UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Final Paper 2020

Title: Visual Query Formulation: Generating SQLP queries using EER diagrams

Author: Bradley Malgas

Project Abbreviation: KnowID

Supervisor(s): Maria Keet

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	10
Theoretical Analysis	*0*	*25*	0
Experiment Design and Execution	*0*	*20*	0
System Development and Implementation	*0*	*20*	20
Results, Findings and Conclusions	*10*	*20*	20
Aim Formulation and Background Work	*10*	*15*	10
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	0
Total marks		**80**	**80**

Visual Query Formulation: Generating SQLP queries using EER diagrams

Bradley Malgas
Department of Computer Science
University of Cape Town
South Africa
MLGBRA002@myuct.ac.za

ABSTRACT

Visual querying is a method that makes use of visual components to allow users, especially those without experience in structured query languages to formulate queries. Users are provided with a query environment which they can use to interact with any visual elements to put together queries that will be transformed into the relevant querying language associated with the database which they are querying. A visual query system was created to allow users to load extended entity relationship diagrams and use them to construct queries. It was found that while this is possible, the checking of valid attribute paths is a dynamic process that requires a large grammar and is limiting factor of the queries that can be created.

1 INTRODUCTION

Intelligent data access is often limited by a user's technical knowledge and experience with data storage and querying languages. In most cases, users without that knowledge (such as those in other scientific fields) who require custom data queries must rely on skilled IT professionals to construct it for them. In order to simplify the data access without requiring users to learn structured query languages, a visual query system is proposed to facilitate querying of a database. The visual query system will allow users to interact with an interface that will allow them to intelligently create complex queries. These queries, over an EER diagram, will be generated in SQLP and can then be run over a database. We aim to provide users with extended entity relationship (EER) diagrams which they can use to construct queries in an extended version of regular Structured Query Language (SQL) known as SQL^{path} (SQLP). The solution is a React WebApp that allows users the ability to load models and while it only supports a subset of queries, we explore what this means for visual query systems and the design and implementation taken to create this tool.

2 BACKGROUND

2.1 SQLP

SQLP was developed in order to simplify the syntax required to construct table joins. In doing so, it speeds up the construction of conjunctive queries [2]. It does so by incorporating *attribute paths* as an extension to the base SQL. This is achieved by making use of abstract tables which use an abstract primary key of *self* [of datatype of OID (object identifier)]. Each abstract table has only one primary key, *self*, and makes use of a constraint *pathfd* where we can either specify concrete attributes in the current table or concrete attributes from other tables. This allows us to specify multiple foreign key joins using dot-notation to separate the attributes. To illustrate this, consider the example of a university system depicted in abstract tables shown in *Figure 1*.

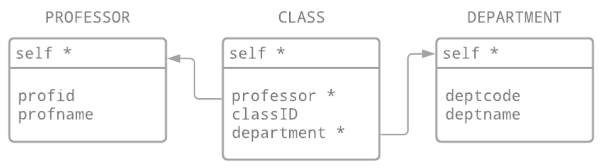


Fig. 1. ARM schema showing a simplified university system

In this example, we see that the abstract table `CLASS` contains foreign key references to both the `DEPARTMENT` and `PROFESSOR` abstract tables. This means that `self`, `classid`, `professor.profid` and `department.deptname` are all considered to be valid attribute paths for the CLASS table, whereas `profname` or `deptcode` are not since `profname` and `deptcode` are not contained as attributes in the table `CLASS`. This notation is visibly shorter than the SQL equivalent. This shorthand notation has been shown to reduce errors in query construction [2] as well as reduce the time taken to construct a query [1] when compared to SQL. For these reasons, queries in the visual query system are constructed in SQLP.

2.2 Visual Query Systems

Visual queries are carried out by using a Visual Query System (VQS). A VQS is a system comprised of (1) A visual querying environment containing the visual elements the user interacts with and (2) A Visual Query Language that dictate how the user will interact with query environment. The VQS will visually represent the data that can be queried and the actual requests for data.

There is currently no support for constructing SQLP queries in any of the existing visual query systems. Existing systems that use structured query languages, such as the ViziQuer [5], use SPARQL. Another possible query language is EQL-Lite used in the WONDER system [4].

