Master Thesis

Query Inference component for DataMockups

Author(s):
Asani, Zera

Publication Date:
2016

Permanent Link:
https://doi.org/10.3929/ethz-a-10807276

Rights / License:
In Copyright - Non-Commercial Use Permitted
Query Inference component for DataMockups

Master Thesis

Zera Asani
<zasani@student.ethz.ch>

Prof. Dr. Moira C. Norrie
Alfonso Murolo

Global Information Systems Group
Institute of Information Systems
Department of Computer Science
ETH Zurich

2nd November 2016
Abstract

The web engineering community has recently started to look into new hybrid approaches that try to combine aspects from model-driven web engineering and interface-driven development. There are some tools that ease the development of websites starting from interface mockups and uses them to generate a model and/or a prototype, such as DataMockups. Indeed, DataMockups allows one-way generation process; it enables digital mockups with real content to be created, and generates a database code based on the data content. However, the generated database is not used by any of the mockups, which remain static HTML pages.

This master thesis aims at extending the DataMockups tool with a Query Inference (QI) component to infer the query that generates the corresponding output from the mockup. Since DataMockups is aimed at designers that have some basic knowledge of HTML and CSS and little or nothing of databases and queries, the QI simplifies the way inferred queries are presented to the user. Nevertheless, the more expert users may view and modify the query in SQL. To make use of the inferred query and bring functionality to the mockups, the code to execute the inferred queries is inserted in an augmented version of the mockup, which becomes a working prototype that actively queries the database. The QI component was tested in various mockup designs with data from different domains. The results showed that the QI is able to infer queries of different complexity that generate the same data displayed in the mockup.
# Contents

1 Introduction ............................................... 1

2 Background .................................................. 5
   2.1 Web Engineering community work ......................... 5
      2.1.1 MockupDD ........................................... 5
      2.1.2 MockAPI ............................................. 6
      2.1.3 DataMockups ......................................... 6
   2.2 Database community work .................................. 7
      2.2.1 Query by Output ..................................... 7
      2.2.2 Reverse Engineering Complex Join Queries ............. 7
      2.2.3 Query reverse Engineering ............................ 7
      2.2.4 Discovering queries based on example tuples .......... 7
      2.2.5 Query from Examples ................................ 8
      2.2.6 Learning Queries from Examples and Their Explanations .. 8
      2.2.7 The Universal Relation as a User Interface ............ 8
   2.3 Work positioning ......................................... 9

3 Design of the Query Inference component .................. 11
   3.1 Target audience .......................................... 11
   3.2 DataMockup components used by Query Inference .......... 11
      3.2.1 Element Detection .................................... 12
      3.2.2 Schema Formation ..................................... 13
      3.2.3 Database Generation .................................. 14
   3.3 Schema conversion to Relational Database ................ 15
      3.3.1 Using the schema conversion ........................ 15
   3.4 Query Inference .......................................... 17
      3.4.1 Using the Query Inference component ................. 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>Mockup Augmentation</td>
<td>25</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Using Mockup Augmentation</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>Architecture and Implementation</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Choice of technologies</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Architecture</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>DataMockups data structures used by the Query Inference</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>Schema conversion to Relational Database</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>Query Inference</td>
<td>31</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Initialize the user interface elements</td>
<td>31</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Initialize and store the objects from the clusters present in the IM</td>
<td>32</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Infer FROM Clause</td>
<td>33</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Infer SELECT Clause</td>
<td>34</td>
</tr>
<tr>
<td>4.5.5</td>
<td>Infer WHERE Clause</td>
<td>39</td>
</tr>
<tr>
<td>4.5.6</td>
<td>Simplified SQL interface and SQL interface</td>
<td>43</td>
</tr>
<tr>
<td>4.6</td>
<td>Mockup Augmentation</td>
<td>45</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Evaluation in terms of performance and quality of generated queries</td>
<td>49</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Study description</td>
<td>49</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Results of the training set of projects</td>
<td>51</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Results of the test set of projects</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>Evaluation in terms of efficiency</td>
<td>79</td>
</tr>
<tr>
<td>5.3</td>
<td>Discussion</td>
<td>80</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>81</td>
</tr>
<tr>
<td>A</td>
<td>The InferWhere function code snippet</td>
<td>83</td>
</tr>
<tr>
<td>B</td>
<td>Example of augmented mockup</td>
<td>87</td>
</tr>
<tr>
<td>C</td>
<td>List of inferred query SQL statements</td>
<td>91</td>
</tr>
</tbody>
</table>
Introduction

In the mid 1990s, web development practitioners and researchers introduced a new discipline called Web Engineering, which promotes a systematic process for developing large scale websites [9, 10]. From then on, web engineering community has seen an expeditious evolution of website technologies, developing tools and development approaches.

Model-driven web engineering and interface-driven development have developed into the most prominent approaches for enabling rapid and collaborative web application development. Whereas the first approach starts with model creation and the interface is designed at the end of the process, the second approach process starts from interface prototypes or mockups. Nevertheless, none of them provides full support for the whole life cycle of website development. Of particular interest is the model-driven approach, which besides its gaps and shortcomings, is still advocated by most of the web engineering community. However, Aragón et al. in their analysis have shown that there is still lack of tools supporting model-driven web engineering approaches leading to a narrow adoption [1]. Moreover, a recent survey [11] has confirmed that most practitioners are unfamiliar with modelling techniques or have little background even in general computer science knowledge, meaning they probably don’t even know how to abstract models.

Therefore, the web engineering community has recently started to look into new hybrid approaches, trying to combine aspects from model-driven web engineering and interface-driven development. Mockups help the designer or developer to represent the structure of information, visualize the content and even discuss the requirements of a web application or website under development. Expressive and realistic mockups can ease the process of analysis and discussion of the requirements with stakeholders that typically involve the customers, the designers, and the developers. Furthermore, it stimulates feedback from customers and helps clarifying the requirements.

Additionally, mockups with realistic content are of great importance for the discussion process. Avoiding *Lorem ipsum* can help the designers to get as close to the customer experience
as possible [5]. However, there is a lack of tools and approaches on how to continue after the mockup is created. There are some tools that ease the development of websites by starting from interface mockups and go through a model-based representation at some stage, such as MockupDD [12], MockAPI [13] and DataMockups [4] but they have some limitations. While MockupDD approach requires prior knowledge of the modelling tools, MockupAPI approach does not need knowledge of model-driven web engineering models. However, it is limited to the generation of APIs and cannot be used for more general web development scenarios.

DataMockups, a system developed in the GlobiS 1 group currently allows one-way generation process. This means that the design is created first and then the database schema in relational form is generated, together with the database API to let the developer access the database in the code. Unfortunately, the created mockups are static, and the generated database does not bring dynamicity to the mockup. In order to describe the problem, let’s consider a simple example, which is illustrated in Figure 1.1. Assume we have a mockup displaying some data $D$. Some query $Q$ may generate the corresponding result set to display $D$. As mentioned before, DataMockups is able to generate the database schema and the API from the content that is housed in the mockup. We can see that after the generation process, the mockups do not use the generated database. Therefore, they cannot be considered as working prototypes of the website.

Figure 1.1: Query Inference for DataMockups

This thesis aims at extending the DataMockups tool with a Query Inference (QI) component to infer a query $Q'$ that the prototype would need to execute on the database such that $Q'$ also generates $D$ as a result set ready to be displayed in the prototype. It is not however necessary that $Q' == Q$ as long as it produces the same results. To do so, the mockup housing the content should be augmented with code that uses the generated API to query the database and, as a result, produces the output shown in the mockup. The augmented mockup would therefore become a working prototype of the website that actively queries the database.

1https://globis.ethz.ch/
Contributions

The QI component for DataMockups developed in this project is an extension of DataMockups tool. This component aims at finding a query $Q'$ that generates the corresponding output from the mockup, based on the data and the clusters presented in the mockup and the data structures holding the schema information inferred from it. The schema information includes the entities (with their attributes and their relationships) for a particular project. However, the users may also assist in this process, as the query inference does not need to be entirely automated.

Since DataMockups is aimed at designers that have some basic knowledge of HTML and CSS and little or nothing of databases and queries, the tool simplifies the way queries are presented to the user. Nevertheless, the more expert users may view and modify the query in SQL (Structured Query Language). Finally, the code for the inferred queries is inserted in the augmented version of the mockup, which would therefore become a working prototype that actively queries the database.

The rest of the thesis is structured as follows. Chapter 2 contains an overview of the work related to the areas of databases in the query inference topic and web engineering. Chapter 3 presents the design and functionality of the QI component. In Chapter 4 the architecture and implementation of the component are described in detail. Chapter 5 reports the evaluation results in terms of quality of the generated results and efficiency. Chapter 6 concludes the thesis and outlines the limitations of the component and possible future work.
The chapter discusses the background related to the thesis. The work this thesis aims to accomplish is to infer a query $Q'$ which is used to display some results $D$ in the mockup. Hence we need to define a hybrid issue from the web engineering and database domains. We are focused in observing different approaches that ease the development of websites and those assisting non-expert database users to infer queries. Defined in terms of databases, the results $D$ in the mockup corresponds to a result table $T$, which is the output of some query $Q$ on database $D$. This problem of inferring the query $Q'$ corresponds to an inverse problem of finding the query given its result set. Moreover, the inferred queries should be as descriptive as possible, suitable to be understood by users at any level of expertise.

The first section of this chapter explains the research related to hybrid approaches of mixed model-driven web engineering and interface-driven development, that has been done in web engineering community. The next section reviews the existing work in the database community on finding the query given its answer in different forms. The final section provides the positioning of this thesis.

2.1 Web Engineering community work

2.1.1 MockupDD

MockupDD [12] is a model-driven approach that eases the incorporation of agile software development approaches to Model-Driven Web Engineering (MDWE) methodologies. This approach uses a User Interface Mockup-Driven process to develop Web Applications where mockups can be refined until obtaining the final presentation structure of the website. The MockupDD process starts with a requirements gathering stage, producing a User Stories set to be fulfilled, which are mapped to the Structural User Interface (SUI) models. The SUI models
can be augmented with tags that are used to specify concepts and features of the website. After the tag tasking is finished, the complete tagged SUI model can be used to generate the MDWE models (currently WebML [2] and UWE [7]). Although the approach allows automatic model generation and a sandbox "demo" application, it may seem complicated for developers with no prior knowledge of the modelling tools (e.g. navigation or entity modelling) owing to the fact that whole process involves designer skills and irreplaceable human supervision for the mockup tagging.

2.1.2 MockAPI

MockAPI [13] similar to MockupDD tries to combine the advantages of agile and Model-Driven Development by eliciting requirements through user interface mockups. The process allows automatic generation of the starter/starting API implementation from annotated mockups. Annotations include functionalities related to the content, navigation and custom functions. Compared to MockupDD, MockAPI requires no knowledge of model-driven web engineering models, however it is limited to the generation of APIs and does not bring functionality to the mockup.

2.1.3 DataMockups

DataMockups (briefly introduced in Chapter 1), is a tool which aims to enable developers to create digital mockups with real content. Using this tool the users are able to create different projects, in which they can create multiple HTML mockups with real content without writing any code. The created mockups can be edited and customized with arbitrary CSS styles, and later exported as HTML and CSS files so that can be used in real websites. Once the mockup is finished and content has been placed, the database schema generation process can begin.

The first step in this process is done through the detection component, which detects groups of similar data within a page and suggests them to the users. They can name the clusters that they find relevant and discard the irrelevant ones. The detection process can be repeated for all the mockups of the same project for later inferring a complete schema. The schema component is used for the schema formation of the previously detected elements. Through analysis of the saved clusters and content, an entity-relationship (ER) schema is inferred automatically and it can be improved by adding the content from each mockup of the project. Once the ER schema is complete, users can generate an SQL schema. In addition, a PHP-based API is generated based on the detected schema, which is in turn used to populate the database.

However, DataMockups until now has only targeted the generation of the database schema in relational form starting from the content that is housed in various pages of the mockups, allowing for user customization. Moreover, the created HTML-based mockups are static and do not make use of the generated database. In other words, it does not generate a working prototype of the website but only the API necessary to let the developer access the database in the code.
2.2 Database community work

2.2.1 Query by Output

Query by Output (QBO) [15] is a data-driven approach which aims to discover (SPJ) select-project-join instance-equivalent queries. It takes as input an optional query $Q$, database $D$ and the query’s output $Q(D)$ and constructs an alternative query $Q'$ such that $Q'(D)$ is equivalent to $Q(D)$. $Q$ and $Q'$ should be instance-equivalent queries, not necessarily be equal as long as they produce the same result. To derive select-project-join (SPJ) instance-equivalent queries, QBO determines three components of $Q$: $rel(Q')$, $sel(Q')$, and $proj(Q')$. Queries are generated on a data classification-based technique called TALOS (Tree-based classifier with At Least One Semantic). The most critical part is generation of "good" $sel(Q')$ and TALOS uses decision tree classifier for generating $sel(Q')$, whereas $rel(Q')$ and $proj(Q')$ are more straightforward. This approach is limited to our work because it finds the query given the input result set $Q(D)$ in tabular form, as it would be returned from the relational database. Our thesis aims to find the query from a set of separated clusters and the content elements from the mockup as input result set. The mockup itself can bring some relevant cues for query inference, which would be lost with the QBO approach. Therefore, the query result set format used by our work makes difficult the usage of this approach that builds a decision tree to split the tuples of the result set.

2.2.2 Reverse Engineering Complex Join Queries

Zhang et al. [17] approach similarly to QBO of [15] takes as input a database $D$ and the query’s output $Q(D)$, to compute the query $Q'$. While QBO focuses on discovering select-project-join queries, this approach considers reverse engineering of project-join queries (i.e. without any selection predicates). It first generates minimal project join candidate queries and tries to find additional joins that yield exact equivalence. The limitation that this work imposes is very straightforward as we are interested in more complex queries.

2.2.3 Query reverse Engineering

Query Reverse Engineering (QRE) [16] is an extension of QBO that generalizes it in three key dimensions of the problem space: original query $Q$ is unknown, the derived query $Q'$ belongs to a more expressive query fragment beyond SPJ queries and database $D$, which has multiple versions. Additionally, TALOS has been extended on a novel dynamic data classification formulation to support the three key dimensions of the QRE problem mentioned above. Although this approach efficiently solves the QRE problem, supporting different IEQ fragments it starts from a database schema and the input result set is in tabular form, different from the one used in our work.

2.2.4 Discovering queries based on example tuples

The work by Shen et al. [14] is related to [15, 17, 16] because the intended query is automatically inferred based on the table given as input. The goal of the system is to discover
minimal valid project-join queries whose answers in the database $D$ are consistent with a user-provided table $T$ (table with rows and columns). In order to discover all the minimal valid project join queries with respect to a given example table, the system first generates a set of candidate queries and classifies them with a filter-based approach in order to efficiently find the correct set of valid minimal queries. This work targets big datasets such as IMDB \(^1\) with size of 10 GB, while this thesis will focus on smaller datasets coming from the data in the mockups.

2.2.5 Query from Examples

Query from Examples (QFE) [8] approach is developed with the aim to assist non-specialist database users in formulating queries. It only requires from the users to identify whether a given output table is the result of his/her intended query on a given database. To start the construction the user provides the pair of inputs, a database $D$ and the query’s output $Q(D)$ as in [17, 16]. QFE exploits two challenges on how to generate candidate queries and how minimize the user’s effort to identify the desired query with optimizing user-feedback interactions. In order to generate the set of queries it employs the QBO approach of [15], because it supports more expressive queries than [17]. To minimize user’s effort and identify user’s target query the system iteratively presents a slightly modified database-result pairs making it possible to eliminate candidate queries from user’s feedback and identify the target query.

2.2.6 Learning Queries from Examples and Their Explanations

Learning Queries from Examples and Their Explanations [3] is a novel approach developed with a similar aim as QFO in [8], to help non-experts in constructing database queries. It is built on the foundations of other frameworks for automatically inferring queries such as [14, 15, 17] where the set of input and output examples are extended with additional information in the form of explanations. It is able to infer complex conjunctive queries as it does not impose restrictions on the domain of queries as [15, 17].

A conjunctive query is a restricted form of first-order queries which are equivalent in expressive power to SPJ queries in the relational algebra. The solution is implemented in a prototype system responsible for inferring the queries which consists of a user-friendly GUI and allows users to input examples and explanations. Explanations are formed by simply drag-and-dropping database tuples. Authors have demonstrated that the provision of explanations for database non-experts is feasible and the system is able to propose queries with high quality. However, the usage of such explanations in our system is not possible because it should infer the query based on the data encoded in the mockup.

2.2.7 The Universal Relation as a User Interface

In [6] Ullman introduces a new concept called ”window functions” which allows to differentiate the various approaches to interpreting queries over universal relations. A universal

\(^1\)http://www.imdb.com/
relation represents the entire database as one relation. In this work Ullman provides an approach on how System/U, a universal relation experimental database system developed by Ullman et al. at Stanford, or any system that has support for universal relations, decides on the responses to queries.

