www.editada.org

International Journal of Combinatorial Optimization Problems and Informatics, 13(3), Sep-Dec 2022, 34-42. ISSN: 2007-1558.

Wizard for creating semantic views in a natural language interface to databases

Rodolfo A. Pazos[1*], José A. Martínez[1], Héctor J. Fraire[1], Héctor J. Puga[2], Juana Gaspar[1]
[1] *Instituto Tecnológico de Cd. Madero, Mexico.*
[2] *Instituto Tecnológico de León, Mexico.*
r_pazos_r@yahoo.com.mx, jose.mtz@gmail.com, automatas2002@yahoo.com.mx, pugahector@yahoo.com, jgasparhdz@gmail.com
[*]corresponding author (*r_pazos_r@yahoo.com*)

Abstract. When using natural language interfaces to databases (NLIDBs) for database queries that involve many tables, the resulting SQL query may include semantically implicit entities. These entities are related to the semantic meaning of a query, when upon referring to an entity (table), another entity (or entities) is (are) semantically implied with which the first entity is related, and the user might ignore the relationship between the two (or more) entities. In a previous work, this problem was addressed by using a semantic view. However, the configuration of the NLIDB is very complex for the database administrator for dealing with this kind of queries. This means that the semantic view can only be created and configured by the NLIDB developers. This article describes a method to configure a dictionary of semantic information to create a virtual semantic view in a NLIDB, thus facilitating the creation of semantic views in a NLIDB.

Keywords: Natural language interface, Relational database, Semantic view.

Article Info
Received June 27, 2022
Accepted August 16, 2022

1 Introduction

Currently, society needs to access more information in a fast and reliable way, which requires new ways to retrieve information. Natural language interfaces to databases (NLIDBs) allow users to formulate queries in their own language, facilitating access to information without requiring knowledge of a query language for databases (DBs) or programming [1].

To date, there are many NLIDBs, and recently new interfaces continue to be developed [2]. Most of the NLIDBs developed so far perform natural language (NL) query processing in the same way for any type of query. However, considering the structure of DBs and the structure of the NL queries, the queries can be classified into simple queries and complex queries.

In the field of NLIDBs, the term complex query is mentioned in various publications, such as [3], [4], [5]. These publications mention that there are DBs with complex schemas (i.e., DBs whose schemas have many tables and foreign keys). The translation into SQL of some NL queries formulated by means of a NLIDB for accessing complex database results in a SQL query that involves several joins, in some cases, nested subqueries, and may include aggregation and ambiguities.

These NL queries are considered complex by various authors because the query contains implicit information about entities in the DB schema. The said entities have various relationships that are not specified in the NL query, and therefore it is difficult for a NLIDB to identify the entities that are not mentioned in the NL query (implicit) and to understand the relationship(s) existing between them.

The prototype NLIDB [6] that we have developed has a mechanism to answer a type of complex queries, specifically queries that involve semantically implied entities. This mechanism is called a semantic view, which is a SQL view to which semantic information is added. A key element of the NLIDB is the Semantic Information Dictionary (SID) [6], which is used for storing the definition of the semantic view.

The creation of semantic views in the SID can be done using the NLIDB Domain Editor. However, it is very difficult to define a semantic view using this tool, and therefore its use is limited to the NLIDB developers.

In order to improve the usability of the NLIDB, this article describes a method that allows users that are not very familiar with the interface to define semantic views in the SID of the NLIDB by means of a SQL statement.

This article is organized as follows. Section 2 describes the operation of the NLIDB and its configuration. In Section 3 the implementation of the Wizard for defining semantic views is explained. Section 4 describes some functional tests performed on the Wizard to verify its correct operation. In Section 5 the conclusions of this work are presented.

2 State of the art

Nowadays, there are many NLIDBs. However, most of them do not have mechanisms to correctly answer complex queries.

The NLIDBs considered in this state of the art (Table 1) were chosen using the following criteria: (1) commercial NLIDBs that have been tested with complex DBs or have mechanisms to answer complex queries, such as English Query [7] and ELF [8]; (2) prototype NLIDBs whose software is available for testing, like C-Phrase [9]; and (3) NLIDBs that consider some mechanism to answer complex queries, such as NaLIR [10] and DBPal [11].

