CS 1555 Term Project ExpressRailway Database Application

Group Members

Zachary Whitney zdw9 3320178 Github: MisterZW
Joe Reidell jmr240 4138996 Github: jreid2f

Repository URL

https://github.com/MisterZW/cs1555_term_project

Instantiating the Database Schema

Prior to running the user application, it is necessary to set up the database DDL and DML commands. All of the application logic assumes that this step has been completed in advance, so it will not work otherwise.

The first necessary step is to run **\i sql/schema.sql** in the PostgreSQL CLI. This is a sequence of DDL commands comprising the table schemas from their respective sql files, the indexes for the database in *indexes.sql*, and the triggers from *triggers.sql*. Next, run **\i sql/dml.sql** in the CLI to set up the DML commands the application will run on top of the schema. These steps should only need to be done once.

Compiling and Running the Application

We created a Makefile to compile and run the application.

1. Make Clean(Just in case anything was compiled beforehand)
2. Make
3. Make Run

Dependencies

ExpressRailway makes use of two external libraries. We utilized FlipTable from author Jake Wharton to display ResultSet data in a well-formatted way in the Java application. We changed the source code to support pagination of ResultSet information and by wrapping it in the src package.

We also utilized ScriptRunner from author Clinton Begin to aid in running SQL scripts for importing/exporting database data. In addition to packaging this class in src, we disabled the script IO to prevent the user from being bombarded with SQL statements in the application layer.

Both libraries are available for use under the APACHE license version 2.0. They will be automatically compiled and prepared by the Makefile instructions in the “Compiling and Running the Application” section above.

Menu Instructions

1. Once the application is compiled and running, enter ‘16’ and import the large test dataset(will take a few seconds to import)
2. Next, enter ‘1’ for your choice and create a new account
3. Next, enter ‘3’ and enter the ID number your account was given to view your account
    a. Ex. Success! Matt's new ID # is 302
4. Next, enter ‘2’ and change your email while keeping everything else the same

5. Enter ‘3’ again and enter the same ID number your account was given to view the changes
6. Next enter ‘4’ to enter Single Route Trip Search
    a. Look for arrival station# 1 and destination station# 20
    b. Look for travel day 3 and use any sorting options
7. Next enter ‘5’ to enter Combination Route Trip Search
    a. Look for arrival station# 1 and destination station# 20
    b. Look for travel day 3 and use any sorting options
8. Then enter ‘6’ to Add Reservation to Single Route
    a. Enter your customer ID# and a schedule ID#
    b. Reserve 2 tickets and enter arrival station# 1 and destination station# 20
9. Then enter ‘7’ to Add Reservation for Sequence of Trips
    a. Enter your customer ID#
    b. Reserve 2 tickets and enter for a trip to 1, 3, 6, & 20
    c. Enter -1 to finish
10. Next enter ‘8’
11. Enter ‘9’ to look for routes which travel for more than one line
    a. Enter ‘Y’ to look at another 10 routes. ‘N’ to exit
12. Enter ‘10’ to look for routes that pass through the same stations but different stops
13. Enter ‘11’ to look for stations that all trains pass through. (Should fine no result)
14. Enter ‘12’ to find all trains that do not stop at a specific station
    a. Enter any number station ID# and type ‘Y’ till you’re done. Then type ‘N’
15. Enter ‘13’ for find routes that stop at at least a specified percentage of stations
    a. Enter in 80, type ‘Y’ till you’re done. Then type ‘N’
16. Enter ‘14’ to display the schedule of a route
    a. Enter in 100
17. Enter ‘15’ to find the availability of a route at every stop on a specific day
    a. Enter route ID# 100, travel day 7, hour 7, minutes 00
18. Enter ‘17’ to export the database. Name the file whatever you would like
19. Enter ‘18’ to delete the database. Type ‘Y’
20. Enter ‘19’ to exit the application

Import/Export/Delete Database Commands

The import command asks the user whether they want to use a small test dataset SQL file, a large test dataset SQL file, or a custom dataset.The small test dataset (sql/small.sql) inserts a small amount of data transactions into the tables. For more information about that, look at **“Small Test Data File and Tests”**. The large test dataset (sql/mock_data.sql) inserts a larger amount of data transactions into the tables.

