COMP3311 20T3 Database Systems

Assignment 1
Mapping ER Designs to SQL

Last updated: **Wednesday 30th September 7:58am**
Most recent changes are shown in red ... older changes are shown in brown.

Aims

The aims of this assignment are to:

- read and understand an ER data model
- translate the ER model to an SQL schema

Admin

Submission:	use the command `give cs3311 ass1 schema.sql` or use Webcms3
Deadline:	**Sunday 11th October 23:59**
Marks:	contributes 12% of the final mark
Late Penalty:	Late submissions will have marks deducted from the maximum achievable mark at the rate of 0.08 marks *per hour* that they are late (i.e., around 2 marks per day).

Description

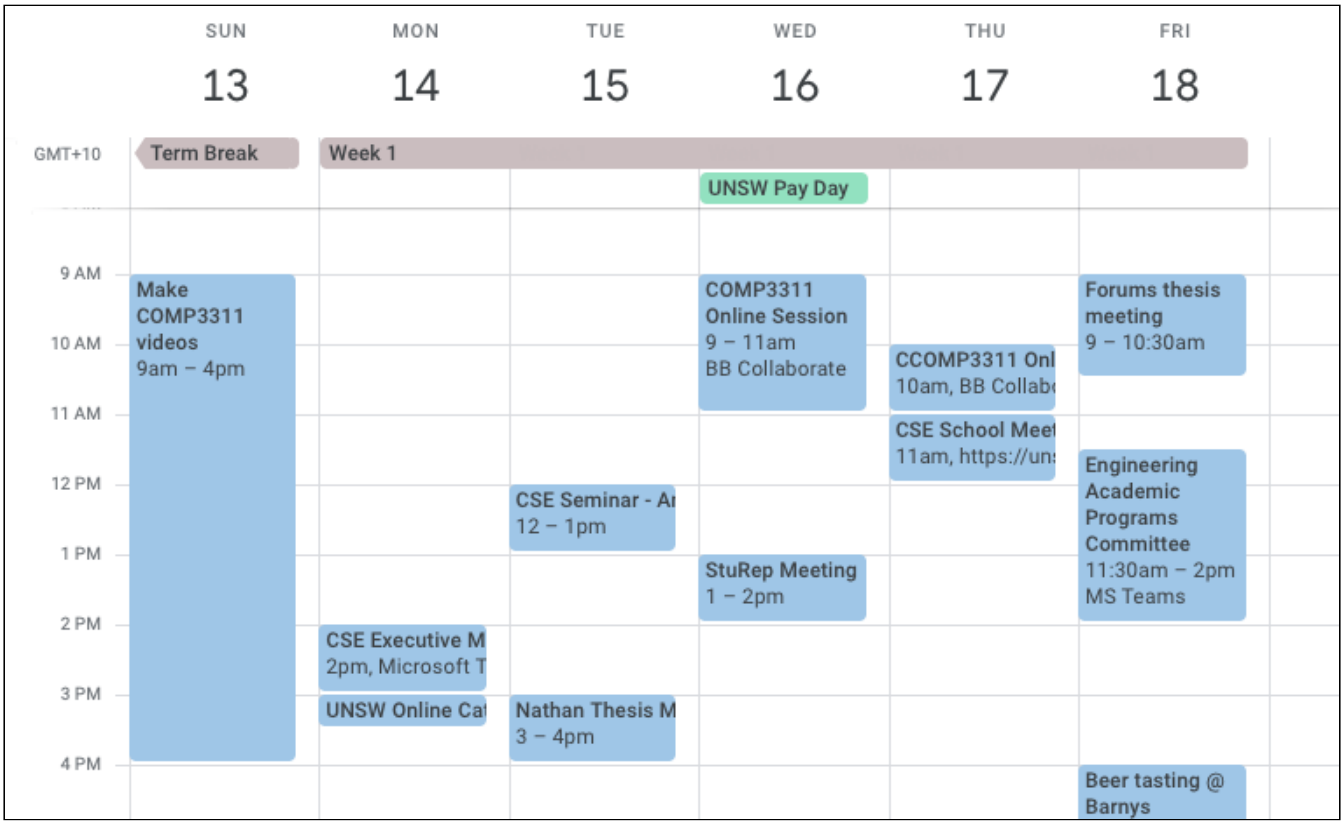
Assessment

The aim of this assignment is to take a given ER design and convert it to a database schema expressed in the SQL data definition language. In some sense, this comes in part-way through the database development process; you would nomally start with requirements and develop an ER model and then convert it to SQL. We have done the first bit for you.

