COMP9311 22T3 Database Systems

Assignment 1 Stage 2 (Standard ER Design)

Last updated: **Friday 7th October 3:09am**
Most recent changes are shown in red ... older changes are shown in brown.

Introduction

This document contains the standard ER design for Stage 2 of Assignment 1. You must convert this design into a PostgreSQL relational schema (a collection of `create table` statements) and submit it via the Assignments link on the course web site. In performing the conversion from the ER design to a relational schema, you should follow the approach given in the lecture notes on “ER to Relational Mapping”.

Due Date

This assignment is due Week 5, **Friday, 14 October 23:59:59** (i.e. before midnight).

The UNSW standard late penalty for assessment is 5% per day for 5 days - this is implemented hourly for this assignment.

Each hour your assignment is submitted late reduces its mark by 0.2%.

For example, if an assignment worth 60% was submitted 10 hours late, it would be awarded 58.8%.

Beware - submissions more than 5 days late will receive zero marks. This again is the UNSW standard assessment policy.

Submission

For Stage 2, you are expected to submit a single file named **ass1.sql**, containing the relational schema definitions.

You may submit your work via the "Make Submission" tab at the top of this assignment page on WebCMS, or via give's web interface.

Alternatively, when logged into CSE, and ensuring **ass1.sql** is in your current directory, you may submit using the following **give** command:

```
$ give cs9311 a1s2 ass1.sql
```

You can check your latest submission via the "Check Submission" tab at the top of this assignment page on WebCMS, or on CSE servers with:

```
$ 9311 classrun check a1s2
```

You can submit multiple times, only your last submission will be marked.

Requirements

The schema you submit will be primarily marked by a program (auto-marked). In order for the program to recognise what you've done as being correct, your SQL **must adhere to the following requirements**:

- all entities and attributes must be given the same names as in the design (although case does not matter since SQL is case-insensitive)
- single-attribute foreign keys should be named after the **table** that they are referring to OR named after the **relationship** that they implement
- all tables must have an appropriate primary key defined; all foreign keys must be identified
- wherever possible, not-null, unique and domain constraints must be used to enforce constraints implied by the ER design

- wherever possible, participation constraints should be implemented using the appropriate SQL constructs
- all relationhips should be mapped using the approaches described in the lecture notes; in particular, you should avoid "over-generalising" your SQL schema relative to the ER design (e.g. a 1:n relationship should *not* be mapped in such a way that it can actually be used to form an n:m relationship)
- when mapping an n:m relationship R between two tables S and T call the resulting table SRT
- when mapping the People/Users class hierarchy, use the **ER-style** mapping (i.e. one table for each entity class)
- when mapping the Events class hierarchy use the **single-table style** (i.e. one table for the whole hierarchy)
- when mapping composite attributes, name the new attributes by concatenating the basename and the component name (e.g. addressStreet, addressSuburb, addressCity)
- when mapping multi-valued attributes, name the new table by concatenating the entity and attribute names (e.g. eventsAlarms)

Place the schema in a file called **ass1.sql** To give you a head-start, a template for the schema is available.

The reason for insisting on strict conformance to the above is that your submission will be auto-marked as follows:

- we will create an initially empty database (no tables, etc.)
- we will load your schema into this database
- we will use PLpgSQL functions to extract the schema in a fixed format
- we will try to reconcile this with the expected schema

Following the instructions above is considered to be a requirement of this assignment. If you stray from the expected schema, your submission will be marked as incorrect. Our auto-checking scripts have a little flexibility, but not much, so don't rely on it.

Please don't try to second-guess or "improve" the standard design below. Even if you think it's complete rubbish, just translate it as given. If you want to give opinions on the standard schema use the Ed Forum "Assignment 1". That is, we can discuss the improvement, but we will still use the given schema as the basis for the Stage 2 exercise.

Design

This ER design gives one possible data model for the CSEcal on-line calendar application introduced in the first stage of this assignment. The design here is based on the spec and on my experience with using vairous Calendar systems. This isn't necessarily the design that would be used in practice; it has been modified slightly to make Stage 2 of the assignment more interesting (i.e. to give you experience with a range of modelling constructs and translation mechanisms).