Imports about 400 stations in a 20x20 grid. For more information about this, look at **“Large Test Data File and Tests”**. A custom dataset is a SQL file with a dataset from a previous exported database.

The export command asks the user to enter a name for the file that the database will be written into. It writes the database queries into a custom SQL file that can later be imported.

The delete command asks the user whether they would like to delete the database or not. If they would like to delete the database, it drops all queries in the tables and lets the user know whether it was successful or not.

Schema Design Choices

We first started off with figuring out what objects could be used to connect different elements (i.e. Trains, Passengers, Stations, etc.). Even though we knew which tables were needed, the most difficult part was deciding how to make the connections between each one. We made one relationship that was many-to-many between Passenger booking tickets to a Train Schedule. We made another relationship between the route being for the Train Schedule that was one-to-many. The Train Schedule had another relationship with the Train being serviced by the Schedule. That relationship is a one-to-many. The next relationship was established between Route and Connection. A route runs on many different connections in the system. This relationship is many-to-many. Plenty of routes can be connected to different stations and rail lines. Another relationship is established between Station and Connection. This relationship is a one-to-many. One station can be connected to many different routes and rail lines. The last relationship was established between Rail Lines and Connections. This relationship is one-to-one. One rail line can be connected to one station or route.

Database Triggers

The database has only 2 trigger functions. The first (TRIGGER: trig_sell_tickets, PROCEDURE: update_seats_left()) is very simple -- each time a new booking for a trip is made, it updates the number of available seats on that trip. If there are not enough seats available to accommodate the booking, the trigger blocks the insert and raises an error explaining the conflict.

The second (TRIGGER: sched_needs_trips, PROCEDURE: create_trips()) is more complex. Whenever a new schedule is inserted into the database, this trigger creates the sequence of trips that comprise the schedule (individual edges from one station to another on that scheduled route at that day and time). While doing so, the trigger checks that:

a) the rail lines needed are not in use at that time already
b) the stations along the schedule where this train stops are all open when the train gets there

If either constraint is violated, the trigger blocks the entire schedule from being inserted into the database as invalid. Otherwise, the trips are given calculated data to store to ease route processing later (for example, distance, time and price for the trip) and stored in the TRIP table.

Indexes

We elected to include four additional secondary B+ tree indexes on the database in addition the the default indexes created by PostgreSQL.

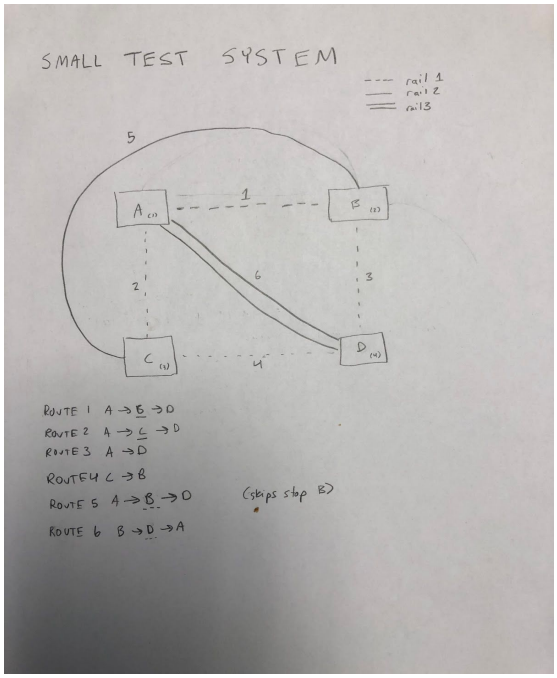
1) Index on t_route attribute of SCHEDULE table
   Since a schedule is an instantiation of an individual train route on a particular day/direction, it is a commonly referenced but rarely altered field. We selected this in hopes of speeding the Single Trip Route Search and several advanced queries.
2) Index on route_id AND Index on station_id attributes (distinct indexes) both on the ROUTE_STATIONS table
   Since entire purpose of the ROUTE_STATIONS relation is to express a many to many relationship between TRAIN_ROUTE and STATION, providing indexing on both attributes should improve the efficiency of that translation.
3) Index on sched_id attribute of TRIP table
   Semantically, a schedule consists of a list of trips, so they are very tightly related in most queries. Since we often want to know which schedule a trip is for and the TRIP table is quite large, indexing on this attribute should improve performance substantially.