This assignment is worth a total of **12 marks**. You will be assessed automatically (with scope for manual override) according to how completely and accurately your SQL schema translates the ER design. Part of "accuracy" involves following the naming conventions we give below (under **Your Task**). Please try to restrain your boundless creativity and use the naming scheme that we suggest. There are other rules for the translation in **Your Task** apart from naming; please follow these as well.

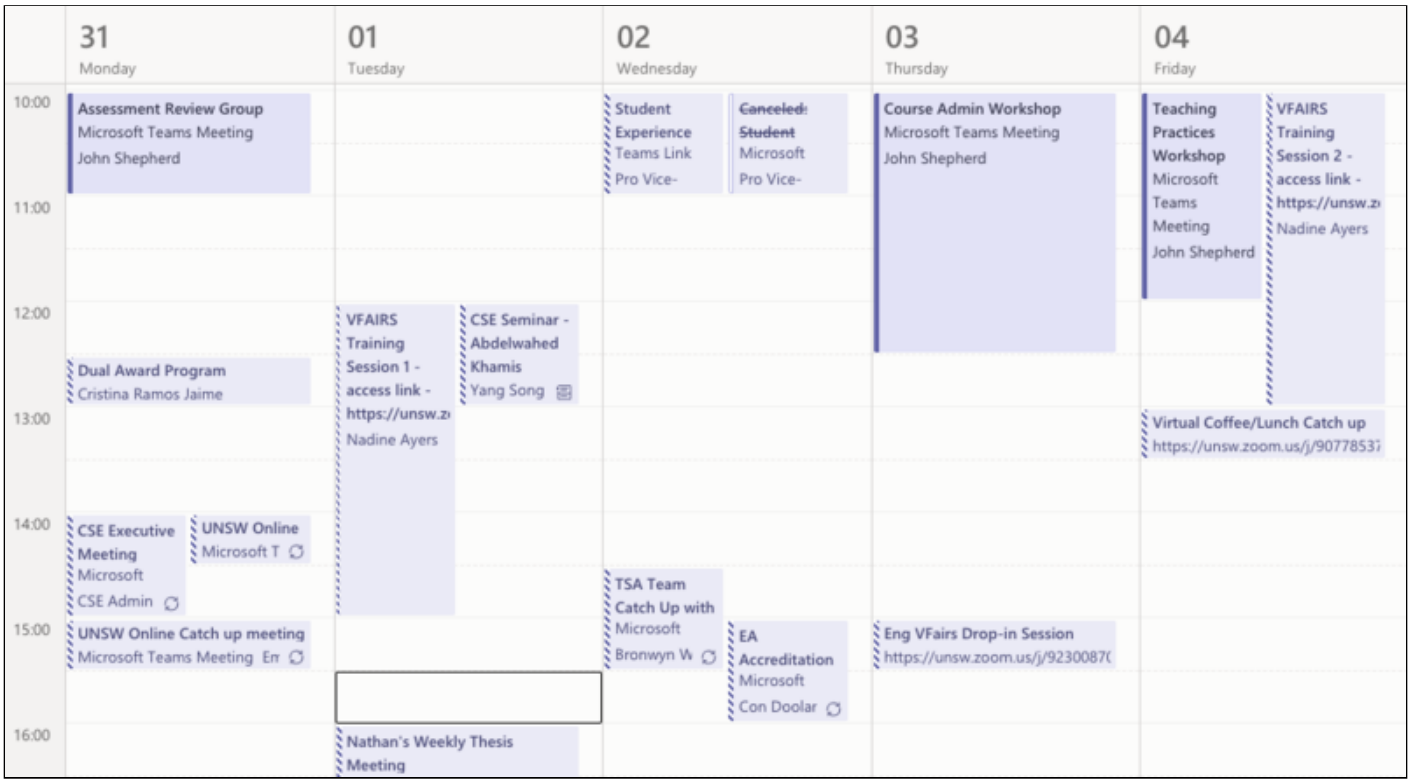
The Problem Domain

Online calendars are now essential for managing activities in most organisations. UNSW uses Outlook to provide a global calendar system; other online calendars include Google Calendar and Apple's Calendar. For example, here is my Google calendar for one week:



You can see in the above example that the calendar contains various kinds of events; some are one-offs (e.g. CSE Seminar) while others are repeating (e.g. COMP3311 Online Sessions (despite the time for the Wednesday session being incorrect)) Events are also coloured, because they are drawn from different calendar sources and merged into this single view.

