Sharif University of Technology
Department of Computer Engineering

Database Design Project - Phase 2

Sahand Akramipour, Arash Parsa, Alireza Aalaei
June 29, 2024

1 SQL Compatibility

1. target demographic specialization: split into 3 individual entities
2. reservation split into video and banner reservation entities separately
3. ads specialization: split into separate ad entities
4. each reservation is now connected to the corresponding ad type
5. account specialization removed. only publisher/advertiser types remain. no specific changes since nothing was directly connected to account
6. new lists added called "CampaignVideo" and "CampaignBanner" to break the M:M relation between campaign and video/banner ads. basically a list of the video ads and banner ads the campaign will get.
7. made corrections on certain Primary Key determinations. for example made certain adslot's primary key is the tuple (websiteID, adslotID)
8. relation between advertisers and bills removed. instead an entity called Advertisers Campaigns is added
9. SIM provider entity added to break the M:M relation between SIM card and SMS provider. basically a list of SIM cards the provider will be sending the messages to
10. SMSID added as a partial key to the SMS entity
11. SMS Campaign ID added to SMS campaign as a partial key
12. Bill ID is considered a partial key in Bills entity
13. new Goals entity added with attributes goalID and goalDescription.

2 Anomalies Before Normalization

1. Anomaly: Redundancy and update anomalies due to repeated data about demographics across different entities. Example: If demographic information (e.g., age range, location, gender) is duplicated across multiple tables (campaign_demographics, website_demographics, catalog_demographics), updating demographic data in one place may lead to inconsistencies in others.
2. Insert Anomaly without splitting target demographic specialization into separate entities (such as campaign_demographics, website_demographics, catalog_demographics): Example: If a new campaign is inserted with demographic details directly embedded, the same demographic data might need to be inserted redundantly into other tables where the campaign is relevant (e.g., website or catalog demographics), leading to duplication of data and potential inconsistencies if updates are not properly synchronized.
3. If reservations are not split into video and banner reservations: Example: When inserting a reservation for a specific ad slot, it may not be clear whether it pertains to a video ad or a banner ad. This ambiguity can lead to errors in reservation management and ad allocation.
4. Anomaly: Functional dependency and update anomalies if reservations for video and banner ads are not separated. Example: Without separate entities (video_reservation and banner_reservation), changes in reservation details (such as price, duration) for one type could affect the other type unintentionally.
5. Anomaly: Non-atomicity and redundancy in ad-related data. Example: Without distinct entities for different types of ads (e.g., video_ads, banner_ads), managing specific attributes related to each type (e.g., video file for video ads, PNG file for banner ads) becomes complex and may result in redundancy.
6. Delete anomaly between Advertisers and Bills Deleting an advertiser might require deleting associated bills manually or leaving orphaned bills without an advertiser reference, causing referential integrity issues and making it difficult to manage billing data.

3 Normalization

1. ”goals” entity added to take ”campaigns” to 1NF (goals used to be a multivalued attribute)
2. ”Advert Format” entity added to take ”website” to 1NF (available advert formats used to be a multivalued attribute)
3. ”Adslot” now has slotID as partial key and the primary key consists of (URL, slotID) to take it to 3NF (used to have partial dependencies on the prime attributes. prime attributes were the website URL (foreign key) and slotID. this way with the composite key there are no partial dependencies)
4. available advert formats used to be an attribute for websites, which made it not in 1NF. for normalization advert format is now an entity that has a M:1 relation with websites
5. removed the direct relation between Bills and Advertisers. in the example set of tables given to us for the query section of the first phase, the table Bills also included an AdvertiserID attribute, implying a direct relationship. we implemented that in the first phase. but with that implementation we get a new FD which is CampaignID $\longrightarrow$ AdvertiserID which results in a partial dependency. removing this direct relation and implementing a new AdvertiserCampaigns entity we can get advertiserID and BillID in the same table using two joins now. this way our structure is in 3NF.
6. added the table available_ad_opportunities to normalize the Publisher table. Available ad opportunity used to be a tuple making Publishers not in 1NF