The System/U is intended to test the use of universal relation. Its query language employs a syntax similar to a regular query language but with more simplified syntax and enriched semantics, because it refers to the universal relation rather than to the actual database schema. To define a query in System/U query language, the users just list the attributes of the universal relation. There is no need for declarations of tuple variables because all database tables are represented as one relation. The work also discusses a query-interpretation algorithm that uses the notion of universal relation. There is no need for declarations of tuple variables because all database tables are represented as one relation. The work also discusses a query-interpretation algorithm that uses the notion of universal relation. This work is similar to ours in the way it approaches the SELECT part. Firstly, it defines the window for X, denoted by \( [X] \), which is the natural join of the minimal set of relations whose schema includes all of the attributes in set X. Therefore, the query mentions only the attributes that are contained in set X. Moreover, the subject of this work may serve as a good indication for providing a more simplified user interface for the inferred queries.

2.3 Work positioning

Our work is similar to \([16, 14]\) in terms of the input it requires to infer the query \( Q' \). The results \( D \) displayed in the mockup using a query \( Q' \) correspond to a result table \( T \) and the complete schema encoded in the mockups of a particular project corresponds to database \( D \). However, we have to do with a schema that is encoded in the mockup which is not a database schema and the datasets are different with regard to their size. They possibly start from a big dataset, while this thesis aims at finding this query based on a smaller dataset.

Regarding the target audience, it fits to \([8, 3]\) but the approach they use cannot be applied to our scenario. The goal of this thesis is to find a convenient approach to infer the query \( Q' \) and insert it in the augmented version of the mockup, which would therefore then become a working prototype that actively queries the database. Our work is the first, to the best of our knowledge, that finds a query \( Q' \) (generating query) from the schema information encoded in the mockup and creates a working prototype of the website.
Design of the Query Inference component

The main goal of this thesis is to find a generating query that produces the corresponding output from the mockups based on the data and the clusters present in the mockup and the data structures holding the schema information inferred from it. Moreover it should provide support for mockup augmentation. This chapter describes in detail the design of the suggested approach. After presenting an overview of the functions provided by DataMockups tool, which are used in our work, we will focus into the goals and functions of the QI component and the module of mockup augmentation.

3.1 Target audience

DataMockups employs a hybrid approach of model-driven web engineering and interface-driven development, adapted for designers and developers that have some basic knowledge of HTML and CSS, and possibly no knowledge at all about database design. Indeed, the QI component primarily should target non-database specialists and fit to the DataMockups audience. Although the queries the system supports are automatically inferred and presented in a simplified user interface, it is assumed that users have some knowledge about SQL queries in case they need to change or improve the inferred query.

3.2 DataMockup components used by Query Inference

QI component is an extension of DataMockups tool and necessarily needs to use its functions in order to infer the query for a specific mockup. As introduced in Section 2, the design editor
of the tool enables designers and developers to create mockups, edit and style directly within the tool which is illustrated in Figure 3.1. The panel on top of the tool offers the option to create new mockups and consists of a page switcher for viewing the created mockups. On the left side of this area, users can save or discard the current changes in the mockup, whereas the button on the right side allows users to download the HTML and CSS files of the current mockup. Created mockups can be enriched with elements from the palette of elements located beneath the above-mentioned panel or any other HTML element as it is implemented as a web application.

General and Editing & Styling tabs allow the user to customize the mockup page, define the viewport, style elements with arbitrary CSS properties, etc. The creation of the mockups is not a very important issue regarding the work this thesis aims to accomplish and one can read more in [4]. Concerning the work this thesis targets, we are interested in the functions provided by the following components: element detection, schema formation and database generation, which will be described in more detail in the next three subsections.

![Figure 3.1: DataMockups design tool](image)

### 3.2.1 Element Detection

Element Detection component is responsible for automatically detecting groups of similar elements on the chosen mockup after it has been designed and content has been placed on it. The first step in this process is done through the Detection tab. Using the Choose area button, the user should select the area where the content of their designed web page is located. The selected area is marked with a purple overlay and after the Done choosing area button is clicked the detection will be completed. The tool detects groups of similar data within a page based on the visual and structural similarity of elements and suggests them to the user. The users can name the clusters that they find relevant, discard the irrelevant ones and click the Finished naming/discarding clusters button to finish the detection process. This process should be repeated and saved for all mockups of the same project in order to get a complete schema of the project. Figure 3.2 shows a detected detail page of a particular movie, where genre, actors and movies themselves form different clusters.
3.2.2 Schema Formation

Schema & Data tab is used for the schema formation of the previously detected elements. Through analysis of the saved clusters and content, an entity-relationship (ER) schema is inferred automatically. The ER schema consists of entities with their attributes and the relationships between them which are detected automatically. The users can modify the cardinality and name of the relationship if they are not satisfied with proposed ones, and refine properties such entity and attribute names and attribute types using the blue button on the right-hand side of schema elements.

By clicking the Add button for each of the mockups selected from the page switcher on the top of the tool, the user can add the previously detected clusters in the mockups and get a complete ER schema for the desired project. Additionally, the data from the pages is displayed immediately on the Detected Data section. Indeed, the data displayed corresponds to the data that will be inserted in the database. Figure 3.3 shows the Schema & Data tab, which contains the complete schema and detected data.
3.2. DATABASE GENERATION

Database Generation service is able to create the database and fill the created tables with the content displayed in the mockups. This step needs to be executed after the user has generated the complete schema over all mockups. In addition to generated database, the users receive and object-oriented PHP server-side API that is able to manage the entities and relationships in the schema. Once the ER schema is complete, the users can start the generation of the database by clicking the Create Database button in the Schema & Data pane. The users have two options, to create the database and fill the data directly or export them as scripts. However, they need to fill their database information such as: database address, username and password as shown in Figure 3.4.
3.3 Schema conversion to Relational Database

The first step in the query inference process is the conversion of the data structures holding the complete schema information into a relational database in order to have a relational form of the database inside the tool. The reasons behind the schema conversion are the following:

- it allows the users to see a representation of the created database inside the tool
- creates the structure of the data from the data structures holding the schema information, especially the relationships

3.3.1 Using the schema conversion

To start the process of schema conversion we should first perform the steps described in Section 3.2, as they provide the needed functions for inferring the query. In Figure 3.3 we show the Schema & Data tab which contains the complete schema and detected data. Once the schema is generated, before proceeding to the schema conversion, users should first create the database by clicking in the Create database button. The purpose of this step is to ensure that the database is created before queries are inferred because we need to check their correctness referring the database data.

Finally, the Query Inference button should be clicked, which directs the users to a new interface with a slightly different form than that in Figure 3.1 hiding unused functionality by our component as shown in Figure 3.5. Unused functions include: Save and Discard buttons, the option to create new mockups and the button which allows users to download the HTML and CSS files of the current mockup. The reason behind this change is to have a separate user interface from the one of the other components.
Nevertheless, the users have the option to return in the original DataMokups interface using the Change Schema button. Hence they can modify the schema or start a new schema detection from the beginning. In the Relational Database Schema pane illustrated in Figure 3.6 the users have the possibility to view the results of schema converted in relational database. The pane shows the schema of the Movies database and lists all the entities relationships. Each entity of the ER schema is converted to a table and their attributes to columns as shown by the symbols near each element. Additionally, each table contains an ID as primary key.

To support the N-N relationships detected by the tool, a third table is created. For instance, the N-N relationship between movies and genre is expressed in the moviesgenre table, which primary key consists of moviesID and genreID used to reference participating tables. Hence, they are also the foreign keys in the moviesgenre table. In case of 1-N or N-1 relationships the primary key of the table with cardinality 1 is placed within the table N as a foreign key, whereas for the 1-1 relationships the placement of the primary key can be done on both relations, used as foreign key to reference the other table.

Besides, displaying the schema as a relational database and give a clear idea of the created database, it creates the structure of the detected relationships based on their type. Each of the schema relationships is converted to an object that consists of the values of the participating entities, relationship name and its type. Furthermore, the converted schema is used to get the tables that are created from the N to N relationship types. More details about this function will be given in Section 4.4.
3.4 Query Inference

As described in Section 2.3 and as its name suggest this component should infer the query that generates the corresponding output from the mockups. The goals for the query inference are as follows:

- automatically infer the query for the chosen mockup
- simplify the way queries are presented to users
- check query accuracy with the corresponding mockup
- allow expert users to view and edit the query in SQL language
3.4.1 Using the Query Inference component

To begin the query inference process, the users should first choose the desired mockup based on the assumption that the mockup content is generated by only one query. The desired mockup can be selected from the Select mockup select list on top of the tool and click the Infer Query button to automatically infer the generating query. As mentioned in Section 3.1, derived queries should be presented in a simplified user interface and in SQL language described in next two sub sections.

Simplified presentation of queries

In a first step, the inferred query is displayed on a simplified user interface designed for use by people who have little knowledge on databases. The derived query is displayed on the Inferred Query pane. Examples of automatically inferred queries are shown in Figure 3.7 and 3.8. As we can see the queries are presented to users in a simplified user interface which depends on the type of query.

The inferred query shown in Figure 3.7 is of the type select-project-join (where-select-from clauses in SQL). It is used to generate the data that are displayed on a mockup that shows details of the movie with title “Avengers: Age of Ultron”. Figure 3.8 shows the inferred query of the type project-join (select-from clauses in SQL). The inferred query is used to generate the data that are displayed on a mockup showing a list of movies along with their details.

The SHOW keyword is the translation of the SQL SELECT clause, and LIST the keyword for GROUP_CONCAT() function. In order to make queries as understandable as possible, we omit the FROM clause and join predicates because it is not relevant to target users. The most important part of the query is the select predicate (i.e. condition in the WHERE clause). The keyword OF is used instead of FROM and WHERE and the comparison predicate (movies.title = "Avengers of Ultron") is expressed as English sentence. Obviously, the sentence depends on the type of predicate (i.e. comparison, BETWEEN or LIKE) supported by the system. However, mockups can display data that are not filtered based on some predicate. In this case, we have a project-join query as shown in Figure 3.8, the lack of predicate is indicated by the all keyword.
Figure 3.7: Inferred query for a mockup that displays the details of the movie with title "Avengers: Age of Ultron"
In order to give feedback on the discovered query, *Inferred Query* pane contains a progress bar which displays the query accuracy and errors with a percentage. For example, a 100% would mean that query results correspond to the result set displayed in the mockup, whereas an 80% would mean that the query is partially accurate and a warning message displays the cause of incompleteness or error. Additionally, the users can view the results of the inferred query in table format by clicking the *View Results* button. A sample result set corresponding to inferred query in Figure 3.7 is illustrated in Figure 3.9.
In case that the query is accurate on more than one condition predicates and displayed result set is sorted by one or more columns, they appear as suggestions. The badge on the right-hand side of WHERE and ORDER BY displays the number of suggestions. If the amounts are greater than zero, the users can follow them in the SQL Query interface, retrieved by clicking View/Edit SQL Query button, that is explained in the next sub-section. The Save button located on the bottom of the pane is used for mockup augmentation described in Section 3.5.

**SQL Query**

Since the users may also assist in the process of query inference, as it does not need to be entirely automated, it should be possible to view the query in SQL language, which provides more fine grained control on the inferred queries. Users can view and update the inferred SQL query. However, the modification of SQL query clauses requires a fair degree of expertise in using the SQL query language. The current version of DB-API-Generator used in DataMockups only supports MySQL databases, therefore the inferred queries should be expressed in SQL. However, the format in which queries are specified is supported by all the relational DBMSs. The inferred SQL query consists of the basic clauses listed below:

- **SELECT** - indicates which table columns are displayed on the result set
- **FROM** - indicates the tables that contain the data displayed on the result set
- **WHERE** - consists of join and condition predicates
  - **JOIN** - specifies the inner join predicates for tables participating in a relationship using implicit join notation (i.e. joins are expressed in the WHERE clause)
- **CONDITIONS** - specified using comparison operators like `=`, `≥`, `≤` and logical operators BETWEEN or LIKE, if result set is displayed based on some condition predicate

Other possible clauses that we consider are:

- **GROUP BY** - used to group result set, used in conjunction with `GROUP_CONCAT()` function
- **ORDER BY** - specifies the order if the result set is sorted by one or more columns in ascending or descending order
- **DISTINCT** - used to remove duplicates and return distinct values
- **GROUP_CONCAT()** - concatenates values from a group and returns a concatenated string
- **HAVING** - specify a search condition for `GROUP_CONCAT()` function used in the **SELECT** clause

Indeed, the **HAVING** clause is not supported by the QI and cannot be inferred automatically, but it can be added in the **Additional Functions** by expert users as it will be described below. In order to see the query in SQL language the user should click the **View/Edit SQL Query** button. An example of inferred SQL query is illustrated in Figure 3.10. This query corresponds to the query displayed in a simple user interface in Figure 3.8. We can see that each of the above-mentioned clauses are separated in different textareas allowing the users to make changes in case they are unsatisfied with the query. Apart from that, a complete code of the SQL query statement is visible in the gray block code.

**Additional Functions** text area provides a very significant feature to the system. In this area, the expert users can add additional SQL functions in case they want to improve the inferred query. Another important function supported by this work and mentioned in the previous subsection is the consideration of suggestions. The inferred query displayed in Figure 3.11 contains one suggestion for the **WHERE** clause, i.e. a LIKE condition predicate which gives the same accuracy as the comparison predicate added directly on the **CONDITIONS** text area. Using the toggle switch, the users can add this suggestion and changes will be reflected in the **CONDITIONS** text area, gray block code and in the simplified user interface. Since both conditions give the same accuracy, they can be joined manually by the users with an **AND** operator.

**ORDER BY** suggestions can be added in the same manner using the toggle switch button and changes will be reflected in the gray block that contains the complete SQL query statement (see Figure 3.11) and in the simplified user interface. To simplify the representation of the **ORDER BY** suggestions, next to each suggestion the users can see an icon which is customized based on the order and type of data. These suggestions should be taken in consideration in cases where the result set displayed in the mockup should be identically ordered in the augmented mockup. After the modifications, the users can see the results displayed in a table format in a modal window and get feedback on the query accuracy using the **View Results** button.
SELECT movies.img_alt, movies.img_src, movies.title, movies.rating, movies.releasedate, movies.overview, GROUP_CONCAT(DISTINCT genre.genre_att SEPARATOR ' | ') AS genre_att
FROM movies, genre, moviesgenre
WHERE moviesgenre.moviesID=movies.ID AND moviesgenre.genreID=genre.ID
GROUP BY movies.title

SELECT movies.img_alt, movies.img_src, movies.title, movies.rating, movies.releasedate, movies.overview, GROUP_CONCAT(DISTINCT genre.genre_att SEPARATOR ' | ') AS genre_att
FROM movies, genre, moviesgenre
WHERE moviesgenre.moviesID=movies.ID AND moviesgenre.genreID=genre.ID
GROUP BY movies.title

WHERE

ORDER BY

No suggestions

Figure 3.10: Inferred SQL query for a mockup which displays a list of movies
Figure 3.11: Inferred SQL query for a mockup which displays a list of products by a particular manufacturer
3.5 Mockup Augmentation

In order to make use of the inferred query and bring functionality to the mockup, the mockups hosting the content should be augmented with code, that uses the generated API to query the database and, as a result, produces the output shown in the mockups. Specifically, they should house and run the query once they get generated. The main goals of mockup augmentation module are:

- make use of the generated database and inferred query (i.e. query the database)
- generate a working prototype of the website with a dynamic output

3.5.1 Using Mockup Augmentation

Once the user is satisfied with the inferred query, the next step is to augment the mockup with the needed PHP statements and the query, so that the latter can be run and show the results in the augmented mockup dynamically. This can be achieved using the Save button. The augmented mockup will contain the statements that do the following:

- make the connection and select the database
- perform the inferred query on the generated database
- replicating the presentation created in the mockup design
- close the connection

Although the augmentation process should not be visible to users, it should perform the next steps. First of all, it creates a database configuration file `dbconfig.php` that contains database settings, values given by the users in the database generation process. This file is generated once the first augmented mockup is created and used by all mockups in the project. Secondly, mockups are augmented with code that includes the database configuration file, database connection command, and the necessary commands to run the query and display the results. Nevertheless, the users get a message for the outcome of the save process. If the action is successful, the users get the message displayed in Figure 3.12 informing that the augmented mockup (i.e. the PHP file) was created successfully. They can click on the file name and navigate to the generated augmentation and see the mockup dynamically retrieving the results from the generated database, using the inferred query. In addition, the users can browse the file to view the structure of the augmented mockup in the location path displayed on the message.

![Figure 3.12: Mockup Augmentation outcome](image-url)
4

Architecture and Implementation

In this chapter, we will describe the details of QI component architecture and implementation. After giving an overview of the used technologies, we describe the overall architecture in Section 4.2. Section 4.3 describes the DataMockups objects used by the QI component. In the next section, we look in the details of the schema conversion to LovefieldDB\(^1\). Section 4.5 describes the algorithms for the query inference process, specifically the inference of SELECT, FROM and WHERE clauses. The last section presents the implementation details of the mockup augmentation module.

