ISSN 2395-8618

Processing Natural Language Queries via a Natural Language Interface to Databases with Design Anomalies

Rodolfo A. Pazos, José A. Martínez, and Alan G. Aguirre

***Abstract*—Natural language interfaces to databases (NLIDBs) have proven to be very promising tools when trying to obtain information from a relational database since they require the end user to have very little training and knowledge about databases to use them. However, their development has not been easy due to problems related to natural language processing. In addition to this, most authors overlook an important factor in developing these tools, which is the quality of the design of the database to be queried by the NLIDB. The problem arises because there can be many alternatives for the design of databases, and some contain design anomalies. Many NLIDBs would not work correctly for these databases since they were designed under the assumption that they would be used with databases without anomalies. This article describes an improvement to the processing performed by a domain-independent interface to treat databases with design anomalies and for the interface to be able to correctly process queries involving such anomalies. The literature on NLIDBs has not mentioned this problem and much less addressed it.**

***Index Terms*—Natural language interfaces, relational databases, user interfaces**

1. Introduction

NOWADAYS, information plays a very important role in business. Most of the information is, in many cases, stored in databases. However, for a user to obtain information from a database (DB), he must have knowledge of a query language for databases (such as SQL).

Due to this situation, access to DBs by inexperienced users is very limited. On many occasions, the information must be accessed by inexperienced users; therefore, it is necessary that they use easy-to-use software that does not require knowledge about DBs. To this end, a large number of tools have been developed, whose main characteristic is to show the DB schema and offer methods for obtaining information based on visual schemas. Although the mentioned tools allow building SQL queries from visual schemas, they are not very easy to use, as they require a certain degree of SQL knowledge.

NLIDBs are tools that allow inexperienced users to compose SQL queries using a natural language (NL) expression [1]. These interfaces are very easy to use for inexperienced users; however, their development has been delayed due to problems related to natural language processing (NLP) and the semantic content of DBs.

Nowadays, it is very common to find DBs that have design anomalies [2][3][4]. This is because sometimes developers include some anomalies in their DB designs to satisfy the needs of the applications for which they were designed (e.g., surrogate keys). However, this affects the performance of NLIDBs, since they are designed to work with correctly designed DBs, i.e., DBs without design anomalies.

In summary, the problem addressed in this article consists of developing a method that allows an NLIDB to be used for querying DBs that have design anomalies and answering correctly as if the DBs had no anomalies.

This paper describes methods for improving an NLIDB, allowing it to process queries formulated on DBs that have design anomalies. The anomalies dealt with are the following: the absence of primary and foreign keys, use of surrogate keys, use of columns for storing aggregate function calculations, and use of repeated columns in two or more tables. Additionally, to demonstrate the efficiency of these methods, we performed experimental tests of our NLIDB using DBs with design anomalies. Additionally, to demonstrate the efficiency of these methods, we performed experimental tests of our NLIDB using DBs with design anomalies.

2. State of the Art

Several NLIDBs have been developed since the 1970s, such as LUNAR [5], RENDEZVOUS [6], LADDER [7], among others. These interfaces could also be configured to work with other DBs despite having been designed for a DB in particular; however, this task was very difficult due to technical limitations at the time.

In [8] some NLIDBs are considered relevant to this work. It is important to remark that for most NLIDBs, there is no software that can be used for testing. These NLIDBs are relevant for two reasons: First, there is a prototype (C-Phrase) or commercial software (ELF) of the NLIDB, which can be tested and, thus, demonstrate if they have any mechanism to treat design anomalies. Second, the other NLIDBs have been recently developed.

Manuscript received on October 12, 2020, accepted for publication on November 14, 2020, published on December 30, 2020.

The author is with the Instituto Tecnológico de Cd. Madero, Tecnológico Nacional de México, Cd. Madero, Tamaulipas 89440, Mexico (e-mail: r_pazos_r@yahoo.com, jose.mtz@gmail.com, li.aguirre.lam@hotmail.com).