Here are three examples of functional dependencies:

- Campaign

$$Campaign(CampaignID, AdvertiserID, GoalListID, Budget, StartDate, EndDate, Name) \tag{1}$$

$$\begin{aligned} FDs: \{&CampaignID \longrightarrow AdvertiserID \\ &CampaignID \longrightarrow Budget, StartDate, EndDate, Name \\ &CampaignID \longrightarrow GoalListID\} \end{aligned} \tag{2}$$

$$\text{Primary Key}: \{CampaignID\} \tag{3}$$

  Used to have goals as a tuple of names which was not even 1NF. with the addition of GoalListID we take the table to 3NF and remove anomalies like an update anomaly. if we separated them and had a row for each goal then an update anomaly exists. for a change in goal name every campaign having them has to change their row.

- Bills

$$Bills(BillID, CampaignID, BillStatus, BillRate, BillAmount) \tag{4}$$

$$FDs: \{BillID, CampaignID \longrightarrow BillStatus, BillRate, BillAmount\} \tag{5}$$

$$\text{Primary Key}\{BillID, CampaignID\} \tag{6}$$

  according to normalization changes above, number 5, the anomaly present was for example an insert anomaly. no new Bill addition was possible without an advertiserID included. this way we just need campaignID for it since it happens in a campaign.

- AdSlot

$$AdSlot(SlotID, URL, SlotLocation, SlotType, SlotDimensions, No.Ads) \tag{7}$$

$$FDs: \{URL, SlotID \longrightarrow SlotLocation, SlotType, SlotDimensions, No.Ads\} \tag{8}$$

$$\text{Primary Key}: \{Slot, URL\} \tag{9}$$

  without using URL and SlotID as the primary key, we have these FDs:

$$\begin{aligned} \{&URL, SlotID \longrightarrow SlotLocation \\ &SlotID \longrightarrow SlotType, SlotDimensions, No.Ads\} \end{aligned} \tag{10}$$

  which is not in 3NF because of the present partial dependency that way for example delete anomalies exist. if we want to update the URL of a website we have to update all the rows and columns that URL exists in.

4 Query Results

4.1 Find All Active Campaigns

Explanation: This query selects the 'campaign_id' and name of all campaigns where the current date is between the start _date and end _date.

4.2 Find Contact Info for Advertisers in Industry "Technology"

Explanation: This query selects the 'contact_info' of advertisers whose 'industry_type' is "Technology".

4.3 Return CampaignID and CTR (clicks / impressions) for all Campaigns

Explanation: This query calculates the Click-Through Rate (CTR) for each campaign by dividing the number of clicks by impressions, casting the result to a float for precision, and selecting the 'campaign_id' and calculated ctr.

4.4 Find all Ad Slots in Websites with Over 100,000 Traffic

This query joins the ‘ad_slot’ and ‘website’ tables to find ad slots located on websites with an average traffic greater than 100,000, selecting various details about those ad slots.

4.5 Return (bill_ID, bill_amount, date_issued) for all Advertisers in Industry ”Finance”

Explanation: This query joins the ‘bills’, ‘campaign’, ‘advertisement_campaign’, and ‘advertisers’ tables to find the ‘bill_id’, ‘bill_amount’, and ‘date_issued’ for all advertisers in the ”Finance” industry.

4.6 Return Slot_ID for all Ad Slots with CTR above 0.05

Explanation: This query joins the ‘ad_slot’, ‘click’, and ‘performance’ tables to find unique ad slot IDs with a CTR (clicks/impressions) greater than 0.05.

4.7 Find all Advertisers with no Active Campaign

Explanation: The subquery checks if there are any rows in the ‘advertisement_campaign’ table (joined with the campaign table) where: The ‘advertiser_id’ matches the current row’s ‘advertiser_id’ from the main query. The current date (CURRENT_DATE) falls between the ‘start_date’ and ‘end_date’ of the campaign.