English Query and ELF are commercial interfaces that work with Microsoft SQL Server 2000 and Microsoft Access database management systems (DBMSs), respectively.

For this reason, it was easy to perform tests with these DBMSs. The tests consisted in trying to configure the NLIDBs to answer three complex queries of the type that involve semantically implied entities from the ATIS query corpus: (1) List fares for all flights leaving after 1200 from Boston to Baltimore, (2) How much does it cost to fly from Boston to Oakland one-way?, (3) Please show the airlines which fly from Dallas to Denver.

Table 1. State of the art of NLIDBs

NLIDB	Year	Test DBs	Complex query
English Query	2000	Pubs, Northwind	✓
ELF	2004	ATIS, Geoquery	✕
C-Phrase	2010	Geoquery	✕
NaLIR	2017	Geoquery	✓
DBPal	2018	Patients, Geoquery	✓
Our NLIDB	2014	ATIS, Geoquery	✓

When testing English Query with the ATIS corpus, like our NLIDB, English Query allows to perform an automatic initial configuration tuning based on the DB schema. Subsequently, for English Query to be able to answer complex queries, it was necessary to fine-tune its configuration using its Suggestion Wizard, which allows the DB Administrator (DBA) to configure the models that contain semantic information so that the NLIDB understands NL elements and maps these elements to the DB schema. Unfortunately, it was not possible to make a configuration through the Suggestion Wizard, which would allow answering any of the three queries mentioned because the Suggestion Wizard only allows to relate search values to DB columns. To answer this type of queries, it is also necessary to indicate in the configuration the joins that are involved in the query.

Additionally, ELF also has an initial automatic configuration based on the metadata of the DB in use. In this case, we used the ATIS corpus to perform the test. When trying to fine-tune the ELF configuration with the available tools, it was not possible to make a configuration that would allow the queries shown previously to be answered correctly. ELF, similarly to English Query, lacks a tool to indicate the joins necessary to answer the query.

C-Phrase is a prototype interface that comes configured for use with the Geobase DB and has been tested with the Geoquery corpus. This interface has a tool to define patterns using logical expressions in English. You can also define synonyms for various patterns. However, using this tool for experimenting is very difficult because it is complicated to configure a pattern that allows complex queries to be processed, such as, for example, the queries mentioned before from the ATIS corpus. Additionally, C-Phrase does not report results on experimentation with ATIS.

NaLIR is a NLIDB that processes queries in NL by generating weighted SQL query templates [10]. The translation process is carried out through a dialogue between the user and the interface, where the interface allows the user to choose the correct interpretation of his query. NaLIR may allow to answer complex queries. However, it does not report results for queries of this type. This NLIDB has been tested with the Microsoft Academic Search dataset and reported an accuracy of 89.79% (88 correct queries out of 98).

DBPal is an interface based on deep neural network models. This system was trained with a set of queries and its SQL translation. With this information, the NLIDB can create SQL query templates to correctly answer NL queries. This interface has a tool for authoring queries that allows the user to formulate complex queries through suggestions. This NLIDB has been tested with two benchmark datasets: Patients and Geoquery, obtaining an accuracy of 75.93% and 48.9%, respectively. It is worth mentioning that Patients DB only has one table, while Geobase DB has around 11 tables, and therefore the queries corresponding to the Patients dataset are not complex, while some of the Geobase ones are.

2.1 Conclusions

As noted in previous studies, there is a large number of approaches to process NL queries by means of NLIDBs. English Query, ELF and C-Phrase can be configured manually by a DBA, unfortunately the configuration mechanisms are not suitable for configuring these NLIDBs to answer complex queries.

As for NaLIR and DBPal, when trained with sets of queries, these interfaces generate query templates, which are shown to users so that they can indicate which query they want to formulate. Therefore, it is possible to formulate complex queries using these interfaces, but only the queries that these interfaces consider in their templates.

The task of developing a NLIDB that can be easily configured by any DBA lies outside the capabilities of the aforementioned approaches.

3 Wizard for defining semantic views

3.1 Graphic user interface

To explain how the developed Wizard works, it is necessary to understand the SID tables involved in the definition of semantic views (Fig. 1).

Table *views*. Contains the name of the semantic view as well as its SQL statement.