ISSN 2395-8618

TABLE 1
STATE OF THE ART NLIDBS

NLIDB	Year	Design Anomaly Treatment
NADAQ	2019	No
Cross-domain NLI	2019	No
TEMPLAR	2019	No
nQuery	2017	No
Aneesah	2015	No
NL2CM	2015	No
NaLIR	2014	No
NLWIDB	2013	No
C-Phrase	2010	Partially
ELF	2004	Partially

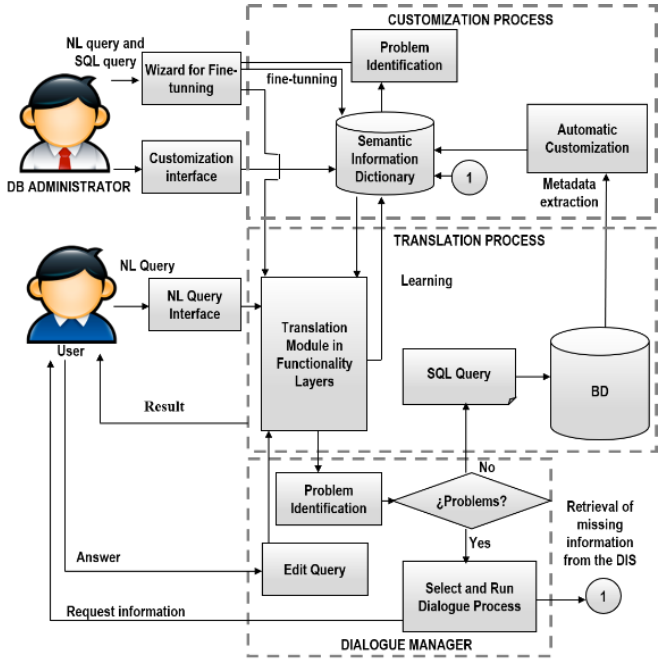

Fig. 1. NLIDB architecture.

In this article, the criteria mentioned are used to choose the relevant NLIDBs for the state of the art.

It is worth mentioning that there are also software tools for design and analysis of DBs. Some examples of these are Visual Paradigm, Vertabelo, DbSchema, Toad Data Modeler, among others.

The aforementioned tools, in addition to allowing users to easily design DBs from entity-relationship diagrams, also have analysis tools on the structure of the designed DB. However, most of the analysis and error correction tools that these software tools have been for correcting errors related to dirty DBs.

Additionally, the design anomalies considered in this article require semantic information that only the user can provide. Unfortunately, none of the mentioned software considers this kind of problem.

Although no work mentioned so far has addressed the problem of design anomalies in DBs, the only readily available interfaces that can be used for testing are included in this analysis of the state of the art (Table 1), such as C-PHRASE [9] and ELF [10]. Others that have been recently developed are also included, such as NADAQ [11], Cross-domain NLI [12], TEMPLAR [13], nQuery [14], Aneesah [15], NL2CM [16], NaLIR [17], NLWIDB [18]. In the former, the performance they have with DBs with design anomalies is evaluated.

It is important to mention that in this analysis, domain-specific NLIDBs are omitted since they are designed to work with the design anomalies of the DB for which they were designed.

Both the more recently developed NLIDBs and commercial NLIDBs have not considered design anomalies in DBs as a problem that must be solved in order to correctly translate NL queries to SQL queries. Therefore, it cannot be guaranteed that the operation of most of the NLIDBs listed will be correct when using DBs with design anomalies.

3. OUR NLIDB

The NLIDB used in this work is a domain independent prototype interface for the Spanish language [19].

A crucial component of this interface is the Semantic Information Dictionary (SID), shown in Figure 1. The SID is a DB that stores metadata of the DB in use (DB schema, columns, tables) and useful semantic information (nominal, verbal and prepositional descriptors, among others) to relate the query in NL to the elements of the SQL statement that the interface needs to build.

In addition to the SID, this interface consists of two main processes: the customization process and the translation process shown in Figure 1.

The customization process is carried out by the DB administrator (DBA), where the DBA configures the SID by entering semantic information regarding the tables, columns and relationships that exist in the schema of the DB in use.

The process of translating a query consists of three subprocesses: lexical analysis, syntactic analysis and semantic analysis (Figure 2).

Each sub-process is carried out by functional layers, where each layer deals with a problem that must be solved so that the interface can obtain a correct transla-tion of the NL query.

Lexical analysis. It performs a lexical tagging process, obtaining the syntactic category (part of speech) of each word in the query from a lexicon stored in a DB. In this layer, lexical errors are corrected, syntactic ambiguity and homography problems are resolved. The result obtained consists of a tagged query.

Syntactic analysis. The tagged query is used to build a syntactic tree, where syntactic errors are corrected, syntactic ellipsis is resolved, and anaphora problems are detected.

