

Florida State University Libraries

2022

Leveraging Object-Relational Mapping to Reduce Entity Impedance in Natural Language Interfaces for Relational Databases

Jess Moorefield



THE FLORIDA STATE UNIVERSITY
COLLEGE OF ARTS & SCIENCES

LEVERAGING OBJECT
RELATIONAL MAPPING TO
REDUCE ENTITY IMPEDANCE IN
NATURAL LANGUAGE
INTERFACES FOR RELATIONAL
DATABASES

By

JESS MOOREFIELD

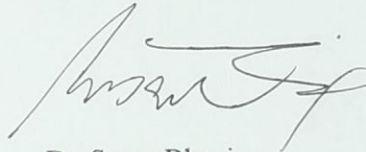
A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the requirements for graduation with
Honors in the Major

Degree Awarded:
Spring 2023

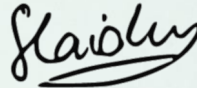
The members of the Defense Committee approve the thesis of Jess Moorefield defended on April 21, 2022.

A handwritten signature in black ink, appearing to be 'CM' with a long, sweeping underline.

Dr. Chris Mills
Thesis Director

A handwritten signature in black ink, appearing to be 'Susan Blessing' in a cursive style.

Dr. Susan Blessing
Outside Committee Member

A handwritten signature in black ink, appearing to be 'Sonia Haiduc' in a cursive style.

Dr. Sonia Haiduc
Committee Member

Problem Statement (Abstract)

This thesis demonstrates the use of an object-relational mapping (ORM) framework to reduce entity impedance in a natural language interface for relational databases (NLI-DB). Specifically, we present a database-agnostic SQL view construction algorithm that partitions an arbitrary database into entities that maximize performance of a natural language understanding (NLU) model. This model represents an important contribution to automating the creation of a natural language interface for a single database in two ways. First, it eliminates ambiguity when mapping a user’s information need to a specific database table. Second, it allows for the use of simple natural language models for generating SQL queries to service a given information need. Practically, our approach aims to increase database access for non-technical users by minimizing their need for SQL knowledge. We accomplish this by mapping a user’s information need to a specific set of database columns exposed by a specific entity view.

Motivation

For many business systems, data is usually housed in an underlying relational database management system (RDBMS), and non-technical users rely upon these systems to access the required information for their work. Moreover, the recent growth of businesses’ interest in data analysis has led to an increasing amount of non-technical users interacting with RDBMSs [1], even if indirectly. However, because the system is navigable only through a database query language such as SQL, users must either have training in the language or submit a request for information from the RDBMS to someone else within the business who has the requisite technical knowledge. Adjusting their workflow to wait until their request is fulfilled or

undergoing on-the-job training in SQL in addition to their position’s non-technical responsibilities can reduce users’ productivity as well as job satisfaction.

Furthermore, even when assuming fluency in a query language, data stored in RDBMSs is not necessarily easy to access. A detailed query might require a user to access multiple different tables that are not obviously related to one another. Therefore, a user might spend a significant amount of time searching through tables to find the name of a field that holds the information they want. The potential syntactic complexity of these queries often requires more than simply a working knowledge of SQL, which erodes the utility of on-the-job SQL training for non-technical positions.

This scenario is of particular interest to our work. It is common for a user to have an information need that relates to an aggregation of database tables, which makes up what the business intelligence community calls a “business object.” As a concrete example, a user might need the address of an employee, but that address might exist in a dedicated **Address** table linked to an **Employee** table by a foreign-key relationship. While the user’s information need exists in the **Address** table, the user’s inquiry is a level of abstraction higher—the employee. We call this problem “entity impedance,” which represents a limiting factor to all approaches to this scenario that work at the level of database tables.

This thesis focuses on resolving the entity impedance problem, which is a major hurdle for achieving automated natural language database interfaces for non-technical users. Here, *entity* refers to an abstract concept that encapsulates data, such as an employee or a work order. *Impedance* refers to the gap between how an entity is mentioned in natural language and how an entity is represented in a database. It also refers to the challenge of navigating between tables in a

database to arrive at desired information that may be conceptually related but dispersed across multiple database tables.

