Written by Riya Walia

Importing Tool

Objective

The support team deals with a large number of tickets each day and would like to be able to use “Tableau” to visualize the meta-data associated with each ticket.
To achieve this, a database was designed and a console application was created to import the flat file that is exported from Request Tracker.

Database Design

Summary

This database is designed using basic table normalization (Third Normal Form). The core of the database centers on the table “Ticket” while all of the other tables are related to it. All tables' columns derive from the flat source file, as provided by Request Tracker. There are two types of tables in this database.

Lookup Tables

- Ticket
- User
- Course
- Tool
- RequestorType
- CourseType

Link Tables

- TicketCourse (Ticket to Course)
- TicketTool (Ticket to Tool)
- Owner (Ticket to User)
- Admin (Ticket to User)
- Cc (Ticket to User)
- Requestor (Ticket to User and RequestorType)

Notes

The Ticket has one-to-many relationships with TicketCourse, TicketTool, Admin, Cc and Requestor. However, Ticket has a one-to-one relationship with Owner. Multiple Tools and Courses associated with a single ticket are not differentiated. Two courses/requestors with the same name but different types are recorded as two different entries in the Course/Requestor table.

Application Design

Summary

The application imports a flat file (generated by Request Tracker - .tsv/.txt) into an appropriate database structure using object oriented paradigms to maintain relationships between data. The flat file must be of the following format: the first line of data is column names, rest of the lines are ticket data. Each field is separated by tab spaces.

User Interface

The user is presented with a menu on the console with five options.

- **Add New File**
  This prompts the user to enter the name of a file and loads file onto the program to perform operations on it. This option facilitates multiple file handling.
- **Save File**
  This saves all the data from the file loaded in the program into the database. If no file is loaded, 'Add New File' is automatically executed. The user is then notified of how many tickets were saved successfully, and the Ids of tickets that failed, if any.
- **Verify File**
  This quickly checks whether information in the database is accurate, according to the file loaded on the program. If some data is incorrect, the user is notified which tickets are not up to date.
- **Save and Verify File**
  Saves data onto database and then automatically verifies it.
- **Quit**
  The user can perform as many commands on as many files as required until the user chooses to quit.

The menu also displays the name of the file currently loaded on the program, for the benefit of keeping track of performing operations on multiple files.

Handling a text file as a whole

File.NewFile()

The file name is taken from user input in the console. The directory is set in App.config. The user is notified if the directory does not exist or file name is invalid.

File.LoadData()

The file is parsed through with pre-set delimiters (tabs) and stored in a raw data format in a list of string arrays. The first line of words in the text file must be field names. These are maintained in a < string,int > dictionary to keep track of column names and index numbers. This is because columns may appear in different orders in multiple text files.

File.CreateTickets()

A list of Tickets is created and iterated over while performing Ticket.Create()

File.SaveTickets()

The list of tickets is iterated over while performing Ticket.DropAllRelationships() and Ticket.Save(). Ids of tickets that failed to save to the database are kept track of so that the user can be notified on the Console later.

Handling a single Ticket in particular

Ticket.Create()

A single raw Ticket a is iterated over, strings are parsed and saved in appropriate data types (Eg: Timestamps are parsed into DateTime objects). Pre-set delimiters are used to separate courses, users etc. that were stored in a single string field. Once the string is split, the delimited names are saved into appropriate objects (courses, tools, users) and relationships are maintained in the main Ticket object.

Ticket.DropAllRelationships()

As each raw Ticket (string array from raw data) is encountered, the id retrieved and the corresponding rows in the link table are dropped and re-inserted to ensure that relationships always remain updated. Courses/Users/Tools remain in the core tables to avoid regeneration of Ids and re-insertion of data for the other tickets they're connected to.

Ticket.Save()

Using an SQL connection (connection string is saved in App.config), the data in the ticket object is passed as SQL Parameters to SQL Stored Procedures. The Ids of all data that already exists in the database/ or is newly inserted, are retrieved (with the help of queries and identity_Current()) and maintained in < string,int > dictionaries to reduce unnecessary calls to the database.

Verifying data

Verifying simple fields

The raw text file (list of string arrays) is iterated over. For each row, an id is selected and used to retrieve the corresponding row from Ticket Table in the database. Each field from the row is parsed into a string and compared with the corresponding fields in the raw text file. Any unequal strings are recorded in the Verification Log.

Verifying courses, users, and tools

Link tables and lookup tables are extracted from the database using a DataTable object. The id retrieved from the raw text file is used to select all related course/tool/user Ids in the link tables. I maintain these Ids as my "table data". To get my "raw data", I retrieve course/tool/user names from the raw text file and look up in the tables for the correct id. These Ids are parsed as strings into hash sets which are then checked for equality. If unequal, the specific unequal values are logged, along with other details.

Verifying course/requestor types

TypeName is retrieved from the raw text file. The corresponding Id is look up and compared to the Ids recorded in the course/requestor tables, according to the Id of the ticket being handled at the time.

A list of Ids of tickets that failed to verify is maintained in order to notify the user later.

Maintaining Logs

Error Log

A try...catch is wrapped around every loop that iterates over the list of tickets to ensure that if a single ticket failed, execution continues. The error message of the exception caught is written in the errorlog.txt file using a StreamWriter, along with the line on which the exception was found and the corresponding ticket Id.

Verification Log

Any discrepancies found between data extracted from the table and data extracted from the raw format of the text file is recorded on a text file VerificationLog.txt, along with the ticketId and column name

Optimization

Time Efficiency (Big O Notation)

This program runs in O(4 * N) = O(N) (linear) time where N is the number of tickets in a file, and thus the number of lines in a .tsv file.

Space Efficiency

In order to retrieve and maintain Ids generated by SQL Server, a dictionary is kept for each look-up table. While this takes up a bit of extra space, it is a trade off for better time efficiency. The alternate, brute-force approach is to iterate over the list of tickets for each unique data encountered in a single Ticket object and insert the Id for all courses/tools etc. This makes the program run in O(n * n) time. As the average number of lines per file is about 300 and no more than a couple of files are loaded per month, space efficiency was traded off for time efficiency.