**DB Design**

**Tables:**

|  |  |  |
| --- | --- | --- |
| **Participants** | | |
| Field Name | Type | Comment |
| Aid | Integer | >0, FK |
| Sid | Integer | >0, FK |
| Payment | Integer |  |
| Place | Integer | NULL OR {1,2,3} |

|  |  |  |
| --- | --- | --- |
| **Athletes** | | |
| Field Name | Type | Comment |
| Id | Integer | >0, PK |
| Name | Text | Not Null |
| Country | Text |
| Active | Boolean |

|  |  |  |
| --- | --- | --- |
| **Sports** | | |
| Field Name | Type | Comment |
| Id | Integer | >0, PK |
| Name | Text | Not Null |
| City | Text |
| Counter | Integer | >=0, Default: 0 |

|  |  |  |
| --- | --- | --- |
| **Friends** | | |
| Field Name | Type | Comment |
| Aid1 | Integer | >0, FK  Aid1!=Aid2 |
| Aid2 | Integer | >0, FK  Aid1!=Aid2 |

Explanation:

* Athletes relation holds all the athletes in the DB using their ID’s as PK and saving all their relevant data (such as name, country and active status).
* Sports relation holds all the sports in the DB using their ID’s as PK and saving all their relevant data (such as name, city and active athletes counter).
* Participants relations connects between an athlete and a sport event which he participates in. using the athlete’s id and sport’s id as a tuple as a PK. Saving the data relevant to that specific athletes participating in that sport event such as Payment( important for observers) and Place in case the athlete has won a medal.
* Another table is the Friends which holds the friendship connections between athletes using their both ID’s as PK. We accomplished symmetric friendship by making a double-sided check and adding for such friendship only one row instead of two. e.g. - (A,B) : A is friend of B and B is friend of A.

**Views:**

|  |  |  |
| --- | --- | --- |
| **Observers** | | |
| Field Name | Type | Comment |
| Aid | Integer | >0, FK |
| Sid | Integer | >0, FK |
| Payment | Integer |  |

|  |  |  |
| --- | --- | --- |
| **MedalsScore** | | |
| Field Name | Type | Comment |
| AId | Integer | >0, FK |
| Golds | Integer |  |
| Silvers | Integer |  |
| Bronzes | Integer |  |
| Rating | Integer |  |

|  |  |  |
| --- | --- | --- |
| **Winners** | | |
| Field Name | Type | Comment |
| AId | Integer | >0, FK |
| Name | TEXT | NOT NULL |
| Country | TEXT | NOT NULL |
| Sid | Integer | >0, FK |
| Place | Integer | {1,2,3} |

|  |  |  |
| --- | --- | --- |
| **ActiveParticipants** | | |
| Field Name | Type | Comment |
| Aid | Integer | >0, FK |
| Sid | Integer | >0, FK |

Explanation:

* Observers view holds all the athletes that observe (inactive) for a sport they participate in. built using the Participants and Athletes tables and selecting only the inactive athletes. Using both athlete ID and attended sport ID as a PK. Useful for checking if a given athlete is an observer to a given sport event.
* ActiveParticipants view holds all the athletes that are active in a sport they participate in. built using the Participants and Athletes tables and selecting only the active athletes. Using both athlete ID and attended sport ID as a PK. Useful for checking if a given athlete is an active athlete for a given sport event.
* Winners view holds information about the athlete that won a medal in event. Built using the Athletes and Participants Tables. Useful for combining information of the athlete’s attributes to medals and wins. (e.g. country)
* MedalScore view holds information for each of the active athletes in the DB and how many gold/silver/bronze medals they have and a rating accordingly. Allowing quick comparisons and lookups for medal info per athlete.

**Summary**

The DB design that we chose represent the entities with minimal relationships between them as needed for the further API. Deletion of the dependant will cascade delete its related relashipships and will maintain a steady DB without dangling keys. The views were chosen according to repeated queries and such who will provide fast and easy access to specific data across multiple tables.

**API Explanations:**

* **ReturnValue addAthlete(Athlete athlete)**

Inserting an athlete object as tuple to the Athletes table using the get methods to extract relevant data from the object.

Return:

\*OK in case of success

\*BAD\_PARAMS in case of null in one of name,country,active parameters or a non positive ID. \*ALREADY\_EXISTS if an athlete with the same ID already exists in Athletes table

\* ERROR in case of a database error

* **Athlete getAthleteProfile(Integer athleteID)**

Getting an athlete object from the the Athletes table using the set methods to make an athlete object according to the returned tuple.

