IT 13 087
Examensarbete 15 hp
December 2013

A Bird-watching Database System

Conny Andersson

Abstract

A Bird-watching Database System

Conny Andersson

This report describes in detail the working process of constructing a normalized relational database with an associated graphical user interface, based on raw data data sets in CSV-format acquired from the Avian Knowledge Network. The data files contained the bird sightings from the year 2009 in North America. An entity-relationship model of the database was designed, including a table representing the raw data as well as tables representing the corresponding normalized relational data. As a first step the bulk loading facility of the DBMS is used for loading a CSV file into the raw data table. Then a SQL stored procedure is used for populating the final relational tables by transforming and cleaning the rows of the raw data table. Performance measurements were made about the data transformation as well as a comparison between querying the raw data table versus the final normalized tables. In addition a graphical user interface (GUI) was developed that allows a user to query the database in a flexible way. The performance measurements indicated that querying the normalized tables was more efficient than querying the raw data table.

Handledare: Silvia Stefanova
Ämnesgranskare: Tore Risch
Examinator: Olle Gällmo
IT 13 087
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

The main goal of this project is to implement a bird-watching database system to be used for analysing the developments and trends of bird populations. The current file representation used consists of a file of the type comma-separated values (.csv), which is not an efficient nor a flexible format for a database. Raw data can be complicated and slow to query since information is encoded inside text strings. For example, in the Bird database CSV files each row of a CSV file contains a bird observation. For each bird observation there is a field *species*, which contains a string representing all bird species that have been observed and how many of each species. If such information is stored in the tables of a database as raw data strings the querying becomes very slow, since no indexing on species can be made. This leads to the entire table having to be scanned to find all observations of a species. In addition the queries become clumsy because substring matching has to be used rather than logical conditions.

A relational database should be normalized if it is to be structured for ease of update and querying. The advantages of normalization are that it makes the database simpler to update and avoids redundant data representations [1]. The main four levels of relational database normalization are 1NF (First Normal Form), 2NF (Second Normal Form), 3NF (Third Normal Form), and BCNF. In the project a normalized relational database schema was designed as BCNF, being a high degree of normalization.

The raw CSV data is converted to a format suitable for storage in the normalized relational database. The process of populating the database is illustrated by Figure 1 below. First the CSV file is bulk loaded without any data transformations into a raw data table, called *DBimport*, in the database server. Then the raw data table is processed to clean and convert the data in order to populate the normalized tables. This process is called ETL, (Extract, Transform, Load). In this case the ETL process is implemented as a stored procedure in SQL, which is a program running inside the database server.

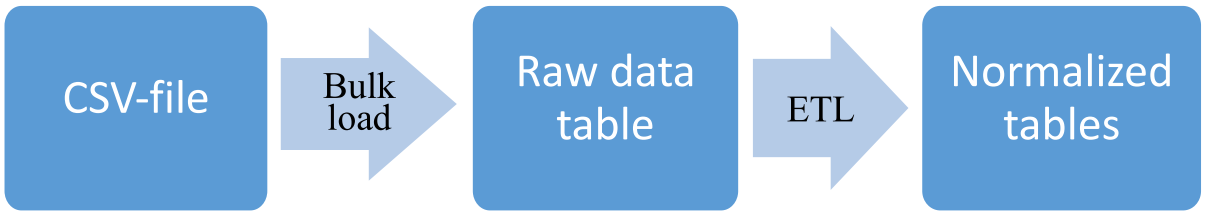


Figure 1: The process of populating the database system

2 Requirements

There were some requirements to be fulfilled for the project to be successful, as listed below. Some of the original requirements were modified and others added during the course of the projects.

1. A relational database should be designed based on an analysis of the bird-watching data found on the website of Avian Knowledge Network [2]. The relational database is to be stored using RDBMS (relational database management system). The design of the database should be general enough to allow extensions later on.
2. The database should be populated with data from Avian Knowledge Networks website using ETL-techniques. The ETL-scripts used to load the database should be written so that data effortlessly can be added at a later time.
3. At least ten SQL queries should be defined to be used to analyse bird populations.
4. The database should be tuned to speed up the execution time of queries, if needed.
5. In order to understand the performance implications of using a normalized database, the performance of queries executed directly on the raw data table should be compared with the newly constructed normalized database.
6. To analyse the performance and tune the database, the performance of a tuned and an untuned database when executing queries should be measured.
7. A GUI should be implemented that demonstrates the functionality of the entire system.
8. To document and analyse the work, a report that satisfactory documents the system should be written. The report should include an overview of related technologies.
9. The implementation of the project should be made with MySQL and Java. It should also be portable to different DBMSs and operating systems.

3 System development

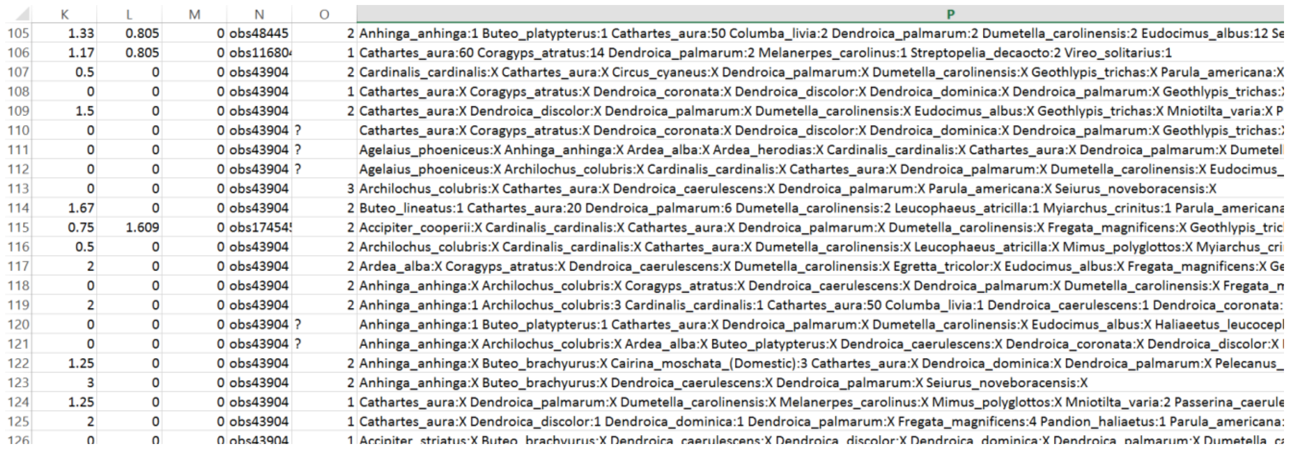
The development of the system was the main part of the project. This chapter is divided into two major parts; design and implementation. The design part describes the steps taken to develop the design of the database system, and the implementation part describes the process of implementing the whole database system.

3.1 Design

The design of the database system was made by constructing an ER-diagram that should fulfil the BCNF normal form. A relational schema was designed including foreign key constraints between tables.

Before designing the database system it was important to first analyse the contents of the CSV file. It is important to understand the information stored in the different fields of the CSV file in order to design the database correctly and be sure not to miss any important data. The detailed explanations of the data stored in each field of the file is shown in Appendix 2. The bird-watching data provided for this project was divided into different CSV files containing the data gathered that particular year. For this project the data from 2009 was used. As can be seen in Figure 2, there are a few issues with the contents of the CSV files. Some of the fields that at first seemingly only contains integers also sometimes have occurrences of '?' in them, indicating unknown values (column O).

In the example shown in Figure 2 the contents of the *species* field (column P) is structured as first the name of the species followed by a colon character and then an integer value that describes the amount of individual birds of that particular species that was observed. A space character is then used to separate the different species and their corresponding values from each other.



