Databases

Yue Xiong, LMU, yue.xiong@stat.uni-muenchen.de

Chapter 01 Databases

Basic Introduction to SQL and Relational Databases

SQL is a language designed for a very specific purpose: to interact with relational databases.

- **Database**: A database is a structured collection of data. There are various different ways of structuring the database, and there may or may not be information about the relationship between entities in the database.
- **Query**: A query is a request for data from the database.
- **Database Management System (DBMS)**: A DBMS is a system of storing and managing databases, including querying the database.
- **Relational Database Management System (RDBMS)**: In an RDBMS, data records are stored in *tables*, each of which has a predefined set of *columns*, the pieces of information captured for each record in a table, and *rows* in the table, where each row has a place to store a value for every column in the table.

Tables, including their columns, column types and relationships with other tables, are defined in a database **schema**. Many times, tables will contain a **primary key**, one or more columns that uniquely define a row. You can think of the primary key as a kind of ID, in which each row is given a unique ID. Tables can also contain **foreign keys**, which are column(s) that comprise the primary key in another table and, thus, provides a way of matching between multiple tables.

In this notebook, we will use SQL to: - Select data subsets - Sum over groups - Create new tables - Count distinct values of desired variables - Order data by chosen variables - Join tables together

Step 1: Estabilish a Connection to the Database using `R`

In order to get access to the corresponding SQL database, we need to firstly install the required R package developed for accessing databases: `RSQlite`. Furthermore, by installing the `dbplyr` package, it helps provide an interface focusing on retrieving and analyzing datasets by generating `SELECT` SQL statements.

```r
# install.packages(c("dbplyr", "RSQLite", "DBI"))  # install the required packages
library('dbplyr')  # load the corresponding libraries
```

```
## Warning: package 'dbplyr' was built under R version 4.0.5
```

```r
library('dplyr')
```

```
## Warning: package 'dplyr' was built under R version 4.0.5
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:dbplyr':
##
##     ident, sql

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library('RSQLite')
```

```
## Warning: package 'RSQLite' was built under R version 4.0.5
```

```
library('DBI')
```

```
## Warning: package 'DBI' was built under R version 4.0.5
```

```
# connect to the corresponding database
database_path = "F:/hiwi_work_notebook/bdss-notebooks/awards.db"
awards <- DBI::dbConnect(RSQLite::SQLite(), database_path)
```

```
src_dbi(awards)
```

```
## src:  sqlite 3.36.0 [F:\hiwi_work_notebook\bdss-notebooks\awards.db]
## tbls: awards, employees, students
```

Formulate Data Query To connect to tables within a database, we can use the `tbl()`function from `dplyr`. Also, this function can be used to send queries the database. Let's demonstrate this functionality by looking at the **awards** data.

```
tbl(awards, sql("SELECT * FROM awards LIMIT 20"))
```

```
## # Source:   SQL [?? x 2]
## # Database: sqlite 3.36.0 [F:\hiwi_work_notebook\bdss-notebooks\awards.db]
##    employee award
##    <chr>    <chr>
##  1 90014540 aw1000
##  2 90026679 aw1001
##  3 90017693 aw1002
##  4 90028696 aw1003
##  5 90013819 aw1004
##  6 90001118 aw1005
##  7 90001300 aw1006
##  8 90023086 aw1007
##  9 90017074 aw1008
## 10 90025003 aw1009
## # ... with more rows
```

You should see 20 rows of the awards dataset. Let's go over the basics of this SQL command.

- **SELECT:** We start out with the SELECT statement. The SELECT statement specifies which variables (columns) you want.
  - Here, we used SELECT *. The "*" just says that we want all the variables.
  - If we wanted a few columns, we would use the column names separated by commas instead of "*".
- **FROM:** Now, let's look at the next part of the query, FROM awards. This part of the query specifies the table, awards, from which we want to retrieve the data. Most of your queries will begin in this fashion, describing which columns you want and from which table.
- **LIMIT:** We typically include a LIMIT statement at the end of our query so that we don't get overloaded with rows being output. Here, LIMIT 20 means that we just want the first ten rows. Many times, the LIMIT that you want will be higher than 20 – you might generally prefer to use 1000 or so. Having a LIMIT for all queries is highly recommended even if you know only a few rows will be shown, since it acts as a safety precaution against (for example) displaying millions of rows of data.

In this case, we've put everything in one line, but that's not necessary. We could have split the code up into multiple lines, like so:

```
SELECT *
FROM awards
LIMIT 20;
```

This gives the same output as our original query. Generally, once queries start getting longer, breaking up the code into multiple lines can be very helpful in organizing your code and making it easier to read.