Small Test Data File and Tests

Prior to scaling the application to the large scale of the project requirements, we created a minimal data model for correctness testing. Since there are only a handful of trains, stations, and routes to reason about, it is much easier to determine the expected behavior of functionality.

The SQL statements to import this data to the database reside in sql/small.sql. It is also available as an option in the Java application's "Import Database" command. For a rough visualisation of the layout of this data, see below.

We also drafted a simple test battery to run on this dataset (*src/small_tests.sql*). To run this test, simply run the command **\i small_tests.sql** at the PostgreSQL command line while the current directory is the sql/ directory. The test file will drop all tables and functions, recreate them, and load the small dataset prior to running the tests. Each test case states the expected behavior of the test prior to printing the test results for easy verification.



Large Test Data File and Tests

In some ways, the large test dataset is less sophisticated than the small one. It is simply a 20x20 grid of 400 stations with rail lines/connections in a simple matrix formation. There are routes which run along these rows and columns as well as a large number of randomly generated routes of 3 connected stations. We generated a large number of candidate schedules (~6000) and tested them against the database integrity constraints. We removed the invalid tuples from the dataset and were left with roughly 2000 valid schedules comprising some 8000+ trips overall. All data files were originally built using simple python scripts which are available for review in the data/ directory.

The SQL statements to import this data to the database reside in *sql/mock_data.sql*. It is also available as an option in the Java application's "Import Database" command. Though the dataset is fairly simplistic, it nevertheless is helpful in determining how the performance of our algorithmic choices might scale.

To verify the performance and correctness of our DML commands on the larger scale, we wrote a test battery targeted specifically to this dataset as well. To run this

test, simply run the command **\i tests.sql** at the PostgreSQL command line while the current directory is the sql/ directory. The test file will drop all tables and functions, recreate them, and load the large dataset prior to running the tests. Each test case states the expected behavior of the test prior to printing the test results for easy verification.

Possible Improvements

1) Allow agent and DBA logins to extend finer grained access control and sophistication to the application. The schema supports this option as a vestige of the previous requirement, but it is not fully implemented in the application layer.
2) Add a trigger to remove trips from the TRIP table if schedules are deleted from the SCHEDULE table. Would improve consistency preservation of the database.
3) Add a trigger to ensure that no train is running concurrently in different locations. This would improve the realism of the simulation somewhat, though this isn't specified as a project requirement.
4) Generate a more realistic dataset and better UI experience.

Difficulties

1) Using the SERIAL data type for id values in our tables was convenient in some ways during data generation, but it made importing/exporting the database substantially more challenging because it was another piece of state which needed to be considered on every operation. Dropping the database without resetting the serials broke the test import scripts. Importing custom datasets without setting the serials back to their proper values meant further inserts would break (because they'd overlap existing table values). Giant headache.
2) Allowing for schedules to run in reverse complicated all the trip scheduling algorithms in the DML quite a bit. In retrospect, the schema could have been designed to handle this aspect better and saved a lot of grief in the queries.
3) The FlipTables library has some issues if the console window is too small. I'm not sure how to fix this without making major modification to the library, which is kind of beyond the scope of this project.