	K	L	M	N	O	P
105	1.33	0.805	0	obs48445	2	Anhinga_anhinga:1 Buteo_platypterus:1 Cathartes_aura:50 Columba_livia:2 Dendroica_palmarum:2 Dumetella_carolinensis:2 Eudocimus_albus:12 Se
106	1.17	0.805	0	obs11680	1	Cathartes_aura:60 Coragyps_atratus:14 Dendroica_palmarum:2 Melanerpes_carolinus:1 Streptopelia_decaocto:2 Vireo_solitarius:1
107	0.5	0	0	obs43904	2	Cardinalis_cardinalis:X Cathartes_aura:X Circus_cyaneus:X Dendroica_palmarum:X Dumetella_carolinensis:X Geothlypis_trichas:X Parula_americana:X
108	0	0	0	obs43904	1	Cathartes_aura:X Coragyps_atratus:X Dendroica_coronata:X Dendroica_discolor:X Dendroica_dominica:X Dendroica_palmarum:X Geothlypis_trichas:
109	1.5	0	0	obs43904	2	Cathartes_aura:X Dendroica_discolor:X Dendroica_palmarum:X Dumetella_carolinensis:X Eudocimus_albus:X Geothlypis_trichas:X Mniotilta_varia:X P
110	0	0	0	obs43904	?	Cathartes_aura:X Coragyps_atratus:X Dendroica_coronata:X Dendroica_discolor:X Dendroica_dominica:X Dendroica_palmarum:X Geothlypis_trichas:
111	0	0	0	obs43904	?	Agelaius_phoeniceus:X Anhinga_anhinga:X Ardea_alba:X Ardea_herodias:X Cardinalis_cardinalis:X Cathartes_aura:X Dendroica_palmarum:X Dumetel
112	0	0	0	obs43904	?	Agelaius_phoeniceus:X Archilochus_colubris:X Cardinalis_cardinalis:X Cathartes_aura:X Dendroica_palmarum:X Dumetella_carolinensis:X Eudocimus_
113	0	0	0	obs43904	3	Archilochus_colubris:X Cathartes_aura:X Dendroica_caerulescens:X Dendroica_palmarum:X Parula_americana:X Seiurus_noveboracensis:X
114	1.67	0	0	obs43904	2	Buteo_lineatus:1 Cathartes_aura:20 Dendroica_palmarum:6 Dumetella_carolinensis:2 Leucophaeus_atricilla:1 Myiarchus_crinitus:1 Parula_americana
115	0.75	1.609	0	obs17454	2	Accipiter_cooperii:X Cardinalis_cardinalis:X Cathartes_aura:X Dendroica_palmarum:X Dumetella_carolinensis:X Fregata_magnificens:X Geothlypis_tric
116	0.5	0	0	obs43904	2	Archilochus_colubris:X Cardinalis_cardinalis:X Cathartes_aura:X Dumetella_carolinensis:X Leucophaeus_atricilla:X Mimus_polyglottos:X Myiarchus_cri
117	2	0	0	obs43904	2	Ardea_alba:X Coragyps_atratus:X Dendroica_caerulescens:X Dumetella_carolinensis:X Egretta_tricolor:X Eudocimus_albus:X Fregata_magnificens:X Ge
118	0	0	0	obs43904	2	Anhinga_anhinga:X Archilochus_colubris:X Coragyps_atratus:X Dendroica_caerulescens:X Dendroica_palmarum:X Dumetella_carolinensis:X Fregata_n
119	2	0	0	obs43904	2	Anhinga_anhinga:1 Archilochus_colubris:3 Cardinalis_cardinalis:1 Cathartes_aura:50 Columba_livia:1 Dendroica_caerulescens:1 Dendroica_coronata:
120	0	0	0	obs43904	?	Anhinga_anhinga:1 Buteo_platypterus:1 Cathartes_aura:X Dendroica_palmarum:X Dumetella_carolinensis:X Eudocimus_albus:X Haliaeetus_leucocepl
121	0	0	0	obs43904	?	Anhinga_anhinga:X Archilochus_colubris:X Ardea_alba:X Buteo_platypterus:X Dendroica_caerulescens:X Dendroica_coronata:X Dendroica_discolor:X
122	1.25	0	0	obs43904	2	Anhinga_anhinga:X Buteo_brachyurus:X Cairina_moschata_(Domestic):3 Cathartes_aura:X Dendroica_dominica:X Dendroica_palmarum:X Pelecanus_
123	3	0	0	obs43904	2	Anhinga_anhinga:X Buteo_brachyurus:X Dendroica_caerulescens:X Dendroica_palmarum:X Seiurus_noveboracensis:X
124	1.25	0	0	obs43904	1	Cathartes_aura:X Dendroica_palmarum:X Dumetella_carolinensis:X Melanerpes_carolinus:X Mimus_polyglottos:X Mniotilta_varia:2 Passerina_caerule
125	2	0	0	obs43904	1	Cathartes_aura:X Dendroica_discolor:1 Dendroica_dominica:1 Dendroica_palmarum:X Fregata_magnificens:4 Pandion_haliaetus:1 Parula_americana:
126	0	0	0	obs43904	1	Accipiter_striatus:X Buteo_brachyurus:X Dendroica_caerulescens:X Dendroica_discolor:X Dendroica_dominica:X Dendroica_palmarum:X Dumetella_ca

Figure 2: An example of the contents of the CSV file

One of the goals when designing this database was to make it BCNF which stands for Boyce-Codd Normal Form. This is done to make the database more flexible when it comes to updates, and to reduce redundancy in the database. A downside of BCNF is that it can in some cases make the database slower since more joins are needed in particular queries. As a means to achieve BCNF an Entity-Relationship (ER) -diagram was created, which is shown in Figure 3. The ER-diagram improves the understanding of the data that is to be stored in the database, and it will also show a general view of how the database is going to be structured when it is implemented [3].

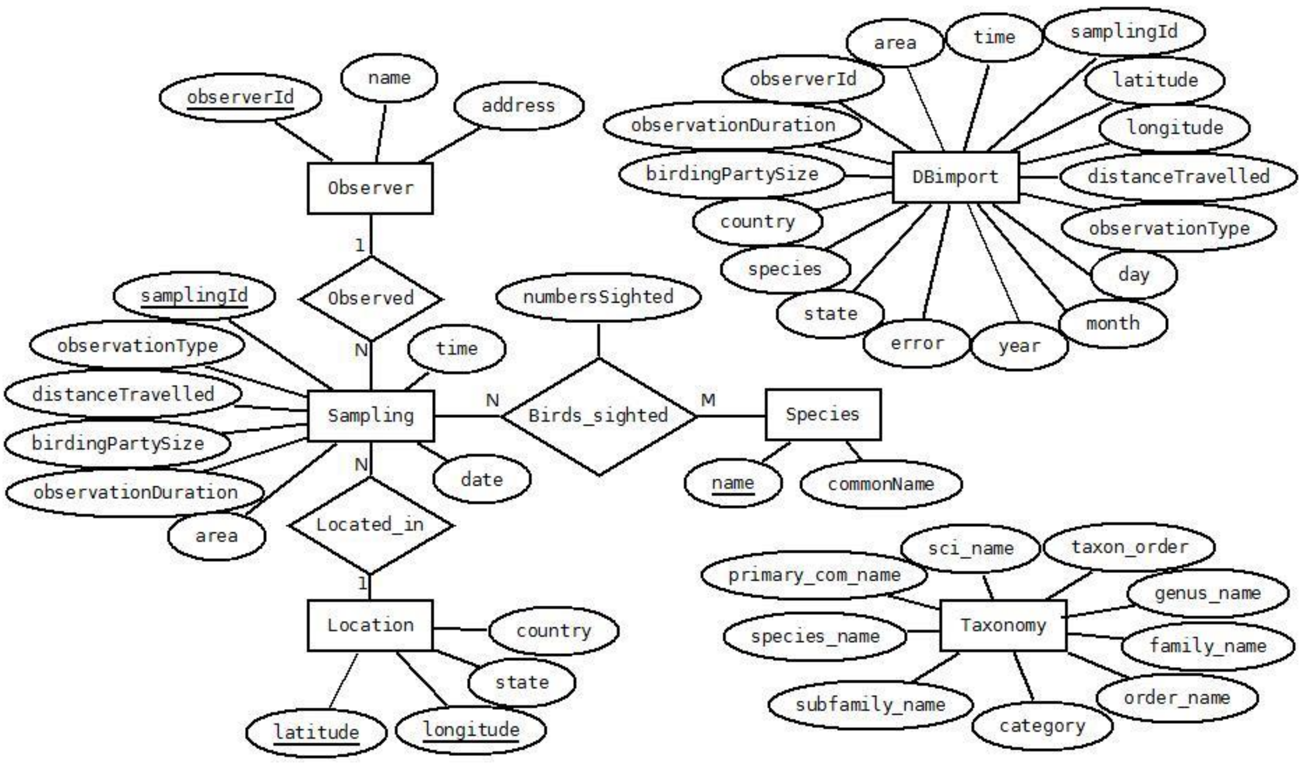


Figure 3: The ER-diagram of the database

The explanation of the attributes of the raw data tables *DBimport* and *Taxonomy,* depicted as islands in the ER-diagram above, can be viewed in Appendix 2 and Appendix 3. There are four attributes in the ER-diagram that are not covered in those appendices, since they have been extracted from other attributes or added for eventual future usage. These attributes are *date, numbersSighted, name* and *address* (*name* of the Species table is covered by *sci_name* attribute of Taxonomy table). It should also be noted that the *error* attribute of the *DBimport* table was added later in order to register errors that can't be handled in a proper way when data is loaded.

The *date* attribute is the attributes *year, month* and *day* merged into one attribute of the *date* SQL data type. The *numbersSighted* attribute belonging to the *Birds_sighted* relationship describes how many birds of each species was sighted during a certain observation. The *Observer* table contains information about the people who made the bird observation, i.e. *name* and *address*. Worth noting is that there is no data to insert regarding the names and addresses of the observers at this time, but these attributes were chosen to be part of the design anyway to facilitate the potential need for these attributes at a later stage.

As a step in the process of designing a relational database it is necessary to design a relational schema. A relational schema is different from ER-diagrams in the way that it also shows which attributes are primary key of each table, as well as showing the foreign key constraints between the different tables. The relational schema is shown below in Figure 4, where primary keys are underlined and *FK* indicates foreign keys.