ISSN 2395-8618

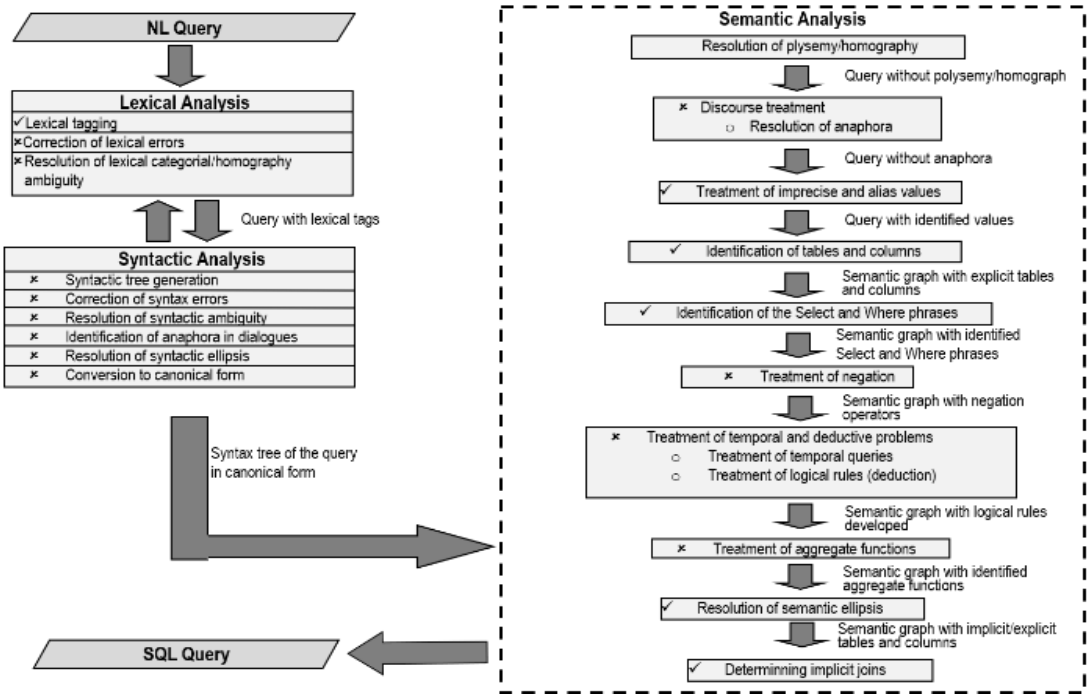

Fig. 2. Functionality layers of the translation module.

Semantic analysis. A representation of the meaning of the tagged query is constructed, which can be used to translate it into SQL. This layer is the most complex since most of the problems are due to the interpretation of the meaning of the query. This layer is divided into the following sub-layers:

1. Treatment of polysemy and homography.
2. Anaphora treatment.
3. Treatment of imprecise values and aliases.
4. Identification of tables and columns.
5. Identification of Select and Where phrases.
6. Treatment of negative queries.
7. Treatment of temporary and deductive problems.
8. Treatment of aggregation and grouping functions.
9. Resolution of semantic ellipsis.
10. Determination of implicit joins.

It is important to mention that, currently, only the most necessary layers (with a checkmark ✓) have been implemented (Figure 2).

Useful information is collected when processing the NL query via the functional layers. With this information, the NLIDB constructs a SQL query whose result contains the information requested by the user.

4. Processing NL Queries Formulated in DBs with Design Anomalies

The NLIDB [19] is designed on the assumption that the DB in use has no design anomalies; therefore, it does not perform well when translating queries to DBs with this problem.

Four design anomalies are considered in this article:

1. Absence of primary and foreign keys.
2. Use of surrogate keys.
3. Columns for storing aggregate function (AF) calculations.
4. Repeated columns in two or more tables.

Before the end user can use the NLIDB to answer queries, an initial setup process must be performed. This process is carried out semi-automatically by the NLIDB with the help of the DBA. In principle, the DBA must choose the DB with which he will work, and the NLIDB obtains the metadata from the DB and uses it to configure the SID. It is important to mention that the metadata obtained includes the design anomalies contained in the DB; therefore, the SID configuration has a representation of the DB with design anomalies.

