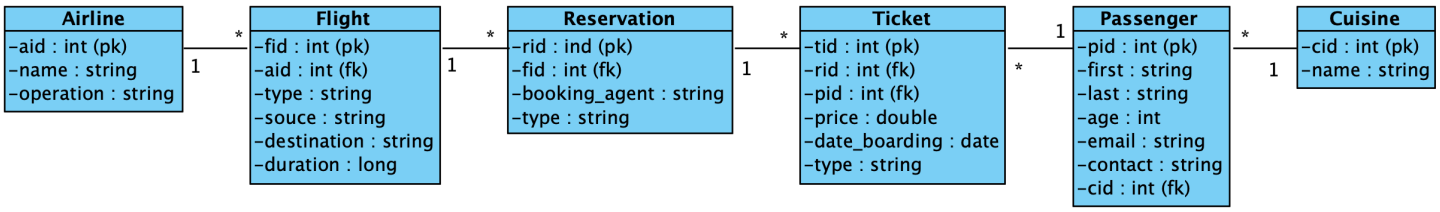
Airline Database



Based on the Airline database, write a SQL query to calculate how many reservations were made with the JetBlue flights for a cost higher than the average cost of all JetBlue reservations?
Note that the cost of a reservation is the sum of the price of all tickets in that reservation. Only list the total number of reservations. Rename the calculated field NumReservations>AvgForJB.

Step by step solution:

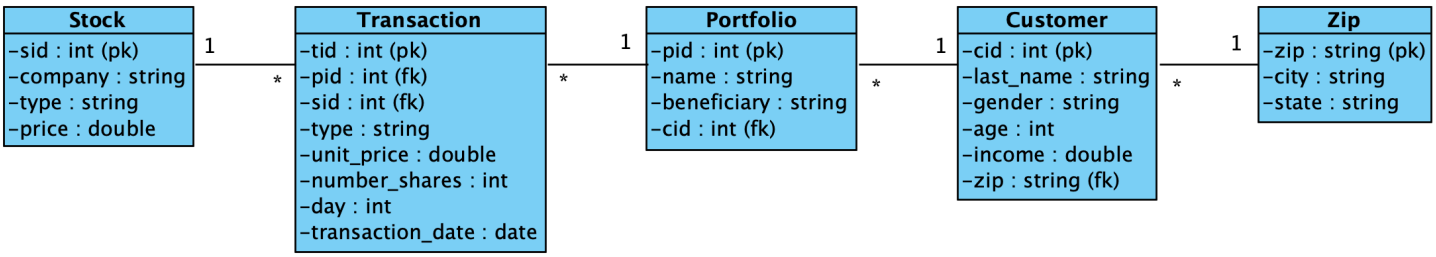
```
-- average per reservation
select r.rid, avg(t.price) as avg -- (A)
from airline a, flight f, reservation r, ticket_info t
where name='JetBlue'
and a.aid=f.aid and f.fid=r.fid and r.rid=t.rid
group by r.rid;

-- total average
select avg(t.price) as avg -- (B)
from airline a, flight f, reservation r, ticket_info t
where name='JetBlue'
and a.aid=f.aid and f.fid=r.fid and r.rid=t.rid
group by a.aid

select count(avg_per_reservation.rid) as 'NumReservations>AvgForJB'
from
        (select r.rid, avg(t.price) as avg -- (A)
        from airline a, flight f, reservation r, ticket_info t
        where name='JetBlue'
        and a.aid=f.aid and f.fid=r.fid and r.rid=t.rid
        group by r.rid) avg_per_reservation,

        (select avg(t.price) as avg -- (B)
        from airline a, flight f, reservation r, ticket_info t
        where name='JetBlue'
        and a.aid=f.aid and f.fid=r.fid and r.rid=t.rid
        group by a.aid) avg_total
where avg_per_reservation.avg > avg_total.avg;
```

Stock Database



Based on the Stock database, write a SQL query to
calculate the number of cities with 3 or more customers having completed a transaction. Only list the total number of cities, rename the calculated field NumCitiesLotsOfCustomers.

Step by step solution

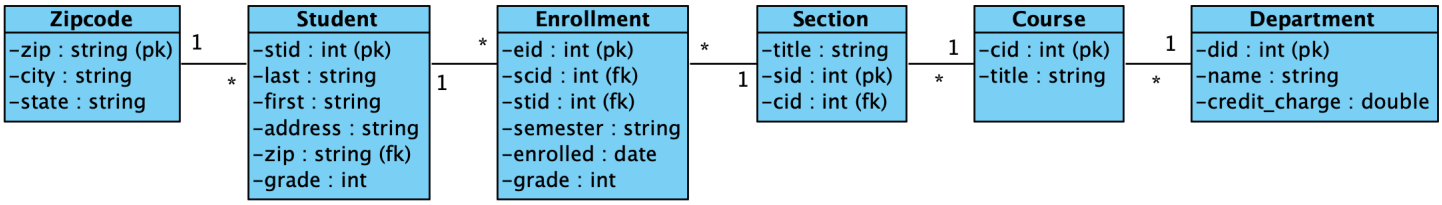
```
-- get all cities
select z.city, z.zip
from zips z;

-- get all customers and zip
select z.city, z.zip, c.last_name
from zips z, customers c
where z.zip=c.zip;

-- count customers per zip
select z.city, z.zip, count(c.cid)
from zips z, customers c
where z.zip=c.zip
group by z.city, z.zip;

-- count customers per zip that have a portfolio
select z.city, z.zip, count(c.cid) -- (A)
from zips z, customers c, portfolios p
where z.zip=c.zip
and p.cid=c.cid
group by z.city, z.zip;

-- count cities that have 3 or more customers with portfolios
select count(customer_count.city)
from zips z,
        (select z.city, z.zip, count(c.cid) as count -- (A)
        from zips z, customers c, portfolios p
        where z.zip=c.zip
        and p.cid=c.cid
        group by z.city, z.zip) customer_count
where customer_count.count >= 3
and z.zip=customer_count.zip
```



