Project Report

Platform

As our respective Schedule didn't allow us to meet often, we had to find a way to synchronize our work remotely. We choose to use git for the code and the other resources.
Cloudbees was chosen for the database, as it is a free host that allows mysql DB, with remote admin(we used phpmyadmin), and right management for group access to various part of a project.
Cloudbees has no native php hosting. So we used the quercus and resin java libraries to be able to use the php code already written for the first and second part of the project on a java server.

Phpmyadmin was chosen because of the ease it has to import .cvs and other files directly in tables. It also provide facilities to manage the structure and constraints of the tables.

Installation

To run the project you will need to:
1. import the given DB dump on a mysql DB.
2. run the "views.sql" script to add views to the DB.
3. run a php server. The php code for the web app is in the webapp folder, and is not linked to cloudbees in any way.
4. put the webapp folder on your server and change the config.php file to link to your DB.
5. open the index.php page on the server.
*The latest version of the project is online on Cloudbees, here.

Use

1. Use the search text field to look for what you want then follow hyperlinks to navigate through informations.
or
2. Use special queries links to access special queries information.

You can see some pictures of the working web app in the folder "images"

Design Remarks

Design Choices

Team table
The use of team table allows the management of a team receiving a medal (rather than giving a medal to each athlete of the team, in football teams for example).

This solution can still be used for an athlete to participate alone, being a team of one person.

We also choose to make the team represent a country. By this way we can manage cases where athletes participate to several games for different countries (change of country or country modification like USSR, Yugoslavia etc.). The disadvantage of this solution is to have to check that an athlete does not represents differents countries for the same games or takes part to the same event in different teams. Several queries on athlete are also a little more complex, as we need a membership table that links the teams and athletes.

We decided to use a team table with whole ranking instead of recording medals. Even if it takes more space, it provides an intrinsic coherence (and more information).
As the data doesn't provide info on other ranks, athletes that didn't get a medal don't have a team, and thus are not represented as having participated in games and events.

The given data don't indicate what events was a team competition and what was not. It also doesn't give any information regarding the athletes and in which team they participated. We only have infos on teams that won a medal. Theses we could link to the corresponding event and athletes, as the medal table indicate several athlete names when the event was a team event.

Required extra checks

Membership table

When adding an athlete to the Membership table, one needs to check

- that the athlete does not have two teams representing different countries for the same games.
- that the athlete is not a member of two different teams participating in the same event.

IOC code

We assume that each country has an ioc code and that it is unique. This allows us to use it as an index for the countries table. We invented ioc when none was existing.

No cascade behaviour for foreign keys

We left the default (restrict) behaviour for our foreign key. The reason is that even if there is a good reason to add cascade (e.g. between participation and event or between discipline and sport), the risk of deleting a lot of data is not balanced by the ease of using cascade. Moreover, there is practically no reason to delete elements in this table since it's role is mostly to record an history of Olympic games.

Enum-like entities

Season and Sport entities are very simple tables that just restrict the choice of values in Games resp. Discipline tables.

Indexes & performance of queries

We could notice the advantage of indexes while making queries on temporary tables to prepare the data for our final tables. Queries that required too much time to complete (a 300 sec timeout on phpmyadmin) were able to complete in approximatively 10-20 sec after we thought of creating indexes on the relevant column of the temporary table.

The downside is that there is an increase of the size of the DB due to the size of the index. In mysql, we could see an increase of 0.3 MB for the 18k entries Membership table, for example.

Performances :

Query	Time (sec)
I	0.5
J	0.45
K	1.5
L	0.15
M	0.2
N	1.5
O	65
P	0.15
Q	0.5
R	2.5
S	9
T	0.91
U	Unimplemented
V	2.5

Distribution of cost :

During queries implementation, we were really careful with nested queries. Thus there is no subquery with a parameter depending on the outer query to avoid complexity increase. So, we will not have this kind of costly structure.

In query 'V' (List top 10 countries according to their success on the events which appear at the Olympics for the first time), there is a lot of comparison about the rank. Because we have to order countries and select maximum values by gold-silver-bronze, there are big joins on unindexed ranking which slows the execution.

In query 'K' and all following queries (Compute which country in which Olympics has benefited the most from playing in front of the home crowd. The benefit is computed as the number of places it has advanced its position on the medal table compared to its average position for all Olympic Games. Repeat this computation separately for winter and summer games), we used views. This has a main impact on the query performance. In fact, there is a lot of minimum maximum extractions that requires twice the same table. Thus using views avoid useless duplication. The downside is that views has to created at each execution.

Query 'O' (For all disciplines, compute the country which waited the most between two successive medals) We didn't find any fancy way to implement it. We use a lot of cross products that tend to create very large views. Thus, a lot of time is spent to create and join those views.

Changes relative to the second part

The database structure has not changed from the project 01 and project 02. It still corresponds to the provided Schema.

The ranking rule is like that : 1 = gold medal, 2 = silver, 3 = bronze, 4+ = other ranks without medal, but we didn't have them in the given data, so we put some random data to see how it worked at first. In the given database, the other ranks are simply not set, as it didn't make much sense to create random team and assign random rank to 70'000 athletes. Instead, as said previously, we used the medal table to know the first tree teams/athlete in a competition.

If we wanted to signal a disqualification, we could put a -1 in the rank. Equalities should be managed by hand.