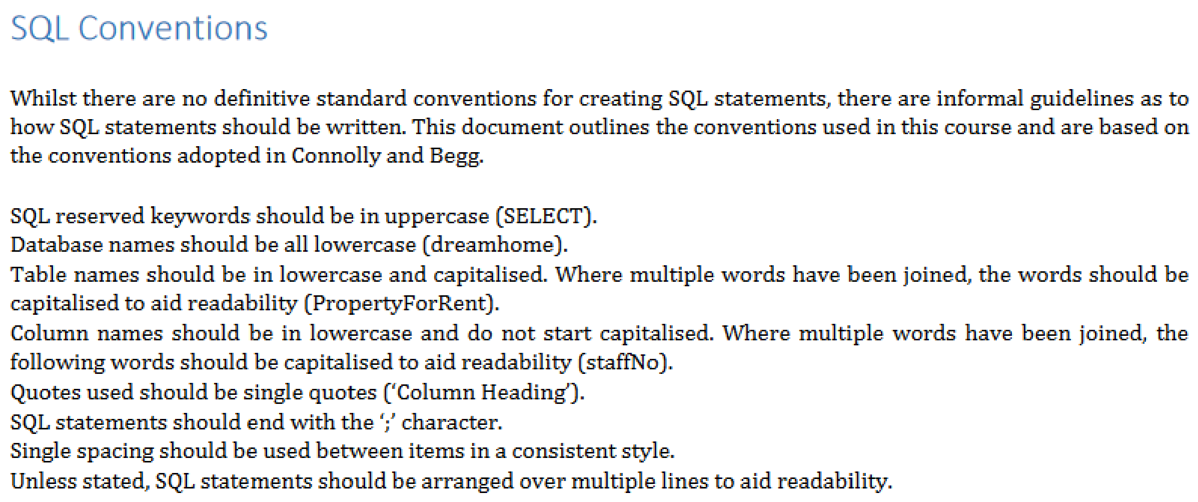
SQLCAA documentation

Note: Due to the structure of ruby tests we were forced to break some of the conventions. Namely the multiline convention and in some cases column capitalization convention i.e COUNT(cfound) instead of COUNT(cFound) due to the expected results being case sensitive.



SQL Conventions

Whilst there are no definitive standard conventions for creating SQL statements, there are informal guidelines as to how SQL statements should be written. This document outlines the conventions used in this course and are based on the conventions adopted in Connolly and Begg.

SQL reserved keywords should be in uppercase (SELECT).
Database names should be all lowercase (dreamhome).
Table names should be in lowercase and capitalised. Where multiple words have been joined, the words should be capitalised to aid readability (PropertyForRent).
Column names should be in lowercase and do not start capitalised. Where multiple words have been joined, the following words should be capitalised to aid readability (staffNo).
Quotes used should be single quotes ('Column Heading').
SQL statements should end with the ';' character.
Single spacing should be used between items in a consistent style.
Unless stated, SQL statements should be arranged over multiple lines to aid readability.

The following are our solutions to the questions, with a brief explanation. All of the statements were tested in the ruby and SQLite viewer (https://inloop.github.io/sqlite-viewer/), on the SQLite 3 engine.

0. SELECT * FROM Users;

   The above query returns all fields within the 'Users' table. It is the easiest solution to the question.

1. SELECT uid, username FROM Users;

   In this query we specify the fields we wish to retrieve. We agreed that this was a good solution.

2. SELECT uid AS 'UserID', username AS 'UN'
   FROM Users;

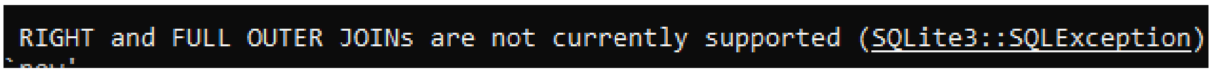
   Expanding on the query for no.1 we simply assign an alias to 'uid' and 'username' columns. There was a proposition use the commands ‘ALTER TABLE’ and ‘ALTER COLUMN’ - to rename the column, but we agreed that this would be needlessly complex.

