PostgreSQL Exercises

This is a compilation of all the questions and answers on Alisdair Owen's PostgreSQL Exercises. Don't forget that actually solving these problems will make you go further than just skimming through this guide, so make sure to pay PostgreSQL Exercises a visit.

Table of Contents

Getting Started

It's pretty simple to get going with the exercises: all you have to do is open the exercises, take a look at the questions, and try to answer them!

The dataset for these exercises is for a newly created country club, with a set of members, facilities such as tennis courts, and booking history for those facilities. Amongst other things, the club wants to understand how they can use their information to analyse facility usage/demand. **Please note:** this dataset is designed purely for supporting an interesting array of exercises, and the database schema is flawed in several aspects - please don't take it as an example of good design. We'll start off with a look at the Members table:

```
CREATE TABLE cd.members
(
    memid integer NOT NULL,
    surname character varying(200) NOT NULL,
    firstname character varying(200) NOT NULL,
    address character varying(300) NOT NULL,
    zipcode integer NOT NULL,
    telephone character varying(20) NOT NULL,
    recommendedby integer,
    joindate timestamp not null,
    CONSTRAINT members_pk PRIMARY KEY (memid),
    CONSTRAINT fk_members_recommendedby FOREIGN KEY (recommendedby)
        REFERENCES cd.members(memid) ON DELETE SET NULL
);
```

Each member has an ID (not guaranteed to be sequential), basic address information, a reference to the member that recommended them (if any), and a timestamp for when they joined. The addresses in the dataset are entirely (and unrealistically) fabricated.

```
CREATE TABLE cd.facilities
(
    facid integer NOT NULL,
    name character varying(100) NOT NULL,
    membercost numeric NOT NULL,
    guestcost numeric NOT NULL,
    initialoutlay numeric NOT NULL,
    monthlymaintenance numeric NOT NULL,
    CONSTRAINT facilities_pk PRIMARY KEY (facid)
);
```

The facilities table lists all the bookable facilities that the country club possesses. The club stores id/name information, the cost to book both members and guests, the initial cost to build the facility, and estimated monthly upkeep costs. They hope to use this information to track how financially worthwhile each facility is.

```
CREATE TABLE cd.bookings
(
    bookid integer NOT NULL,
    facid integer NOT NULL,
    memid integer NOT NULL,
    starttime timestamp NOT NULL,
    slots integer NOT NULL,
    CONSTRAINT bookings_pk PRIMARY KEY (bookid),
    CONSTRAINT fk_bookings_facid FOREIGN KEY (facid) REFERENCES
cd.facilities(facid),
    CONSTRAINT fk_bookings_memid FOREIGN KEY (memid) REFERENCES cd.members(memid)
);
```

Finally, there's a table tracking bookings of facilities. This stores the facility id, the member who made the booking, the start of the booking, and how many half hour 'slots' the booking was made for. This idiosyncratic design will make certain queries more difficult, but should provide you with some interesting challenges - as well as prepare you for the horror of working with some real-world databases :-).

Okay, that should be all the information you need. You can select a category of query to try from the menu above, or alternatively **start from the beginning**.

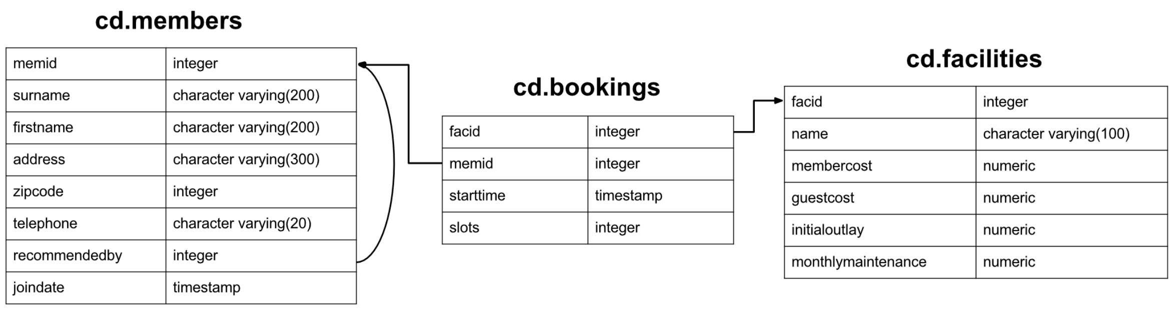
I want to use my own Postgres system

No problem! Getting up and running isn't too hard. First, you'll need an install of PostgreSQL, which you can get from **here**. Once you have it started, **download the SQL**.

Finally, run `psql -U <username> -f clubdata.sql -d postgres -x -q` to create the 'exercises' database, the Postgres 'pgexercises' user, the tables, and to load the data in. Note that you may find that the sort order of your results differs from those shown on the web site: that's probably because your Postgres is set up using a different locale to that used by PGExercises (which uses the C locale)

When you're running queries, you may find psql a little clunky. If so, I recommend trying out pgAdmin or the Eclipse database development tools.

Schema



cd.members

memid	integer
surname	character varying(200)
firstname	character varying(200)
address	character varying(300)
zipcode	integer
telephone	character varying(20)
recommendedby	integer
joindate	timestamp

cd.bookings

facid	integer
memid	integer
starttime	timestamp
slots	integer

cd.facilities

facid	integer
name	character varying(100)
membercost	numeric
guestcost	numeric
initialoutlay	numeric
monthlymaintenance	numeric

Simple SQL Queries

This category deals with the basics of SQL. It covers select and where clauses, case expressions, unions, and a few other odds and ends. If you're already educated in SQL you will probably find these exercises fairly easy. If not, you should find them a good point to start learning for the more difficult categories ahead!

If you struggle with these questions, I strongly recommend **Learning SQL**, by Alan Beaulieu, as a concise and well-written book on the subject. If you're interested in the fundamentals of database systems (as opposed to just how to use them), you should also investigate An Introduction to Database Systems by C.J. Date.

Retrieve everything from a table

How can you retrieve all the information from the cd.facilities table?

Expected results:

facid	name	membercost	guestcost	initialoutlay	monthlymaintenance
0	Tennis Court 1	5	25	10000	200
1	Tennis Court 2	5	25	8000	200
2	Badminton Court	0	15.5	4000	50
3	Table Tennis	0	5	320	10
4	Massage Room 1	35	80	4000	3000
5	Massage Room 2	35	80	4000	3000
6	Squash Court	3.5	17.5	5000	80
7	Snooker Table	0	5	450	15
8	Pool Table	0	5	400	15

Answer:

```
select * from cd.facilities;
```

The `SELECT` statement is the basic starting block for queries that read information out of the database. A minimal select statement is generally comprised of `select [some set of columns] from [some table or group of tables]`.

In this case, we want all of the information from the facilities table. The from section is easy - we just need to specify the `cd.facilities` table. 'cd' is the table's schema - a term used for a logical grouping of related information in the database.

Next, we need to specify that we want all the columns. Conveniently, there's a shorthand for 'all columns' - *. We can use this instead of laboriously specifying all the column names.

Retrieve specific columns from a table

You want to print out a list of all of the facilities and their cost to members. How would you retrieve a list of only facility names and costs?

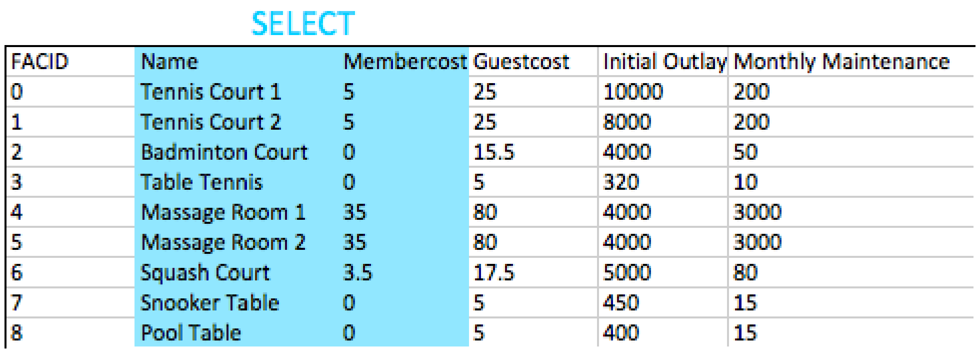
Expected results:

name	membercost
Tennis Court 1	5
Tennis Court 2	5
Badminton Court	0
Table Tennis	0
Massage Room 1	35
Massage Room 2	35
Squash Court	3.5
Snooker Table	0
Pool Table	0

Answer:

```
select name, membercost from cd.facilities;
```

For this question, we need to specify the columns that we want. We can do that with a simple comma-delimited list of column names specified to the select statement. All the database does is look at the columns available in the FROM clause, and return the ones we asked for, as illustrated below


SELECT

FACID	Name	Membercost	Guestcost	Initial Outlay	Monthly Maintenance
0	Tennis Court 1	5	25	10000	200
1	Tennis Court 2	5	25	8000	200
2	Badminton Court	0	15.5	4000	50
3	Table Tennis	0	5	320	10
4	Massage Room 1	35	80	4000	3000
5	Massage Room 2	35	80	4000	3000
6	Squash Court	3.5	17.5	5000	80
7	Snooker Table	0	5	450	15
8	Pool Table	0	5	400	15

Generally speaking, for non-throwaway queries it's considered desirable to specify the names of the columns you want in your queries rather than using *. This is because your application might not be able to cope if more columns get added into the table.

Control which rows are retrieved

How can you produce a list of facilities that charge a fee to members?

Expected results:

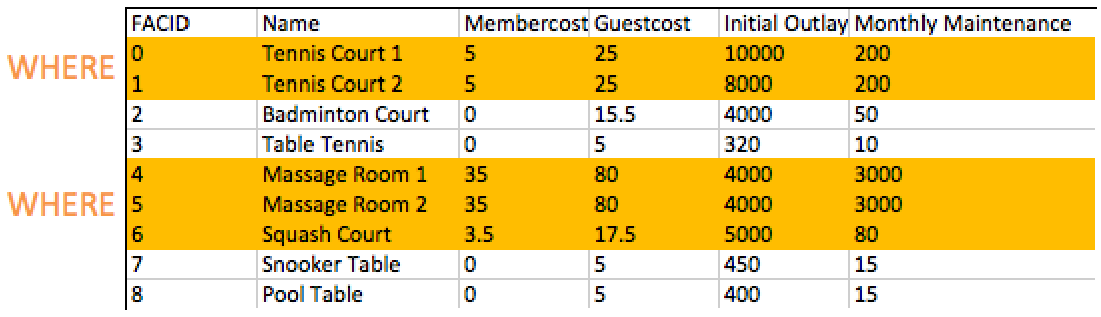
facid	name	membercost	guestcost	initialoutlay	monthlymaintenance
0	Tennis Court 1	5	25	10000	200
1	Tennis Court 2	5	25	8000	200
4	Massage Room 1	35	80	4000	3000
5	Massage Room 2	35	80	4000	3000
6	Squash Court	3.5	17.5	5000	80

Answer:

```
select * from cd.facilities where membercost > 0;
```

The `FROM` clause is used to build up a set of candidate rows to read results from. In our examples so far, this set of rows has simply been the contents of a table. In future we will explore joining, which allows us to create much more interesting candidates.

Once we've built up our set of candidate rows, the `WHERE` clause allows us to filter for the rows we're interested in - in this case, those with a membercost of more than zero. As you will see in later exercises, `WHERE` clauses can have multiple components combined with boolean logic - it's possible to, for instance, search for facilities with a cost greater than 0 and less than 10. The filtering action of the `WHERE` clause on the facilities table is illustrated below:


WHERE
WHERE

FACID	Name	Membercost	Guestcost	Initial Outlay	Monthly Maintenance
0	Tennis Court 1	5	25	10000	200
1	Tennis Court 2	5	25	8000	200
2	Badminton Court	0	15.5	4000	50
3	Table Tennis	0	5	320	10
4	Massage Room 1	35	80	4000	3000
5	Massage Room 2	35	80	4000	3000
6	Squash Court	3.5	17.5	5000	80
7	Snooker Table	0	5	450	15
8	Pool Table	0	5	400	15

Control which rows are retrieved, Part 2

How can you produce a list of facilities that charge a fee to members, and that fee is less than 1/50th of the monthly maintenance cost? Return the facid, facility name, member cost, and monthly maintenance of the facilities in question.

Expected results:

facid	name	membercost	monthlymaintenance
4	Massage Room 1	35	3000
5	Massage Room 2	35	3000

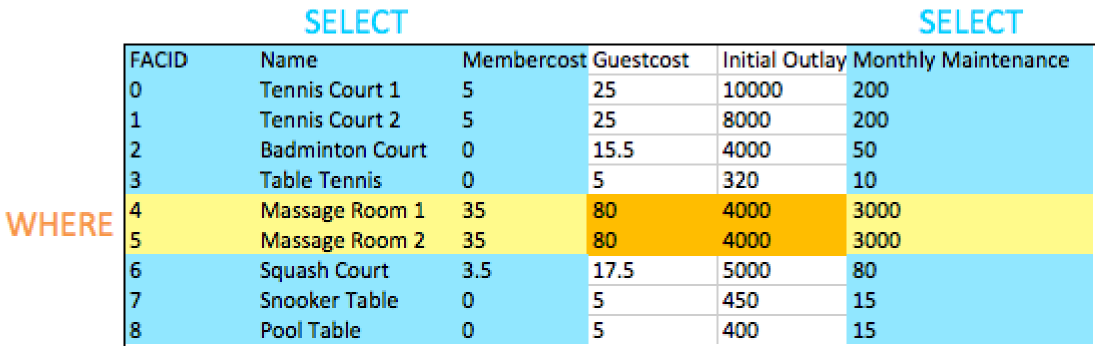
Answer:

```sql
select facid, name, membercost, monthlymaintenance
    from cd.facilities
    where
        membercost > 0 and
        (membercost < monthlymaintenance/50.0);
```

The `WHERE` clause allows us to filter for the rows we're interested in - in this case, those with a membercost of more than zero, and less than 1/50th of the monthly maintenance cost. As you can see, the massage rooms are very expensive to run thanks to staffing costs!

When we want to test for two or more conditions, we use `AND` to combine them. We can, as you might expect, use `OR` to test whether either of a pair of conditions is true.

You might have noticed that this is our first query that combines a `WHERE` clause with selecting specific columns. You can see in the image below the effect of this: the intersection of the selected columns and the selected rows gives us the data to return. This may not seem too interesting now, but as we add in more complex operations like joins later, you'll see the simple elegance of this behaviour.



FACID	Name	Membercost	Guestcost	Initial Outlay	Monthly Maintenance
0	Tennis Court 1	5	25	10000	200
1	Tennis Court 2	5	25	8000	200
2	Badminton Court	0	15.5	4000	50
3	Table Tennis	0	5	320	10
4	Massage Room 1	35	80	4000	3000
5	Massage Room 2	35	80	4000	3000
6	Squash Court	3.5	17.5	5000	80
7	Snooker Table	0	5	450	15
8	Pool Table	0	5	400	15

Basic string searches

How can you produce a list of all facilities with the word 'Tennis' in their name?

Expected results:

facid	name	membercost	guestcost	initialoutlay	monthlymaintenance
0	Tennis Court 1	5	25	10000	200
1	Tennis Court 2	5	25	8000	200
3	Table Tennis	0	5	320	10

Answer:

```sql
select *
    from cd.facilities
    where
        name like '%Tennis%';
```

SQL's `LIKE` operator provides simple pattern matching on strings. It's pretty much universally implemented, and is nice and simple to use - it just takes a string with the % character matching any string, and _ matching any single character. In this case, we're looking for names containing the word 'Tennis', so putting a % on either side fits the bill.

There's other ways to accomplish this task: Postgres supports regular expressions with the ~ operator, for example. Use whatever makes you feel comfortable, but do be aware that the `LIKE` operator is much more portable between systems.

Matching against multiple possible values

How can you retrieve the details of facilities with ID 1 and 5? Try to do it without using the OR operator.

Expected results:

facid	name	membercost	guestcost	initialoutlay	monthlymaintenance
1	Tennis Court 2	5	25	8000	200
5	Massage Room 2	35	80	4000	3000

Answer:

```
select *
    from cd.facilities
    where
        facid in (1,5);
```

The obvious answer to this question is to use a `WHERE` clause that looks like `where facid = 1 or facid = 5`. An alternative that is easier with large numbers of possible matches is the `IN` operator. The `IN` operator takes a list of possible values, and matches them against (in this case) the facid. If one of the values matches, the where clause is true for that row, and the row is returned.

The `IN` operator is a good early demonstrator of the elegance of the relational model. The argument it takes is not just a list of values - it's actually a table with a single column. Since queries also return tables, if you create a query that returns a single column, you can feed those results into an `IN` operator. To give a toy example:

```
select *
    from cd.facilities
    where
        facid in (
            select facid from cd.facilities
            );
```

This example is functionally equivalent to just selecting all the facilities, but shows you how to feed the results of one query into another. The inner query is called a *subquery* .

Classify results into bucket

How can you produce a list of facilities, with each labelled as 'cheap' or 'expensive' depending on if their monthly maintenance cost is more than $100? Return the name and monthly maintenance of the facilities in question.

