Independent degree project – second cycle

*Master's thesis AV, 30 higher credits*

**Master of Science in Computer Engineering**

**Relational database web application**
Web administration interface for visualizing and predicting relationships to manage relational databases

**Andreas Hansson**
MID SWEDEN UNIVERSITY
The Department of Information and Communication Systems (IKS)

Examiner: Prof. Mikael Gidlund, <mikael.gidlund@miun.se>
Supervisors: Jimmy Åhlander, <jimmy.ahlander@miun.se>
Johan Deckmar, <johan.deckmar@dewire.com>
Author: Andreas Hansson, <anha1045@student.miun.se>
Degree Programme: Master of Science in Computer Engineering, 300 credits
Semester, Year: Spring, 2015
Relational database web application
- Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

Abstract

There is a need of storing and keeping track of things. As the amount of information increases, so does the demand for suitable applications that can manage the data. This thesis has had its focus on developing a web administration interface for relational databases, where the focus has been on managing and visualizing the data, where relationships between data within the database could be predicted through an algorithm. During the thesis, it was revealed that administrators can utilize naming conventions for databases, a property which can be used to predict its relationships. Furthermore, existing applications for managing databases has been compared with the thesis’ implementation. Notable differences are that existing solutions are focused towards the structure of the data, rather than the data itself. To accomplish all this, an agile method was chosen for fast results within the deadline, together with standardized web development tools and JavaScript frameworks. The resulting implementation consists of a front- and backend. The frontend was developed using the Ember.JS framework for making web applications and the backend was implemented using Node.JS, together with a component for handling different database dialects called Sequelize. It has been concluded that the prototype this thesis has resulted in works as a proof of concept, complete with a prediction algorithm that can suggest relationships within databases that utilizes convenient and consistent naming conventions. In the future, further research and tests could be conducted to evaluate the security, reliability and usability of the application, to ensure its production quality.

Keywords: Relational databases, single page application, web, predicting relationships, Ember.JS, Node.JS, Sequelize
Acknowledgements

I would like to thank and express my gratitude to those who have helped me complete and realize this thesis, a sincerest thanks goes to...

My supervisor at Mid Sweden University, Jimmy Åhlander, for always being accessible and for his reviews of the report. His help towards improving this thesis report has been invaluable.

My supervisor at Dewire, Johan Deckmar, who has provided me with the idea for this thesis, together with a background and a description of the problem. He has also continuously reviewed my results and has provided me with helpful ideas and comments on how to improve the report and the implementation of the application.

My family and friends, for their love and support.
Table of contents

Terminology.................................................................................................................. vi
Acronyms and abbreviations ......................................................................................... vi

1 Introduction ..................................................................................................................... 1
1.1 Background and problem motivation ........................................................................ 1
1.2 Overall aim .................................................................................................................. 2
1.3 Scope .......................................................................................................................... 3
1.4 Concrete and verifiable goals ..................................................................................... 4
1.5 Ethical aspects and concerns ....................................................................................... 4
1.6 Outline ........................................................................................................................ 5
1.7 Contribution ............................................................................................................... 6

2 Databases ....................................................................................................................... 8
2.1 Introduction to databases ............................................................................................ 8
2.2 Database management systems .................................................................................. 9
2.3 Concepts of relational databases ............................................................................... 9
2.4 Naming conventions for relationships ...................................................................... 11
2.5 Managing databases .................................................................................................. 11
2.6 Information Schema .................................................................................................. 12

3 Developing web applications ....................................................................................... 13
3.1 Structure and programming languages ...................................................................... 13
3.2 Web applications for multiple devices ...................................................................... 14
3.3 Single page applications ............................................................................................ 14
3.4 Ember.JS .................................................................................................................... 15

4 Existing applications .................................................................................................... 18
4.1 PhpMyAdmin .............................................................................................................. 18
4.2 MySQL Workbench .................................................................................................... 20

5 Method .......................................................................................................................... 22
5.1 Development methods and work process .................................................................. 22
5.2 Frontend development .............................................................................................. 22
5.2.1 Base tools ............................................................................................................ 22
5.2.2 Ember.JS ............................................................................................................. 22
5.3 Backend development ............................................................................................. 23
5.3.1 Node.JS ................................................................................................................. 23
Relational database web application - Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

Table of contents

2015-07-10

5.3.2 Restify 23
5.3.3 Sequelize 23
5.3.4 Sequelize-Auto 23
5.4 Development tools 24
5.4.1 Layout and design 24
5.4.2 Programming environment 24
5.5 Measurements 24
5.5.1 Predicting relationships 24
5.5.2 Prediction evaluation 25
5.5.3 Prediction algorithm performance evaluation 26
5.5.4 Tested databases 26

6 Implementation 29
6.1 System architecture 29
6.2 Frontend 30
6.3 Backend 33
6.4 Algorithm for determining relationships 36

7 Results 37
7.1 Employees database results 37
7.2 Sakila database results 40
7.3 World database results 44

8 Conclusions 48
8.1 Evaluations 48
8.1.1 Theory 48
8.1.2 Methods and development 49
8.1.3 Implementation 49
8.1.4 Results of the prediction algorithm 50
8.2 Future work 51

References 52

Appendix A: Screenshots of the implementation 57
Relational database web application
- Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

---

**Terminology**

**Acronyms and abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ad</td>
<td>Advertisement</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CRUD</td>
<td>Create, read, update, delete</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>DML</td>
<td>Data Manipulation Language</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>JS</td>
<td>JavaScript</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>ORM</td>
<td>Object Relational Mapper</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>SAD</td>
<td>System Architecture Document</td>
</tr>
<tr>
<td>SPA</td>
<td>Single Page Application</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
</tbody>
</table>
1 Introduction

The introduction chapter describes the background for this thesis, the purpose of it, what it aims to result in and the problems it aims to solve.

1.1 Background and problem motivation

Due to the limitations of our memory and the need for organization within stored information, databases became important in terms of how data is stored and accessed within computer oriented systems. Continuously growing demands have been the cause for steady development and different approaches on how to manage the data in a collaborative way were a team can share the same view of the information.

There are graphical user interfaces (GUI) for managing different types of databases, where the intention is to create an abstract view over the data inside with a do less accomplish more approach. However, these are often strictly bound and only suitable for one type of database. Furthermore, relationships between data are not visually represented enough and needs to be hardcoded, which can make data management more difficult and time-consuming than it has to be.

The current solutions have their focus on managing the structure of the data. Administrating the data itself can be accomplished, but it can be a time-consuming task, especially when it comes down to relationships. Chained events and recursion across data is not shown and therefore requires the user to be well experienced with the application. This means that the user needs to spend time on learning the structure of the database and how the connections between the data are setup. There are also risks that editing data in one place can lead to unexpected changes, if the user is not well-experienced with the database.

The intention of this thesis has been to deliver a solution to the problem which can shorten the data administration time by providing developers and testers with a visual system for data management. The solution consists of a backend that can adopt itself to relational databases and
derive relationships by itself, which can be presented within a web application. This solution has the intention to fill the gaps that is present in current applications, for example when testers and developers aim to manage the data, rather than the structure of the data. These gaps can be filled in order to save time spent learning the structure and connections of the database. Furthermore, events and recursion in the data are visually shown, which can lead to less unintentional errors when editing the data. This could therefore prove to be a valuable system that can be utilized by testers and developers in need of a better overview and other ways of managing data within a relational database.