4.1 Choice of technologies

Since the QI component is an extension of DataMockups, it depends heavily on the technologies used by it. It runs on Apache Web Server and uses several web technologies and libraries. The back-end is implemented using the CodeIgniter Web Framework\(^2\) with the PHP server scripting language. Our work functions are implemented in PHP and are integrated in the original tool by following the same style in terms of the code structure and conventions. Client-side functionalities are implemented entirely in HTML5, CSS and JavaScript similar to DataMockups. The most relevant libraries and frameworks are:

- Bootstrap\(^3\) framework is used for the overall design
- jQuery\(^4\) JavaScript library for DOM manipulations
- Lovefield JavaScript library for converting the schema to a relational database

\(^1\)https://google.github.io/lovefield/
\(^2\)https://www.codeigniter.com/
\(^3\)http://getbootstrap.com/
\(^4\)https://jquery.com/
Inferred queries are expressed in SQL and they operate on the MySQL database management system to query the generated database for each of the created projects.

4.2 Architecture

In this section, we present the architecture of the system with an emphasis on the QI component. Figure 4.1 gives a schematic representation of the architecture which is a classical client-server. Similarly to the used technology, it depends on the architecture of DataMockups. Components surrounded by the blue rectangle belong to the DataMockups architecture. However, they are enriched with functions performed by the QI component. Apart from the functions provided by DataMockups described in [4], the server accepts AJAX requests from the client in order to check the accuracy and view the results of the inferred query. The server queries the MySQL database of the corresponding project (in the bottom part of Figure 4.1) and returns to the client a JSON formatted data. Additionally, it is responsible for creating the augmented mockups. Client work includes the following: convert the schema to LovefieldDB (a relational database for web applications) and display it, infer the query, simplify the way queries are presented to users, send AJAX request to the server to query the database and manipulate DOM elements for mockup augmentation.

Figure 4.1: System Architecture

---

5 http://fontawesome.io/
6 https://api.jquery.com/category/ajax/
4.3 DataMockups data structures used by the Query Inference

After the element detection and schema formation process, DataMockups stores the schema information in JavaScript data structures of data type objects. Our work uses the JavaScript objects listed below:

1. **DataMockups.namesDictionary** is a look-up object where the unique identifiers are mapped to the cluster names. Each schema element such as entity, attribute and relationship gets a unique auto-generated identifier.

2. **DataMockups.schema** object stores the schema information where the keys are the entity identifiers. It comprises of three values: attributes, generated attributes and relations. We only use the generated attributes values which consist of the final attributes identifiers of the entity along with their type.

3. **DataMockups.relationships** object stores the relationship information where the keys are relationship identifiers. The value consists of two objects which represent the two entities involved in the relationship.

4. **DataMockups.dataContent** object stores all of the content of the relevant elements from all the mockups, added incrementally (one by one).

4.4 Schema conversion to Relational Database

After the schema formation is done, we can use the JavaScript objects mentioned in the previous section to convert the schema to relational database using the lovefield\(^7\) JavaScript library. Lovefield is a JavaScript SQL-like database library for web developers who want the benefits of a relational database inside their web applications. By having a relational database inside our web application, we do not have to query the MySQL database to get the required data for the query inference process. However, LovefieldDB does not accept raw SQL statements; instead, it has a builder-pattern API to build queries. Converting the raw SQL statements to LovefieldDB queries was difficult and out of the scope of this thesis. Therefore, we decided to use the converted schema only for displaying the database and getting the relationships along with their types in the QI component as shown in Figure 3.6.

The schema conversion process is shown in Listing 4.1. As mentioned in Section 3.3, it starts by clicking the Query Inference button, which also directs user to a new interface. The updateQueryInit function uses the jQuery hide and show methods to hide and show the corresponding elements. The JSDatabase object is a wrapper of the lovefield DB that acts as an interface with DataMockups. It is used to create a new instance of the database (jsdb) or connect to the corresponding instance if it already exists, which in turn is created using the data provided by objects 1 to 3 in Section 4.3. The database_name parameter is the database name provided by the user in the database generation process as explained on Section 3.2.3.

\(^7\)https://github.com/google/lovefield/blob/master/dist/lovefield.js
function updateQueryInference()
{
  $('#pane-generation #infer-query').click(function(ev) {
    ev.preventDefault();
    updateQueryInit();
    var database_name = $('#db_name').val();
    var isempty = jQuery.isEmptyObject(jsdb);
    if (isempty == true)
    {
      jsdb = new JSDatabase(database_name);
      jsdb.migrateSchema();
    }
    else
    {
      jsdb.removeDatabase();
      jsdb = new JSDatabase(database_name);
      jsdb.migrateSchema();
    }
    DataMockups.jsdb = jsdb;
    $('#schema_name').text(database_name);
    tables = jsdb.tables;
    displayTables();
    displayRelationships();
  });
}

Listing 4.1: Schema conversion to LovefieldDB

The functions displayTables and displayRelationships are defined with the purpose of displaying the created database. The first function displays the entities (with their attributes) from the tables object of the jsdb database instance. The displayRelationships function besides displaying a detailed view of relationships, it creates a JavaScript object that stores the junction tables created from the N to N relationships, plus their columns. This data structure is created using the functions addTable and addColumn displayed in Listing 4.2 and is used for inferring the FROM clause described in the next section. Each junction table contains the foreign keys referred to primary keys of both tables as its columns.

exports.addTable = function(tname, table)
{
  if (table[tname] === undefined)
  {
    table[tname] = { columns: [] ,};
  }
}

exports.addColumn = function(tname, key, table)
{
  table[tname].columns.push(key);
}

Listing 4.2: Functions that create the objects used for the query inference process
4.5 Query Inference

Once the processes of schema formation and schema conversion to LovefieldDB are finished, the stored data structures mentioned in the previous two sections can be used for query inference process. Before inferring the three clauses we need to do some initializations of the user interface elements and initialize and store some objects with the data from the clusters present in the investigated mockup (IM). IM is the mockup for which we want to infer the query. Therefore, the whole process consists of the following steps.

- Initialize the user interface elements
- Initialize and store objects from clusters present in the IM
- Infer FROM clause
- Infer SELECT clause
- Infer WHERE clause
- Display the inferred query in a simplified SQL interface and SQL interface

As we can see on the code snippet displayed in Listing 4.3, the query inference process starts upon a click on the Infer Query button. The steps mentioned above and the corresponding functions will be described in more detail in the next subsections.

```javascript
function inferQuery()
{
  $('#find-query').click(function(ev)
  {
    ev.preventDefault();
    inferQueryInit();
    DataMockups.dataContentPage={};
    DataMockups.generation.doneClusteringPage();
    var page_tables=getPageTables();
    var relationships=getRelationships(jsdb);
    var from_clause=DataMockups.queryfrom.inferFrom(junction_tables,page_tables);
    var select_clause=DataMockups.queryselect.inferSelect(from_clause);
    var where_clause=DataMockups.querywhere.inferWhere(from_clause,relationships);
    setQuery(from_clause,select_clause,where_clause);
  });
}
```

Listing 4.3: The inferQuery function

4.5.1 Initialize the user interface elements

As mentioned in Section 3.4.1, after the query is inferred the users can see the results in two different interfaces which contain different HTML elements. The elements appearance in the user interface and their values depend on the inferred query type. Therefore, before
adding the corresponding values for each of the new inferred query we need to hide elements that show query statements that are not contained by every query. Such elements display the values of GROUP BY or ORDER BY statements and are hidden using the jQuery hide function. Additionally, we need to set the values of elements such as textarea and span to an empty using the jQuery text and val methods. This is achieved by the inferQueryInit function shown in Listing 4.3.

### 4.5.2 Initialize and store the objects from the clusters present in the IM

The next step before inferring the three clauses is to initialize and store some data structures from the data present in the IM. As shown in Listing 4.3, we first define and initialize the `DataMockups.dataContentPage` object. The object structure is similar to `DataMockups.dataContent` provided by DataMockups at the end of schema formation process. However, the `doneClusteringPage` function only stores the content for the clusters present in the IM. The object is created by adding a new JavaScript object for each of schema entities where the key is the entity identifier and the value consists of an array of attribute objects. The attribute object key represents the attribute identifier and the value is the attribute content. This object is created for each of the entity attributes and is added to the previously mentioned array. If the data from a particular entity are not displayed in the IM, the corresponding array is empty. The structure of `DataMockups.dataContentPage` object is shown in Listing 4.4.

```javascript
DataMockups.dataContentPage = {
  <entity>:
    [0:{<attribute>:attribute_content},
     1:{<attribute>:attribute_content},
     ]
}
```

Listing 4.4: The structure of the dataContentPage object

Next we get the database relationships using the `getRelationships` function which has as input parameter the `jsdb` database instance described in Section 4.4. The `jsdb` contains three objects for each particular relationship type. The code snippet for this function in shown in Listing 4.7. As we can see the function returns a concatenated array of all relationships objects, where each object stores the values for the relationship entities, name and type.
Lastly, we create the page_tables object that stores the names of the tables and the names of their corresponding columns from the clusters present in the IM, using the getPageTables function. This function goes through the DataMockups.dataContentPage elements to get the names of the tables and their corresponding columns for the clusters displayed in the IM. After that it creates the page_tables object using the functions displayed in Listing 4.2.

### 4.5.3 Infer FROM Clause

The first clause that we need to obtain is the FROM, due to the dependence of other clauses on it. Indeed, if FROM clause lists all the tables that contain the data displayed on a single mockup, then SELECT clause can be obtained from these tables. Inferring the FROM clause is the most straightforward part of the query inference process. The first step in the inference process is to include the tables that are displayed in the IM on the FROM clause. These tables are found on a separate process by looking at the corresponding clusters of the IM and passed to the infer FROM clause process. In addition, the IM may have data from the tables that participate in an N to N relationship. However, their corresponding junction table is created after the schema is converted to LovefieldDB and cannot be obtained through the clusters of the mockup. Therefore, we need to check if a junction table is created from the tables present in the IM and display it in the FROM clause.

The code snippet for the inferFrom function is shown in Listing 4.6. As we can see, it has two input parameters which were defined in the previous subsections, respectively page_tables and junction_tables. The latter is created using the displayRelationships function described in Section 4.4.

First of all, we set the value of from_clause_tables to the value of tables displayed in the IM. Then we check if the object junction_tables is not empty and go through the junction tables to get the names of the tables the junction table refers to. As described in Section 3.3.1, the foreign key names of junction tables consist of table name + ID. Therefore, we can get the table name by removing the ID keyword using the JavaScript String split method. Next, we check if both tables the junction table refers to, are included in the page_tables. If so, page_tables is extended with the junction tables. The list of tables to be shown on the FROM clause is the array of page_tables keys.

```javascript
exports.inferFrom = function (junction_tables, page_tables) {
  //check if page tables participate in an N to N relationship, if so add the
  //junction table to the list of tables in the FROM clause
  var query = DataMockups.queryinference;
  var from_clause_tables = page_tables;
```
```javascript
var page_table_names = Object.keys(page_tables);
if(!(jQuery.isEmptyObject(junction_tables)))
{
    for(var tables in junction_tables)
    {
        var table1 = junction_tables[tables].columns[0];
        var table2 = junction_tables[tables].columns[1];
        table1 = table1.split("ID");
        table2 = table2.split("ID");
        if((jQuery.inArray(table1[0],page_table_names))!=-1 &&
            (jQuery.inArray(table2[0],page_table_names))!=-1)
        {
            query.addTable(tables,from_clause_tables);
            query.addColumn(tables,table1+"ID",from_clause_tables);
            query.addColumn(tables,table2+"ID",from_clause_tables);
        }
    }
}
return from_clause_tables;
```

**Listing 4.6: The inferFrom function**

### 4.5.4 Infer SELECT Clause

As mentioned in the previous section SELECT clause heavily depends on the columns of the tables obtained in the FROM clause, excluding the columns of junction tables. However, not all columns are added to the SELECT clause and their addition depends on the mockup content.

First of all, we need to find out if the IM is a detailed or list page. As mentioned in Section 3.2.2, schema formation process is based on a clustering algorithm which classifies elements based on a parent, child and sibling relationship. The main (parent) cluster element (element with `data-cluster` attribute) is the container element for all children cluster elements. In order to determine the type of mockup, we first need to find the number of occurrences (`numOcc`) of the main cluster in the IM. Figure 4.2 shows two example mockups from the movies project. The main cluster element in these mockups is the element with `data-cluster="movies"` attribute. As we can see, the Figure 4.2 (a) displays the details about only one movie therefore, we have a detailed page and the `numOcc = 1`. On the other hand, the Figure 4.2 (b) displays details about multiple movies hence, we have a list page and the `numOcc > 1`.

Secondly, we check the size of the array for each column `C` (`size(C)`) displayed in the IM from the `DataMockups.dataContentPage` object. If all the columns have the same size of values, the columns are added to the SELECT clause without the `GROUP_CONCAT()` function. If the column sizes are different, we need to compare the values of `numOcc` and `size(C)` and check whether there are other columns `T` such that `size(T) < numOcc`. We go through the elements of the `DataMockups.dataContentPage` object and find out if there are columns `T` such that `size(T) < numOcc` and return a boolean value. To distinguish between
two cases on whether $(\text{size}(T) < \text{numOcc}) == \text{false}$ or $(\text{size}(T) < \text{numOcc}) == \text{true}$, let us describe two examples.

(a) Detailed page

(b) List page

Figure 4.2: Mockups displaying data from movies project
To illustrate the first case \((\text{size}(T) < \text{numOcc}) = \text{false}\), let’s consider the example mockups shown in Figure 4.2. Both show details about movies, (a) corresponds to a detail page and (b) to a list page. As we can see, the movie may belong to more than one genre. The main cluster movies is in relationship with genres and both have cardinality N. However, the mockup displays a movie tuple which has multiple values for the genre column and exactly one value for the other columns. Therefore, there are no columns T such that \(\text{size}(T) < \text{numOcc}\). For the detailed page, the \(\text{numOcc} = 1 \&\& \text{size}(C) \geq \text{numOcc}\) and for the list page \(\text{numOcc} > 1 \&\& \text{size}(C) \geq \text{numOcc}\). The ideal inferred SELECT statement for columns that have multiple values or grouped result set i.e. \(\text{size}(C) > \text{numOcc}\) should contain a \text{GROUP\_CONCAT()} function that uses the \text{DISTINCT} keyword to eliminate duplicate values.

Since the SELECT clause uses the \text{GROUP\_CONCAT()} function, the inferred query needs an added \text{GROUP BY} statement to group the result set. Obviously, we only need to group results using the unique attribute that identifies the tuples in case we have a list page. The unique attribute is the column which uniquely identifies a particular row. However, in the SELECT clause inference we just need to determine whether the query needs a \text{GROUP BY} statement. A detailed description about the unique attribute will be provided in the next section.

![Mockup displaying movie titles from a particular actor](image)

Figure 4.3: Mockup displaying movie titles from a particular actor

An example for the second case \((\text{size}(T) < \text{numOcc}) = \text{true}\) is a mockup which displays a list of movie titles from a particular actor illustrated in Figure 4.3. Similarly to the previous case, the clusters movies and actors are connected with an N to N relationship. Although the clusters have a different representation the movies cluster represents the main cluster based on the order of clusters in the \text{DataMockups.dataContentPage}. The main (top most) cluster is the parent element of other clusters regardless of the representation.
Since \( \text{size}(\text{actor.name}) < \text{numOcc} \) we need a \texttt{GROUP\_CONCAT()} function on the main cluster column \textit{movies.title}. In contrast to the first case, we need to group by the results by the column which has lower size and has to be a unique attribute. In our example, we need to group by the \textit{actor.name}. After the whole process is finished we get the complete SELECT clause statement and an indication for the presence of the \texttt{GROUP\_BY} statement.

As shown in Listing 4.3, the responsible function for inferring the SELECT clause is the \texttt{inferSelect} function, which takes as input parameter the result returned from the \texttt{inferFrom} function. The function returns the SELECT clause and a boolean value indicating the presence of a \texttt{GROUP\_BY} statement. In the first step of SELECT clause inference process we determine if the IM is a detail page or a list page as previously explained. Next, we check the size of values for each column displayed in the IM using the \texttt{compareColumns} function. Each column size greater than zero, is added to the \texttt{column\_size} array. Then the \texttt{allValuesSame} function checks if the array values are the same by returning a true or false boolean value.

```javascript
1  function compareColumns(page_content)  
2   {  
3     var column_size= [];  
4     for(var c in page_content)  
5     {  
6       if(page_content[c].length>0)  
7       {  
8         column_size.push(page_content[c].length);  
9       }  
10     }  
11     var same_values=allValuesSame(column_size);  
12     return same_values;  
13   }
```

Listing 4.7: The compareColumns function

Based on the result of this function, we infer the SELECT clause by adding the columns to the \texttt{select\_array} using two different functions, \texttt{getSelectSameValues} and \texttt{getSelectNotSameValues}. The first function adds each column to the \texttt{select\_array} without any function as previously explained and shown in Listing 4.8.

```javascript
1  function getSelectSameValues(page_content,from_clause_objects,dictionary)  
2  {  
3     var from_clause=Object.keys(from_clause_objects);  
4     var select_array=[];  
5     for (var i = 0; i < from_clause.length; ++i) {  
6       for (var c in page_content) {  
7         if (dictionary[c] == from_clause[i]) {  
8           var cols=[];  
9           cols=from_clause_objects[from_clause[i]].columns;  
10           for(var j=0;j<cols.length;++j) {  
11             var isid = cols[j].includes("ID");  
12             if(!isid) {  
13               var column =from_clause[i] + "." + cols[j];  
14               select_array.push(column);  
15             }  
16           }  
17         }  
18       }  
19     }  
20  }
```

Listing 4.8: The getSelectSameValues function
The `getSelectSameValues` function determines the SELECT clause on a mockup with different values for column sizes, which indicates the presence of a grouped result set. The code snippet for the `getSelectSameValues` function conditions is shown in Listing 4.9. However, not all columns have grouped results and we need to distinguish between five cases that depend on the occurrence of the main cluster in the IM defined as `numOcc` and the amount of values for a specific column defined as `size(C)`.