Expected results:

name	cost
Tennis Court 1	expensive
Tennis Court 2	expensive
Badminton Court	cheap
Table Tennis	cheap
Massage Room 1	expensive
Massage Room 2	expensive
Squash Court	cheap
Snooker Table	cheap
Pool Table	cheap

Answer:

```
select name,
    case when (monthlymaintenance > 100) then
        'expensive'
    else
        'cheap'
    end as cost
    from cd.facilities;
```

This exercise contains a few new concepts. The first is the fact that we're doing computation in the area of the query between `SELECT` and `FROM`. Previously we've only used this to select columns that we want to return, but you can put anything in here that will produce a single result per returned row - including subqueries.

The second new concept is the `CASE` statement itself. `CASE` is effectively like if/switch statements in other languages, with a form as shown in the query. To add a 'middling' option, we would simply insert another `when ... then` section.

Finally, there's the `AS` operator. This is simply used to label columns or expressions, to make them display more nicely or to make them easier to reference when used as part of a subquery.

Working with dates

How can you produce a list of members who joined after the start of September 2012? Return the memid, surname, firstname, and joindate of the members in question.

Expected results:

memid	surname	firstname	joindate
24	Sarwin	Ramnaresh	2012-09-01 08:44:42
26	Jones	Douglas	2012-09-02 18:43:05
27	Rumney	Henrietta	2012-09-05 08:42:35
28	Farrell	David	2012-09-15 08:22:05
29	Worthington-Smyth	Henry	2012-09-17 12:27:15
30	Purview	Millicent	2012-09-18 19:04:01
33	Tupperware	Hyacinth	2012-09-18 19:32:05
35	Hunt	John	2012-09-19 11:32:45
36	Crumpet	Erica	2012-09-22 08:36:38
37	Smith	Darren	2012-09-26 18:08:45

Answer:

```
select memid, surname, firstname, joindate
    from cd.members
    where joindate ≥ '2012-09-01';
```

This is our first look at SQL timestamps. They're formatted in descending order of magnitude: `YYYY-MM-DD HH:MM:SS.nnnnnn`. We can compare them just like we might a unix timestamp, although getting the differences between dates is a little more involved (and powerful!). In this case, we've just specified the date portion of the timestamp. This gets automatically cast by postgres into the full timestamp `2012-09-01 00:00:00`.

Removing duplicates, and ordering results

How can you produce an ordered list of the first 10 surnames in the members table? The list must not contain duplicates.

Expected results:

surname
Bader
Baker
Boothe
Butters
Coplin
Crumpet
Dare
Farrell
GUEST
Genting

Answer:

```
select distinct surname
    from cd.members
order by surname
limit 10;
```

There's three new concepts here, but they're all pretty simple.

- Specifying `DISTINCT` after `SELECT` removes duplicate rows from the result set. Note that this applies to *rows* : if row A has multiple columns, row B is only equal to it if the values in all columns are the same. As a general rule, don't use `DISTINCT` in a willy-nilly fashion - it's not free to remove duplicates from large query result sets, so do it as-needed.
- Specifying `ORDER BY` (after the `FROM` and `WHERE` clauses, near the end of the query) allows results to be ordered by a column or set of columns (comma separated).
- The `LIMIT` keyword allows you to limit the number of results retrieved. This is useful for getting results a page at a time, and can be combined with the `OFFSET` keyword to get following pages. This is the same approach used by MySQL and is very convenient - you may, unfortunately, find that this process is a little more complicated in other DBs.

Combining results from multiple queries

You, for some reason, want a combined list of all surnames and all facility names. Yes, this is a contrived example :-). Produce that list!

Expected results:

surname
Tennis Court 2
Worthington-Smyth
Badminton Court
Pinker
Dare
Bader
Mackenzie
Crumpet
Massage Room 1
Squash Court

Answer:

```
select surname
    from cd.members
union
select name
    from cd.facilities;
```

The `UNION` operator does what you might expect: combines the results of two SQL queries into a single table. The caveat is that both results from the two queries must have the same number of columns and compatible data types.

`UNION` removes duplicate rows, while `UNION ALL` does not. Use `UNION ALL` by default, unless you care about duplicate results.

Simple aggregation

You'd like to get the signup date of your last member. How can you retrieve this information?

Expected results:

latest
2012-09-26 18:08:45

Answer:

```
select max(joindate) as latest
    from cd.members;
```

This is our first foray into SQL's aggregate functions. They're used to extract information about whole groups of rows, and allow us to easily ask questions like:

- What's the most expensive facility to maintain on a monthly basis?
- Who has recommended the most new members?

- How much time has each member spent at our facilities?

The MAX aggregate function here is very simple: it receives all the possible values for joindate, and outputs the one that's biggest. There's a lot more power to aggregate functions, which you will come across in future exercises.

More aggregation

You'd like to get the first and last name of the last member(s) who signed up - not just the date. How can you do that?

Expected results:

firstname	surname	joindate
Darren	Smith	2012-09-26 18:08:45

Answer:

```
select firstname, surname, joindate
    from cd.members
    where joindate =
        (select max(joindate)
            from cd.members);
```

In the suggested approach above, you use a *subquery* to find out what the most recent joindate is. This subquery returns a *scalar* table - that is, a table with a single column and a single row. Since we have just a single value, we can substitute the subquery anywhere we might put a single constant value. In this case, we use it to complete the `WHERE` clause of a query to find a given member.

You might hope that you'd be able to do something like below:

```
select firstname, surname, max(joindate)
        from cd.members
```

Unfortunately, this doesn't work. The `MAX` function doesn't restrict rows like the `WHERE` clause does - it simply takes in a bunch of values and returns the biggest one. The database is then left wondering how to pair up a long list of names with the single join date that's come out of the max function, and fails. Instead, you're left having to say 'find me the row(s) which have a join date that's the same as the maximum join date'.

As mentioned by the hint, there's other ways to get this job done - one example is below. In this approach, rather than explicitly finding out what the last joined date is, we simply order our members table in descending order of join date, and pick off the first one. Note that this approach does not cover the extremely unlikely eventuality of two people joining at the exact same time :-).

```
select firstname, surname, joindate
    from cd.members
order by joindate desc
limit 1;
```

Joins and Subqueries

This category deals primarily with a foundational concept in relational database systems: joining. Joining allows you to combine related information from multiple tables to answer a question. This isn't just beneficial for ease of querying: a lack of join capability encourages denormalisation of data, which increases the complexity of keeping your data internally consistent.

This topic covers inner, outer, and self joins, as well as spending a little time on subqueries (queries within queries). If you struggle with these questions, I strongly recommend **Learning SQL**, by Alan Beaulieu, as a concise and well-written book on the subject.

Retrieve the start times of members' bookings

How can you produce a list of the start times for bookings by members named 'David Farrell'?

Expected results:

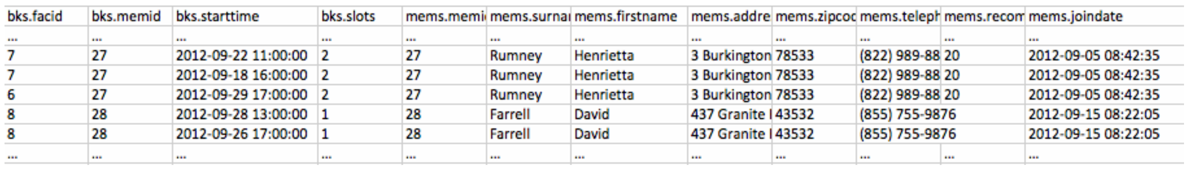
starttime
2012-09-18 09:00:00
2012-09-18 17:30:00
2012-09-18 13:30:00
2012-09-18 20:00:00
2012-09-19 09:30:00
2012-09-19 15:00:00
2012-09-19 12:00:00
2012-09-20 15:30:00
2012-09-20 11:30:00
2012-09-20 14:00:00

Answer:

```
select bks.starttime
    from
        cd.bookings bks
        inner join cd.members mems
            on mems.memid = bks.memid
    where
        mems.firstname='David'
        and mems.surname='Farrell';
```

The most commonly used kind of join is the `INNER JOIN`. What this does is combine two tables based on a join expression - in this case, for each member id in the members table, we're looking for matching values in the bookings table. Where we find a match, a row combining the values for each table is returned. Note that we've given each table an *alias* (bks and mems). This is used for two reasons: firstly, it's convenient, and secondly we might join to the same table several times, requiring us to distinguish between columns from each different time the table was joined in.

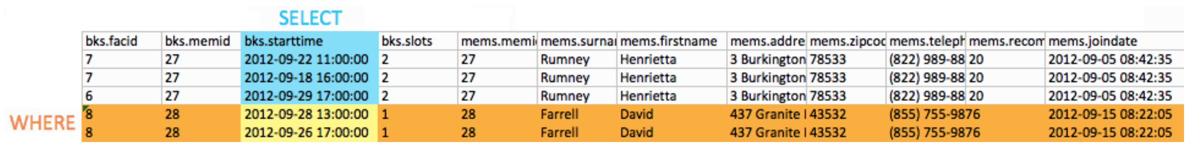
Let's ignore our select and where clauses for now, and focus on what the `FROM` statement produces. In all our previous examples, `FROM` has just been a simple table. What is it now? Another table! This time, it's produced as a composite of bookings and members. You can see a subset of the output of the join below:



bks.facid	bks.memid	bks.starttime	bks.slots	mems.memi	mems.surna	mems.firstname	mems.addre	mems.zipcoc	mems.teleph	mems.recom	mems.joindate
...	...	...	...	...	...	...	...	...	...	...	...
7	27	2012-09-22 11:00:00	2	27	Rumney	Henrietta	3 Burkington	78533	(822) 989-88	20	2012-09-05 08:42:35
7	27	2012-09-18 16:00:00	2	27	Rumney	Henrietta	3 Burkington	78533	(822) 989-88	20	2012-09-05 08:42:35
6	27	2012-09-29 17:00:00	2	27	Rumney	Henrietta	3 Burkington	78533	(822) 989-88	20	2012-09-05 08:42:35
8	28	2012-09-28 13:00:00	1	28	Farrell	David	437 Granite I	43532	(855) 755-9876		2012-09-15 08:22:05
8	28	2012-09-26 17:00:00	1	28	Farrell	David	437 Granite I	43532	(855) 755-9876		2012-09-15 08:22:05
...	...	...	...	...	...	...	...	...	...	...	...

For each member in the members table, the join has found all the matching member ids in the bookings table. For each match, it's then produced a row combining the row from the members table, and the row from the bookings table.

Obviously, this is too much information on its own, and any useful question will want to filter it down. In our query, we use the start of the `SELECT` clause to pick columns, and the `WHERE` clause to pick rows, as illustrated below:


SELECT
WHERE

bks.facid	bks.memid	bks.starttime	bks.slots	mems.memi	mems.surna	mems.firstname	mems.addre	mems.zipcoc	mems.teleph	mems.recom	mems.joindate
7	27	2012-09-22 11:00:00	2	27	Rumney	Henrietta	3 Burkington	78533	(822) 989-88	20	2012-09-05 08:42:35
7	27	2012-09-18 16:00:00	2	27	Rumney	Henrietta	3 Burkington	78533	(822) 989-88	20	2012-09-05 08:42:35
6	27	2012-09-29 17:00:00	2	27	Rumney	Henrietta	3 Burkington	78533	(822) 989-88	20	2012-09-05 08:42:35
8	28	2012-09-28 13:00:00	1	28	Farrell	David	437 Granite I	43532	(855) 755-9876		2012-09-15 08:22:05
8	28	2012-09-26 17:00:00	1	28	Farrell	David	437 Granite I	43532	(855) 755-9876		2012-09-15 08:22:05

That's all we need to find David's bookings! In general, I encourage you to remember that the output of the `FROM` clause is essentially one big table that you then filter information out of. This may sound inefficient - but don't worry, under the covers the DB will be behaving much more intelligently :-).

One final note: there's two different syntaxes for inner joins. I've shown you the one I prefer, that I find more consistent with other join types. You'll commonly see a different syntax, shown below:

```
select bks.starttime
        from
                cd.bookings bks,
                cd.members mems
        where
                mems.firstname='David'
                and mems.surname='Farrell'
                and mems.memid = bks.memid;
```

This is functionally exactly the same as the approved answer. If you feel more comfortable with this syntax, feel free to use it!

Work out the start times of bookings for tennis courts

How can you produce a list of the start times for bookings for tennis courts, for the date '2012-09-21'? Return a list of start time and facility name pairings, ordered by the time.

Expected results:

start	name
2012-09-21 08:00:00	Tennis Court 1
2012-09-21 08:00:00	Tennis Court 2
2012-09-21 09:30:00	Tennis Court 1
2012-09-21 10:00:00	Tennis Court 2
2012-09-21 11:30:00	Tennis Court 2
2012-09-21 12:00:00	Tennis Court 1
2012-09-21 13:30:00	Tennis Court 1
2012-09-21 14:00:00	Tennis Court 2
2012-09-21 15:30:00	Tennis Court 1
2012-09-21 16:00:00	Tennis Court 2
2012-09-21 17:00:00	Tennis Court 1
2012-09-21 18:00:00	Tennis Court 2

Answer:

```
select bks.starttime as start, facs.name as name
    from
        cd.facilities facs
        inner join cd.bookings bks
            on facs.facid = bks.facid
    where
        facs.facid in (0,1) and
        bks.starttime >= '2012-09-21' and
        bks.starttime < '2012-09-22'
order by bks.starttime;
```

This is another `INNER JOIN` query, although it has a fair bit more complexity in it! The `FROM` part of the query is easy - we're simply joining facilities and bookings tables together on the facid. This produces a table where, for each row in bookings, we've attached detailed information about the facility being booked.

On to the `WHERE` component of the query. The checks on starttime are fairly self explanatory - we're making sure that all the bookings start between the specified dates. Since we're only interested in tennis courts, we're also using the `IN` operator to tell the database system to only give us back facility IDs 0 or 1 - the IDs of the courts. There's other ways to express this: We could have used `where facs.facid = 0 or facs.facid = 1`, or even `where facs.name like 'Tennis%'`.

The rest is pretty simple: we `SELECT` the columns we're interested in, and `ORDER BY` the start time.

Produce a list of all members who have recommended another member

How can you output a list of all members who have recommended another member? Ensure that there are no duplicates in the list, and that results are ordered by (surname, firstname).

Expected results:

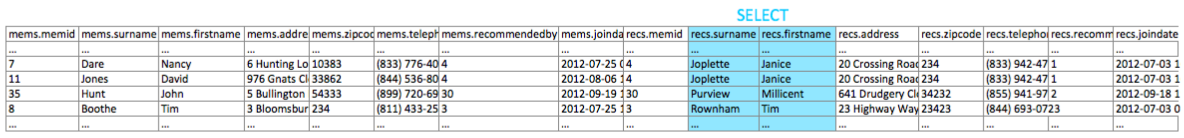
firstname	surname
Florence	Bader
Timothy	Baker
Gerald	Butters
Jemima	Farrell
Matthew	Genting
David	Jones
Janice	Joplette
Millicent	Purview
Tim	Rownam
Darren	Smith
Tracy	Smith
Ponder	Stibbons
Burton	Tracy

Answer:

```
select distinct recs.firstname as firstname, recs.surname as surname
    from
        cd.members mems
        inner join cd.members recs
            on recs.memid = mems.recommendedby
order by surname, firstname;
```

Here's a concept that some people find confusing: you can join a table to itself! This is really useful if you have columns that reference data in the same table, like we do with recommendedby in cd.members.

If you're having trouble visualising this, remember that this works just the same as any other inner join. Our join takes each row in members that has a recommendedby value, and looks in members again for the row which has a matching member id. It then generates an output row combining the two members entries. This looks like the diagram below:



mems.memid	mems.surname	mems.firstname	mems.addre	mems.zipcod	mems.teleph	mems.recommendedby	mems.joinda	recs.memid	recs.surname	recs.firstname	recs.address	recs.zipcode	recs.telephoi	recs.recomm	recs.joindate
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
7	Dare	Nancy	6 Hunting Lo	10383	(833) 776-40	4	2012-07-25 (	4	Joplette	Janice	20 Crossing Road	234	(833) 942-47	1	2012-07-03 1
11	Jones	David	976 Gnats Cl	33862	(844) 536-80	4	2012-08-06 1	4	Joplette	Janice	20 Crossing Road	234	(833) 942-47	1	2012-07-03 1
35	Hunt	John	5 Bullington	54333	(899) 720-69	30	2012-09-19 1	30	Purview	Millicent	641 Drudgery Cl	34232	(855) 941-97	2	2012-09-18 1
8	Boothe	Tim	3 Bloomsbur	234	(811) 433-25	3	2012-07-25 1	3	Rownham	Tim	23 Highway Way	23423	(844) 693-0723		2012-07-03 0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Note that while we might have two 'surname' columns in the output set, they can be distinguished by their table aliases. Once we've selected the columns that we want, we simply use `DISTINCT` to ensure that there are no duplicates.

Produce a list of all members, along with their recommender

How can you output a list of all members, including the individual who recommended them (if any)? Ensure that results are ordered by (surname, firstname).