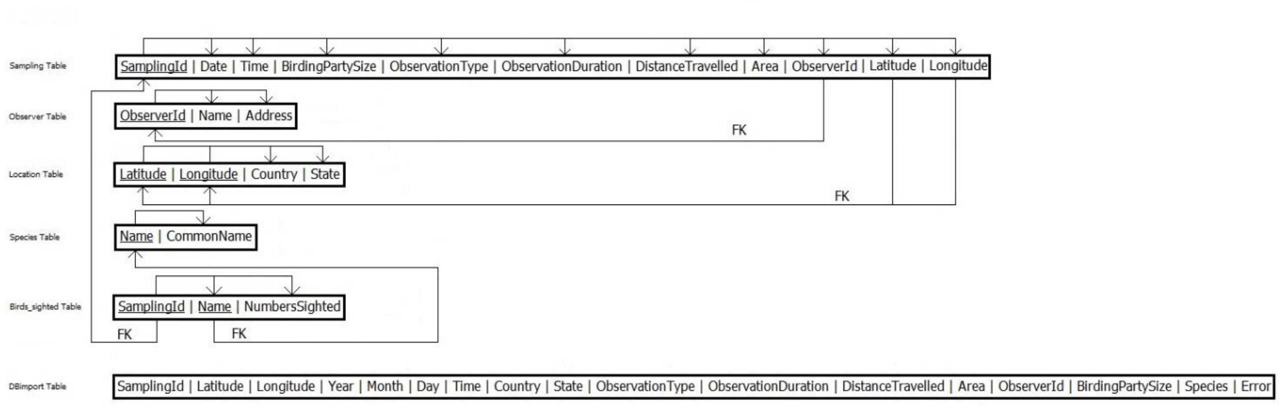

Sampling Table
SamplingId | Date | Time | BirdingPartySize | ObservationType | ObservationDuration | DistanceTravelled | Area | ObserverId | Latitude | Longitude
Observer Table
ObserverId | Name | Address
FK
Location Table
Latitude | Longitude | Country | State
FK
Species Table
Name | CommonName
Birds_sighted Table
SamplingId | Name | NumbersSighted
FK
FK
DBimport Table
SamplingId | Latitude | Longitude | Year | Month | Day | Time | Country | State | ObservationType | ObservationDuration | DistanceTravelled | Area | ObserverId | BirdingPartySize | Species | Error

Figure 4: The relational schema of the database

3.2 Implementation

The implementation of this project is divided into three major parts: the software to populate the database, the implementation of the required data transformations into normalized tables, and the construction of the GUI. A data parser was needed as a tool to help split unnormalized strings in the *species* column into atomic values to be inserted into the tables of the database. The SQL stored procedure code snippets referred to in the text, as well as all written code can be found in Appendix 1.

3.2.1 Population of the database

The process of populating the database was to first bulk load the data from the CSV file into the temporary table *DBimport*, which is used as an intermediate storage where the cleaning of the data takes place to populate the tables of the normalized database. The cleaning process consists of two steps:

1. The first step is to correct flawed data, for example changing occurrences of question marks, indicating that there is no data for a particular table cell, to null.

2. The second step is to report and mark any rows that are erroneous in some way.

3.2.1.1 Bulk loading into the temporary table

The data was imported into the database without any data transformations using the MySQL bulk load mechanism LOAD DATA INFILE [4]. The specific statement used in this project is shown in Appendix 1. The statement takes the data in the CSV file where the tables columns are separated based on a chosen field separator (here ',') and loads it directly into the table specified in the statement. A problem was that MySQL was originally running in InnoDB Strict Mode [5] [6], which does automatic data type validity checks of inserted values as a guard against populating the database with illegal values. All rows containing illegal data are raising errors that stops the loading. There were two different approaches to solve this issue: one is to in a pre-processing step use another programming language, suited for parsing strings, to check the CSV file for erroneous values and correct them before loading the CSV file into the tables of the database, and the other is to disable strict mode and handle the issues at a later stage. The latter option mentioned was chosen to solve this task, since it is easier to correct the problems when data is stored in a temporary table than having to create a program in another programming language that could eventually be very time consuming to implement.

It originally took 500 seconds to import a CSV file that contained roughly 450 000 entries, which raised some suspicion whether it could execute a lot faster. The course of action taken to speed up the execution of the statement was to increase the value of the buffer pool in the settings of the DBMS. This means that more of the computer's RAM is used as a buffer when doing the bulk load, which leads to less read/writes to disk and therefore provides a faster execution time. By now the query would execute properly but it skipped five rows due to violations of the primary key constraint in the temporary table *DBimport*. The reason for this was that the contents of the column *samplingId* was expected to be unique, which was not the case. Because of this it was decided that there is no point to have a primary key attribute in *DBimport*, since errors such as non-unique *samplingId* values will be handled later in the process anyway. After dropping the primary key constraint in *DBimport* the query executed successfully after 32 seconds.

3.2.1.2 Populating the normalized database with the content of the temporary table

The population of the *Observer* table was done using a simple straight forward insertion of all distinct *observerId* attribute values from the temporary table *DBimport* into the *Observer* table, using the query "INSERT INTO Observer (observerId) SELECT DISTINCT observerId FROM DBimport;".

While trying to populate the *Location* table with the *latitude* and *longitude* values from the temporary table, an issue revealed itself. The latitude and longitude values were imported into a column in the temporary table that only allows values of the type INT, which led to the decimals being cut off. As a solution to this problem all the tables in the database system that had the *latitude* and *longitude* columns had to be changed so that the type of the columns was DECIMAL(15, 12) instead. The numbers used as arguments in this type declaration decides that the maximum number of total digits is 15, and 12 of those 15 digits are the decimals to the right of the decimal point [7]. These values were chosen because many decimals are needed to make the latitude- and longitude values exact enough, while latitude values fit inside the interval (-90,90) and longitude inside the interval (-180,180), so the integer part doesn't need to be bigger than three integers.

The temporary table *DBimport* now contained erroneous values in the *latitude* and l*ongitude* fields. This was solved by deleting all contents of the *DBimport* table by executing the command TRUNCATE TABLE, which performs a fast deletion of the contents of a table by dropping the table completely and recreating it again [8]. The CSV file was then bulk loaded into *DBimport* again so that the complete latitude and longitude values would be included. When that was done the

Location table could get populated with the values of *latitude*, *longitude*, *country* and *state* from the temporary table.

The original CSV file contained the date of the sampling in the form of three columns, one for the year, one for the month and one for the day of the year. Since this is not only unnecessarily cluttering but also a waste of memory it was decided that one column with the type DATE would be used in the new database. Fortunately MySQL provides a function called MAKEDATE that takes the year and the day of the year as arguments and returns the date in DATE format [9].

The time and the duration of the sampling had to be manipulated as well, since they were in decimal form. In this case the number 12.75 was used to represent the time 12:45 for example. This issue was solved by, in the INSERT query multiplying the *time* value with 3600 to get the time in seconds, and then use the function SEC_TO_TIME with the previous value as argument to get the time in TIME format.

There was a CSV file called *Taxonomy* among the other files retrieved from the Avian Knowledge Network. The most relevant contents of this file, such as the birds Latin names and their English names, was imported into the *Species* table of the database using a query very similar to the one used to populate the *DBimport*.

There were also occurrences of '?' in the columns *birdingPartySize*, *observationDuration*, and *time*, of the original CSV file, which are not allowed in an INT- or TIME-type column. These were replaced by *null* instead. However a column of type TIME does not allow null values at default, so this had to be changed before it was possible to replace the question marks with null. The solution to this was to drop the *Sampling* table and set the *time* attribute to have *null* as default value when the *Sampling* table was recreated, thereby forcing the *time* attribute to accept null values.

The only task left to be done on the implementation of the database was to populate the *Sampling* table with the data from *DBimport* including the adjusted values of *date* and *time* mentioned earlier. With the database completely populated the foreign keys of the tables were added to the system afterwards and the whole database was now entirely implemented.

3.2.2 Constructing the stored procedures for the parser

The populating of the *Birds_sighted* table turned out to be very complicated, which led to the decision of implementing a parser that could divide the *species* strings into parts small enough to be inserted into the *Birds_sighted* table. This parser was implemented using two stored procedures: one that parses strings, splits them and inserts them into the *Birds_sighted* table, and one that provides the other stored procedure with strings extracted from the *species* column of *DBimport*.

The *Birds_sighted* table contains the columns *samplingId*, *species* and *numbersSighted* (the number of individuals of a species that was sighted during an observation). The strings in the *species* column of *DBimport* contain all this information as one text string that in some cases is very long. The strings need to be split into appropriate pieces by parsing them, before they can be inserted into the *Birds_sighted* table. The algorithm for doing this is outlined below where the long text string is referred to as *S*. See Appendix 2 or Figure 2 for examples of the contents of the *species* column in the temporary table *DBimport*.