The application will also present a solution that will solve the problem of proposing relationships within databases where connections have not been setup. This is a necessity that has to be fulfilled in order for the application to be usable at all. If relationships have not been setup in a correct way, they need to be derived based on predictions that can be gathered through the content of the database.

1.2 Overall aim

The intention of this thesis has been to develop a single page web application for managing relational databases, with focus on relational databases. Furthermore, a backend was developed serving as middleware between a given database and the web application. The purpose of the application was to allow the user to manage a database within the application where the data will be visually represented with relationships, with as little effort as possible from the user. When relationships are setup, the data should be available for management between tables in terms of editing the information where the consequences will be shown to the user. Without existing relationships the application will not be able to visualize the connections between tables and its data, thus an algorithm was implemented which could predict the relationships within an existing database.

The outcome of the thesis could contribute towards information regarding how to determine relationships within arbitrary data, how such connections can be setup with a backend, how a web application could
be built to graphically present it as a relational database and how the data could be managed in a visual manner.

1.3 Scope

The theoretical part aims at giving the reader a brief overview of what databases are in order to understand the reasoning and the results of the thesis. Furthermore some already existing solutions will be presented to help the reader understand what the thesis aims to accomplish in order to be superior in some selected areas than other already existing solutions.

The practical portions of the thesis has had its focus on developing a basic model that visualizes one solution to the problem, but was not complete enough to be considered as a finished product. This, in other terms, means that there had to be certain limitations regarding some aspects. For instance some features were hardcoded and static, there was no user registration and user information management developed. The focus was to make the application suitable for desktop usage, which excludes native approaches that could have been made for other devices such as smart phones and tablets. The web application features responsive design, such that it fits accordingly on different screens and resolutions. Since making a web application, that looks and runs the same on different web browsers, can be a time-consuming task [1], it is only guaranteed that the application works as intended on the latest version of the browser Google Chrome.

Due to time limitations, only one category of databases (i.e. relational databases) was taken into account, although many different relational databases are available for use within the application.

A necessary aspect and concern of any web application today is security. In terms of security, it was not a main focus during this thesis, since it can be applied afterwards. The presented solution developed within this thesis will not be used to manage important and sensitive information used in real-life databases, unless on a protected network.
1.4 Concrete and verifiable goals

This thesis aims at answering the following theoretical questions:

- What category of databases exists and how do they differentiate from each other?

- How are relationships created and can this knowledge be used to implement an algorithm that predicts relationships to solve the problem where they are nonexistent?

- How can interfaces be developed to manage existing databases?

- How will the resulting application in this thesis differentiate from already existing approaches?

Practically, the thesis aims at producing a single page application (SPA) for data management, with the following properties regarded as the minimal requirements:

- Add and manage an already existing database

- Search, sort, filter and edit data within tables of the database

- Add and delete rows within tables of the database

- Display relationships among tables and rows of the database

1.5 Ethical aspects and concerns

While browsing through the web some sites and application stores and caches data about the visitor for a variety of reasons. For instance, the information can be used in order to speed up the website where the purpose is to enhance the user experience. Other areas of use include finding suitable advertisements (ads) for directional marketing, based on the behavior and/or personal attributes of the user.
Cookies are often used by web pages for saving small pieces of information on a client’s computer [2]. This information could, for example, consist of a username and password, where the purpose is to sign in the user automatically without repeatedly requiring credentials for every recurring visit to the web page. It is known that there are certain risks by using cookies, since they can be used in harmful manners. The Swedish law states that all websites utilizing cookies must inform the user that they are doing so and also provide reasons for why they are utilizing cookies, together with some other useful information [2]. The reasoning behind this decision is based upon upholding and protecting the user’s integrity.

This thesis aims at following ethical aspects and also the Swedish law, thus the resulting product will not record any sensitive and personal information about the user. If cookies are to be used within the application, the users would be informed of this fact.

All user interactions within the application were private, such that no information was stored about the user for directional ads, predictions or espionage. Also no data will be stored from the databases managed by the application.

The intention of the resulting application within this thesis is not to manage sensitive data, unless on a protected network. If a computer which possesses the application is stolen, it still needs to be connected to the same network as the added databases for it to be accessible. If an unauthorized user was to extract any of the data stored within a visualized database, data such as passwords and other sensitive information needs to have been encrypted by the database administrator from the beginning. No additional encryption to the data is added within the application.

1.6 Outline

Chapter 2 aims at providing a brief explanation of what databases are, what they are used for and what different kinds of databases that exists.
Relational database web application
- Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

Furthermore it will present the evolutional process of databases and how they handle relationships.

Chapter 3 explains the fundamentals of web development, for example what programming languages that are used, how to deploy for many different devices and a clarification of what defines a web application.

Chapter 4 demonstrates already existing solutions and their approaches for managing data, creating relationships between tables and how this is visualized.

Chapter 5 presents the methods and tools used for the thesis, such that anyone aiming to do an alike research will be able to make similar approaches, if one so desires.

Chapter 6 aims at presenting how the application was developed in terms of structure and design, together with explanations of important algorithms.

Chapter 7 provides the results of the tests that have been utilized to measure the correctness of the predictions the implemented algorithm could make and also a comparison in terms of performance of the tests.

Chapter 8 contains all the conclusions in an analyzing manner, complete with evaluations and personal reflections that have been observed and noted during the thesis work. It also presents a proposal on how the results can be improved upon in future works to create a more complete implementation for a final product.

1.7 Contribution
The prediction algorithm, front- and backend were developed, designed and implemented by the author. The idea for this thesis was provided by Johan Deckmar at Dewire AB, together with a background and description of the problem.
This thesis work has contributed towards on how Dewire could handle test data within an application that can be used internally by their developers and testers.

For database management as a whole, the work has contributed towards on how data within relational databases can be visualized and managed in a relational matter. Furthermore, it has contributed towards on how relationships between tables can be automatically setup by utilizing an algorithm which can make predictions of the relationships.

This report will continue by presenting the theory, method, implementation, results and conclusions of this thesis.
2 Databases

The databases chapter briefly introduces the fundamentals of databases and different types of database models with focus on the relational model.

2.1 Introduction to databases

At many workplaces today there is a need of keeping track of things [3]. A typical company might need to keep track of, among other things, their customers, sales and staff. For a few rows of data, this could be done by writing a list by hand. As the number of rows increases, so does the needs for having a more organized system for the data and other ways to access it, which cannot be achieved by using a list.

Historically a common method for managing computerized data was to store it within files, often in table like representations [4]. When one was to manage this data, the administrator had to create a program that would meet the specified requirements, for example retrieving sales within a time period for a specific product. Abstraction between files became complicated, since there were different programs for each file. Furthermore, each file was also often stored locally at one department. Information sharing between departments became complicated due to version managing and data inconsistency. The problems with the file system approach were thus that it became hard to find, manage and access data.

Throughout the years there have been different solutions on how to build a database in a more suitable manner [4]. A common name for this became data models, which serves as a description and a set of rules on how the data should be structured and managed within the database. It is a way for administrators, programmers and other staff to get a clear and shared view of the data and how it should be organized.