4.8 Find the Total Money Spent by Each Advertisers

Explanation: This query calculates the total money spent by each advertiser by joining the ‘bills’, ‘campaign’, ‘advertisement_campaign’ and ‘advertisers’ tables and grouping the results by the advertiser ID and name, summing up the ‘bill_amount’ for each group.

4.9 Find all Campaigns with Campaign Demographic with Females Between the Ages 18 - 24 with Location ”New York”

Explanation: This query joins the campaign and ‘campaign_demographics’ tables to find campaigns targeting females aged 18-24 in New York, selecting the ‘campaign_id’ and name. the specific selection is stated in the WHERE clause of the query.

4.10 Find all Campaigns with Budget over 50,000 Dollars

This query selects the ‘campaign_id’ and ‘name’ of campaigns where the ‘budget’ is greater than 50,000.

5 Indexing

5.1 Campaign Table Index

Primary Key Index: The campaign_id is already indexed as the primary key, which is good for quick lookups by campaign ID. Index on Date Columns: Since queries often filter campaigns by date (start_date and end_date), creating indexes on these columns can improve performance for queries that involve date range conditions.

5.2 Advertisers Table Index

Primary Key Index: The advertiser_id is already indexed as the primary key. Index on Industry Type: If queries frequently filter or join based on industry_type, creating an index on this column can improve performance.

5.3 Bills Table Index

Composite Index: Create a composite index on (campaign_id, bill_id) if queries often involve filtering bills by campaign or retrieving bills for a specific campaign.

6 REST API

The REST API of this database is implemented in Java using SpringBoot framework. It has many different parts, and here is an example:

7 Ad Slot API

This Java code defines a REST controller for managing "AdSlot" entities in a Spring Boot application. Here's a detailed breakdown of each part:

Package and Imports

```
package com.example.restapi.controller;

import com.example.restapi.model.AdSlotDto;
import com.example.restapi.service.AdSlotServiceImp;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
```

- **Package Declaration:** The `package com.example.restapi.controller;` line specifies the package this class belongs to.
- **Imports:** Necessary classes from Spring and other packages are imported. These include `AdSlotDto` for the data transfer object, `AdSlotServiceImp` for the service layer, and various Spring annotations and classes for handling HTTP requests and responses.

Class Declaration

```
@org.springframework.web.bind.annotation.RestController
@RequestMapping(value = "/api/AdSlot")
public class AdSlotRestController {
```

- **Class Annotation:** `@RestController` indicates that this class will handle HTTP requests and return JSON/XML responses.
- **Request Mapping:** `@RequestMapping(value = "/api/AdSlot")` sets the base URL for all the endpoints in this controller to `/api/AdSlot`.

Constructor

```
    private final AdSlotServiceImp adSlotServiceImp;

    @Autowired
    public AdSlotRestController(AdSlotServiceImp adSlotServiceImp) {
        this.adSlotServiceImp = adSlotServiceImp;
    }
```

- **Dependency Injection:** The `AdSlotServiceImp` service is injected into the controller using the constructor and `@Autowired` annotation.

Endpoints

Get All AdSlots

```
@GetMapping("")
public ResponseEntity<List<AdSlotDto>> getAllAdSlot() {
    return ResponseEntity.ok(adSlotServiceImp.getAll());
}
```

- **HTTP GET:** `@GetMapping("")` maps this method to the base URL `/api/AdSlot`.
- **Return Type:** The method returns a list of `AdSlotDto` wrapped in a `ResponseEntity` with HTTP status 200 (OK).

Get AdSlot by ID

```
@GetMapping("/{id}")
public ResponseEntity<AdSlotDto> getAdSlot(@PathVariable int id) {
    try {
        return ResponseEntity.ok(adSlotServiceImp.getAdSlotById(id));
    } catch (Exception e) {
        return ResponseEntity.ok(null);
    }
}
```

- **HTTP GET:** `@GetMapping("/id")` maps this method to `/api/AdSlot/{id}`.
- **Path Variable:** The `@PathVariable int id` annotation binds the `id` from the URL to the method parameter.
- **Error Handling:** In case of an exception, the method returns a `ResponseEntity` with a null body.