Most calendar apps provide a similar set of functionalities. For example, the following Outlook calendar shows a different mix of repeating and one-off events:



Since a real calendar app would have a significant amout of complexity, we have distilled the requirement for our app down to a minimum useful set.

Requirements

The first thing we need to do in database development is to analyse the requirements and build a data model based on them. The following set of requirements focusses on the entities in the system. Operational issues are mentioned, but

only to inform thinking on what data might need to be stored in the back-end of a calendar app:

Users

- individuals who use the calendar
- we need to know at least their name and email address
- they also have a non-empty password for authentication
- some (very few) users have administration privileges
  (we don't need to specify what these privileges are in the data model)

Groups

- named collections of individual users
- useful as shorthand for scheduling events for specific groups
- each group is owned by a user, who maintains the list of members

Calendars

- named collections of events (e.g. "John's Weekly Meetings/Classes")
- each event is attached to a specific calendar
- each calendar has accessibility restrictions (per user and default)
  (e.g. some users have read/write, some have read-only, some have no access)
- if a user has read permission on a calendar, they see private event titles instead of "Busy"
- each calendar is owned by a user; a user may own many calendars;
- users may subscribe to other peoples' calendars (if they can read them)
- each calendar has a colour (set by its owner); a subscriber may set a different colour for their own view

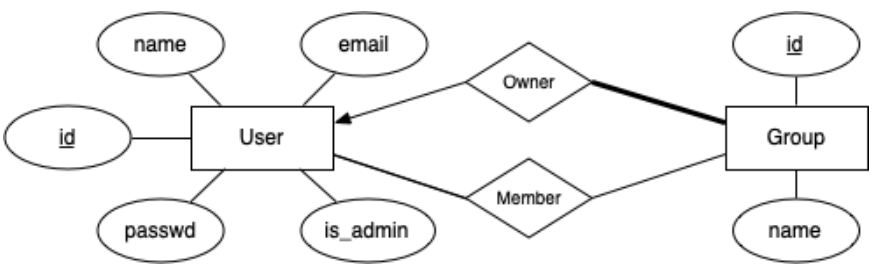
Events

- there are various kinds of events
  - associated with a particular day/date (e.g. birthday)
  - scheduled at a given time on a given day (e.g. a meeting)
  - recurring on a regular basis (e.g. a COMP3311 lecture)
- each event is owned by the individual user who creates it
- each event has a title and visibility (public, private)
  - a private event is shown simply as "Busy" in the interface
- an event may be associated with a location (where it will occur)
- an event may be associated with a set of individual users (invitees)
- an event may recur in a number of ways
  - on a particular day of the week (Mon,Tue,Wed,Thu,Fri,Sat,Sun)
  - weekly, every 2/3/4 weeks
  - monthly (on same date of month), every 2/3/.../11 months
  - on the first/second/third/last Xday of each month
  - for a fixed number of times (e.g. 10 times)
  - annually
- a recurring event will have a starting date and may have an ending date
- at specified times before each event an alarm event can be triggered
- there may be multiple alarms associated with an event
  (e.g. 15 mins before, 5 mins before, 1 minute before)
- an event can have an associated list of users who are invited
  (we don't associate events with groups; groups are used to generate a list of invited users whne the event is created)
  - users on the list can be flagged as "Invited", "Accepted", or "Declined"

