student_id` `ORDER BY count(*)` `DESC LIMIT 1`
Q5	Which highschoolers have more than five friends?	`SELECT T1.name` `FROM Highschooler AS T1` `JOIN Friend AS T2` `ON T1.id = T2.student_id` `GROUP BY T1.id` `HAVING count(*) > 5`

2.2 Anatomy of NLIDB

Figure 2.2 depicts the key components of a basic pipeline-based NLIDB and how they connect with each other [198]. As can be seen, the core of an NLIDB consists of three major components: The first component, **Query Understanding**, translates a given question in natural language into an internal logic representation, often known as a **query interpretation**. The second component, **Query Translation**, translates each query interpretation into the corresponding structured query that can be then executed against the underlying data store. Finally the last component, **Results Display**, returns the results of the executed query to the user. In later part of this book, we will introduce recent approaches, where the key components, particularly Query Understanding and Query Translation, are built into a one single end-to-end model.

In addition to the aforementioned three core components, a real-world NLIDB often includes additional components. First of all, both Query Understanding and Query Translation may rely on **External Knowledge** to better interpret the input questions. External knowledge may be automatically constructed based on the underlying database or external sources (e.g., DBPedia [185]). External knowledge may also be given directly by a user, such as synonym dictionaries. Secondly, the NLIDB may also include **Interaction Generation** to generate explanations for its answers as well as to obtain feedback from the user. Explanations are usually

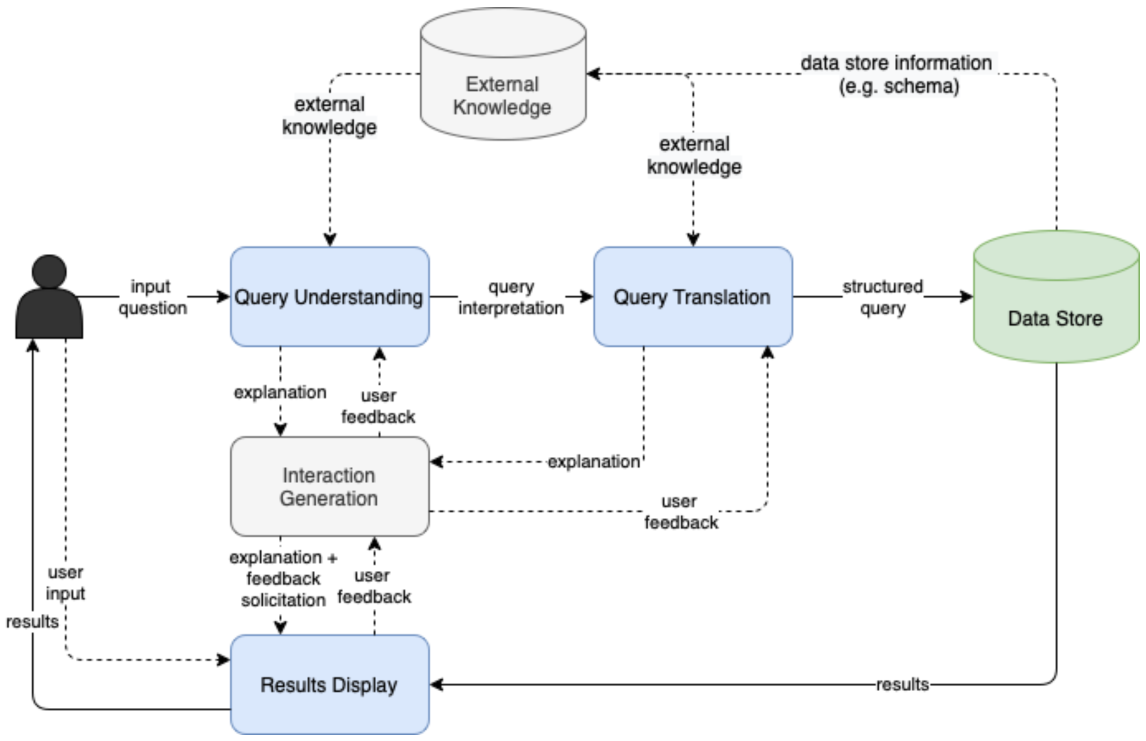


Fig. 2.2: Anatomy of a conventional pipeline-based natural language interface to databases.

displayed along with results. User feedback can be implicit (e.g., inducted based on user behavior such as query history) or explicit (e.g., provided by user input via UI interactions).

