CITS2232 Databases Project 2013

Django Unchained

Database Design Document

This document describes the rationale behind the design of the database for a sports and recreation club listing web site.

Most of the schema is as detailed in the Project Specification, and many of the fields are self describing. There are a number of attributes in certain relations however, that need to be outlined in more depth. We also added four more tables to get the most out of the application.

Users in our application are the only entities that interact with the database. Users can add Members and Clubs. A User with administering privileges (obtained either by being in the "End Admin" User Group, or by being the creator/owner of the Member or Club) can edit and delete the entities they manage.

This is modelled on the scenario of one User of the website managing multiple Members; such as a parent managing the applications for their children, or a school teacher managing their student's memberships in a variety of extra-curricular clubs.

To manage and maintain the entire website's database, we have the "End Admin" User Group. An admin can add, edit and delete any Member or Club. An admin can promote an end user to admin status, however once a user has become an admin, only themselves can revoke that privilege.

We have allowed it so that an anonymous user (someone that has not registered to the site) can view certain areas of the database – the list of Clubs and their public attributes, for example – although to be able to interact with the website, the user must register.

Many of the tables handled contact information. All fields, except for latitude and longitude, were stored as fixed length strings in the database. This was because contact information can be highly variable. For example, phone numbers can contain non-numeric characters, such as +(61), or it may start with 0, such as for mobile numbers. This cannot be represented with an integer data type.

Within the Club relation, we took "Recruiting Members" to mean an answer to the yes or no question: "is this Club recruiting new members?" As a result we defined recruiting members in the club relation in terms of a Boolean field. A Member can only apply to a Club if its "recruiting members" field is set to "true".

We interpreted "Contact" to be a member of the Club that is listed as the club's public representative. From this, the Club table can if need be, access all elements of the "contact" Member – name, email, phone etc.

The "Owner" field within our Club and Member relations was not stipulated in the Project Specification, although they are key component of our web application.

A Member's owner is the User that created the Member. A Club's owner is the administering Member of the Club. Only the User who owns the Member that owns the Club can edit the said Club.

Although Users can be deleted at the database level, from the application level a User cannot be deleted – only marked as active or inactive by an administrator. Thus, the Members of a User will exist until they are deleted either by an administrator or the User themselves. When Members are deleted, their Memberships are automatically revoked, as well as any relation to being the Owner or Contact of a Club.

Since a Club's owner has important roles and responsibilities within the application, we have made our software do it's best to avoid ownerless Clubs. When adding or editing a club, the website does not allow the user to leave the Club's owner field empty. However, it is possible to delete a Member who happens to be the

owner of a Club. If this occurs, the User is warned that they are about to delete the owner of a Club, where the owner field of that Club is set to "NULL" on confirmation of deletion.

The website's administrators are then notified on the front page that there exists a Club without an owner, and they can assign any Member of that Club to become the owner, or to simply delete that Club.

The four extra tables created were Membership Application, Club Type, Club Tag and Club Tags.

A row in the Membership Application table is created when a Member applies to a club. It is removed either when the Member who originally applied deletes the application, or when the Club's owner accepts or rejects the application. If the application is accepted, a new row in the Membership table will be created with the said Member and Club as Foreign Keys.

The Club Type table is static and defined within the initial data scripts. We require all Clubs to have exactly one Club Type, enumerated in the relation. We have chosen to not allow Users to add to this list of Types, so it does not become too long and chaotic. We aimed to include enough Club Types so the list covers a large spectrum of clubs that exist in the real world, without the list becoming too long.

Our web application's back end uses the Django framework which includes a "super user" capability. This super user has to ability to add to the Club Types relation, if absolutely necessary.

The Club Tag and Club Tags relations are used to collate each Club's tags in a central location, to make the "search by tags function possible". Each Club can have any number of tags, and each tag can be related to any number of Clubs. This "many-to-many" relation requires the Club Tags table to manage the multiplicities, so that our schema remains in 4NF.

For each tag that a Club's admin creates, a new row in the Club Tags table is created. This row references the ID of the Club and the ID of the Tag if it already exists. If the Tag does not exist, the database first creates a new row in the Club Tag table.

To reduce redundancy in the database, it is important that our relations are in Boyce-Codd Normal Form (BCNF) and Fourth Normal Form (4NF). We believe that since our schema was well designed from the beginning, there was no formal "normalisation process", just a quick check to ensure that the schema was in the appropriate normal forms.

As explained in the lecture notes, "A relation with Functional Dependencies (FDs) is in BCNF if: for each A → B, A is a key." In each of our relations, we have the FD: [ID *pk*]→ [rest]. That is, all of the elements in each of our tables functionally depend on the unique integer identifier at the start of each relation. Since this ID is a key, our schema satisfies BCNF.

Other examples of FDs that exists in our relation are [username] → [rest], or [club name] → [rest]. In these cases, [username] and [club name] can also be considered keys. Hence Boyce-Codd Normal Form is maintained.

The case is similar for Fourth Normal Form: "a relation with Multivalued Dependencies (MVDs) is in 4NF if: for each A -->> B, A is a key."

In our schema, it is designed such that there are no MVDs. Thus, our database is also in 4NF.

We can add any row to any of our tables, and that does not imply or require the existence of any other row. Since this is true, we can conclude that our schema satisfies 4NF.