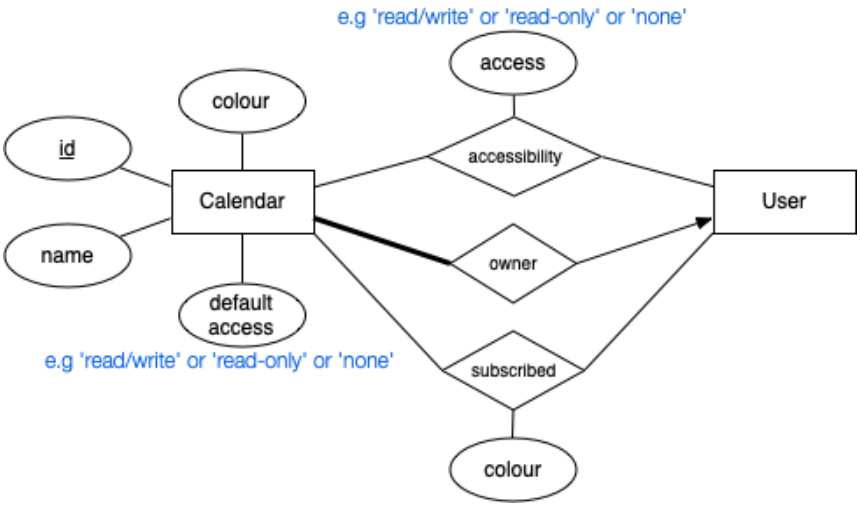
ER Design

Based on the above list of requirements, we have developed an entity-relationship (ER) data model for the calendar app.