2.3 Building an NLIDB

In this section, we describe step by step how to build the aforementioned key components in a pipeline-based NLIDB. Later on in Chapter 4, we will discuss how to build the capabilities of the key components into one single end-to-end model. In real-world settings, an NLIDB may take a combination of both approaches to best balance multiple factors such as quality and runtime efficiency.

2.3.1 Query Understanding

One simple yet effective approach is to build Query Understanding on top of more basic analysis of the syntactic structure of sentences, such as **part-of-speech tagging** and **dependency parsing**. Figures 2.3 to 2.6 show the dependency parse trees of the example input questions from Table 2.1 along with the POS tags for individual parse trees. We now describe in details how to leverage such syntactic structure to generate query interpretation.

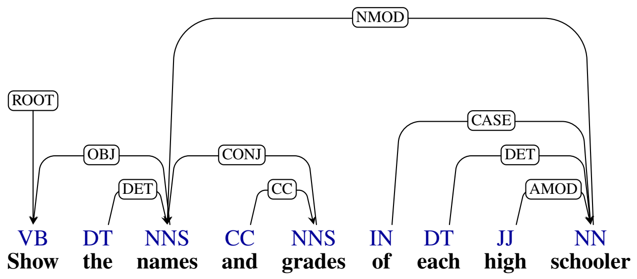

NMOD
ROOT
OBJ
CONJ
CASE
DET
DET
CC
AMOD
VB DT NNS CC NNS IN DT JJ NN
Show the names and grades of each high schooler

Fig. 2.3 Dependency Parse Tree for Example Input Question Q1

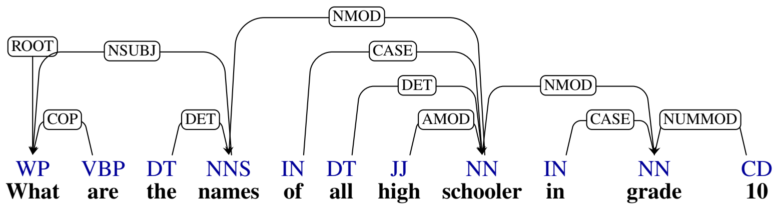

NMOD
ROOT
NSUBJ
CASE
DET
NMOD
COP
DET
AMOD
CASE
NUMMOD
WP VBP DT NNS IN DT JJ NN IN NN CD
What are the names of all high schooler in grade 10

Fig. 2.4 Dependency Parse Tree for Example Input Question Q2

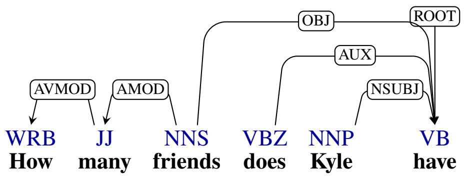


Fig. 2.5 Dependency Parse Tree for Example Input Question Q3

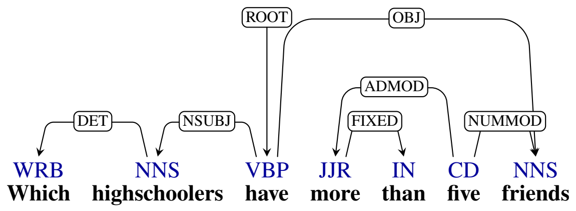

ROOT
OBJ
ADMOD
DET
NSUBJ
FIXED
NUMMOD
WRB NNS VBP JJR IN CD NNS
Which highschoolers have more than five friends

Fig. 2.6 Dependency Parse Tree for Example Input Question Q5

Table 2.2: Types of Parse Tree Nodes.