The hierarchical data model was one of the first ways of modeling databases [4]. It is structured in a way that all records could have many
children, but only one parent. Later came the network model, which improved upon its predecessor by allowing each record to have more than one parent and more ways to structure the database such that it could be viewed by different units in a more flexible way. However, these also caused unplanned changes that had to be implemented into the database became difficult to handle. A major breakthrough to data models came with the relational model, which will be discussed in deeper detail in chapter 2.3. A modern way to treat data is to store it within objects in an object-oriented model. It gives a clear representation of the real world, while at the same time it shares similarities with object-oriented programming. Objects contain both data and relationships with other objects and they can be grouped into classes where each class contains methods for managing the objects and the contained data.

2.2 Database management systems

A solution to the issues with, for example file systems, were solved with database management systems (DBMS), which is a method to abstract raw data stored within a database from the user. A DBMS can (among other things) assist by [3]:

- Adding better and more consistent managing of data stored within the database

- Enabling the database for access by multiple users and applications at the same time

- Improving the security by, for example, adding password protection for authorization and encryption to the data

- Data consistency and version management

2.3 Concepts of relational databases

Relational databases are modeled as tables and thus store the data in rows and columns [3][4]. Each row contains data that belongs to the
Relational database web application
- Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

A column within the table consists of a descriptive title for categorizing the data within the table, together with a specific data type. A relational database consists of a collection of tables where each table contains data that in some sense is connected with each other. Due to the simplicity and methods for visualizing data, relational databases have since its introduction become a commonly used approach worldwide.

Tables within the database can be related [4]. For example, a table defining a vendor can be related to a table consisting of products. Each vendor within the vendor table is unique and is therefore assigned with a unique identifier called primary key. The reason why the primary key must be unique is that each row needs to be uniquely identifiable. For each product, the vendor id alone can be stored and inserted on the same row, which not only saves memory, but also serves as a connection between the tables so that it can be known which product is sold by which vendor.

Depending on the structure that the administrator wants to achieve, there are different types of relationships that can be established within the database, which consists of [4]:

- One to one relationships (1:1). One instance from one table is related to another in a second table. For example, a professor only belongs to one department.

- One to many relationships (1:M). One instance from one table can be related to others in a second table. For example, a painter can have painted many paintings.

- Many to many relationships (M:N). Many instances from one table can be related to many others in a second table. For example, each class can have many students and each student can take many classes
To create one to one relationships and one to many relationships, foreign keys are utilized [4]. Foreign keys can be explained as a connection that shows which primary key it is pointing at. This means that the primary key and foreign key requires the same data type. Defining many to many relationships between tables in relational databases is done by creating another table, where each row generally consists of two foreign keys which refers to primary keys in two different tables.

Defining relationships does not only help by visualizing connections between tables, but can also be used for practical purposes when it comes to managing data in a relational table [3]. The property cascade can be set on foreign keys, such that when data is edited or deleted in one table, they will trigger an event on every related table which the corresponding foreign key is related to. Other options can also be set upon deleting and updating data, such that no action is performed, parent data cannot be changed without first deleting its children and that changed data with connections is set to null.

2.4 Naming conventions for relationships

Being aware of naming conventions gives anyone viewing the database a hint about a foreign key’s originating table [4]. For example, the columns can have the same name, or nearly the same name, as the table it relates to. The table name could be concatenated as a prefix, together with a suffix, which could consist of: "id", ".id", "_code" or "code". For tables with long names, the name of the table can be abbreviated, for example "employee" can be written as "emp" and project can be written "proj".

2.5 Managing databases

Since writing single-purpose programs for managing data within early types of databases were too time-consuming (and thus ineffective), there was a need for a more abstract approach which could adapt to different needs [4]. Over the years many languages for database management have been developed with different paradigms and could be combined for multi-purpose usages [5]. Common types of languages that could be found in databases were data definition language (DDL), view defini-
tion language (VDL) and data manipulation language (DML). DDL was used to define and create the structure of tables, VDL could specify user views and their mappings to tables and DML essentially was a language which could create, read, update and delete data (CRUD).

Structured query language (SQL) is a language used for managing relational databases and could be viewed as a combination of DDL, VDL and DML, but with even more features [5]. It is used to create tables for which data can be inserted, read, updated and deleted. Due to the simplicity and readability of SQL it is one of the reasons for the popularity of relational databases. SQL featured data types, which typically meant that only one type of data was allowed and thus could be inserted into a field belonging to a certain column. These types are: numeric, character string, bit string, Boolean, date and time. Some of these data types featured various sizes, for example a numeric value could be specified as INT, SMALLINT, FLOAT, REAL and DOUBLE. These sizes set the allowed range of the stored numbers.

Instead of having the development based upon a single database, an object relational mapping (ORM) is a programming technique that can be used to have a single code working for several databases[6].This approach is implemented in many languages like Java, Ruby and .NET.ORM is a way to define the relational model of a relational database as an object model [7]. This means that ORM tables can be typed as classes, where significant queries are stored as methods within the class.

2.6 Information Schema
The Information Schema is the name of a collection of views automatically created by some relational databases, for example PostgreSQL [8], MySQL [9], MariaDB [10], SQLite [11] and MSSQL [12] creates an Information Schema. It contains views with metadata about, for example, all databases, tables, columns and relationships that currently exist on the server. This information can be used to check which tables exist within a given database, which columns it has and which relationships it may have.
3 Developing web applications

The following chapter introduces some fundamentals and practices of web application development.

3.1 Structure and programming languages

Web applications are applications that require a web browser in order to be used [13]. There are many frameworks, libraries and programming languages that web developers must choose between when developing a web application today. Some are more fundamental than others and some are more situational.

A standard web application can usually be divided into three layers, which are [14]:

- The presentation layer, this layer is what the user sees and interacts with while using the web application.

- The logic layer, enhances the presentation layer by adding more dynamic interactivity while also solving computational and programmatic issues.

- The data layer. Stores, retrieves and edits information within a database.

The programming languages used for web development are usually divided into two categories, imperative and declarative languages [14]. Declarative languages are typically used for single purpose usages, such as creating the structure of an application. These are usually hypertext markup language (HTML) which is used for marking the application with tags. Cascading style sheets (CSS) which is used for styling the content by adjusting fonts, colors and sizes. Imperative languages can be used for multiple approaches within programming and are not only bound to web applications. Imperative languages that are used within
web programming are many, but the most popular ones are JavaScript, Java, PHP and C languages. JavaScript is an important web development language used on the client side to define the behavior of the web application, while the other languages usually resides on the server side and serves as an interface between the client and server.

3.2 Web applications for multiple devices

Due to the specifications of smart phones, such as high performance, high resolution and more, they are fully capable of running advanced web applications, compared to their early counterparts [1]. Developers faces new challenges when developing mobile-friendly web applications, since smart phones comes in many different shapes regarding resolution, pixel ratio and physical size.