4.1 Queries that Involve Absence of Primary and Foreign Keys

The absence of foreign keys directly affects the performance of the NLIDB. This is because the NLIDB, by means of the foreign keys defined in the DB schema, saves in the SID the existing relationships between the tables that have foreign keys defined and the tables to which they refer. This process is used in the sublayer Determination of Implicit Joins, where a semantic graph is created to determine the joins between tables when two or more tables are involved in the query.

Without the foreign keys properly defined in the DB, the NLIDB constructs a semantic graph where some tables are not connected as they should be. Additionally, the NLIDB is not

ISSN 2395-8618

able to deduce the existing relationships between tables and, therefore, it is not able to build a SQL query with the joins required by the NL query.

To solve the aforementioned problem, a configuration module was implemented that allows the DB administrator to specify relationships between tables whose foreign keys are not defined in the DB. In this way, the NLIDB can use these relationships to build a semantic graph, which contains enough information to define the joins in the SQL query in case they are required.

For example, consider the following NL query for the Geobase DB [20]:

¿Qué ríos pasan por el estado de Alaska?
Which rivers run through the state of Alaska?

Assuming that the DB does not have foreign keys defined, the SQL query built by the NLIDB would be the following:

1: SELECT *river.river_name* FROM *river, state*
2: WHERE *state.state_name* LIKE ‘Alaska’;

As can be seen, the NLIDB only detects the tables directly involved in the query; however, there is an intermediate table (*riverstate*), which is not included in the query. Furthermore, relationships between tables are not defined in the SID; therefore, the NLIDB does not com-pose the joins between them.

Once the relationships between tables have been defined in the SID using the configuration module, the following query is obtained in SQL, which contains the intermediate tables and joins necessary in the query:

1: SELECT *river.river_name*
2: FROM *river, state, riverstate*
3: WHERE *state.state_name* LIKE ‘Alaska’
4: AND *state.abbreviation = riverstate.state_abbreviation*
5: AND *riverstate.river_id = river.river_id*;

4.2 Queries that Involve Surrogate Keys

The use of surrogate keys affects the performance of the NLIDB, because this anomaly can cause data redundancy and lack of relationships between tables. Data redundancy affects the results obtained by the NLIDB. In these cases, some of the rows returned by the NLIDB can be confusing for the end user. The lack of relationships affects the creation of joins in the SQL query and, therefore, the result.

To solve this problem, a configuration interface for the SID was developed to define surrogate keys and relationships through foreign keys between the tables that have a surrogate key and other base tables in the SID [21]. Additionally, an algorithm was implemented to improve the processing of the interface so that it may be able to use the relationships defined by the configuration interface and ignore the relationships defined in the DB schema related to surrogate keys.

For example, consider the NL query:

¿En cuál estado se encuentra el lago Ontario?
In which state is Lake Ontario?

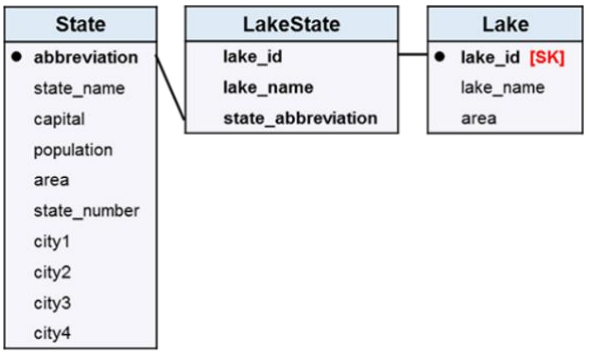

Fig. 3. Geobase DB fragment with a surrogate key.

The tables involved in the query are *State*, *LakeState*, and *Lake*. In Figure 3 a fragment of the DB schema with the design anomaly is presented, where *Lake.lake_id* is a surrogate key, *LakeState.lake_id* is a foreign key that connects with the surrogate key, and there could be a natural relationship between *Lake.lake_name* and *LakeState.lake_name*. However, due to the use of the surrogate key, no such relationship was defined in the DB schema.

The surrogate key *Lake.lake_id* and the foreign key (*LakeState.lake_name* REFERENCES *Lake.lake_name*) are defined in the SID by means of the configuration module. When defining the foreign key, the NLIDB marks the foreign key related to the surrogate key (*LakeState.lake_id* REFERENCES *Lake.lake_id*) as null so that it is ignored in the NL query translation process.