Return:

\*Athlete object in case the athlete exists in the Athletes table (according to his id)

\*BadAthlete object if it doesn’t

* **ReturnValue deleteAthlete(Athlete athlete)**

deleting an athlete from the Athletes table

Return:

\*OK in case of such athlete exists in the table and was deleted properly.

\*NOT\_EXISTS if an athlete with the such ID doesn’t exist in the Athletes table

\* ERROR in case of a database error

* **ReturnValue addSport(Sport sport)**

Inserting a sport object as tuple to the Sports table using the get methods to extract relevant data from the object.

Return:

\*OK in case of success

\*BAD\_PARAMS SQL check violation in one of name,city parameters or a non positive ID. \*ALREADY\_EXISTS if Sport with the same ID already exists in Sports table

\* ERROR in case of a database error

* **Sport getSport (Integer SportID)**

Getting a Sport object from the Sports table using the set methods to make an sport object according to the returned tuple.

Return:

\*Sport object in case the sport exists in the Sports table (according to it’s id)

\*BadSport object if it doesn’t

* **ReturnValue deleteSport (Sport sport)**

deleting a sport from the Sports table

Return:

\*OK in case of such sprort exists in the table and was deleted properly.

\*NOT\_EXISTS if sport with the such ID doesn’t exist in the Sports table

\* ERROR in case of a database error

**BASIC API:**

* **ReturnValue athleteJoinSport(Integer sportID, Integer athleteID)**

Checking if both athlete and sport exists according to their IDs, checking for the active status of the athlete and adding a new tuple for in the Participants table representing that the given athlete now participating/observing the sport event. In case of observer payment will be 100$ (0 otherwise). Additionally updates the sport in the Sports table by increasing the sport’s active counter if it’s an active athlete that has joined.

Return:

\* OK in case of success adding for existing athlete and sport

\* NOT\_EXISTS if sport/athlete does not exist in the Sports/Atheltes tables

\* ALREADY\_EXISTS if there is already Participant tuple for it

\* ERROR in case of a database error

* **ReturnValue athleteLeftSport(Integer sportID, Integer athleteID)**

Deleting the tuple for in the Participants table representing that the given athlete that attends to the given sport event. Additionally updates the sport in the Sports table by deacresing the sport’s active counter if it’s an active athlete that has left it.

Return:

\* OK in case of success deleting for existing attending athlete and sport

\* NOT\_EXISTS if sport/athlete does not exist in the Sports/Atheltes tables or there is no such tuple in the Participants table

\* ERROR in case of a database error

* **ReturnValue confirmStandings(Integer sportID, Integer athleteID, Integer place)**

Checking if the given athlete exists in the ActiveParticipants View and updating the Participants table entry for that given sport with the place field accordingly.

Return:

\* OK in case of success

\* NOT\_EXISTS if this athletes or sports don’t exists there will not be a participact tuple for them or if he is not active he won’t be in the ActiveParticipants View.

\* BAD\_PARAMS if the SQL QUERY returned an CHECK violation

\* ERROR in case of a database error.

* **ReturnValue athleteDisqualified(Integer sportID, Integer athleteID)**

Checking if the given athlete exists in the Participants table and updating the entry for that given sport with the place field to null.

Return:

\* OK in case of success

\* NOT\_EXISTS if this athletes or sports don’t exists or the athlete isn’t attending to the sport there will not be a participact tuple in Participants table for them.

\* ERROR in case of a database error.

* **ReturnValue makeFriends(Integer athleteID1, Integer athleteID2)**

Checking first that both athletes exists by searching the Athletes table and then checking that there is no such friendship already in the Friends table by checking for both directions (symmetric friendship as we discussed above) , adding a new entry to the Friends table

Return:

\* OK in case of success

\* BAD\_PARAMS if both ids are the same athlete

\* NOT\_EXISTS if one of the athletes wasn’t found the in Athletes table

\* ALREADY\_EXISTS if the friendship was found in the Friends Table

\* ERROR in case of database error.

* **ReturnValue removeFriendship (Integer athleteID1, Integer athleteID2)**

Checking first that both athletes exists by searching the Athletes table and then checking that there is such friendship already in the Friends table by checking for both directions (symmetric friendship as we discussed above) and deleting the entry from the Friends table.

Return:

\* OK in case of success

\* NOT\_EXISTS if one of the athletes wasn’t found the in Athletes table or friendship wasn’t found in the Friends table (both directions)

\* ERROR in case of database error.