Create AdSlot

```
@PostMapping("/create")
@ResponseStatus(HttpStatus.CREATED)
public ResponseEntity<AdSlotDto> createAdSlot(@RequestBody Integer webIID,
    ↪ String slotLocation, String slotDimension, Integer numberOfAds, String
    ↪ slotType) {
    AdSlotDto adSlotDto = new AdSlotDto(webIID, slotLocation, slotDimension,
        ↪ numberOfAds, slotType);
    return ResponseEntity.ok(adSlotServiceImp.createAdSlot(adSlotDto));
}
```

- **HTTP POST:** `@PostMapping("/create")` maps this method to `/api/AdSlot/create`.
- **Response Status:** `@ResponseStatus(HttpStatus.CREATED)` sets the response status to 201 (Created).
- **Request Body:** `@RequestBody` binds the request payload to the method parameters.
- **Creating AdSlot:** An `AdSlotDto` object is created and passed to the service layer.

Update AdSlot

```
    @PutMapping("/{id}/update")
    public ResponseEntity<AdSlotDto> updateAdSlot(@RequestBody String slotLocation,
        ↪  String slotDimension, Integer numberOfAds, String slotType,
        ↪ @PathVariable("id") int currentId) {
        AdSlotDto adSlotDto = new AdSlotDto(slotLocation, slotDimension,
            ↪ numberOfAds, slotType);
        return ResponseEntity.ok(adSlotServiceImp.updateAdSlot(currentId, adSlotDto
            ↪ ));
    }
```

- **HTTP PUT:** `@PutMapping("/id/update")` maps this method to `/api/AdSlot/{id}/update`.
- **Updating AdSlot:** An `AdSlotDto` object is created with the updated information and passed to the service layer.

Delete AdSlot

```
    @DeleteMapping("/{id}/delete")
    public ResponseEntity<String> deleteAdSlot(@PathVariable int id) {
        try {
            adSlotServiceImp.deleteAdSlotById(id);
            return ResponseEntity.ok("Deleted");
        } catch (Exception e) {
            return ResponseEntity.ok(e.getMessage());
        }
    }
```

- **HTTP DELETE:** `@DeleteMapping("/id/delete")` maps this method to `/api/AdSlot/{id}/delete`.
- **Deleting AdSlot:** The method calls the service to delete the AdSlot by ID and returns a confirmation message.

Summary

This controller provides CRUD (Create, Read, Update, Delete) operations for ”AdSlot” entities. Each method corresponds to an HTTP verb and performs a specific action by interacting with the service layer, which handles the business logic.

8 Changes

In the following images, the left image has changed to the right one to maintain normalization and SQL compatibility.