Therefore, there are mainly two options, go native or choose another approach [1]. Programming native apps generates higher performance since the code is compiled, but it also requires the code to be written in a specific language such as Java for Android and Objective-C for iOS. The other option is to let the application be run within the browser of the device, since all browsers can run JavaScript, but some animations and other craving operations can be impaired by low performance. However some devices feature hardware acceleration, which produces smooth results for the device's web browser.

Even though choosing a web approach rather than going native, it requires the developer to be well experienced with how much certain operations and styles costs in terms of performance [1]. Also there are more devices and browsers that the application must be tested on.

3.3 Single page applications

Normally when interacting with links on the web, one is taken to a new page, which results in a refresh or a replacement of the whole page even though some content is still the same and thus does not need to be loaded again [15]. The idea of a SPA is to let the content be updated, rather than replaced, which results in quicker response times and other
ways of managing the data. This is done by enabling the browser to retain a page even though it requires server communication.

3.4 Ember.JS

Ember.JS is an open source framework for developing single page applications. It consists of different concepts which can be named as models, routes, controllers, views and templates, see Figure 1.

A model describes the data which is to be rendered and share similarities with database concepts [16]. For example a model could describe unique users with a name and an id.

A router in Ember.JS contains many routes [16]. A route essentially works as a glue between a controller and a template and makes the loaded model available to the controller. The different routes are loaded based on the unique resource location (URL) typed into the browser. For instance, visiting www.blog.com would be served through an index route, while visiting www.blog.com/posts would be served through a posts router.

A model is bound to the controller by the router so that values from the model can be accessed by the controller [17]. Controllers can be used to create computed properties and observe changes on the loaded model.
Furthermore values can be defined to keep track of template states, but could also be used in templates.

A view can be used to handle click events made by the user on templates [18], but they can also be defined to act as custom components to be used in a template, which can save code when one needs a custom element which is continuously used.

Templates are the pages which contain the rendered content that the user can interact with [16]. They are written in HTML, but with Ember.JS Handlebars computed properties, HTML elements and data loaded from within a model can be rendered. The Handlebars are included in the templates and are replaced by values from either the route, controller or view which has the same name as the Handlebar.

For example, if one were to develop a blog in Ember.JS and the developer would like to show posts to the user it needs to have at least one route, controller, template and optionally a view [19], see Figure 2.

**Figure 2: Concepts and naming conventions are important in Ember.JS**

The name of a template decides which controller, route and view it will use, therefore naming conventions is essential in Ember.JS [19]. These different components are stored in unique folders, which makes it easier to keep them separated from each other. Other names for every component can be chosen, but it is not recommended.
A representational state transfer application programming interface (REST API) adapter can be used in Ember.JS to send hypertext transfer protocol (HTTP) requests to a server [20]. The HTTP requests can be sent to update (PUT), retrieve (GET), delete (DELETE) and add (POST) data in for example a database. All responds to the REST adapter are expected to be formatted as JavaScript Object Notation (JSON). Ember.JS automatically sends HTTP requests based on the URL and the model that is connected to this URL. For example, if the URL is http://example.com/posts/ Ember.JS will send a GET request to the server, which will return a list of all posts in JSON format. Ember.JS expects the JSON to be formatted in a specific way. For example, a post could contain a title, some text and some comments that are bound to it. Ember.JS expects this to be formatted as:

```javascript
{
  post: {
    "id": 1,
    "title": "example title",
    "text": "example text",
    "comments": [1,2]
  }
}
```

Ember.JS would interpret this as a post with id 1, title "example title" and comments 1 and 2 [20]. For each comment, Ember will send another request to http://example.com/comment/id, where "id" is the id of the comment.
4 Existing applications

Chapter 4.1-4.2 presents PhpMyAdmin and MySQL Workbench GUI’s, with focus on their solution of the functionality that the application this thesis will result in.

4.1 PhpMyAdmin

PhpMyAdmin is a web application that is used for managing MySQL databases and is considered to be a simple and available system for its purposes [21]. PhpMyAdmin can for example be used to CRUD databases, tables and data[22]. This is done by browsing through menus, filling in input fields and selecting suitable options from dropdown lists. All of these operations can also be written in SQL commands which will produce the same result.

In PhpMyAdmin tables can be managed individually [22], see Figure 3.

![Figure 3: An example table with editable data in PhpMyAdmin](image)

Data is visually presented as tables with rows and columns containing data and attributes [22]. The data can be sorted, filtered and managed. Rows can be deleted by clicking on the delete button of the corresponding row, also new rows can be added by filling out a form.
Relational database web application
- Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

Existing applications

In PhpMyAdmin relationships can be created by providing required values in a form [22], see Figure 4.

**Figure 4: Form for creating relationships within PhpMyAdmin**

Different options can be set to control what happens upon editing and deleting related data [22]. Currently there is no way for PhpMyAdmin to automatically detect and define relationships within a database, all relationships needs to be done by SQL queries or with forms.

Delete and update events set on foreign keys is useful when managing tables [22]. If delete and update is set to restrict, any changes to a parent insertion will prohibit the user action and send an error message instead. However, if delete and update is set to cascade and a parent element is changed, only the changes to the parent element will be shown since tables can only be viewed one at a time.
Relational database web application - Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

Existing applications

Relationships can be shown in an overview with the help of a feature called Designer [22], see Figure 5.

![Designer overview showing relationships between a set of tables.](image)

Designer is a drag and drop tool which can be used in order to show the structure of a database and the relationships between tables [22]. It is a visual tool that shows columns and their data types, but no changes to the tables can be made in this view.

### 4.2 MySQL Workbench

MySQL Workbench is a GUI application for portraying data modeling, physical database design, database creation and management [23]. It is developed by Oracle, the current owner of MySQL [24].

In MySQL Workbench tables can be managed individually, see Figure 6.

![An example table within MySQL Workbench with editable data](image)

Tables can be managed individually in the GUI, or by typing queries in the SQL command window. Data can be sorted, filtered and rows can be deleted and added. To add rows no additional tabs needs to be opened.
Relationships can be setup either by SQL queries or in the GUI, see Figure 7.

Figure 7: Foreign keys declaration, references and options in MySQL workbench

A foreign key within MySQL Workbench must be assigned with a name, together with the columns that are to be connected. Also on delete and on update options need to be set.

A feature called "reverse engineer" within MySQL Workbench can be used to view an already existing set of tables and see their structure and relationships. The reverse engineer diagram shows the tables columns, names, attributes etcetera, see Figure 8. Relationships between tables can be setup by using the diagram tools and CRUD operations can be performed on one table at a time.

Figure 8: A diagram showing tables and their relationships within MySQL Workbench

There is no feature that automatically suggests or adds relationships within MySQL Workbench, this needs to be done manually by the user.
5 Method

The method chapter declares how the thesis work was done, which approaches, tools and measurements that were used and how they were performed.

5.1 Development methods and work process

The method chosen for this project was an agile approach, due to time limitations, needs for fast results and that it is a commonly used method by developers worldwide. The thesis was initialized by first understanding the problem, researching the background and estimating a time table. A system architecture document (SAD) was conducted to summarize what had to be implemented in the application together with mockups, which served as a proposal on how the final implementation would look like. The reasoning for this was to make sure that the presented idea could be regarded as a solution to the problem, before any code was written. The application was then implemented and has been continuously tested during the whole development with acceptance tests. The criteria’s that these tests had to pass was that the user stories that were defined in the SAD had to be fulfilled within a humane time limit. Lastly, the implemented prediction algorithm was tested.