Node Type	Corresponding SQL Component
Select Node (SN)	SQL keyword: `SELECT`
Operator Node (ON)	a SQL operator, e.g., =, <=, `!=`, `contains`
Function Node (FN)	an aggregation function, e.g., `AVG`, `MAX`
Name Node (NN)	a relation name or attribute name
Value Node (VN)	an attribute value
Quantifier Node (QN)	`ALL`, `ANY`
Logic Node (LN)	`AND`, `OR`, `NOT`

Table 2.3: Example Mapping Rules.

Node Type	Example Mapping Rules
Select Node	*select*, *show*, *what* ⇒ `SELECT`
Operator Node	*equal* ⇒ =, *less than* ⇒ <, *more than* ⇒ >
Function Node	*average* ⇒ `AVG`, *the greatest number of*, *most* ⇒ `MAX`
Quantifier Node	*all* ⇒ `ALL`, *any* ⇒ `ANY`
Logic Node	*and* ⇒ `AND`, *or* ⇒ `OR`, not ⇒ `NOT`

2.3.1.1 Parse Tree Node Classification

The first step is to identify the parse tree nodes that can be mapped to query components and to categorize them into different types, based on the corresponding SQL components that they can be mapped into. Table 2.2 summarizes the common types of parse tree nodes [187, 199, 251] and the corresponding SQL components that they may map into.

The type of a parse tree node highly depends on its part of speech (POS) tag. In general, common nouns (e.g., NN, NNS) are usually Name nodes; proper names (e.g., NNP, NNPS) are usually Value nodes; determiners (DET) are often Quantifier nodes; adjectives (JJ) are often Quantifier nodes; and verbs (e.g., VB) are often Select nodes or Function nodes.

2.3.1.2 Parse Tree Node Mapping

The second step is to map all parse tree nodes into the corresponding database artifacts. The result of such a mapping is called a **query tree**, a specific type of query interpretation. The mapping process for all types of nodes, excluding Name nodes and Value nodes, is typically independent of the underlying database and relies on simple mapping rules, such as the ones in Table 2.3.

The mapping of Name and Value nodes can be more complex. Not only the mapping depends on the underlying database itself, it often needs to take the entire query into consideration and sometimes requires user input to resolve possible ambiguity or to bridge the semantic gap between the input question and the underlying databases. For instance, if the user question asks "*in which grade is John?*" over our example database in Figure 2.1 and there exists multiple students with first name "John", then the system may return information for all students or request user input to decide which student is of interest.

One way to facilitate the mapping of Name and Value nodes is to build a **translation index** [22]. A translation index is essentially a semantic index of database artifacts, including names of relations and attributes as well as actual values. It generates variants for each database artifacts, and maintains a map that allows reverse look-up for the values. The generation of variants can be based on a combination of simple mapping dictionaries, domain-specific ontologies [186], and more sophisticated automatic variant generation [23, 260].

Table 2.4 illustrates a fragment of a possible translation index corresponding to our example database in Figure 2.1. Variants for relation *Highschooler* are based on predefined mapping dictionaries, variants for the values of attribute *Highschooler.grade* come from an education domain ontology, and variants for the values of attribute *Highschooler.name* are based on automatic variant generation for Person type. Such a translation index helps bridging the semantic gap between vocabularies in the input questions and the underlying database. In addition, each variant is also often associated with a confidence level, indicating how likely the mapping is correct. The system can then leverage such information to resolve ambiguity

Table 2.4: Example Translation Index Snippet.

Database Artifact	Variants	Confidence
Highschooler	high schooler, high school student	high
Highschooler	student	low
Highschooler.grade = 9	grade 9, 9th grader, freshman	high
Highschooler.grade = 10	grade 10, 10th grader, sophomore	high
Highschooler.grade = 11	grade 11, 11th grader, junior	high
Highschooler.grade = 12	grade 12, 12th grader, senior	high
Highschooler.name = John	John	high
Highschooler.name = John	Johnny, Jack	medium
Highschooler.name = John	J.	low

when multiple interpretations are possible. For instance, given the input question “*List all names for all high school students?*”, based on the translation index, we can map “*high school student*” into Highschooler with *high* confidence and “*student*” into Highschooler with *low* confidence. In such a case, we will rank the former mapping higher than the latter.

