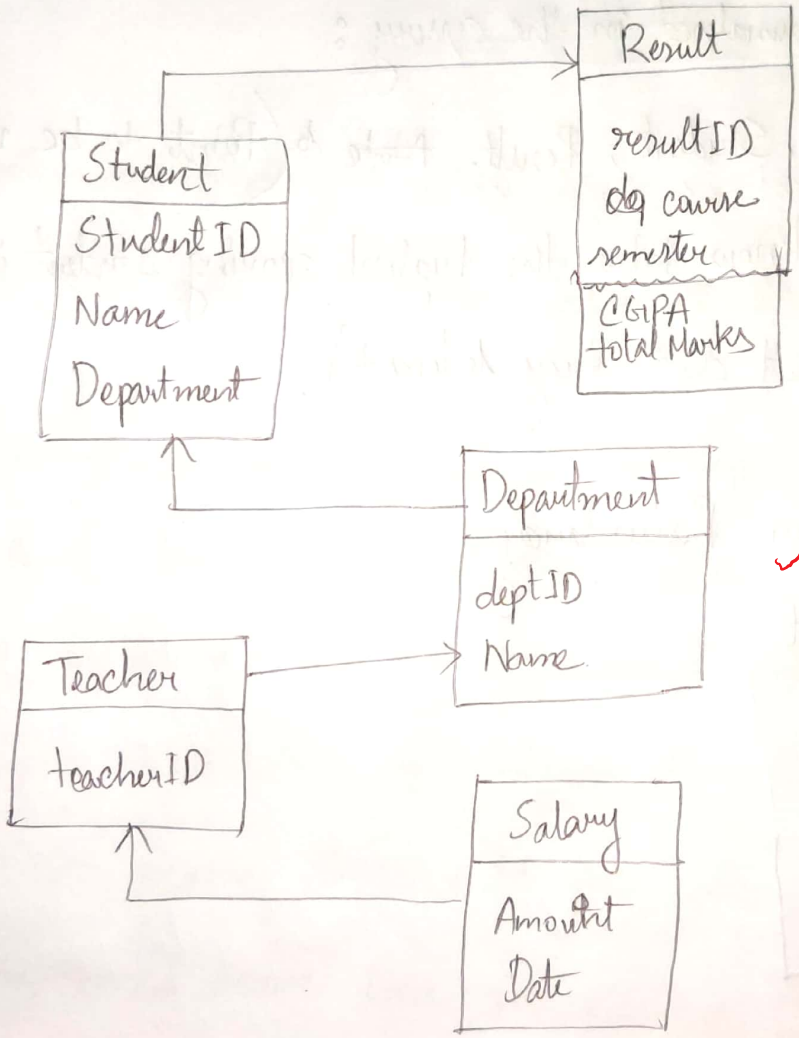
Ans to Q.1 (a)

Normalised DB with 3-5 tables & relations between:



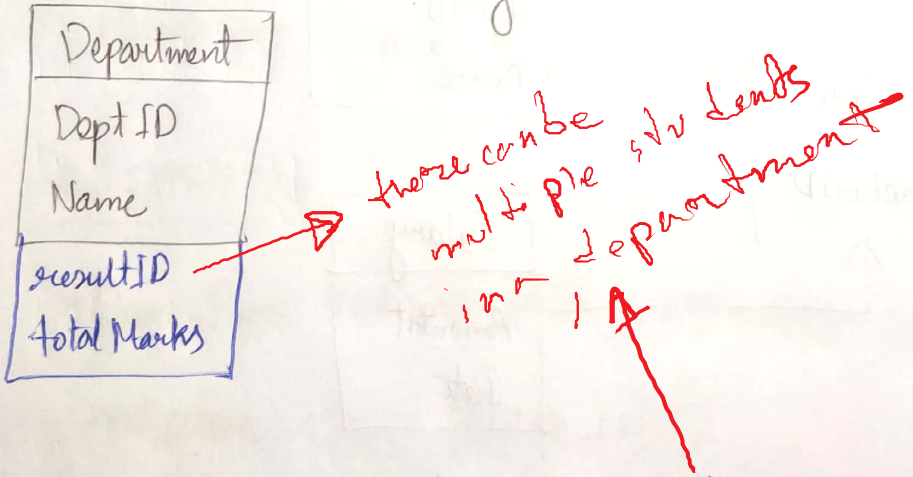
The query to be optimized:

"Which dept has the student with highest score?"