As an illustrative example of the entity impedance problem, a user might input “What is Alice’s address?” to the NLI-DB, where the database contains tables called `Address` and `Employee`. A statistical language model applied to the natural language input might understand that the user is looking for a piece of information called something like “address,” and that the information is likely stored in the `Address` table. However, the `Address` table does not store a field for an employee’s name; that field is found in the `Employee` table. Instead, the `Address` table contains a field called `EmployeeID`, and the `Employee` table also has this field. Therefore, to get the address for an employee named “Alice,” the NLI-DB must first search the `Employee` table for the record where the `Name` field is “Alice,” retrieve the value of the `EmployeeID` field from that record, join to the `Address` table on that value, and then finally retrieve the value of the `Address` field. The entity impedance problem lies in understanding 1) that the join is required, and 2) how to make the join between the tables.

To address this problem, we present a proof-of-concept NLI-DB model built on entity SQL views that aggregates often disparate tables into an abstract entity representation which more closely resembles an entity of common user information needs. Based on this approach, user inquiries accepted in natural language can be translated into statements in a relevant query language and then executed against a database to retrieve the user’s desired information. Practically, this approach frees a non-technical user from the need to learn a query language or engage with an external resource that has query language skills by allowing them to interact with the database through an intermediary NLI guaranteed to produce a valid SQL query in response

to a natural language inquiry. Furthermore, compared to contemporary approaches, our technique shifts the complexity of query generation from runtime to design time by compressing complex join conditions into an SQL view rather than generating them on the fly.

Related Literature

Because a significant semantic gap exists between natural language and a database query language, several different methods have been proposed to accomplish this translation. Takenouchi et al. [1] took an approach of programming-by-example (PBE) to produce an algorithm called PATSQL. PBE is a technique that allows users to create a few examples of their desired output or function of a system instead of writing a program to produce that output [2]. Providing these samples to an interface is easier for non-technical users than attempting to learn the appropriate query syntax for a given database system [2].

The PATSQL algorithm produces queries by looking at an example set of input-output (I/O) tables provided by the user [2]. The I/O tables demonstrate which data the user wants to select from a table within the relational database: the input table is a limited sample from the original table, and the output table contains a sample set of rows with a reduced set of columns that hold the data the user is requesting [2]. The user also specifies the constant(s) value(s) that determine whether a row is present in the output table [2]. For example, if the user wants to select all rows where the value for the column `Month` is “October,” they would specify the constant “October” along with the I/O tables to the algorithm.

PATSQL analyzes the I/O tables to produce a sketch written in the syntax of a Domain-Specific Language (DSL) created by Takenouchi et al, where the syntax corresponds to SQL operations [2]. The sketch outlines the correct step-by-step order of the operations and their

relevant data [2]. The algorithm converts the sketch written in the DSL into an SQL query, and the user executes it against the table. PATSQL supports highly expressive (i.e., specific) queries, allowing users to perform advanced operations through the I/O tables without learning complex syntax in SQL [2].

Unfortunately, programming-by-example requires the user to be very familiar with the RDBMS' tables to be able to generate the I/O tables that demonstrate their request. Even for non-technical users comfortable in their ability to create the I/O tables, a request with many columns would be time-consuming. For every user who has never interacted with the database before, there is a non-trivial overhead to familiarize themselves with all the tables and their schemas. Additionally, schemas may change over time and differ from table to table [2]. Furthermore, Takenouchi et al.'s approach does not account for the fact that schemas may have esoteric naming conventions which users would have to learn to understand what information they can access.

To assist users first learning to interact with a relational database, Wang et al. [3] proposed an NLI-DB that generates a natural language description of a query execution plan (QEP). QEPs are provided by a commercial RDBMS and outline the steps to form an SQL query specific to the vendor's conventions [3]. These aids can be in visual or textual format [3]. However, Wang et al. note that "[textual QEPs] are not user-friendly and assume deep knowledge of vendor-specific implementation details. On the other hand, the visual format is relatively more user-friendly but hides important details." (1) As such, they proposed the LANTERN interface to create natural language descriptions for pairs of an SQL query and its QEP.

Wang et al. developed an algorithm called RULE-LANTERN to accomplish translation. A team of subject matter experts (SMEs) was recruited to label the operators of an RDBMS with natural language descriptions using the POOL framework developed by the researchers [3]. This labelling work must be done by SMEs to ensure a high degree of accuracy [3]. Because all QEPs are made up of some combinations of these operators, RULE-LANTERN stitches the labels together to create a full description for a specific QEP and returns it to the user. [3].