Figures 2.7, 2.8 and 2.9 illustrate possible parse tree node mappings for questions Q1, Q2 and Q3 from Table 2.1 respectively. We keep the dependency relations for the corresponding parse tree nodes for the next step — Query Translation. For simplicity, we remove all parse tree nodes with no mapping to a database artifact, except for any root node.

Besides using mapping rules and translation index, more complex keyword mapping can be built, for instance, based on word embedding and deep neural work [318]. We will describe such techniques in more details later in Chapter 4.

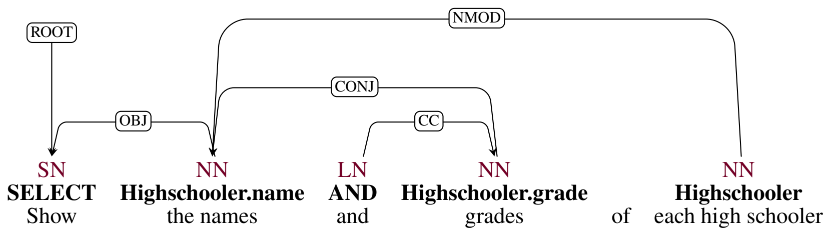


Fig. 2.7: Query Tree for Example Input Question Q1

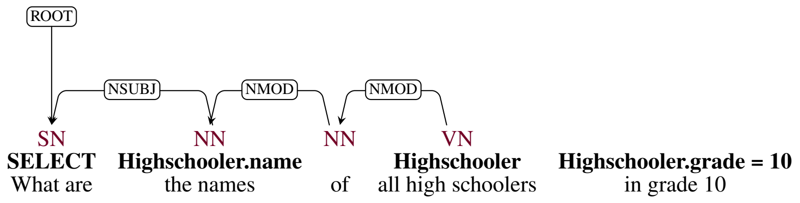


Fig. 2.8: Query Tree for Example Input Question Q2

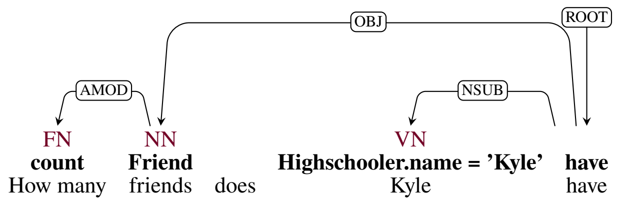


Fig. 2.9 Query Tree for Example Input Question Q3

2.3.1.3 Query Tree Reformulation

For input questions corresponding to simple selection queries, the two steps described above is sufficient. In some cases, however, to facilitate Query Translation, we may need to adjust the query tree structure to account for (1) potential parse tree mistakes; and (2) omissions in the original input questions to facilitate query translation, as previously proposed [188, 251].

For example, Figure 2.10 illustrates the query tree corresponding to Q5 in Table 2.1. In the original question, the expression '*more than 5 friends*' is a more concise and arguably more natural way to express the semantics of '*the number of friends is more than 5*.' Since the expression describes the aggregation implicitly, the query sub-tree corresponding to the former expression contains no function node. As a result, a direct translation of the query tree into a SQL clause will lead to the wrong omission of an aggregation function. One way to address this issue is to reformulate the query tree to add the omitted function node count back as well as move the nodes so that the name node appears before the operator node, resulting in the adjusted query tree in Figure 2.11.