1. Split the string *S* into two substrings based on the first encountered space. The first part is now called *S_first* and the second *S_rest*, where *S_first* is a string containing a sighted bird species followed by a colon followed by the number of sighted birds belonging to that species, *S_rest* is the rest of the original string *S*.

2. Split the string *S_first* at the colon and call the first part of it *S_first_species* and the second *S_first_number*.

3. The string *S_first_species* now contains the species and the string *S_first_number* contains the number of birds. Insert into the *Birds_sighted* table a row containing the id of the sampling, the species in *S_first_species* and the number of observations in *S_first_number*. If the string *S_rest* is not empty, go to step 1 with *S* assigned to *S_rest*.

This algorithm was implemented as a stored procedure, in MySQL, called *Parsetext*. It uses a function that was created for the purpose of splitting these strings, called *Split_str(str, char, int)* that takes a string *str* (to be split), a character *char* (which is used to decide where the string is split) and an integer *int* (that decides if it's the first (1) or the second (2) part of the string that is to be returned) as arguments. It splits the string into two at a certain characters position and returns either the first or the second part of the string without the character that acted as splitting point. Both the stored procedure *Parsetext* and the function *Split_str* can be viewed in Appendix 1.

However, it was found that this parser was not as efficient as it could be and therefore a new parser using the following algorithm was implemented, also using a stored procedure. This algorithm instead builds up strings by parsing the original string character by character. If it encounters a colon character, ":", it is known that the string that has been built-up so far is the name of a species, and if it encounters a space, " ", the built-up string is in fact the number of individuals sighted of that particular species. In the algorithm depicted below the long text string is referred to as *B*. The variable *counter_val* is initialized to the integer 1. Set the variable *length_val* to be the amounts of strings in *B* that are separated by spaces (' ').

1. Extract the substring between the *counter_val*th and the *counter_val*-1th occurrence of space (' ') in *B* and call it *cur_str*.
2. Split *cur_str* at the point of the colon (':') and insert the first part into *name_val* and the second part into *number_val*.
3. If *number_val* is 'X' then insert null into the variable *number_val_int* else insert the value of *number_val* into *number_val_int*.
4. Insert the values of *name_val*, *number_val_int,* and the *samplingId* into the *Birds_sighted* table.
5. If *length_val* is not zero then increase *counter_val* by one and decrease *length_val* by one and go back to point 1, else exit.

Another stored procedure, *Looptable*, was used to help with the population of the *Birds_sighted* table, since the stored procedure *Parsetext* needs to be called for every row in the temporary table. This was done by using a cursor in *Looptable* that for each row in the temporary table *DBimport* calls *Parsetext*. This stored procedure can be viewed in Appendix 1.

A major setback in this project was the performance of the stored procedure *Looptable*. There existed some suspicion that the execution of it could take quite a while, so it was decided to try to execute this stored procedure for only the first ten rows of *DBimport*. The result of the execution of the stored procedure was satisfying except the fact that it took 12 seconds to finish for only ten rows, which is unsatisfactory. Some simple calculations show that it would take more than six days effective time to execute the query with the full table, which contains roughly 450 000 rows. This was considered to be too long because of the fact that there is a chance something goes wrong on the fifth day, which could lead to five days of execution being wasted. Therefore a few more options were explored; one was to export the results to a CSV file by using the SELECT INTO OUTFILE query and then bulk-load it back, but this was rejected because in each loop in *Parsetext* the query would be called and each query results in a new file being created, so at the end of this there would be an extreme amount of files on the computer hard drive potentially using up all the memory. The other option was to use the statement TEE to print all the results from the queries to a file, but this would not work either, since the TEE statement prints every single query and statement, so every select-query and variable assignment would be printed to the file, which would lead to the major part of the file consisting of unnecessary data.

As the stored procedure *Looptable* was run using the MySQL Workbench, which is a visual tool for designing and maintaining databases, a few issues arose. One of these issues was that the query ended if the execution time exceeded 600 seconds, even though the time limit was changed to 0, which is interpreted as infinity, it still wouldn't run past the 600 second limit. Therefore *Looptable* was executed in the windows command prompt-environment instead, where there are no limits on the execution time.

Another issue was that *Looptable* stopped after about an hour with the error 'ERROR 1172 (42000): Result consisted of more than one row'. To see on which row of the temporary table the error occurred on, a print statement was added in *Looptable* that printed which row was currently being processed. This showed that the last row being processed was row 16944 in the temporary table *DBimport*. After some time it was discovered that the source of this error was that two different entries had the same *SamplingId*, which should never happen since this is supposed to be a unique identifier. This issue was solved by adding the attribute *error* in the *DBimport* table, which is a flag to decide if the row should be skipped because it can't be handled at this moment. Then it was easy to just exclude the rows that contained errors when, in the stored procedure *Looptable,* choosing which rows that should be processed by the stored procedure *Parsetext*. The values of these rows was also excluded from being inserted into the normalized tables. With these issues resolved the stored procedure finished successfully after 13 hours execution time, which is a better time than earlier estimated.

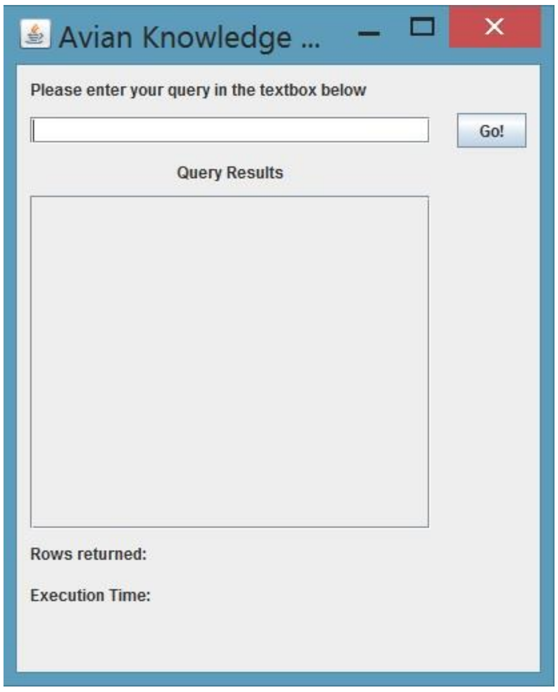
3.2.3 GUI implementation

The GUI was implemented using Java (JavaSE-1.7), Swing and the JDBC (Java DataBase Connectivity) driver Connector/J (version 5.1.23). By using these it is possible to execute queries to the MySQL database through a Java GUI, which is exactly what was sought to achieve in this project.

As a first step a simple connection from Java to the MySQL database was established, and a SHOW TABLES query was executed with the result of the query shown in the console. With this important functionality implemented the programming of the GUI was started. The major design choice of the GUI-implementation was the decision to use GridBagLayout[10], which basically just divides the application-window into a grid where different graphical objects can be placed inside or span across different grid-squares.

The basic idea of the GUI is to have a text box where the user enters the query to be executed once the button beside it is pressed. The result of the query will be displayed as a table, and there will also be a number of labels that will display relevant information like execution time and eventual error messages.

To successfully implement this design the following Swing objects was used; a JTextField for the users query, JButton for the query to be executed, JTable with a JScrollPane for the query results and some JLabels for various information. The final appearance of the GUI is shown in Figure 5, Figure 6 and Figure 7. Figure 5 shows the start window of the application, Figure 6 shows how the application looks after a syntax error in the query, and Figure 7 displays the result of a successful query to the database.



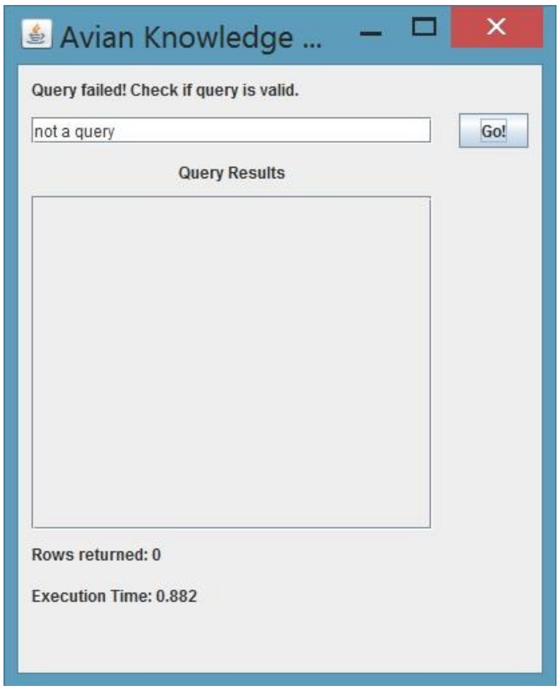
Please enter your query in the textbox below

Query Results

Rows returned:

Execution Time:

Figure 5: The start window



Query failed! Check if query is valid.