Expected results:

memfname	memsname	recfname	recsname
Florence	Bader	Ponder	Stibbons
Anne	Baker	Ponder	Stibbons
Timothy	Baker	Jemima	Farrell
Tim	Boothe	Tim	Rownam
Gerald	Butters	Darren	Smith
Joan	Coplin	Timothy	Baker
Erica	Crumpet	Tracy	Smith
Nancy	Dare	Janice	Joplette
David	Farrell		
Jemima	Farrell		
GUEST	GUEST		
Matthew	Genting	Gerald	Butters
John	Hunt	Millicent	Purview
David	Jones	Janice	Joplette
Douglas	Jones	David	Jones
Janice	Joplette	Darren	Smith
Anna	Mackenzie	Darren	Smith
Charles	Owen	Darren	Smith
David	Pinker	Jemima	Farrell
Millicent	Purview	Tracy	Smith
Tim	Rownam		
Henrietta	Rumney	Matthew	Genting
Ramnaresh	Sarwin	Florence	Bader
Darren	Smith		
Darren	Smith		
Jack	Smith	Darren	Smith
Tracy	Smith		
Ponder	Stibbons	Burton	Tracy
Burton	Tracy		
Hyacinth	Tupperware		

memfname	memsname	recfname	recsname
Henry	Worthington-Smyth	Tracy	Smith

Answer:

```
select mems.firstname as memfname, mems.surname as memsname, recs.firstname as
recfname, recs.surname as recsname
    from
        cd.members mems
        left outer join cd.members recs
            on recs.memid = mems.recommendedby
order by memsname, memfname;
```

Let's introduce another new concept: the `LEFT OUTER JOIN`. These are best explained by the way in which they differ from inner joins. Inner joins take a left and a right table, and look for matching rows based on a join condition (`ON`). When the condition is satisfied, a joined row is produced. A `LEFT OUTER JOIN` operates similarly, except that if a given row on the left hand table doesn't match anything, it still produces an output row. That output row consists of the left hand table row, and a bunch of `NULLS` in place of the right hand table row.

This is useful in situations like this question, where we want to produce output with optional data. We want the names of all members, and the name of their recommender *if that person exists*. You can't express that properly with an inner join.

As you may have guessed, there's other outer joins too. The `RIGHT OUTER JOIN` is much like the `LEFT OUTER JOIN`, except that the left hand side of the expression is the one that contains the optional data. The rarely-used `FULL OUTER JOIN` treats both sides of the expression as optional.

Produce a list of all members who have used a tennis court

How can you produce a list of all members who have used a tennis court? Include in your output the name of the court, and the name of the member formatted as a single column. Ensure no duplicate data, and order by the member name.

Expected results:

member	facility
Anne Baker	Tennis Court 2
Anne Baker	Tennis Court 1
Burton Tracy	Tennis Court 2
Burton Tracy	Tennis Court 1
Charles Owen	Tennis Court 2
Charles Owen	Tennis Court 1
Darren Smith	Tennis Court 2
David Farrell	Tennis Court 2
David Farrell	Tennis Court 1
David Jones	Tennis Court 1
David Jones	Tennis Court 2
David Pinker	Tennis Court 1
Douglas Jones	Tennis Court 1
Erica Crumpet	Tennis Court 1
Florence Bader	Tennis Court 1
Florence Bader	Tennis Court 2
GUEST GUEST	Tennis Court 2
GUEST GUEST	Tennis Court 1
Gerald Butters	Tennis Court 1
Gerald Butters	Tennis Court 2
Henrietta Rumney	Tennis Court 2
Jack Smith	Tennis Court 1
Jack Smith	Tennis Court 2
Janice Joplette	Tennis Court 1
Janice Joplette	Tennis Court 2
Jemima Farrell	Tennis Court 2
Jemima Farrell	Tennis Court 1
Joan Coplin	Tennis Court 1
John Hunt	Tennis Court 1
John Hunt	Tennis Court 2

member	facility
Matthew Genting	Tennis Court 1
Millicent Purview	Tennis Court 2
Nancy Dare	Tennis Court 2
Nancy Dare	Tennis Court 1
Ponder Stibbons	Tennis Court 2
Ponder Stibbons	Tennis Court 1
Ramnaresh Sarwin	Tennis Court 2
Ramnaresh Sarwin	Tennis Court 1
Tim Boothe	Tennis Court 1
Tim Boothe	Tennis Court 2
Tim Rownam	Tennis Court 1
Tim Rownam	Tennis Court 2
Timothy Baker	Tennis Court 2
Timothy Baker	Tennis Court 1
Tracy Smith	Tennis Court 2
Tracy Smith	Tennis Court 1

Answer:

```
select distinct mems.firstname || ' ' || mems.surname as member, facs.name as
facility
    from
        cd.members mems
        inner join cd.bookings bks
            on mems.memid = bks.memid
        inner join cd.facilities facs
            on bks.facid = facs.facid
    where
        bks.facid in (0,1)
order by member
```

This exercise is largely a more complex application of what you've learned in prior questions. It's also the first time we've used more than one join, which may be a little confusing for some. When reading join expressions, remember that a join is effectively a function that takes two tables, one labelled the left table, and the other the right. This is easy to visualise with just one join in the query, but a little more confusing with two.

Our second `INNER JOIN` in this query has a right hand side of cd.facilities. That's easy enough to grasp. The left hand side, however, is the table returned by joining cd.members to cd.bookings. It's important to emphasise this: the relational model is all about tables. The output of any join is another table. The output of a query is a table. Single columned lists are tables. Once you grasp that, you've grasped the fundamental beauty of the model.

As a final note, we do introduce one new thing here: the `||` operator is used to concatenate strings.

Produce a list of costly bookings

How can you produce a list of bookings on the day of 2012-09-14 which will cost the member (or guest) more than $30? Remember that guests have different costs to members (the listed costs are per half-hour 'slot'), and the guest user is always ID 0. Include in your output the name of the facility, the name of the member formatted as a single column, and the cost. Order by descending cost, and do not use any subqueries.

Expected results:

member	facility	cost
GUEST GUEST	Massage Room 2	320
GUEST GUEST	Massage Room 1	160
GUEST GUEST	Massage Room 1	160
GUEST GUEST	Massage Room 1	160
GUEST GUEST	Tennis Court 2	150
Jemima Farrell	Massage Room 1	140
GUEST GUEST	Tennis Court 1	75
GUEST GUEST	Tennis Court 2	75
GUEST GUEST	Tennis Court 1	75
Matthew Genting	Massage Room 1	70
Florence Bader	Massage Room 2	70
GUEST GUEST	Squash Court	70.0
Jemima Farrell	Massage Room 1	70
Ponder Stibbons	Massage Room 1	70
Burton Tracy	Massage Room 1	70
Jack Smith	Massage Room 1	70
GUEST GUEST	Squash Court	35.0
GUEST GUEST	Squash Court	35.0

Answer:

```
select mems.firstname || ' ' || mems.surname as member,
    facs.name as facility,
    case
        when mems.memid = 0 then
            bks.slots*facs.guestcost
        else
```

```
            bks.slots*facs.membercost
    end as cost
        from
                cd.members mems
                inner join cd.bookings bks
                        on mems.memid = bks.memid
                inner join cd.facilities facs
                        on bks.facid = facs.facid
        where
        bks.starttime >= '2012-09-14' and
        bks.starttime < '2012-09-15' and (
            (mems.memid = 0 and bks.slots*facs.guestcost > 30) or
            (mems.memid != 0 and bks.slots*facs.membercost > 30)
        )
order by cost desc;
```

This is a bit of a complicated one! While its more complex logic than we've used previously, there's not an awful lot to remark upon. The `WHERE` clause restricts our output to sufficiently costly rows on 2012-09-14, remembering to distinguish between guests and others. We then use a `CASE` statement in the column selections to output the correct cost for the member or guest.

Produce a list of all members, along with their recommender, using no joins

How can you output a list of all members, including the individual who recommended them (if any), without using any joins? Ensure that there are no duplicates in the list, and that each firstname + surname pairing is formatted as a column and ordered.

Expected results:

member	recommender
Anna Mackenzie	Darren Smith
Anne Baker	Ponder Stibbons
Burton Tracy	
Charles Owen	Darren Smith
Darren Smith	
David Farrell	
David Jones	Janice Joplette
David Pinker	Jemima Farrell
Douglas Jones	David Jones
Erica Crumpet	Tracy Smith
Florence Bader	Ponder Stibbons
GUEST GUEST	
Gerald Butters	Darren Smith
Henrietta Rumney	Matthew Genting
Henry Worthington-Smyth	Tracy Smith
Hyacinth Tupperware	
Jack Smith	Darren Smith
Janice Joplette	Darren Smith
Jemima Farrell	
Joan Coplin	Timothy Baker
John Hunt	Millicent Purview
Matthew Genting	Gerald Butters
Millicent Purview	Tracy Smith
Nancy Dare	Janice Joplette
Ponder Stibbons	Burton Tracy
Ramnaresh Sarwin	Florence Bader
Tim Boothe	Tim Rownam
Tim Rownam	
Timothy Baker	Jemima Farrell
Tracy Smith	

Answer:

```
select distinct mems.firstname || ' ' ||  mems.surname as member,
    (select recs.firstname || ' ' || recs.surname as recommender
        from cd.members recs
        where recs.memid = mems.recommendedby
    )
    from
        cd.members mems
order by member;
```

This exercise marks the introduction of subqueries. Subqueries are, as the name implies, queries within a query. They're commonly used with aggregates, to answer questions like 'get me all the details of the member who has spent the most hours on Tennis Court 1'.

In this case, we're simply using the subquery to emulate an outer join. For every value of member, the subquery is run once to find the name of the individual who recommended them (if any). A subquery that uses information from the outer query in this way (and thus has to be run for each row in the result set) is known as a *correlated subquery* .

Produce a list of costly bookings, using a subquery

The **Produce a list of costly bookings** exercise contained some messy logic: we had to calculate the booking cost in both the WHERE clause and the CASE statement. Try to simplify this calculation using subqueries. For reference, the question was:

How can you produce a list of bookings on the day of 2012-09-14 which will cost the member (or guest) more than $30? Remember that guests have different costs to members (the listed costs are per half-hour 'slot'), and the guest user is always ID 0. Include in your output the name of the facility, the name of the member formatted as a single column, and the cost. Order by descending cost.

Expected results:

member	facility	cost
GUEST GUEST	Massage Room 2	320
GUEST GUEST	Massage Room 1	160
GUEST GUEST	Massage Room 1	160
GUEST GUEST	Massage Room 1	160
GUEST GUEST	Tennis Court 2	150
Jemima Farrell	Massage Room 1	140
GUEST GUEST	Tennis Court 1	75
GUEST GUEST	Tennis Court 2	75
GUEST GUEST	Tennis Court 1	75
Matthew Genting	Massage Room 1	70
Florence Bader	Massage Room 2	70
GUEST GUEST	Squash Court	70.0
Jemima Farrell	Massage Room 1	70
Ponder Stibbons	Massage Room 1	70
Burton Tracy	Massage Room 1	70
Jack Smith	Massage Room 1	70
GUEST GUEST	Squash Court	35.0
GUEST GUEST	Squash Court	35.0

Answer:

```
select member, facility, cost from (
    select
        mems.firstname || ' ' || mems.surname as member,
        facs.name as facility,
        case
            when mems.memid = 0 then
                bks.slots*facs.guestcost
            else
                bks.slots*facs.membercost
        end as cost
        from
            cd.members mems
            inner join cd.bookings bks
                on mems.memid = bks.memid
            inner join cd.facilities facs
                on bks.facid = facs.facid
        where
            bks.starttime >= '2012-09-14' and
            bks.starttime < '2012-09-15'
    ) as bookings
    where cost > 30
order by cost desc;
```

This answer provides a mild simplification to the previous iteration: in the no-subquery version, we had to calculate the member or guest's cost in both the `WHERE` clause and the `CASE` statement. In our new version, we produce an inline query that calculates the total booking cost for us, allowing the outer query to simply select the bookings it's looking for. For reference, you may also see subqueries in the `FROM` clause referred to as *inline views* .

Modifying Data

Querying data is all well and good, but at some point you're probably going to want to put data into your database! This section deals with inserting, updating, and deleting information. Operations that alter your data like this are collectively known as Data Manipulation Language, or DML.

In previous sections, we returned to you the results of the query you've performed. Since modifications like the ones we're making in this section don't return any query results, we instead show you the updated content of the table you're supposed to be working on. You can compare this with the table shown in 'Expected Results' to see how you've done.

If you struggle with these questions, I strongly recommend **Learning SQL** , by Alan Beaulieu.

Insert some data into a table

The club is adding a new facility - a spa. We need to add it into the facilities table. Use the following values:

- facid: 9, Name: 'Spa', membercost: 20, guestcost: 30, initialoutlay: 100000, monthlymaintenance: 800.

Expected results:

facid	name	membercost	guestcost	initialoutlay	monthlymaintenance
0	Tennis Court 1	5	25	10000	200
1	Tennis Court 2	5	25	8000	200
2	Badminton Court	0	15.5	4000	50
3	Table Tennis	0	5	320	10
4	Massage Room 1	35	80	4000	3000
5	Massage Room 2	35	80	4000	3000
6	Squash Court	3.5	17.5	5000	80
7	Snooker Table	0	5	450	15
8	Pool Table	0	5	400	15
9	Spa	20	30	100000	800

Answer:

```
insert into cd.facilities
    (facid, name, membercost, guestcost, initialoutlay, monthlymaintenance)
    values (9, 'Spa', 20, 30, 100000, 800);
```

`INSERT INTO ... VALUES` is the simplest way to insert data into a table. There's not a whole lot to discuss here: `VALUES` is used to construct a row of data, which the `INSERT` statement inserts into the table. It's a simple as that.

You can see that there's two sections in parentheses. The first is part of the `INSERT` statement, and specifies the columns that we're providing data for. The second is part of `VALUES`, and specifies the actual data we want to insert into each column.

If we're inserting data into every column of the table, as in this example, explicitly specifying the column names is optional. As long as you fill in data for all columns of the table, in the order they were defined when you created the table, you can do something like the following:

```
insert into cd.facilities values (9, 'Spa', 20, 30, 100000, 800);
```

Generally speaking, for SQL that's going to be reused I tend to prefer being explicit and specifying the column names.

Insert multiple rows of data into a table

In the previous exercise, you learned how to add a facility. Now you're going to add multiple facilities in one command. Use the following values:

- facid: 9, Name: 'Spa', membercost: 20, guestcost: 30, initialoutlay: 100000, monthlymaintenance: 800.
- facid: 10, Name: 'Squash Court 2', membercost: 3.5, guestcost: 17.5, initialoutlay: 5000, monthlymaintenance: 80.

Expected results:

facid	name	membercost	guestcost	initialoutlay	monthlymaintenance
0	Tennis Court 1	5	25	10000	200
1	Tennis Court 2	5	25	8000	200
2	Badminton Court	0	15.5	4000	50
3	Table Tennis	0	5	320	10
4	Massage Room 1	35	80	4000	3000
5	Massage Room 2	35	80	4000	3000
6	Squash Court	3.5	17.5	5000	80
7	Snooker Table	0	5	450	15
8	Pool Table	0	5	400	15
9	Spa	20	30	100000	800
10	Squash Court 2	3.5	17.5	5000	80

Answer:

```
insert into cd.facilities
    (facid, name, membercost, guestcost, initialoutlay, monthlymaintenance)
    values
        (9, 'Spa', 20, 30, 100000, 800),
        (10, 'Squash Court 2', 3.5, 17.5, 5000, 80);
```

`VALUES` can be used to generate more than one row to insert into a table, as seen in this example. Hopefully it's clear what's going on here: the output of `VALUES` is a table, and that table is copied into cd.facilities, the table specified in the `INSERT` command.

While you'll most commonly see `VALUES` when inserting data, Postgres allows you to use `VALUES` wherever you might use a `SELECT`. This makes sense: the output of both commands is a table, it's just that `VALUES` is a bit more ergonomic when working with constant data.

Similarly, it's possible to use `SELECT` wherever you see a `VALUES`. This means that you can `INSERT` the results of a `SELECT`. For example:

```
insert into cd.facilities
    (facid, name, membercost, guestcost, initialoutlay, monthlymaintenance)
    SELECT 9, 'Spa', 20, 30, 100000, 800
    UNION ALL
        SELECT 10, 'Squash Court 2', 3.5, 17.5, 5000, 80;
```

In later exercises you'll see us using `INSERT ... SELECT` to generate data to insert based on the information already in the database.

Insert calculated data into a table

Let's try adding the spa to the facilities table again. This time, though, we want to automatically generate the value for the next facid, rather than specifying it as a constant. Use the following values for everything else:

- Name: 'Spa', membercost: 20, guestcost: 30, initialoutlay: 100000, monthlymaintenance: 800.

Expected results:

facid	name	membercost	guestcost	initialoutlay	monthlymaintenance
0	Tennis Court 1	5	25	10000	200
1	Tennis Court 2	5	25	8000	200
2	Badminton Court	0	15.5	4000	50
3	Table Tennis	0	5	320	10
4	Massage Room 1	35	80	4000	3000
5	Massage Room 2	35	80	4000	3000
6	Squash Court	3.5	17.5	5000	80
7	Snooker Table	0	5	450	15
8	Pool Table	0	5	400	15
9	Spa	20	30	100000	800

Answer:

```
insert into cd.facilities
    (facid, name, membercost, guestcost, initialoutlay, monthlymaintenance)
    select (select max(facid) from cd.facilities)+1, 'Spa', 20, 30, 100000, 800;
```

In the previous exercises we used `VALUES` to insert constant data into the facilities table. Here, though, we have a new requirement: a dynamically generated ID. This gives us a real quality of life improvement, as we don't have to manually work out what the current largest ID is: the SQL command does it for us.