Table *view_tables*. Contains information of the tables involved in the semantic view.

Table *view_columns*. Contains information about the columns that belong to tables involved in the semantic view.

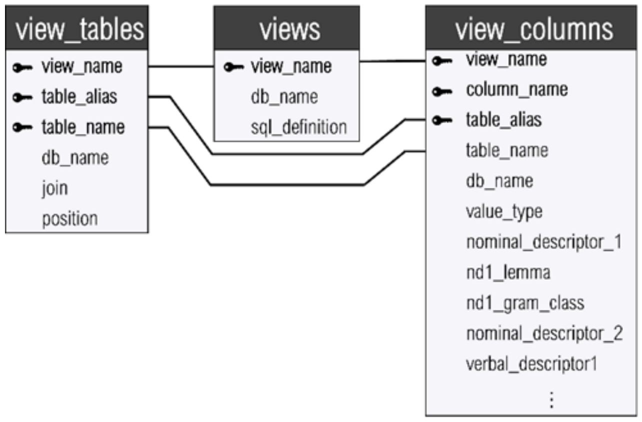


Fig. 1. Database schema for storing semantic views.

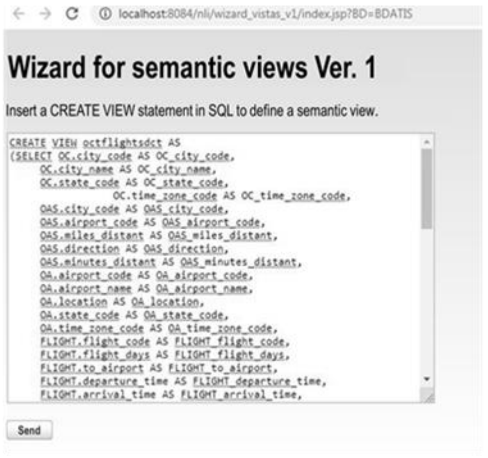
The main idea of the Wizard is that the DBA formulates a SQL statement that defines a view. This statement must be like Statement 1 (at the end of this list) and must have the following characteristics:

1) It must be a CREATE VIEW statement.
2) It must contain a valid view name.
3) It must be a statement of the form Select, From, Where, which defines the desired view.
4) The user must specify all the tables involved in the view with their respective aliases in the From clause of this statement.
5) All columns in the Select clause of the statement must have the alias of their respective table as specified in the From clause.
6) The Select clause must contain all the columns of each table specified in the From clause.
7) The Where clause should only contain the joins between the tables specified in the From clause.

Statement 1:

```
CREATE VIEW octflightsdct AS
(SELECT OC.city_code AS OC_city_code,
        OC.city_name AS OC_city_name,
          …
        FLIGHT.flight_code AS FLIGHT_flight_code,
        FLIGHT.arrival_time AS FLIGHT_arrival_time,
FROM    city AS OC,
       airport_service AS OAS,
       airport AS OA,
              flight AS FLIGHT,
       airport AS DA,
       airport_service AS DAS,
       city AS DC
WHERE  OC.city_code = OAS.city_code AND
       OAS.airport_code = OA.airport_code AND
       OA.airport_code = FLIGHT.from_airport AND
       FLIGHT.to_airport = DA.airport_code AND
       DA.airport_code = DAS.airport_code AND
       DAS.city_code = DC.city_code)
```

The creation of semantic views in the SID is easier to do using the Wizard than using the Domain Editor, since it only requires the DBA to have knowledge about SQL and to be able to define a view using a SQL expression. However, when using the Domain Editor for fine-tuning the SID, the DBA must specify a large amount of information using various controls and might insert erroneous information in the SID.



Wizard for semantic views Ver. 1

Insert a CREATE VIEW statement in SQL to define a semantic view.