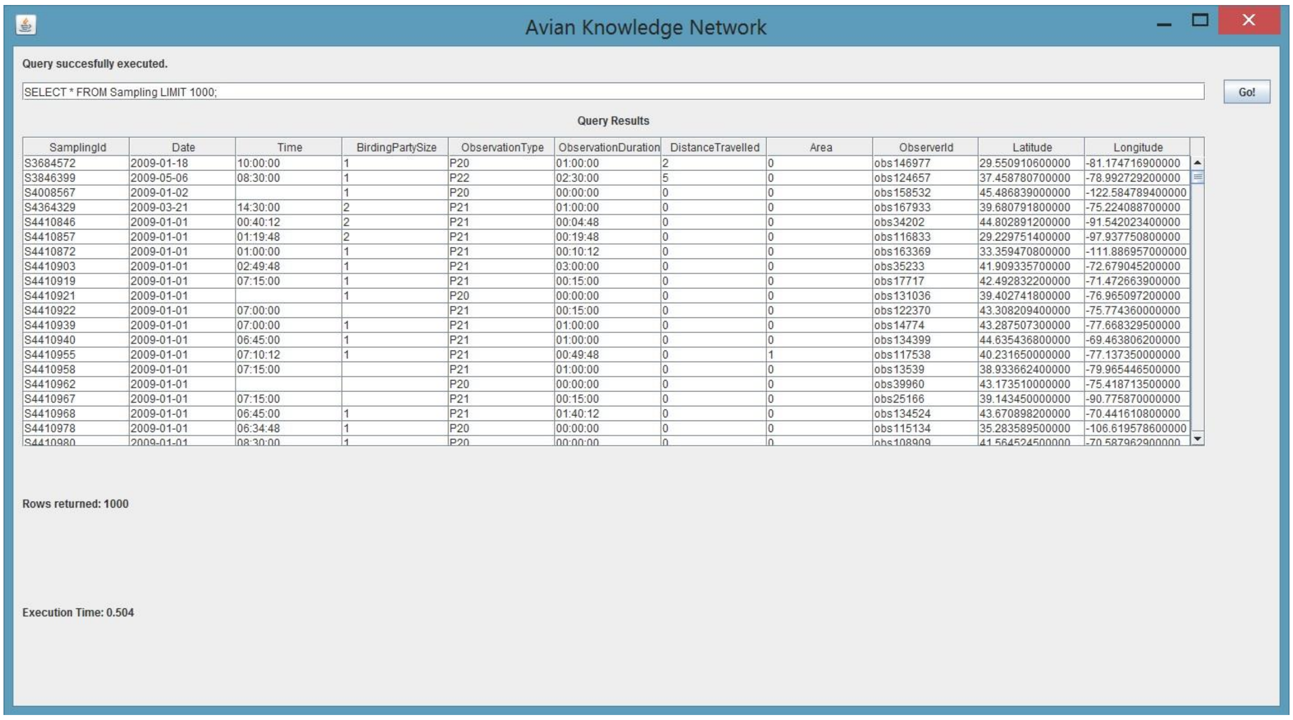
not a query

Query Results

Rows returned: 0

Execution Time: 0.882

Figure 6: Example of a failed query



Query succesfully executed.

```
SELECT * FROM Sampling LIMIT 1000;
```

Query Results

SamplingId	Date	Time	BirdingPartySize	ObservationType	ObservationDuration	DistanceTravelled	Area	ObserverId	Latitude	Longitude
S3684572	2009-01-18	10:00:00	1	P20	01:00:00	2	0	obs146977	29.550910600000	-81.174716900000
S3846399	2009-05-06	08:30:00	1	P22	02:30:00	5	0	obs124657	37.458780700000	-78.992729200000
S4008567	2009-01-02		1	P20	00:00:00	0	0	obs158532	45.486839000000	-122.584789400000
S4364329	2009-03-21	14:30:00	2	P21	01:00:00	0	0	obs167933	39.680791800000	-75.224088700000
S4410846	2009-01-01	00:40:12	2	P21	00:04:48	0	0	obs34202	44.802891200000	-91.542023400000
S4410857	2009-01-01	01:19:48	2	P21	00:19:48	0	0	obs116833	29.229751400000	-97.937750800000
S4410872	2009-01-01	01:00:00	1	P21	00:10:12	0	0	obs163369	33.359470800000	-111.886957000000
S4410903	2009-01-01	02:49:48	1	P21	03:00:00	0	0	obs35233	41.909335700000	-72.679045200000
S4410919	2009-01-01	07:15:00	1	P21	00:15:00	0	0	obs17717	42.492832200000	-71.472663900000
S4410921	2009-01-01		1	P20	00:00:00	0	0	obs131036	39.402741800000	-76.965097200000
S4410922	2009-01-01	07:00:00		P21	00:15:00	0	0	obs122370	43.308209400000	-75.774360000000
S4410939	2009-01-01	07:00:00	1	P21	01:00:00	0	0	obs14774	43.287507300000	-77.668329500000
S4410940	2009-01-01	06:45:00	1	P21	01:00:00	0	0	obs134399	44.635436800000	-69.463806200000
S4410955	2009-01-01	07:10:12	1	P21	00:49:48	0	1	obs117538	40.231650000000	-77.137350000000
S4410958	2009-01-01	07:15:00		P21	01:00:00	0	0	obs13539	38.933662400000	-79.965446500000
S4410962	2009-01-01			P20	00:00:00	0	0	obs39960	43.173510000000	-75.418713500000
S4410967	2009-01-01	07:15:00		P21	00:15:00	0	0	obs25166	39.143450000000	-90.775870000000
S4410968	2009-01-01	06:45:00	1	P21	01:40:12	0	0	obs134524	43.670898200000	-70.441610800000
S4410978	2009-01-01	06:34:48	1	P20	00:00:00	0	0	obs115134	35.283589500000	-106.619578600000
S4410980	2009-01-01	08:30:00	1	P20	00:00:00	0	0	obs108909	41.564524500000	-70.587962900000

Rows returned: 1000

Execution Time: 0.504

Figure 7: Example of a successful query

4 Evaluation

To evaluate this project and its results a few performance measurements has been performed. Two performance tests was run, the results from them are shown in Table 1 and the diagram in Figure 8. One of the tests, illustrated as *Parser* in the table and diagram, was to run the stored procedure that parses the *species* string and inserts it into the correct table, on different amount of records (rows in the temporary table). The other test, illustrated as *Populating* in the table and diagram, was to populate the rest of the database, on different amount of records from the temporary table. These performance measurements was made to illustrate which part of populating the database that was the most time consuming. The column *Records* references to the rows of the temporary table, which contains all the original data from the CSV file.

Records * 10^3	Parser (min)	Populating (min)
0	0	0
1	0.4338	0.05155
5	3.4965	0.065083
10	6.6155	0.083316
15	9.7236	0.09275
20	13.3696	0.1172
50	40.3418	0.155483
100	79.9	0.3165
150	120.4	0.50936
200	160.9	0.72445
250	201.4	0.822383
300	241.9	1.54793
350	282.4	1.822383
400	322.9	1.91953
450	363.4	2.26273