Processing the query using the NLIDB without treat-ing the design anomaly would result in the following SQL query:

1: SELECT *State.state_name*
2: FROM *State, LakeState, Lake*
3: WHERE *Lake.lake_name* LIKE ‘Ontario’
4: AND *State.abbreviation=LakeState.state_abbreviation*
5: AND *LakeState.lake_id = Lake.lake_id*;

While the query obtained when treating the design anomaly through the configuration interface is as fol-lows:

1: SELECT *State.state_name*
2: FROM *State, LakeState, Lake*
3: WHERE *Lake.lake_name* LIKE ‘Ontario’
4: AND *State.abbreviation = LakeState.state_abbreviation*
5: AND *LakeState.lake_name = Lake.lake_name*;

The first query includes joins and the surrogate key, while the second query includes a natural join using the columns *LakeState.lake_name* and *Lake.lake_name*.

4.3 Queries that Involve Columns for Storing Aggregate Function Calculations

The columns that can be calculated with AFs cause the NLIDB to obtain erroneous results since the values ob-tained can also be calculated by means of an AF applied to a column of another table.

ISSN 2395-8618

Algorithm 1 Pseudo-code for processing columns for storing AF calculations

```
1:  procedure AFColumnsProcess(Q,n)
2:      for i=0 to n-1 do
3:          if isAFColumn(Q_i) then
4:            AFColVal = getAF(Q_i) + "("+getAFColumn(Q_i)+")"
5:              setFinalTag(Q_i, AFColVal)
6:          endif
7:      endfor
```

Fig. 4. Algorithm 1.

For example, consider the following NL query:

¿Cuál es la población del estado de Mississippi?
What is the population of the state of Mississippi?

The population of a state could be calculated in two ways: using the column *State.population* or by adding the populations of the cities of the state. The DBA must decide whether to use the anomalous column or to use an AF instead.

For this reason, a configuration module was devel-oped that allows the DBA to define the columns that can be calculated with AF in the SID and assign them an AF associated with a column so that it can be used instead of the anomalous column.

An algorithm that modifies the internal functioning of the NLIDB in such a way that it can use the AFs associ-ated with this type of columns was also developed.

Algorithm 1 shows the pseudocode that describes the processing of columns for storing AF calculations. In the pseudocode, *Q* is the NL query entered by the user, Q_i is a token (word of NL query) of query *Q*, *n* is the total number of tokens in *Q*, and *AFColVal* is a variable used to store the final label that the referred token will be assigned. In line 3, for each token of query *Q*, it is verified if the column to which the token Q_i refers to is a column that stores an AF calculation. In this case, in line 4, an expression (character string) is stored in *AFColVal*. The expression consists of the AF (avg, count, etc.) stored in the SID for the mentioned column and the name of the column to which the AF will be applied. Lastly, the final label of token Q_i is updated with the information from *AFColVal*.

Regarding the example, when processing the NL query without treating the design anomaly, the NLIDB builds a query in SQL with the column State.population in the Select clause as follows:

```
1: SELECT State.population
2: FROM State
3: WHERE State.name LIKE 'Mississippi';
```

On the other hand, by defining the column *State.population* as a column that stores AF calculations in the SID and assigning the AF SUM(*City.population*) instead, the NLIDB will detect that there are two tables involved in the query: *City* in the Select clause and *State* in the Where clause; therefore, a query will be generated with their respective join:

```
1: SELECT SUM(City.population)
2: FROM State, City
3: WHERE State.name LIKE 'Mississippi' AND
4: State.abbreviation = City.state_abbreviation;
```

It is important to mention that the DBA can configure the NLIDB for using the column with the design anomaly or using the AF in query processing.

4.4 Queries that Involve Repeated Columns in Two or More Tables

The presence of repeated columns in multiple tables is an anomaly that creates conflict in the NLIDB processing when deciding to choose the right column from the DB that has several occurrences in different tables.

The SID contains information to match the words of an NL query with the columns of a DB. However, when in a DB there are two or more tables with repeated columns, the NLIDB is not able to know which is the right column that contains the information referred to by the user. Most of the time, columns that suffer from this problem can contain incorrect information due to insertion errors. Therefore, when the NLIDB refers to such information, it will be erroneous.

To solve the problems related to this anomaly, the DBA must identify the column that contains the correct information (usually a column that corresponds to an attribute of a strong entity). Afterwards, the DBA must indicate the columns that are repeated through the con-figuration module implemented for this purpose. Once this is done, the SID will have the necessary information to carry out the processing of queries that involve this type of columns.