Additionally, VQSs that use EER diagrams to facilitate the query creation are not as popular with the more predominant approach being graph-based query systems. The systems, represent the data as graphs and allow the user to traverse the graph, selecting data to use in the query. One such system is the GBLENDER [3] system. The user interface of GBLENDER allows users to drag labels from a label pane into a graph query pane. Edges can be added to link together different labels. While the user adds more and more labels, the current query gets built and is processed using the GBLENDER algorithm. Another example is the WONDER system [4]. In this system, the user interacts with a query graph which represents the ontology that it is based on. Nodes are dragged from the ontology graph and can join them in a query pane.

3 SYSTEM REQUIREMENTS & DESIGN

The software can be divided into 3 separate parts:

(1) The EER/ARM transformation

This is tool has been loaded over from the previous years' project in which a JSON file can be loaded into a user interface and then appropriately represented as either a EER diagram or an ARM which can then subsequently be transformed from one to the other.

(2) Querying interface

The EER diagram is used as the main querying diagram. Users can interact with the attributes represented graphically, selecting those which they want to query, and a subsequent SQLP query will be generated and checked for correctness once a user indicates that they have selected all required elements.

(3) Background query checking

In the background, using the provided values, the system will use a grammar to assess whether the selected parameters constitute a valid query. This is done by passing the values to a grammar of the SQLP query structure and the result is passed back to the user interface.

[algorithm design = encoded grammar; explain how it is will be assured to be the same query over arm]

3.1 System workflow

The system is designed so that multiple components will be linked together in order to get the workflow from an input file to an SQLP query. Each component is a standalone software tool and together they work in order to create visual queries.

The software tool is designed to take in JSON file as input. The system does not check whether the provided file is of the correct format as it is assumed the user has the correctly formatted file and is only concerned with using it to construct a query.

The JSON file will contain all the details about a specific EER model including all `entities, relationships, attributes` etc. These will all be contained in the JSON file. The interface will allow the user to load this file and represent it diagrammatically. The model represented will also be interactive, giving users the ability to move different elements around as well as click on them to create queries.

3.2 Supported Queries (Requirements)

The system will allow users to construct a limited set of queries. This is mainly demonstrating the actual flow of information from the loading of the EER model to the final query. The range of supported queries can be built upon by tweaking the grammar.

*Example Query 1: **type 1 query***
Using the same example of a university system introduced in *Figure 1*. Let's say a user wants to retrieve the list of all `deptcode` from `DEPARTMENT`:

This query in SQLP would be:

```
select distinct deptcode from DEPARTMENT
```

This is the first kind of supported query. Throughout this paper, we will refer to this as a "type 1" query.

*Example Query 1: **type 2 query***
Let's say a user wants to retrieve the list of all `class` offered by a specific `DEPARTMENT`:

This query in SQLP would be:

```
select distinct classid from CLASS
where class.department.deptname == "CS"
```

This is the second kind of supported query. Throughout this paper, we will refer to this as a "type 2" query.

3.3 Interaction Design

The user will interact with the system by click on specific buttons. The key part of this interaction is order. Since the order in which users select attributes will affect what they are effectively trying to query.

When selecting attributes, this is affectively just choosing which elements you wish to query. These attributes must be selected in the order which they are designed. To create *Example Query 1*, a user would have to click on `deptcode` and the interactions would be done. The user can skip through all other select prompts and just generate the query.

However, *Example Query 2* contains a "where" clause. The user would need to firstly click on `classid`. But unlike *Example Query 1*, once the user is done, they need to then select the "WHERE" clause. To do this, the user would click on `class`, then `deptname`

3.4 Architecture Design

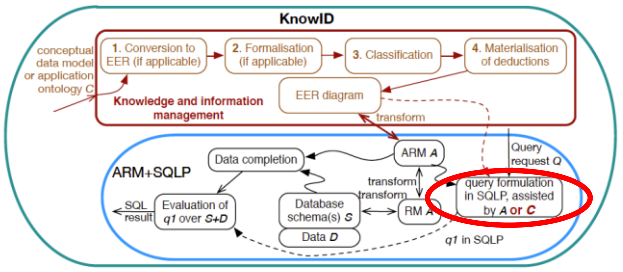

KnowID
conceptual data model or application ontology C
1. Conversion to EER (if applicable)
2. Formalisation (if applicable)
3. Classification
4. Materialisation of deductions
EER diagram
Knowledge and information management
transform
ARM+SQLP
Data completion
ARM A
Query request Q
transform
transform
query formulation in SQLP, assisted by A or C
SQL result
Evaluation of q1 over S+D
Database schema(s) S
RM A
Data D
q1 in SQLP

Fig. 2: The KnowID architecture, with the focus of the project circled in red

The system will be building on a tool designed for the KnowID architecture. The aim is to have this tool form part of the overall process as shown in *Figure 2*.