To make the presentation clearer, the design is broken into a number of sections. Note that an entity will have its attributes and class hierarchy defined exactly once. If an entity is used in a later section of the design (e.g. to show relationships), it will simply be shown as an unadorned entity box (and you should assume all of the attributes and sub/super-classes from its original definition).

The development of any significant design requires assumptions. Assumptions specific to particular entities and relationships are presented below.

Data Types

Several specialised kinds of data exist in the system:

- **Dates and Times**
  We assume that the calendar will only be used by people in and around the UNSW Kensington campus, and so they'll all be operating in the same time-zone. If we wanted to allow multiple campuses in different time-zones, we could add time-zone as an attribute of each calendar. Note that `time` is a built-in SQL data type and would be suitable to use here.

  Dates are also available as a standard data type `date` in SQL. PostgreSQL also provides a `timestamp` data type which combines a date and time (down to microsecond accuracy). This if

often useful, but is not relevant for this particular application.

- **Alarms**
  An alarm is defined simply by a time-interval before an event when the user will be alerted to the event's occurrence. There are two different ways of alerting a user: via email, or by popping up an alert on their screen if they're logged in to the calendar system (we haven't gotten into things like SMS to your mobile phone yet).

  In determining what kind of alarm to use, where *diff* = *eventTime* - *currentTime*:

  - if the person is not a user and *diff* ≥ *1 hour*, send them email
  - if the person is not a user and *diff* < *1 hour*, do nothing
  - if the person is a user and *diff* > *30 mins*, send them email
  - if the person is a logged-in user and *diff* ≤ *30 mins*, use a pop-up alert

  Note that `interval` is a built-in PostgreSQL data type for describing intervals of time and would be suitable to use here.

- **Access levels**
  Different users have different access rights to calendars/events. Access rights are always assigned at the Calendar level, and apply to all of the events contained in the calendar. The owner of a calendar specifies the access rights for users and groups, and also sets default access rights. Access rights are specified via an access level; possible access levels are:

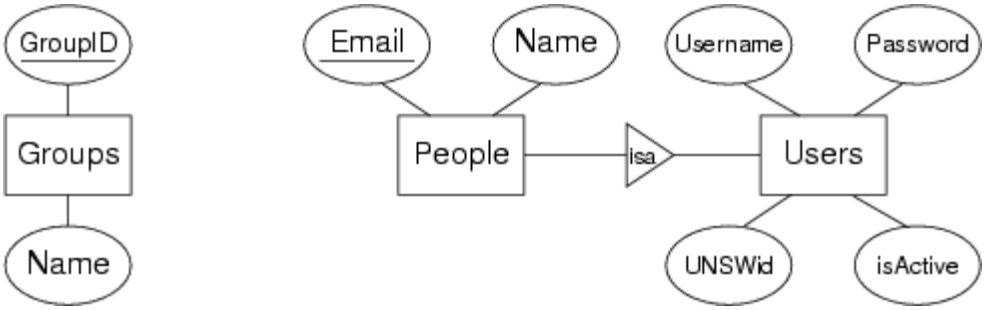
  - no access at all (don't even know that events exist)
  - time-block access (can see events as “busy” only, no details)
  - read-only access (can see details of event)
  - read-write access (can see and change event details)

  Each Calendar has a default access level which applies when no Group or User access rights exist. Access rights for calendars can be assigned to groups, in which case all members of the group inherit the group access level. Access rights for a calendars can also be assigned to individuals. The access level as seen by a user is determined by first checking their individual access level (if any), then checking the access levels from any groups they are a member of (if several groups have access rights to the calendar, then choose the highest access level), and finally, if they have no specified user or group access, the calendar's default access level is used. Access rights are not associated with people who are not users, because they do not have direct access to the calendars.

  When a Calendar is first created, the default default access level is “no-access”. If the owner makes the default access level to a Calendar anything other than “no-access”, then it becomes a public calendar. In the most lenient case, giving a Calendar default access level of “read-write” means that any user can add events to that calendar, unless they are blocked by having a lower access level applied specifically to them, or to all Groups that they are a member of.