In addition to the above, an algorithm was developed to allow the NLIDB to identify these columns and, thus, be able to correctly process queries.

In line 3 of Algorithm 2, for each token in query *Q*, the tokens that refer to a column are identified (the meanings of *Q*, Q_i and *n* are the same as those for Algorithm1). Subsequently, in lines 4 and 5, the columns *repCols* that are repeated in the DB with their respective labels *repColsTags* (the columns with the most reliable information) are obtained. This column information is obtained from the SID. In lines 6 to 10 a verification is carried out to identify the token that refers to a repeated column; in case of identifying one, in line 8 the token is labeled with the correct column.

ISSN 2395-8618

TABLE 2
EXPERIMENTAL TESTS RESULTS

Design Anomaly	Our NLIDB		ELF	
	ATIS	Geobase	ATIS	Geobase
Abs. of PKs and FKs	5	5	0	0
Use of Surrogate keys	5	5	0	0
Columns for storing AF calculations	5	5	0	2
Repeated columns	5	5	1	1
Total	20	20	1	3

Algorithm 2. Pseudocode for Processing repeated columns

```
1:  procedure TreatmentOfRepColumns(Q,n)
2:    for i=0 to n-1 do
3:      if isColumn(Q_i) then
4:        repCols = getRepeatedCols()
5:        repColsTags = getRepeatedColsTags()
6:        for j=0 to sizeOf(repCols)
7:          if getColumnTag(Q_i) == repCols_j
8:            setFinalTag(Q_i, repColsTags_j)
9:          endif
10:       endfor
11:     endif
12:   endfor
```

Fig. 5. Algorithm 2.

To exemplify the operation of the mentioned pseudocode, consider the query:

¿Qué montañas están en el estado de Alaska?
Which mountains are in the state of Alaska?

When processing the previous query by the NLIDB, it detects the *Mountain.mountain_name* column for the Select clause and the *Mountain.state_name* column for the Where clause with its search value Alaska. The *state_name* column is found in 5 tables of the DB (*City*, *State*, *Mountain*, *High-Low*, *Border*). However, the column that has the most reliable information is *State.state_name*, and the other columns contain duplicate and unreliable information. When running Algorithm 2, the NLIDB avoids the use of the *Mountain.state_name* column as it does not contain reliable information and uses the *State.state_name* column instead. Once this is done, the NLIDB proceeds to identify the implicit joins between tables, and finally obtains the following SQL query:

```
1: SELECT Mountain.mountain_name
2: FROM State, Mountain
3: WHERE State.state_name LIKE 'Mississippi' AND
   State.abbreviation = Mountain.state_abbreviation;
```

5. EXPERIMENTAL RESULTS

In the experimental tests, a comparison of our NLIDB and the ELF NLIDB was made [10]. The ATIS [20] and Geoquery880 [20] corpora were used for the tests. For each design anomaly, five queries from the ATIS corpus and five queries from the Geoquery corpus were selected, giving a total of 40 queries. Each of the test cases has the following characteristics: it considers an NL query that involves a fragment of the DB, a design anomaly (created or existing) was considered in any of the tables involved in the query, and the SQL query resulting from NLIDB processing without/with anomaly treatment.

It is important to mention that for the purposes of this project, some queries of the two aforementioned corpora were modified since to perform tests on some design anomalies, columns that did not exist in the tables had to be introduced to simulate the design anomalies.

Table 2 shows the results of the comparative tests described. Our NLIDB correctly answers 20 queries from the ATIS corpus and 20 queries from the Geobase corpus. This is due to the handling of design anomalies since, without this mechanism, the NLIDB would not answer any query correctly.

To test the absence of foreign keys, they were removed from the DB and from the SID. To carry out the test with ELF, only foreign keys were eliminated from the DB schema, and an express configuration was used. ELF was not able to correctly answer any of these queries because, when configured, it uses the foreign keys defined in the DB schema to store them in its dictionary and, thus, to be able to build the required joins in the SQL queries. However, as the DB schema does not have foreign keys defined, it is not possible for ELF to define the necessary joins. Furthermore, ELF does not offer a tool to specify foreign keys in a DB without modifying the structure of its schema.