Since the `VALUES` clause is only used to supply constant data, we need to replace it with a query instead. The `SELECT` statement is fairly simple: there's an inner subquery that works out the next facid based on the largest current id, and the rest is just constant data. The output of the statement is a row that we insert into the facilities table.

While this works fine in our simple example, it's not how you would generally implement an incrementing ID in the real world. Postgres provides `SERIAL` types that are auto-filled with the next ID when you insert a row. As well as saving us effort, these types are also safer: unlike the answer given in this exercise, there's no need to worry about concurrent operations generating the same ID.

Update some existing data

We made a mistake when entering the data for the second tennis court. The initial outlay was 10000 rather than 8000: you need to alter the data to fix the error.

Expected results:

facid	name	membercost	guestcost	initialoutlay	monthlymaintenance
0	Tennis Court 1	5	25	10000	200
1	Tennis Court 2	5	25	10000	200
2	Badminton Court	0	15.5	4000	50
3	Table Tennis	0	5	320	10
4	Massage Room 1	35	80	4000	3000
5	Massage Room 2	35	80	4000	3000
6	Squash Court	3.5	17.5	5000	80
7	Snooker Table	0	5	450	15
8	Pool Table	0	5	400	15

Answer:

```
update cd.facilities
    set initialoutlay = 10000
    where facid = 1;
```

The `UPDATE` statement is used to alter existing data. If you're familiar with `SELECT` queries, it's pretty easy to read: the `WHERE` clause works in exactly the same fashion, allowing us to filter the set of rows we want to work with. These rows are then modified according to the specifications of the `SET` clause: in this case, setting the initial outlay.

The `WHERE` clause is extremely important. It's easy to get it wrong or even omit it, with disastrous results. Consider the following command:

```
update cd.facilities
    set initialoutlay = 10000;
```

There's no `WHERE` clause to filter for the rows we're interested in. The result of this is that the update runs on every row in the table! This is rarely what we want to happen.

Update multiple rows and columns at the same time

We want to increase the price of the tennis courts for both members and guests. Update the costs to be 6 for members, and 30 for guests.

facid	name	membercost	guestcost	initialoutlay	monthlymaintenance
0	Tennis Court 1	6	30	10000	200
1	Tennis Court 2	6	30	8000	200
2	Badminton Court	0	15.5	4000	50
3	Table Tennis	0	5	320	10
4	Massage Room 1	35	80	4000	3000
5	Massage Room 2	35	80	4000	3000
6	Squash Court	3.5	17.5	5000	80
7	Snooker Table	0	5	450	15
8	Pool Table	0	5	400	15

Answer:

```
update cd.facilities
    set
        membercost = 6,
        guestcost = 30
    where facid in (0,1);
```

The `SET` clause accepts a comma separated list of values that you want to update.

Update a row based on the contents of another row

We want to alter the price of the second tennis court so that it costs 10% more than the first one. Try to do this without using constant values for the prices, so that we can reuse the statement if we want to.

Expected results:

facid	name	membercost	guestcost	initialoutlay	monthlymaintenance
0	Tennis Court 1	5	25	10000	200
1	Tennis Court 2	5.5	27.5	8000	200
2	Badminton Court	0	15.5	4000	50
3	Table Tennis	0	5	320	10
4	Massage Room 1	35	80	4000	3000
5	Massage Room 2	35	80	4000	3000
6	Squash Court	3.5	17.5	5000	80
7	Snooker Table	0	5	450	15
8	Pool Table	0	5	400	15

Answer:

```
update cd.facilities facs
    set
        membercost = (select membercost * 1.1 from cd.facilities where facid = 0),
        guestcost = (select guestcost * 1.1 from cd.facilities where facid = 0)
    where facs.facid = 1;
```

Updating columns based on calculated data is not too intrinsically difficult: we can do so pretty easily using subqueries. You can see this approach in our selected answer.

As the number of columns we want to update increases, standard SQL can start to get pretty awkward: you don't want to be specifying a separate subquery for each of 15 different column updates. Postgres provides a nonstandard extension to SQL called `UPDATE ... FROM` that addresses this: it allows you to supply a `FROM` clause to generate values for use in the `SET` clause. Example below:

```
update cd.facilities facs
    set
        membercost = facs2.membercost * 1.1,
        guestcost = facs2.guestcost * 1.1
    from (select * from cd.facilities where facid = 0) facs2
    where facs.facid = 1;
```

Delete all bookings

As part of a clearout of our database, we want to delete all bookings from the cd.bookings table. How can we accomplish this?

Expected results:

bookid	facid	memid	starttime	slots

Answer:

```
delete from cd.bookings;
```

The `DELETE` statement does what it says on the tin: deletes rows from the table. Here, we show the command in its simplest form, with no qualifiers. In this case, it deletes everything from the table. Obviously, you should be careful with your deletes and make sure they're always limited - we'll see how to do that in the next exercise.

An alternative to unqualified `DELETE`s is the following:

```
truncate cd.bookings;
```

`TRUNCATE` also deletes everything in the table, but does so using a quicker underlying mechanism. It's not **perfectly safe in all circumstances**, though, so use judiciously. When in doubt, use `DELETE`.

Delete a member from the cd.members table

We want to remove member 37, who has never made a booking, from our database. How can we achieve that?

Expected results:

memid	surname	firstname	address	zipcode	telephone	recommendedby	joindate
0	GUEST	GUEST	GUEST	0	(000) 000-0000		2012-07-01 00:00:00
1	Smith	Darren	8 Bloomsbury Close, Boston	4321	555-555-5555		2012-07-02 12:02:05
2	Smith	Tracy	8 Bloomsbury Close, New York	4321	555-555-5555		2012-07-02 12:08:23
3	Rownam	Tim	23 Highway Way, Boston	23423	(844) 693-0723		2012-07-03 09:32:15
4	Joplette	Janice	20 Crossing Road, New York	234	(833) 942-4710	1	2012-07-03 10:25:05
5	Butters	Gerald	1065 Huntingdon Avenue, Boston	56754	(844) 078-4130	1	2012-07-09 10:44:09
6	Tracy	Burton	3 Tunisia Drive, Boston	45678	(822) 354-9973		2012-07-15 08:52:55
7	Dare	Nancy	6 Hunting Lodge Way, Boston	10383	(833) 776-4001	4	2012-07-25 08:59:12
8	Boothe	Tim	3 Bloomsbury Close, Reading, 00234	234	(811) 433-2547	3	2012-07-25 16:02:35
9	Stibbons	Ponder	5 Dragons Way, Winchester	87630	(833) 160-3900	6	2012-07-25 17:09:05
10	Owen	Charles	52 Cheshire Grove, Winchester, 28563	28563	(855) 542-5251	1	2012-08-03 19:42:37
11	Jones	David	976 Gnats Close, Reading	33862	(844) 536-8036	4	2012-08-06 16:32:55
12	Baker	Anne	55 Powdery Street, Boston	80743	844-076-5141	9	2012-08-10 14:23:22
13	Farrell	Jemima	103 Firth Avenue, North Reading	57392	(855) 016-0163		2012-08-10 14:28:01

memid	surname	firstname	address	zipcode	telephone	recommendedby	joindate
14	Smith	Jack	252 Binkington Way, Boston	69302	(822) 163-3254	1	2012-08-10 16:22:05
15	Bader	Florence	264 Ursula Drive, Westford	84923	(833) 499-3527	9	2012-08-10 17:52:03
16	Baker	Timothy	329 James Street, Reading	58393	833-941-0824	13	2012-08-15 10:34:25
17	Pinker	David	5 Impreza Road, Boston	65332	811 409-6734	13	2012-08-16 11:32:47
20	Genting	Matthew	4 Nunnington Place, Wingfield, Boston	52365	(811) 972-1377	5	2012-08-19 14:55:55
21	Mackenzie	Anna	64 Perkington Lane, Reading	64577	(822) 661-2898	1	2012-08-26 09:32:05
22	Coplin	Joan	85 Bard Street, Bloomington, Boston	43533	(822) 499-2232	16	2012-08-29 08:32:41
24	Sarwin	Ramnaresh	12 Bullington Lane, Boston	65464	(822) 413-1470	15	2012-09-01 08:44:42
26	Jones	Douglas	976 Gnats Close, Reading	11986	844 536-8036	11	2012-09-02 18:43:05
27	Rumney	Henrietta	3 Burkington Plaza, Boston	78533	(822) 989-8876	20	2012-09-05 08:42:35
28	Farrell	David	437 Granite Farm Road, Westford	43532	(855) 755-9876		2012-09-15 08:22:05
29	Worthington-Smyth	Henry	55 Jagbi Way, North Reading	97676	(855) 894-3758	2	2012-09-17 12:27:15
30	Purview	Millicent	641 Drudgery Close, Burnington, Boston	34232	(855) 941-9786	2	2012-09-18 19:04:01
33	Tupperware	Hyacinth	33 Cheerful Plaza, Drake Road, Westford	68666	(822) 665-5327		2012-09-18 19:32:05

memid	surname	firstname	address	zipcode	telephone	recommendedby	joindate
35	Hunt	John	5 Bullington Lane, Boston	54333	(899) 720-6978	30	2012-09-19 11:32:45
36	Crumpet	Erica	Crimson Road, North Reading	75655	(811) 732-4816	2	2012-09-22 08:36:38

Answer:

```
delete from cd.members where memid = 37;
```

This exercise is a small increment on our previous one. Instead of deleting all bookings, this time we want to be a bit more targeted, and delete a single member that has never made a booking. To do this, we simply have to add a `WHERE` clause to our command, specifying the member we want to delete. You can see the parallels with `SELECT` and `UPDATE` statements here.

There's one interesting wrinkle here. Try this command out, but substituting in member id 0 instead. This member has made many bookings, and you'll find that the delete fails with an error about a foreign key constraint violation. This is an important concept in relational databases, so let's explore a little further.

Foreign keys are a mechanism for defining relationships between columns of different tables. In our case we use them to specify that the memid column of the bookings table is related to the memid column of the members table. The relationship (or 'constraint') specifies that for a given booking, the member specified in the booking **must** exist in the members table. It's useful to have this guarantee enforced by the database: it means that code using the database can rely on the presence of the member. It's hard (even impossible) to enforce this at higher levels: concurrent operations can interfere and leave your database in a broken state.

PostgreSQL supports various different kinds of constraints that allow you to enforce structure upon your data. For more information on constraints, check out the PostgreSQL documentation on foreign keys

Delete based on a subquery

In our previous exercises, we deleted a specific member who had never made a booking. How can we make that more general, to delete all members who have never made a booking?

Expected results:

memid	surname	firstname	address	zipcode	telephone	recommendedby	joindate
0	GUEST	GUEST	GUEST	0	(000) 000-0000		2012-07-01 00:00:00
1	Smith	Darren	8 Bloomsbury Close, Boston	4321	555-555-5555		2012-07-02 12:02:05
2	Smith	Tracy	8 Bloomsbury Close, New York	4321	555-555-5555		2012-07-02 12:08:23
3	Rownam	Tim	23 Highway Way, Boston	23423	(844) 693-0723		2012-07-03 09:32:15
4	Joplette	Janice	20 Crossing Road, New York	234	(833) 942-4710	1	2012-07-03 10:25:05
5	Butters	Gerald	1065 Huntingdon Avenue, Boston	56754	(844) 078-4130	1	2012-07-09 10:44:09
6	Tracy	Burton	3 Tunisia Drive, Boston	45678	(822) 354-9973		2012-07-15 08:52:55
7	Dare	Nancy	6 Hunting Lodge Way, Boston	10383	(833) 776-4001	4	2012-07-25 08:59:12
8	Boothe	Tim	3 Bloomsbury Close, Reading, 00234	234	(811) 433-2547	3	2012-07-25 16:02:35
9	Stibbons	Ponder	5 Dragons Way, Winchester	87630	(833) 160-3900	6	2012-07-25 17:09:05
10	Owen	Charles	52 Cheshire Grove, Winchester, 28563	28563	(855) 542-5251	1	2012-08-03 19:42:37
11	Jones	David	976 Gnats Close, Reading	33862	(844) 536-8036	4	2012-08-06 16:32:55
12	Baker	Anne	55 Powdery Street, Boston	80743	844-076-5141	9	2012-08-10 14:23:22
13	Farrell	Jemima	103 Firth Avenue, North Reading	57392	(855) 016-0163		2012-08-10 14:28:01

memid	surname	firstname	address	zipcode	telephone	recommendedby	joindate
14	Smith	Jack	252 Binkington Way, Boston	69302	(822) 163-3254	1	2012-08-10 16:22:05
15	Bader	Florence	264 Ursula Drive, Westford	84923	(833) 499-3527	9	2012-08-10 17:52:03
16	Baker	Timothy	329 James Street, Reading	58393	833-941-0824	13	2012-08-15 10:34:25
17	Pinker	David	5 Impreza Road, Boston	65332	811 409-6734	13	2012-08-16 11:32:47
20	Genting	Matthew	4 Nunnington Place, Wingfield, Boston	52365	(811) 972-1377	5	2012-08-19 14:55:55
21	Mackenzie	Anna	64 Perkington Lane, Reading	64577	(822) 661-2898	1	2012-08-26 09:32:05
22	Coplin	Joan	85 Bard Street, Bloomington, Boston	43533	(822) 499-2232	16	2012-08-29 08:32:41
24	Sarwin	Ramnaresh	12 Bullington Lane, Boston	65464	(822) 413-1470	15	2012-09-01 08:44:42
26	Jones	Douglas	976 Gnats Close, Reading	11986	844 536-8036	11	2012-09-02 18:43:05
27	Rumney	Henrietta	3 Burkington Plaza, Boston	78533	(822) 989-8876	20	2012-09-05 08:42:35
28	Farrell	David	437 Granite Farm Road, Westford	43532	(855) 755-9876		2012-09-15 08:22:05
29	Worthington-Smyth	Henry	55 Jagbi Way, North Reading	97676	(855) 894-3758	2	2012-09-17 12:27:15
30	Purview	Millicent	641 Drudgery Close, Burnington, Boston	34232	(855) 941-9786	2	2012-09-18 19:04:01
33	Tupperware	Hyacinth	33 Cheerful Plaza, Drake Road, Westford	68666	(822) 665-5327		2012-09-18 19:32:05

memid	surname	firstname	address	zipcode	telephone	recommendedby	joindate
35	Hunt	John	5 Bullington Lane, Boston	54333	(899) 720-6978	30	2012-09-19 11:32:45
36	Crumpet	Erica	Crimson Road, North Reading	75655	(811) 732-4816	2	2012-09-22 08:36:38

Answer:

```
delete from cd.members where memid not in (select memid from cd.bookings);
```

We can use subqueries to determine whether a row should be deleted or not. There's a couple of standard ways to do this. In our featured answer, the subquery produces a list of all the different member ids in the cd.bookings table. If a row in the table isn't in the list generated by the subquery, it gets deleted.

An alternative is to use a *correlated subquery* . Where our previous example runs a large subquery once, the correlated approach instead specifies a smaller subqueryto run against every row.

```
delete from cd.members mems where not exists (select 1 from cd.bookings where memid
= mems.memid);
```

The two different forms can have different performance characteristics. Under the hood, your database engine is free to transform your query to execute it in a correlated or uncorrelated fashion, though, so things can be a little hard to predict.

Aggregation

Aggregation is one of those capabilities that really make you appreciate the power of relational database systems. It allows you to move beyond merely persisting your data, into the realm of asking truly interesting questions that can be used to inform decision making. This category covers aggregation at length, making use of standard grouping as well as more recent window functions.

If you struggle with these questions, I strongly recommend **Learning SQL** , by Alan Beaulieu and **SQL Cookbook** by Anthony Molinaro. In fact, get the latter anyway - it'll take you beyond anything you find on this site, and on multiple different database systems to boot.

Count the number of facilities

For our first foray into aggregates, we're going to stick to something simple. We want to know how many facilities exist - simply produce a total count.

Expected results:

count
9

Answer:

```
select count(*) from cd.facilities;
```

Aggregation starts out pretty simply! The SQL above selects everything from our facilities table, and then counts the number of rows in the result set. The count function has a variety of uses:

- `COUNT(*)` simply returns the number of rows
- `COUNT(address)` counts the number of non-null addresses in the result set.
- Finally, `COUNT(DISTINCT address)` counts the number of *different* addresses in the facilities table.

The basic idea of an aggregate function is that it takes in a column of data, performs some function upon it, and outputs a *scalar* (single) value. There are a bunch more aggregation functions, including `MAX`, `MIN`, `SUM`, and `AVG`. These all do pretty much what you'd expect from their names :-).

One aspect of aggregate functions that people often find confusing is in queries like the below:

```
select facid, count(*) from cd.facilities
```

Try it out, and you'll find that it doesn't work. This is because count(*) wants to collapse the facilities table into a single value - unfortunately, it can't do that, because there's a lot of different facids in cd.facilities - Postgres doesn't know which facid to pair the count with.