5.2 Frontend development

5.2.1 Base tools

The frontend was essentially built on HTML and CSS. HTML was used for marking and creating the structure of the application. CSS was used for styling the page in terms of text, colors and icons. Also some animations were added by using CSS. These tools were used since they are standard for web development [14].

5.2.2 Ember.JS

Ember.JS is a JavaScript library used for making fast and responsive single page web applications. It was used to shorten the development
time [13], as it follows a do less - accomplish more approach [25]. Other frontend SPA frameworks were taken into consideration, but they had different programming paradigms, thus Ember.JS was selected since the author had previous experience with it and no time had to be spent learning how to use it.

5.3 Backend development

5.3.1 Node.JS

Node.JS, often referred to as Node, is a backend server environment focusing on high performance and low memory consumption [26].

Node was selected for this thesis due to its functionality and capacity, while also minimizing the amount of programming languages that had to be used within the project [26]. Node also has a large community, where users have developed many libraries which can be downloaded and installed as packages within Node.

5.3.2 Restify

Restify is a Node module that is built for creating REST API’s running on a Node server [27]. It can be used as a component for handling HTTP requests from external sources such as the REST adapter utilized by Ember.JS.

5.3.3 Sequelize

Sequelize is an open source ORM written in JavaScript as a package to be used together with a Node server [28]. Sequelize is compatible with different relational databases, such as: MySQL, MariaDB, SQLite and PostgreSQL. Sequelize was chosen to minimize the number of programming languages to be used within the thesis, to have the application work with several databases and since the author had experience using it in previous projects.

5.3.4 Sequelize-Auto

In order for Sequelize to query tables of a given database, a model needs to be defined for each table [28]. Sequelize-Auto is an unlicensed add-on to Sequelize that automatically builds table models by providing the
credentials of a database [29]. The generated model contains the name of the table, together with its corresponding columns and data types. One thing to note is that Sequelize-Auto does not import or predict relationships between tables.

5.4 Development tools

5.4.1 Layout and design

The initial mockups were made using a regular pen and paper. These sketches were later digitalized and improved upon using Photoshop, which is an image editing tool. The layout of the application was inspired by using design patterns from Google Android [30] and Apple iOS [31].

Font Awesome is a set vector icons licensed under the SIL Open Font License (OFL) [32] used as scalable images that would adapt themselves to fit the resolution of the active display. Some of these icons provided by Font Awesome was used within the application and thus not made from scratch by the author [33]. Instead properties, like size and color, were edited within Photoshop.

Graphs and diagrams used within the thesis were made using yEd Graph Editor, which is a visual editor for creating graphs, diagrams and charts [34].

5.4.2 Programming environment

The integrated development environment (IDE) Sublime Text 2 was used for editing and producing code used for the thesis, both on the front- and backend [35]. It was used to fasten and ease code production due to its many features.

5.5 Measurements

5.5.1 Predicting relationships

In order for the application to be usable it needs to know the relationships of any given relational database. This means that there must be a fallback option if the database administrator has not defined the relationships. This has been the reason to why an algorithm which can
predict relationships had to be developed as a necessary component of the application.

It is known from chapter 2.3 that a primary key and a foreign key needs to have the same data type in order for them to be related and from chapter 2.4 that tables and columns can have similar names to give the user a hint that they might be related. Therefore developing an algorithm which can solve the problem with databases that do not have any predefined relationships by predicting these was implemented, focusing on these properties as its main strategy.

An already existing algorithm for finding the difference between two strings is called the Levenshtein algorithm [36]. It was the most suitable algorithm found for comparing strings. Given two strings, the output will yield a numeric value which is the number of changes (edits, insertions and deletions) that has to be made for the strings to be equal. This means that two strings with low distance are more closely related to each other than two strings with a long distance. Since column and table names can be similar, if they utilize convenient naming conventions, a low Levenshtein distance between a table and a column name can be seen as a hint that they might be related.

An algorithm was developed which would loop over every table name in the database and compare its name, by using the Levenshtein algorithm, with the names of the columns in other tables and then comparing the data types of the proposed relationship. A substring function was used together with the Levenshtein distance to determine relationships in a second test, the results from this test was compared with the results from using the Levenshtein distance only to predict relationships. The reason for why the substring function was used was because, from chapter 2.4, it is common that column names have the table name left unchanged and used as a prefix.

5.5.2 Prediction evaluation
A program was implemented in order to measure the amount of correct predictions the algorithm could make, this was done in two different tests. During the first test, any table name and column name with less
than or equal Levenshtein distance compared to the current iteration number was predicted to be related. During the second test, the table name had to be a substring of the column name before the Levenshtein distance was measured.

The tests were performed in eleven iterations. Starting from zero going up to eleven, tables were predicted to have a relationship if the Levenshtein distance of a table and a column name, in the other table, was less or equal to the iteration number. In chapter 2.4, the suffix for relationships were found to be about two to five characters long, which means that it would require six iterations for the algorithm to predict relationships in all databases which follows this standard. The reason for doing the tests in eleven iterations was to cover up cases where the suffix might be longer than expected.

In chapter 2.4, abbreviations of the related table name were also found to be used as a prefix. Since abbreviations are subjective and thus can be hard to predict, the algorithm was not constructed to take this property into consideration.

5.5.3 Prediction algorithm performance evaluation
Test were performed to measure the performance of the algorithm and to compare the methods of using only Levenshtein and a combination with substring. The iterations described in the previous chapter was repeated 100 000 times for each iteration and from this the mean value was calculated. This was done to reduce errors in the measurements. The performance was measured in time from when the algorithm started to predict relations until it was done. The tests were performed on a Macbook Air with 2 GHz Intel Core i7 processor and 8 GB 1 600 MHz DDR3 RAM.

5.5.4 Tested databases
The measurements were performed on example databases found on MySQL’s official web page [37]. Provided on the web page are four databases which are the "employee", "sakila", "world" and "menagerie" database. The menagerie database only consists of two tables, thus it does not require an algorithm to predict its relationships and was
therefore not an eligible candidate. The other three were used for testing the algorithm.

The employees database is centered around employees and describes, with tables, their relationship to properties such as their department and title [38]. The employees database is licensed under the Creative Commons Attribution-Share alike 3.0 Unported License [39], which means that anyone is free to copy, redistribute and/or modify the database[40]. The employees database uses abbreviations for their prefix, for example employee is abbreviated to emp and department is abbreviated to dept [38]. These are combined with a suffix "_no" or "_name". The employees database consists of 6 tables and 6 relationships. Every table can have a total of 5 relationships, thus the maximum number of possible relationships is 30.

The sakila database is a standard schema that can be used in, for example, books, tutorials and articles [41]. The structure of the sakila database centers around a film rental store which is categorized into an inventory of films, customers and business [42]. The sakila database is licensed under the new BSD license [43], which means it can be modified and redistributed [44]. The sakila database uses strict table names as a prefix for its foreign keys, together with ",_id" as the prefix [45]. The sakila database consists of 16 tables with 21 relationships. Since every table can have a total of 15 relationships, the maximum number of possible relationships is 240.