Table 1: Performance measurements

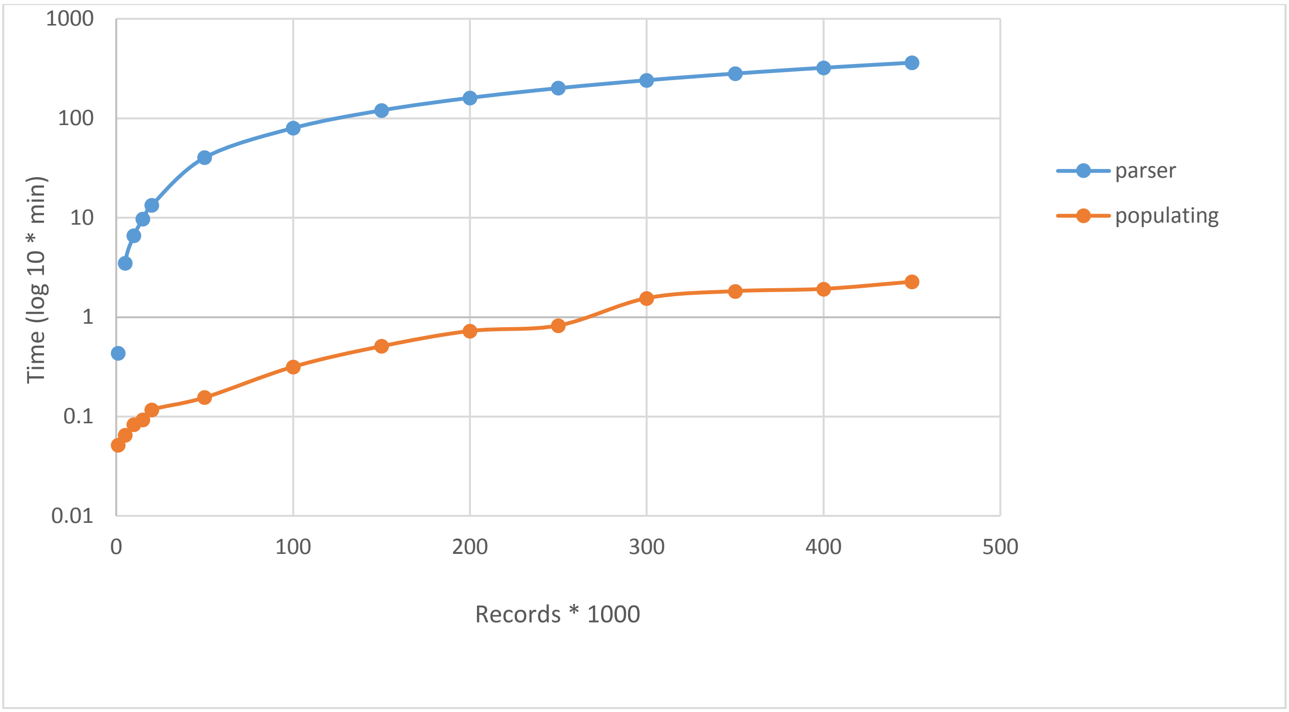


Figure 8: Performance measurements

By looking at Figure 8 it is easy to see that the time consuming step of the complete task of populating the database is the parsing of the *species* strings. The reason why the execution time of the parsing is so much higher than the execution time of populating the rest of the database is that for each row in the temporary table the parser needs to go through each character of the string, stored in the column *species*, one-by-one in a means to split it up into the correct pieces so the data can be inserted in the *Birds_sighted* table accurately. These strings can be as long as 4200 character in the particular CSV file that was used in this project, which means that in most cases the parsing will have much more time-consuming task on its hands than the task of just populating the tables with values that can easily be extracted from the temporary table.

Another performance test was made to evaluate if the normalized database is more efficient to query than the unnormalized raw data table. This was done by timing the same five queries on each of the two databases, both when the database was warm and cold, and comparing the results. A database is said to be cold when the cache has recently been cleared, and warm when the cache has data from previous queries in it. The five queries that the databases were tested on are showed below, and the results are shown in Table 2.

1.
   ```
   SELECT COUNT(*) FROM DBimport WHERE species LIKE '%Mycteria_americana%';
   SELECT COUNT(*) FROM Birds_sighted WHERE name = 'Mycteria_americana';
   ```
   This query is used to find out on how many occasions this particular species of birds has been sighted.

2.
   ```
   SELECT AVG(time) FROM DBimport;
   SELECT AVG(HOUR(time)) FROM Sampling;
   ```
   This query shows the average time of all the bird observations in the table.
   (Note that these queries will return different values since the *DBimport* table has '?' in the *time* field, while the normalized table *Sampling* will have null. When executing the query "SELECT AVG('?');" the return value will be zero, so these zero-values will be counted in when the query is run on *DBimport*. The *Sampling* table will disregard these values since they were changed to null before being inserted into *Sampling*).

3.
   ```
   SELECT COUNT(*) FROM DBimport WHERE species LIKE '%Catharus_ustulatus%'
   AND month = 5;
   SELECT COUNT(*) FROM Birds_sighted AS c, Sampling AS s WHERE c.samplingId =
    s.samplingId AND c.name = 'Catharus_ustulatus' AND MONTH(s.date) = 5;
   ```
   This query returns the amount of observation of the bird species *Catharus ustulatus* during the month of May.

4.
   ```
   SELECT COUNT(*) FROM DBimport WHERE country = 'United_States' AND state =
   'Kentucky' AND month = 6;
   SELECT COUNT(*) FROM Location AS l, Sampling AS s WHERE l.country =
   'United_States' AND l.state = 'Kentucky' AND s.latitude = l.latitude AND s.longitude =
   l.longitude AND MONTH(s.date) = 6;
   ```
   This query displays the amount of bird observations made in Kentucky, USA during June.

5.
   ```
   SELECT samplingId, latitude, state FROM DBimport WHERE latitude =
   (SELECT MAX(latitude) FROM DBimport);
   SELECT s.samplingId, l.latitude, l.state FROM Sampling AS s, Location AS l WHERE
   s.latitude = (SELECT MAX(latitude) FROM Location) AND s.latitude = l.latitude AND
   s.longitude = l.longitude;
   ```
   This query shows in which state the northernmost bird observation took place.

In Table 2 below, the term *unnormalized raw data table* refers to using *DBimport* to store the whole content of the original CSV file. The *normalized database* refers to using the normalized tables populated with the data from the original CSV file. Cold and warm database denotes whether or not the cache of the DBMS is empty (cold) or if it's filled with data from previous queries (warm).

Query	**Normalized database, cold (sec)**	**Unnormalized raw data table, cold (sec)**	**Normalized database, warm (sec)**	**Unnormalized raw data table, warm (sec)**
1	0.016	2.234	0.000	1.562
2	1.657	1.437	0.156	0.282
3	0.844	1.422	0.015	0.453
4	0.609	1.422	0.062	0.250
5	0.016	1.750	0.000	0.594

Table 2: Performance measurements between the normalized database and the unnormalized raw data table, with and without cache

These results illustrates the fact that the normalized database is faster than the unnormalized raw data table on all five queries when executed on a warm database, and it is also faster in the majority of the cases when the query is executed on a cold database. This reflects that the normalized database is more time-efficient, on average, than queries to the unnormalized raw data table, no matter if the results are cached or not. The reason for it being more time-efficient is that if a query containing the MAX()-function is executed on an unnormalized raw data table, all data needs to be scanned to obtain the result requested, since all information is in one table. If the same query is executed on a normalized database, which in this case consists of a few more but smaller tables, there is a big chance that only one of the small tables needs to be scanned, which will be much less time-consuming. Another scenario that works in favour of the normalized database is if a user wants to know the maximum amount of birds sighted at one moment and of what species, because then only the *Birds_sighted* table, which is indexed on the *numbersSighted* column, have to be scanned for its maximum value. While if the same information is to be extracted from the *DBimport* table, each *species* string needs to be parsed to find the maximum value, which is an unnecessarily complex and time-consuming query to perform.

5 Related work

Microsoft Access [11] has a feature called Table Analyzer [12], which can help a user normalize a database by dividing tables that contains repeating information into separate tables where each type of information is stored only once.

Another software that can assist with normalizing databases is NORMA [13], which is a conceptual modelling tool. NORMA is a free and open source plug-in for Microsoft Visual Studio [14] (versions 2005, 2008, 2010, and 2012).

There is also software that can aid in the task of loading a CSV file into a database. One of these is SSIS [15] (SQL Server Integration Services). An important thing to note is that this software is for Microsoft SQL Server, but it is still relevant since it can connect to MySQL using the Connector/Net driver. With the software a user can load a CSV file into a database using a graphical user interface with many different features, such as tools to transform data.

Mysqlimport [16] is another software that can be used to import CSV files into a database. It is a command-line interface for the SQL statement LOAD DATA INFILE, which basically means that the options the user selects in mysqlimport relates directly to different clauses inside the LOAD DATA INFILE syntax. The mysqlimport software doesn't have any features for parsing or cleaning data.

MySQL Workbench [17] also provides a function with which the user can load a CSV file into a database. However this was tried during this project without success, which was the reason that the LOAD DATA INFILE statement was used instead.

6 Conclusions and future work

My work in this project resulted in a normalized database system with the purpose of storing, retrieving or updating information about bird observations acquired by the Avian Knowledge Network.

In this project I have contributed with a robust and scalable database design that will greatly improve the usability of the data in the CSV file. This design was then used when the database was constructed, populated and later optimized. It is because of this design that the final results of the project were so positive.

I have also spent a lot of time and effort into constructing a runnable script that can be used to construct and populate a database with the contents of a CSV file. This script is written in a very general way, therefore it is easy to change the script if one would need to use it on a CSV file that is structured in a slightly different way.

When the database was implemented a series of tests was performed on it to see if it was up to standards. These tests displayed positive results, such as querying the normalized database being faster than querying the raw data table *DBimport*. This indicates that all the effort I put into this project was not in vain.

One could argue that there is a better solution to the problem with the unnecessarily long execution time of the stored procedure *Parsetext*, for example to process the *species* rows in another language that is faster than MySQL, since it doesn't seem to be very effective. I believe that if the *species* rows instead were to be exported and processed using a faster programming language such as Pearl, Java or C, and then exported to a CSV file and later bulk loaded into the database, the overall execution time would be a lot shorter. However, the fact that the parser was shown to be the most time consuming step of the process does not mean that this project was unsuccessful, since bulk loading data that has to be parsed and cleaned is done a lot less frequently than querying the database. As a whole, the requirements of this project have been met.