Instead, if you wanted a query that returns all the facids along with a count on each row, you can break the aggregation out into a subquery as below:

```
select facid,
    (select count(*) from cd.facilities)
    from cd.facilities
```

When we have a subquery that returns a scalar value like this, Postgres knows to simply repeat the value for every row in cd.facilities.

Count the number of expensive facilities

Produce a count of the number of facilities that have a cost to guests of 10 or more.

count
6

Answer:

```
select count(*) from cd.facilities where guestcost >= 10;
```

This one is only a simple modification to the previous question: we need to weed out the inexpensive facilities. This is easy to do using a `WHERE` clause. Our aggregation can now only see the expensive facilities.

Count the number of recommendations each member makes

Produce a count of the number of recommendations each member has made. Order by member ID.

Expected results:

recommendedby	count
1	5
2	3
3	1
4	2
5	1
6	1
9	2
11	1
13	2
15	1
16	1
20	1
30	1

Answer:

```
select recommendedby, count(*)
    from cd.members
    where recommendedby is not null
    group by recommendedby
order by recommendedby;
```

Previously, we've seen that aggregation functions are applied to a column of values, and convert them into an aggregated scalar value. This is useful, but we often find that we don't want just a single aggregated result: for example, instead of knowing the total amount of money the club has made this month, I might want to know how much money each different facility has made, or which times of day were most lucrative.

In order to support this kind of behaviour, SQL has the `GROUP BY` construct. What this does is batch the data together into groups, and run the aggregation function separately for each group. When you specify a `GROUP BY`, the database produces an aggregated value for each distinct value in the supplied columns. In this case, we're saying 'for each distinct value of recommendedby, get me the number of times that value appears'.

List the total slots booked per facility

Produce a list of the total number of slots booked per facility. For now, just produce an output table consisting of facility id and slots, sorted by facility id.

Expected results:

facid	Total Slots
0	1320
1	1278
2	1209
3	830
4	1404
5	228
6	1104
7	908
8	911

Answer:

```
select facid, sum(slots) as "Total Slots"
    from cd.bookings
    group by facid
order by facid;
```

Other than the fact that we've introduced the `SUM` aggregate function, there's not a great deal to say about this exercise. For each distinct facility id, the `SUM` function adds together everything in the slots column.

List the total slots booked per facility in a given month

Produce a list of the total number of slots booked per facility in the month of September 2012. Produce an output table consisting of facility id and slots, sorted by the number of slots.

Expected results:

facid	Total Slots
5	122
3	422
7	426
8	471
6	540
2	570
1	588
0	591
4	648

Answer:

```
select facid, sum(slots) as "Total Slots"
    from cd.bookings
    where
        starttime >= '2012-09-01'
        and starttime < '2012-10-01'
    group by facid
order by sum(slots);
```

This is only a minor alteration of our previous example. Remember that aggregation happens after the `WHERE` clause is evaluated: we thus use the `WHERE` to restrict the data we aggregate over, and our aggregation only sees data from a single month.

List the total slots booked per facility per month

Produce a list of the total number of slots booked per facility per month in the year of 2012. Produce an output table consisting of facility id and slots, sorted by the id and month.

Expected results:

facid	month	Total Slots
0	7	270
0	8	459
0	9	591
1	7	207
1	8	483
1	9	588
2	7	180
2	8	459
2	9	570
3	7	104
3	8	304
3	9	422
4	7	264
4	8	492
4	9	648
5	7	24
5	8	82
5	9	122
6	7	164
6	8	400
6	9	540
7	7	156
7	8	326
7	9	426
8	7	117
8	8	322
8	9	471

Answer:

```
select facid, extract(month from starttime) as month, sum(slots) as "Total Slots"
    from cd.bookings
    where
        starttime >= '2012-01-01'
        and starttime < '2013-01-01'
    group by facid, month
order by facid, month;
```

The main piece of new functionality in this question is the `EXTRACT` function. `EXTRACT` allows you to get individual components of a timestamp, like day, month, year, etc. We group by the output of this function to provide per-month values. An alternative, if we needed to distinguish between the same month in different years, is to make use of the `DATE_TRUNC` function, which truncates a date to a given granularity.

It's also worth noting that this is the first time we've truly made use of the ability to group by more than one column.

Find the count of members who have made at least one booking

Find the total number of members who have made at least one booking.

Expected results:

count
30

Answer:

```
select count(distinct memid) from cd.bookings
```

Your first instinct may be to go for a subquery here. Something like the below:

```
select count(*) from
    (select distinct memid from cd.bookings) as mems
```

This does work perfectly well, but we can simplify a touch with the help of a little extra knowledge in the form of `COUNT DISTINCT`. This does what you might expect, counting the distinct values in the passed column.

List facilities with more than 1000 slots booked

Produce a list of facilities with more than 1000 slots booked. Produce an output table consisting of facility id and hours, sorted by facility id.

Expected results:

facid	Total Slots
0	1320
1	1278
2	1209
4	1404
6	1104

Answer:

```
select facid, sum(slots) as "Total Slots"
        from cd.bookings
        group by facid
        having sum(slots) > 1000
        order by facid
```

It turns out that there's actually an SQL keyword designed to help with the filtering of output from aggregate functions. This keyword is `HAVING`.

The behaviour of `HAVING` is easily confused with that of `WHERE`. The best way to think about it is that in the context of a query with an aggregate function, `WHERE` is used to filter what data gets input into the aggregate function, while `HAVING` is used to filter the data once it is output from the function. Try experimenting to explore this difference!

Find the total revenue of each facility

Produce a list of facilities along with their total revenue. The output table should consist of facility name and revenue, sorted by revenue. Remember that there's a different cost for guests and members!

Expected results:

name	revenue
Table Tennis	180
Snooker Table	240
Pool Table	270
Badminton Court	1906.5
Squash Court	13468.0
Tennis Court 1	13860
Tennis Court 2	14310
Massage Room 2	15810
Massage Room 1	72540

Answer:

```
select facs.name, sum(slots * case
            when memid = 0 then facs.guestcost
            else facs.membercost
        end) as revenue
    from cd.bookings bks
    inner join cd.facilities facs
        on bks.facid = facs.facid
    group by facs.name
order by revenue;
```

The only real complexity in this query is that guests (member ID 0) have a different cost to everyone else. We use a case statement to produce the cost for each session, and then sum each of those sessions, grouped by facility.

Find facilities with a total revenue less than 1000

Produce a list of facilities with a total revenue less than 1000. Produce an output table consisting of facility name and revenue, sorted by revenue. Remember that there's a different cost for guests and members!

Expected results:

name	revenue
Table Tennis	180
Snooker Table	240
Pool Table	270

Answer:

```
select name, revenue from (
    select facs.name, sum(case
                when memid = 0 then slots * facs.guestcost
                else slots * membercost
            end) as revenue
        from cd.bookings bks
        inner join cd.facilities facs
            on bks.facid = facs.facid
        group by facs.name
    ) as agg where revenue < 1000
order by revenue;
```

You may well have tried to use the `HAVING` keyword we introduced in an earlier exercise, producing something like below:

```
select facs.name, sum(case
        when memid = 0 then slots * facs.guestcost
        else slots * membercost
    end) as revenue
    from cd.bookings bks
    inner join cd.facilities facs
        on bks.facid = facs.facid
    group by facs.name
    having revenue < 1000
order by revenue;
```

Unfortunately, this doesn't work! You'll get an error along the lines of `ERROR: column "revenue" does not exist`. Postgres, unlike some other RDBMSs like SQL Server and MySQL, doesn't support putting column names in the `HAVING` clause. This means that for this query to work, you'd have to produce something like below:

```
select facs.name, sum(case
        when memid = 0 then slots * facs.guestcost
        else slots * membercost
    end) as revenue
    from cd.bookings bks
    inner join cd.facilities facs
        on bks.facid = facs.facid
    group by facs.name
    having sum(case
        when memid = 0 then slots * facs.guestcost
        else slots * membercost
    end) < 1000
order by revenue;
```

Having to repeat significant calculation code like this is messy, so our anointed solution instead just wraps the main query body as a subquery, and selects from it using a `WHERE` clause. In general, I recommend using `HAVING` for simple queries, as it increases clarity. Otherwise, this subquery approach is often easier to use.

Output the facility id that has the highest number of slots booked

Output the facility id that has the highest number of slots booked. For bonus points, try a version without a LIMIT clause. This version will probably look messy!

Expected results:

facid	Total Slots
4	1404

Answer:

```
select facid, sum(slots) as "Total Slots"
    from cd.bookings
    group by facid
order by sum(slots) desc
LIMIT 1;
```

Let's start off with what's arguably the simplest way to do this: produce a list of facility IDs and the total number of slots used, order by the total number of slots used, and pick only the top result.

It's worth realising, though, that this method has a significant weakness. In the event of a tie, we will still only get one result! To get all the relevant results, we might try using the `MAX` aggregate function, something like below:

```
select facid, max(totalslots) from (
    select facid, sum(slots) as totalslots
        from cd.bookings
        group by facid
    ) as sub group by facid
```

The intent of this query is to get the highest totalslots value and its associated facid(s). Unfortunately, this just won't work! In the event of multiple facids having the same number of slots booked, it would be ambiguous which facid should be paired up with the single (or *scalar*) value coming out of the `MAX` function. This means that Postgres will tell you that facid ought to be in a `GROUP BY` section, which won't produce the results we're looking for.

Let's take a first stab at a working query:

```
select facid, sum(slots) as totalslots
    from cd.bookings
    group by facid
    having sum(slots) = (select max(sum2.totalslots) from
        (select sum(slots) as totalslots
        from cd.bookings
        group by facid
        ) as sum2);
```

The query produces a list of facility IDs and number of slots used, and then uses a HAVING clause that works out the maximum totalslots value. We're essentially saying: 'produce a list of facids and their number of slots booked, and filter out all the ones that doen't have a number of slots booked equal to the maximum.'

Useful as `HAVING` is, however, our query is pretty ugly. To improve on that, let's introduce another new concept: **Common Table Expressions** (CTEs). CTEs can be thought of as allowing you to define a database view inline in your query. It's really helpful in situations like this, where you're having to repeat yourself a lot.

CTEs are declared in the form `WITH CTEName as (SQL-Expression)`. You can see our query redefined to use a CTE below:

```
with sum as (select facid, sum(slots) as totalslots
    from cd.bookings
    group by facid
)
select facid, totalslots
    from sum
    where totalslots = (select max(totalslots) from sum);
```

You can see that we've factored out our repeated selections from cd.bookings into a single CTE, and made the query a lot simpler to read in the process!

BUT WAIT. There's more. It's also possible to complete this problem using Window Functions. We'll leave these until later, but even better solutions to problems like these are available.

That's a lot of information for a single exercise. Don't worry too much if you don't get it all right now - we'll reuse these concepts in later exercises.

List the total slots booked per facility per month, Part 2

Produce a list of the total number of slots booked per facility per month in the year of 2012. In this version, include output rows containing totals for all months per facility, and a total for all months for all facilities. The output table should consist of facility id, month and slots, sorted by the id and month. When calculating the aggregated values for all months and all facids, return null values in the month and facid columns.

Expected results:

facid	month	slots
0	7	270
0	8	459
0	9	591
0		1320
1	7	207
1	8	483
1	9	588
1		1278
2	7	180
2	8	459
2	9	570
2		1209
3	7	104
3	8	304
3	9	422
3		830
4	7	264
4	8	492
4	9	648
4		1404
5	7	24
5	8	82
5	9	122
5		228
6	7	164
6	8	400
6	9	540
6		1104
7	7	156
7	8	326

facid	month	slots
7	9	426
7		908
8	7	117
8	8	322
8	9	471
8		910
		9191

Answer:

```
select facid, extract(month from starttime) as month, sum(slots) as slots
    from cd.bookings
    where
        starttime >= '2012-01-01'
        and starttime < '2013-01-01'
    group by rollup(facid, month)
order by facid, month;
```

When we are doing data analysis, we sometimes want to perform multiple levels of aggregation to allow ourselves to 'zoom' in and out to different depths. In this case, we might be looking at each facility's overall usage, but then want to dive in to see how they've performed on a per-month basis. Using the SQL we know so far, it's quite cumbersome to produce a single query that does what we want - we effectively have to resort to concatenating multiple queries using `UNION ALL`:

```
select facid, extract(month from starttime) as month, sum(slots) as slots
    from cd.bookings
    where
        starttime >= '2012-01-01'
        and starttime < '2013-01-01'
    group by facid, month
union all
select facid, null, sum(slots) as slots
    from cd.bookings
    where
        starttime >= '2012-01-01'
        and starttime < '2013-01-01'
    group by facid
union all
select null, null, sum(slots) as slots
    from cd.bookings
    where
        starttime >= '2012-01-01'
        and starttime < '2013-01-01'
order by facid, month;
```

As you can see, each subquery performs a different level of aggregation, and we just combine the results. We can clean this up a lot by factoring out commonalities using a CTE:

```
with bookings as (
    select facid, extract(month from starttime) as month, slots
    from cd.bookings
    where
        starttime ⩾ '2012-01-01'
        and starttime < '2013-01-01'
)
select facid, month, sum(slots) from bookings group by facid, month
union all
select facid, null, sum(slots) from bookings group by facid
union all
select null, null, sum(slots) from bookings
order by facid, month;
```

This version is not excessively hard on the eyes, but it becomes cumbersome as the number of aggregation columns increases. Fortunately, PostgreSQL 9.5 introduced support for the `ROLLUP` operator, which we've used to simplify our accepted answer.

`ROLLUP` produces a hierarchy of aggregations in the order passed into it: for example, `ROLLUP(facid, month)` outputs aggregations on (facid, month), (facid), and (). If we wanted an aggregation of all facilities for a month (instead of all months for a facility) we'd have to reverse the order, using `ROLLUP(month, facid)`. Alternatively, if we instead want all possible permutations of the columns we pass in, we can use CUBE rather than `ROLLUP`. This will produce (facid, month), (month), (facid), and ().

`ROLLUP` and `CUBE` are special cases of `GROUPING SETS`. `GROUPING SETS` allow you to specify the exact aggregation permutations you want: you could, for example, ask for just (facid, month) and (facid), skipping the top-level aggregation.

List the total hours booked per named facility

Produce a list of the total number of *hours* booked per facility, remembering that a slot lasts half an hour. The output table should consist of the facility id, name, and hours booked, sorted by facility id. Try formatting the hours to two decimal places.

Expected results:

facid	name	Total Hours
0	Tennis Court 1	660.00
1	Tennis Court 2	639.00
2	Badminton Court	604.50
3	Table Tennis	415.00
4	Massage Room 1	702.00
5	Massage Room 2	114.00
6	Squash Court	552.00
7	Snooker Table	454.00
8	Pool Table	455.50

Answer:

```
select facs.facid, facs.name,
    trim(to_char(sum(bks.slots)/2.0, '9999999999999999D99')) as "Total Hours"

    from cd.bookings bks
    inner join cd.facilities facs
        on facs.facid = bks.facid
    group by facs.facid, facs.name
order by facs.facid;
```

There's a few little pieces of interest in this question. Firstly, you can see that our aggregation works just fine when we join to another table on a 1:1 basis. Also note that we group by both `facs.facid` and `facs.name`. This is might seem odd: after all, since `facid` is the primary key of the facilities table, each `facid` has exactly one name, and grouping by both fields is the same as grouping by facid alone. In fact, you'll find that if you remove `facs.name` from the `GROUP BY` clause, the query works just fine: Postgres works out that this 1:1 mapping exists, and doesn't insist that we group by both columns.

Unfortunately, depending on which database system we use, validation might not be so smart, and may not realise that the mapping is strictly 1:1. That being the case, if there were multiple `names` for each `facid` and we hadn't grouped by `name`, the DBMS would have to choose between multiple (equally valid) choices for the `name`. Since this is invalid, the database system will insist that we group by both fields. In general, I recommend grouping by all columns you don't have an aggregate function on: this will ensure better cross-platform compatibility.

Next up is the division. Those of you familiar with MySQL may be aware that integer divisions are automatically cast to floats. Postgres is a little more traditional in this respect, and expects you to tell it if you want a floating point division. You can do that easily in this case by dividing by 2.0 rather than 2.

Finally, let's take a look at formatting. The `TO_CHAR` function converts values to character strings. It takes a formatting string, which we specify as (up to) lots of numbers before the decimal place, decimal place, and two numbers after the decimal place. The output of this function can be prepended with a space, which is why we include the outer `TRIM` function.

List each member's first booking after September 1st 2012

Produce a list of each member name, id, and their first booking after September 1st 2012. Order by member ID.