- If `size(C) == numOcc`, where C is a column in the IM and there are no columns T such that `size(T) < numOcc`, the column C is added without the GROUP_CONCAT() function (e.g. other columns, except the genre of the example mockups shown in Figure 4.2).

- If `size(C) == numOcc`, where C is a column in the IM and there are columns T such that `size(T) < numOcc`, the column C is added with the GROUP_CONCAT() function, then the groupby variable is set to true because we have a list page (e.g. the title column of the example mockup shown in Figure 4.3).

- If `size(C) < numOcc`, where C is a column in the IM and there are columns T such that `size(T) < numOcc`, the column C is added without the GROUP_CONCAT() function (e.g. the actor column of the example mockup shown in Figure 4.3).

- If `numOcc == 1` and `size(C) > numOcc` where C is a column in the IM and there are no columns T such that `size(T) < numOcc`, the column C has grouped results and is added with GROUP_CONCAT() function and does need a group by statement since it is a detailed page (e.g. genre column of the example mockup shown in Figure 4.2 (a)).

- If `numOcc > 1` and `size(C) > numOcc` where C is a column in the IM and there are no columns T such that `size(T) < numOcc`, column C has grouped results and is added with GROUP_CONCAT() function, then the groupby variable is set to true because we have a list page (e.g. genre column of the example mockup shown in Figure 4.2 (b)).

```java
if (attribute_size==page_numOcc && mc_greater_others==false)
{
    var column =from_clause[i] + "." + cols[j];
    select_array.push(column);
}
else if (attribute_size==page_numOcc && mc_greater_others==true)
{
    return select_array;
}
```
4.5.5 Infer WHERE Clause

The last and the most challenging part of the query inference is the WHERE clause. This clause should be concise (without too many conditions adapted for target users) and minimize the imprecision between the inferred query and the query used to display the data in the mockup. As described in Section 3.2.2, the WHERE clause consists of the join and comparison predicates, and they are inferred in two separate processes.

The first process gets the join predicates for the tables that contain the data displayed in the IM. For each of the detected relationships, we check whether the relationship is contained on data displayed in the IM. If so, the corresponding join predicate is constructed based on the relationship type. If the number of relationships contained in the data displayed on the IM is greater than one, the join predicates for each of them are connected with an \texttt{AND} operator. Finally, we get the complete string of the join predicate(s).

The second part of the WHERE clause consists of the condition that is specified using comparison or logical operators, clearly if the original query for the data displayed in the IM has a particular condition. To find out the conditions, we first need to check if the rate of occurrence of the main cluster in the \texttt{DataMockups.dataContent} and \texttt{DataMockups.dataContentPage} is equal. In this case, the page contains \texttt{ALL} the data that the entire project shows, therefore we cannot infer a selection predicate. We can immediately proceed to check the query accur-
acy and return the complete SQL query to the user.

Otherwise, we exclude image columns and continue to find if there are conditions on other columns. The reason for excluding image columns is that, image’s alternate text or source attribute usually are not conditions in the WHERE clause. To find out the conditions for other columns, we first need to distinguish between mockup types. For detailed pages, we just need to find the column that uniquely identifies that particular row from the table. For list pages, we need to compare the content of the elements shown in the IM with the content of all elements in the project, i.e. `DataMockups.dataContentPage` and `DataMockups.dataContent` respectively, by checking the similarity of the elements in both objects. If the values of a particular column are the same, we have an equality condition regardless of the value type. Otherwise, we distinguish between number and string values. With number column values we determine ranges and numerical conditions, whereas with strings values equalities and LIKE conditions. Once the conditions are found, the next step is to check which of the conditions minimizes the difference between the data returned by the query and data displayed in the IM and return the condition that gives the highest accuracy to the user.

As introduced in Listing 4.3, the WHERE clause is inferred using the `inferWhere` (see Appendix A for the complete code snippet). This inference process starts with the mapping of both `DataMockups.dataContentPage` and `DataMockups.dataContent` objects to two new objects such that each key represents the cluster name and the value corresponds to an array of the column values called `page_content_array` and `all_content_array`. Before going through the column values in the `page_content_array`, we check if the data in the IM are displayed using a selection predicate. Therefore, we check if the sizes of the `page_content_array` and `all_content_array` objects are different. If so, for each key (i.e. column) of the `page_content_array`, the function first removes the image columns. Then, if the size of the array of column values is one (for detailed pages which columns does not have grouped result sets), this column with the corresponding value is added as where condition with the `=` comparison operator. Otherwise, we need to check for similarity between column values.

However, to have a concise condition the columns belonging to the data displayed in the IM are ranked using the `elementRanking` function. Only the column that uniquely identifies the table rows (i.e. has highest rank) is added to the `where_conditions` array. The `elementRanking` function computes the rank or uniqueness of the element based on four factors which are stored in four different arrays:

- tag type (`tagrank`)
- font-size of the content (`fontsizerank`)
- length of the content (`lengthrank`)
- elements current top position relative to the offset parent (`positionrank`)

The function first defines a `tagrank` array where `< h1 >` HTML heading tag has the highest ranking, located in position zero in the array, proceeded by the other five headings and other html tags such as span, div, p, etc. After that for each of the elements, it determines the other three factors using the JavaScript `length` method and the jQuery `css`("font-size")
and position().top methods, and puts the values in the three corresponding factor arrays listed above. The lengthrank and positionrank arrays are sorted in ascending order. This means that the element with the shortest length and the element in the topmost position have the highest ranking, located in position zero in the sorted array.

In contrast, the fontsizerank array is sorted in descending order, meaning that the element content with the biggest font-size has the highest ranking, located in position zero in the sorted array. The ranking of each element is computed as the sum of elements position in each of the above-mentioned arrays as shown in Equation 4.1.

\[
\text{attr\_rank}(\text{element}) = \text{tagrank}.indexOf(\text{element}) \\
+ \text{lengthrank}.indexOf(\text{element}) \\
+ \text{positionrank}.indexOf(\text{element}) \\
+ \text{fontsizerank}.indexOf(\text{element})
\] (4.1)

Finally, the element with the smallest sum value is the column that uniquely identifies the table rows and has the highest ranking. This unique column is also used to group the elements in case a GROUP BY statement was detected in the SELECT clause inference described in the previous section.

In order to find out if there is a condition in the list page or detailed page that has columns which contain grouped result set, the inferWhere function checks for similarities between column values in the page\_content\_array object and compares them with the corresponding column values in the all\_content\_array object. We create two objects that map the elements in the mockups to the amount of times they occur, both in the mockup (in the page\_occ object) and in the whole project (in the all\_pages\_occ object).

If the size of page\_occ is one, then all values for that column are the same, which indicates the presence of a select predicate in the IM. Therefore, the condition for the column name and the value is added where\_conditions array, using the = comparison operator. Otherwise, we distinguish between the number and string types of the array elements and the WHERE conditions are determined in different manner as previously explained.

If the values of the array elements are not numbers, the function checks the size of the of all\_pages\_occ and page\_occ for each specific column C and computes the ratio between the two sizes. If the ratio is smaller than 1, the IM contains less values than the whole collection for the C in the database, meaning a select condition might exist. However, before adding the condition we need to check whether the values of the column C belong to values that uniquely identify a particular element. For example, a mockup which displays a subset of movies that belong to a particular genre cannot have a condition on the title, but we need to find the common genre of movies. We define two new boolean variables that would help us to eliminate the unnecessary conditions. The respective function checks if all the elements in the mockups occur only once and sets a boolean value to both variables, both for the mockup (page\_occ\_one) and in the whole project (all\_occ\_one).

Therefore, we check whether the occurrences of column values in all\_pages\_occ and page\_occ are all one, all\_occ\_one and page\_occ\_one respectively, to eliminate the unnecessary conditions. If all\_occ\_one == false && page\_occ\_one == false, there exists a column
value displayed in the IM, which is common to multiple elements and used as a select predicate by the original query. Finally, the condition for the respective column/value pair is added to the where_conditions array using the = comparison operator.

If the values of the array elements are numbers, we try to determine if there is a range query or a numerical condition. We first find the minimum (minval) and maximum (maxval) values for each of the column arrays and add three different conditions to the where_conditions array defined as follows.

- \textit{column} \geq \textit{minval}
- \textit{column} \leq \textit{minval}
- \textit{column} BETWEEN \textit{minval} AND \textit{maxval}

Additionally, the WHERE clause can contain conditions with logical operator such as LIKE. Those conditions are found using the getLikeConditions function. This function infers LIKE conditions for columns that uniquely identify the table rows as previously explained, and checks for a common pattern between column values. The function splits the values of each column by words using the JavaScript \texttt{split} method with a space separator and check for common words between the rows in that column. If the common word occurrence is equal to the number of rows the column with the common word is added as LIKE condition.

Based on the described heuristics the number of conditions can be zero or equal and greater than one. Therefore, to find out the WHERE clause for both cases we need to check the query with all inferred clauses using the check_query function. In case the number of conditions is greater than one, we need check multiple queries and return the best one. The input parameters to this function are: FROM clause, SELECT clause, an array containing the WHERE conditions, the string containing the JOIN predicates, the boolean value of group by variable and the unique column for the GROUP by statement. Depending on the input parameter values it constructs the complete SQL query statement, which along with database credentials are sent to the server via an AJAX request using the jQuery post method. The server runs the query and returns the result encoded in JSON format shown on Listing 4.10.

```json
[
    {
        "img_alt": "",
        "img_src": "https://image.tmdb.org/t/p/w300_and_h450_bestv2\t90Y3G8UGQp0f0DrP60wRu9gfrH.jpg",
        "title": "Avengers: Age of Ultron",
        "rating": "7.4",
        "releasedate": "2011-04-21",
        "overview": "When Tony Stark tries to jumpstart a dormant peacekeeping program, things go awry and Earths Mightiest Heroes are put to the ultimate test as the fate of the planet hangs in the balance.",
        "genre_att": "Adventure|Fantasy",
        "actors_att": "Robert Downey Jr.|Chris Hemsworth|Mark Ruffalo|Chris Evans"
    }
]
```

Listing 4.10: The JSON result returned from the database
The check_query function then compares the returned JSON data name/value pairs result with the columns in the IM (i.e. DataMockups.dataContentPage). Firstly, it compares if each column name is included in the JSON data names and then compares the values of the columns with JSON data values. If the values in both objects for a specific column are the same, the function increases a counter for each matched column which is later used to infer the query accuracy. The return value is an object that contains the following values:

- If the query has a condition, it returns the condition that gives higher query accuracy, otherwise it is empty.
- The join predicates of the generating query.
- Query accuracy computed as ratio between the value of the counter described above and the number of columns in the IM.
- Columns for which the inferred query results and data displayed are incompatible, clearly if the query accuracy is not 100%.
- Additional where condition suggestions for which the query is accurate. The query may be accurate for more than one conditions since the heuristics in the conditions detection process may find conditions that select the same rows. For example, a mockup which displays movies with rating 8.0 to 8.3 and 8.3 is the highest rating. In this case, the possible conditions include movies.rating BETWEEN 8 AND 8.3 and movies.rating ≥ 8.

4.5.6 Simplified SQL interface and SQL interface

Once the query inference process is finished, each of the SQL clauses should be displayed on the simplified user interface and SQL interface. Moreover, it should be possible to view the query results and update the SQL query. Setting the SQL interface is more straightforward because we just need to set the values to the corresponding textareas using the jQuery `val` function. In the simplified interface, instead the `html` function is used. However, the html content is adapted based on the SELECT clause columns, WHERE clause condition types and the presence of GROUP BY statement.

The query accuracy is displayed on a Bootstrap Stacked Progress Bar\(^\text{8}\) which contains two bars for displaying the query accuracy and errors. Their width is updated based on the query accuracy ratio using the `set_queryAccuracy` function shown in Listing 4.11. If the accuracy ratio is one the Stacked Progress Bar shows only one bar with full width, otherwise both bars are shown with their corresponding width. An additional label shows the column names for which the query result and mockup data are incompatible.

\(^\text{8}\)http://www.w3schools.com/bootstrap/bootstrap_progressbars.asp
In the SQL interface, the users have the possibility to update the SQL clauses by changing their values in the corresponding textareas. A change event handler is bound to each of the textareas. When a change event occurs all the elements that display that value are updated (i.e. the corresponding element in the simplified interface and the gray block element that contains the complete SQL query code).

In addition, the users can add the ORDER BY suggestions, as briefly introduced in Section 3.4.1. The ORDER BY keyword, is detected separately from the query inference process, because it does not affect the query accuracy. However, it should be taken in consideration, before the augmentation process if the result set displayed in the IM should be sorted. Basically, the function responsible for detecting the sorted columns, checks whether the values of the columns are sorted in ascending or descending order for three types of data; number, date and string and displays to the user the corresponding column, the ASC/DESC keyword and a Font Awesome icon customized on the order and data type. Each of the suggested columns can be added using a toggle switch button which has attached a change event handler and updates the SQL query code as previously described.

Once all the necessary changes are performed, the values of each of the SQL query clauses
are updated. By clicking the View Results button, the ViewQueryResults function shown in Listing 4.11 is called. This function makes two POST requests to the server. The first request is accomplished using the recheck_query, which gets a JSON format as shown in Listing 4.10 and computes the query accuracy in the same manner as check_query described in the previous section. Then the query accuracy is displayed using the set_queryAccuracy function explained above. The second request returns the query result in a tabular format so that users can have a database representation of the query result, corresponding to the data displayed in the IM.

4.6 Mockup Augmentation

As mentioned in Section 3.5, once the users are satisfied with the inferred query, they can use it to create the augmented mockup. The final value of the complete SQL statement is displayed in the gray block (see Figure 3.10) which is got using the jQuery html method. The mockup augmentation process is done in six steps and depends on both server and client work. The steps of the process are shown in Figure 4.4.

Step 1 consists of extracting the HTML elements that will be inserted in the mockup. The extraction of elements is mostly implemented using the jQuery manipulation methods. Furthermore, some of them are used to get the values from the desired elements as the above mentioned html method.

In order to show the results of the query in the mockup, we need to replace the values of the cluster elements with the PHP code that will print the results in the mockup. To achieve this, we need to extract the parent cluster element and clone it. The parent cluster element, is the first HTML element that contains the data-cluster attribute and is the parent of other children cluster elements. The reason for cloning is to prevent the modification of the elements content of the current mockup displayed on the DataMockups tool.

---

9https://api.jquery.com/category/manipulation/
4.6. MOCKUP AUGMENTATION

To separate the rest of the content from the cloned element and identify the area in which we need to place the PHP code, we insert an invisible markup element (i.e. HTML comment). This markup is inserted before and after the parent cluster element and later replaced by the PHP statements. The insertion of markups is done using the `insertBefore` and `insertAfter` jQuery methods. However, this insertion depends on the type of the mockup. If it is a list page (i.e. parent cluster occurrence is greater than one) the markup is inserted before the first and after the last cluster parent element, otherwise it is inserted before and after the cluster parent element. A code snippet for this insertion is shown in Listing 4.12.

```javascript
if (cluster_elements.length==1)
{
    $("<!--markupb-->").insertBefore(cluster_elements[0]);
    $("<!--markupe-->").insertAfter(cluster_elements[0]);
    data=$(cluster_elements[0]).clone(true);
}
else
{
    var elem_length=cluster_elements.length
    $("<!--markupb-->").insertBefore(cluster_elements[0]);
    $("<!--markupe-->").insertAfter(cluster_elements[elem_length-1]);
    data=$(cluster_elements).clone(true);
}
```

Listing 4.12: Markup insertion in the HTML document

Using the markup elements, we can separate the HTML content in three parts as follows:
• HTML elements located before the parent cluster element(s)
• parent cluster element(s)
• HTML elements located after the parent cluster element(s)

The separation is performed using the JavaScript `split` method. This method operates on the `outerHTML` attribute of the whole page HTML content. Once we have divided the content and before sending the extracted data to the server, we need to replace the text content of the cloned parent cluster elements with the respective PHP command that outputs the data from the query result. While, the modification of elements is done on the client using JavaScript, the altered PHP file is created on the server. However, the content to be modified is not always the text part of the element and we need to differentiate between HTML elements as follows.

- `<div><p><span><h1>...<h6><li>`, etc. - replace text content
  $\text{(this).text("<?php echo row['" + name + "']; ?>")}

- `<a>` - replace the link text content and the URL the link points to
  $\text{(this).text("<?php echo row['" + name + "']; ?>")}$\text{(this).attr("href","<?php echo row['" + name + "_link" + + "']; ?>")};$

- `<img>` - replace the alternate text for an image and the image’s source attribute
  $\text{(this).attr("alt","<?php echo row['" + name + "_alt" + + "']; ?>")}$\text{(this).attr("src","<?php echo row['" + name + "_src" + + "']; ?>")};$

The `name` is the cluster name of the element, whereas `_link`, `_alt`, `_src` are appended values to the original cluster in the Element Detection process of DataMockups tool. They all correspond to the SELECT clause columns.