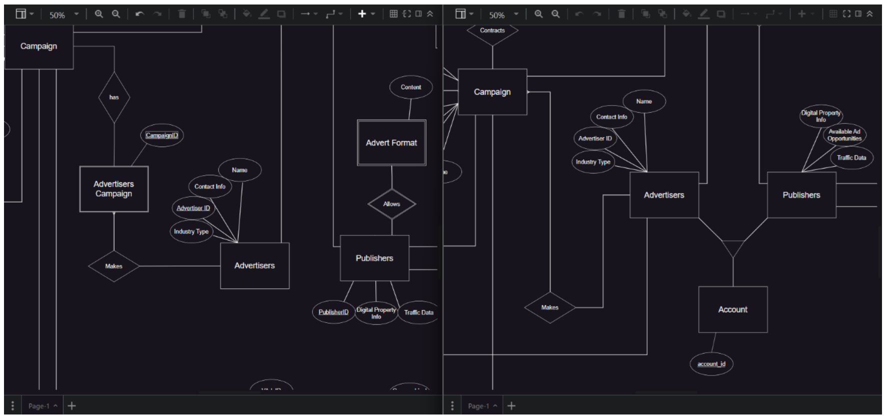

Figure 1: Account: split specialization

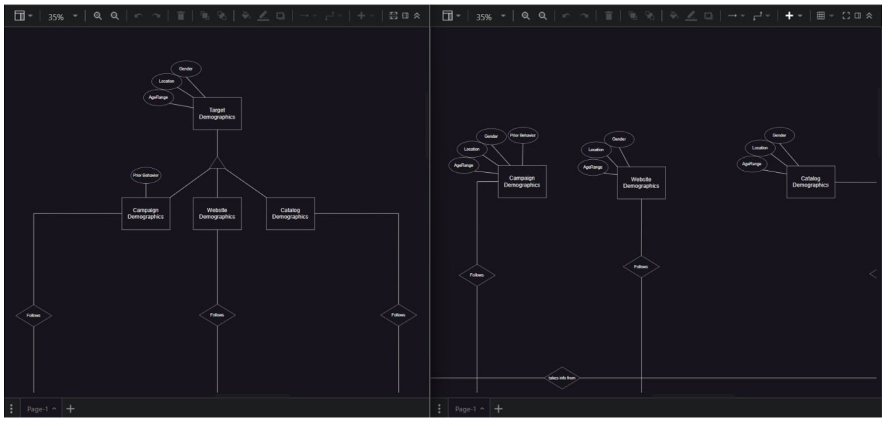

Figure 2: Demographics: split specialization

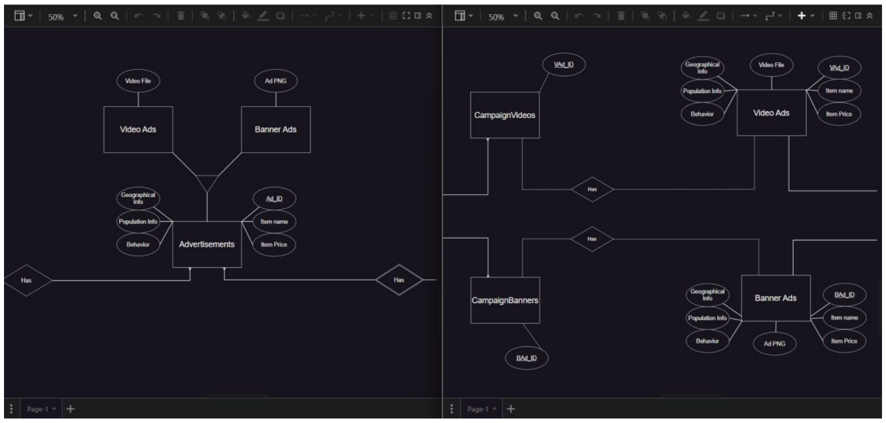

Figure 3: Advertisements: split specialization

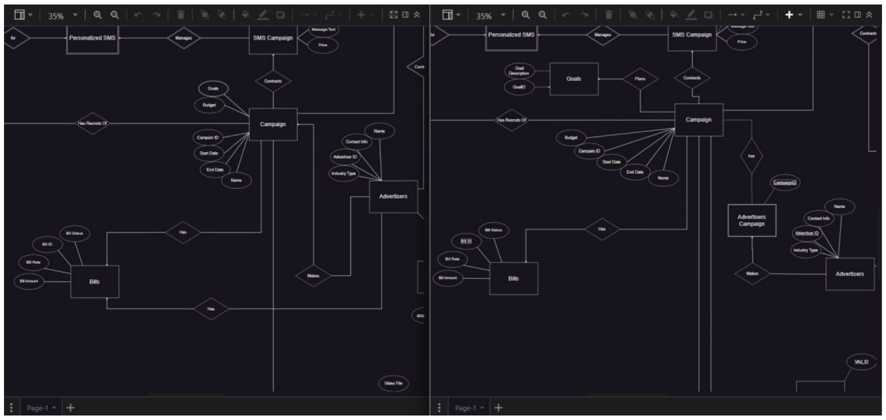

