Taxi Order Data Visualization System

CS241 Final Project

davidwang200099

Department of Computer Science,
Shanghai Jiao Tong University, Shanghai, China

1 Introduction

Nowadays, online car-hailing system such as Didi emerges in China, bringing great convenience to passengers. However, there are also some problems, such as demand-supply gap, which challenges the systems. Therefore, an analysis of order data is necessary, with which online car-hailing companies can distribute taxis to different regions to satisfy most of the passengers' demand. In this project, the author implemented a Taxi Order Data Visualization System to help analysing taxi order data. In this project, one can visualize spatial-temporal demand patterns, probability distribution of other demand patterns, and order distribution on the map. Besides visualization, the system can also make predictions about the data imported and allow users to do free style query with SQL.

2 Implementation Details

2.1 Import and In-memory Storage of Dataset

The first step is to import the dataset. At the very beginning, a QDialog widget appears, which allows user to select location of dataset, choose the fields and intervals they are interested in. The options of fields to import are implemented with QCheckBox, because they are discrete. The options of dates users are interested in are implemented with two QComboBoxes, because this kind of visualization needs to show the change of an amount in a continuous period of time. Therefore discrete dates are meaningless.

To gain better performance, memory-based database is introduced. In my design, the data are imported and organized into different tables with the help of SQLite. In my design, orders in the same date are put into the same table. In this way, the database can spend less time querying data when users are interested in only a small part of the imported dataset.

The import of data is implemented via multi-thread. A subclass of QThread: fileReadThread, is used to import the dataset. Each fileReadThread is in charge of importing data of one date. Therefore, there will be no racing condition when importing dataset because one thread will write into only one table in the database.

A progress bar is introduced to show the progress of the import of dataset. When the user selects the interested dates, several threads are created to load the data into memory. Each thread will find out the number of files it should cope with and send relevant messages back to the main thread. The main thread will aggregate the information received from the sub-threads and update the total of number of files. After a fileReadThread finishes reading a file, it will send a signal to the main thread, which reflects that there has been some progress. Then the main thread will calculate the progress and update the progress bar.

To protect the program from crashing because data files are not completely read, special precaution is used. The button asking the program to draw graphs, do free query and make predictions will be disabled until the data set loading process is finished.

2.2 Basic Data Visualization

The task is divided into three parts: basic data visualization, free query and prediction. Therefore I organize the GUI with 4 QTabWidgets, each of which is responsible for one task mentioned above.

In this part, I implemented the visualization of change of orders over time, the distribution of travel time and the distribution of fee. Users can set the start date & time and the end date & time with four QComboBoxes. Time step and interested grid can also be tuned. The "Import" button must be clicked to import the selected part of dataset before users can click the "Draw" button to visualize the data.

Users can select the type of visualization task with a QCombobox at the top right corner. If the user did not tune the time period, the system will remind him or her to tune the interested periods. If the system can not finish a task because related data field was not imported, the system will not execute the task. Then a QMessageBox will appear to remind the user.

After the user click ”Draw” button, this button will be disabled until the drawing process is finished. The drawing function will query the database to extract the change of orders/travel time/fee during the interested time period. Because the datatime representation is too complicated, but unix timestamp is just an integer, therefore the data is classified in terms of unix timestamp. The system will maintain a vector, which stores the number of orders/travel time/fee in one step. Take change of number of orders as example. The user has specify the steps (for example, one hour). Then the system will calculate the

2.3 Free Query

In this part, I implemented free query with QLineEdit, QPushButton and QTableWidget. Free Query allow users to query the dataset with SQL by inputting the instruction and the click ”query” button. The system will parse the instruction, ask SQLite to exeute the SQL instruction and then display the results in the QTableWidget in the form of a table.

Structure of SELECT Instruction Here are some examples of the SQL SELECT instructions, which are used to query data in the database.

$$\text{SELECT } field_1(, field_2, field_3, ..., field_n) \text{ FROM } < \textit{tablename} >$$

$$\text{SELECT } field_1(, field_2, field_3, ..., field_n) \text{ FROM } < \textit{tablename} > \text{ ORDER BY } field_i$$

$$\text{SELECT } field_1(, field_2, field_3, ..., field_n) \text{ FROM } < \textit{tablename} > \text{ WHERE } \textit{boolean} - \textit{exp}$$

$$\text{SELECT } SUM(field_1) \text{ FROM } < \textit{tablename} >$$

Obviously, there are many different kinds of SELECT instructions. At first I tried to parse the instruction first to figure out the interested fields, but found it to complicated. Then I use QSqlModel to simplify the query process. The QSqlModel can store some useful information such as table header, number of selected records, etc., which facilitate the freeQuery function.

Ask SQLite to execute the SQL instruction The query is implemented via multi-thread. A subclass of QThread: freeQueryThread, is introduced to finish this task. It will build a connection with the database, input the instruction and store the return items into a vector of vectors of strings. Then it will display the items in the QTableWidget.

Display the Result The result is organized as a table in the QTableWidget. Each cell in the table is an instance of QTableWidgetItem. With the help of QSqlModel, information such as table header and the number of selected records can be used to generate the table in the QTableWidget.

Prediction In prediction, I implemented the prediction of destination given departure place and time, and that of fee given departure, destination and time.

For destination prediction, the system will analyse all the data imported into the database, and output 5 most frequently visited grids, which are the places passengers are most likely to go to. For fee prediction, the system will analyse the fee, and then output the maximum, minimum and median of the fee.