Another thing that we need to consider are the elements that need to display data from the columns that have grouped result set (i.e. use `GROUP_CONCAT` function). The corresponding replacement for these elements is `elem.text("<?php echo $ + name + "");` Since the query for these columns will return a grouped result we would need to cycle through the result and display them in the respective element. In order to identify the elements which contain grouped result set, we add a markup element in the same way as described in the beginning of this section. The replacement of content is done similarly as described above, by replacing the markups with the needed PHP code.

Once all the needed data to create the augmented mockup are extracted, client sends the database information and the extracted data to the DataMockups server (step 2) using the jQuery `post` method.

Before the augmented mockup can query the database, it needs to connect to the server and the corresponding database. Since the database parameters such as: database address, username,
password and database name are the same for all the mockups of a particular project the server first creates a database configuration file called `dbconfig.php` which is shown in Listing 4.13 and corresponds to the step 3 of the mockup augmentation process.

```php
$servername = "localhost";
$username = "root";
$password = "root";
$dbName = "Recipes";
```

Listing 4.13: The dbconfig.php file content

In the fourth step, the server creates the augmented mockup. Besides the data received from the client, the server adds additional PHP code to the content that will be inserted in the augmented mockup. The content of the augmented mockup should include the following:

- The HTML elements that are located before the parent cluster element
- The PHP code statements that include the database configuration file, query the database and display the query results in the cluster elements
- The HTML elements that are located after the parent cluster element

An example of augmented mockup and a detailed description of the content is shown in Appendix B. If the mockup augmentation is completed successfully (steps 1-4), the client receives a message that everything succeeded, otherwise it gets an error message. In addition, the server sends the file name and its path location along with the message (step 5). In the final step in case of success, a success callback function is called that puts the message, the link that points to the created file and the path where the file is located in an alert modal. In case of failure an error callback function is executed that displays the error message in an alert modal.
This chapter describes the evaluation of the QI component. Section 5.1 describes an evaluation performed on different mockup designs with data from different domains and reports on how well the QI component infers the queries. In Section 5.2, we report the results on the efficiency of the query inference algorithm in terms of runtime. The last section summarizes the results from both evaluations.

5.1 Evaluation in terms of performance and quality of generated queries

5.1.1 Study description

In order to tune the heuristics described in the algorithms of Section 4, we started with a simple set of 'training' projects such as recipes, movies, blog, online clothes shop, job search platform, whose mockups have different designs and the content covers different domains that will be presented later. The reason for choosing those different domains is based on the type of data they provide and some of these domains are also used in evaluation data extraction systems. The complexity of inferred schemas is different, which has a great impact on the way the data is displayed on a specific mockup and on the type of queries that the QI component is able to infer. At this stage of the assessment, we found that the schema suggestion component had several bugs. Two important issues were the detection of the wrong type of relationship and the dependency of the schema formation process on the order that mockup clusters were added to the schema. However, with help from another developer of the DataMockups system we managed to fixed some of these bugs. We found out that the heuristics work correctly for most of the 'training' set of projects. We will describe the projects individually in the next section and emphasize the issues that occurred.
At the beginning of the training stage, each of the project consisted of three mockups. Two detailed pages that display specific information for a particular record and one list page that displays information for all records.

After the schema was inferred correctly and the database was generated, we could infer different types of queries of the following forms:

- **SELECT - FROM**, a select query that displays specific columns of all the records in a particular database
  - **ORDER BY** for sorted result sets either in ascending or descending order
  - **GROUP BY** for a grouped result set, used in conjunction with **GROUP_CONCAT()** function
- **SELECT - FROM - WHERE** select query that displays specific columns of a particular record
- **SELECT - FROM - WHERE** select query that displays specific columns of all records that match a particular condition
  - **ORDER BY** for sorted result sets either in ascending or descending order
  - **GROUP BY** for grouped result set, used in conjunction with **GROUP_CONCAT()** function

Additional mockups were added to each of the projects in order to prove that QI is able to detect queries for different representations of data. In order to run the heuristics of solution and prove that is able to generate queries of different types for more sophisticated mockup designs and various types of content, we created a new set of 'test' projects. Their domains include data for books, boat booking, notebook shop, apartment rental and FoodCASE.

To evaluate the performance and quality of generated queries, we will compare three metrics the **query accuracy** and **precision & recall**. The first metric is shown to the user after the query is inferred in percentages, which was generally described in Section 4.5.5. A percentage represents a ratio out of 100 and is calculated based on the difference of the results displayed in the mockup and results that are generated from the inferred query. Basically, it checks for each column if the query result cells are identical to the respective values displayed in the IM. A column is taken as accurate only if all values match. In contrast to the **precision & recall**, it does not measure the percentage of all relevant items/cells that is returned by the inferred query, in case it returns more or less items/cells for a specific column, that are not displayed in the IM (i.e. recall).

The other two metrics as previously introduced are the **precision & recall**. In our scenario, the **precision & recall** metrics are defined on the individual columns/fields of the query results and are defined as follows.

**Precision** - a measure of the ability of the query to present only the relevant data. (i.e. only the data displayed in the mockup)

\[
\text{precision} = \frac{\text{number of relevant items/cells retrieved}}{\text{total number of items/cells retrieved}}
\]
Recall - a measure of the ability of the query to generate all relevant data (i.e. all data displayed in the mockup)

\[
recall = \frac{\text{number of relevant items/cells retrieved}}{\text{total number of relevant items/cells}}
\]

However, we also check in the values of the individual items/cell, due to the usage of the \texttt{GROUP\_CONCAT()} function on the inferred queries. The \texttt{GROUP\_CONCAT()} function returns a string with concatenated values from a grouped result set. If the concatenated string is incomplete or different from the one expected, we mark it as irrelevant/wrong.

The results corresponding to each of the evaluated projects are displayed on individual tables and each table row consists of the values of the following columns.

- MID - a unique identifier for each mockup
- mockup type - describes the data displayed on a particular mockup
- query type - describes the type of the inferred query
- query accuracy - accuracy measurement expressed in percentages
- precision & recall - results for the precision & recall metrics

### 5.1.2 Results of the training set of projects

As mentioned in the previous section, the QI component was evaluated in two stages with a training and test set of projects. The training set of project was used while developing the system in order to train the heuristics. In the next paragraphs, we will describe the projects used to train the QI component and show the results for the two measurements described in the previous section. While this section presents only a description of the mockup types, inferred queries and the measurement results, the complete list of the inferred queries SQL code for each project can be found in Appendix C.

**Recipes**

The recipes project consists of several mockups that display data about recipes, such as recipe title, ingredients, directions, recipe type and preparation time. Figure 5.1 shows screenshots of two different mockups. The mockup on the left-hand side is a detailed page that displays data about a particular recipe, whereas the one on the right-hand side is a list page that shows data about all recipes.
5.1. EVALUATION IN TERMS OF PERFORMANCE AND QUALITY OF GENERATED QUERIES

### My recipe site

**Oatmeal chocolate chip cookies**

**Ingredients**
- 1 cup shortening or margarine
- 1 cup brown sugar
- 1 cup white sugar
- 2 eggs
- 2 tablespoons water
- 1 tablespoon vanilla extract
- 1 1/2 cups flour
- 1 teaspoon baking soda
- 1 teaspoon salt
- 5 cups oats
- 1 cup chocolate chips

**Directions**
Cream together the shortening and the sugar. Add the eggs, water and vanilla extract. Mix together the flour, baking soda and salt and add to above. Add the oats with the dry ingredients and mix well. Stir the chocolate chips into dough. Drop from a teaspoon onto greased baking sheet or lined with parchment paper. Bake at 375 10-15 minutes. Makes about 5 dozen cookies.

### (a) Detailed Page

<table>
<thead>
<tr>
<th>Recipe Name</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mousse au chocolate</td>
<td>Dessert</td>
</tr>
<tr>
<td>Cheesecake</td>
<td></td>
</tr>
<tr>
<td>Cranberry Carrot Cake</td>
<td>Cake</td>
</tr>
<tr>
<td>Chocolate Chunky Sour Cream Coffee Cake</td>
<td>Cake</td>
</tr>
<tr>
<td>Date Cake</td>
<td>Cake</td>
</tr>
<tr>
<td>Dried Tomato Cake</td>
<td>Date</td>
</tr>
<tr>
<td>Dried Minted Lemon Cake</td>
<td>Date</td>
</tr>
<tr>
<td>Eggless Date Cake</td>
<td>Date</td>
</tr>
<tr>
<td>Zucchini Bread</td>
<td>Date</td>
</tr>
<tr>
<td>Oatmeal Chocolate chip cookies</td>
<td>Cookies</td>
</tr>
<tr>
<td>Peanut butter cookies</td>
<td>Cookies</td>
</tr>
<tr>
<td>Shortbread cookies</td>
<td>Cookies</td>
</tr>
<tr>
<td>Ginger cookies</td>
<td>Cookies</td>
</tr>
<tr>
<td>Almond cookies</td>
<td>Cookies</td>
</tr>
<tr>
<td>Almond fudge chocolate chip cookies</td>
<td>Cookies</td>
</tr>
<tr>
<td>Almond fudge chocolate chip cookies</td>
<td>Cookies</td>
</tr>
<tr>
<td>Pecan cake</td>
<td>Pecan cake</td>
</tr>
<tr>
<td>Cherry pie</td>
<td>Cherry pie</td>
</tr>
<tr>
<td>Black Forest cake</td>
<td>Black Forest cake</td>
</tr>
<tr>
<td>Black Forest cake</td>
<td>Black Forest cake</td>
</tr>
<tr>
<td>Black Forest cake</td>
<td>Black Forest cake</td>
</tr>
<tr>
<td>Black Forest cake</td>
<td>Black Forest cake</td>
</tr>
<tr>
<td>Black Forest cake</td>
<td>Black Forest cake</td>
</tr>
</tbody>
</table>

### (b) List Page

**My recipe site**

<table>
<thead>
<tr>
<th>Recipe Name</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mousse au chocolate</td>
<td>Dessert</td>
</tr>
<tr>
<td>Cheesecake</td>
<td></td>
</tr>
<tr>
<td>Cranberry Carrot Cake</td>
<td>Cake</td>
</tr>
<tr>
<td>Chocolate Chunky Sour Cream Coffee Cake</td>
<td>Cake</td>
</tr>
<tr>
<td>Date Cake</td>
<td>Cake</td>
</tr>
<tr>
<td>Dried Tomato Cake</td>
<td>Date</td>
</tr>
<tr>
<td>Dried Minted Lemon Cake</td>
<td>Date</td>
</tr>
<tr>
<td>Eggless Date Cake</td>
<td>Date</td>
</tr>
<tr>
<td>Zucchini Bread</td>
<td>Date</td>
</tr>
<tr>
<td>Oatmeal Chocolate chip cookies</td>
<td>Cookies</td>
</tr>
<tr>
<td>Peanut butter cookies</td>
<td>Cookies</td>
</tr>
<tr>
<td>Shortbread cookies</td>
<td>Cookies</td>
</tr>
<tr>
<td>Ginger cookies</td>
<td>Cookies</td>
</tr>
<tr>
<td>Almond cookies</td>
<td>Cookies</td>
</tr>
<tr>
<td>Almond fudge chocolate chip cookies</td>
<td>Cookies</td>
</tr>
<tr>
<td>Almond fudge chocolate chip cookies</td>
<td>Cookies</td>
</tr>
<tr>
<td>Pecan cake</td>
<td>Pecan cake</td>
</tr>
<tr>
<td>Cherry pie</td>
<td>Cherry pie</td>
</tr>
<tr>
<td>Black Forest cake</td>
<td>Black Forest cake</td>
</tr>
<tr>
<td>Black Forest cake</td>
<td>Black Forest cake</td>
</tr>
<tr>
<td>Black Forest cake</td>
<td>Black Forest cake</td>
</tr>
<tr>
<td>Black Forest cake</td>
<td>Black Forest cake</td>
</tr>
<tr>
<td>Black Forest cake</td>
<td>Black Forest cake</td>
</tr>
</tbody>
</table>

**Figure 5.1: Recipe project example mockups**

In addition to the mockups shown above, the recipes project includes other mockups that represent the recipe database data in various forms as listed in Table 5.1. The inferred schema from the listed mockups, is shown in Figure 5.2.
The inferred query types are defined as follows:

- **QT1**: Viewing only selected columns of all recipes stored in the database
- **QT2**: Viewing only selected columns of a particular recipe
- **QT3**: Viewing only selected columns of recipes that matches a specific condition

The complete list of the inferred queries SQL code be found in Table C.1 (Appendix C).

<table>
<thead>
<tr>
<th>MID</th>
<th>Mockup Type</th>
<th>Query Type</th>
<th>Query Accuracy</th>
<th>Precision &amp; Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>List page that displays the recipe title, type and preparation time of all recipes</td>
<td>QT1</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>R2</td>
<td>Detailed page that displays recipe title, a list of ingredients and a detailed description of recipe with title &quot;Oatmeal chocolate chip cookies&quot;</td>
<td>QT2</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>R3</td>
<td>Detailed page that displays recipe title, a list of ingredients and a detailed description of recipe with title &quot;Peanut butter cookies&quot;</td>
<td>QT2</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>R4</td>
<td>List page that displays recipe title, type and preparation time of all recipes with type Cake</td>
<td>QT3</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
</tbody>
</table>
5.1. EVALUATION IN TERMS OF PERFORMANCE AND QUALITY OF GENERATED QUERIES

| R5 | List page that displays recipe name, type and preparation time all the recipes with Preparation time: 30 minutes | QT3 | 100 % | P=1, R=1 |

Table 5.1: Recipe Project evaluation results

As we can see from the results displayed in Table 5.1, there is a strong correlation between the query accuracy and precision & recall measurements. In this case, the correlation is strong for all mockups, the inferred query accuracy is 100%, meaning that the inferred query can generate the same result set displayed in the mockup. The high precision value means that the query returned only the data displayed in the mockup, while the high recall value means that the query returned all the data displayed in the mockup.

**Movies**

The screenshots for the movies project mockups can be seen in Figure 5.4. They show two different types of mockups, that display a list of movies and details about a particular movie. A special characteristic of the movies data are the genre and actors. As we can see from the mockups, movies may belong to more than one genre and multiple actors can cast in the same movie. Therefore, the inferred queries for movies mockup should contain a `GROUP_CONCAT()` function in the SELECT clause. The complete list of the inferred SQL queries be found in Table C.2 (Appendix C). The movies project consists of several mockups listed in Table 5.2 and the inferred schema from the listed mockups, is shown in Figure 5.3.

![Figure 5.3: Movies database schema](image)

The inferred query types are defined as follows:

- **QT1**: Viewing only selected columns of all movies stored in the database
• QT2: Viewing only selected columns of a particular movie
• QT3: Viewing only selected columns of the movies that matches a specific condition
• QT4: Viewing actors name of all actors stored in the actors table

Figure 5.4: Movies project example mockups
## 5.1. Evaluation in terms of performance and quality of generated queries

<table>
<thead>
<tr>
<th>MID</th>
<th>Mockup Type</th>
<th>Query Type</th>
<th>Query Accuracy</th>
<th>Precision &amp; Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>List page that displays the movie image, title, rating, release date, genre(s) and overview of all movies</td>
<td>QT1</td>
<td>100%</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>M2</td>
<td>Detailed page that displays the movie image, title, rating, release date, genre(s), overview and top billed cast of movie with title &quot;Avengers: Age of Ultron&quot;</td>
<td>QT2</td>
<td>100%</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>M3</td>
<td>Detailed page that displays the movie image, title, rating, release date, genre(s), overview and top billed cast of movie with title &quot;Thor&quot;</td>
<td>QT2</td>
<td>100%</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>M4</td>
<td>List page that displays movie titles of the actor Chris Hemsworth.</td>
<td>QT3</td>
<td>100%</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>M5</td>
<td>List page that displays movies that belong do genre &quot;Adventure&quot;</td>
<td>QT3</td>
<td>86%</td>
<td>P = 6/7 ≈ 0.86, R = 6/7 ≈ 0.86</td>
</tr>
<tr>
<td>M6</td>
<td>List page that displays movies whose range is between 8.0 and 8.3</td>
<td>QT3</td>
<td>100%</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>M7</td>
<td>List all actors who play in the movies</td>
<td>QT4</td>
<td>100%</td>
<td>P=1, R=1</td>
</tr>
</tbody>
</table>

Table 5.2: Movies project evaluation results

The results shown in Table 5.2, indicate that there is a strong correlation between the query accuracy and precision & recall measurements. We can see that only the accuracy of the query for mockup M5 is 86%, but still the accuracy value is high and gives room for improvement. The inferred query for the M5 mockup is shown in Listing 5.1.

```sql
1 SELECT movies.img_alt, movies.img_src, movies.title,
2 movies.rating, movies.releasedate, movies.overview,
3 GROUP_CONCAT(DISTINCT genre.genre_att SEPARATOR ' | ') AS genre_att
4 FROM movies, genre, moviesgenre
5 WHERE moviesgenre.moviesID=movies.ID
6 AND moviesgenre.genreID=genre.ID
7 AND genre.genre_att='Adventure'
8 GROUP BY movies.title
```

Listing 5.1: The inferred query for the M5 mockup
As we can see, the WHERE clause specifies a condition for the `genre_att="Adventure"`. However, the WHERE clause gets processed before the GROUP BY, and so it does not have access to `genre_att`, the concatenated value of genres for a particular movie. This means that the WHERE clause applies to individual rows and cannot be specify a search condition for a group or an aggregate function used in the SELECT clause.