* **ReturnValue changePayment(Integer athleteID, Integer sportID, Integer payment)**

Checking first that the athlete is an observer by searching the Observers view and then updating the Participants table for the given athlete and sport id payment field accordingly.

Return:

\* OK in case of success

\* NOT\_EXISTS if the athlete wasn’t found in the Observers view he doesn’t observe the given sport or if there is no such entry for him in the Participants table he or the sport don’t exsit.

\* BAD\_PARAMS SQL check violation when payment is a negative number

\* ERROR in case of database error.

* **Boolean isAthletePopular (Integer athleteID)**

Creating a set of all of his friends attended events using the Friends and Participants tables and checking if they are contained in all of athletes attended sport events using the Participants table – if they are then he is popular, otherwise he isn’t

Return:

Boolean result according to the answer. If the athlete doesn’t exist there will not be an entry for him in the Participants Table and false will be returned for such case.

* **Integer getTotalNumberOfMedalsFromCountry(String country)**

Using the Winners view and Group by the given country name we count the number of rows of the result and this is the number of medals that that country has won. We return it.

Return:

In case there is no athletes from this country there will be no rows in the result and 0 will be returned.

* **Integer getIncomeFromSport(Integer sportID)**

Using the Participants table and the fact that active athletes pay 0$ we sum the payment attribute for the given sport and return it.

Return:

In case there is no such sport there will be no rows in the result and 0 will be returned. (or no money payed for)

* **String getBestCountry()**

Using the Winners View and grouping by the country name and counting the number of medals won per country ordering first by number of medals won and secondary sort by country’s name.

Return:

The name of the country with must medals. In case there is no countries in the DB or no medals won at all returning “”(empty string) – null in any other case.

* **String getMostPopularCity ()**

Using the Sports Table and the active athletes counter attribute it has, we sort the sports by the average of the counter after grouping them by city name. Second sort will be by city name is descending order.

Return:

The name of the city with the highest average of active athletes in it. In case there is no cities in the DB returning “”(empty string) – null in any other case.

* **ArrayList getAthleteMedals(Integer athleteId)**

Using the MedalsScore view we first find the row by the given athleteId and then inserting each of the golds,silvers,bronzes attributes to the returned array accordingly.

Return:

A 3 index integer array with the number of gold,silver,bronze medals the athlete won. If the athlete doesn’t exist or if athlete is an observer we return an array with three zeros in it.

* **ArrayList getMostRatedAthletes()**

Using the MedalsScore view we first compute the rating for each of the athletes (active and inactive) and then we sort by score (higher first) and secondary sort by id(lowest first).

Return:

An index array with id corresponding to the ids of the atheltes with the highest score first according to the given rating system(in case of equality by their id as said above). The length of the array is no longer than 10. If there are less than 10 athletes we return a shorter array accordingly. In any other case we return an empty array.

* **ArrayList getCloseAthletes(Integer athleteID)**

We get the number of sports that the athlete participating from 'Participants' Table.

After wards we find the sports the athleteID is participating from 'Participants' Table and do inner join with all the athletes , sports\_ids from 'Participants'(with out the athleteID), on sport id. This gives us all athletes with the sports that are the same as athleteID from 'Participants' table. Afterwards we count how many rows each athlete got from the last result, this gives us the number of same sports as athleteID. From this result we pick only the athletes with at least 50% as athleteID. In the end we order by aid and pick the first 10.

In case athleteID participates in 0 sports, we just pick the first 10 athletes orderd by id.

Return:

An index array with id corresponding to the ids of the atheltes that are participating at least 50% of the athleteID sports. The list is orderd by ids The length of the array is no longer than 10. If there are less than 10 athletes we return a shorter array accordingly. In any other case we return an empty array.

* **ArrayList getSportsRecommendation (Integer athleteID)**

We use the query in getCloseAthletes as sub query to find the close athletes. Afterwards we do inner join with all the athletes and sports id's from 'Participants' table where the sports id's are not part of athleteID sports(the inner join is on athletes ids). This gives us all the sports the close athletes take that athleteID doesn’t take. We pick from the last result only the sports id, and we count how many rows each sport is mentioned. This gives us the number of popularity of each sport(that athleteID doesn’t take) that the close athletes take. We order by the popularity desc and afterwards by id asc, and pick the first 3

Return:

An index array with id corresponding to the ids of the sports that are the most popular among the close athletes of athleteID, which athleteID doesn’t take. The list is orderd by popularity desc and then by sport id asc . The length of the array is no longer than 3. If there are less than 3 sports we return a shorter array accordingly. In any other case we return an empty array.