Expected results:

surname	firstname	memid	starttime
GUEST	GUEST	0	2012-09-01 08:00:00
Smith	Darren	1	2012-09-01 09:00:00
Smith	Tracy	2	2012-09-01 11:30:00
Rownam	Tim	3	2012-09-01 16:00:00
Joplette	Janice	4	2012-09-01 15:00:00
Butters	Gerald	5	2012-09-02 12:30:00
Tracy	Burton	6	2012-09-01 15:00:00
Dare	Nancy	7	2012-09-01 12:30:00
Boothe	Tim	8	2012-09-01 08:30:00
Stibbons	Ponder	9	2012-09-01 11:00:00
Owen	Charles	10	2012-09-01 11:00:00
Jones	David	11	2012-09-01 09:30:00
Baker	Anne	12	2012-09-01 14:30:00
Farrell	Jemima	13	2012-09-01 09:30:00
Smith	Jack	14	2012-09-01 11:00:00
Bader	Florence	15	2012-09-01 10:30:00
Baker	Timothy	16	2012-09-01 15:00:00
Pinker	David	17	2012-09-01 08:30:00
Genting	Matthew	20	2012-09-01 18:00:00
Mackenzie	Anna	21	2012-09-01 08:30:00
Coplin	Joan	22	2012-09-02 11:30:00
Sarwin	Ramnaresh	24	2012-09-04 11:00:00
Jones	Douglas	26	2012-09-08 13:00:00
Rumney	Henrietta	27	2012-09-16 13:30:00
Farrell	David	28	2012-09-18 09:00:00
Worthington-Smyth	Henry	29	2012-09-19 09:30:00
Purview	Millicent	30	2012-09-19 11:30:00
Tupperware	Hyacinth	33	2012-09-20 08:00:00
Hunt	John	35	2012-09-23 14:00:00
Crumpet	Erica	36	2012-09-27 11:30:00

Answer:

```
select mems.surname, mems.firstname, mems.memid, min(bks.starttime) as starttime
    from cd.bookings bks
    inner join cd.members mems on
        mems.memid = bks.memid
    where starttime >= '2012-09-01'
    group by mems.surname, mems.firstname, mems.memid
order by mems.memid;
```

This answer demonstrates the use of aggregate functions on dates. `MIN` works exactly as you'd expect, pulling out the lowest possible date in the result set. To make this work, we need to ensure that the result set only contains dates from September onwards. We do this using the `WHERE` clause.

You might typically use a query like this to find a customer's next booking. You can use this by replacing the date '2012-09-01' with the function `now()`

Produce a list of member names, with each row containing the total member count

Produce a list of member names, with each row containing the total member count. Order by join date.

Expected results:

count	firstname	surname
31	GUEST	GUEST
31	Darren	Smith
31	Tracy	Smith
31	Tim	Rownam
31	Janice	Joplette
31	Gerald	Butters
31	Burton	Tracy
31	Nancy	Dare
31	Tim	Boothe
31	Ponder	Stibbons
31	Charles	Owen
31	David	Jones
31	Anne	Baker
31	Jemima	Farrell
31	Jack	Smith
31	Florence	Bader
31	Timothy	Baker
31	David	Pinker
31	Matthew	Genting
31	Anna	Mackenzie
31	Joan	Coplin
31	Ramnaresh	Sarwin
31	Douglas	Jones
31	Henrietta	Rumney
31	David	Farrell
31	Henry	Worthington-Smyth
31	Millicent	Purview
31	Hyacinth	Tupperware
31	John	Hunt
31	Erica	Crumpet

count	firstname	surname
31	Darren	Smith

Answer:

```
select count(*) over(), firstname, surname
    from cd.members
order by joindate
```

Using the knowledge we've built up so far, the most obvious answer to this is below. We use a subquery because otherwise SQL will require us to group by firstname and surname, producing a different result to what we're looking for.

```
select (select count(*) from cd.members) as count, firstname, surname
    from cd.members
order by joindate
```

There's nothing at all wrong with this answer, but we've chosen a different approach to introduce a new concept called window functions. Window functions provide enormously powerful capabilities, in a form often more convenient than the standard aggregation functions. While this exercise is only a toy, we'll be working on more complicated examples in the near future.

Window functions operate on the result set of your (sub-)query, after the `WHERE` clause and all standard aggregation. They operate on a *window* of data. By default this is unrestricted: the entire result set, but it can be restricted to provide more useful results. For example, suppose instead of wanting the count of all members, we want the count of all members who joined in the same month as that member:

```
select count(*) over(partition by date_trunc('month',joindate)),
    firstname, surname
    from cd.members
order by joindate
```

In this example, we partition the data by month. For each row the window function operates over, the window is any rows that have a joindate in the same month. The window function thus produces a count of the number of members who joined in that month.

You can go further. Imagine if, instead of the total number of members who joined that month, you want to know what number joinee they were that month. You can do this by adding in an `ORDER BY` to the window function:

```
select count(*) over(partition by date_trunc('month',joindate) order by joindate),
    firstname, surname
    from cd.members
order by joindate
```

The `ORDER BY` changes the window again. Instead of the window for each row being the entire partition, the window goes from the start of the partition to the current row, and not beyond. Thus, for the first member who joins in a given month, the count is 1. For the second, the count is 2, and so on.

One final thing that's worth mentioning about window functions: you can have multiple unrelated ones in the same query. Try out the query below for an example - you'll see the numbers for the members going in opposite directions! This flexibility can lead to more concise, readable, and maintainable queries.

```
select count(*) over(partition by date_trunc('month',joindate) order by joindate
asc),
    count(*) over(partition by date_trunc('month',joindate) order by joindate
desc),
    firstname, surname
    from cd.members
order by joindate
```

Window functions are extraordinarily powerful, and they will change the way you write and think about SQL. Make good use of them!

Produce a numbered list of members

Produce a monotonically increasing numbered list of members, ordered by their date of joining. Remember that member IDs are not guaranteed to be sequential.

Expected results:

row_number	firstname	surname
1	GUEST	GUEST
2	Darren	Smith
3	Tracy	Smith
4	Tim	Rownam
5	Janice	Joplette
6	Gerald	Butters
7	Burton	Tracy
8	Nancy	Dare
9	Tim	Boothe
10	Ponder	Stibbons
11	Charles	Owen
12	David	Jones
13	Anne	Baker
14	Jemima	Farrell
15	Jack	Smith
16	Florence	Bader
17	Timothy	Baker
18	David	Pinker
19	Matthew	Genting
20	Anna	Mackenzie
21	Joan	Coplin
22	Ramnaresh	Sarwin
23	Douglas	Jones
24	Henrietta	Rumney
25	David	Farrell
26	Henry	Worthington-Smyth
27	Millicent	Purview
28	Hyacinth	Tupperware
29	John	Hunt
30	Erica	Crumpet

row_number	firstname	surname
31	Darren	Smith

Answer:

```
select row_number() over(order by joindate), firstname, surname
    from cd.members
order by joindate
```

This exercise is a simple bit of window function practise! You could just as easily `use count(*) over(order by joindate)` here, so don't worry if you used that instead.

In this query, we don't define a partition, meaning that the partition is the entire dataset. Since we define an order for the window function, for any given row the window is: start of the dataset -> current row.

Output the facility id that has the highest number of slots booked, again

Output the facility id that has the highest number of slots booked. Ensure that in the event of a tie, all tieing results get output.

Expected results:

facid	total
4	1404

Answer:

```
select facid, total from (
    select facid, sum(slots) total, rank() over (order by sum(slots) desc) rank
            from cd.bookings
        group by facid
    ) as ranked
    where rank = 1
```

You may recall that this is a problem we've already solved in an earlier exercise. We came up with an answer something like below, which we then cut down using CTEs:

```
select facid, sum(slots) as totalslots
    from cd.bookings
    group by facid
    having sum(slots) = (select max(sum2.totalslots) from
        (select sum(slots) as totalslots
        from cd.bookings
        group by facid
        ) as sum2);
```

Once we've cleaned it up, this solution is perfectly adequate. Explaining how the query works makes it seem a little odd, though - 'find the number of slots booked by the best facility. Calculate the total slots booked for each facility, and return only the rows where the slots booked are the same as for the best'. Wouldn't it be nicer to be able to say 'calculate the number of slots booked for each facility, rank them, and pick out any at rank 1'?

Fortunately, window functions allow us to do this - although it's fair to say that doing so is not trivial to the untrained eye. The first key piece of information is the existence of the éfunction. This ranks values based on the `ORDER BY` that is passed to it. If there's a tie for (say) second place), the next gets ranked at position 4. So, what we need to do is get the number of slots for each facility, rank them, and pick off the ones at the top rank. A first pass at this might look something like the below:

```
select facid, total from (
    select facid, total, rank() over (order by total desc) rank from (
        select facid, sum(slots) total
            from cd.bookings
            group by facid
        ) as sumslots
    ) as ranked
where rank = 1
```

The inner query calculates the total slots booked, the middle one ranks them, and the outer one creams off the top ranked. We can actually tidy this up a little: recall that window function get applied pretty late in the select function, after aggregation. That being the case, we can move the aggregation into the `ORDER BY` part of the function, as shown in the approved answer.

While the window function approach isn't massively simpler in terms of lines of code, it arguably makes more semantic sense.

Rank members by (rounded) hours used

Produce a list of members, along with the number of hours they've booked in facilities, rounded to the nearest ten hours. Rank them by this rounded figure, producing output of first name, surname, rounded hours, rank. Sort by rank, surname, and first name.

Expected results:

firstname	surname	hours	rank
GUEST	GUEST	1200	1
Darren	Smith	340	2
Tim	Rownam	330	3
Tim	Boothe	220	4
Tracy	Smith	220	4
Gerald	Butters	210	6
Burton	Tracy	180	7
Charles	Owen	170	8
Janice	Joplette	160	9
Anne	Baker	150	10
Timothy	Baker	150	10
David	Jones	150	10
Nancy	Dare	130	13
Florence	Bader	120	14
Anna	Mackenzie	120	14
Ponder	Stibbons	120	14
Jack	Smith	110	17
Jemima	Farrell	90	18
David	Pinker	80	19
Ramnaresh	Sarwin	80	19
Matthew	Genting	70	21
Joan	Coplin	50	22
David	Farrell	30	23
Henry	Worthington-Smyth	30	23
John	Hunt	20	25
Douglas	Jones	20	25
Millicent	Purview	20	25
Henrietta	Rumney	20	25
Erica	Crumpet	10	29
Hyacinth	Tupperware	10	29

Answer:

```
select firstname, surname,
    ((sum(bks.slots)+10)/20)*10 as hours,
    rank() over (order by ((sum(bks.slots)+10)/20)*10 desc) as rank

    from cd.bookings bks
    inner join cd.members mems
        on bks.memid = mems.memid
    group by mems.memid
order by rank, surname, firstname;
```

This answer isn't a great stretch over our previous exercise, although it does illustrate the function of `RANK` better. You can see that some of the clubgoers have an equal rounded number of hours booked in, and their rank is the same. If position 2 is shared between two members, the next one along gets position 4. There's a different function, `DENSE_RANK`, that would assign that member position 3 instead.

It's worth noting the technique we use to do rounding here. Adding 5, dividing by 10, and multiplying by 10 has the effect (thanks to integer arithmetic cutting off fractions) of rounding a number to the nearest 10. In our case, because slots are half an hour, we need to add 10, divide by 20, and multiply by 10. One could certainly make the argument that we should do the slots -> hours conversion independently of the rounding, which would increase clarity.

Talking of clarity, this rounding malarky is starting to introduce a noticeable amount of code repetition. At this point it's a judgement call, but you may wish to factor it out using a subquery as below:

```
select firstname, surname, hours, rank() over (order by hours desc) from
    (select firstname, surname,
        ((sum(bks.slots)+10)/20)*10 as hours

        from cd.bookings bks
        inner join cd.members mems
            on bks.memid = mems.memid
        group by mems.memid
    ) as subq
order by rank, surname, firstname;
```

Find the top three revenue generating facilities

Produce a list of the top three revenue generating facilities (including ties). Output facility name and rank, sorted by rank and facility name.

Expected results:

name	rank
Massage Room 1	1
Massage Room 2	2
Tennis Court 2	3

Answer:

```
select name, rank from (
    select facs.name as name, rank() over (order by sum(case
                when memid = 0 then slots * facs.guestcost
                else slots * membercost
            end) desc) as rank
        from cd.bookings bks
        inner join cd.facilities facs
            on bks.facid = facs.facid
        group by facs.name
    ) as subq
    where rank <= 3
order by rank;
```

This question doesn't introduce any new concepts, and is just intended to give you the opportunity to practise what you already know. We use the `CASE` statement to calculate the revenue for each slot, and aggregate that on a per-facility basis using `SUM`. We then use the `RANK` window function to produce a ranking, wrap it all up in a subquery, and extract everything with a rank less than or equal to 3.

Classify facilities by value

Classify facilities into equally sized groups of high, average, and low based on their revenue. Order by classification and facility name.

Expected results:

name	revenue
Massage Room 1	high
Massage Room 2	high
Tennis Court 2	high
Badminton Court	average
Squash Court	average
Tennis Court 1	average
Pool Table	low
Snooker Table	low
Table Tennis	low

Answer:

```
select name, case when class=1 then 'high'
        when class=2 then 'average'
        else 'low'
        end revenue
    from (
        select facs.name as name, ntile(3) over (order by sum(case
                when memid = 0 then slots * facs.guestcost
                else slots * membercost
            end) desc) as class
        from cd.bookings bks
        inner join cd.facilities facs
            on bks.facid = facs.facid
        group by facs.name
    ) as subq
order by class, name;
```

This exercise should mostly use familiar concepts, although we do introduce the `NTILE` window function. `NTILE` groups values into a passed-in number of groups, as evenly as possible. It outputs a number from 1->number of groups. We then use a `CASE` statement to turn that number into a label!

Calculate the payback time for each facility

Based on the 3 complete months of data so far, calculate the amount of time each facility will take to repay its cost of ownership. Remember to take into account ongoing monthly maintenance. Output facility name and payback time in months, order by facility name. Don't worry about differences in month lengths, we're only looking for a rough value here!

Expected results:

name	months
Badminton Court	6.8317677198975235
Massage Room 1	0.18885741265344664778
Massage Room 2	1.7621145374449339
Pool Table	5.3333333333333333
Snooker Table	6.9230769230769231
Squash Court	1.1339582703356516
Table Tennis	6.4000000000000000
Tennis Court 1	2.2624434389140271
Tennis Court 2	1.7505470459518600

Answer:

```
select  facs.name as name,
    facs.initialoutlay/((sum(case
            when memid = 0 then slots * facs.guestcost
            else slots * membercost
        end)/3) - facs.monthlymaintenance) as months
    from cd.bookings bks
    inner join cd.facilities facs
        on bks.facid = facs.facid
    group by facs.facid
order by name;
```

In contrast to all our recent exercises, there's no need to use window functions to solve this problem: it's just a bit of maths involving monthly revenue, initial outlay, and monthly maintenance. Again, for production code you might want to clarify what's going on a little here using a subquery (although since we've hard-coded the number of months, putting this into production is unlikely!). A tidied-up version might look like:

```
select  name,
    initialoutlay / (monthlyrevenue - monthlymaintenance) as repaytime
    from
        (select facs.name as name,
            facs.initialoutlay as initialoutlay,
            facs.monthlymaintenance as monthlymaintenance,
            sum(case
                when memid = 0 then slots * facs.guestcost
                else slots * membercost
            end)/3 as monthlyrevenue
        from cd.bookings bks
        inner join cd.facilities facs
            on bks.facid = facs.facid
        group by facs.facid
    ) as subq
order by name;
```

But, I hear you ask, what would an automatic version of this look like? One that didn't need to have a hard-coded number of months in it? That's a little more complicated, and involves some date arithmetic. I've factored that out into a CTE to make it a little more clear.

```
with monthdata as (
    select  mincompletemonth,
        maxcompletemonth,
        (extract(year from maxcompletemonth)*12) +
            extract(month from maxcompletemonth) -
            (extract(year from mincompletemonth)*12) -
            extract(month from mincompletemonth) as nummonths
    from (
        select  date_trunc('month',
                (select max(starttime) from cd.bookings)) as maxcompletemonth,
            date_trunc('month',
                (select min(starttime) from cd.bookings)) as mincompletemonth
    ) as subq
)
select  name,
    initialoutlay / (monthlyrevenue - monthlymaintenance) as repaytime

    from
        (select facs.name as name,
            facs.initialoutlay as initialoutlay,
            facs.monthlymaintenance as monthlymaintenance,
            sum(case
                when memid = 0 then slots * facs.guestcost
                else slots * membercost
            end)/(select nummonths from monthdata) as monthlyrevenue

            from cd.bookings bks
            inner join cd.facilities facs
                on bks.facid = facs.facid
            where bks.starttime < (select maxcompletemonth from monthdata)
```

```
              group by facs.facid
          ) as subq
order by name;
```

This code restricts the data that goes in to complete months. It does this by selecting the maximum date, rounding down to the month, and stripping out all dates larger than that. Even this code is not completely-complete. It doesn't handle the case of a facility making a loss. Fixing that is not too hard, and is left as (another) exercise for the reader!

Calculate a rolling average of total revenue

For each day in August 2012, calculate a rolling average of total revenue over the previous 15 days. Output should contain date and revenue columns, sorted by the date. Remember to account for the possibility of a day having zero revenue. This one's a bit tough, so don't be afraid to check out the hint!

