CHAPTER 2

Building an NLIDB: The Basics

The main purpose of this chapter is to help readers to form a high-level understanding of NLIDB in order to better understand and leverage different techniques and approaches for different aspects of NLIDBs to be introduced in the rest of the book

In the rest of the chapter, we first describe an example database and example input questions against the database. We then introduce the common architecture of a modern NLIDB before introducing terminologies to be used throughout the rest of the book. We then describe in details a basic process to construct a NLIDB step-by-step for the example database to handle the varieties of input question. Finally, we discuss possible practical challenges in building such an NLIDB and how to extend the baseline NLDIB to handle these challenges.

2.1 EXAMPLE DATABASE

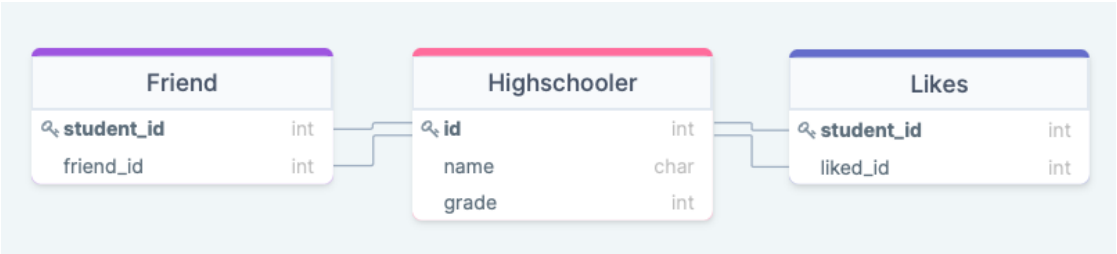

Figure 2.1: Example Database

Consider a database with schema illustrated in Figure 2.1. It is a simple database with only three tables, where each table contains only two to three columns. However, 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 others. Table 2.1 [1] lists a few examples of possible input questions against the database and the corresponding SQL queries that can retrieve the correct output

[1]The database itself and most of the example input questions are from the SPIDER dataset [73].

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 NLIDB that takes such questions as input and translates them into SQL queries over the database schema. We will also discuss related key challenges in building such a NLIDB with additional examples.

Table 2.1: Example input questions and corresponding SQL queries.

ID	Input Question	SQL Query
Q1	Show the names and grades 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 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 typical NLIDB and how they connect with each other [40]. As can be seen, the core of a NLIDB consists of three major components: The first component is **Query Understanding**, which translates a given question in natural language into an internal representation, often known as a