3.
   ```
   SELECT uid, username FROM Users
   WHERE uid='U002';
   ```

   Here we take the query from no. 1 and use the filtering clause ‘WHERE’ to match our criteria ‘uid’ of value ‘U002’.

4.
   ```
   SELECT uid, username, Owners.oCacheId, Owners.oCreated
   FROM Users
   INNER JOIN Owners ON Users.uid = owners.ouid
   AND owners.ouid = 'U002';
   ```

   We chose ‘INNER JOIN’ as this join is the equivalent of an intersection between two tables, returning records found in both tables only. This makes it perfect for most Join operations. An additional reason is that SQLite 3 does not currently support these.



```
RIGHT and FULL OUTER JOINs are not currently supported (SQLite3::SQLException)
```

   We join the tables at uid value as that is the key they have in common, after which we filter using ‘WHERE’ like in no. 3

5.
   ```
   SELECT uid, username, oCacheId, oCreated
   FROM Users, Owners
   WHERE uid = ouid AND uid = 'U002';
   ```

   This is a more relational solution, and from our research is less preferred then ‘INNER JOIN’, as the ‘WHERE’ keyword also implies a join, but is less clear. However, this is required to achieve the criteria in the question.

6.
   ```
   SELECT U.uid, U.username, T.tid, T.tName
   FROM Users AS U, Trackables AS T
   WHERE U.uid = 'U001' AND T.tid = 'T001';
   ```

We based this solution on no. 5, simply adding an alias for both tables due to the fact that both have the 'uid' column and not specifying which table to draw from leaves ambiguity.

7. SELECT uid, username, cacheId, cName
   FROM Users, Caches
   WHERE cacheId BETWEEN 'C003' AND 'C005' AND uid = 'U003';

Basing on no.5 again, we only added the 'BETWEEN' operator to specify the range of values ('C003' to 'C005' inclusive), and changed the ID to be relevant to the user in this question.

8. SELECT U.uid, U.username, C.cacheId, C.cName
   FROM Users AS U, Owners AS O, Caches AS C
   WHERE C.cStatus = 'Active'
   AND O.ouid ='U003'
   AND O.ouid = U.uid
   AND O.oCacheID = C.cacheId;

   This query is a combination of no.6 and no.5 in that we once again rely on the 'WHERE' clause. Once again we use aliases for different tables. Lastly, we have broken the criteria into simpler predicates. We thought to use 'SELECT DISTINCT' statement, to filter out duplicates, but ultimately decided this was more readable.

9. SELECT gid, uid, username
   FROM Users
   ORDER BY gid DESC

   Applying occam's razor once more we came up with a simple query. To sort the data we used 'ORDER BY' on the gid column in descending order.

10. SELECT uid AS 'ID', username AS 'UN'
    FROM Users ORDER BY gid;

Again nothing new is added, we have already mentioned aliases and ‘ORDER BY’. Since we want the data sorted in ascending order we no longer need to specify ‘DESC’ as ‘ASC’ is default.

11. SELECT COUNT(ouid) FROM Owners WHERE Ouid = 'U003';

    With this query we use the aggregate function ‘COUNT()’ to well count the number of records for owner with ouid value of ‘U003’

12. SELECT COUNT(tid) AS 'Number of trackables'
    FROM Trackables WHERE uid = 'U001';

    Same as above for a different table with an ‘AS’ alias.

13. SELECT uid, COUNT(tid) AS 'Number of trackables'
    FROM trackables GROUP BY uid;

    In this query we use the ‘GROUP BY’ statement to divided the ‘Number of trackables’ individually for each user (uid).

14. SELECT uid, COUNT(tid) AS 'Number of trackables'
    FROM Trackables WHERE uid = 'U003' LIMIT 0;

We struggled with this query for a long time as it would return null and 0 due to there being no records which match the criteria. However, we found the LIMIT statement which allows us to reduce the number of rows to 0.

15. SELECT Owners.ouid, Users.fName, Users.lName,
    COUNT(cacheId) AS 'Number of caches owned'
    FROM Caches
    LEFT JOIN Owners ON Owners.oCacheId = Caches.cacheId
    LEFT JOIN Users ON Users.uid = Owners.Ouid
    GROUP BY Users.uid;"

For this query we decided on LEFT JOIN which “returns all records from the left table, and the matched records from the right table” (source: https://www.w3schools.com/sql/sql_join_left.asp)

We essentially attach the three tables together using the common fields of ‘uid’ and ‘cacheId’ and then divide the result into groups based on ‘uid’.