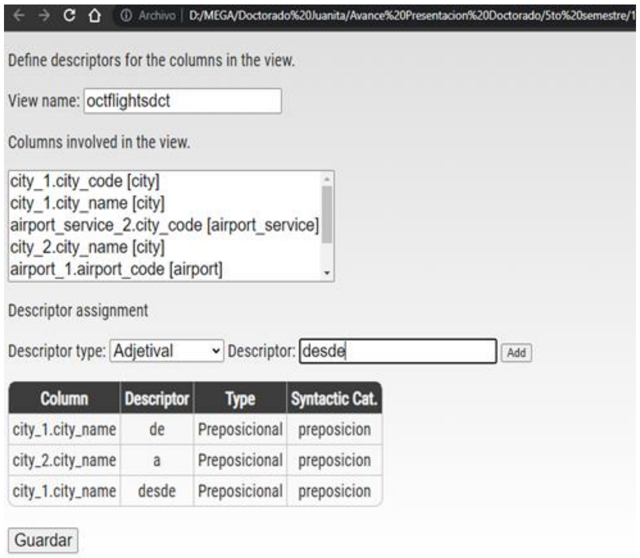
```
CREATE VIEW octflightsdct AS
(SELECT OC.city_code AS OC_city_code,
     OC.city_name AS OC_city_name,
     OC.state_code AS OC_state_code,
                 OC.time_zone_code AS OC_time_zone_code,
     OAS.city_code AS OAS_city_code,
     OAS.airport_code AS OAS_airport_code,
     OAS.miles_distant AS OAS_miles_distant,
     OAS.direction AS OAS_direction,
     OAS.minutes_distant AS OAS_minutes_distant,
     OA.airport_code AS OA_airport_code,
     OA.airport_name AS OA_airport_name,
     OA.location AS OA_location,
     OA.state_code AS OA_state_code,
     OA.time_zone_code AS OA_time_zone_code,
     FLIGHT.flight_code AS FLIGHT_flight_code,
     FLIGHT.flight_days AS FLIGHT_flight_days,
     FLIGHT.to_airport AS FLIGHT_to_airport,
     FLIGHT.departure_time AS FLIGHT_departure_time,
     FLIGHT.arrival_time AS FLIGHT_arrival_time,
```

Fig. 2. Wizard's main window.

The DBA just has to enter Statement 1 in the Wizard (Fig. 2) to define a new semantic view in the SID. In addition, the NLIDB will ask the user to specify some descriptors whether nominal, verbal, adjectival, or prepositional in NL to associate them to the corresponding columns of the semantic view (Fig. 3).

After entering Statement 1 in the Wizard, the NLIDB will show the DBA all the columns detected in the Select clause of that statement. Later, the DBA may assign up to two types of descriptors (nominal, verbal, prepositional or adjectival), which can be added by means of a list for selecting the type and by writing the descriptor in a text box to define it (Fig. 3).

Finally, the NLIDB will take the information entered by the user (CREATE VIEW instruction and descriptors) to configure the sections of the SID corresponding to the definition of semantic views (*views*, *view_tables*, and *view_columns* tables).



Archivo | D:/MEGA/Doctorado%20Juanita/Avance%20Presentacion%20Doctorado/5to%20semestre/1

Define descriptors for the columns in the view.

View name: octflightsdct

Columns involved in the view.

city_1.city_code [city]
city_1.city_name [city]
airport_service_2.city_code [airport_service]
city_2.city_name [city]
airport_1.airport_code [airport]

Descriptor assignment

Descriptor type: Adjetival Descriptor: desde Add

Column	Descriptor	Type	Syntactic Cat.
city_1.city_name	de	Preposicional	preposicion
city_2.city_name	a	Preposicional	preposicion
city_1.city_name	desde	Preposicional	preposicion

Guardar

Fig. 3. Window for defining natural language descriptors.

3.2 Wizard Processing

This section describes the internal processing of the Wizard to define a semantic view in the SID.

The Wizard defines a semantic view in the SID by entering information in three tables: *views*, *view_tables*, and *view_columns*. The filling of these tables is carried out by 4 algorithms.

Algorithm 1 describes the process of filling the *views* table, where the name of the view is obtained from the Select clause of Statement 1. Subsequently, it verifies if the view does not exist in the SID to be able to record it in the *views* table (line 4).

```
Algorithm 1: Pseudocode for filling table views
// Q is the statement introduced in the wizard (Figure 2)
// V is the name of the view
// DB is the name of the DB in use
1:  procedure fillViews(Q)
2:      V ← getViewName(Q) // Obtains the name of the view from the Create View clause
3:      if V does not exist in SID then
4:          insertIntoViews(DB, V) // Stores DB and V in table views
5:      endif
6:  end
```

Fig. 4. Pseudocode for populating the *views* table.