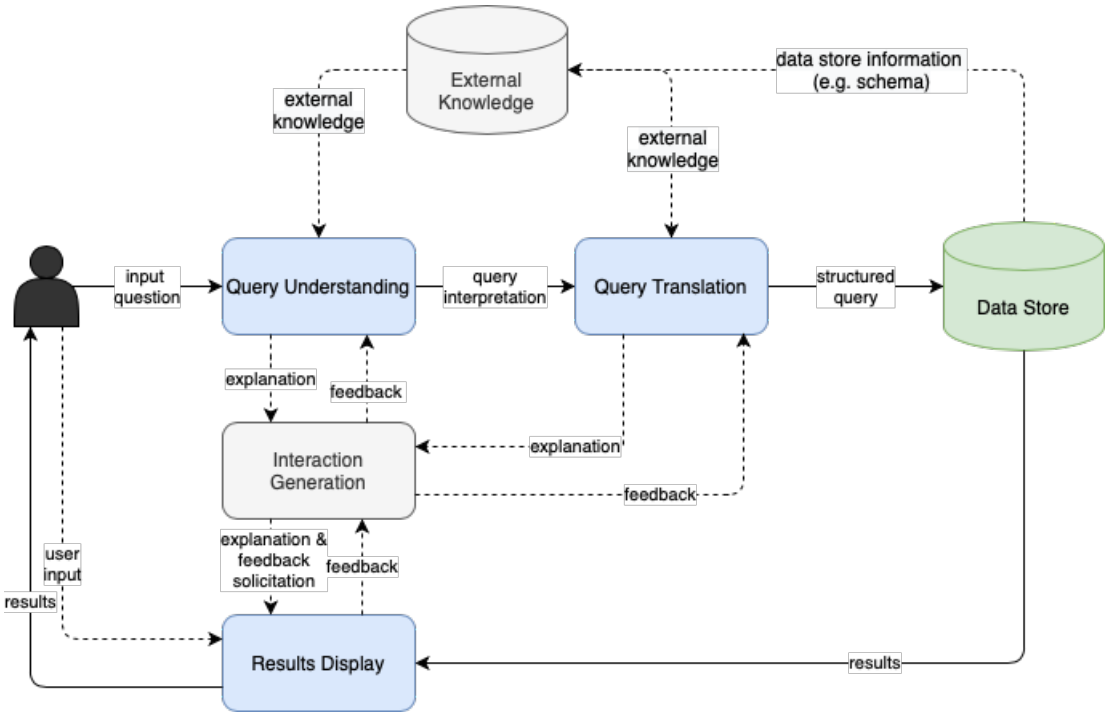

Figure 2.2: Anatomy of a natural language interface to databases.

query interpretation; The second component is **Query Translation**, which 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 addition to the three core components, a real-world NLIDB often includes additional components. In particular, both Query Understanding and Query Translation may rely on **External Knowledge**. External knowledge may be automatically constructed based on the underlying database or external sources. External knowledge may also be given directly by a user, such as synonym dictionaries. Commonly used sources for constructing external knowledge include (1) DBPedia [33] and ConceptNet [60] for general domain as common sense; (2) domain-specific ontologies such as International Classification of Diseases [48] code for the biomedical domain and Financial Industry Business Ontolog [7] for the financial domain. Furthermore, the NLIDB may also generate explanation for its answers as well as solicit feedback from the user via **Interaction Generation**. The explanation is usually displayed along with results and feedback solicitation if any. The feedback can be implicit (e.g.

inducted based on user behavior such as query history) as well as explicit (e.g. provided by user input via UI interactions).

[**Yunyao**: Add citations to DBPedia, etc]

2.3 BUILDING A NLIDB

To provide a better understanding of the details of the anatomy of NLIDB, we describe a conventional pipeline-based approach towards the building of the NLIDB [40] in this section. In later part of this book, we introduce alternative approaches such as end-to-end models, where some of the components (such as Query Understanding and Query Translation) may be built as a one single model. In the real-world, 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