Along those lines, note that we used a semi-colon at the end of the query to mark the end of the query. That isn't absolutely necessary here, but it does help mark the end of a query and is required in other applications of SQL, so it's good practice to use it.

> **Side note about capitalization**
>
> If you notice, we've been using all caps for SQL commands and all lowercase for data table and schema names. This is simply a convention, as SQL is not case sensitive. For example, we could have run select * from inmate limit 20 and it would have given us the exact same output as the first query.
>
> This does mean you need to be careful when using column names. If your column name has capital letters in it, you need use double quotes (e.g. "INMATE_DOC_NUMBER") to preserve the capitalization. For this reason, you might find that using all lowercase letters in column names is preferable, which is what we've done here.
>
> Note that the LIMIT provides one simple way to get a "sample" of data; however, using LIMIT does **not provide a *random*** sample. You may get different samples of data than others using just the LIMIT clause, but it is just based on what is fastest for the database to return.

Basic Summaries of the Data One of the basic things you might be interested in doing is finding out how many rows there are in the dataset. You can do this using the COUNT statement.

Let's find the total number of employees that we have in our awards table.

```
tbl(awards, sql("SELECT COUNT(employee) FROM awards LIMIT 20"))
```

```
## # Source:   SQL [?? x 1]
## # Database: sqlite 3.36.0 [F:\hiwi_work_notebook\bdss-notebooks\awards.db]
##   ‘COUNT(employee)‘
##               <int>
## 1               903
```

To count the number of unique cases, you can use the DISTINCT statement. This checks how many unique values of that variable there are.

Let’s find the total number of unique individuals who were in those employees.

```
tbl(awards, sql("SELECT COUNT(DISTINCT employee) FROM awards LIMIT 20"))
```

```
## # Source:   SQL [?? x 1]
## # Database: sqlite 3.36.0 [F:\hiwi_work_notebook\bdss-notebooks\awards.db]
##   ‘COUNT(DISTINCT employee)‘
##                        <int>
## 1                        632
```

You can also do other basic summaries, such as finding sum using SUM or average using AVG. For instance,

```
tbl(awards, sql("SELECT AVG(employee) FROM awards LIMIT 20"))
```

```
## # Source:   SQL [?? x 1]
## # Database: sqlite 3.36.0 [F:\hiwi_work_notebook\bdss-notebooks\awards.db]
##   ‘AVG(employee)‘
##             <dbl>
## 1       86366789.
```

Conditional Statements Suppose we want to look at a subset of the data. We can use conditional statements to do this.

```
tbl(awards, sql("SELECT * FROM awards WHERE employee = '90014540'"))
```

```
## # Source:   SQL [?? x 2]
## # Database: sqlite 3.36.0 [F:\hiwi_work_notebook\bdss-notebooks\awards.db]
##   employee award
##   <chr>    <chr>
## 1 90014540 aw1000
```

The WHERE statement is used to return only data that meets certain conditions. Here, we used it to find the information for employee whose id is 90014540.

Common Comparison Operators Though there are some more complicated comparison operators (if you’re curious, feel free to look up what LIKE and IN do), these should cover most of what you want to do. - =: equal to - != or "<>": not equal to - <: less than - <=: less-than-or-equal-to - >: greater than - >=: greater-than-or-equal-to - IS NULL and IS NOT NULL: The signifier of a row in a column not having a value is a special keyword: NULL. To check for NULL, you use IS NULL or IS NOT NULL, rather than "=" or "!=". For example, to count the number of rows with NULL values for employee we might use the following:

```
SELECT *
FROM awards
WHERE employee IS NOT NULL
LIMIT 10;
```

Using Aggregation Functions What if we wanted to get summaries of the data, such as counts, aggregated by a categorical variable? We can do this using the `GROUP BY` statement.

- Using `GROUP BY` with single variable

Firstly, let's take a look at the students dataset.

```
tbl(awards, sql("SELECT * FROM students LIMIT 50"))
```

```
## # Source:   SQL [?? x 3]
## # Database: sqlite 3.36.0 [F:\hiwi_work_notebook\bdss-notebooks\awards.db]
##    employee occupation classification
##    <chr>    <chr>      <chr>
##  1 90045405 STUDENT    Student
##  2 90032828 STUDENT    Student
##  3 90022211 STUDENT    Student
##  4 90032459 STUDENT    Student
##  5 90020980 STUDENT    Student
##  6 90044261 STUDENT    Student
##  7 90022590 STUDENT    Student
##  8 90049372 STUDENT    Student
##  9 90048480 STUDENT    Student
## 10 90031317 STUDENT    Student
## # ... with more rows
```

