Database-Systems-236363 Homework 2 Report

Yosef Goren & Yonatan Kahalani

June 8, 2023

Part I
Wet

1 Entities

The entities relevant to our DB are:

- Photos: `ptable(photoID, description, size)`
- Disks: `dtable(diskID, company, speed, free_space, cost)`
- RAMs: `rtable(ramID, company, size)`

2 Relations

The relations that we will store directly in our DB are:

- Photo On Disk: `(podtable(photoID, diskID))`
- RAM On Disk: `(rodtable(ramID, diskID))`

3 Relations & Properties

Note that for each of the Entity tables, the ID is used as a primary key, and that each of these ID's are used as foreign keys in the tables that link the Disks table to both the RAMs table and the Photos table.
Also, we not that we have a 'many to one' relation in two cases:

1. $photoID \longrightarrow diskID$ (in the `podtable`)
2. $ramID \longrightarrow diskID$ (in the `rodtable`)

4 ERD Design Considirations

Our main considirations upon choosing this design were it's ability to be very space efficient, since there are no duplicates in a non-key field and there are also no potential NULL attributes allowed in our table.
This design also made sense to us due to the requirement of the CURD API - to which this design enabled a elegan implementation.
Moveover - with this implementation we were also capable of implementing the more advanced APIs efficiently.

5 API Implementation Design Considirations

5.1 CRUD API

Most of the CRUD API functions were simple Getters & Setters and hence did not require much in terms of design; yet, we were able to simplify the implementation of the CRUD API by having implicit deletes thanks to proper usage of CASCDE semantics in the definition of our tables.
Additionally, some of the methods required more sophisticated implementations; noteably `deletePhoto()` had a more complex implementation since we had to be able to recognize when an update shouldnt be made AND make an update if nessecery - all in one querry.

5.2 Basic API

In the Basic API there were a few methods with intresting design constrains:

- `removePhotoFromDisk()`: In this method - the main issue was we need to updated the free space of the associated disk properly - even if the photo did not actually exist. While the trivial implementation of this method is quite simple - this edge case required us to find a much more complicated implementation - in which we relay on the properties of the 'SUM' aggregation operator to handle the 'empty case'.

- `isCompanyExclusive()`: In this method, we are provided with a specific disk (id), and we want to check if the disk's company matches that of all of it's associated RAMs. The tricky part was that we need to be able to check if an entier group (nested querry) matchs the same value (which was also a nested querry); to solve this - we used the 'ALL' quantifier - which was actually it's only use case for us in this project.

5.3 Advanced API

The interesting method in the Advanced API was `getClosePhotos()` which not only required the longest query we have made so far, but also required us to utilize the 'GROUP' operator while filtering the associated groups using a nested query that counts the number of matching photos - in order to be able

to determine the proportion of the photos which were in the same group as the one asscoicated with the argument `photoID`.
Morever - the hardest part of the design of this method (and perhaps the project) was the rquirement to handle the 'empty case' - in which we have to return all photos if the provided photo is not on any disk.
This esstially required a branching in the logic - but since we were only allowed to use a single atomic sequence of queries - we have created a modular query that uses the 'UNION' operator to effevely implement an 'if' statment without actually using 'if'.