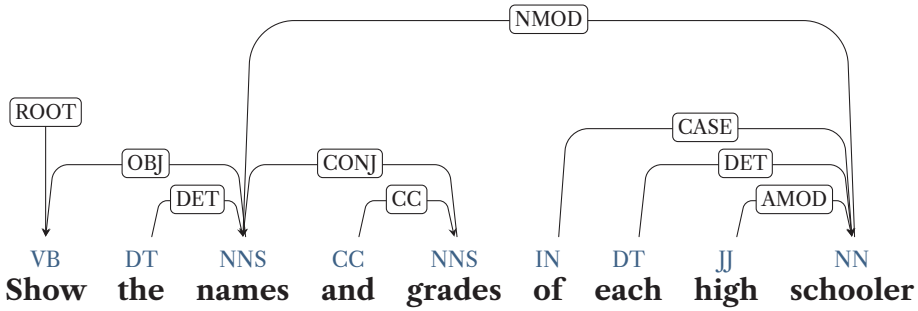

Figure 2.3: Dependency Parse Tree for Example Input Question Q1

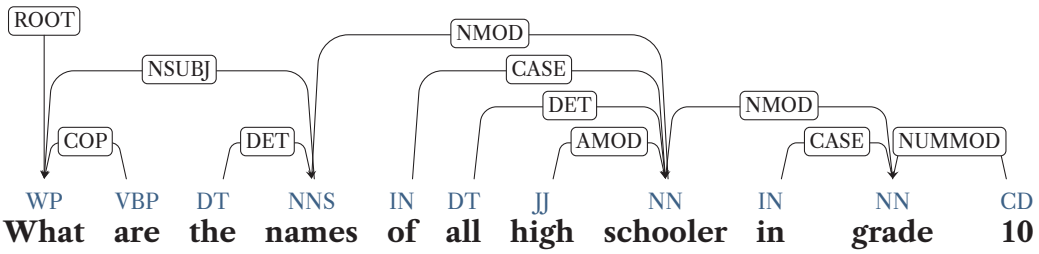

Figure 2.4: Dependency Parse Tree for Example Input Question Q2

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**

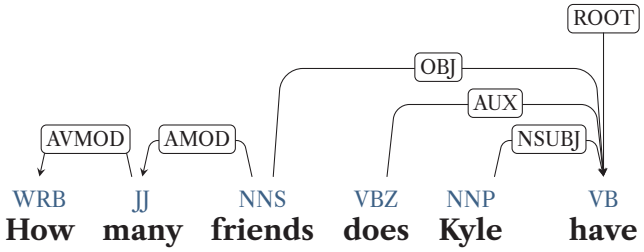

Figure 2.5: Dependency Parse Tree for Example Input Question Q3

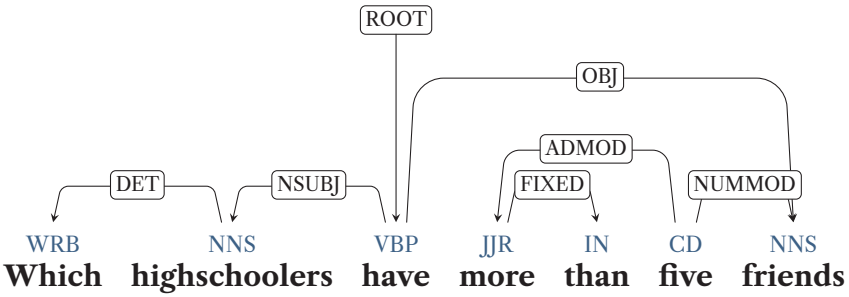

Figure 2.6: Dependency Parse Tree for Example Input Question Q5

tagging and **dependency parsing**, introduced earlier in Chapter 1.3. 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.

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`

Parse Tree Node Classification

The first step is to identify the parse tree nodes that can be mapped to query components. We can further categorize the individual parse tree nodes 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 [35, 41, 50] and the corresponding SQL components that they may map into.

The type of a parse tree node highly depends on its POS. 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 are often Select nodes or Function nodes.

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`

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 of all types of nodes, except for Name nodes and Value nodes, can usually be done independent of the underlying database based on External Knowledge in the form of 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 solicit 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** [8]. 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

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

mapping dictionaries, domain-specific ontology [34], and more sophisticated automatic variant generation [9, 54].

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 information can be to resolve ambiguity 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 based on word embedding and deep neural work, for instance [68]. We will describe such techniques in more details later.

[**Yunyao**: **To Do**: Add a link to a later section covering keyword mapping]