Based on the results of surveys to assess user preference of the visual, textual, and natural language QEP formats, Wang et al. concluded that LANTERN could serve as an effective complement to visual QEPs, namely by pairing the visual’s general overview with a more detailed explanation [3]. However, they received feedback that user interest waned during use of the system; users felt that the descriptions for queries that used the same operations were repetitive [3]. Reading the same words repeatedly led to boredom, and the users “reported that they started skipping several sentences in the descriptions.” (8) To mitigate this effect, Wang et al. developed a deep-learning framework within LANTERN called NEURAL-LANTERN to include language variability in the natural language descriptions and found positive user response [3].

Even with the aid of a detailed QEP created by LANTERN, the complexity of SQL knowledge is still on the shoulders of the non-technical user—the NLI is only designed to be a reference. Moreover, there is no guarantee that a user will sufficiently understand the QEP to successfully execute a query, and so they may still require the assistance of someone else specifically trained to access the database. This is particularly true for unfamiliar or advanced queries. Moreover, the necessary addition of NEURAL-LANTERN underscores the importance of understanding that non-technical users want to minimize the effort needed to operate the NLI;

a prolonged degree of high concentration on a repetitive task is unsustainable. Feelings of boredom can lead to both negative user reception of the NLI and to reduced ability to successfully execute queries. Thus, a tool designed for non-technical users must acknowledge that ease of use is informed by motivation to use the interface.

Brunner and Stockinger [4] proposed two natural language-to-SQL systems with a focus on inquiries that do not clearly indicate the correct values to be included in a `WHERE` clause. The ValueNet light system is an end-to-end, encoder-decoder neural architecture that accepts as input both the database schema and a list of candidate values generated from the natural language inquiry. The neural network learns to select the correct candidate value and learns from the database schema to construct an intermediary output in the form of abstract syntax trees in the context-free grammar SemQL [4]. The post-processing step transforms these trees into Select-Project-Join SQL statements with a `WHERE` clause including the value [4]. ValueNet light then executes the statement and provides its results to the user.

ValueNet builds upon ValueNet light to include a pre-processing step that accepts the natural language inquiry as input rather than the list of candidate values. ValueNet extracts the inquiry's values and generates reasonable candidates through a combination of heuristics and named entity extraction [4]. ValueNet also receives hints about the database schema during pre-processing [4]. Its output is equivalent to ValueNet light [4].

Brunner and Stockinger's approach relies on a long period of training for the neural architectures to resolve the impedance problem, including cross-training for the 200 databases within their dataset [4]. In particular, the amount of training required for the network to learn to generate join statements can entail a large overhead. If the network is learning simply to

associate values with column names for individual tables, the model would require a large set of training data to learn to join on columns that hold equivalent data but do not have consistent naming across the tables. This method of learning is more likely to fail in contrast to determining a join based on the underlying structure of the schema such as foreign-key relationships.

ValueNet and ValueNet light also leave the method of providing the database schema open-ended, and how the schema is provided to the network plays an integral part of how the network learns to generate the joins.

Furthermore, the logic of the joins is invisible to the user of the systems, which can spell trouble for questions that involve high levels of entity impedance by asking for information across multiple tables that are not obviously related to one another. While ValueNet light and ValueNet might generate valid, interpretable SQL queries, if they include an incorrect join statement, the user may not understand the results returned. If a user does not recognize how the system made the connection between the tables, they likely would not know how to rephrase their question to receive the correct results.

Approach

This thesis provides a proof-of-concept model for an alternative NLI-DB technique informed by the shortcomings previously discussed. Our work partially replicates the method introduced by Alghamdi et al. [5] of using an object-relational mapping (ORM) framework connected to a user interface that receives natural language input. We advocate that Alghamdi et al.’s methodology is preferable because it best reduces the complexity and length of a non-technical user’s interaction with the database and limits the difficulty of directly translating natural language to a query language (e.g., SQL) at runtime. As opposed to PATSQL or LANTERN, which require some form of interaction with the database, the user’s only

responsibility is to provide an inquiry in natural language to indicate what information they are requesting. Additionally, unlike ValueNet and ValueNet light, our model does not require extensive training of a neural architecture to understand the database schema. Moreover, we can utilize our entity views to automatically extract training data for a natural language conversational model (which is left for future work.)