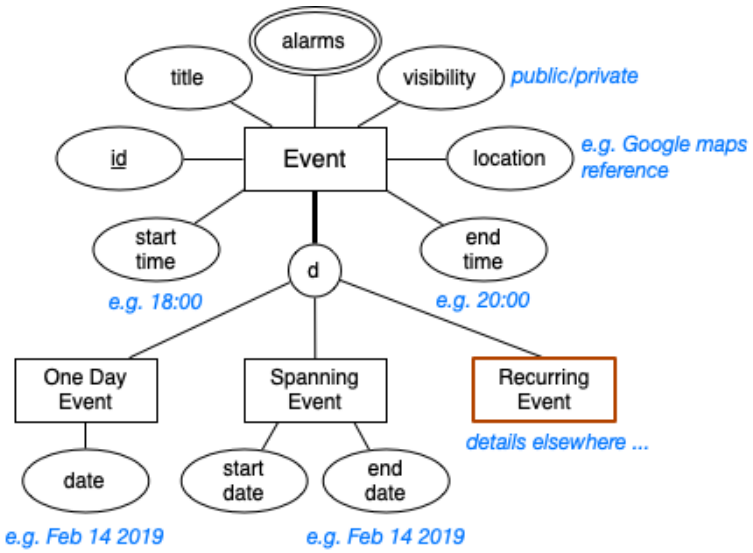
User and Group Entities



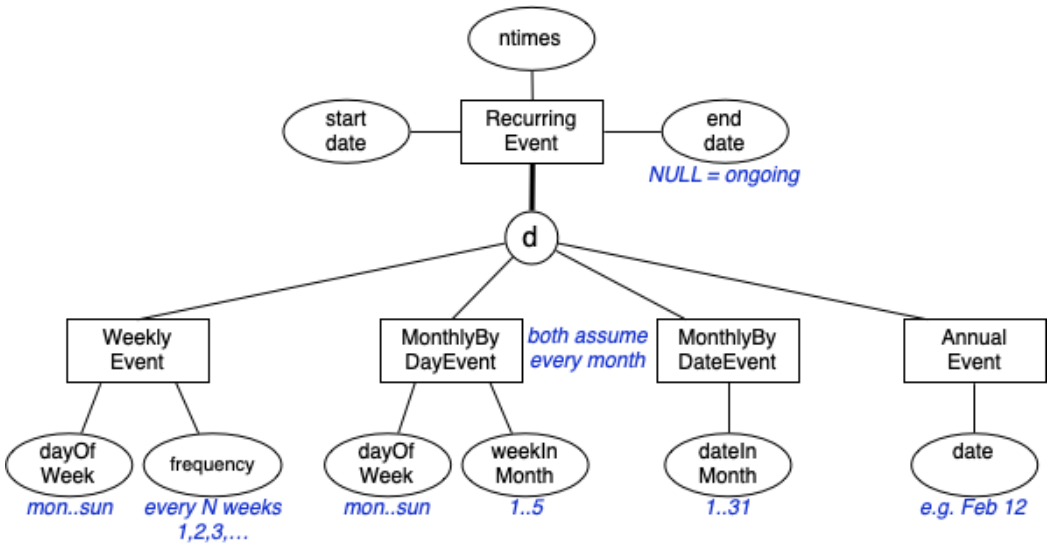
Calendar Entity



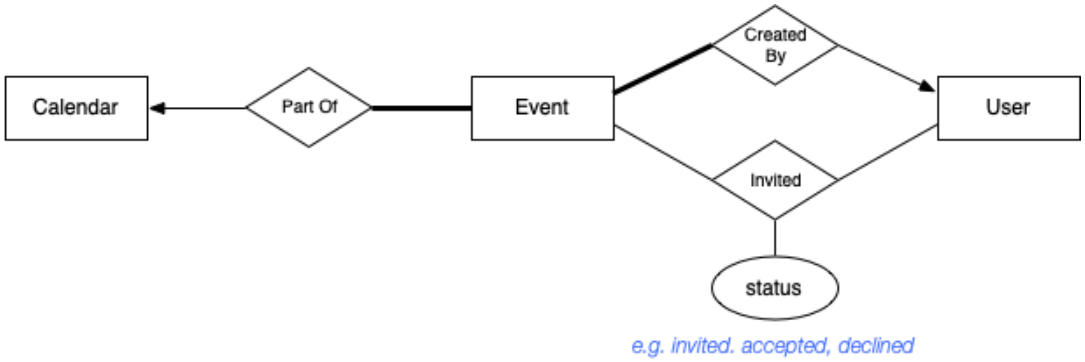
Event Entity Classes



Recurring Event Entity Classes



Relationships



Only includes relationships not covered in previous diagrams.

Your Task

Note that some of the potential primary keys (e.g. User.email) are useful for an abstract design like ER, but are not useful in a database. For each entity containing such a potential primary key, we have added an extra attribute called `id` to be the primary key. All such `id` attributes should be defined as `serial`.

After doing the above, you should convert the above design into an SQL schema, using the following rules:

- all tables based on entities are given plural names
- attributes in entities are given the same name in ER and SQL
- attributes in relationships are given the same name in ER and SQL
- multi-valued attributes typically have pluralised names,
  in the SQL schema these names should be written in singular form
- names containing multiple words, should use underscores to separate the words
- ER key attributes are defined using `primary key`
- give attribute constraints where useful
- do not add `NOT NULL` if the attribute should be able to be null
- add `UNIQUE` if it is clear that the attribute will never contain duplicate values
- text-based attributes are defined with type `text`,
  unless there is a size which is obvious from the context
- attribute domains can be PostgreSQL-specific types where useful
- identify and add declarations for all primary and foreign keys.
- foreign keys within entity tables are named after the relationship
- foreign keys in relationship tables are named *`table`*`_id`
- implement all class hierarchies by the ER mapping

Follow these rules as closely as you can, as well as following the standard ER→SQL mapping strategies from the slides.

If you think that some aspect of the ER design cannot be expressed in SQL, add a comment to you schema to explain.

One thing you should not do is try to second-guess the requireents or the ER design, come up with your own ER, and implement that. This would be like taking some specs from your boss and implementing something that you thought was better than the specs.

Feel free to discuss the supplied ER model in the forum, and propose a "better" solution. If you propose something that is genuinely better, or you clarify some ambuguity or unclearness in the spec, and you do this *by Monday 28 September*, we will consider changing the spec.

Doing the Work

Type your SQL schema into a single file called `schema.sql`. A template for this file is available, which already contains some obvious SQL declarations.

```
/home/cs3311/web/20T3/assignments/ass1/schema.sql
```

Unfortunately, we cannot give you sample tuples to check your schema. If we did, we'd effectively be giving you most the schema that we want you to build.

Submission and Testing

You should ensure that `schema.sql` is syntactically correct and will load in a single pass into a newly-created database. We will test your schema as follows:

```
... we will login to grieg an run a PostgreSQL server ...
$ dropdb cal
```

```
$ createdb cal
$ psql cal -f schema.sql
... we will run our auto-marking script to analyse the contents of the database ...
```

Make sure that you run the three middle commands yourself on Grieg before submitting your assignment.

Assessment

10/12 marks come from the automatic assessment of your `schema.sql`

2/12 marks come from manual assessment of your coding style, e.g. consistent naming strategy, cnsistent and readable layout.

The auto-marking script will examine properties of your loaded schema, such as: appropriate mapping of tables for entities and relationships, correct subclass mapping, defining apropriate primary and foreign keys, correct use of NOT NULL definitions, naming that follows the above requirments, etc.

If your schema has errors when loading into an empty database, you will receive a 3 mark penalty. If the errors aren't widespread, we will attempt to fix them and reload `schema.sql`. The penalty is deducted from whatever mark you receive after it *does* load.

Have fun, *jas*