Expected results:

date	revenue
2012-08-01	1126.8333333333333333
2012-08-02	1153.0000000000000000
2012-08-03	1162.9000000000000000
2012-08-04	1177.3666666666666667
2012-08-05	1160.9333333333333333
2012-08-06	1185.4000000000000000
2012-08-07	1182.8666666666666667
2012-08-08	1172.6000000000000000
2012-08-09	1152.4666666666666667
2012-08-10	1175.0333333333333333
2012-08-11	1176.6333333333333333
2012-08-12	1195.6666666666666667
2012-08-13	1218.0000000000000000
2012-08-14	1247.4666666666666667
2012-08-15	1274.1000000000000000
2012-08-16	1281.2333333333333333
2012-08-17	1324.4666666666666667
2012-08-18	1373.7333333333333333
2012-08-19	1406.0666666666666667
2012-08-20	1427.0666666666666667
2012-08-21	1450.3333333333333333
2012-08-22	1539.7000000000000000
2012-08-23	1567.3000000000000000
2012-08-24	1592.3333333333333333
2012-08-25	1615.0333333333333333
2012-08-26	1631.2000000000000000
2012-08-27	1659.4333333333333333
2012-08-28	1687.0000000000000000
2012-08-29	1684.6333333333333333
2012-08-30	1657.9333333333333333

date	revenue
2012-08-31	1703.4000000000000000

Answer:

```
select  dategen.date,
    (
        -- correlated subquery that, for each day fed into it,
        -- finds the average revenue for the last 15 days
        select sum(case
            when memid = 0 then slots * facs.guestcost
            else slots * membercost
        end) as rev

        from cd.bookings bks
        inner join cd.facilities facs
            on bks.facid = facs.facid
        where bks.starttime > dategen.date - interval '14 days'
            and bks.starttime < dategen.date + interval '1 day'
    )/15 as revenue
    from
    (
        -- generates a list of days in august
        select  cast(generate_series(timestamp '2012-08-01',
            '2012-08-31','1 day') as date) as date
    )  as dategen
order by dategen.date;
```

There's at least two equally good solutions to this question. I've put the simplest to write as the answer, but there's also a more flexible solution that uses window functions.

Let's look at the selected answer first. When I read SQL queries, I tend to read the `SELECT` part of the statement last - the `FROM` and `WHERE` parts tend to be more interesting. So, what do we have in our `FROM`? A call to the `GENERATE_SERIES` function. This does pretty much what it says on the tin - generates a series of values. You can specify a start value, a stop value, and an increment. It works for integer types and dates - although, as you can see, we need to be explicit about what types are going into and out of the function. Try removing the casts, and seeing the result!

So, we've generated a timestamp for each day in August. Now, for each day, we need to generate our average. We can do this using a *correlated subquery* . If you remember, a correlated subquery is a subquery that uses values from the outer query. This means that it gets executed once for each result row in the outer query. This is in contrast to an uncorrelated subquery, which only has to be executed once.

If we look at our correlated subquery, we can see that it's correlated on the dategen.date field. It produces a sum of revenue for this day and the 14 days prior to it, and then divides that sum by 15. This produces the output we're looking for!

I mentioned that there's a window function-based solution for this problem as well - you can see it below. The approach we use for this is generating a list of revenue for each day, and then using window function aggregation over that list. The nice thing about this method is that once you have the per-day revenue, you can produce a wide range of results quite easily - you might, for example, want rolling averages for the previous month, 15 days, and 5 days. This is easy to do using this method, and rather harder using conventional aggregation.

```
select date, avgrev from (
    -- AVG over this row and the 14 rows before it.
    select  dategen.date as date,
        avg(revdata.rev) over(order by dategen.date rows 14 preceding) as avgrev
    from
        -- generate a list of days.  This ensures that a row gets generated
        -- even if the day has 0 revenue.  Note that we generate days before
        -- the start of october - this is because our window function needs
        -- to know the revenue for those days for its calculations.
        (select
            cast(generate_series(timestamp '2012-07-10', '2012-08-31','1 day') as
date) as date
        )  as dategen
        left outer join
            -- left join to a table of per-day revenue
            (select cast(bks.starttime as date) as date,
                sum(case
                    when memid = 0 then slots * facs.guestcost
                    else slots * membercost
                end) as rev

                from cd.bookings bks
                inner join cd.facilities facs
                    on bks.facid = facs.facid
                group by cast(bks.starttime as date)
            ) as revdata
            on dategen.date = revdata.date
    ) as subq
    where date >= '2012-08-01'
order by date;
```

You'll note that we've been wanting to work out daily revenue quite frequently. Rather than inserting that calculation into all our queries, which is rather messy (and will cause us a big headache if we ever change our schema), we probably want to store that information somewhere. Your first thought might be to calculate information and store it somewhere for later use. This is a common tactic for large data warehouses, but it can cause us some problems - if we ever go back and edit our data, we need to remember to recalculate. For non-enormous-scale data like we're looking at here, we can just create a view instead. A view is essentially a stored query that looks exactly like a table. Under the covers, the DBMS just subsititutes in the relevant portion of the view definition when you select data from it. They're very easy to create, as you can see below:

```
create or replace view cd.dailyrevenue as
    select  cast(bks.starttime as date) as date,
        sum(case
            when memid = 0 then slots * facs.guestcost
            else slots * membercost
        end) as rev

        from cd.bookings bks
        inner join cd.facilities facs
            on bks.facid = facs.facid
        group by cast(bks.starttime as date);
```

You can see that this makes our query an awful lot simpler!

```
select date, avgrev from (
    select  dategen.date as date,
        avg(revdata.rev) over(order by dategen.date rows 14 preceding) as avgrev
    from
        (select
            cast(generate_series(timestamp '2012-07-10', '2012-08-31','1 day') as
date) as date
        )  as dategen
        left outer join
            cd.dailyrevenue as revdata on dategen.date = revdata.date
        ) as subq
    where date ⩾ '2012-08-01'
order by date;
```

As well as storing frequently-used query fragments, views can be used for a variety of purposes, including restricting access to certain columns of a table.

Working with Timestamps

Dates/Times in SQL are a complex topic, deserving of a category of their own. They're also fantastically powerful, making it easier to work with variable-length concepts like 'months' than many programming languages.

Before getting started on this category, it's probably worth taking a look over the PostgreSQL **docs page** on date/time functions. You might also want to complete the aggregate functions category, since we'll use some of those capabilities in this section.

Produce a timestamp for 1 a.m. on the 31st of August 2012

Produce a timestamp for 1 a.m. on the 31st of August 2012.

Expected results:

timestamp
2012-08-31 01:00:00

Answer:

```
select timestamp '2012-08-31 01:00:00';
```

Here's a pretty easy question to start off with! SQL has a bunch of different date and time types, which you can peruse at your leisure over at the excellent **Postgres documentation**. These basically allow you to store dates, times, or timestamps (date+time).

The approved answer is the best way to create a timestamp under normal circumstances. You can also use casts to change a correctly formatted string into a timestamp, for example:

```
select '2012-08-31 01:00:00'::timestamp;
select cast('2012-08-31 01:00:00' as timestamp);
```

The former approach is a Postgres extension, while the latter is SQL-standard. You'll note that in many of our earlier questions, we've used bare strings without specifying a data type. This works because when Postgres is working with a value coming out of a timestamp column of a table (say), it knows to cast our strings to timestamps.

Timestamps can be stored with or without time zone information. We've chosen not to here, but if you like you could format the timestamp like "2012-08-31 01:00:00 +00:00", assuming UTC. Note that timestamp with time zone is a different type to timestamp - when you're declaring it, you should use `TIMESTAMP WITH TIME ZONE 2012-08-31 01:00:00 +00:00.`

Finally, have a bit of a play around with some of the different date/time serialisations described in the Postgres docs. You'll find that Postgres is extremely flexible with the formats it accepts, although my recommendation to you would be to use the standard serialisation we've used here - you'll find it unambiguous and easy to port to other DBs.

Subtract timestamps from each other

Find the result of subtracting the timestamp '2012-07-30 01:00:00' from the timestamp '2012-08-31 01:00:00'

Expected results:

interval
32 days

Answer:

```
select timestamp '2012-08-31 01:00:00' - timestamp '2012-07-30 01:00:00' as
interval;
```

Subtracting timestamps produces an `INTERVAL` data type. `INTERVAL`s are a special data type for representing the difference between two `TIMESTAMP` types. When subtracting timestamps, Postgres will typically give an interval in terms of days, hours, minutes, seconds, without venturing into months. This generally makes life easier, since months are of variable lengths.

One of the useful things about intervals, though, is the fact that they *can* encode months. Let's imagine that I want to schedule something to occur in exactly one month's time, regardless of the length of my month. To do this, I could use `[timestamp] + interval '1 month'`.

Intervals stand in contrast to SQL's treatment of `DATE` types. Dates don't use intervals - instead, subtracting two dates will return an integer representing the number of days between the two dates. You can also add integer values to dates. This is sometimes more convenient, depending on how much intelligence you require in the handling of your dates!

Generate a list of all the dates in October 2012

Produce a list of all the dates in October 2012. They can be output as a timestamp (with time set to midnight) or a date.

Expected results:

ts
2012-10-01 00:00:00
2012-10-02 00:00:00
2012-10-03 00:00:00
2012-10-04 00:00:00
2012-10-05 00:00:00
2012-10-06 00:00:00
2012-10-07 00:00:00
2012-10-08 00:00:00
2012-10-09 00:00:00
2012-10-10 00:00:00
2012-10-11 00:00:00
2012-10-12 00:00:00
2012-10-13 00:00:00
2012-10-14 00:00:00
2012-10-15 00:00:00
2012-10-16 00:00:00
2012-10-17 00:00:00
2012-10-18 00:00:00
2012-10-19 00:00:00
2012-10-20 00:00:00
2012-10-21 00:00:00
2012-10-22 00:00:00
2012-10-23 00:00:00
2012-10-24 00:00:00
2012-10-25 00:00:00
2012-10-26 00:00:00
2012-10-27 00:00:00
2012-10-28 00:00:00
2012-10-29 00:00:00
2012-10-30 00:00:00

ts
2012-10-31 00:00:00

Answer:

```
select generate_series(timestamp '2012-10-01', timestamp '2012-10-31', interval '1
day') as ts;
```

One of the best features of Postgres over other DBs is a simple function called `GENERATE_SERIES`. This function allows you to generate a list of dates or numbers, specifying a start, an end, and an increment value. It's extremely useful for situations where you want to output, say, sales per day over the course of a month. A typical way to do that on a table containing a list of sales might be to use a `SUM` aggregation, grouping by the date and product type. Unfortunately, this approach has a flaw: if there are no sales for a given day, it won't show up! To make it work properly, you need to left join from a sequential list of timestamps to the aggregated data to fill in the blank spaces.

On other database systems, it's not uncommon to keep a 'calendar table' full of dates, with which you can perform these joins. Alternatively, on some systems you can write an analogue to generate_series using recursive CTEs. Fortunately for us, Postgres makes our lives a lot easier!

Get the day of the month from a timestamp

Get the day of the month from the timestamp '2012-08-31' as an integer.

Expected results:

date_part
31

Answer:

```
select extract(day from timestamp '2012-08-31');
```

The `EXTRACT` function is used for getting sections of a timestamp or interval. You can get the value of any field in the timestamp as an integer.

Work out the number of seconds between timestamps

Work out the number of seconds between the timestamps '2012-08-31 01:00:00' and '2012-09-02 00:00:00'

Expected results:

date_part
169200

Answer:

```
select extract(epoch from (timestamp '2012-09-02 00:00:00' - '2012-08-31
01:00:00'));
```

The above answer is a Postgres-specific trick. Extracting the epoch converts an interval or timestamp into a number of seconds, or the number of seconds since epoch (January 1st, 1970) respectively. If you want the number of minutes, hours, etc you can just divide the number of seconds appropriately.

If you want to write more portable code, you will unfortunately find that you cannot use `extract epoch`. Instead you will need to use something like:

```
select  extract(day from ts.int)*60*60*24 +
    extract(hour from ts.int)*60*60 +
    extract(minute from ts.int)*60 +
    extract(second from ts.int)
    from
        (select timestamp '2012-09-02 00:00:00' - '2012-08-31 01:00:00' as int) ts
```

Answer:

This is, as you can observe, rather awful. If you're planning to write cross platform SQL, I would consider having a library of common user defined functions for each DBMS, allowing you to normalise any common requirements like this. This keeps your main codebase a lot cleaner.

Work out the number of days in each month of 2012

For each month of the year in 2012, output the number of days in that month. Format the output as an integer column containing the month of the year, and a second column containing an interval data type.

Expected results:

month	length
1	31 days
2	29 days
3	31 days
4	30 days
5	31 days
6	30 days
7	31 days
8	31 days
9	30 days
10	31 days
11	30 days
12	31 days

Answer:

```
select  extract(month from cal.month) as month,
    (cal.month + interval '1 month') - cal.month as length
    from
    (
        select generate_series(timestamp '2012-01-01', timestamp '2012-12-01',
interval '1 month') as month
    ) cal
order by month;
```

This answer shows several of the concepts we've learned. We use the `GENERATE_SERIES` function to produce a year's worth of timestamps, incrementing a month at a time. We then use the `EXTRACT` function to get the month number. Finally, we subtract each timestamp + 1 month from itself.

It's worth noting that subtracting two timestamps will always produce an interval in terms of days (or portions of a day). You won't just get an answer in terms of months or years, because the length of those time periods is variable.

Work out the number of days remaining in the month

For any given timestamp, work out the number of days remaining in the month. The current day should count as a whole day, regardless of the time. Use '2012-02-11 01:00:00' as an example timestamp for the purposes of making the answer. Format the output as a single interval value.

Expected results:

remaining
19 days

Answer:

```
select (date_trunc('month',ts.testts) + interval '1 month')
        - date_trunc('day', ts.testts) as remaining
    from (select timestamp '2012-02-11 01:00:00' as testts) ts
```

The star of this particular show is the `DATE_TRUNC` function. It does pretty much what you'd expect - truncates a date to a given minute, hour, day, month, and so on. The way we've solved this problem is to truncate our timestamp to find the month we're in, add a month to that, and subtract our timestamp. To ensure partial days get treated as whole days, the timestamp we subtract is truncated to the nearest day.

Note the way we've put the timestamp into a subquery. This isn't required, but it does mean you can give the timestamp a name, rather than having to list the literal repeatedly.

Work out the end time of bookings

Return a list of the start and end time of the last 10 bookings (ordered by the time at which they end, followed by the time at which they start) in the system.

Expected results:

starttime	endtime
2013-01-01 15:30:00	2013-01-01 16:00:00
2012-09-30 19:30:00	2012-09-30 20:30:00
2012-09-30 19:00:00	2012-09-30 20:30:00
2012-09-30 19:30:00	2012-09-30 20:00:00
2012-09-30 19:00:00	2012-09-30 20:00:00
2012-09-30 19:00:00	2012-09-30 20:00:00
2012-09-30 18:30:00	2012-09-30 20:00:00
2012-09-30 18:30:00	2012-09-30 20:00:00
2012-09-30 19:00:00	2012-09-30 19:30:00
2012-09-30 18:30:00	2012-09-30 19:30:00

Answer:

```
select starttime, starttime + slots*(interval '30 minutes') endtime
    from cd.bookings
    order by endtime desc, starttime desc
    limit 10
```

This question simply returns the start time for a booking, and a calculated end time which is equal to `start time + (30 minutes * slots)`. Note that it's perfectly okay to multiply intervals.

The other thing you'll notice is the use of order by and limit to get the last ten bookings. All this does is order the bookings by the (descending) time at which they end, and pick off the top ten.

Return a count of bookings for each month

Return a count of bookings for each month, sorted by month

Expected results:

month	count
2012-07-01 00:00:00	658
2012-08-01 00:00:00	1472
2012-09-01 00:00:00	1913
2013-01-01 00:00:00	1

Answer:

```
select date_trunc('month', starttime) as month, count(*)
    from cd.bookings
    group by month
    order by month
```

This one is a fairly simple reuse of concepts we've seen before. We simply count the number of bookings, and aggregate by the booking's start time, truncated to the month.

Work out the utilisation percentage for each facility by month

Work out the utilisation percentage for each facility by month, sorted by name and month, rounded to 1 decimal place. Opening time is 8am, closing time is 8.30pm. You can treat every month as a full month, regardless of if there were some dates the club was not open.