References

[1] Stephens K, Ryan; Plew R, Ronald, *Database Design,* 2001, s 206, Sams Publishing, Indianapolis, Indiana, USA, ISBN: 0-672-31758-3

[2] “Avian Knowledge Network” http://www.avianknowledge.net/ (2014-02-19 14:32)

[3] “ER-modeling” http://www.databasedesign.co.uk/bookdatabasesafirstcourse/chap3/chap3.htm Chapter 3.1 (2014-02-11 10:35)

[4] “LOAD DATA INFILE Syntax” http://dev.mysql.com/doc/refman/5.1/en/load-data.html (2013-11-19 18:16)

[5] “InnoDB Strict Mode” http://dev.mysql.com/doc/innodb/1.1/en/innodb-other-changes-strict-mode.html (2013-11-08 11:31)

[6] “InnoDB Storage Engine” http://dev.mysql.com/doc/refman/5.5/en/innodb-storage-engine.html (2013-11-08 11:32)

[7] “MySQL Documentation, decimal” http://dev.mysql.com/doc/refman/5.1/en/precision-math-decimal-changes.html. (2013-07-11 19:38)

[8] “Truncate table statement” http://dev.mysql.com/doc/refman/5.0/en/truncate-table.html (2014-02-12 09:32)

[9] “MySQL Documentation, MAKEDATE()” http://dev.mysql.com/doc/refman/5.6/en/date-and-time-functions.html#function_makedate. (2013-07-12 12:25)

[10] “GridBagLayout documentation” http://download.java.net/jdk7/archive/b123/docs/api/java/awt/GridBagLayout.html (2013-11-08 12:48)

11] “Microsoft Access” http://office.microsoft.com/en-gb/access/ (2013-11-12 09:27)

[12] “Table Analyzer” http://office.microsoft.com/en-us/access-help/about-the-table-analyzer-HP005275385.aspx (2013-11-12 09:28)

[13] “NORMA” http://www.ormfoundation.org/files/folders/norma_the_software/default.aspx (2013-11-12 09:29)

[14] “Microsoft Visual Studio” http://msdn.microsoft.com/en-US/vstudio/ (2013-11-12 09:30)

[15] “SQL Server 2012 Integration Services, SSIS.” http://www.microsoft.com/en-us/sqlserver/solutions-technologies/enterprise-information-management/integration-services.aspx. (2013-07-18 10:04)

[16] “Mysqlimport.” http://dev.mysql.com/doc/refman/5.0/en/mysqlimport.html (2013-07-18 11:23)

[17] “MySQL Workbench.” http://www.mysql.com/products/workbench/ (2013-07-18 11:30)

Appendix 1

The import query

```
USE birdObservations;
LOAD DATA LOCAL INFILE
"/path_to_AKN_data/eBird/ebird_americas_data_grouped_by_year_v2.0/2009/checklists.csv"
INTO TABLE DBimport
COLUMNS TERMINATED BY ','
ESCAPED BY '"'
LINES TERMINATED BY '\n';
```

The SPLIT_STR function

```
USE birdObservations;
DROP FUNCTION IF EXISTS SPLIT_STR;
DELIMITER //
CREATE FUNCTION SPLIT_STR(str VARCHAR(4500), splittingchar VARCHAR(2), x INT )
RETURNS VARCHAR(4500)
BEGIN
        RETURN SUBSTR(SUBSTRING_INDEX(str, splittingchar, x),
                LENGTH(SUBSTRING_INDEX(str, splittingchar, x-1))+IF(x > 1, 2, 1));
END //
DELIMITER ;
```

The stored procedure Parsetext (first parser)

```
USE birdObservations;
DROP PROCEDURE IF EXISTS Parsetext;
DELIMITER //
CREATE PROCEDURE Parsetext(sid VARCHAR(45))
BEGIN
        DECLARE str VARCHAR(4200);
        DECLARE cur_str VARCHAR(70);
        DECLARE name_val VARCHAR(50);
        DECLARE number_val INT;
        DECLARE length_val INT DEFAULT 1;
        DECLARE counter_val INT DEFAULT 1;

        SELECT species FROM DBimport WHERE samplingId = sid INTO str;
        loop_label: LOOP
                        IF length_val != 0 THEN
                                SELECT SPLIT_STR(str,' ',counter_val) INTO cur_str;
                                SELECT SPLIT_STR(cur_str,':', 1) INTO name_val;

                                IF BINARY(SPLIT_STR(cur_str,':',2)) != 'X' THEN
                                        SELECT SPLIT_STR(cur_str,':', 2) INTO number_val;
                                ELSE
                                        SELECT null INTO number_val;
                                END IF;
                                INSERT IGNORE INTO Birds_sighted(samplingId, name,
                                numbersSighted)
                                VALUES (sid, name_val, number_val);
                                SELECT CHAR_LENGTH(SPLIT_STR(str,' ',counter_val + 2))
                                INTO length_val;

                                SET counter_val = counter_val + 1;
                                ITERATE loop_label;
                        ELSE
                                LEAVE loop_label;
                        END IF;
        END LOOP;
END //
DELIMITER ;
```

The stored procedure Parsetext (improved version)

```
DROP PROCEDURE IF EXISTS Parsetext;
DELIMITER //
CREATE PROCEDURE Parsetext(sid VARCHAR(45))
BEGIN
        DECLARE str VARCHAR(4200);
        DECLARE name_val VARCHAR(50) DEFAULT '';
        DECLARE number_val VARCHAR(10 );
        DECLARE number_val_int INT;
        DECLARE length_val INT DEFAULT 1;
        DECLARE counter_val INT DEFAULT 1;
        DECLARE cur_str VARCHAR(50) DEFAULT '';

                                SELECT species FROM DBimport WHERE samplingId = sid
                               INTO str;
        SELECT (CHAR_LENGTH(str) - CHAR_LENGTH(REPLACE(str,' ','')))
         INTO length_val;
        loop_string: LOOP
                        IF (length_val != 0) THEN
                                SELECT SUBSTRING_INDEX(SUBSTRING_INDEX(str,' ',
                                counter_val),' ',-1) INTO cur_str;
                                SELECT SUBSTRING_INDEX(cur_str,':',1) INTO name_val;
                                SELECT SUBSTRING_INDEX(cur_str,':',-1) INTO number_val;
                                IF (BINARY(number_val) = 'X') THEN
                                        SELECT null INTO number_val_int;
                                ELSE
                                        SELECT number_val INTO number_val_int;
                                END IF;
                                INSERT INTO Birds_sighted(samplingId, name, numbersSighted)
                                VALUES (sid, name_val, number_val_int);
                                SET counter_val = counter_val + 1;
                                SET length_val = length_val - 1;
                                ITERATE loop_string;
                        ELSE
                                LEAVE loop_string;
                        END IF;
                END LOOP;

END //
DELIMITER ;
```

The stored procedure Looptable

```
USE birdObservations;
DROP PROCEDURE IF EXISTS Looptable;
DELIMITER //
CREATE PROCEDURE Looptable()
BEGIN
        DECLARE sid_val VARCHAR(45);
        DECLARE no_more_rows BOOLEAN;
        DECLARE loop_cntr INT DEFAULT 0;
        DECLARE num_rows INT DEFAULT 0;

        DECLARE table_cur CURSOR FOR
        SELECT samplingId FROM DBimport WHERE error = 0;

        DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET no_more_rows = TRUE;

        OPEN table_cur;
        SELECT FOUND_ROWS() INTO num_rows;

        the_loop: LOOP

                FETCH table_cur INTO sid_val;

                IF no_more_rows THEN
                        CLOSE table_cur;
                        LEAVE the_loop;
                END IF;
                CALL Parsetext(sid_val);
                SET loop_cntr = loop_cntr + 1;

        END LOOP the_loop;
        SELECT num_rows, loop_cntr;
END //
DELIMITER ;
```