The world database describes the relationship between countries, cities and languages [46]. The structure of the world database is unlicensed, but the data is copyrighted by Copyright Statistics Finland. Since only the structure of the database was of importance, this could be ignored. The prefix for the foreign keys is the name of the table and the suffix is "Code". The world database consists of three tables with two relationships. Each table could have two relationships, thus the maximum number of possible relationships is six.

The MySQL example databases were chosen for testing since it was one of the database dialects that were supported by Sequelize [28] and that
the author had previous experience with it. Additionally, some of them were also recommended to be used as example databases [37] [41]. Furthermore, in order to check if the predictions were correct, the database used within the tests had to have its relationships predefined keys by its administrator, which were the case with these databases.

The gathered information has been analyzed, revised and compiled into an implementation and a result.
6 Implementation

Chapter 6.1 - 6.3 describes how the application was implemented, from an overview of the system architecture to detailed information about the front- and backend.

6.1 System architecture

The developed system consists of a database which is connected to a backend which serves the frontend to a user, see Figure 9.

![Figure 9: The system architecture](image)

The user interacts with the SPA to add an existing database to the backend server. The backend queries the database in order to serve data to the SPA, which the user can interact with. It is not required by any of the components to have a connection to the Internet, as long as they are reachable on the same network. Furthermore, the backend is not limited to one database, several databases can be added within the SPA.
6.2 Frontend

The frontend is a web application built with the Ember.JS framework, see Appendix A for screenshots. It consists of several routes, controllers, templates, models and views, where the index, database and table route are the components of most significance, see Figure 10.

The index route serves as the main route, which is accessed by visiting the base URL. The other routes and its attached components are loaded depending on the user navigation.

The application is populated with data from a given database and is stored within dynamic models. The database model is the root model which can have many tables, while a table can have many columns and rows and a row can have many rowcontent. An ER-diagram was drawn to illustrate the model structure, see Figure 11.
By implementing the models in this manner a database and its corresponding tables can be added dynamically without the need for creating specific models for each and every database and table.

The application can be divided into three parts, which are the header, menu and content. The user must first provide the credentials of an already existing database for the application to be usable. After the credentials are sent, the request is handled by the backend for which a detailed explanation is provided in chapter 6.3. After the request is handled by the backend, the database is added to the menu together with its corresponding tables, see Figure 12.

![Figure 12: The menu with a selected table and a related highlighted table](image)

Clicking on a database will expand it into a list of tables that are held in the database. Clicking on a table in the aside menu will show the selected table together with its related table(s) in the content area. Tables
can be filtered by providing text in a search box, which then will only show tables that match the input.

The header displays the name of the current database and selected table, together with options for managing the tables such as deleting/adding rows and showing the attributes, see Figure 13.

![Figure 13: The header title, options and actions](image)

In the header there are buttons for controlling the content. The "structure button" toggles between displaying the actual data in the tables and the column attributes. There is also a button for showing/hiding the side menu.

Tables are shown on the largest part of the website, the content area. They are visualized with a title and their corresponding columns, rows and data, see Figure 14.

![Figure 14: Two related example tables](image)

A variety of actions can be performed on each table. Each column can be sorted with the values ascending or descending, every value can be edited by simply clicking on it and attributes can be changed by selecting a different attribute in a selection list. Values can be searched for by providing text in a search box, which then will only show rows that has
content that matches the input. Rows can be added by selecting a table and then click the "Add row" button in the header. Rows can also be deleted by marking them by clicking their corresponding checkbox. The number of marked rows is displayed in the header and can be removed by clicking the "Remove row" button. Rows that are marked will also filter the tables, by only showing rows that have a relationship, see Figure 15.

![Figure 15: Marked rows that are filtered on user id 1](image)

The filtering is done by looking at the primary and/or foreign key(s) of the marked row and comparing them with rows in other tables. Only rows which match the filter are shown and marked.

### 6.3 Backend

The backend is a Node server which communicates with databases added within the application. The databases are communicated through Sequelize, an ORM which supports dialects such as MySQL, MariaDB, PostgreSQL, SQLite and MSSQL. Communications to the server are handled by Restify, which processes the request and completes necessary queries by forwarding it to Sequelize. Restify compiles the result from the query made by Sequelize into a JSON format that is expected to be received by Ember.JS.
A database is added in the web application when the user provides necessary credentials. The credentials consists of a name, address, port, username and password of the database. Once this is done the server must:

1. Collect the information which is provided by the user, these are stored in a table designated to the application for keeping track of which databases that has been added within the application

2. The modified version of Sequelize-Auto queries for metadata from the provided database’s information schema, to retrieve names, attributes and relationships

3. The algorithm continues by predicting relationships that has not been defined by the database administrator by comparing table names with columns in other tables (explained in greater detail in chapter 6.4)

4. Models for each table are created from the metadata and are stored on the server

Once all these steps are done, the provided database and its corresponding tables can be accessed and managed within the application and will be shown in the menu list.
Relational database web application
- Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson 2015-07-10

To meet the requirements of the Ember REST adapter, the following routes were setup with Restify, see table 1.

**Table 1: Server side REST routes**

<table>
<thead>
<tr>
<th>Action</th>
<th>Request</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find All</td>
<td>GET</td>
<td>/database</td>
</tr>
<tr>
<td>Find</td>
<td>GET</td>
<td>/table/123</td>
</tr>
<tr>
<td>Find</td>
<td>GET</td>
<td>/column/123</td>
</tr>
<tr>
<td>Create</td>
<td>POST</td>
<td>/row</td>
</tr>
<tr>
<td>Find</td>
<td>GET</td>
<td>/row/123</td>
</tr>
<tr>
<td>Delete</td>
<td>DELETE</td>
<td>/row/123</td>
</tr>
<tr>
<td>Find</td>
<td>GET</td>
<td>/rowcontent/123</td>
</tr>
<tr>
<td>Update</td>
<td>PUT</td>
<td>/rowcontent/123</td>
</tr>
</tbody>
</table>

Ember will automatically send an HTTP requests to the REST routes depending on which model that is loaded by the frontend. For example if the database model is loaded, Ember will automatically send an HTTP request for each table associated with the database.

The tables are the heaviest objects the frontend needs to get from the backend and thus require the most requests. The number of requests needed when a table route is requested is:

\[
\sum_{i=1}^{a} n_i + n_i \cdot m_i
\]

Where \(a\) is the number of tables (the table itself and all its related tables), \(n\) is the number of columns and \(m\) is the number of rows.
Relational database web application
- Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

Implementation

For example, if we have two tables which are related, where the first table has 5 columns and 1 000 rows and the other table has 10 columns and 500 rows this would result in:

\[ \sum_{i=1}^{a} n_i + n_i \cdot m_i = (5 + 5 \cdot 1000) + (10 + 1 \cdot 500) = 5515 \text{ requests} \]

For this example, the application would have to send 5 515 requests, which the server would have to respond to one at a time.