4 SYSTEM IMPLEMENTATION

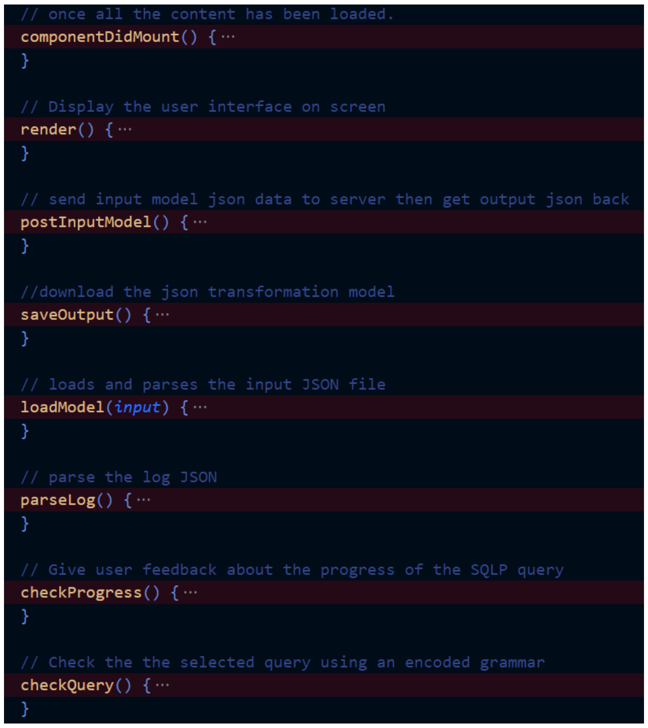
The software was based on the KnowID EER to ARM transformation tool developed by Maria Keet [6]. The system is developed using Python and JavaScript. The libraries/frameworks that were used to support it are React, JointJS & Flask. Each library played a hand in developing components of the system.

Flask: This framework was used to facilitate python queries in JavaScript. The JavaScript file makes an API call to the WebServer backend that is running through Flask and can also return results back to the JavaScript file.
React: This library was used to develop the user interface and run it as a web application. The main application is a React app coded in JavaScript.
JointJS: This diagramming framework was used to assist with representing the EER models and also helped make it interactive.

The various components that were developed will now be discussed in detail.

4.1 User Interface

The user interface for the software tool are shown in *Figure S1*. The user interface is simple and contains a blank canvas (where the EER diagram will be loaded), a header and a footer containing a button. The top heading indicates to the user that they need to load a new model in order to begin the querying process. The "Load model" button is used by the user to load input. The structure of this interface is defined in HTML inside of the *render()* method shown in *Figure 3*.



```
// once all the content has been loaded.
componentDidMount() {⋯
}

// Display the user interface on screen
render() {⋯
}

// send input model json data to server then get output json back
postInputModel() {⋯
}

//download the json transformation model
saveOutput() {⋯
}

// loads and parses the input JSON file
loadModel(input) {⋯
}

// parse the log JSON
parseLog() {⋯
}

// Give user feedback about the progress of the SQLP query
checkProgress() {⋯
}

// Check the the selected query using an encoded grammar
checkQuery() {⋯
}
```

