CS223: Transaction Processing and Distributed Data Management
Spring 2015

Project Report

Ling Ji
Joel Fuentes

June 12, 2015

1 Project Description

This project is based on the implementation of a Resource Manager for a Rental Company by supporting data store, processing and concurrent transactions holding ACID properties. The main entities for this database are:

```
FLIGHTS(String flightNum, int price, int numSeats, numAvail)
HOTELS(String location, int price, int numRooms, int numAvail)
CARS(String location, int price, int numCars, int numAvail)
CUSTOMERS(String custName)
RESERVATIONS(String custName, int resvType, String resvKey)
```

Where:

- There is only one airline, and all seats on a given flight have the same price; flightNum is a primary key for FLIGHTS.
- All hotel rooms in a given location cost the same; location is a primary key for HOTELS.
- All rental cars in a given location cost the same; location is a primary key for CARS. custName is a primary key for CUSTOMERS.
- The RESERVATIONS table contains an entry corresponding to each reservation made by a customer for a flight, car, or hotel.

2 Design of our solution

In our solution we use Java classes to represent the database entities. We store the data in memory using several hashtables. We classes to represent the tables are as follow:

- *Flights* class containing a hashmap with flight number as key and Flight object containing Flight information as value to represent the Flights table,
- *Cars* class containing a hashmap with location as key and Car object containing information about a rental car as value to represent the Cars table,
- *Hotels* class containing a hashmap with location as key and Hotel object containing information of a hotel as value to represent Hotels table,
- *Reservations* class containing a hashmap with customer's name as key and ResvPair object as value to represent a combination of Customers table and Reservations table.

And there are several small additional classes to help implement our functions:

- *Flight* class, an object of Flight class contains every information about one flight,
- *Hotel* class, an object of Hotel class contains every information about one hotel,
- *Car* class, an object of Car class contains every information about a given location's rental cars,
- *ResvPair* class, combine a reservation's type and key of corresponding reserved item together to help with the implement of Reservations class,
- *OperationPair* class, combine an operation's associated table and the changed value's key together to help implement the hashmap.

2.1 ACID Properties

In this section are explained the detail of our solution to achieve ACID properties.

a) Atomicity

To achieve atomicity, we use shadowing by maintaining two copies of the in-memory database. All of the operations of a transaction before it commits are only taken within the active copy. We only merge the updates to the stable copy until the transaction commits.

Moreover, we use a hashmap (key: transaction id, value: Arraylist of ResvPair) to record the table and the key of items the active transactions change. So that at the commit stage of a transaction, we can select changes caused by it from the active copy of database to merge into the stable copy.

b) Consistency

We need to keep the data consistent according to the semantics. There are detailed checks of semantics consistency and locking strategy to keep the data consistent.

- Reservation and deletion should check available items before it is made.
- An item (flight, car, or hotel room) with existing reservation should not be deleted from the database.
- Price should never be negative.

There are also several Exception that are thrown when an inconsistent action occurs.Thus, these methods help ensure data always satisfy the semantics.

c) Isolation

To achieve isolation, we lock the item before the operation taken effect on them. Operations that add and delete items require a WRITE lock before they start. Operations that query about items require a READ lock before they start. All these locks are released only once a transaction commits or aborts.

The example of using locks is as follows:

```
try{
   if(!lm.lock(xid, FLIGHT + flightNum, LockManager.WRITE)){
      return false;
   }
}catch(DeadlockException e){
   abort(xid);
   throw new TransactionAbortedException(xid, "Transaction aborted by deadlock issue");
}
```

We try to lock the item associated with the operation properly. If lock is not available at that time, it means the operation conflicts with other granted locks, the operation cannot happen at this time and must wait. If a deadlock happens according to the timeout strategy, lock manager will throw out a Deadlock Exception. After catching the deadlock exception, the system solve this problem by aborting the mentioned transaction.

d) Durability

The durability property is achieved by having persistent copies of the tables on hard disk. These tables on disk are maintained updated when any transaction completes a commit operation after doing changes on some table. The detail of the main actions for durability are:

- Maintaining disk image: The write and read operations of the tables on disk are doing by Object Serialization in Java, this technique allows the objects to be stored and restored from files with the same state that their previous ones on main memory. As we mentioned above, we use shadowing which means two copy of each table (active and non-active) are maintaining on main memory and this implies to keep their copies on hard disk as well.
- Recovering: When the system as a server starts, it needs to do the following actions for recovering:
  - Check if exists a stored version of the database. In case the database exists on disk all tables are restored on memory, even active and non-active tables. If the database does not exist on disk the system creates a new one from scratch.
  - After restoring the database on memory, a checking step is done to verify if there are transactions that had not committed at the moment of a failure. For these uncommitted transactions the abort operation is executed.

3 Main changes for Part 2

For Part 2 of the project, we implemented a new class called WorkflowController which is the front-end and has the objective of calling the proper Resource Manager, considering that his Resource Manager is now distributed over different locations. The Transaction Manager class is also important in this part, and it was developed in order to deal with the transaction management (commit, abort, etc.).
On this part of the project, there also new methods associated with transactions that involve accessing to the data from different location, acting as a itinerary of operation. Here we use the same Resource Data class but working with different data according their methods (RM for flights, RM for cars, etc.).

4 Test cases

A set of regular and new test cases were carried out achieving successful for the following situations:

- Basic add, remove and query operation over every relation in the database.
- Locking operations by calling the operations mentioned above in a concurrent manner.
- Store and recover the database in different situations to guarantee consistency after failures.

- Commit and abort operations calls by concurrent transactions.
- All the rest to achieve the semantic of the relations in the database.

At the same time, a set of test for distributed resource manager were carried out successfully.