Upon executing Algorithm 2, the NLIDB populates the *view_tables* table. To this end, it first gets the tables and their aliases from the From clause of Statement 1. For each table found in the view, it verifies if the tables exist in the SID (lines 3 to 5) to insert them in the *view_tables* table.

```
Algorithm 2: Pseudocode for filling table view_tables
// T is a set of pairs table-alias
// $N_t$ is the number of elements in T
// $T_i$ is the i-th element of T
// DB is the name of the DB in use
// V is the name of the view
1:  procedure fillView_tables(Q)
2:      T ← getTablesAliasFrom(Q) // Obtains tables and aliases from the From clause
                                     and stores them in T
3:      for i=0 to  $N_t$-1do
4:          if $T_{iT}$ exists in SID then // $T_{iT}$ is the table of pair $T_i$, and $T_{iA}$ is the alias of $T_{iT}$
5:              InsertIntoViewTables(DB, V, $T_{iT}$, $T_{iA}$) // Stores pair in table view_tables
6:          endif
7:      endfor
8:  end
```

Fig. 5. Pseudocode for populating the *view_tables* table.

It is worth mentioning that populating the *view_tables* table is performed in two steps. Algorithm 3 describes the process to insert the relationships (joins) between the tables of the view defined in Statement 1. Initially, the relationships of the Where clause (line

2) are obtained. For each relationship, its existence is verified in the SID (lines 3 and 4), if it exists, the relationship is inserted in the *view_tables.join* column (line 5).

```
Algoritmo 3: Seudocódigo para llenar las relaciones en la tabla view_tables
// N_R is the number of joins in R
// R_i is the i-th element in R
1:    procedure fillJoinsView_tables(Q)
2:    R ← getJoins(Q) // Obtains the joins from the Where clause
3:    for i=0 to  N_R-1do
4:        if R_i exists in SID then
5:            insertJoin(R_i) // Stores join in table view_tables
6:        endif
7:    endfor
8:    end
```

Fig. 6. Pseudocode for inserting the relationships in the *view_tables* table.

Finally, the *view_columns* table is populated (Algorithm 4). For this purpose, the columns of the view are obtained from the Select clause of Statement 1 (line 2). For each identified column, its aliases are obtained (line 4), its existence in the SID is verified, the name of the base table to which it corresponds is obtained (line 6), and finally all the information mentioned in the *view_columns* table is inserted. The information regarding the descriptors is inserted through the window shown in Fig. 3.

```
Algorithm 4: Pseudocode for filling table view_columns
// C is a set of pairs columns-alias
// C_i is the i-th element of C
// N_C is the number of pairs in C
// DB is the name of DB in use
// V is the name of the view
1:    procedure fillView_columns(Q)
2:    C ← getColumns(Q) // Obtains the columns from the Select clause
3:    for i=0 to N_C-1do
4:        A_t ← obtenerAliasTabla(C_iC) // A_t stores alias of the table of column C_iC
5:        if A_t exists in SID then
6:            C_iT = getTable(A_t) // Obtains the name of the base table for A_t
7:              insertIntoView_columns(DB, V, C_iC, C_iT, A_t) // Stores all the information
8:        endif
9:    endfor
10:   end
```

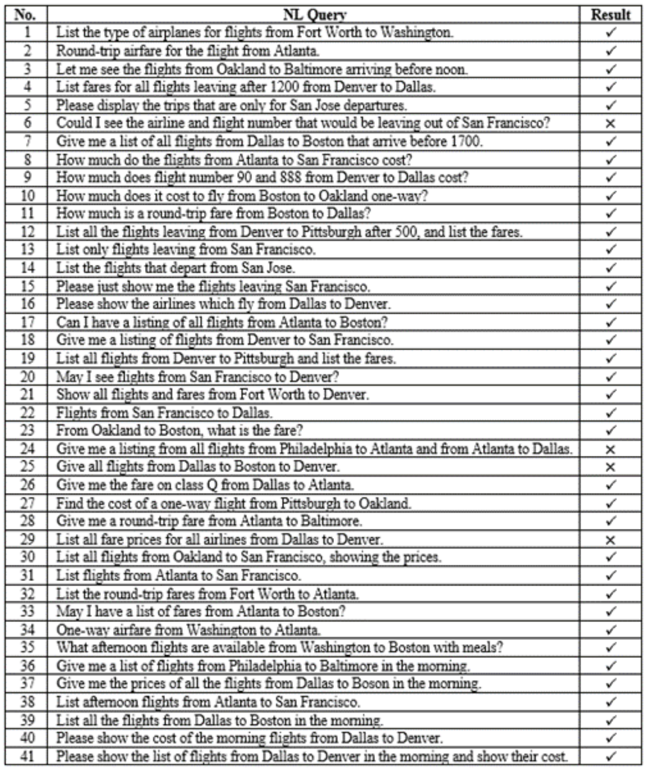
Fig. 7. Pseudocode for populating the *view_columns* table.