Fig. 3: App.js file with methods highlighted in red

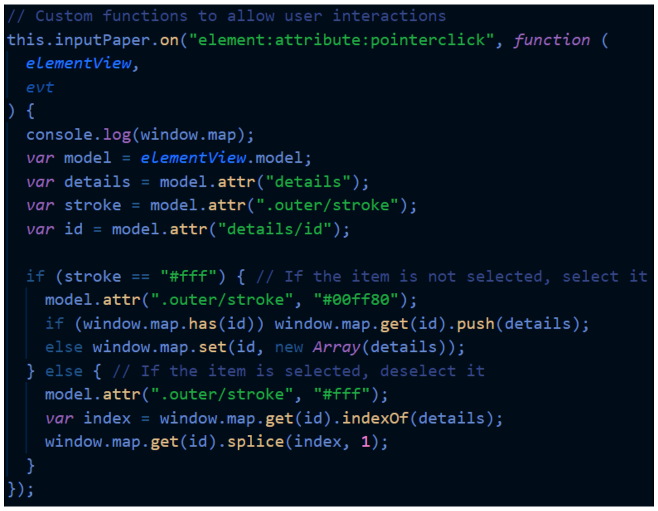
When a user select the "Load model" button as shown in the interface, the *onClick()* method of the button is called and brings up the choose file dialog, allowing a user to selected a JSON file. It should be noted that the assumption is that the user will choose a JSON file of the correct structure. Once the file is loaded, this triggers the *loadModel(input)* with the input JSON file being passed as an argument. The *drawERGraph*(*graph, inputFile)* is then also called and is used to draw the graph on the canvas.

4.2 Query Process

Once the model is loaded, the user can now begin to construct a query. Note, the user is informed that once loaded they will need to rearrange the elements of the model first and this may incur accidental presses however these can be undone. The top heading of the user interface has now updated to indicate to the user that they need to select attributes which they wish to query.

The querying process is split up into two parts:
(1) Select attributes to view
(2) Selecting the constraints on the results obtained

In order to select an attribute, a user simply needs to click on it. This will do two things: firstly, it triggers an event on the canvas that will change the outline of the attribute chosen and additionally, it will add the details about the attribute to a *Map* structure with key being the `id` associated with the element and the value being the array of details.



```
// Custom functions to allow user interactions
this.inputPaper.on("element:attribute:pointerclick", function (
  elementView,
  evt
) {
  console.log(window.map);
  var model = elementView.model;
  var details = model.attr("details");
  var stroke = model.attr(".outer/stroke");
  var id = model.attr("details/id");

  if (stroke == "#fff") { // If the item is not selected, select it
    model.attr(".outer/stroke", "#00ff80");
    if (window.map.has(id)) window.map.get(id).push(details);
    else window.map.set(id, new Array(details));
  } else { // If the item is selected, deselect it
    model.attr(".outer/stroke", "#fff");
    var index = window.map.get(id).indexOf(details);
    window.map.get(id).splice(index, 1);
  }
});
```

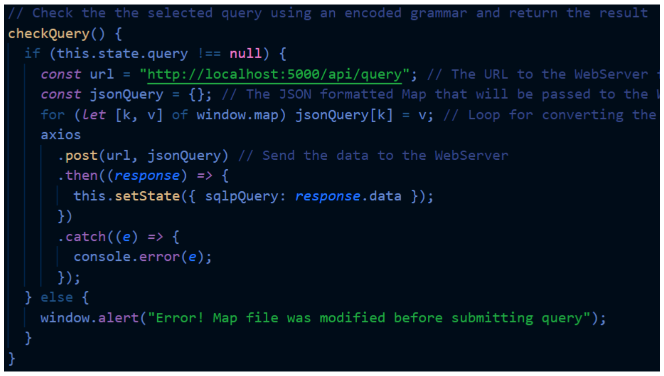
Fig. 4: App.js file showing the mapping of an attribute click to a function

If a user accidentally selects an attribute, they can deselect it by clicking on it as well. Both of these functions are shown in *Figure 4*. Constraints are selected in the same way; however, the user first needs to click the "Finish attribute selection" button. Similarly, once a user has finished selecting the constraints, they can click the "Finish WHERE clause selection" button. After the user has completed all their selections, they can click on the "Generate SQLP Query" button which will the *checkQuery()* method that conducts the Backend query check.

4.3 Backend Check

The backend check is done by the *checkQuery()* function which consists of cleaning a call to the WebServer the current *Map* object to be passed as an argument to the Python file that will check the query for correctness.