6.4 Algorithm for determining relationships

The algorithm for determining relationships is written in JavaScript and implemented on the Node backend server. It is a necessary component that ensures the application’s usability when predefined foreign keys are missing. The test is started by sending an HTTP request, which is handled by Restify. The request takes three arguments, which database the test is going to be utilized upon, the allowed Levenshtein distance and which one of the two available tests that should be used. Restify continues by passing the name of the database to Sequelize which queries the Information Schema for all tables which belongs to the provided database, together with each table’s columns and their corresponding data types.

When all necessary data has been gathered, the algorithm compares the data types of the columns (since this operation is less complex) and if they match it continues by comparing each table name with each column in all the other tables. If the column and table name is within the accepted Levenshtein distance, they are proposed to be related.

The time complexity for the algorithm is (with database queries excluded) \( O(n^2 \cdot m) \), where \( n \) is the number of tables and \( m \) is the number of columns. This due to it requires three loops to check all table names with all column names in other tables. The time complexity for the Levenshtein distance algorithm is \( O(n \cdot m) \), where \( n \) and \( m \) is the number of characters for each string, where the complexity for the substring function is \( O(n) \).
Chapter 7.1 - 7.3 presents the results from the MySQL example databases, the employees, sakila and world database. The results of the first and second tests are presented with the number of predictions the algorithm has made, how many of the predictions that were correct and incorrect, how many actual relationships the database has and how many possible maximum relationships the database could have. Finally, for comparison, the test results correctness and performance are shown.

7.1 Employees database results

The prediction algorithm was run first with test one on the Employees database, which only utilized the Levenshtein distance, see Figure 16.

Figure 16: results from the employees database using the Levenshtein distance only

The algorithm managed to predict relationships starting at the third iteration, however these were all wrong. At the sixth iteration all relationships had been found, while the number of wrong predictions had increased.
The prediction algorithm was then run with test two, which utilized a substring function. During this test, the algorithm did not manage to predict any relationships in the employees database utilizing the substring function. Looking closer at the structure of the database, the cause for this failed attempt to predict relationships is that the foreign keys contain abbreviations. For example, employee and department have been abbreviated to emp and dept.

Results from Figure 16 were compiled to show the amount of predictions that were correct in %, see Figure 17.

![Employees database: Levenshtein test in %](image_url)

**Figure 17: The correct amount of predictions in the employees database**

As can be seen the first test which only utilized the Levenshtein distance managed to make predictions, but the second test which only allowed substrings could not do that. This because abbreviations often contains some of the same letters as the original word, this yields a distance within the iteration interval, which the algorithm can predict as a relationship.
The performance of the two test cases were performed, see Figure 18.

Figure 18: Performance results using both test cases on the employees database

During the first six iterations both test cases are almost equal in performance, but as more predictions are done by the Levenshtein distance only, there is also an increase in time for which it takes to treat these predictions, thus more time needs to be spent.
7.2 Sakila database results

The sakila database was tested with the first test of the prediction algorithm, which only utilized the Levenshtein distance, see Figure 19.

![Sakila database results graph](image)

Figure 19: Test results from the sakila database using only the Levenshtein distance

After four iterations the algorithm manages to find all the correct relationships. This because sakila uses the table name as a prefix and the three character long suffix "_id". As the number of iterations increases, so does the number of incorrect predictions, because the search span is increased beyond the suffix, more candidates are within the Levenshtein distance. This in turn leads to a full mesh of relationships, since with a high enough accepted Levenshtein distance, two strings can be completely different, but still be accepted by the algorithm.
The second test utilized a substring check together with the Levenshtein distance, see Figure 20.

![Sakila database: With substring](image)

**Figure 20:** Test results from the sakila database extended with substring

The naming convention of the sakila database used the strict table names as prefixes for the foreign keys, thus the algorithm managed to find all the correct relationships after four iterations, since the suffix consisted of three characters.
The results shown in Figure 19 and Figure 20 were plotted to show the percentage of predictions that were correct, see Figure 21.

![Sakila database: Prediction comparison between Levenshtein and substring](image)

Figure 21: The correct amount of predictions in % on the sakila database

The results from the sakila database shows that the algorithm's predictions with an allowed Levenshtein distance of three gives a 100 % correct result in both cases. However, as the allowed distance increases, the amounts of wrong predictions are increased in test case one, which only checked the Levenshtein distance.
Performance tests were also done on the sakila database, see Figure 22.

As with the employees database the performance of the test cases are pretty equal in the beginning, but as the number of predictions increases that are made by using the Levenshtein distance only, so does the amount of operations to treat the predictions and therefore there is a decrease in performance.
7.3 World database results

Initially the prediction algorithm was used, measuring only the Levenshtein distance to predict relationships, see Figure 23.

![World database: Levenshtein distance only](chart.png)

**Figure 23: Test results from the world database using only the Levenshtein distance**

The results show that at the fifth iteration the algorithm manage to find all the correct predictions, but together with an error. As the number of allowed difference in Levenshtein distance increases, the algorithm comes closer to predicting a full mesh between the table’s relationships.
Secondly the prediction algorithm was used on the world database, including the substring function for predicting relationships, see Figure 24.

The world database uses the table name as its prefix and "Code" as its suffix, which is the explanation for why the algorithm successfully finds all the correct relationships in the world database, without errors after five iterations.
Relational database web application
- Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

The results from Figure 23 and Figure 24 were combined for comparison, see Figure 25.

Figure 25: The correct amount of predictions in % on the world database

The second test which only allowed substrings managed to predict only correct relationships, while the first test had errors.
Relational database web application - Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

Results

Lastly performance tests were made on the world database, see Figure 26.

![World database: Performance comparison between Levenshtein and substring](image)

Figure 26: Performance results using both test cases on the world database

Using the Levenshtein distance performs better on the world database since it does not have the extra complexity that comes with using the substring function.
8 Conclusions

Chapter 8.1 - 8.2 evaluates the theoretical and practical work that has been conducted during the thesis and also proposes tasks that can be done in the future to enhance finalize the prototype into a complete product.

8.1 Evaluations

8.1.1 Theory

The theory study has shown that there is a variety of approaches for managing larger amounts of data and that they differentiate in how the data is structured and stored. One approach for managing and structuring information is done with relational databases. The data is stored in tables within cells, which belongs to rows and columns. Relationships can be made by creating foreign keys in one table, which is a reference to a primary key in the other table. The most important conclusion from the theory study is that it has been found that if the database is constructed with a consistent and certain naming convention, relationships can be found by comparing table names and column names.

Standardized tools are used to develop web applications, but there are many approaches that need to be taken into consideration when developing them. Much depends on which web browser the user has, but also on which devices. Ember.JS can be used to make SPAs which does not require the whole page to be updated. Instead, only the content that is changed needs to be refreshed, which enhances the response time of the application. Combining these tools and approaches, it has been shown that they can be used to create a web administration interface for relational databases.

There are existing applications, for example PhpMyAdmin and MySQL workbench, that can handle relational databases, but their approaches are focused towards managing the structure of the database, rather than the data itself. Thus it requires more operations to perform the practical
tasks that this thesis has aimed to implement. Furthermore, relationships needs to be added manually within these applications, there is no future to automatically add or suggest relationships, which makes the resulting application in this thesis stand out from other implementations.