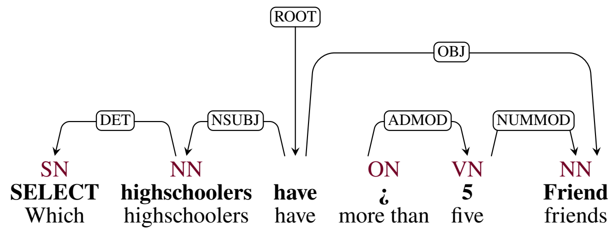


Fig. 2.10 Query Tree for Example Input Question Q5

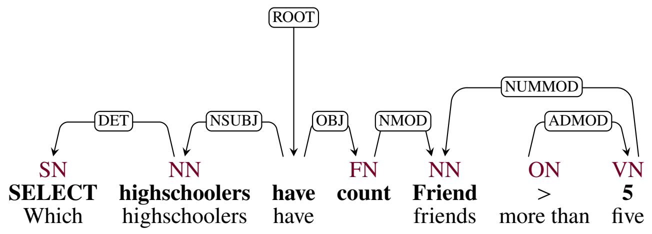


Fig. 2.11 Adjusted Query Tree for Example Input Question Q5

2.3.2 Query Translation

Query Translation operates on top of a query tree and generate a final executable structured query. Based on the query tree, the query type of the final executable structured query is detected and a different query translation process may be applied to each type.

2.3.2.1 Simple SELECT Query

When the query tree contains neither function node nor quantifier node, the corresponding structured query will be a simple SELECT query without an aggregation function or a subquery. The translation for such a query is straightforward using the following steps.

SELECT Clause	Add every Name node (NN) corresponding to an attribute name under the Select node to the `SELECT` clause.
FROM Clause	Add every Name node (NN) corresponding to a relation name to the `FROM` clause.
WHERE Clause	For every Operator node (ON), add a predicate to the `WHERE` clause based on the Name node - Value node pair attached to the node. For every Value node (VN) not connected with an Operator node (ON), add a predicate with a "=" function.
JOIN Clause	If there are more than one relation added to the `FROM` clause, add a `JOIN` clause for each pairs of *foreign key* - *primary key* between the relations based on the database schema (see Section 3.2 for definitions of primary key and foreign key). This step is also referred to as **join path inference**.

Based on the above process, we can obtain a correct SQL statements for the input questions Q1 and Q2 in Table 2.1 from query trees such as the ones shown in Figures 2.7 and 2.8.

2.3.2.2 Simple Aggregate Query

A simple aggregate query is a query that contains one or more aggregate functions but no subquery. When the query tree contains only one function node and no quantifier node, the corresponding query is usually a simple aggregate query. Both Q3 and Q4 in Figure 2.1 are simple aggregate queries. The translation for simple aggregate queries is slightly more complex than for simple SELECT queries.

SELECT Clause	Add every Name node (NN) not attached by a Function node (FN) and corresponding to an attribute name under the Select node to the `SELECT` clause; if the path of the only Function node (FN) to the root contains only one Name node, then add the corresponding SQL function to the `SELECT` clause.
FROM Clause	Add every Name node (NN) corresponding to a relation name to the `FROM` clause.
WHERE Clause	For every Operator node (ON), add a predicate to the `WHERE` clause based on the Name node - Value node pair attached to the node. For every Value node (VN) not connected with an Operator node (ON), add a predicate with a "=" function.
JOIN Clause	If there are more than one relation added to the `FROM` clause, we need to add `JOIN` clause(s) for the relations. Unless the question explicitly defines the join condition(s), we examine the path between the relations based on the database schema and add `JOIN` clause(s) accordingly (see Table 2.1 Q4 as an example). This step also referred to as **join path inference**.

ORDER BY Clause If the Function node (FN) corresponds to `max` or `min`, then add `ORDER BY count(*) DESC LIMIT 1` for `max`, and `ORDER BY count(*) ASC LIMIT 1` for `mix`.
HAVING Clause If the Function node (FN) is attached to an Operator Node (ON), then add the corresponding predicate to the `HAVING` clause.

2.3.3 External Knowledge

To properly perform Query Understanding, an NLIDB needs to both understand the underlying databases as well as be aware of the common knowledge its users may have with regard to the domain corresponding to the underlying databases.