  The template defines a suitable `AccessLevel` type using the `create domain` statement. See the PostgreSQL manual for details.

People



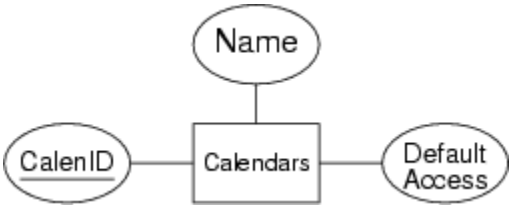
Comments:

- there are two* different kinds of people who access CSEcal: people within UNSW and people not associated with UNSW (note that CSE plans to be generous and allow the whole of UNSW to use

its calendar system)
- people within UNSW can create and manage calendars and events; they are also called “users” and all have a unique UNSW id
- people from outside UNSW may be invited to events inside UNSW, and so might need to be referred to from within the system (e.g. we might want to send them an invitation or reminder)
- people from outside UNSW clearly don't have a UNSW id
- we assume that everyone has an email address (so that e.g. we can send them invitations), and that email addresses are unique, so that email can be a primary key for all people in the system
- since it is more convenient for users to login using a mnemonic username (rather than an email address or UNSW id), we include a username
- for authentication, a real implementation of this system would most likely use an external system such as UniPASS; however, we include a password field so that authentication can be handled entirely within the calendar system
- in order to give the system administrator some control over users, each user has an associated “active” flag which indicates whether they are allowed to log in or not
- groups need a name, which is displayed to users of the system
- however, group names are unlikely to be unique across all users (e.g. many people might have a “My friends” group)
- thus, we introduce a numeric attribute to form a key

* In fact, there is a second type of “user” for CSEcal: the CSEcal administrator. The administrator will have additional privileges, such as adding users, creating groups, and so on. As far as the data model is concerned, however, the administrator is just a “user” and their additional privileges will be implemented via switches in the code for the system.