8.1.2 Methods and development
The methods and tools used within the development have resulted in a working application that can be used as a proof of concept to demonstrate the functionality of a web administration interface with focus on relational databases. With more time, the used methods could have focused more on testing to measure and ensure the correctness of the prototype. With more people assigned to the project, much of the work could have been divided into different parts, based on the team member’s expertise.

However, there are many other programming languages, frameworks and software that might have been better to use. With more time, more research could have been put into examining other solutions, which might have produced a better result, rather than relying solely on old developing experiences.

In the thesis the amount of correct predictions were measured that the implemented algorithm could make. The algorithm could have, with more time available, been further evaluated in terms of, for example, performance.

8.1.3 Implementation
It has been shown that an application centered on databases and its contents can be built based on metadata, which could be queried from an Information Schema, even if this only can be guaranteed to apply for the databases within the scope of this thesis.

The implemented application specializes in showing relationships between tables and can predict them if they have not been defined by an administrator. All the minimal requirements have been met, an already existing database can be added for management, data within it and its
tables can be searched, sorted, filtered and edited. Tables are shown complete with rows, columns and data together with its related tables. Furthermore, rows within tables can be added and deleted. When relationships are not predefined, an algorithm can be utilized to predict them so that the application remains usable for the user.

8.1.4 Results of the prediction algorithm

The tests were performed on example databases provided by MySQL and showed that when the database had an consistent and certain naming convention, the algorithm would make correct predictions. The algorithm should prioritize the substring approach, but in cases where the algorithm cannot make any predictions at all, it could make predictions based on the lowest Levenshtein distance possible. It was shown that the algorithm failed to make any predictions based on the substring method where the column names were abbreviated. An improved version of the algorithm could, if the substring approach fails, make an attempt to abbreviate the word and fallback to using only Levenshtein.

Even though the algorithm made 100 % correct predictions in the tests where the database had suitable naming conventions, it cannot be guaranteed that this will be the case for all databases. The algorithm should therefore not be considered as a solution to finding all the relationships within a database; instead it should be regarded and presented to the user as suggested way to find relationships. The suggested relationships could be presented as a list, where the user will have option to modify and accept the relationships.

Using only the Levenshtein distance for making predictions reduces the complexity of the algorithm, but in practice this method takes longer time. It is naive and will thus make more predictions and in the observed cases there is a larger amount of wrong predictions than correct predictions, as the iterations increases. In conclusion, this does not only make this method slower in terms of performance, but it also more unreliable. The longest observed time for making predictions was approximately 8 ms by using the Levenshtein distance only and approximately 4 ms when combining it with the substring function. Running
the prediction algorithm on larger databases resulted in a decrease in performance. However, by running the application on a server should decrease the time, since the tests were performed on a laptop and thus not optimal.

8.2 Future work
Since the application was single handedly designed and implemented by the author, usability of the application should be measured and evaluated to further improve the application. This could be done by focus groups, questionnaires or usability tests.

Throughout the development of the frontend and backend the only tests that have been utilized are acceptance tests. These tests do not guarantee that the implemented solution works as intended for all cases. Therefore the application should be further evaluated with more thorough tests. Furthermore, the application has only been tested on a couple of devices with the web browser Google Chrome. The application should be tested on a variety of different devices, with different screen sizes and resolutions, together with different browsers to see if it works as intended in all cases.

The application in its current state needs to send many requests to the database due to the behavior of the Ember REST Adapter. In the future the backend server should be improved so that it embeds the required data for each route into one single request, since there is no need for making more requests than necessary since it decreases the performance of the application. The frontend should also be improved such that, for example, a limited amount of rows per table is fetched and not all at once.

The prediction algorithm should be tested on a larger set of databases from different sources, preferably databases that are in a production state. This could provide with useful information regarding the general correctness and performance of the predictions that the algorithm. If the algorithm fails to make correct predictions on a portion of the databases, these could be in-depth analyzed in order to improve the algorithm.
Relational database web application
- Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

References


Relational database web application
- Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

References

Retrieved 2015-05-09

Retrieved 2015-05-09

[12] MSSQL, "Information Schema Views (Transact-SQL)
Retrieved 2015-05-09


http://guides.Ember.JS.com/v1.11.0/concepts/core-concepts/
Retrieved 2015-04-13

[17] Ember.JS, Controllers [www]
http://guides.Ember.JS.com/v1.11.0/controllers/
Retrieved 2015-04-13

[18] Ember.JS, Views [www]
http://guides.Ember.JS.com/v1.11.0/views/
Retrieved 2015-04-13
Relational database web application - Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

References

[19] Ember.JS, Naming Conventions [www]
http://guides.Ember.JS.com/v1.11.0/concepts/naming-conventions/
Retrieved 2015-04-13

http://guides.Ember.JS.com/v1.11.0/models/the-rest-adapter/
Retrieved 2015-04-14


Retrieved 2015-03-05

http://Ember.JS.com/about/
Retrieved 2015-01-20


[27] Restify, "About Restify" [www]
http://restifyjs.com/
Retrieved 2015-05-10

[28] Sequelize, "Sequelize" [www]
http://sequelizejs.com/
Retrieved 2015-02-23
Relational database web application
- Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

References

[29] GitHub, "Sequelize-Auto" [www]
https://github.com/sequelize/sequelize-auto
Retrieved 2015-05-10

https://developer.android.com/design/patterns/index.html
Retrieved 2015-01-19

[31] Apple, "Designing for iOS" [www]
Retrieved 2015-01-19

[32] SIL, "Sil Open Font License" [www]
http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&id=OFL
Retrieved 2015-01-23

[33] Font Awesome, "The Icons" [www]
http://fortawesome.github.io/Font-Awesome/
Retrieved 2015-01-19

[34] yWorks, "yEd Graph Editor" [www]
Retrieved 2015-04-14

[35] Sublime Text, "Sublime Text" [www]
http://www.sublimetext.com/
Retrieved 2015-04-14


[37] MySQL, "MySQL Documentation: Other MySQL Documentation" [www]
http://dev.mysql.com/doc/index-other.html
Retrieved 2015-05-22
Relational database web application
- Web administration interface for visualizing and predicting relationships to manage relational databases
Andreas Hansson

References

[38] MySQL, "Employees Structure" [www]
Retrieved 2015-05-22

[39] MySQL, "License for the Employees Database" [www]
Retrieved 2015-05-22

[40] Creative Commons, "Creative Commons Attribution-Share Alike 3.0 Unported License" [www]
http://creativecommons.org/licenses/by-sa/3.0/
Retrieved 2015-05-22

Retrieved 2015-05-22

Retrieved 2015-05-22

[43] MySQL, "License for the Sakila Sample Database" [www]
Retrieved 2015-05-22

http://opensource.org/licenses/bsd-license.php
Retrieved 2015-05-22

[45] MySQL, "The film Table" [www]
Retrieved 2015-05-23

[46] MySQL, "Setting Up the world Database" [www]
Retrieved 2015-05-22
Appendix A: Screenshots of the implementation

This appendix consists of screenshots from the implementation. A screenshot was taken when table User was active, together with its related table File, see Figure 27.

Figure 27: Screenshot from the application showing two related tables
Appendix A: Screenshots of the implementation

A screenshot was taken showing the menu list together with the tables from Figure 27, see Figure 28.

![Figure 28: Screenshot from the application with the menu list expanded](image-url)