External knowledge relevant to NLIDBs includes (1) linguistic knowledge such as WordNet [223], a lexical database for English, and FrameNet [17] that provides semantic frames for words, (2) world knowledge that provides explicit knowledge about specific instances of entities, such as what provided by YAGO [299], Freebase [25], DBPedia [185] and Wikidata [324] (e.g., "The capital of the United States is Washington, D.C."), (3) commonsense knowledge such as ConceptNet [294] that describes implicit general facts (e.g., "*A house typically has a window*"), and (4) domain knowledge that consists of facts about a specific domain, also referred to as domain-specific ontologies, such as International Classification of Diseases [339] code for the biomedical domain and Financial Industry Business Ontology [19] for the financial domain.

The translation index introduced in Chapter 2.3.1 is one simple way to capture external knowledge. We can construct translation indices based on underlying databases and external resources. For example, in the translation index shown in Table 2.4, for "Highschooler.grade = 9", the variants "grade 9" and "9th grader" can be automatically generated based on domain knowledge and underlying database schema [259], while the variant "freshman" needs to be added explicitly via a mapping dictionary. Such external knowledge is the key to enable the NLIDB to handle questions with world knowledge not available in the underlying database, such as the concept of "freshmen" in "*Who are all the freshmen?*"

Later on in Chapter 7.1 we will discuss techniques to solve the ambiguity problem in NLIDBs, often by leveraging external knowledge. We can incorporate external knowledge as a separate component such as a translation index or as part of a machine learning model that takes external knowledge into account in a more sophisticated fashion.

2.3.4 Interaction Generation

In an ideal world, an NLIDB is able to handle arbitrary input questions and translate them into the corresponding structured queries correctly. In reality, however, an NLIDB often encounters input questions that it cannot handle. In such a case, as with many human-centered AI systems, it is desirable for the system to explain to the user what it can and cannot understand and actively solicit user feedback that may be given explicitly or implicitly.

Spell correction and query auto-completion are common techniques to reduce ambiguous and erroneous questions. In addition, one common challenge that often leads to the failure of an NLIDB is the lack of domain-specific knowledge, as the external knowledge ingested into the NLIDB in advance may not suffice. For instance, the example NLIDB described so far is not able to handle the input question "*Who is the most popular high school student?*," since it fails to categorize and map the corresponding parse tree nodes for "*the most popular*". In such a case, it can invoke Interaction Generation to inform the user that it does not understand the term "*the most popular*". It can then ask the user to reformulate the question into one that expresses the semantics of "*the most popular*" more explicitly such as "*Who is senior with the most number of friends?*" or "*Who is senior liked by the most number of students?*" Each query reformulation may be added into its External Knowledge for better handling of future queries. Chapter 7 discusses such interaction techniques for query disambiguation and query suggestion in more details.

In addition, even when the NLIDB has successfully understood and translated an input question, the system may support additional interactions for the user with the returned results, as detailed next.

2.3.5 Result Generation

Once a structured query is successfully obtained, the system sends the query to the data store that then executes the query and returns the execution results back. Results Generation then returns the results to the user with potentially additional information, depending on the execution results.

2.3.5.1 Non-Empty Result

If the execution result of a query is not empty, Results Generation can return the resulting rows directly to the user. In addition, as described earlier in Chapter 1.2, it may support additional interactivity to improve the usability of the system such as follows:

Result Description	For a more natural interaction experience and to help users better comprehend the answer, a natural language description of the results may be returned instead of a raw output in tabular form.
Result Visualization	The result may be visualized to help the user better understand and interact with the query answer.
Result Explanation	An explanation of how the results are obtained may be provided to help the user better understand and trust the returned results. The most basic explanation for an input question can be generated by highlighting keywords in the original input questions that are mapped into database artifacts or non-trivial query fragment.
Result Exploration	Additional user interactions such as sorting and drilling down with the query results may be provided to allow the user further explore the query results without the needs to issue additional queries.