Expected results:

name	month	utilisation
Badminton Court	2012-07-01 00:00:00	23.2
Badminton Court	2012-08-01 00:00:00	59.2
Badminton Court	2012-09-01 00:00:00	76.0
Massage Room 1	2012-07-01 00:00:00	34.1
Massage Room 1	2012-08-01 00:00:00	63.5
Massage Room 1	2012-09-01 00:00:00	86.4
Massage Room 2	2012-07-01 00:00:00	3.1
Massage Room 2	2012-08-01 00:00:00	10.6
Massage Room 2	2012-09-01 00:00:00	16.3
Pool Table	2012-07-01 00:00:00	15.1
Pool Table	2012-08-01 00:00:00	41.5
Pool Table	2012-09-01 00:00:00	62.8
Pool Table	2013-01-01 00:00:00	0.1
Snooker Table	2012-07-01 00:00:00	20.1
Snooker Table	2012-08-01 00:00:00	42.1
Snooker Table	2012-09-01 00:00:00	56.8
Squash Court	2012-07-01 00:00:00	21.2
Squash Court	2012-08-01 00:00:00	51.6
Squash Court	2012-09-01 00:00:00	72.0
Table Tennis	2012-07-01 00:00:00	13.4
Table Tennis	2012-08-01 00:00:00	39.2
Table Tennis	2012-09-01 00:00:00	56.3
Tennis Court 1	2012-07-01 00:00:00	34.8
Tennis Court 1	2012-08-01 00:00:00	59.2
Tennis Court 1	2012-09-01 00:00:00	78.8
Tennis Court 2	2012-07-01 00:00:00	26.7
Tennis Court 2	2012-08-01 00:00:00	62.3
Tennis Court 2	2012-09-01 00:00:00	78.4

Answer:

```
select name, month,
    round((100*slots)/
        cast(
            25*(cast((month + interval '1 month') as date)
            - cast (month as date)) as numeric),1) as utilisation
    from  (
        select facs.name as name, date_trunc('month', starttime) as month,
sum(slots) as slots
            from cd.bookings bks
            inner join cd.facilities facs
                on bks.facid = facs.facid
            group by facs.facid, month
    ) as inn
order by name, month
```

The meat of this query (the inner subquery) is really quite simple: an aggregation to work out the total number of slots used per facility per month. If you've covered the rest of this section and the category on aggregates, you likely didn't find this bit too challenging.

This query does, unfortunately, have some other complexity in it: working out the number of days in each month. We can calculate the number of days between two months by subtracting two timestamps with a month between them. This, unfortunately, gives us back on interval datatype, which we can't use to do mathematics. In this case we've worked around that limitation by converting our timestamps into *dates* before subtracting. Subtracting date types gives us an integer number of days.

A alternative to this workaround is to convert the interval into an *epoch* value: that is, a number of seconds. To do this use `EXTRACT(EPOCH FROM month)/(24*60*60)`. This is arguably a much nicer way to do things, but is much less portable to other database systems.

String Operations

String operations in most RDBMSs are, arguably, needlessly painful. Fortunately, Postgres is better than most in this regard, providing strong regular expression support. This section covers basic string manipulation, use of the LIKE operator, and use of regular expressions. I also make an effort to show you some alternative approaches that work reliably in most RDBMSs. Be sure to check out Postgres' string function **docs page** if you're not confident about these exercises.

Anthony Molinaro's **SQL Cookbook** provides some excellent documentation of (difficult) cross-DBMS compliant SQL string manipulation. I'd strongly recommend his book, particularly as it's published by O'Reilly, whose ethical policy of DRM-free ebook distribution deserves rich rewards.

Format the names of members

Output the names of all members, formatted as 'Surname, Firstname'

Expected results:

name
GUEST, GUEST
Smith, Darren
Smith, Tracy
Rownam, Tim
Joplette, Janice
Butters, Gerald
Tracy, Burton
Dare, Nancy
Boothe, Tim
Stibbons, Ponder
Owen, Charles
Jones, David
Baker, Anne
Farrell, Jemima
Smith, Jack
Bader, Florence
Baker, Timothy
Pinker, David
Genting, Matthew
Mackenzie, Anna
Coplin, Joan
Sarwin, Ramnaresh
Jones, Douglas
Rumney, Henrietta
Farrell, David
Worthington-Smyth, Henry
Purview, Millicent
Tupperware, Hyacinth
Hunt, John
Crumpet, Erica

name
Smith, Darren

Answer:

```
select surname || ', ' || firstname as name from cd.members
```

Building strings in sql is similar to other languages, with the exception of the concatenation operator: ||. Some systems (like SQL Server) use +, but || is the SQL standard.

Find facilities by a name prefix

Find all facilities whose name begins with 'Tennis'. Retrieve all columns.

Expected results:

facid	name	membercost	guestcost	initialoutlay	monthlymaintenance
0	Tennis Court 1	5	25	10000	200
1	Tennis Court 2	5	25	8000	200

Answer:

```
select * from cd.facilities where name like 'Tennis%';
```

The SQL `LIKE` operator is a highly standard way of searching for a string using basic matching. The % character matches any string, while _ matches any single character.

One point that's worth considering when you use `LIKE` is how it uses indexes. If you're using the 'C' locale, any `LIKE` string with a fixed beginning (as in our example here) can use an index. If you're using any other locale, `LIKE` will not use any index by default. See here for details on how to change that.

Perform a case-insensitive search

Perform a case-insensitive search to find all facilities whose name begins with 'tennis'. Retrieve all columns.

Expected results:

facid	name	membercost	guestcost	initialoutlay	monthlymaintenance
0	Tennis Court 1	5	25	10000	200
1	Tennis Court 2	5	25	8000	200

Answer:

```
select * from cd.facilities where upper(name) like 'TENNIS%';
```

There's no direct operator for case-insensitive comparison in standard SQL. Fortunately, we can take a page from many other language's books, and simply force all values into upper case when we do our comparison. This renders case irrelevant, and gives us our result.

Alternatively, Postgres does provide the `ILIKE` operator, which performs case insensitive searches. This isn't standard SQL, but it's arguably more clear.

You should realise that running a function like `UPPER` over a column value prevents Postgres from making use of any indexes on the column (the same is true for `ILIKE`). Fortunately, Postgres has got your back: rather than simply creating indexes over columns, you can also create indexes over **expressions**. If you created an index over `UPPER(name)`, this query could use it quite happily.

Find telephone numbers with parentheses

You've noticed that the club's member table has telephone numbers with very inconsistent formatting. You'd like to find all the telephone numbers that contain parentheses, returning the member ID and telephone number sorted by member ID.

Expected results:

memid	telephone
0	(000) 000-0000
3	(844) 693-0723
4	(833) 942-4710
5	(844) 078-4130
6	(822) 354-9973
7	(833) 776-4001
8	(811) 433-2547
9	(833) 160-3900
10	(855) 542-5251
11	(844) 536-8036
13	(855) 016-0163
14	(822) 163-3254
15	(833) 499-3527
20	(811) 972-1377
21	(822) 661-2898
22	(822) 499-2232
24	(822) 413-1470
27	(822) 989-8876
28	(855) 755-9876
29	(855) 894-3758
30	(855) 941-9786
33	(822) 665-5327
35	(899) 720-6978
36	(811) 732-4816
37	(822) 577-3541

Answer:

```
select memid, telephone from cd.members where telephone ~ '[()]';
```

We've chosen to answer this using regular expressions, although Postgres does provide other string functions like `POSITION` that would do the job at least as well. Postgres implements POSIX regular expression matching via the ~ operator. If you've used regular expressions before, the functionality of the operator will be very familiar to you.

As an alternative, you can use the SQL standard `SIMILAR TO` operator. The regular expressions for this have similarities to the POSIX standard, but a lot of differences as well. Some of the most notable differences are:

- As in the `LIKE` operator, `SIMILAR TO` uses the '_' character to mean 'any character', and the '%' character to mean 'any string'.
- A `SIMILAR TO` expression must match the whole string, not just a substring as in posix regular expressions. This means that you'll typically end up bracketing an expression in '%' characters.
- The '.' character does not mean 'any character' in `SIMILAR TO` regexes: it's just a plain character.

The `SIMILAR TO` equivalent of the given answer is shown below:

```
select memid, telephone from cd.members where telephone similar to '%[()]%';
```

Finally, it's worth noting that regular expressions usually don't use indexes. Generally you don't want your regex to be responsible for doing heavy lifting in your query, because it will be slow. If you need fuzzy matching that works fast, consider working out if your needs can be met by **full text search**.

Pad zip codes with leading zeroes

The zip codes in our example dataset have had leading zeroes removed from them by virtue of being stored as a numeric type. Retrieve all zip codes from the members table, padding any zip codes less than 5 characters long with leading zeroes. Order by the new zip code.

Expected results:

zip
00000
00234
00234
04321
04321
10383
11986
23423
28563
33862
34232
43532
43533
45678
52365
54333
56754
57392
58393
64577
65332
65464
66796
68666
69302
75655
78533
80743
84923
87630

zip
97676

Answer:

```
select lpad(cast(zipcode as char(5)),5,'0') zip from cd.members order by zip
```

Postgres' `LPAD` function is the star of this particular show. It does basically what you'd expect: allow us to produce a padded string. We need to remember to cast the zipcode to a string for it to be accepted by the `LPAD` function.

When inheriting an old database, It's not that unusual to find wonky decisions having been made over data types. You may wish to fix mistakes like these, but have a lot of code that would break if you changed datatypes. In that case, one option (depending on performance requirements) is to create a view over your table which presents the data in a fixed-up manner, and gradually migrate.

Count the number of members whose surname starts with each letter of the alphabet

You'd like to produce a count of how many members you have whose surname starts with each letter of the alphabet. Sort by the letter, and don't worry about printing out a letter if the count is 0.

Expected results:

letter	count
B	5
C	2
D	1
F	2
G	2
H	1
J	3
M	1
O	1
P	2
R	2
S	6
T	2
W	1

Answer:

```
select substr (mems.surname,1,1) as letter, count(*) as count
    from cd.members mems
    group by letter
    order by letter
```

This exercise is fairly straightforward. You simply need to retrieve the first letter of the member's surname, and do some basic aggregation to achieve a count. We use the `SUBSTR` function here, but there's a variety of other ways you can achieve the same thing. The `LEFT` function, for example, returns you the first n characters from the left of the string. Alternatively, you could use the `SUBSTRING` function, which allows you to use regular expressions to extract a portion of the string.

One point worth noting: as you can see, string functions in SQL are based on 1-indexing, not the 0-indexing that you're probably used to. This will likely trip you up once or twice before you get used to it :-)

Clean up telephone numbers

The telephone numbers in the database are very inconsistently formatted. You'd like to print a list of member ids and numbers that have had '-','(',')', and ' ' characters removed. Order by member id.

Expected results:

memid	telephone
0	0000000000
1	5555555555
2	5555555555
3	8446930723
4	8339424710
5	8440784130
6	8223549973
7	8337764001
8	8114332547
9	8331603900
10	8555425251
11	8445368036
12	8440765141
13	8550160163
14	8221633254
15	8334993527
16	8339410824
17	8114096734
20	8119721377
21	8226612898
22	8224992232
24	8224131470
26	8445368036
27	8229898876
28	8557559876
29	8558943758
30	8559419786
33	8226655327
35	8997206978
36	8117324816

memid	telephone
37	8225773541

Answer:

```
select memid, translate(telephone, '-() ', '') as telephone
    from cd.members
    order by memid;
```

The most direct solution is probably the `TRANSLATE` function, which can be used to replace characters in a string. You pass it three strings: the value you want altered, the characters to replace, and the characters you want them replaced with. In our case, we want all the characters deleted, so our third parameter is an empty string.

As is often the way with strings, we can also use regular expressions to solve our problem. The `REGEXP_REPLACE` function provides what we're looking for: we simply pass a regex that matches all non-digit characters, and replace them with nothing, as shown below. The 'g' flag tells the function to replace as many instances of the pattern as it can find. This solution is perhaps more robust, as it cleans out more bad formatting.

```
select memid, regexp_replace(telephone, '[^0-9]', '', 'g') as telephone
    from cd.members
    order by memid;
```

Making automated use of free-formatted text data can be a chore. Ideally you want to avoid having to constantly write code to clean up the data before using it, so you should consider having your database enforce correct formatting for you. You can do this using a CHECK constraint on your column, which allow you to reject any poorly-formatted entry. It's tempting to perform this kind of validation in the application layer, and this is certainly a valid approach. As a general rule, if your database is getting used by multiple applications, favour pushing more of your checks down into the database to ensure consistent behaviour between the apps.

Occasionally, adding a constraint isn't feasible. You may, for example, have two different legacy applications asserting differently formatted information. If you're unable to alter the applications, you have a couple of options to consider. Firstly, you can define a trigger on your table. This allows you to intercept data before (or after) it gets asserted to your table, and normalise it into a single format. Alternatively, you could build a view over your table that cleans up information on the fly, as it's read out. Newer applications can read from the view and benefit from more reliably formatted information.

Recursive Queries

Common Table Expressions allow us to, effectively, create our own temporary tables for the duration of a query - they're largely a convenience to help us make more readable SQL. Using the WITH RECURSIVE modifier, however, it's possible for us to create recursive queries. This is enormously advantageous for working with tree and graph-structured data - imagine retrieving all of the relations of a graph node to a given depth, for example.

This category shows you some basic recursive queries that are possible using our dataset.

Find the upward recommendation chain for member ID 27

Find the upward recommendation chain for member ID 27: that is, the member who recommended them, and the member who recommended that member, and so on. Return member ID, first name, and surname. Order by descending member id.

Expected results:

recommender	firstname	surname
20	Matthew	Genting
5	Gerald	Butters
1	Darren	Smith

Answer:

```
with recursive recommenders(recommender) as (
    select recommendedby from cd.members where memid = 27
    union all
    select mems.recommendedby
        from recommenders recs
        inner join cd.members mems
            on mems.memid = recs.recommender
)
select recs.recommender, mems.firstname, mems.surname
    from recommenders recs
    inner join cd.members mems
        on recs.recommender = mems.memid
order by memid desc
```

`WITH RECURSIVE` is a fantastically useful piece of functionality that many developers are unaware of. It allows you to perform queries over hierarchies of data, which is very difficult by other means in SQL. Such scenarios often leave developers resorting to multiple round trips to the database system.

You've seen `WITH` before. The Common Table Expressions (CTEs) defined by WITH give you the ability to produce inline views over your data. This is normally just a syntactic convenience, but the `RECURSIVE` modifier adds the ability to join against results already produced to produce even more. A recursive `WITH` takes the basic form of:

```
WITH RECURSIVE NAME(columns) as (
    <initial statement>
    UNION ALL
    <recursive statement>
)
```

The initial statement populates the initial data, and then the recursive statement runs repeatedly to produce more. Each step of the recursion can access the CTE, but it sees within it only the data produced by the previous iteration. It repeats until an iteration produces no additional data.

The most simple example of a recursive `WITH` might look something like this:

```
with recursive increment(num) as (
    select 1
    union all
    select increment.num + 1 from increment where increment.num < 5
)
select * from increment;
```

The initial statement produces '1'. The first iteration of the recursive statement sees this as the content of `increment`, and produces '2'. The next iteration sees the content of `increment` as '2', and so on. Execution terminates when the recursive statement produces no additional data.

With the basics out of the way, it's fairly easy to explain our answer here. The initial statement gets the ID of the person who recommended the member we're interested in. The recursive statement takes the results of the initial statement, and finds the ID of the person who recommended them. This value gets forwarded on to the next iteration, and so on.

Now that we've constructed the recommenders CTE, all our main `SELECT` statement has to do is get the member IDs from recommenders, and join to them members table to find out their names.

Find the downward recommendation chain for member ID 1

Find the downward recommendation chain for member ID 1: that is, the members they recommended, the members those members recommended, and so on. Return member ID and name, and order by ascending member id.

Expected results:

memid	firstname	surname
4	Janice	Joplette
5	Gerald	Butters
7	Nancy	Dare
10	Charles	Owen
11	David	Jones
14	Jack	Smith
20	Matthew	Genting
21	Anna	Mackenzie
26	Douglas	Jones
27	Henrietta	Rumney

Answer:

```
with recursive recommendeds(memid) as (
    select memid from cd.members where recommendedby = 1
    union all
    select mems.memid
        from recommendeds recs
        inner join cd.members mems
            on mems.recommendedby = recs.memid
)
select recs.memid, mems.firstname, mems.surname
    from recommendeds recs
    inner join cd.members mems
        on recs.memid = mems.memid
order by memid
```

This is a pretty minor variation on the previous question. The essential difference is that we're now heading in the opposite direction. One interesting point to note is that unlike the previous example, this CTE produces multiple rows per iteration, by virtue of the fact that we're heading down the recommendation tree (following all branches) rather than up it.

Produce a CTE that can return the upward recommendation chain for any member

Produce a CTE that can return the upward recommendation chain for any member. You should be able to select recommender from recommenders where member=x. Demonstrate it by getting the chains for members 12 and 22. Results table should have member and recommender, ordered by member ascending, recommender descending.

Expected results:

member	recommender	firstname	surname
12	9	Ponder	Stibbons
12	6	Burton	Tracy
22	16	Timothy	Baker
22	13	Jemima	Farrell

Answer:

```
with recursive recommenders(recommender, member) as (
    select recommendedby, memid
        from cd.members
    union all
    select mems.recommendedby, recs.member
        from recommenders recs
        inner join cd.members mems
            on mems.memid = recs.recommender
)
select recs.member member, recs.recommender, mems.firstname, mems.surname
    from recommenders recs
    inner join cd.members mems
        on recs.recommender = mems.memid
    where recs.member = 22 or recs.member = 12
```

```
order by recs.member asc, recs.recommender desc
```

This question requires us to produce a CTE that can calculate the upward recommendation chain for any user. Most of the complexity of working out the answer is in realising that we now need our CTE to produce two columns: one to contain the member we're asking about, and another to contain the members in their recommendation tree. Essentially what we're doing is producing a table that flattens out the recommendation hierarchy.

Since we're looking to produce the chain for every user, our initial statement needs to select data for each user: their ID and who recommended them. Subsequently, we want to pass the member field through each iteration without changing it, while getting the next recommender. You can see that the recursive part of our statement hasn't really changed, except to pass through the 'member' field.