A similar, alternative approach to using an ORM is to build one's own mapping of all connected tables through analyzing their foreign-key relationships. However, an off-the-shelf ORM provides a quick, cost-effective solution to mapping the relational structure of a database. We chose this approach to limit the number of heuristics that must be constructed and tuned for the current project.

Alghamdi et al. used the Hibernate framework in their work [5]; however, because Hibernate is an older, more inflexible ORM, our model uses Microsoft Entity Framework (EF). We refer to our application of the framework as the *ORM engine*. The database we supplied to the engine is the Microsoft AdventureWorks2019 database, hosted by Microsoft SQL Server implemented on an Azure Virtual Machine. AdventureWorks is well-organized and populated with cleaned data that does not use encoded or prohibitively esoteric naming conventions. As such, we selected the database to simplify our methods by ensuring that the fidelity of the model is not compromised by flaws within the RDBMS itself. AdventureWorks is also widely used for training and as a testbed for SQL and related technologies. Therefore, by being tuned to this database architecture, our approach should natively support any arbitrary database built with the same best practices in mind.

The object models generated by the ORM engine are C# classes, where each table in the database is mapped to a class. The engine can read structural components of the database schema, such as foreign-key relationships, and encode this information within the classes as C# properties. Using these object models to understand relationships between entities in a user inquiry and in tables within the database, we developed a reverse engineering process to create entity views that aggregate these relationships at design time by precomputing joins and join conditions.

Specifically, we developed a script to automatically generate a set of SQL views, where each view encapsulates the data of an object model. The script is driven by an algorithm that extracts the fields and properties from the class declarations and inserts them into a series of pre-written SQL statements. The statements are known to produce valid, interpretable queries and retrieve the necessary information to construct the view. This includes retrieval of a table's schema, primary key, and columns; determining the appropriate column (i.e., key) on which to join two tables; and validating the column selected for the joins. The algorithm also determines whether an inner or left join statement should be used. The information returned from these queries is then inserted into the statement that creates the view.

Each view can be considered an entity, as it centralizes information from multiple different tables across the database to encapsulate one larger concept. For example, a view named **Employee** would represent the concept of an employee. An employee might have an ID number, a job title, an address, a history of sales, and many other characteristics. In the database, these pieces of information may not be stored all in the same table (i.e., a user inquiry for this information natively experiences entity impedance), so the **Employee** view creates a central collection for the information. Our view-generation algorithm automatically partitions the

database into these entities that can then be mapped to entities within natural language utterances. Retrieving the user's desired information by querying the views thereby simplifies the SQL statements the model must generate on-the-fly.

We note that there can be overlap between entities in this approach. Similar entities might encapsulate some of the same data, and as such, it is likely that some user inquiries may ask for information that more than one view contains. We account for this overlap by using intent recognition while training the model on natural language input. Furthermore, even in the presence of ambiguity that cannot be resolved by training with our intent recognition approach, the data presented by each overlapping entity is equivalent. Therefore, as long as any entity contains the data, that data is reachable if the training data contains at least one exemplar.

A view includes a `SELECT` statement for all columns of the current table and all columns of each table it holds a foreign-key relationship with. These columns are followed by a series of corresponding join statements. The completed statement outputted by our view-generation script follows the format:

```
CREATE VIEW nli.<CurrentTableName> AS

SELECT <CurrentTableSchema>.<CurrentTableName>.FieldName AS
<CurrentTableName_FieldName>
...

<JoinedTableSchema>.<JoinedTableName> FieldName AS
<JoinedTableName_FieldName>
...

FROM <CurrentTableSchema>.<CurrentTableName> extent0

<JoinType> JOIN <JoinedTableSchema>.<JoinedTableName> <extentn>
ON extent0.<JoinColumn> = <extentn>.<JoinColumn>

...;
```

Where $n = 1$, increasing for each join. The script then automatically executes the statements against the database to create each view.

We developed a set of four heuristics to determine the correct column for every join. The algorithm first searches for an exact match of a column name between two tables. If no match is found, the algorithm looks for a column named `ID` in the first table and a column named `<Table1Name>ID` in the second table. If no pair exists, it looks for a column named `<Table1Name>ID` in the first table and a column name that contains `<Table1Name>ID` in the second table. If these heuristics fail, it is likely that the primary key of the first table has an arbitrary name, and so the algorithm lastly retrieves this primary key and searches for a column named `<Table1Name>ID` in the second table.