In the test for the use of surrogate keys, some columns with identifiers were included to simulate surrogate keys, and foreign keys were created with other tables referring to the defined surrogate key. For testing our NLIDB, DBs with the appropriate characteristics were created, and the SID was configured to include the surrogate keys, as well as their relationships with other tables. For ELF, only the DB schema was modified, and an express configuration was used. In this type of query, our NLIDB correctly answered because, through the configuration module for design anomalies, the surrogate keys were defined, and the existing relationships were modified using natural foreign keys. In contrast, ELF was unable to answer correctly, as it was repeatedly unable to interpret the

ISSN 2395-8618

NL query. Other times it misinterprets the query by including Boolean values in queries about flight fares.

Concerning columns used for storing AF calculations, in our NLIDB the test was carried out by creating columns with this characteristic in the DB and additionally specifying these columns in the SID. Subsequently, this anomaly was treated using the configuration module for design anomalies, and finally, the NLIDB used the processing algorithm to detect and process these columns with anomalies. Regarding ELF, the columns with the anomaly were also defined in the DB schema. However, for ATIS DB queries, ELF built erroneous queries be-cause it used tables and columns that were not required in the query, and it also ignored the use of AFs to process the queries. For Geobase queries, ELF was able to correctly answer two queries, as the structure of these queries was easier to understand, and the Geobase schema is much smaller than that of ATIS. Due to the above, ELF was able in these two cases to ignore the column used to store AF calculations and apply the necessary AF.

A test for the use of repeated columns in multiple tables was carried out in our NLIDB by repeating columns that are not foreign keys in different tables and whose information is already defined in another column of another table. In this test, the NLIDB had no problem when using the columns with the most reliable information because, when it found a repeated column, it was already defined in the SID, and the column with the most reliable information was associated with it. On the other hand, ELF obtained a correct query from the ATIS corpus and a correct query from the Geobase corpus. The main reason why ELF got only two successful queries with this type of anomaly is that, when it finds the repeated column in a table closest to the column associated with a search value, it always uses this column; otherwise, ELF can ignore it and build the query using the column that has the most reliable information.

6. Conclusions and Discussion

There are many NLIDBs. Most of these tools use approaches that aim to solve only the problems related to NL processing, leaving aside the problems inherent to the structure of the DB in use. One of the most important aspects of using a DB in an NLIDB is its design. On many occasions, this design may have anomalies; therefore, most NLIDBs would not work properly with a large number of DBs that suffer from this problem.

In this work, a mechanism to treat design anomalies in DBs was presented, which was implemented in a do-main-independent NLIDB [19]. To treat these design anomalies, a configuration module was implemented to introduce information about the anomalies of a DB in the SID. In this way, it is possible to present to the NLIDB a representation of the DB without design anomalies so that it works correctly. The above is important since a tool of this type should not modify the schema of the DB in use since it is often used by various applications, which would also have to be modified. Consequently, algorithms were implemented to process queries involving DB fragments that have design anomalies. These algorithms, in conjunction with additional information on the anomalies contained in the SID, allow the NLIDB to work with a DB with design anomalies as if it were a DB without anomalies.

The absence of foreign keys affects only the creation of joins when processing queries with this anomaly. However, if an NLIDB has no way of specifying relationships without modifying the DB schema, it is very difficult for it to be able to correctly construct SQL queries.

In the tests presented in Section 5, ELF is able to correctly construct only four queries, of which three were from Geobase and one from ATIS. Of these queries, two involved columns for storing AF calculations. This was possible because ELF ignored the use of columns that had this anomaly. However, for anomalies that required more specific processing, such as missing foreign keys and the use of surrogate primary keys, ELF could not answer any queries correctly.

Two more anomalies have been detected in the literature on NLIDBs: the absence of the second normal form of some DB table and the absence of the third normal form. A method for dealing with these two anomalies will be developed in the near future.

Acknowledgment

PhD student Alan Gabriel Aguirre Lam acknowledges the scholarship (Grantee No. 510415) by the Consejo Nacional de Ciencia y Tecnología, Mexico.

References

[1] I. Androutsopoulos, G. Ritchie, and P. Thanisch, "Natural Language Interface to Databases: An Introduction," Natural Language Engineering, vol. 1, no. 1, pp. 29–81, 1995.

[2] O. Pivert and H. Prade, "Handling Dirty Databases: From User Warning to Data Cleaning Towards an Interactive Approach," Proc. Fourth International Conference on Scalable Uncertainty Management, France, 2010.