4 Experimental tests

Functional tests were carried out to ensure the correct operation of the tool implemented in the NLIDB. The tests consisted in defining a semantic view through the Wizard so that the NLIDB had information to be able to answer queries that involved information about flights from one city to another. The instruction entered in the Wizard was Statement 1, found in Section 3 of this article.

To test that the semantic view defined by the Wizard was configured correctly, from a corpus of 70 queries from the ATIS DB corpus described in [12], 41 queries involving information about flights from one city to another were taken (Fig. 8). These queries were introduced one by one to the NLIDB, and their results and the SQL statements obtained were verified. It is important to make clear that our NLIDB only answers queries in Spanish, however, because of the similarities among European languages, the process of our NLIDB can be applied also to English. Therefore, for the sake of clarity the examples presented in Fig. 8 are in English.

The result obtained was 37 out of 41 questions answered correctly. The four queries that were not answered correctly by the NLIDB were due to various problems found in the structure of the query or processing problems of the NLIDB. For example, queries 24 and 25 request information about flights departing from more than two cities at the same time. The problem posed by this type of queries is not considered in the implementation of the NLIDB. In query 6, due to the structure of the query, and because the NLIDB parser only performs a shallow parsing, it is not possible for the NLIDB to correctly process the columns involved in the Select phrase. Query 29 has the same problem as query 6.



No.	NL Query	Result
1	List the type of airplanes for flights from Fort Worth to Washington.	✓
2	Round-trip airfare for the flight from Atlanta.	✓
3	Let me see the flights from Oakland to Baltimore arriving before noon.	✓
4	List fares for all flights leaving after 1200 from Denver to Dallas.	✓
5	Please display the trips that are only for San Jose departures.	✓
6	Could I see the airline and flight number that would be leaving out of San Francisco?	×
7	Give me a list of all flights from Dallas to Boston that arrive before 1700.	✓
8	How much do the flights from Atlanta to San Francisco cost?	✓
9	How much does flight number 90 and 888 from Denver to Dallas cost?	✓
10	How much does it cost to fly from Boston to Oakland one-way?	✓
11	How much is a round-trip fare from Boston to Dallas?	✓
12	List all the flights leaving from Denver to Pittsburgh after 500, and list the fares.	✓
13	List only flights leaving from San Francisco.	✓
14	List the flights that depart from San Jose.	✓
15	Please just show me the flights leaving San Francisco.	✓
16	Please show the airlines which fly from Dallas to Denver.	✓
17	Can I have a listing of all flights from Atlanta to Boston?	✓
18	Give me a listing of flights from Denver to San Francisco.	✓
19	List all flights from Denver to Pittsburgh and list the fares.	✓
20	May I see flights from San Francisco to Denver?	✓
21	Show all flights and fares from Fort Worth to Denver.	✓
22	Flights from San Francisco to Dallas.	✓
23	From Oakland to Boston, what is the fare?	✓
24	Give me a listing from all flights from Philadelphia to Atlanta and from Atlanta to Dallas.	×
25	Give all flights from Dallas to Boston to Denver.	×
26	Give me the fare on class Q from Dallas to Atlanta.	✓
27	Find the cost of a one-way flight from Pittsburgh to Oakland.	✓
28	Give me a round-trip fare from Atlanta to Baltimore.	✓
29	List all fare prices for all airlines from Dallas to Denver.	×
30	List all flights from Oakland to San Francisco, showing the prices.	✓
31	List flights from Atlanta to San Francisco.	✓
32	List the round-trip fares from Fort Worth to Atlanta.	✓
33	May I have a list of fares from Atlanta to Boston?	✓
34	One-way airfare from Washington to Atlanta.	✓
35	What afternoon flights are available from Washington to Boston with meals?	✓
36	Give me a list of flights from Philadelphia to Baltimore in the morning.	✓
37	Give me the prices of all the flights from Dallas to Boson in the morning.	✓
38	List afternoon flights from Atlanta to San Francisco.	✓
39	List all the flights from Dallas to Boston in the morning.	✓
40	Please show the cost of the morning flights from Dallas to Denver.	✓
41	Please show the list of flights from Dallas to Denver in the morning and show their cost.	✓