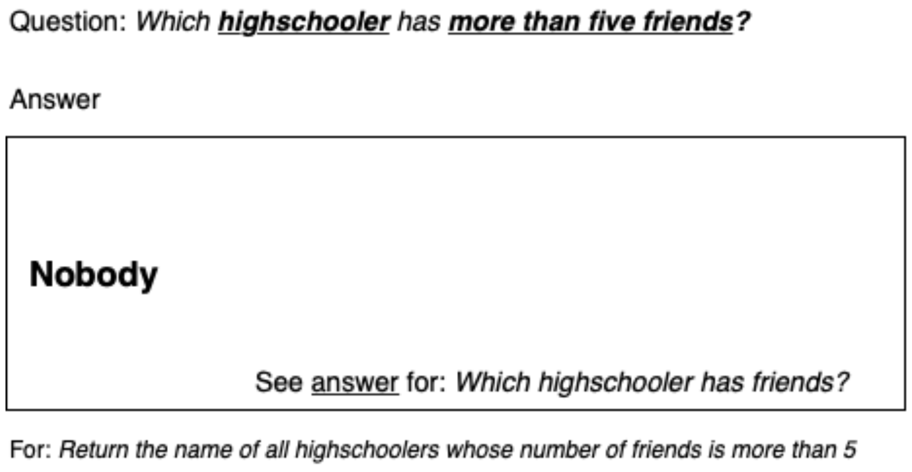
In Chapter 6, we will provide a comprehensive review of techniques to generate text from data, often used for result description. In Chapter 7, we will discuss techniques related to result visualization, explanation and exploration in more details.

2.3.5.2 Empty Result

If the execution result is empty, it is important to explain to the user the possible reasons behind the empty result. One possibility of empty result is that the translated query from input question is wrong. By leveraging the explanation techniques described earlier, the user can examine whether the query understanding by the system correctly reflects the semantics of the original input question. The other possibility for an empty result is that the answer to the original input question is indeed empty. In such a case, the system may

Table 2.5: Relaxed Version of Q5: *Which highschoolers have friends?*

name	count(T2.student_id)
Haley	1
Alexis	2
Jordan	1
Austin	1
Tiffany	1
Kris	2
Jessica	1
Jordan	2
Logan	1
Gabriel	2
Cassanra	1
Andrew	3
Gabriel	1
Kyle	1


Question: *Which* ***highschooler*** *has* ***more than five friends?***

Answer

Nobody

See answer for: *Which highschooler has friends?*

For: *Return the name of all highschoolers whose number of friends is more than 5*

Fig. 2.12: Example Result Display Page for Q5: *Which highschoolers have friends?*

provide additional assistants to help the user to obtain a non-empty result. The system may provide query suggestions based on query history such as similar queries issued before with non-empty results. The system may also help via query relaxation, potentially based on user interactions (e.g., similar to IQR [226]).

Figure 2.12 shows an example with the different options discussed above. An explanation of query understanding is provided by highlighting the natural language description of the executed query result and an alternative query for empty results.

2.4 Summary

In this chapter, we introduce the common architecture of an NLIDB, which consists of query understanding and query translation, and present a step-by-step guide to build a basic implementation of such an NLIDB from scratch. We also discuss ideas on how to extend such a baseline implementation to handle challenges in query understanding and result display by leveraging user feedback and external knowledge. In the rest of the book, we will go into much more depth on the related topics. Specifically, in Chapter 4 we will go into great depth of various modern techniques related to query understanding and translation, including end-to-end neural models and in Chapter 6 we will present methods on natural language generation from data. We will discusses in Chapter 5 how to evaluate various aspects of NLIDBs. Finally, we review in Chapter 7 different aspects of interactivity in NLIDBs, including conversational NLIDBs, and common interactive techniques applicable to improve the effectiveness and usefulness of NLIDBs.

We refer any interested reader to an earlier book [198] with comprehensive discussions on conventional NLIDBs.