[3] M.L. Pedro-de-Jesus and P.M.A. Sousa, "Selection of Reverse Engineering Methods for Relational Databases," Proc. European Conference on Software Maintenance and Reengineering, IEEE, 1999.

[4] N. Mfourga, "Extracting Entity-Relationship Schemas from Relational Databases: A Form-Driven Approach," Proc. Fourth Working Conference on Reverse Engineering, IEEE, 1997.

[5] W. Woods, R. Kaplan and B. Nash-Webber, "The Lunar Sciences Natural Language Information System: Final Report," BBN Report 2378, Bolt Beranek and Newman Inc., 1972.

[6] E.F. Codd, "Seven Steps to Rendezvous with the Casual User," Proc. IFIP Working Conference Data Base Management, pp. 179–200, 1974.

[7] G. Hendrix, E. Sacerdoti, D. Sagalowicz, and J. S. Locum, "Developing a Natural Language Interface to Complex Data," ACM Transactions on Database Systems, vol. 3, no. 2, pp. 105–147, 1978.

[8] R.A. Pazos, J.A. Martínez, A.G. Aguirre, and M.A. Aguirre, "Issues in Querying Databases with Design Anomalies Using Natural Language Interfaces," Fuzzy Logic Augmentation of Neural and Optimization Algorithms: Theoretical Aspects and Real Applications, Studies in Computational Intelligence, vol. 749, pp. 461–473, 2018.

[9] M. Minock, "C-phrase: A system for Building Robust Natural Language Interfaces to Databases," Data Knowl. Eng., vol. 69, no. 3 290–302, 2010.

[10] S. Conlon, J. Conlon, and T. James, "The Economics of Natural Language Inter-

ISSN 2395-8618

faces: Natural Language Processing Technology as a Scarce Resource," Decis. Support Syst., vol. 38, no. 1, pp. 141–159, 2004.

[11] X. Boyan, C. Ruichu, Z. Zhenjie, Y. Xiaoyan, H. Zhifeng, L. Zijian, and L. Zhihao, "NADAQ: Natural Language Database Querying Based on Deep Learning," IEEE Access, vol. 7, pp. 35012–35017, 2019.

[12] W. Wang, "A Cross-Domain Natural Language Interface to Databases Using Adversarial Text Method," Proc. VLDB 2019 PhD Workshop, 2019.

[13] C. Baik, H.V. Jagadish, and Y. Li, "Bridging the Semantic Gap with SQL Query Logs in Natural Language Interfaces to Databases," Proc. IEEE International Conference on Data Engineering (ICDE), 2019.

[14] N. Sukthankar, S. Maharnawar, P. Deshmukh, Y. Haribhakta, and V. Kamble, "nQuery - A Natural Language Statement to SQL Query Generator," Proc. 55th Annual Meeting of the Association for Computational Linguistics Student Research Workshop, pp. 17–23, 2017.

[15] J. D. O'Shea, K. Shabaz, and K.A. Crockett, "Aneesah: A Conversational Natural Language Interface to Databases," Proc. World Congress on Engineering 2015, vol. 1, 2015.

[16] Y. Amsterdamer, A. Kukliansky, and T. Milo, "A Natural Language Interface for Querying General and Individual Knowledge," in Proc. VLDB Endow., 2015.

[17] F. Li and H.V. Jagadish, "Constructing an Interactive Natural Language Interface for Relational Databases," Proc. VLDB Endow., 2014.

[18] R. Alexander, P. Ruksha, and S. Mahesan, "Natural Language Web Interface for Database (NLWIDB)," Proc. Third International Symposium, SEUSL, Sri Lanka, 2013.

[19] R.A. Pazos R., M.A. Aguirre, J.J. González B., J.A. Martínez, J. Pérez, and A.A. Verástegui, "Comparative Study on the Customization of Natural Language Interfaces to Databases," SpringerPlus vol. 5, 553, 2016.

[20] Geobase880 Query corpus, https://www.cs.utexas.edu/users/ml/ nldata/geoquery.html, 2020.

[21] G. Sidorov, R.A. Pazos, J.A. Martínez, J.M. Carpio, and A.G. Aguirre, "Configuration Module for Treating Design Anomalies in Databases for a Natural Language Interface to Databases," Intuitionistic and Type-2 Fuzzy Logic Enhancements in Neural and Optimization Algorithms: Theory and Applications, vol. 862, Springer, 2020.