The problem can be solved by using a HAVING clause, because HAVING applies to the group as whole and gets processes after GROUP BY. Nevertheless, the QI component does not support HAVING clauses but expert users with little effort can delete the condition in the WHERE clause and put a HAVING clause in the Additional Functions textarea. The correct HAVING clause is, `HAVING genre_att LIKE '%Adventure%'` which is used to constrain the result set on the aggregate function `GROUP_CONCAT()`.

For all other mockups, the inferred query accuracy is 100%, meaning that the inferred query can generate the same result set displayed in the mockup. The high precision value of one means that the query returned only the data displayed in the mockup, while the high recall value means that the query returned all the data displayed in the mockup.

Blog

The blog is a simple one-page website that contains posts by multiple authors. In addition, each post may have multiple comments that can be seen on a detailed page. The screenshots for this project are shown in Figure 5.5. A specific characteristic about blog project is that the author may write blog posts, comment to other blog posts or even reply to comments inside his/her post which increases the difficulty of schema formation and inferred queries.

To evaluate the query inference additional mockups are created, which are listed in Table 5.3 and the inferred schema from the listed mockups, is shown in Figure 5.6. The previously developed schema inference component has a failure in detecting the correct multiplicity for the posts-author relationship, stating it as N to N. In fact, each post has only one author but it can have comments posted by multiple authors. Therefore, the author cluster appears multiple times inside the posts cluster.

The inferred query types are defined as follows:

- **QT1**: Viewing post information about all blog posts
- **QT2**: Viewing post information and comments about a particular post
- **QT3**: Viewing post information of all posts that matches a specific condition
5.1. EVALUATION IN TERMS OF PERFORMANCE AND QUALITY OF GENERATED QUERIES

Figure 5.5: Blog project example mockups
Figure 5.6: Blog database schema

<table>
<thead>
<tr>
<th>MID</th>
<th>Mockup Type</th>
<th>Query Type</th>
<th>Query Accuracy</th>
<th>Precision &amp; Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP1</td>
<td>List page that displays title, content, date, author and category of blog posts</td>
<td>QT1</td>
<td>80%</td>
<td>( P = \frac{23}{25} \approx 0.92, \ R = \frac{23}{25} \approx 0.92 )</td>
</tr>
<tr>
<td>BP2</td>
<td>Detailed page that displays title, content, date, author, category and comments of blog post with title &quot;Music&quot;</td>
<td>QT2</td>
<td>86%</td>
<td>( P = \frac{7}{8} \approx 0.875, \ R = \frac{7}{8} \approx 0.875 )</td>
</tr>
<tr>
<td>BP3</td>
<td>Detailed page that displays title, content, date, author, category and comments of blog post with title &quot;Travelling&quot;</td>
<td>QT2</td>
<td>86%</td>
<td>( P = \frac{7}{8} \approx 0.875, \ R = \frac{7}{8} \approx 0.875 )</td>
</tr>
<tr>
<td>BP4</td>
<td>List page that displays title, content, date, author and category of blog posts from author &quot;Blue Motion&quot;</td>
<td>QT3</td>
<td>100%</td>
<td>( P=1, \ R=1 )</td>
</tr>
</tbody>
</table>
5.1. EVALUATION IN TERMS OF PERFORMANCE AND QUALITY OF GENERATED QUERIES

<table>
<thead>
<tr>
<th></th>
<th>List page that displays title, content, date, author and category of blog posts from &quot;Food&quot; category</th>
<th>QT3</th>
<th>80%</th>
<th>$P = 4/5 \approx 0.8$, $R = 4/5 \approx 0.8$</th>
</tr>
</thead>
</table>

Table 5.3: Blog project evaluation results

We can see from Table 5.3 that the results between two measurements show a clear connection. However, the results for all mockups except for the BP4, are below 100%. The case for BP5 is similar to the movies example mentioned above, because posts can have multiple categories. A HAVING clause with a similar syntax would make the query 100% correct. The query accuracy result for the BP1 mockup is 80% because we get wrong values in the authors column, due to the previously described problem in the schema inference process.

In the BP2 and BP3 case, the accuracy of the inferred queries is 86%, due to the bug in the schema inference problem. Since the generating query needs to return two distinct columns for the author of the post and author of the comments even with the fixed bug we cannot be sure that we will get 100% accuracy. A possible solution for solving the problem could be to use aliases, however due to time constraints we could not fix the failure in the schema inference and cannot train anything to fix it. Nevertheless, the QI component is not able to infer queries that use aliases for mockups that display data from more complicated schemas such as blog schema. The complete list of the inferred SQL queries be found in Table C.3 (Appendix C). Nevertheless, the results are promising because the problem comes from the schema suggestion component and affects the generated queries quality. In general, the QI component is able to infer queries for mockups that display data, which are not generated using a query that requires aliases or HAVING clauses not supported by the QI component itself.

Online Clothes Shop
This project belongs to a classical online shop web-sites design. The mockups display data for various type of clothes. The list page displays less data, such as name, picture and price. On the detailed page, the users can see additional data such as category, size, availability, color and description. The example screenshots for this project are shown in Figure 5.4. An important characteristic of online clothes shops is that clothes titles are very descriptive, making it a perfect domain for experimenting with search queries, which make use of LIKE conditions.
Figure 5.7: Online clothes shop project example mockups

The online clothes shop project consists of several mockups listed in Table 5.4 and the inferred schema from the listed mockups, is shown in Figure 5.8.
5.1. EVALUATION IN TERMS OF PERFORMANCE AND QUALITY OF GENERATED QUERIES

The inferred query types are defined as follows:

- **QT1**: Viewing only selected columns of all products stored in the database
- **QT2**: Viewing only selected columns of a particular product
- **QT3**: Viewing only selected columns of the products that have a specified pattern in a title column
- **QT4**: Viewing only selected columns of the products that matches a specific condition

<table>
<thead>
<tr>
<th>MId</th>
<th>Mockup Type</th>
<th>Query Type</th>
<th>Query Accuracy</th>
<th>Precision &amp; Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>List page that displays the title, image and price of all clothes</td>
<td>QT1</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>C2</td>
<td>Detailed page that displays title, image, price, category, size, availability, color and description of item with title &quot;Floral Print M-Slit Maxi Dress&quot;</td>
<td>QT2</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>C3</td>
<td>Detailed page that displays title, image, price, category, size, availability, color and description of item with title &quot;Mom Jeans&quot;</td>
<td>QT2</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>C4</td>
<td>List page that displays the title, image and price of all clothes that contain &quot;dress&quot; keyword in their title</td>
<td>Q3</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
</tbody>
</table>
Table 5.4: Online clothes shop project evaluation results

The results presented in Table 5.4 show that the tool can infer different types of queries. The inferred queries WHERE clause includes conditions using LIKE and BETWEEN operators. The former operator is included in the query that generates the result for mockup C4, the latter is included in the query that generates the result for mockup C5. The WHERE clause of the mockups C3 and C2 includes a condition using the comparison = operator. The complete list of the inferred SQL queries be found in Table C.4 (Appendix C). It is clear from the results that that there is strong correlation between the results of the query accuracy and the precision & recall metrics, 100% and P=1, R=1 respectively.

Job Search Platform

The last example project assessed on the training stage of the QI component performance is a job platform website that aggregates job listing from all over the Internet. This project contains mockups that display data about job positions from different areas and companies. In the list page attributes such as location, country, job type, and salary are displayed. The detailed page, besides the data displayed on the list page, includes a full description and qualifications list. Figure 5.10 shows two example mockups belonging to this project.

The job search platform project consists of several mockups listed in Table 5.5 and the inferred schema from the listed mockups, is shown in Figure 5.9

| C5 | List page that displays the title, image and price of all clothes within a certain range price | QT4 | 100 % | P=1, R=1 |

Figure 5.9: Job search database schema
5.1. EVALUATION IN TERMS OF PERFORMANCE AND QUALITY OF GENERATED QUERIES

The inferred query types are defined as follows:

- **QT1**: Viewing only selected columns of all jobs stored in database
• QT2: Viewing only selected columns of a particular job

• QT3: Viewing only selected columns of the jobs that matches a specific condition sorted in ascending order

• QT4: Viewing information about all companies

<table>
<thead>
<tr>
<th>MID</th>
<th>Mockup Type</th>
<th>Query Type</th>
<th>Query Accuracy</th>
<th>Precision &amp; Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1</td>
<td>List page that displays the title, company, location, country, job type, salary of all jobs</td>
<td>QT1</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>J2</td>
<td>Detailed page that displays the title, company, location, country, job type, salary, description and a list of qualifications of job with title &quot;Financial Analyst&quot;</td>
<td>QT2</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>J3</td>
<td>Detailed page that displays the title, company, location, country, job type, salary, description and a list of qualifications of job with title &quot;Accounting Manager&quot;</td>
<td>QT3</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>J4</td>
<td>List page that displays the title, company, location, country, job type, salary of all jobs with job type Full Time</td>
<td>QT3</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>J5</td>
<td>List the name, location and country of all companies</td>
<td>QT4</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
</tbody>
</table>

Table 5.5: Job search platform project evaluation results

The results displayed in Table 5.5 display a high accuracy value for all mockup types, meaning that the result set generated by the inferred query correspond to the result set displayed in the IM. Besides the quality of the generated queries, the complete list of the inferred SQL queries code shown in Table C.5 (Appendix C), demonstrates that the QI component is able to detect queries both simple queries such as the query for J5 and more complex ones that that detect ordered results by one or more columns such as the query for the J4 mockup.

5.1.3 Results of the test set of projects

The test set of projects was used to run the heuristics and evaluate their performance. All of them have a systematic structure and consist of mockups generated with list, detail and search query. In the next paragraphs, we will describe the projects used to test the QI component and show the results for the two measurements described in Section 5.1.1. Similarly, this
section presents only a description of the mockup types, inferred queries and the measurement results, the complete list of the inferred queries SQL code for each project can be found in Appendix C.

**Bookstore**

The bookstore example project is the first project assessed in the second stage of the QI component performance. Figure 5.12 shows two example mockups designed in this project. The detailed page on the left-hand side displays data about a particular movie such as picture, title, author, description, genre, publication date, and publisher name. In the list page, we can see a list of all books with reduced amount of information such as picture, title and author.

The bookstore project consists of several mockups listed in Table 5.6 and the inferred schema from the listed mockups, is shown in Figure 5.11.

![Figure 5.11: Bookstore database schema](image)
The inferred query types are defined as follows:

- **QT1**: Viewing only selected columns of all books stored in database
• **QT2**: Viewing all columns of a particular book  
• **QT3**: View only selected columns of the books that matches a specific condition

<table>
<thead>
<tr>
<th>MID</th>
<th>Mockup Type</th>
<th>Query Type</th>
<th>Query Accuracy</th>
<th>Precision &amp; Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>List page that displays the picture, title and author of all books</td>
<td>QT1</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>B2</td>
<td>Detailed page that displays picture, title, author, description, genre, publication date and publisher name of movie with title &quot;The Hike&quot;</td>
<td>QT2</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>B3</td>
<td>Detailed page that displays picture, title, author, description, genre, publication date and publisher name of movie with title &quot;Wish&quot;</td>
<td>QT2</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>B4</td>
<td>List page that displays the picture, title and author of books written by &quot;Anne Korkeakivi&quot;</td>
<td>QT3</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
</tbody>
</table>

Table 5.6: Bookstore project evaluation results

The correlation between results and the query accuracy shown in Table 5.6 provide confidence on the performance and quality of the generated queries from the QI component after the training stage. The query accuracy of all queries is 100% and the precision & recall values are 1, meaning that the queries present all the data and only the data displayed in the IM. The complete list of the inferred SQL queries be found in Table C.6 (Appendix C).

**Boat Booking**
As we can see from the screenshots displayed in Figure 5.13, the boat booking mockup design is slightly different and more sophisticated than the design of the previously described projects. The list page mockup displays a list of boats with pictures and price/day. On picture hover the users can see more detailed information including the boat name, location and number of berths. The detailed page besides the information shown on the list page has additional data such as number of cabins and the year when it was produced.
The boat booking project consists of several mockups listed in Table 5.7 and the inferred schema from the listed mockups, is shown in Figure 5.14.
5.1. EVALUATION IN TERMS OF PERFORMANCE AND QUALITY OF GENERATED QUERIES

Figure 5.14: Boat booking database schema

The types of inferred query are defined as follows:

- **QT1**: Viewing only selected columns of all boats stored in database
- **QT2**: Viewing all columns of a particular boat
- **QT3**: View only selected columns of the boats that matches a specific condition

<table>
<thead>
<tr>
<th>MID</th>
<th>Mockup Type</th>
<th>Query Type</th>
<th>Query Accuracy</th>
<th>Precision &amp; Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB1</td>
<td>List page that displays the picture, price/day, boat name, location and number of births of all boats</td>
<td>QT1</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>BB2</td>
<td>Detail page that displays the picture, price/day, boat name, location and number of births, number of cabins and production year of boat with name &quot;Delphia 47&quot;</td>
<td>QT2</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>BB3</td>
<td>Detail page that displays the picture, price/day, boat name, location and number of births, number of cabins and production year of boat with name &quot;Cruiser 51&quot;</td>
<td>QT2</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
</tbody>
</table>
Table 5.7: Boat booking project evaluation results

The results shown in Table 5.7 indicate that DataMockups can infer the schema and the QI component is able to generate queries of high quality that generate exactly the data displayed in the IM. Again, the correlation between two measurements is high and all queries generate the same result set displayed on the listed mockups. The complete list of the inferred SQL queries be found in Table C.7 (Appendix C).

**Notebook Shop**

The notebook shop project is another example of online shop websites that contains data from a completely different domain that the online clothes shop project presented before. While the purpose of the website is the same the domain, amount of data and design are completely different. Figure 5.16 shows two example mockups of detailed and list page.

The notebook project consists of several mockups listed in Table 5.8 and the inferred schema from the listed mockups, is shown in Figure 5.15
5.1. EVALUATION IN TERMS OF PERFORMANCE AND QUALITY OF GENERATED QUERIES

The inferred query types are defined as follows:

- **QT1**: Viewing only selected columns of all notebooks stored in database
- **QT2**: Viewing all columns of a particular notebook
- **QT3**: View only selected columns of the notebooks that matches a specific condition
- **QT4**: View only selected columns of the notebooks that matches a range condition and the results are sorted

<table>
<thead>
<tr>
<th>MID</th>
<th>Mockup Type</th>
<th>Query Type</th>
<th>Query Accuracy</th>
<th>Precision &amp; Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>List page that displays the title, image, manufacturer and price of all notebooks</td>
<td>QT1</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>N2</td>
<td>Detail page that displays the title, image, manufacturer, price, description and other short features for notebook with title &quot;Dell Inspiron 2-in-1 17.3 Touch-Screen&quot;</td>
<td>QT2</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>N3</td>
<td>Detail page that displays the title, image, manufacturer, price, description and other short features for notebook with title &quot;Dell Inspiron 15.6&quot;</td>
<td>QT2</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>N4</td>
<td>List page that displays the title, image, manufacturer and price of all notebooks from &quot;DELL&quot; manufacturer</td>
<td>QT3</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>N5</td>
<td>List page that displays the title, image, manufacturer and price of all notebooks with price between $225.5 and $387.5 sorted in descending order</td>
<td>QT4</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
</tbody>
</table>

Table 5.8: Notebook shop project evaluation results

Table 5.8 shows the that results for each of the notebook project mockups. An interesting result is the inferred query for the mockup N5, which contains items with price within a specific range. Moreover, it detects the order of the result set. The ability of detecting queries that generate sorted result sets is very important for online shop website development, because most of them provide features for showing items within a price range or sorted by price in both ascending or descending order. The quality of the inferred queries is proved by the results of both metrics, meaning that the queries generate the same data displayed in the IM. The complete list of the inferred SQL queries be found in Table C.8 (Appendix C).

**Apartament Rental**

In Figure 5.17 we can see two mockups that belong to the apartment rental project. They have a different design and domain from the mockups presented in the other projects. The mockup on the left-hand side shows a detailed page that displays various information about...
a particular apartment such as title, picture, price, canton, address, number of rooms, living space, and additional information. In contrast the list page displays less information that includes the title, price, picture, canton, number of rooms, and living space.

Figure 5.17: Apartment rental project example mockups
The apartment rental project consists of several mockups listed in Table 5.9 and the inferred schema from the listed mockups, is shown in Figure 5.18.