The complete script for populating the database

```
CREATE DATABASE birdObservations;
USE birdObservations;
-- create tables
CREATE TABLE Birds_sighted
(
samplingId VARCHAR(45) NOT NULL,
name varchar(50) NOT NULL,
numbersSighted INT,
CONSTRAINT pk_Birds_sightedEntry PRIMARY KEY (samplingId,name)
);

CREATE TABLE DBimport
(
samplingId VARCHAR(45) NOT NULL,
latitude DECIMAL(15,12),
longitude DECIMAL(15,12),
year INT(11),
month INT(11),
day INT(11),
time VARCHAR(45),
country VARCHAR(45),
state VARCHAR(45),
observationType VARCHAR(45),
observationDuration VARCHAR(45),
distanceTravelled INT(11),
area INT(11),
observerId VARCHAR(45),
birdingPartySize VARCHAR(45),
species VARCHAR(4200),
error BOOLEAN NOT NULL DEFAULT 0
);

CREATE TABLE Location
(
latitude DECIMAL(15,12) NOT NULL,
longitude DECIMAL(15,12) NOT NULL,
country VARCHAR(45),
state VARCHAR(45),
PRIMARY KEY (latitude, longitude)
);

CREATE TABLE Observer
(
observerId VARCHAR(45) NOT NULL,
name VARCHAR(45),
address VARCHAR(45),
PRIMARY KEY (observerId)
);

CREATE TABLE Sampling
(
samplingId VARCHAR(45) NOT NULL,
date DATE,
time TIME DEFAULT NULL, -- So that occurrences of '?' can be changed to null
birdingPartySize INT(11),
observationType VARCHAR(45),
observationDuration TIME,
distanceTravelled INT(11),
area INT(11),
observerId VARCHAR(45),
latitude DECIMAL(15,12),
longitude DECIMAL(15,12),
PRIMARY KEY (samplingId)
);

CREATE TABLE Species
(
name VARCHAR(45) NOT NULL,
commonName VARCHAR(100),
PRIMARY KEY (name)
);

CREATE TABLE Taxonomy
(
sci_name VARCHAR(45),
taxon_order INT(11),
primary_com_name VARCHAR(100),
category VARCHAR(45),
order_name VARCHAR(45),
family_name VARCHAR(45),
subfamily_name VARCHAR(45),
genus_name VARCHAR(45),
species_name VARCHAR(45)
);
-- load csv file into temporary table DBimport.
LOAD DATA LOCAL INFILE
"/path_to_AKN_data/eBird/ebird_americas_data_grouped_by_year_v2.0/2009/checklists.csv"
INTO TABLE DBimport
COLUMNS TERMINATED BY ','
ESCAPED BY '"'
LINES TERMINATED BY '\n';
-- load Taxonomy CSV file into temporary table Taxonomy.
LOAD DATA LOCAL INFILE
"/path_to_AKN_data/eBird/ebird_americas_data_grouped_by_year_v2.0/doc/taxonomy.csv"
INTO TABLE Taxonomy
```

```
COLUMNS TERMINATED BY ','
ESCAPED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES;
-- Change all occurences of '?' in column observationDuration to null
UPDATE DBimport
SET observationDuration = null
WHERE observationDuration = '?';
-- Change all occurences of '?' in column birdingPartySize to null
UPDATE DBimport
SET birdingPartySize = null
WHERE birdingPartySize = '?';
-- Change all occurences of '?' in column time to null
UPDATE DBimport
SET time = null
WHERE time = '?';
-- set error column of duplicate samplingId rows to 1
UPDATE DBimport
SET error = 1
WHERE samplingId IN
(SELECT x.samplingId FROM (SELECT DISTINCT samplingId FROM DBimport
GROUP BY samplingId HAVING COUNT(samplingId) > 1) AS x);
-- load observer data into Observer table (no values for name and address exists at the moment)
INSERT INTO Observer (observerId) SELECT DISTINCT observerId FROM DBimport WHERE
error = 0;
--  load species data into Species table
INSERT INTO Species (name, commonName) SELECT DISTINCT sci_name,
primary_com_name FROM Taxonomy;
-- load location data into Location table
INSERT INTO Location (latitude, longitude, country, state) SELECT DISTINCT latitude,
longitude, country, state FROM DBimport WHERE error = 0;
-- load the Birds_sighted-table with the processed values from the species strings in DBimport
-- Looptable calls Parsetext on each entry in the species column of DBimport
CALL Looptable();
-- load sampling data into Sampling table
INSERT INTO Sampling
(SamplingId, Date, Time, BirdingPartySize, ObservationType,
ObservationDuration, DistanceTravelled, Area, ObserverId, Latitude, Longitude)
SELECT SamplingId, makedate(Year, Day), sec_to_time(Time*3600),
BirdingPartySize, ObservationType, sec_to_time(ObservationDuration*3600),
DistanceTravelled, Area, ObserverId, Latitude, Longitude FROM dbimport WHERE error = 0;
-- add index on the Birds_sighted-table
CREATE INDEX num_index ON Birds_sighted(numbersSighted);
-- add foreign key constraints for observerId in Sampling table
ALTER TABLE Sampling
ADD CONSTRAINT fk_ObserverId
FOREIGN KEY (observerId)
REFERENCES Observer(observerId);
-- add foreign key constraints for latitude and longitude in Sampling table
```

```
ALTER TABLE Sampling
ADD CONSTRAINT fk_LatLong
FOREIGN KEY (latitude, longitude)
REFERENCES Location(latitude, longitude);
-- add foreign key constraints for samplingId in Birds_sighted-table
ALTER TABLE Birds_sighted
ADD CONSTRAINT fk_SamplingId
FOREIGN KEY (samplingId)
REFERENCES Sampling(samplingId);
-- add foreign key constraints for name in Birds_sighted-table
ALTER TABLE Birds_sighted
ADD CONSTRAINT fk_Name
FOREIGN KEY (name)
REFERENCES Species(name);
```

Appendix 2

Explanation of the attributes in the *DBimport* CSV file.

SAMPLING_EVENT _ID	Unique identifier for each data sample / checklist.
LATITUDE	Decimal latitude.
LONGITUDE	Decimal longitude.
COUNT_TYPE	What kind of observation this sample is: stationary (P21), traveling (P22, P34), area (P23, P35), or casual (P20). Protocol P34 is a small amount of data contributed from the Rocky Mountain Bird Observatory that is believed to be high quality. Protocol P35 data are backyard area counts made on consecutive days.
COUNTRY	Full name of country /political unit where observation took place.
STATE_PROVINCE	Name of the state, province, or region where observation was made.
YEAR	Year.
MONTH	Month of the year, ranging from 01 through 12.
DAY	Day of the year, ranging from 1 through 366.
TIME	Time when observation started, ranging from [0, 24). Fractional hours represent minutes (e.g. 13.5 = 1:30PM). Times are local times (including daylight savings when / where appropriate).
EFFORT_HRS	Duration of observation, in hours.
EFFORT_DISTANCE_KM	Distance travelled during observation period, in kilometres. Equals 0 for non-traveling counts.
EFFORT_AREA_HA	Size of survey area for area counts, in hectares. Equals 0 for non-area counts.
OBSERVER_ID	Identifier for the person who submitted the data.

NUMBER_OBSERVERS	Number of observers in the birding party.
SPECIES	The species observed followed by a colon and the number of birds of that species that was observed. Eventual other birds observed follow after a space. (e.g. Actitis_macularius:6 Anhinga_anhinga:5 Ardea_herodias:2)

Appendix 3

Explanation of the attributes in the *Taxonomy* CSV file.

SCI_NAME	The Latin name of a bird species.
TAXON_ORDER	The taxonomical order (INTEGER).
PRIMARY_COM_NAME	The primary common name of a species.
CATEGORY	The category of a species. For example if it is a species, a hybrid or a slash (eg. Cackling/Canada Goose).
ORDER_NAME	The order name of the species.
FAMILY_NAME	The family name of the species.
SUBFAMILY_NAME	The subfamily name of the species.
GENUS_NAME	The genus name of the species.
SPECIES_NAME	The name of the species.

Appendix 4

```
-- SELECT samplingId, numbersSighted, name FROM Birds_sighted WHERE numbersSighted =
(SELECT MAX(numbersSighted) FROM Birds_sighted);
-- What is the biggest observation of a particular bird, and which bird is that.

-- SELECT AVG(birdingPartySize) FROM Sampling;
-- What is the average birding party size?
-- Birding party size being how many bird observers was present during the observation.

-- SELECT AVG(HOUR(time)) FROM Sampling;
-- What is the average time for the reported bird observations.

-- SELECT COUNT(*) FROM Birds_sighted AS c, Sampling AS s WHERE c.samplingId =
s.samplingId AND c.name = 'Catharus_ustulatus' AND MONTH(s.date) = 5;
-- See at how many different occasions this particular bird has been seen during May

-- SELECT SUM(numbersSighted) FROM Birds_sighted AS c, Sampling AS s WHERE
c.samplingId = s.samplingId AND c.name = 'Catharus_ustulatus' AND MONTH(s.date) = 5;
-- See how many specimens of this particular species that have been sighted during May.
-- Query above does not take into account the entries where the
-- amount is set to null, previous X, which means the number of birds
-- of that particular species is not 100 % accurate.

-- SELECT name, MAX(numbersSighted) FROM Birds_sighted;
-- What is the biggest observation of a particular bird, and which bird is that.

-- SELECT COUNT(*) FROM Location AS l, Sampling AS s WHERE l.country = 'United_States'
AND l.state = 'Kentucky' AND s.latitude = l.latitude AND s.longitude = l.longitude AND
MONTH(s.date) = 6;
-- How many bird observations have been made during June in Kentucky, USA?

-- SELECT c.name, s.samplingId, s.observerId, s.latitude FROM Sampling AS s, Birds_sighted AS
c WHERE s.latitude = (SELECT MAX(l.latitude) FROM Location AS l, Sampling AS sa WHERE
sa.latitude = l.latitude AND sa.longitude = l.longitude AND MONTH(sa.date) = 2) AND
s.samplingId = c.samplingId;
-- What is the samplingId and latitude of the northernmost bird-sighting in February, and who did
the observation, and what birds were sighted?

-- SELECT COUNT(*) FROM Sampling WHERE observerId = (SELECT observerId FROM
Observer LIMIT 1) AND MONTH(s.date) = 1;
-- How many bird watching sessions have the first person in the observer-table reported in during
January?

-- SELECT s.observerId, s.date, s.time, MAX(c.numbersSighted), l.state, l.country FROM
Birds_sighted AS c, Sampling AS s, Location AS l WHERE l.state = 'Kentucky' AND
c.samplingId=s.samplingId AND s.latitude = l.latitude AND s.longitude = l.longitude;
-- When was the maximum amount of birds observed in Kentucky (numbersSighted)?
```