And then we can use the `GROUP BY` statement.

```
tbl(awards, sql("SELECT employee, count(*) FROM students GROUP BY employee LIMIT 10"))
```

```
## # Source:   SQL [?? x 2]
## # Database: sqlite 3.36.0 [F:\hiwi_work_notebook\bdss-notebooks\awards.db]
##    employee ‘count(*)‘
##    <chr>         <int>
##  1 20479             1
##  2 29193             1
##  3 29227             1
##  4 36987             1
##  5 37759             1
##  6 90020502          1
##  7 90020503          1
##  8 90020531          1
##  9 90020566          1
## 10 90020580          1
```

We can also add queries like `ORDER BY count(*)` to keep the displayed results in an ascending order.

```
tbl(awards, sql("SELECT employee, count(*) FROM students GROUP BY employee ORDER BY count(*) LIMIT 20"))
```

```
## # Source:   SQL [?? x 2]
## # Database: sqlite 3.36.0 [F:\hiwi_work_notebook\bdss-notebooks\awards.db]
##    employee ‘count(*)‘
##    <chr>         <int>
##  1 20479             1
##  2 29193             1
##  3 29227             1
##  4 36987             1
##  5 37759             1
##  6 90020502          1
##  7 90020503          1
##  8 90020531          1
##  9 90020566          1
## 10 90020580          1
## # ... with more rows
```

- Using `GROUP BY` with Multiple Variables

Suppose we wanted to look at counts by employee(id) and by classification We could do this by adding the classification variable to the `GROUP BY` statement.

```
tbl(awards, sql("SELECT employee, classification, count(*) FROM students GROUP BY employee, classificat
```

```
## # Source:   SQL [?? x 3]
## # Database: sqlite 3.36.0 [F:\hiwi_work_notebook\bdss-notebooks\awards.db]
##    employee classification ‘count(*)‘
##    <chr>    <chr>               <int>
##  1 20479    Student                 1
##  2 29193    Student                 1
##  3 29227    Student                 1
##  4 36987    Student                 1
##  5 37759    Student                 1
##  6 90020502 Student                 1
##  7 90020503 Student                 1
##  8 90020531 Student                 1
##  9 90020566 Student                 1
## 10 90020580 Student                 1
## # ... with more rows
```

Notice that we used `DESC` after `ORDER BY`. This orders in descending order instead of the increasing order, so that we can see the groups with the most people at the top.

Conditional Statements After Aggregation Suppose we wanted to display only certain counts. We can use `HAVING` to do this.

```
tbl(awards, sql("SELECT employee, classification, count(*) FROM students GROUP BY employee, classificat
```

```
## # Source:   SQL [?? x 3]
## # Database: sqlite 3.36.0 [F:\hiwi_work_notebook\bdss-notebooks\awards.db]
##    employee classification ‘count(*)‘
##    <chr>    <chr>               <int>
##  1 20479    Student                 1
##  2 29193    Student                 1
##  3 29227    Student                 1
##  4 36987    Student                 1
##  5 37759    Student                 1
##  6 90020502 Student                 1
##  7 90020503 Student                 1
##  8 90020531 Student                 1
##  9 90020566 Student                 1
## 10 90020580 Student                 1
## # ... with more rows
```

This will only display the counts for which the count is greater than 0. Note that this is different from using `WHERE`, since the conditional statement comes after the `GROUP BY` statement. Basically, `HAVING` gives us a way of using the same types of conditional statements after we do our aggregation.

Joins One of the nice things about relational databases is organization using multiple tables that are linked together in some way. For example, suppose we have one table with 6 rows called **Table A**:

id	var1
1	5
2	10
3	2
4	6
5	22
6	9

And another table with 5 rows called **Table B**:

is	var2
2	2
5	4
6	1
7	2
8	0

Let’s say we want to combine Table A and Table B so that we have one table that contains information about `id`, `var1`, and `var2`. We want to do this by matching the two tables by what they have in common, `id`. That is, we want a table that looks like this (let’s call this **Table C**):

id	var1	var2
2	10	2
5	22	4
6	9	1

Table C has each `id` that was in both Table A and Table B. It also contains the appropriate values for `var1` and `var2` corresponding to each `id`. This kind of matching can be quite tricky to figure out manually, since there are different numbers of rows in each table, not all of the `id` values match for the two tables, and there are some `id` values that aren't in both. Fortunately for us, SQL is well-equipped to handle this task using the `JOIN` statement.

```
select * from students LIMIT 20
```

Table 4: Displaying records 1 - 10

employee	occupation	classification
90045405	STUDENT	Student
90032828	STUDENT	Student
90022211	STUDENT	Student
90032459	STUDENT	Student
90020980	STUDENT	Student
90044261	STUDENT	Student
90022590	STUDENT	Student
90049372	STUDENT	Student
90048480	STUDENT	Student
90031317	STUDENT	Student

```
SELECT awards.employee, awards.award, students.occupation, students.classification FROM students
JOIN awards on students.employee = awards.employee
LIMIT 100
```