Each pair produced by these heuristics is added to a list that contains all sets of candidate columns for the join. The candidates are then validated in a two-step process: a check that there is at least one row in an inner join between the candidates and a check that there are no orphaned rows in a left join between the candidates. The algorithm selects the first set of candidate columns that meet both validation criteria.

With the join columns selected, the algorithm next determines whether an inner or left join statement should be selected. Both joins are executed against the database to count the respective numbers of retrieved rows, and the algorithm takes the join type that has the maximum number of retrieved rows. The intent behind this design choice is to maximize the scope of the universe of information contained in the entity views, thereby maximizing the likelihood of a suitable answer for any end-user query.

The heuristics are independent of the database they are applied to; our approach is natively portable to other RDBMSs. For the AdventureWorks2019 database, our algorithm has a 93.976% success rate, with a column validated for 156 of the total 166 joins in the views.

However, it is important to note that we observed shortcomings to this approach. Our testing shows that the set of heuristics is incomplete, as there were some join operations that the algorithm could not determine the appropriate column for. One major contributor to issues with our heuristics in this work is a naming convention in Entity Framework that pluralizes tables when creating model classes. In our case, this causes an issue with tables that are already plural being pluralized again, resulting in table names that defy assumptions in our first pass of heuristics. Furthermore, there are scenarios in which more than one of our join condition heuristics matches. In these scenarios, we choose the *first*, and not necessarily the *best*, join columns. Because of this issue, we found that some views are not “complete” in that they do not provide the answer to any possible user inquiry.

Additionally, for one table, `SalesTerritory`, the final join statement of the view generated an integer overflow error. This error was due to our use of a `COUNT` operation to retrieve the respective numbers of retrieved rows. To resolve this issue, we need to improve the validation criteria such that a full row count of the join (i.e., reading the join’s entire universe of information off of disk) is not required as a kill condition.

The second component of our model is the use of a natural language understanding (NLU) platform to parse a user’s inquiry and output the information retrieved by an SQL query. We used Google’s Dialogflow tool, an off-the-shelf interface with extensive training on natural language data and automatic cleaning and tokenization of text. Although the platform accepts both speech and text utterances, we formatted all training phrases for the model as written

inquiries. Designing our model with Dialogflow enabled us to narrow this thesis' scope by leveraging a pre-made tool instead of developing our own interface, which was Alghamdi et al's approach [5].

Using Dialogflow, we created a conversational agent to perform intent recognition and named entity extraction on a set of training phrases. The intent of a phrase corresponds to the name of the view that should be queried. For example, if the intent of an inquiry is to ask for information about an employee, the request should be serviced with a query that retrieves information from the `Employee` view. Concretely, the intent of a user utterance tells us the value of the `FROM` clause in an SQL query. We note that many different questions can have the same intent, as one view can contain many fields. Our approach compresses the complexity of resolving this ambiguity by answering a simpler question: given a finite number of entity views, which view can service an information need based on intent recognition?

The conversational agent then applies named entity extraction to retrieve the target entity, which represents the specific data request (i.e., the column list for the `SELECT` statement), and the `WHERE` clause entity, which provides the value that narrows the scope of the specific data request. Using our previous example, "What is Alice's address?", the intended entity view is `Employee` (as opposed to `Address`), the target entity is "address," and the `WHERE` clause entity is "`Name = 'Alice'`," as the user has requested a piece of data that is specific to someone named Alice. Once the intent of a phrase is known, the target entity is mapped to a field name within the view.

Together, the intent, target entity, and `WHERE` entity are organized into a single `SELECT` statement, which takes the form

```
SELECT <Target>

FROM nli.<ViewName>

WHERE <Where Entity>;
```

The model then executes the query against the database and returns the results to Dialogflow to be displayed to the console. These methods replicate the same general sequencing of Alghamdi et al.'s [5] engine algorithm to translate a natural language utterance into a SQL query and produce a conversational agent response.

A critical advantage of our model is its strong control over the SQL statements executed against the database. This control is exerted in both the algorithm to generate the SQL views from the ORM classes and in the algorithm to generate SQL queries from a user utterance. By inserting objects from the classes or entities from the utterance into pre-written SQL statements, the model executes only queries that are known to be valid and interpretable. Our approach guarantees that every query executed against the database will produce some output, even if it is not necessarily correct. Therefore, in response to a question, the model will always provide the user with *some* answer rather than an error. Receiving incorrect data that is still related to a user's question may make it easier for a user to recognize how to rephrase their question to retrieve the precise information they desire.