Figure 4: Bills: remove foreign key to campaign

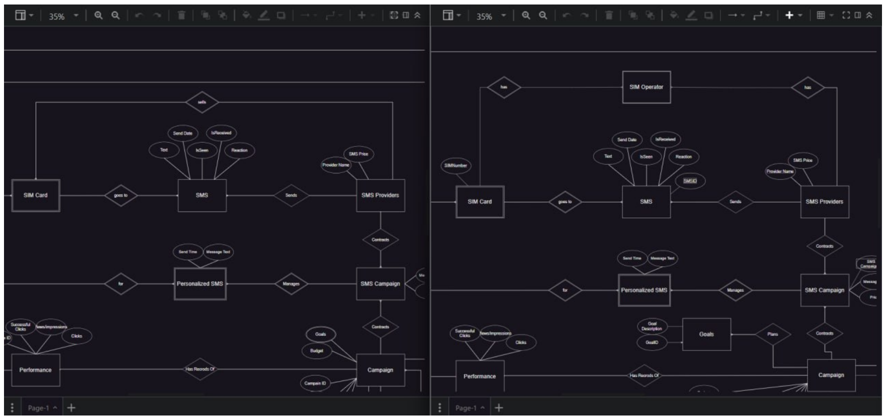

Figure 5: Sim Operator: to maintain normalization

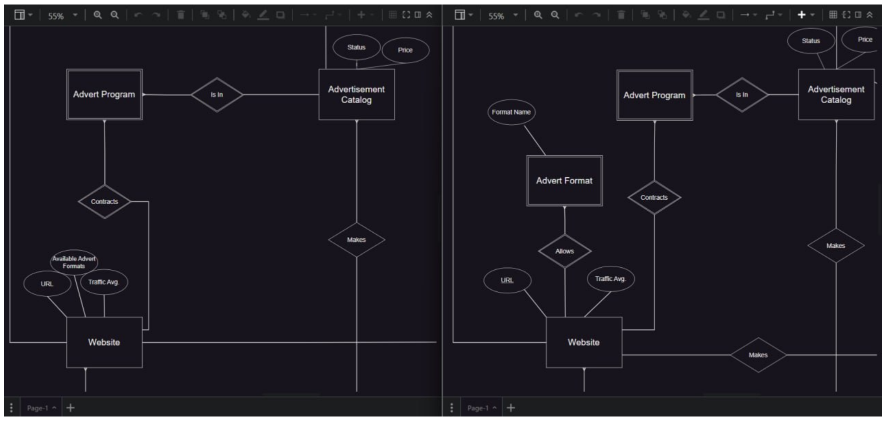

Figure 6: Website: add advertFormat to maintain 1NF

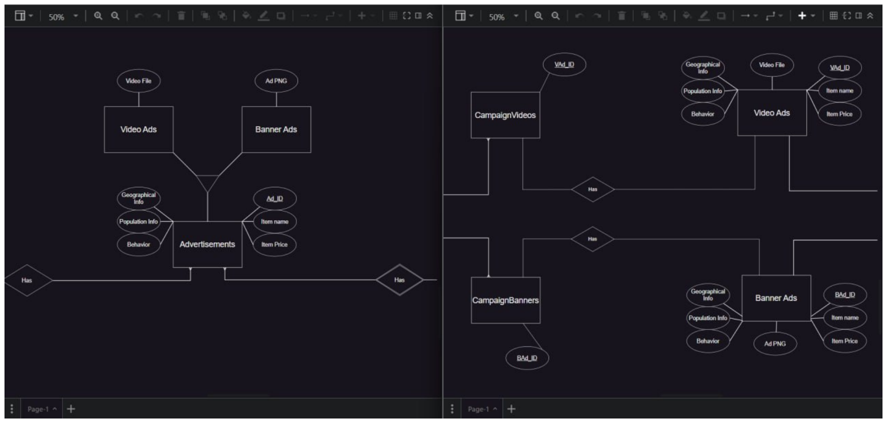

Figure 7: Reservation: because we split specialization of advertisements