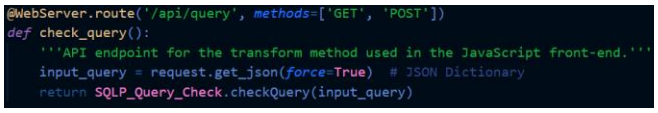
This is achieved by firstly, converting the *Map* object into a JavaScript object. We do this by firstly, declaring the constant `jsonQuery` which will store the actual *Map* data. This loop is shown in *Figure 5*.



```
// Check the the selected query using an encoded grammar and return the result
checkQuery() {
  if (this.state.query !== null) {
    const url = "http://localhost:5000/api/query"; // The URL to the WebServer t
    const jsonQuery = {}; // The JSON formatted Map that will be passed to the W
    for (let [k, v] of window.map) jsonQuery[k] = v; // Loop for converting the
    axios
      .post(url, jsonQuery) // Send the data to the WebServer
      .then((response) => {
        this.setState({ sqlpQuery: response.data });
      })
      .catch((e) => {
        console.error(e);
      });
  } else {
    window.alert("Error! Map file was modified before submitting query");
  }
}
```

Fig. 5: App.js file showing the checkQuery() method

Once the *Map* object is correctly formatted for passing, it is given as for an argument, alongside the *url* for a `POST` to the WebServer. The WebServer retrieves the input from the `request` received from the `POST`. This input, is then passed as an argument for a call to the method *checkQuery()* from the `SQLP_Query_Check` Python module. This is shown in *Figure 6*.



```
@WebServer.route('/api/query', methods=['GET', 'POST'])
def check_query():
    '''API endpoint for the transform method used in the JavaScript front-end.'''
    input_query = request.get_json(force=True)  # JSON Dictionary
    return SQLP_Query_Check.checkQuery(input_query)
```

Fig. 6: WebServer.py file showing the check_query() routing method

At this point, we can actually begin checking whether the selected attributes and clauses form a valid SQLP query. The Python module, SQLP_Query_Check has a single method *checkQuery* which takes as input the data from the original *Map* object and extracts information from it. The data extracted is the same as shown in *Figure x* and follows the logic shown in *Figure Sx*. This method is an encapsulation of a grammar and checks whether the given values can be used to form one of the supported queries. The reason why a Python program was chosen over a traditional grammar is explored in the Results.

The output of this program is either a syntactically and logically correct SQLP Query or a detailed error message and this is shown in *Figure S2*. At this point the query formulation is complete and a user can choose to use their query elsewhere or create another

5 RESULTS

The results obtained from the testing shows us that while the overall system design and related components work well together, the grammar representation is not flexible and dynamic enough to cover the broad scope of SQLP queries. This is due to the fact to the rules need to be adjusted dynamically with variable names constantly changing. The accuracy of queries can therefore not always be guaranteed, as only a subset of queries are supported. These among other observations were made and are explored below.

5.1 Attribute Paths & Grammar

One of the biggest strengths of SQLP is the ability to use the shorthand notation in order to simplify queries. As explained in the *Background*, this is made possible through the use of attribute paths. Since the software is meant to serve as means of simplifying the querying process, the users are not expected to have any understanding of how attribute paths function and are more than likely not going to focus on this during query construction. This leaves much of the system checking to be done by the encoded grammar. The dynamic nature of the problem is not well suited to a grammar but would much rather benefit from a rule-based front-end system that could conduct all the checks while the user selects the query (This is one area for future work). The speed at which queries are checked is not adequate. This could be the manner in which the calls take place in the workflow shown in *Figure x*. This workflow could be further optimized to minimize the required components and allow for asynchronous checking of queries while a user interacts with the interfaces, however, time did not allow for this.

The system does however, meet the minimum requirements set out. It allows the user to load any JSON file containing the representation of an EER diagram, use that model to interact with the diagram and using the provided interface come up with a SQLP query. This proves that despite there being obvious areas for improvement, the software can achieve what it set out to albeit only for a small subset of user queries.

5.2 Query Accuracy & Performance

The system was able to pick up on a few obvious querying errors that users might make. The test cases cover a number of scenarios but that is only for the supported queries. The system would not be able to detect errors where the queries become increasingly complex which does limit its applicability. The queries that it can correctly generate are simple enough to not explicitly need the user interface. However, it should be noted that the time taken to generate the average query is *insert time* seconds. This means that users can benefit from having the ability to conduct multiple queries in a short space of time.