![Figure 5.18: Apartment Rental database schema](image)

The inferred query types are defined as follows:

- **QT1**: Viewing only selected columns of all apartments stored in database
- **QT2**: Viewing all columns of a particular apartment
- **QT3**: View only selected columns of the apartments that matches a specific condition and results are sorted
- **QT4**: View only selected columns of the apartments that matches a range condition

<table>
<thead>
<tr>
<th>MID</th>
<th>Mockup Type</th>
<th>Query Type</th>
<th>Query Accuracy</th>
<th>Precision &amp; Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>List page that displays the title, image, price, location, number of rooms and living space of all apartments</td>
<td>QT1</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>A2</td>
<td>Detail page that displays the title, image, price, location, number of rooms and living space, address and additional information for apartment with title &quot;Sehr grosse Wohnung&quot;</td>
<td>QT2</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>A3</td>
<td>Detail page that displays the title, image, price, location, number of rooms and living space, address and additional information for apartment with title &quot;Sky Lights Schoren&quot;</td>
<td>QT2</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
</tbody>
</table>
5.1. EVALUATION IN TERMS OF PERFORMANCE AND QUALITY OF GENERATED QUERIES

<table>
<thead>
<tr>
<th></th>
<th>List page that displays the title, image, price, location, number of rooms and living space of all apartments located in canton “Zurich”</th>
<th>QT3</th>
<th>100 %</th>
<th>P=1, R=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A4</td>
<td>List page that displays the title, image, price, location, number of rooms and living space of all apartments with price between CHF1380 and CHF1770</td>
<td>QT4</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
</tbody>
</table>

Table 5.9: Apartment rental shop project evaluation results

The data displayed in the mockups listed in Table 5.9 contain a lot of numerical fields and the QI component is able to detect queries that compare numerical values using conditions with ≤, =, ≥ and BETWEEN operators. The complete list of the inferred SQL queries be found in Table C.7 (Appendix C). For simplification purposes we only show one query per mockup. However, in the inferred query for the A5 mockup the WHERE clause can also include the conditions property.price <= 1770 and property.size BETWEEN 60 AND 64 which generate the same result set. Of course, this may not be the case with websites where data supply is large, but the results demonstrate that the QI component can infer queries with different conditions. The correlation of results and the query accuracy again confirm the quality of the generated queries.

**FoodCASE**

FoodCASE\(^1\) (Food Composition And System Environment) is a research project developed at ETH Zurich as well as a food science data management system. The FoodCASE database has a schema of high complexity, because it includes various foods from different participating countries. Its records cover the nutrient and contaminant content of foods of different categories. Due to complexity of the schema and time constraints we could not experiment with different types of queries. However, we were could infer a correct sub schema of the FoodCASEDB, based on a smaller subset of data. A sample mockup corresponding to FoodCASE data is displayed in Figure 5.19.

\(^1\)http://www.foodcase.ethz.ch/index_EN
The FoodCASE project consists of several mockups listed in Table 5.10 and the inferred schema from the listed mockups, is shown in Figure 5.20. As we can see from the picture even in this simplified version the FoodCASE database schema is more complex than most of the other inferred schemas in the 'test' set of projects. Each food belongs to a category that has multiple subcategories, therefore each food belongs to one subcategory. In the actual schema, each food belongs to more than one category but the dataset used for the purposes of our thesis is smaller and each food belongs to only one category and subcategory. Foods consists of multiple components such as energy-kilojoules and energy-kilocalories used in our project and each component has a component value. Each component value has a unit such as $kJ$ and $kcal$ used in our project. Indeed, the schema consists of the following relationships:

- N foods - 1 subcategory
- N subcategory - 1 category
- N component values - 1 component
- N component values - 1 unit
- N foods - N component values

For simplification purposes, we do not show the complete FoodCASE schema, but we can see from the inferred schema in Figure 5.20, that the relationships are correctly detected.
5.1. EVALUATION IN TERMS OF PERFORMANCE AND QUALITY OF GENERATED QUERIES

The inferred query types are defined as follows:

- **QT1**: Viewing all information about all foods stored in database
- **QT2**: Viewing all information about all foods of a specific category
- **QT3**: Viewing all information about all foods of a specific subcategory
- **QT4**: Viewing all information about all foods with energy kilocalories values between a certain range.

<table>
<thead>
<tr>
<th>MID</th>
<th>Mockup Type</th>
<th>Query Type</th>
<th>Query Accuracy</th>
<th>Precision &amp; Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>List page that displays the name, category, subcategory and energy values in kilojoules and kilocalories for all foods</td>
<td>QT1</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>F2</td>
<td>List page that displays the name, category, subcategory and energy values in kilojoules and kilocalories for all foods with category &quot;Vegetables&quot;</td>
<td>QT2</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
<tr>
<td>F3</td>
<td>List page that displays the name, subcategory, category and energy values in kilojoules and kilocalories for all foods with subcategory &quot;Cooked Vegetables&quot;</td>
<td>QT3</td>
<td>100 %</td>
<td>P=1, R=1</td>
</tr>
</tbody>
</table>
CHAPTER 5. EVALUATION

As we can see from the results displayed in Table 5.10, the QI component is able to generate queries for mockups that display a list of all foods (F1) and a list of foods from a particular category (F2) or subcategory (F3). Furthermore, the query accuracy is correlated with the precision & recall values. The last mockup (F4), shows foods that have the kilocalories value within a specific range. While the query accuracy is 0%, the recall value shows that the inferred query is able to generate all the data displayed in the mockup but does not present only the relevant data proved by the precision value. The reason is that the values for the kilojoules and kilocalories belong to the same cluster and the schema inference component is not able to distinguish them. In this case, the schema inference component can correctly detect the relationships but it does not support relationship attributes. Additionally, the QI component cannot infer queries that use aliases to display the data in the IM. Therefore, the values for the kilojoules and kilocalories are concatenated using the GROUP_CONCAT function. The complete list of the inferred SQL queries be found in Table C.10 (Appendix C). We can see from the number of join conditions that the FoodCASE schema is quite complicated even in this simplified version used for evaluation of the QI component.

### 5.2 Evaluation in terms of efficiency

The goal of this evaluation is to test whether the mockup type and query complexity affects the runtime of the query inference. To compare different types of queries, we measured the total time for inferring the query on mockups listed in the previous sections. Basically, we can distinguish between three types of mockups. Detailed page, list page that displays all the data and list page that displays all the data based on a specific condition. The query inference process was repeated two times for each mockup, and results are aggregated based in the mockup type. The column chart in Figure 5.21 illustrates the average total time for inferring the query for the three mockup types. The chart clearly shows that the runtime increases as the mockup complexity is increased. For a detailed page the runtime is relatively low, while for a list page that displays specific data is about five times higher. The reason for this increase is the number of conditions in the WHERE clause inferred by heuristics presented in Section 4.5.5, which depend on the mockup type and the displayed data. The average number of suggested conditions is 4.85 with minimum and maximum values of 1 and 11 respectively.

| F4 | List page that displays the name, category, subcategory and energy values in kilojoules and kilocalories for all foods with kilokalories value between 25 and 45 | QT4 | 0 % | P=3/7, R=1 |

Table 5.10: FoodCASE project evaluation results
Although this maximum value is, in our opinion, high the average value indicates that in most of the cases the number is lower between 4 and 5. In the list page that displays all the data stored for a particular project the runtime is very low, because the QI component is able to identify the cases when a mockup displays all the data and does not search for a condition. Overall, the total runtime is relatively low and acceptable for the size of the dataset used in this thesis.

5.3 Discussion

In general, the results of both evaluations are very encouraging, as most of the findings indicate that the quality of the generated queries corresponds to the data displayed in the mockups. Moreover, the complexity of the inferred schemas for different types of domains indicates that DataMockups and QI component make a great tool, which can be easily used in developing different website types. Another important feature is the ability to generate queries of high quality independently of the website design. Additionally, the correlation between the query accuracy and precision & recall values, and the high accuracy indicate that in most cases the queries generate all the data and only the data displayed in the mockup. The mockup type and the domain of the data displayed in the mockup can greatly affect the efficiency of the QI algorithm in terms of runtime. However, for finding the query based on a smaller dataset and the schema information encoded in the mockup this runtime is satisfiable. Overall, the evaluations have shown that the QI component has achieved the goals of this thesis.
6 Conclusion

We have presented a Query Inference component which was successfully incorporated in the DataMockups tool, allowing developers that have some basic knowledge of HTML and CSS or little or nothing of databases or queries, to create a working prototype of the website with reduced effort. After the design is created and the database generation is performed, the QI component is able to infer the query that the prototype would need to execute on the database such that it generates the same result displayed in the mockup. Once the queries are inferred the users have the possibility to view the queries in a simplified interface adapted for users without databases knowledge. However, the users can see the inferred query in SQL language, and expert users can assist in the query inference process if the quality of the generated query is unsatisfying. At the end, the users can create a working prototype of the website by augmenting the IM with code that can query the database and generate dynamic output.

The results of the evaluations prove that QI can infer queries of different complexity based on the data displayed in the IM. In our test projects, they generate the same data displayed in the IM except for the mockups which display data from more complicated schemas that include unsupported SQL statements. Although the query inference should not be entirely automated, in almost all cases it does not require user intervention to achieve the expected quality.

Future Work

Even though the Query Inference component has reached the goals of this thesis, there is still room for improvement. The next step of development could provide support for additional types of queries such that the user does not have to modify the query in case it does generate the same data displayed in the mockup. Possible improvements include the support for `HAVING` clauses, `aliases` and even more complex queries that are used to display data
for more complex databases such as FoodCASE. Another improvement could be the use of the LovefieldDB for the query inference process. The initial goal of the schema conversion to LovefieldDB was to have a relational query engine that works cross-browser so that the queries do not have to be inferred using the MySQL database. However, it does not accept raw SQL statements; instead, it has a builder-pattern API to build queries. Converting the raw SQL statements to Lovefield queries would be another significant feature that could improve the algorithm runtime. In terms of runtime, the improvement of heuristics for finding the conditions in the WHERE clause could considerably reduce the query inference runtime allowing to infer queries for larger datasets.
exports.inferWhere = function(from_clause_tables, relationships) {
    var from_tables = Object.keys(from_clause_tables);
    var join_conditions = DataMockups.queryinference.getJoins(relationships, from_tables);
    var all_content_array = DataMockups.queryinference.getContent(DataMockups.dataContent);
    var page_content_array = DataMockups.queryinference.getContent(DataMockups.dataContentPage);
    var selectatt = DataMockups.queryselect.inferSelect(from_clause_tables);
    var is_groupby = selectatt.groupby;
    var element_rank = DataMockups.queryinference.elementRangking(page_content_array);
    var rank = element_rank.rank;
    var higher_rank = element_rank.min;
    var unique_att = element_rank.uniqueatt;
    var where_conditions = [];
    var main_cluster_occ = getMCOcc(all_content_array, page_content_array);
    if (main_cluster_occ)
    {
        where_conditions = [];
    }
    else
    {
        for (var element in page_content_array)
        {
            if (main_cluster_occ)
            {
                // conditions
            }
        }
    }
var isimg = element.includes("img"); //check if the atribute is img
if (!isimg)
{
    if (page_content_array[element].length == 1)
    {
        var table = element.split('\\.')[0];
        var col = element.split('\\.')[1];
        var att = col.includes("att");
        if (att) {
            col = col.split('\\_')[0];
        }
        var attcontent = "'" + page_content_array[element] + "'";
        var c = element + '=' + attcontent;

        if (rank[col] == higher_rank)
        {
            where_conditions.push(c);
        }
    }
    else
    {
        var all_pages_occ = getElementMap(all_content_array[element]);
        var page_occ = getElementMap(page_content_array[element]);
        var size = Object.size(page_occ);
        var allsize = Object.size(all_pages_occ);
        var ratio = size / allsize;
        var isNumber = isNumberArray(page_content_array[element]);
        if (size == 1) //if one key
        {
            var c = element + "='" + Object.keys(page_occ) + "'";
            where_conditions.push(c);
        }
        else
        {
            if (!isNumber)
            {
                if (ratio < 1)
                {
                    var allsimarr = [];
                    var simarr = [];
                    for (var el in page_occ)
                    {
                        simarr.push(page_occ[el]);
                    }
                    for (var el in all_pages_occ)
                    {
                        allsimarr.push(all_pages_occ[el]);
                    }
                    var all_occ_one = allsimarr.every(isone);
                    var page_occ_one = simarr.every(isone);
                    for (var el in page_occ)
                    {
                        //...
                    }
                }
            }
        }
    }
}
if (all_occ_one == false && page_occ_one == false) {
    var c = element + "='" + el + "',";
    where_conditions.push(c);
}
}
}

else {
    var minval = arraymin(page_content_array[element]);
    var maxval = arraymax((page_content_array[element]));
    var greaterthen = element + ">=" + minval;
    var lessthen = element + "<=" + maxval;
    var betwcond = element + " BETWEEN " + minval + " AND " + maxval;
    where_conditions.push(betwcond, greaterthen, lessthen);
}

var like_conditions=getLikeConditions(page_content_array, unique_att);
where_conditions=where_conditions.concat(like_conditions);
var str=
DataMockups.queryinference.check_query(from_tables, selectatt.select_string,
where_conditions, join_conditions, is_groupby, unique_att);
str.orderbysugg=
DataMockups.queryorderby.getOrderBySuggestions(page_content_array);
return str;
}
Example of augmented mockup

In order to explain how the augmentation process is done we have separated the augmented mockup content shown in Listing B.1 in five blocks and each of them represents the following.

1. extracted HTML elements that are located before the parent cluster element

2. PHP code before the parent cluster element that does the following
   - code that includes the dbconfig.php file (line 17)
   - creates the connection to the server and the database (line 18)
   - checks the connection (lines 19-21)
   - set up the inferred query in $sql variable (line 23)
   - runs the query and puts the resulting data into a variable called $result (line 24)
   - the function num_rows() checks whether there are more than zero rows returned (line 25)
   - if there are more than zero rows returned, the function fetch_assoc() puts all the results into an associative array that we can loop through (line 28)
   - the while loop loops through the result set and outputs the data from the select columns (line 28)

3. display the query results in the parent cluster element, i.e. children cluster elements where content is replaced as described in step 1.

4. for the elements that contain grouped result set the replacement is different, the concatenated return value is separated with the PHP explode function and a foreach loop is added for replacing the element content (lines 36-38).

5. command that closes the connection (line 47)
6. closing tags of the HTML page
Listing B.1: Augmented mockup example
List of inferred query SQL statements

<table>
<thead>
<tr>
<th>MID</th>
<th>Inferred SQL query</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>SELECT recipe.title, recipe.title_link, type.type_att, time.time_att FROM recipe, type, time WHERE type.ID = recipe.recipetypeID AND time.ID = recipe.recipetimeID</td>
</tr>
<tr>
<td>R2</td>
<td>SELECT recipe.title, recipe.directions, GROUP_CONCAT(DISTINCT ingredients.ingredients_att SEPARATOR '</td>
</tr>
<tr>
<td>MID</td>
<td>Inferred SQL query</td>
</tr>
<tr>
<td>------</td>
<td>--------------------</td>
</tr>
<tr>
<td>M1</td>
<td>SELECT movies.img_alt, movies.img_src, movies.title, movies.rating, movies.releasedate, movies.overview, GROUP_CONCAT(DISTINCT genre.genre_att SEPARATOR '</td>
</tr>
</tbody>
</table>

Table C.1: Recipe project inferred queries
<table>
<thead>
<tr>
<th></th>
<th>SQL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>M4</td>
<td>SELECT GROUP_CONCAT(DISTINCT movies.title SEPARATOR '</td>
</tr>
<tr>
<td>M5</td>
<td>SELECT movies.img_alt, movies.img_src, movies.title, movies.rating, movies.releasedate, movies.overview, GROUP_CONCAT(DISTINCT genre.genre_att SEPARATOR '</td>
</tr>
</tbody>
</table>
### Table C.2: Movies project inferred queries

<table>
<thead>
<tr>
<th>MID</th>
<th>Inferred SQL query</th>
</tr>
</thead>
<tbody>
<tr>
<td>M6</td>
<td>SELECT movies.img_alt, movies.img_src, movies.title, movies.rating, movies.releasedate, movies.overview, GROUP_CONCAT(DISTINCT genre.genre_att SEPARATOR '</td>
</tr>
<tr>
<td>M7</td>
<td>SELECT DISTINCT actors.actors_att FROM actors</td>
</tr>
<tr>
<td>BP1</td>
<td>SELECT posts.title, posts.content, posts.date, author.author_att, GROUP_CONCAT(DISTINCT category.category_att SEPARATOR '</td>
</tr>
<tr>
<td>BP2</td>
<td>SELECT posts.title, posts.content, posts.date, GROUP_CONCAT(DISTINCT author.author_att SEPARATOR '</td>
</tr>
<tr>
<td>BP3</td>
<td>SELECT posts.title, posts.content, posts.date, GROUP_CONCAT(DISTINCT author.author_att SEPARATOR '</td>
</tr>
</tbody>
</table>
Table C.3: Blog project inferred queries