Looking at the previously discussed approaches for NLI-DBs, we hypothesize that our model produces better results than a model with a cross-training approach, namely higher fidelity and a shorter training period. This is due to the absence of vocabulary mismatch when training a model with data from an unrelated domain. A cross-trained model must learn to accept utterances and produce queries that are not relevant to all databases in the training set, as

databases are domain-specific. Such a model therefore requires much more training data to achieve equivalent levels of entity impedance across all databases. Additionally, through intent recognition performed on training phrases, our model answers a user's question by executing a SELECT query against the intent's corresponding SQL view. This simplified process contrasts with a model that must search the database for any relevant fields and generate the corresponding joins on-the-fly. A model's ability to reduce entity impedance is fundamentally dependent on the accuracy of the joins it produces; a high-fidelity model is therefore more difficult to achieve when joins are not already known at the time of query generation.

Future Work

Further development of this model could be approached through improving the view-generation algorithm and through improving the training phase for the NLU. The set of heuristics in the view-generation algorithm could be expanded to minimize the number of failed joins as well as to address the pluralization and join condition validation issues. The algorithm could also be further developed to improve nested composition by supporting entities that have an arbitrary level of compositional depth with other entities.

Future work to improve the NLU's training will focus on automatic generation of database-specific training data by sampling data returned from the entity views. Because the entity name and each column name and value are known by construction, it is possible to automatically generate question-and-answer pairs from that data for use in training an initial natural language conversational model. The next honors-in-the-majors thesis on this topic seeks to perform this task.

Conclusion

This thesis presents a proof-of-concept model that leverages object-relational mapping to reduce entity impedance in a natural language interface for relational databases (NLI-DB). Our model is driven by a first-pass view construction algorithm that partitions the database into entities that can be mapped to entities within natural language utterances. We have focused on a single such entity: `Employee`. Through the Dialogflow NLI, our model categorizes that entity as a specific intent and then analyzes training phrases to determine if they are asking for information about the `Employee` entity. The model performs entity extraction on the training phrase to generate a SQL query and return the results to Dialogflow’s console.

Our approach offers a model with strong fidelity because we are guaranteed to produce a valid, interpretable SQL query in response to the training phrase. The result returned from the query may not necessarily be the correct answer to a user’s question, but the model will always return *some* answer. Moreover, that answer will provide only information contained within the entity mapped to the utterance’s intent. We also hypothesize that our model will achieve higher fidelity than those with cross-training approach because databases are domain-specific.

This thesis makes an important contribution to a larger effort to automate the construction and training of an NLI-DB for a single database. By shifting the complexity of SQL knowledge off the user, NLI-DBs have the potential to be immensely useful to non-technical users who interact with an RDBMS. As more and more information is being stored in RDBMSs and is being retrieved by an ever-increasing number of people, we believe that this research poses a valuable contribution to address the need for systems that are designed for technical and non-technical users alike.

References

- [1] Takenouchi, K., Ishio, T., Okada, J., & Sakata, Y. (2020). PATSQL: Efficient Synthesis of SQL Queries from Example Tables with Quick Inference of Projected Columns. *arXiv preprint arXiv:2010.05807*.
- [2] Fariha, A., Cousins, L., Mahyar, N., & Meliou, A. (2020). Example-Driven User Intent Discovery: Empowering Users to Cross the SQL Barrier Through Query by Example. *arXiv preprint arXiv:2012.14800*.
- [3] Wang, W., Bhowmick, S. S., Li, H., Joty, S., Liu, S., & Chen, P. (2021, June). Towards Enhancing Database Education: Natural Language Generation Meets Query Execution Plans. In *Proceedings of the 2021 International Conference on Management of Data* (pp. 1933-1945).
- [4] Brunner, U. & Stockinger K. (2021). ValueNet: A Natural Language-to-SQL System that Learns from Database Information. In *Proceedings of the IEEE 37th International Conference on Data Engineering* (pp. 2177-2182).
- [5] Alghamdi, A., Owda, M., & Crockett, K. (2017). Natural language interface to relational database (NLI-RDB) through object relational mapping (ORM). In *Advances in Computational Intelligence Systems* (pp. 449-464). Springer, Cham.