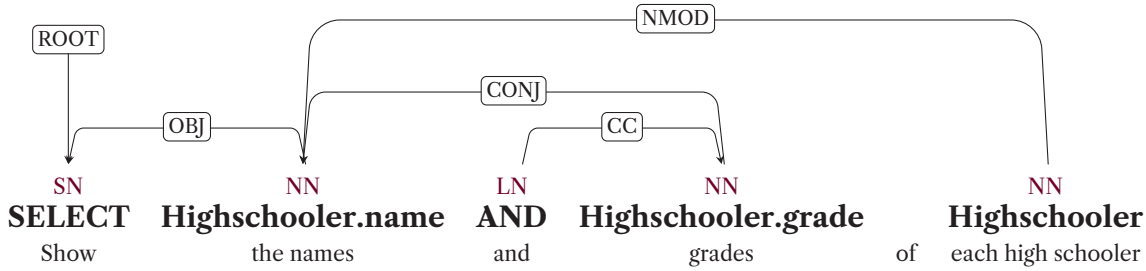

Figure 2.7: Query Tree for Example Input Question Q1

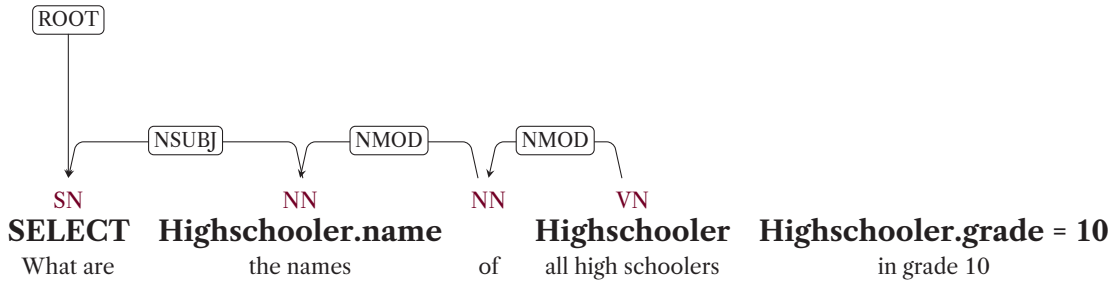

Figure 2.8: Query Tree for Example Input Question Q2

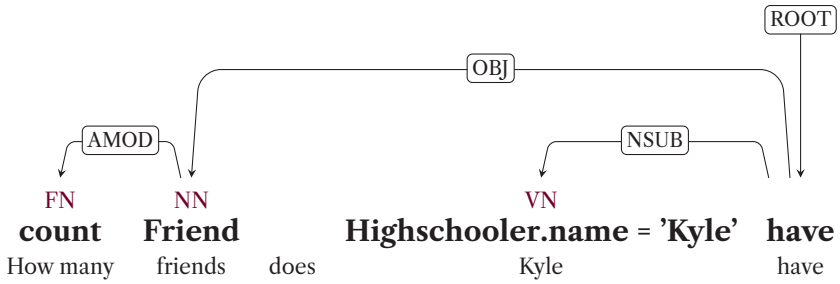

Figure 2.9: Query Tree for Example Input Question Q3

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 by [51] and [36].

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 expresses aggregation implicitly, the query subtree corresponding to the former expression contains no function node. As a result, direct translation of the query tree into a SQL clause will lead to 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 moving the nodes so that the name node appears before the operator node, resulting in the adjusted query tree in Figure 2.11.

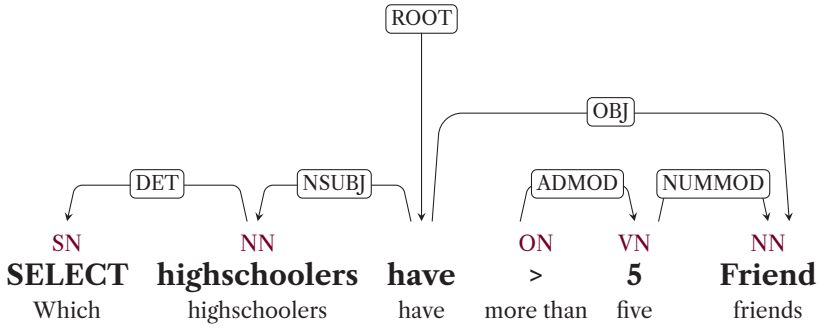

Figure 2.10: Query Tree for Example Input Question Q5

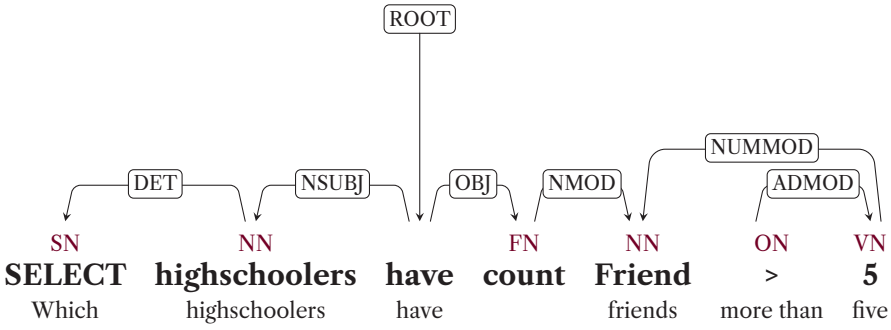

Figure 2.11: Adjusted Query Tree for Example Input Question Q5

2.3.2 QUERY TRANSLATION

Query Translation operates on top of a query tree and generate the final executable structured query. Based on the query tree, the system can determine the query type

of the final executable structured query. Depending on the query type, we can apply different query translation process.

Simple SELECT Query

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

- **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 a Operator node (ON), add a predicate with a "=" function.
- **JOIN ... ON Clause** If there are more than one relation added to the FROM clause, add a JOIN ... ON 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 also referred to as **join path inference**.