<table>
<thead>
<tr>
<th>MID</th>
<th>Inferred SQL query</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>SELECT products.title, products.image_alt, products.image_src, products.price  FROM products</td>
</tr>
<tr>
<td>MID</td>
<td>Inferred SQL query</td>
</tr>
<tr>
<td>-----</td>
<td>--------------------</td>
</tr>
<tr>
<td>C3</td>
<td>SELECT products.title, products.image_alt, products.image_src, products.price, products.category, products.availability, products.color, products.description, size.size_att FROM products, size WHERE size.ID = products.productssizesizeID AND products.title = 'Mom Jeans'</td>
</tr>
<tr>
<td>C4</td>
<td>SELECT products.title, products.image_alt, products.image_src, products.price FROM products WHERE products.title LIKE '%Dress%'</td>
</tr>
<tr>
<td>C5</td>
<td>SELECT products.title, products.image_alt, products.image_src, products.price FROM products WHERE products.price BETWEEN 17.9 AND 21.9</td>
</tr>
</tbody>
</table>

Table C.4: Online clothes shop project inferred queries
| J3 | SELECT jobs.title, jobs.salary, jobs.description, company.name, company.location, country.country_att, jobtype.jobtype_att, GROUP_CONCAT(DISTINCT qualifications.qualifications_att SEPARATOR '|') AS qualifications_att FROM jobs, company, country, jobtype, qualifications, jobscompany WHERE country.ID = company.companycountrycountryID AND jobtype.ID = jobs.jobsjobtypejobtypeID AND jobs.ID = qualifications.jobsqualificationsjobsID AND jobscompany.jobsID = jobs.ID AND jobscompany.companyID = company.ID AND jobs.title = 'Accounting Manager' |
| J4 | SELECT jobs.title, jobs.salary, company.name, company.location, country.country_att, jobtype.jobtype_att FROM jobs, company, country, jobtype, jobscompany WHERE country.ID = company.companycountrycountryID AND jobtype.ID = jobs.jobsjobtypejobtypeID AND jobscompany.jobsID = jobs.ID AND jobscompany.companyID = company.ID AND jobtype.jobtype_att = 'Full Time' ORDER BY jobs.salary ASC |
| J5 | SELECT company.name, company.location, country.country_att FROM company, country WHERE country.ID = company.companycountrycountryID |
Table C.5: Job search platform project inferred queries

<table>
<thead>
<tr>
<th>MID</th>
<th>Inferred SQL query</th>
</tr>
</thead>
</table>
| B1  | List page that displays the picture, title and author of all books
SELECT books.img_alt, books.img_src, books.title, author.author_att
FROM books, author
WHERE author.ID = books.booksauthorauthorID |
| B2  | SELECT books.img_alt, books.img_src, books.title, books.description, books.genre, books.publdate, books.publisher, author.author_att
FROM books, author
WHERE author.ID = books.booksauthorauthorID
AND books.title = 'The Hike' |
| B3  | SELECT books.img_alt, books.img_src, books.title, books.description, books.genre, books.publdate, books.publisher, author.author_att
FROM books, author
WHERE author.ID = books.booksauthorauthorID
AND books.title = 'Wish' |
| B4  | SELECT books.img_alt, books.img_src, books.title, author.author_att
FROM books, author
WHERE author.ID = books.booksauthorauthorID
AND author.author_att = 'Anne Korkeakivi' |

Table C.6: Bookstore project inferred queries
<table>
<thead>
<tr>
<th>MID</th>
<th>Inferred SQL query</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB1</td>
<td>SELECT boats.img_alt,boats.img_src,boats.price, boats.name,location.location_att,berths.berths_att FROM boats,location,berths WHERE location.ID=boats.boatslocationlocationID AND berths.ID=boats.boatsberthsberthsID</td>
</tr>
<tr>
<td>BB2</td>
<td>SELECT boats.img_alt,boats.img_src,boats.price, boats.name,boats.cabins,boats.year, location.location_att,berths.berths_att FROM boats,location,berths WHERE location.ID=boats.boatslocationlocationID AND berths.ID=boats.boatsberthsberthsID AND boats.name='Delphia 47'</td>
</tr>
<tr>
<td>BB3</td>
<td>SELECT boats.img_alt,boats.img_src,boats.price, boats.name,boats.cabins,boats.year, location.location_att,berths.berths_att FROM boats,location,berths WHERE location.ID=boats.boatslocationlocationID AND berths.ID=boats.boatsberthsberthsID AND boats.name='Cruiser 51'</td>
</tr>
<tr>
<td>BB4</td>
<td>SELECT boats.img_alt,boats.img_src,boats.price, boats.name,location.location_att,berths.berths_att FROM boats,location,berths WHERE location.ID=boats.boatslocationlocationID AND berths.ID=boats.boatsberthsberthsID AND location.location_att='Palma de Mallorca, Spain'</td>
</tr>
</tbody>
</table>
### APPENDIX C. LIST OF INFERRED QUERY SQL STATEMENTS

Table C.7: Boat booking project inferred queries

<table>
<thead>
<tr>
<th>MID</th>
<th>Inferred SQL query</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BB5</strong></td>
<td>SELECT boats.img_alt,boats.img_src,boats.price, boats.name,location.location_att,berths.berths_att FROM boats,location,berths WHERE location.ID=boats.boatslocationlocationID AND berths.ID=boats.boatsberthsberthsID AND berths.berths_att='8'</td>
</tr>
<tr>
<td><strong>N1</strong></td>
<td>SELECT products.title,products.img_alt,products.img_src, products.price,manufacturer.manufacturer_att FROM products,manufacturer WHERE manufacturer.ID=products.productsmanufacturermanufacturerID</td>
</tr>
<tr>
<td><strong>N2</strong></td>
<td>SELECT products.title,products.img_alt,products.img_src, products.price,products.description,products.memory, products.color,products.model,manufacturer.manufacturer_att, operatingsys.operatingsys_att,harddrive.harddrive_att FROM products,manufacturer,operatingsys,harddrive WHERE manufacturer.ID=products.productsmanufacturermanufacturerID AND operatingsys.ID=products.productsoperatingsysoperatingsysID AND harddrive.ID=products.productsharddriveharddriveID AND products.title='Dell Inspiron 2-in-1 17.3 Touch-Screen'</td>
</tr>
</tbody>
</table>
Table C.8: Notebook shop project inferred queries

<table>
<thead>
<tr>
<th>MID</th>
<th>Inferred SQL query</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>SELECT property.img_alt,property.img_src,property.price, property.title,property.size,canton.canton_att,rooms.rooms_att FROM property,canton,rooms WHERE canton.ID=property.propertycantoncantonID AND rooms.ID=property.propertyroomsroomsID</td>
</tr>
<tr>
<td>A2</td>
<td>SELECT property.img_alt,property.img_src,property.price, property.title,property.size,property.address, property.addinfo,canton.canton_att,rooms.rooms_att FROM property,canton,rooms WHERE canton.ID=property.propertycantoncantonID AND rooms.ID=property.propertyroomsroomsID AND property.title='Sehr grosse Wohnung'</td>
</tr>
</tbody>
</table>
### Table C.9: Apartment rental shop project inferred queries

<table>
<thead>
<tr>
<th></th>
<th>SQL Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3</td>
<td>SELECT property.img_alt, property.img_src, property.price, property.title, property.size, property.address, property.addinfo, canton.canton_att, rooms.rooms_att FROM property, canton, rooms WHERE canton.ID = property.propertycantoncantonID AND rooms.ID = property.propertyroomsroomsID AND property.title = 'Sky Lights Schoren'</td>
</tr>
<tr>
<td>A4</td>
<td>SELECT property.img_alt, property.img_src, property.price, property.title, property.size, canton.canton_att, rooms.rooms_att FROM property, canton, rooms WHERE canton.ID = property.propertycantoncantonID AND rooms.ID = property.propertyroomsroomsID AND canton.canton_att = 'Zurich' ORDER BY property.price ASC</td>
</tr>
<tr>
<td>A5</td>
<td>SELECT property.img_alt, property.img_src, property.price, property.title, property.size, canton.canton_att, rooms.rooms_att FROM property, canton, rooms WHERE canton.ID = property.propertycantoncantonID AND rooms.ID = property.propertyroomsroomsID AND property.price BETWEEN 1380 AND 1770</td>
</tr>
<tr>
<td>MID</td>
<td>Inferred SQL query</td>
</tr>
<tr>
<td>-----</td>
<td>--------------------</td>
</tr>
</tbody>
</table>
| F1  | SELECT foods.name, category.subcategory,  
      maincategory.maincategory_att,  
      GROUP_CONCAT(DISTINCT compvalues.compvalue SEPARATOR '|')  
      AS compvalue,  
      GROUP_CONCAT(DISTINCT component.component_att SEPARATOR '|')  
      AS component_att,  
      GROUP_CONCAT(DISTINCT unit.unit_att SEPARATOR '|')  
      AS unit_att  
      FROM foods, category, maincategory, compvalues,  
      component, unit, foodscompvalues  
      WHERE category.ID=foods.foodscategorycategoryID  
      AND maincategory.ID=category.categorymaincategorymaincategoryID  
      AND component.ID=compvalues.compvaluescomponentcomponentID  
      AND unit.ID=compvalues.unitcompvaluesunitID  
      AND foodscompvalues.foodsID=foods.ID  
      AND foodscompvalues.compvaluesID=compvalues.ID  
      GROUP BY foods.name |
| F2  | SELECT foods.name, category.subcategory,  
      maincategory.maincategory_att,  
      GROUP_CONCAT(DISTINCT compvalues.compvalue SEPARATOR '|')  
      AS compvalue,  
      GROUP_CONCAT(DISTINCT component.component_att SEPARATOR '|')  
      AS component_att,  
      GROUP_CONCAT(DISTINCT unit.unit_att SEPARATOR '|')  
      AS unit_att  
      FROM foods, category, maincategory, compvalues,  
      component, unit, foodscompvalues  
      WHERE category.ID=foods.foodscategorycategoryID  
      AND maincategory.ID=category.categorymaincategorymaincategoryID  
      AND component.ID=compvalues.compvaluescomponentcomponentID  
      AND unit.ID=compvalues.unitcompvaluesunitID  
      AND foodscompvalues.foodsID=foods.ID  
      AND foodscompvalues.compvaluesID=compvalues.ID  
      AND maincategory.maincategory_att='Vegetables'  
      GROUP BY foods.name |
APPENDIX C. LIST OF INFERRED QUERY SQL STATEMENTS

Table C.10: FoodCASE project inferred queries

<table>
<thead>
<tr>
<th>Query</th>
<th>SQL</th>
</tr>
</thead>
</table>
| F3    | ```
SELECT foods.name, category.subcategory, maincategory.maincategory_att,
GROUP_CONCAT(DISTINCT compvalues.compvalue SEPARATOR '||')
AS compvalue,
GROUP_CONCAT(DISTINCT component.component_att SEPARATOR '||')
AS component_att,
GROUP_CONCAT(DISTINCT unit.unit_att SEPARATOR '||')
AS unit_att
FROM foods, category, maincategory, compvalues, component, unit, foodscompvalues
WHERE category.ID=foods.foodscategorycategoryID
AND maincategory.ID=category.categorymaincategorymaincategoryID
AND component.ID=compvalues.compvaluescomponentcomponentID
AND unit.ID=compvalues.unitcompvaluesunitID
AND foodscompvalues.foodsID=foods.ID
AND foodscompvalues.compvaluesID=compvalues.ID
AND category.subcategory='Cooked vegetables'
GROUP BY foods.name ``` |
| F4    | ```
SELECT foods.name, category.subcategory, maincategory.maincategory_att,
GROUP_CONCAT(DISTINCT compvalues.compvalue SEPARATOR '||')
AS compvalue,
GROUP_CONCAT(DISTINCT component.component_att SEPARATOR '||')
AS component_att,
GROUP_CONCAT(DISTINCT unit.unit_att SEPARATOR '||')
AS unit_att
FROM foods, category, maincategory, compvalues, component, unit, foodscompvalues
WHERE category.ID=foods.foodscategorycategoryID
AND maincategory.ID=category.categorymaincategorymaincategoryID
AND component.ID=compvalues.compvaluescomponentcomponentID
AND unit.ID=compvalues.unitcompvaluesunitID
AND foodscompvalues.foodsID=foods.ID
AND foodscompvalues.compvaluesID=compvalues.ID
AND compvalues.compvalue BETWEEN 25 AND 194
GROUP BY foods.name ``` |
## List of Figures

1.1 Query Inference for DataMockups .............................................. 2

3.1 DataMockups design tool ......................................................... 12
3.2 Clusters detected by DataMockups ........................................... 13
3.3 Complete schema and detected data .......................................... 14
3.4 Database information ......................................................... 15
3.5 Query Inference component ................................................... 16
3.6 Relational Database Schema .................................................. 17
3.7 Inferred query for a mockup that displays the details of the movie with title "Avengers: Age of Ultron" ................................................. 19
3.8 Inferred query for a mockup which displays a list of movies .......... 20
3.9 Query Results ................................................................. 21
3.10 Inferred SQL query for a mockup which displays a list of movies ... 23
3.11 Inferred SQL query for a mockup which displays a list of products by a particular manufacturer ........................................ 24
3.12 Mockup Augmentation outcome .............................................. 25

4.1 System Architecture .......................................................... 28
4.2 Mockups displaying data from movies project ............................. 35
4.3 Mockup displaying movie titles from a particular actor ................ 36
4.4 Steps and HTTP post requests between the client and the server ..... 46

5.1 Recipe project example mockups ............................................. 52
5.2 Recipe database schema ....................................................... 53
5.3 Movies database schema ...................................................... 54
5.4 Movies project example mockups ............................................ 55
5.5 Blog project example mockups .............................................. 58
5.6 Blog database schema ....................................................... 59
5.7  Online clothes shop project example mockups .......................... 61
5.8  Online clothes shop database schema ................................. 62
5.9  Job search database schema .............................................. 63
5.10 Job search platform project example mockups ....................... 64
5.11 Bookstore database schema ............................................. 66
5.12 Bookstore project example mockups .................................. 67
5.13 Boat booking project example mockups ............................... 69
5.14 Boat booking database schema ......................................... 70
5.15 Notebook database schema ............................................. 71
5.16 Notebook shop project example mockups ............................ 72
5.17 Apartment rental project example mockups ......................... 74
5.18 Apartment Rental database schema .................................. 75
5.19 FoodCASE example mockup ............................................. 77
5.20 FoodCASE database schema ........................................... 78
5.21 Query Inference algorithm runtime ................................. 80
List of Tables

5.1 Recipe Project evaluation results ........................................... 54
5.2 Movies project evaluation results ........................................... 56
5.3 Blog project evaluation results ............................................. 60
5.4 Online clothes shop project evaluation results ......................... 63
5.5 Job search platform project evaluation results ......................... 65
5.6 Bookstore project evaluation results ..................................... 68
5.7 Boat booking project evaluation results ................................ 71
5.8 Notebook shop project evaluation results ............................. 73
5.9 Apartment rental shop project evaluation results .................... 76
5.10 FoodCASE project evaluation results .................................. 79

C.1 Recipe project inferred queries ............................................ 92
C.2 Movies project inferred queries .......................................... 94
C.3 Blog project inferred queries ............................................. 96
C.4 Online clothes shop project inferred queries ......................... 97
C.5 Job search platform project inferred queries ......................... 99
C.6 Bookstore project inferred queries .................................... 99
C.7 Boat booking project inferred queries ................................ 101
C.8 Notebook shop project inferred queries ............................. 102
C.9 Apartment rental shop project inferred queries .................... 103
C.10 FoodCASE project inferred queries .................................. 105
List of Listings

4.1 Schema conversion to LovefieldDB ........................................ 30
4.2 Functions that create the objects used for the query inference process ....... 30
4.3 The inferQuery function ...................................................... 31
4.4 The structure of the dataContentPage object .............................. 32
4.5 The getRelationships function .............................................. 33
4.6 The inferFrom function ....................................................... 33
4.7 The compareColumns function .............................................. 37
4.8 The getSelectSameValues function ....................................... 37
4.9 The getSelectNotSameValues function ................................... 38
4.10 The JSON result returned from the database .............................. 42
4.11 The ViewQueryResults function .......................................... 43
4.12 Markup insertion in the HTML document .................................. 46
4.13 The dbconfig.php file content ............................................. 48
5.1 The inferred query for the M5 mockup ..................................... 56
B.1 Augmented mockup example .................................................. 88
Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor Alfonso Murolo for his continuous support and engagement during past six months. His guidance, patience and useful comments have been incredibly helpful for the success of this thesis. Furthermore, I would like to express my appreciation to Prof. Dr. Moira C. Norrie and the GlobiS group for giving me the opportunity to work on my master thesis in such an interesting topic. Special big thanks goes to my friends Edona and Romy for reading this whole thesis and for their valuable advices on my thesis work. I would like express my very profound gratitude to my family and friends, here and abroad. I am forever indebted to my aunt and her husband for their unfailing support and for being my second parents through my years of study and through the process of working on my thesis. Of course, this acknowledgement would not be complete without thanking my parents and my grandfather for their continuous support and encouragement. Last but not least, I am infinitely grateful to my boyfriend, Besart, for giving me unconditional support and love through all this long process.


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

```
Query Inference component for DataMockups
```

**Authored by** (in block letters):

For papers written by groups the names of all authors are required.

<table>
<thead>
<tr>
<th>Name(s):</th>
<th>First name(s):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asani</td>
<td>Zera</td>
</tr>
</tbody>
</table>

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

**Place, date**

Zurich, 02.11.2016

**Signature(s)**

```
Zera  Asani
```

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.