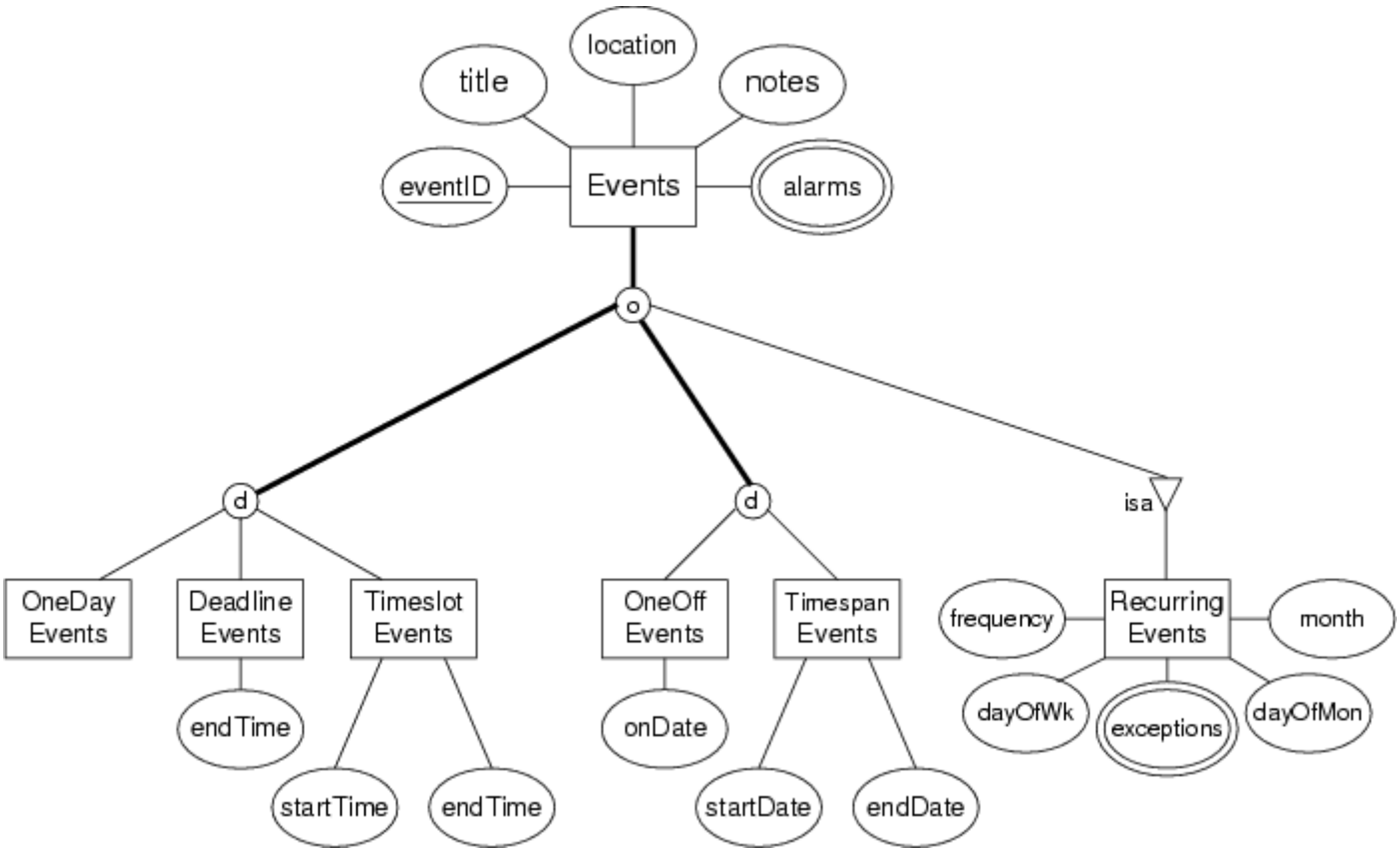
Calendars



Comments:

- calendars need a name, which is displayed to people using the calendar system
- however, calendar names are unlikely to be unique across all users (e.g. many people might have a “My appointments” calendar)
- thus, we introduce a numeric attribute to form a key
- since calendars form the basis for access rights to events, each calendar also needs to specify a default access level (users can make their calendars “public” via this mechanism)

Events



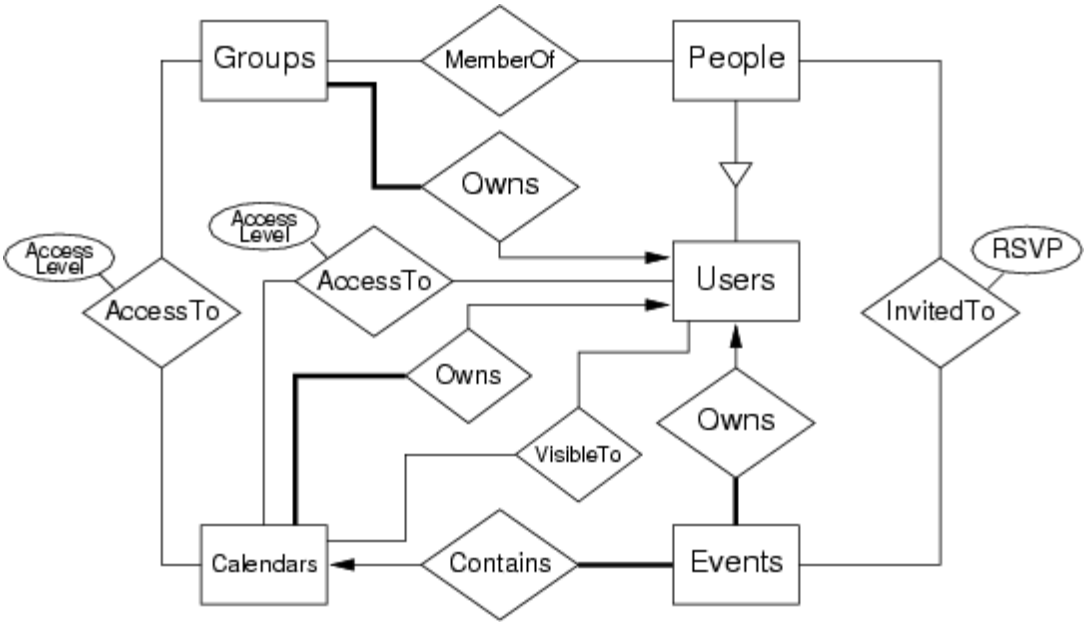
Comments:

- all events need a title (for display on the calendar)
- we also provide the option for any event to have a location and some notes
- since no combination of (title,location,notes) gives a useful primary key, we introduce a numeric attribute for this purpose
- there are many different “styles” of events, e.g. anniversaries (same date each year), meetings (specific time on specific date), period (range of dates e.g. Week 3 22T3), classes (same day each week for a specified period), school meeting (11am on the first Thursday of each month), etc. etc.
- one possible approach might be to think of every possible event style, and have a separate sub-class for each
- our approach is to factor out the time, date and recurrence aspects of the event specification into three separate sub-class hierarchies
- note that every event *must* belong to one of the OneDay, Deadline or Timeslot subclasses; every event must also belong to one of the OneOff or TimeSpan subclasses; events may or may not be recurrent
- by combining different pairs from the time/date class hierarchies, and, optionally, a recurrence event, a range of effects can be achieved (e.g. a OneDay event + a Timespan event over a range of years + a Recurring events with yearly frequency and a specific month and date in month, combine to implement the Anniversary type mentioned above)
- maybe some combinations don't make sense, but these can be handled by the interface preventing these entity combinations from being inserted into the database (ideally, of course, the database itself should prevent nonsensical data from being entered)
- for recurring events, we have effectively collapsed a further subclass hierarchy into a single class; we could potentially have used sub-classes for regular daily/weekly/monthly/annual events on particular dates in particular months or particular days of the week or things like “first Tuesday of each month”
- the value of the `dayOfWk` attribute is one of Mon, Tue, Wed, etc.
- the value of the `dayOfMon` attribute is one of 1..31
- the value of the `month` attribute is one of Jan, Feb, Mar, May, ...
- the `onDate`, `startDate` and `endDate` attributes are specific dates (e.g. 21 July 2022)
- the `startTime` `endTime` attributes are specific times of day (e.g. 3:15pm)
- there is also an assumption that recurrent events don't occur more than once per day; thus, exceptions to recurrent events can be specified as dates (the dates on which the event does *not* occur)
- since there is some overlap amongst the attributes of the subclasses, it seems simplest to use a single-table implementation of the hierarchy, despite the fact that a number of attributes will be

"wasted" for non-recurring events

Important Note: the last point is a requirement: you *must* incorporate all of the sub-class hierarchies into a single `Events` table

Relationships



Comments:

- every group, calendar or event is created by a user and thus owned by that user
- access rights are attached to calendars and determine the access to all events on that calendar (see above for a discussion of access rights values)
- since a user can create as many calendars as they want, they can enforce appropriate access rights to any event they create simply buy putting the event in a calendar with the required access level (if none of their existing calendars have a suitable access level, they can always create a new calendar)
- invitations are associated with People rather than Users to allow external people to be invited to UNSW events; we assume that the system provides a mail-based interface for them to respond to an invitation
- the RSVP attribute records the response simply as "Yes" (I will be attending) or "No" (I will not be attending), with `NULL` indicating that the invitee has not yet responded
- logged-in users can specify which calendars they want to appear in their browser at any given time; since these visibility preferences persist over calendar sessions, they need to be stored, and so they are represented in the diagram via the VisibleTo relationship between Users and Calendars
- note that users can only specify visibility on calendars that they have access rights to, but this is not specified in the diagram
- if a calendar is VisibleTo some user, then the events in that calendar are displayed in their CSEcal window;
- if a calendar is not VisibleTo a user, then none of its events are displayed
- the user can toggle visibility via checkboxes in the interface

If any aspect of this specification requires further clarification, ask for it under topic "Assignment 1" on the course forum.

Don't forget to have fun!