Based on the above process, we can obtain the correct SQL statements correctly 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.

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 a 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.

The translation index introduced in Chapter 2.3.1 is one way to capture domain vocabularies. 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 as regular expression patterns, 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 that contain domain-specific vocabularies such as "*Who are all the freshmen?*"

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 system, 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.

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.

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 RESULTS 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.

In addition to the results itself, it is important for the system to explain how the results are obtained 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, basically name nodes (NN), value nodes (VN), operator nodes (ON) and function nodes (FN) in the query tree.

For instance, based on the query tree of example input question Q5 (Figure 2.7), the question can be highlighted as "*Which* highschoolers *has* more than five friends? " In addition, the system can further indicate the corresponding mapping for each keyword.

While keyword highlighting can quickly inform a user on simple mistakes made by the system, such as omission of important keywords or incorrect keyword mapping, it only reflects part of the structured query generated. To help the user better understand the entire structured query, the system can generate a natural language description of the structured query. As an example, for the same example input question Q5, the system may explain the results obtained corresponding to "Return the

name of all highschoolers whose number of friends is more than 5." Chapter 6 discusses more details on how to generate accurate yet concise natural language from structured queries.

Non-Empty Result

If the execution results is not empty, then Results Generation can return the results directly to the user. In addition, as described earlier in Chapter 1.2.4, it may support additional interactivity to improve the usability of the system such as follows:

- **Results Visualization** To help the user better understand and interact with the returned results, the system can decide the best visualization for the results. For our example queries in Table 2.1, the most appropriate visualization is perhaps just simple tables.
- **Additional Interaction** The system may also support additional user interactions such as sorting and drilling down with the query results to allow the user further explore the query results without the needs to issue additional queries.

Empty Result

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
Cassandra	1
Andrew	3
Gabriel	1
Kyle	1

Figure 2.12: Relaxed Version of Q5: *Which highschoolers have friends?*

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 understood by the system correctly reflects the semantics of the original input question.

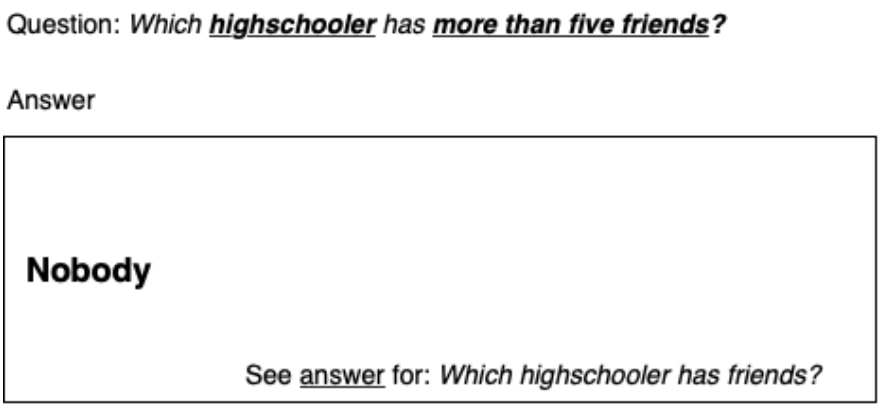

Figure 2.13: Example Result Display Page for Q5: *Which highschoolers have friends?*

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 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 [46]).

For instance, the result for the example input question Q5 is empty. The system can then relax the resulting query by removing the predicate "HAVING count(*) > 5" to obtain the following query:

```
SELECT T1.name, count(T2.student_id)
FROM Highschooler AS T1
JOIN Friend AS T2
ON T1.id = T2.student_id
GROUP BY T1.id
```

As one can see from the execution result of the relaxed query shown in Figure 2.12, the answer for the original query is empty, which is indeed correct since the highschoolers have at most three friends.

Figure 2.13 shows an example with the different options discussed above put together, with explanation by highlighting, natural language explanation of the executed query, and alternative query for empty results. We will discuss in more details later on related techniques on interactivity in Chapter 5 for both interaction generation and results generation.

2.4 SUMMARY

In this chapter, we introduce the common architecture of a modern NLIDB and present a step-by-step guide to build an basic implementation of such an NLIDB from scratch. We also discuss ideas on how to extend such a baseline implementation to handle more complex challenges.