5.3 User Interface

The user interface benefits most from being interactive however it isn't without any issues. Users are guided by the button prompts as to what they have to do but ideally this process should take place in one step. The interface itself also lacks the facility for allowing users to undo mistakes intuitively. It was made clear from the tests that the process would be much easier if there was an undo button for stepping back in the query instead of being forced to start over.

The user interface however does prevent most errors. The intuitive alert messages are somewhat useful however there could have been more action taken to prevent the errors from happening at all. This however was not prioritized due to time constraints.

6 DISCUSSION

These results are relevant to the future development of VQS. It demonstrates that it is possible to make use of a different inputs to construct queries. By showing that the transformations that take place between EER and ARM models can be used to construct SQLP, which is developed for ARM, we can apply this concept to a number of other knowledge graphs (e.g. UML, ORM). They would simply need to have the support for transformation between there JSON encoding and the format used for the EER diagram. The inverse is also a possibility, since the JointJS library treats all shapes in the same way, any knowledge graph represented in this way would only need to call the relevant event listeners developed in for the software project in order to allow it to become interactive and subsequently be used as the querying model in place of the EER diagram.

The aim of creating a visual querying environment for constructing SQLP queries has only been partially achieved. Like SQL, SQLP supports constructs such as “AND” to link multiple query constraints together. Comparisons also extend beyond simple equality with greater than (>), less than (<) and other operands also being supported. These areas are where the software tool falls short. It lacks the ability to account for this wide range of queries and would greatly benefit from an expanded grammar. In its current state, multiple valid queries would be flagged as incorrect. This gap in the tool will be part of the future work.

The flexibility of the tool developed does mean that it can built upon to develop a fully functional querying system. For example, instead of simply displaying a query, if the output is correct it can be passed as an argument to a relevant tool to convert it to SQL which can be used to query a live database. This extension to the program would allow users to retrieve the results of the generated query as well which would vastly improve the usefulness of the software tool.

6.1 Limitations

If time were not a limiting factor, a number of changes could have been introduced. Firstly, the querying time would decrease if user did not have to select query components in parts. The ideal tool should allow it to seamlessly transition between clauses and have the user only click a button when they need to construct the query. Another limitation stems from the chosen diagramming JavaScript library. While it benefits from having many of its shapes be represented in the same way, this is also where it falls short. The listening events, that allow user to click on elements, are applied to all shapes in the diagram. This makes it difficult to have unique ways of dealing with specific object types and events that they throw and also means many attributes cannot be changed on a shape after it is instantiated. A library that has more support for methods that allow for interacting will simplify the amount of code required as well as open up new possibility to explore such as constructing a query by dragging entities and attributes together. The COVID-19 pandemic, naturally, also limited the ability for the querying process to be user tested which would have allowed a greater focused to be played on ease-of-use. The interface design could have also benefited from user focused testing such as heuristic evaluations.

7 CONCLUSIONS

The software tool partially achieves what we aimed to. It is capable of providing users with the ability to create SQLP queries over an EER diagram. The queries supported by the tool, however, are small in comparison to the wide range of possible queries. For this reason, it would be unlikely to be integrated into the larger KnowID architecture in its current state. It meets only the minimum requirements and should be expanded upon in future works.

7.1 SQLP Queries

SQLP Queries are able to be created over a large range of knowledge graphs if it is represented in a way that allow users to interact with the model. Provided we have the attributes, entities and relationships, we can check the validity of a given query.

7.2 Attribute Paths

A traditional grammar does not hold for creating the correct parsing rules for a valid SQLP query. The dynamic nature of attribute paths means that the grammar rules need to be encoded into a program that can hopefully be expanded upon and can also serve as a standalone tool for creating SQLP queries in other environments.

