AUTOR Design Discussion

Team 24: Tyler Enck, Deana Franks, Josh Kersey

Application Design

While developing our software solution, our team made several changes to our initial ER diagram to better capture the constraints given in the problem statement. First, we chose to combine Customers with the Employee ISA hierarchy to form a single Users ISA hierarchy. This simplified the login functionality and allowed us to consolidate the relationships between users and service centers.

Another significant change that we made to our initial design was representing each of the maintenance schedules as a maintenance service instead of a separate entity composed of several individual maintenance services. This modification made it so that both subclasses of the Services entity could be individually scheduled, where in our original design the MaintenanceService subclass could only be scheduled as part of a maintenance schedule.

Our team also elected to remove the Invoices entity from the design, as much of the information stored in Invoices was also being captured by the ServiceEvent entity. Leaving Invoices in the design would introduce redundancy and additional functional dependencies, and would likely require further refinement to reduce to BCNF or 3NF.

Lastly, a few additional entities were added to the ER diagram to model use cases and functional requirements that had been overlooked in our original design. We added the SwapRequest entity to store swaps between mechanics. We also introduced a Schedule entity that kept track of each Mechanic's schedule, making it much easier to capture the minimum mechanics and overbooking constraints.

Functional Dependencies and Normalization

As shown in more detail below, the final design that our team settled on is in 3NF. While we were able to reduce most of our relations to BCNF, the main thing preventing our full design from being in BCNF was the differences in pricing and hours with respect to service centers. Decomposing the Prices relation into different relations for pricing and hours would prevent us from preserving the functional dependency in Prices that service events for a particular model always have the same hours. Therefore, we opt for a 3NF design to guarantee dependency preservation while minimizing redundancy.

ServiceCenters(centerId, minWage, maxWage, address, phone, satOpen, isActive)

- Functional Dependencies
    - centerId-> minWage, maxWage, address, phone, satOpen, isActive
- This is in BCNF.

Receptionists(employeeId, centerId, salary)

- Functional Dependencies
    - employeeId, centerId -> salary
- This relation is in BCNF

Mechanics (employeeId, centerId, wage)

- Functional Dependencies
    - employeeId, centerId -> wage
- This relations is in BCNF form.

RepairServices(serviceId, category, name)

- Functional Dependencies
    - serviceId -> name, category
- This relation is in BCNF form.

MaintenanceServices(serviceId, name, serviceType)

- Functional Dependencies
    - serviceType -> name
- This relation is not in 3NF or BCNF so we added the below BCNF relation. After this addition, MaintenanceServices is in 3NF
    - MaintHasServices(serviceId, name, serviceType)

MaintServicePriced(serviceId, centerId, model, price, hours)

- Functional dependencies
    - serviceId, centerId, model -> price, Hours
    - serviceId, model -> hours
- This is not 3NF or BCNF so, we create the relation that is in 3NF
    - Prices(centerId, model, priceTier, price, hours)

RepairServicePriced(centerId, serviceId, model, price, hours)

- Functional Dependencies
    - centerId, serviceId, model -> price
    - serviceId, model -> hours
- This is not 3NF or BCNF, so we create the relation that is in 3NF
    - Prices(centerId, serviceId, model, price, hours)

Customers(customerId, centerId, firstName, lastName ,address, status, active)

- Functional dependencies
    - customerId -> firstName, lastName, address, status, actice
    - customerId -> centerId
- This relation is in BCNF form.

CustomerVehicles(vin, customerId, model, mileage, year, lastMClass)

- Functional dependencies
  vin -> mileage,year, lastMclass
  vin -> model
  customerId -> vin
- This is in BCNF form.

ServiceEvents(eventId, vin, mechanicId, scheduledService, week, day, startDate, startTimeSlot, endTimeSlot, totalPrice, totalPaid, completed, invoiceStatus)

- Functional dependencies
  - EventId -> scheduledServices, startDate, week, day, startTimeSlot, endTimeSlot, totalPrice, totalPaid, completed, invoiceStatus
  - Eventid -> vin
  - eventid -> mechanicId
- This is in BCNF form.

Table of Constraints:

Project Description Text	**Our Model**
Each employee is associated with only one service center	Foreign key with not null constraint from Users to ServiceCenters
A customer is associated with at least one vehicle which is identified by globally unique vin number	Foreign key with not null constraint from CustomerVehicles to Customers
each center has a manager who manages all employees	Foreign key constraint from Managers to ServiceCenters
Each employee can only play one role at a time	Foreign key with not null constraint from Users to Roles
Individual services belong to only one of the 6 repair service subcategories	Foreign key with not null constraint from RepairServices to RepairServiceCategory
Each mechanic works no more than 50 hours a week	Java business logic to calculate total time scheduled before scheduling a mechanic for additional services
Each customer has a status field stating if their account is in good standing or not (i.e. has an outstanding balance)	Trigger to update status field on creation/payment of a ServiceEvent
There are always at least 3 mechanics present at any given time	Java business logic to check availability of mechanics before scheduling
Customers become “inactive” if their profiles no longer include any vehicles	Triggers to update customer status on add/delete vehicle
Mechanics wage between min and max	Java business logic to check wage

wage of ServiceCenter	before adding Mechanic (was intended to be check constraint but we ran into issues with going between tables)
Scheduling will need to avoid double-booking	Primary key constraint in schedule for (mechanicId, serviceCenterId, week, day, timeslot) to prevent multiple entries at same time

Advanced SQL Features

When it comes to advanced SQL features, our team implemented a number of triggers for ensuring the database fulfilled the desired functionality requirements along with a few constraints being checked via a combination of SQL queries and Java business logic through the JDBC API. In terms of triggers, we created a trigger for updating a customers standing after they have scheduled a service, another for updating a customer's status to inactive after they have deleted a car from their profile, and our last trigger was used for updating the customer's status to active when they add a car to their profile for the first time. In regards to ensuring constraints were checked through Java business logic via JDBC, we implemented checks for ensuring a swap request procedure must verify that indeed the time slot range is assigned to the mechanic being requested, along with checks to ensure that the mechanic being requested during a swap request would not become overbooked (>50hrs) or double-booked. Additionally, we also used sequences to auto-increment attributes within the tables of our schema where deemed necessary.