Fig. 8. Corpus of queries that involve a semantic view.

5 Conclusions

The configuration of the SID, except for the NLIDB developers, using the Domain Editor for defining semantic views, has proven to be very complicated, and therefore, its use is very limited. A Wizard that allows defining semantic views from a SQL statement that defines a view was implemented in our NLIDB. This facilitates the definition of semantic views in the SID by DBAs who are only required to be skillful in SQL.

Functional tests were performed on the new version of our NLIDB. It is worth mentioning that our NLIDB answered 37 out of 41 queries correctly. Queries that were not answered correctly were due to query processing problems and not due to the Wizard. Therefore, it has been shown that the semantic view created by the Wizard and used to answer the test queries is correct and the Wizard works correctly (Fig. 8).

This Wizard does not require the DBA to have knowledge about the SID structure, which allows any DBA to define semantic views in the SID more easily. In summary, generating semantic views through the Wizard is easier than defining them using the Domain Editor.

References

1. Androutsopoulos, I., Ritchie, G. D., Thanisch, P.: Natural language interface to database: An introduction. Natural Language Engineering 1(1), 29-81 (1995)
2. Pazos, R. A., Aguirre, M. A., González, J. J., Martinez, J. A., Pérez, J., Verástegui, A. A.: Comparative study on the customization of natural language interfaces to databases, SpringerPlus 5(553), 1-30 (2016) doi:10.118640064-016-2164-y
3. Li, F., Jagadish, H. V.: Constructing an interactive natural language interface for relational databases. Proc. of the VLDB Endowment 8(1), 73-84 (2014)
4. Pazos, R. A., Rivera, G., Martínez, J. A., Gaspar, J., Florencia, R.: Natural language interfaces to databases: A survey on recent advances. In: Pazos, R. A., Florencia, R., Paredes, M. A. (eds.) Handbook of Research on Natural Language Processing and Smart Service Systems, 1-30. IGI Global, PA, USA (2021) doi:10.4018/978-1-7998-4730-4.ch001
5. Yu, C., Jagadish, H. V.: Querying complex structured databases. Proc. of the 33rd International Conference on Very Large Databases, 1010-1021 (2007)
6. Pazos, R. A., González, J.J., Aguirre, M.A.: Semantic model for improving the performance of natural language interfaces to databases. Proc. Advances in Artificial Intelligence – 10th Mexican International Conference on Artificial Intelligence, 227-290 (2011)
7. Microsoft Corporation: Microsoft SQL Server 2000 Resource Kit. Microsoft Press, WA, USA (2001)

8. ELF Software, ELF Software Documentation Series: Overview. http://www.elfsoft.com/help/accelf/Overview.htm (2009). Accessed 22 May (2022)
9. Minock, M., Olofsson, P., Näslund, A.: Towards building robust natural language interfaces to databases. Proc. 13th International Conference on Applications of Natural Language to Information Systems, 187-198 (2010)
10. Li, F.: Querying RDBMS using natural language. Ph. D. dissertation, University of Michigan, MI, USA (2017).
11. Utama, P., Weir, N., Basık, F., Binnig, C., Cetintemel, U., Hättasch, B. et al.: DBPal: An end-to-end neural natural language interface for databases. https://arxiv.org/pdf/1804.00401 (2018). Accessed 22 May (2022)
12. Aguirre, M. A.: Modelo semánticamente enriquecido de bases de datos para su explotación por interfaces de lenguaje natural. Ph.D. dissertation, Instituto Tecnológico de Tijuana, Tijuana, Mexico (2014)