Tables to be normalized in the query:

Department, Student, Result. ~~Note~~ (Point to be noted is we don't want to know who the highest scoring student in we just want to know what dept. they belong to)

DB with Column Redundancy:



resultID can also be replaced with student ID.

tables joined in denormalized DB: only the Department tables give enough info.

(b) Tables to be joined in denormalized DB:

Result can be obtained from the Department Table only.

Ans to Q.1 (b)

* instead of dividing team by layers we can divide teams based on module, with developers from all the layers in one team.

As Martin fowler said, "developers don't have to be full stack, but teams should be". In smaller systems, there might not be much difference between layer-based & domain-based approach, but as the application gets bigger, the layers start getting packed with too many things. In these scenarios, module-based teams face no problems because there is no change nesecarry in already existing teams.

Ans to Q.1 (c)

When organizing source code organizing them based on application layer can create the following issues:

(1) The more features are added into the software, the more complicated the hierarchy gets.

(2) If there is a bug in a module and we don't know which layer the bug is in, we have to go through all the folders, look for the specific ~~fold~~ file of that module among 10 possibly 100+ other files, which is a big hassle.

∴ its better to use module as top-level folder can keeping the layered files (i.e. html, css, js) inside that folder in a preferable manner.

Ans to Q·2(a)

My opinion about the plan is given below:

(1) Since they are using blueprint UML their main focus is complete design.

(2) the two UML designers are probably senior developers & have more experiences than the 4 programmers implementing it.

(3) The process they are using forward engineering.

(4) The UML design needs to be complete enough to implement

(5) The blueprint should be detailed enought & straightforward so that programmers can implement them.

alternative suggestion: Instead of working as 2 seperate modules, the both parties can work on coding and design. That way errors can be fixed easily and one party won't continously have to knock the other one if they get stuck. They can also use automated tools if the UML gets too tricky. Another interesting

approch can be reverse engineering, where the code is written first then as UML is used to explain it.

Ans to Q. 2 (b)

I think option 2 is a better approach. We can use the current monolith using an API and build the new feature as microservices. That way the new features can continue to use the monolith.

if we look at the microservice characteristic "Evolutionary design" in Martin Fowlers article, we see that the migration from monolith to microservices need to be evolutionary, which means we implement new functionalities in services referring to the original monolith & gradually split services from the actual monolith

This can also be applied from the perspective of decentralized data management, where we decouple the databases.

Zalin - 17004203 2

[7]

Ans to Q. 2 (c)

in case of horizontal scaling, they are increasing the service capacity by adding (CPU, memory etc). They are not adding new instances of the services. In that case, scaling of database is ~~not~~ absolutely necessary ?

0/

decentralized data management is one of the key characteristics of micro services. This means, every microservice should should have a data storage mode of its own. So if there is a new service, it should have an instance of an old DB service or a completely new DB systems. However in the currently scenarion we are not scaling the services vertically, which means they are adding new instances of the services. so they need a data storage system of their own.

Ans to Q. 3(a)

difference between services & libraries:
Services are out-of-process components who communicate with a mechanism such as a web request of, remote call. libraries are components that are linked to a program & called using in-memory fn calls. services are independantly deployable but libraries are deployed with the program. library calls are cheaper than service calls. Service interfaces are more explicit than libraries. so most language don't have a good mechanism to define them.

10/

major benefit of services: services can be invoked on demand. it doesn't have to be active all the time. libraries don't have this advantage.

drawback of services: calling services from out-of-process is an expensive operation. library usage is cheaper.

Ans to Q. 3(b)

if we revert the database into a previous version then the current users who are using the application might not have access to all the data. This can be a problem. In that case, we need to notify the developers can collaborative necessary changes.

An alternative to this can be keeping a repository for the production database that contains its previous versions. so the necessary version compatible with code can be pulled.

if we want to make large changes to the production data, then a better approach would be to take the production DB down and then make changes.

Ans to Q.3 (e)

in order to keep the data in version control but skip it from being inserted in production database, we can keep separate files for test data & migration script. The script will be added in the production database, but the data can stay seperated. ~~They can~~ The data can also be stored in different folders, so the code doesn't get complicated.