16.
```
SELECT Owners.oCacheId, Owners.ouid, Table1.fName,
Table1.lName, Table2.uid, Table2.fName, Table2.lName
FROM Owners
LEFT JOIN Users Table1 ON Owners.Ouid = Table1.uid
LEFT JOIN Users Table2 ON Owners.AUId = Table2.uid;
```

Due to the task requiring data from within the same table we had to Use aliases for two instances of the ‘Owners’ table. We then perform a self join to stitch the tables together. Giving us data about the name of the owner and the user who authorized the cache.

17.
```
SELECT cacheId, COUNT(CASE WHEN cFound = 'NotFound'
THEN 1 ELSE NULL END) AS 'Number of times not found'
FROM Clogs WHERE cFound = 'NotFound'
GROUP BY cacheId;
```

In this query we decided to simply use a ‘CASE’ statement within count. This allows us to specify to count only the records which have a value of ‘NotFound’.

18.
```
SELECT tid, COUNT(CASE WHEN tStatus <> 'Removed'
THEN 1 ELSE NULL END) AS 'Number of caches visited'
FROM Tlogs
GROUP BY tid;
```

Similar to above, however this time we count the number of trackers which do not possess the ‘Removed’ status

19. SELECT uid, cacheId, COUNT(cacheId) AS 'COUNT(cacheid)', MIN(date) FROM Clogs
    WHERE cFound <> 'NotFound'
    GROUP BY cacheId;

Building on previous examples, this time we are required to give the first-find date or the earliest time the cache was found. To achieve this we used the MIN() function which returns the smallest value. The alias in this query is simply for the test to pass and is superfluous to the task otherwise.

20. SELECT Clogs.uid, users.fName, users.lName, cacheId, COUNT(cacheId) AS 'COUNT(cacheid)', MIN(date)
    FROM Clogs JOIN users ON Clogs.uid=users.uid
    WHERE cFound <> 'NotFound'
    GROUP BY cacheId;

Same as above but joined with user name and surname data using ‘JOIN’ (inner join) on matching ‘uid’.

21. SELECT cId, tid FROM Tlogs WHERE tStatus = 'Visited' ;

Due to the structure of the table we assumed it would be enough to query records with the status ‘Visited’, as we believe this is mutually exclusive with ‘PutInCache’ ‘tStatus’, however if that is not the case we could always amend our statement to omit those records where both ‘PutInCache’ and ‘Visited’ are present.

22. SELECT Table1.tid FROM Tlogs Table1 INNER JOIN Tlogs Table2 ON Table2.tid = Table1.tid
    WHERE Table1.date = Table2.date
    AND Table1.tStatus = 'Removed'
    AND Table2.tStatus = 'Visited';

We created 2 instances of ‘Tlogs’ table using the aliases ‘Table1’ and ‘Table2’. Next we joined them together, to compare the date column and select only those records with the same date and tid.

Then it was only a matter of checking whether both ‘Removed’ and ‘Visited’ statuses were present.

23. SELECT cacheId, COUNT(cfound) FROM Clogs WHERE cFound <> 'NotFound' GROUP BY cacheId ORDER BY COUNT(cFound) DESC, cacheId;

In this query we remove caches which have not been found, we then ‘COUNT’ how many times each case has been found. Group these numbers into each unique case, and then sort them on two levels. First we sort them in order of descending number of times found, and within the same level we sort them based on ‘cacheId’.

24. SELECT cacheId, COUNT(cfound) FROM Clogs WHERE cFound <> 'NotFound' GROUP BY cacheId HAVING COUNT(cFound) >= 3 ORDER BY COUNT(cFound) DESC;

Same as no. 23. However with the added case that count must be 3 or greater. Since the WHERE keyword cannot be used with aggregate functions like ‘COUNT()’, we used ‘HAVING’ in it’s place, with the same function - to filter out records with lesser ‘COUNT’.