References

1. Ma, W., Keet, C. M., Oldford, W., Toman, D., & Weddell, G. (2018, November). The Utility of the Abstract Relational Model and Attribute Paths in SQL. In *European Knowledge Acquisition Workshop* (pp. 195-211). Springer, Cham.
2. Ma, W. (2018). On the Utility of Adding an Abstract Domain and Attribute Paths to SQL (Master's thesis, University of Waterloo).
3. Jin, C., Bhowmick, S. S., Xiao, X., Cheng, J., & Choi, B. (2010, June). GBLENDER: towards blending visual query formulation and query processing in graph databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (pp. 111-122).
4. Calvanese, D., Keet, C. M., Nutt, W., Rodríguez-Muro, M., & Stefanoni, G. (2010, March). Web-based graphical querying of databases through an ontology: the WONDER system. In *Proceedings of the 2010 ACM symposium on applied computing* (pp. 1388-1395).
5. Čerāns, K., Šostaks, A., Bojārs, U., Ovčiņņikova, J., Lāce, L., Grasmanis, M., ... & Bārzdiņš, J. (2018, June).

ViziQuer: a web-based tool for visual diagrammatic queries over RDF data. In *European Semantic Web Conference* (pp. 158-163). Springer, Cham.

6. Ss

SUPPLEMENTARY INFORMATION

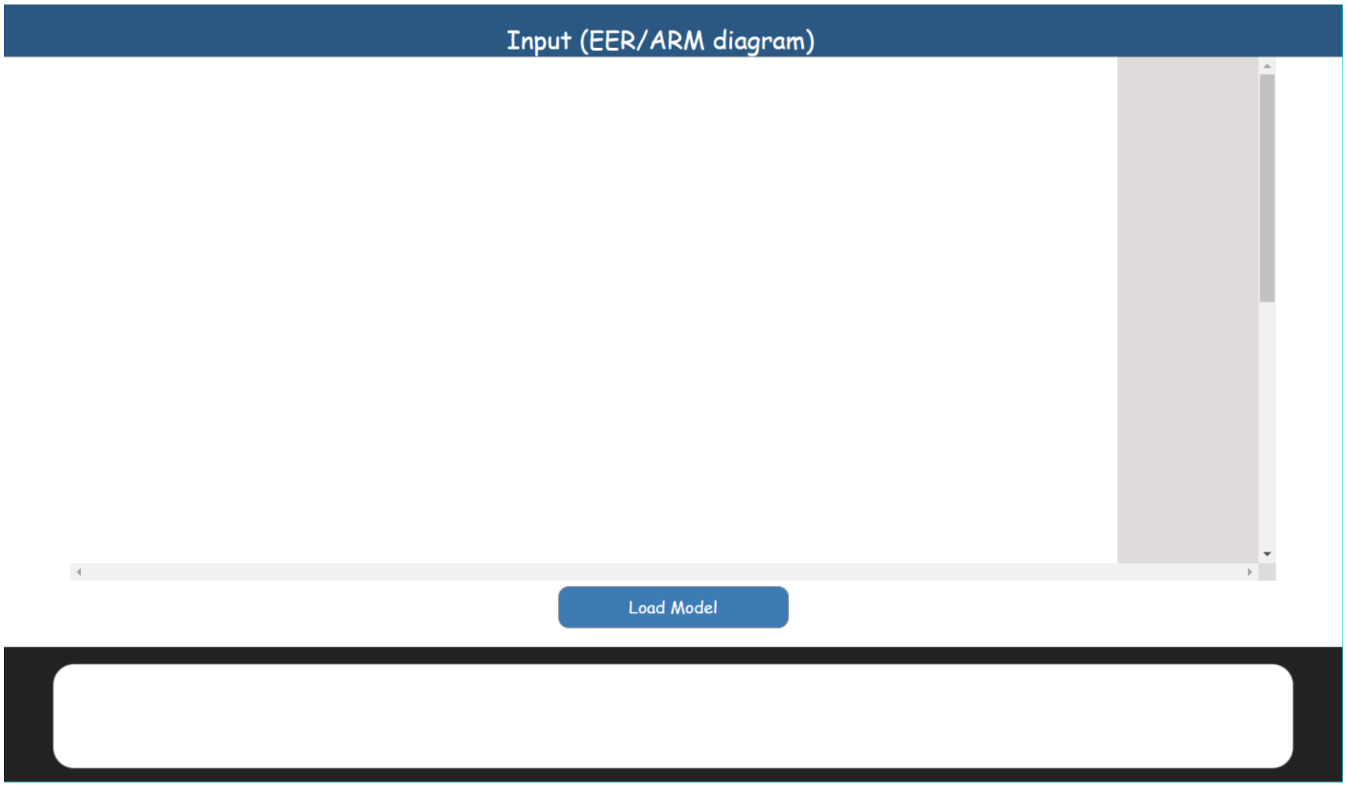
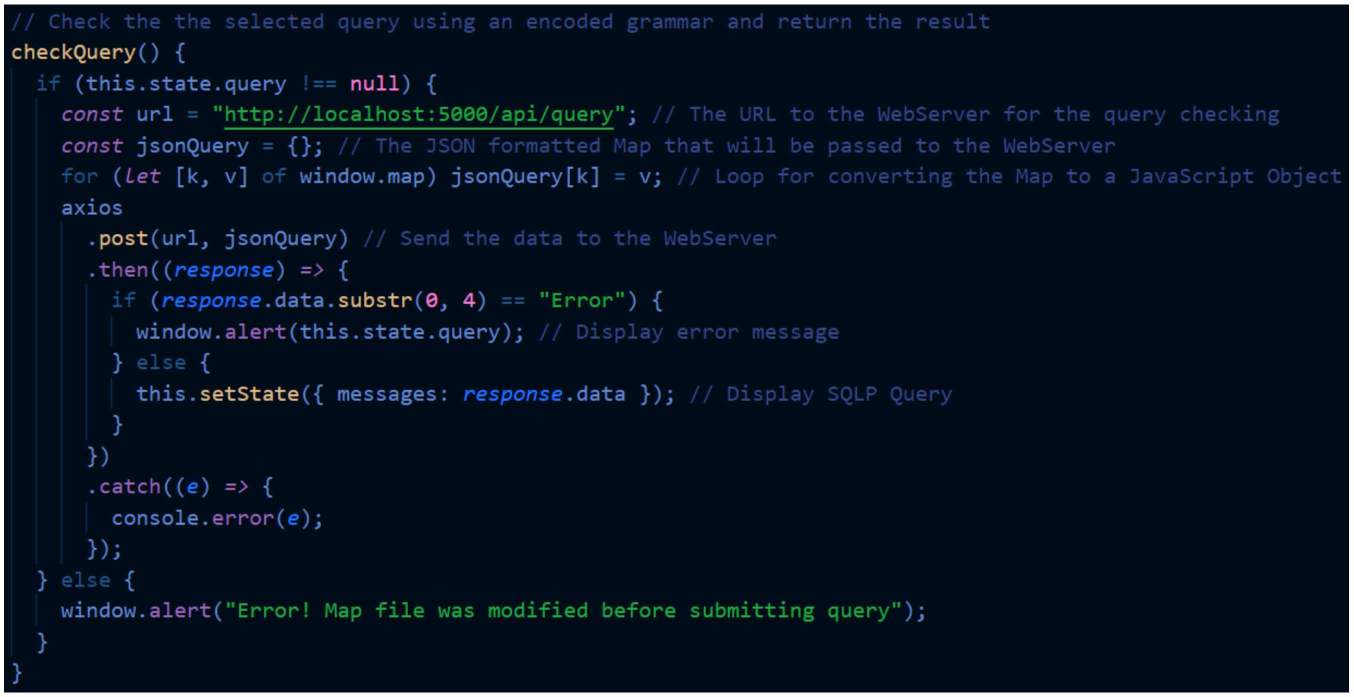


Fig. S1: User Interface for query formulation: Starting application



```
// Check the the selected query using an encoded grammar and return the result
checkQuery() {
  if (this.state.query !== null) {
    const url = "http://localhost:5000/api/query"; // The URL to the WebServer for the query checking
    const jsonQuery = {}; // The JSON formatted Map that will be passed to the WebServer
    for (let [k, v] of window.map) jsonQuery[k] = v; // Loop for converting the Map to a JavaScript Object
    axios
      .post(url, jsonQuery) // Send the data to the WebServer
      .then((response) => {
        if (response.data.substr(0, 4) == "Error") {
          window.alert(this.state.query); // Display error message
        } else {
          this.setState({ messages: response.data }); // Display SQLP Query
        }
      })
      .catch((e) => {
        console.error(e);
      });
  } else {
    window.alert("Error! Map file was modified before submitting query");
  }
}
```

Fig. S2: App.js showing the checkQuery() method