Final Exam Topic Review

- Design
  - UML
    - Text → UML → Text
    - Aggregation / Composition
      - CREATE … CASCADE DELETE - Composition
      - Otherwise - Aggregation
    - Inheritance - triangle on the abstract/base class
    - Reification → convert pure UML to something can be implemented as SQL
    - Mapping class / Association
    - 1 - many
    - many - many
  - Functional Dependencies
    - FD with non keys - bad
    - FD with keys - good
  - Normalization
    - 1NF - fields should be dependent on the key
    - 2NF - FD with part of key - compound keys - the whole key
    - 3NF - FD with some other non key field - nothing but the key
  - Relational Algebra
    - SELECT(TABLE, predicate) ⇒ Sub table with missing rows only the ones that match criteria
    - PROJECTIONS(TABLE, [field1, field2]) ⇒ same table with missing fields
    - PRODUCT(TABLE1, TABLE2) ⇒ all combinations of records in 1 and 2
    - JOIN(TABLE1, TABLE2, predicate)
- Relational Databases
  - SELECT
  - JOINS
  - GROUP BY
  - ~~HAVING~~
  - ~~INSERT~~
  - ~~UPDATE~~
  - ~~DELETE~~
- Programming the database
  - Stored procedures
    -
  - triggers
    - Execute for an event

        - Before/After ⇒ Update/Delete/Insert
        - Can do Update/Delete/Insert
    - ~~JDBC~~
    - ORM
        - @Entity
        - @Table
        - @Id
        - @GeneratedValue
        - @OneToMany(mappedBy="")
        - @ManyToOne
        - @JsonIgnore
- NoSQL
    - MongoDB
        - show dbs - to list existing databases
        - use myDatabase - to create a database
        - db.myCollection.insert({someField: 'someValue'}) - to insert
        - db.myCollection.find() - to find all documents
        - db.myCollection.find({predicate}, {projection}) - to find all documents
            - projection: {field1: 1, field2: 0} ⇒ _id will be included
        - db.myCollection.find({_id: ObjectId("asdfasdfasdf")}) - to retrieve by ID
        - db.myCollection.find({$and : [{someField: 'someValue'}, {predicate2}, {predicate3}])
        - db.myCollection.find({someField: 'someValue'})
        - db.myCollection.update()
            - db.myCollection.update({predicate}, {the update})
            - db.myCollection.update({predicate}, {$set: {field1: newValue1, field2: newValue2}})
            - db.myCollection.update({predicate}, {field1: newValue1, field2: newValue2}) -- this replaces the document
        - db.myCollection.remove()
    - Mongoose
        - schemas
            - const mySchema = mongoose.Schema({
                - title: String,
                - author: {type: String, required: true},
                - semester: {type: String, enums: ["FALL", "SPRING", "SUMMER"]}
                - published: Date,
                - seats: {type: Number, default: 24}
            - }, {collection: "courses"})
        - models
            - const myModel = mongoose.model("MyModel", mySchema)
            - myModel.find()
            - myModel.findById()
            - myModel.findOne()
            - myModel.create()
            - myModel.updateOne({WHERE}, {UPDATE})
            - myMode.removeOne()
    - SQL Injection

        - ~~Types:~~
            - ~~1 = 1~~
            - ~~OR " = "~~
        - ***PreparedStatement()***
            - ps = connection.prepareStatement("SELECT * FROM myTable WHERE someField=?")
            - ps.setString(1, "someValue")
    - Transactions:
        - ACID
            - Atomic
            - Cosisntent
            - Isolate
            - Durability - permanent
        - Demarcate transactions
            - turn off auto commit
            - commit - make changes permanent
            - rollback - undo any changes in the transaction
        - Potential problems — Isolation levels
            - Dirty Read — READ-COMMITTED
            - Non-Repeatable reads — REPEATABLE-READS
            - Phantom reads — SERIALIZABLE — slowest
    - Indexes
        - Full table scan
        - Unique index on a primary key
        - Non-unique index on a non primary key
        - Optimize File IO
            - buffers containing multiple records
        - Given a set of realistic records and File IO reads, compute whether its better to use index or full scan
        - Composite keys
    - ~~XML~~
        - ~~redundancy~~
        - ~~inconsistency~~