Table 5: Displaying records 1 - 10

employee	award	occupation	classification
90045405	aw1375	STUDENT	Student
90032828	aw1376	STUDENT	Student
90022211	aw1377	STUDENT	Student
90032459	aw1378	STUDENT	Student
90020980	aw1379	STUDENT	Student
90044261	aw1380	STUDENT	Student
90022590	aw1381	STUDENT	Student
90049372	aw1382	STUDENT	Student
90048480	aw1383	STUDENT	Student
90031317	aw1399	STUDENT	Student

Now that we're connected and have established a plan for how we're joining two tables together, let's take a look at the SQL code that performs this join and break it down.

```
SELECT awards.employee, awards.award, students.occupation, students.classification FROM students
JOIN awards
on students.employee = awards.employee
LIMIT 100
```

Here, we want to `SELECT` each column from a data table that we get from joining the tables `awards` and `students`.

We can't just mash two tables together though – we need some way of making sure that the appropriate rows match. We do this with this line:

```
ON students.employee = awards.employee
```

This part specifies what we're joining on. That is, what is the ID variable that is in both tables that we want to match. They don't need to be named the same in both tables, though you do need to specify what they are in each table, even if they are the same, as well as which table they are from.

If you run the full code below, you should see the first 100 rows (because of the `LIMIT 100`) of the joined table. You should be able to scroll through all of the variables and see that we've managed to merge the `awards` and `students` tables together according to their IDs.

Different Types of Joins We've so far done only one type of join, an inner join. This is the default join (which is why we didn't need to specify anything more in the code). However, there are different types of joins.

- Left and Right Joins in SQL

Suppose we want to look at every single census block in one table, only filling in information from the second table if it exists. We'll illustrate this using Table A and Table B from before. Recall that our `JOIN` created Table C:

id	var1	var2
2	10	2
5	22	4
6	9	1

Instead, we want to create the following table:

id	var1	var2
1	5	*null*
2	10	2
3	2	*null*
4	6	*null*
5	22	4
6	9	1

Here, we've kept every single row in Table A, and simply filled in the information from Table B if it existed for that `id`. This is called a **LEFT JOIN**, since we're taking the table on the left (that is, Table A) and adding the information from Table B onto that. We could have also done a **RIGHT JOIN**, which does the same thing, except flipping the tables, giving us something that looks like:

id	var1	var2
2	10	2
5	22	4
6	9	1
7	*null*	2
8	*null*	0

```
SELECT awards.employee, awards.award, students.occupation, students.classification FROM students
LEFT JOIN awards on students.employee = awards.employee
LIMIT 100
```

Table 9: Displaying records 1 - 10

employee	award	occupation	classification
90045405	aw1375	STUDENT	Student
90032828	aw1376	STUDENT	Student
90022211	aw1377	STUDENT	Student
90032459	aw1378	STUDENT	Student
90020980	aw1379	STUDENT	Student
90044261	aw1380	STUDENT	Student
90022590	aw1381	STUDENT	Student
90049372	aw1382	STUDENT	Student
90048480	aw1383	STUDENT	Student
90031317	aw1399	STUDENT	Student

- Outer Join

An outer join keeps all unique ids, then puts `NULL` if it isn't part of that table. This is similar to a `LEFT` or `RIGHT JOIN`, except instead of only keeping all IDs from one table, it keeps them from both tables. Consider our example with Table A and Table B. We want to join them such that we get a table that looks like:

id	var1	var2
1	5	*null*
2	10	2
3	2	*null*
4	6	*null*
5	22	4
6	9	1
7	*null*	2
8	*null*	0

In a way, it's like combining the `LEFT` and `RIGHT JOIN`s so that we have all information from both tables.

Notice that we aren't able to show the outer join here, as it isn't supported by SQLite. We've provided the code here, but it won't run, so just make sure to keep it in mind for the future.

```
SELECT awards.employee, awards.award, students.occupation, students.classification
FROM students
FULL OUTER JOIN awards
on students.employee = awards.employee
LIMIT 100
```

Creating New Tables for Future Use

- Creating Tables

So far, we've mostly just been exploring the data without making any changes to the database. However, there might be times when we might want to create new tables. We can do this using `CREATE TABLE`. Let's use a previous example to create a new table.

```sql
CREATE TABLE joinedtable1 AS
SELECT *
FROM students
JOIN awards
ON students.employee = awards.employee
```

This should look mostly familiar, since everything after the first line is stuff we've already done. The first line creates a new table called `joinedtable` from the output.

This is a bit of a mess, though. We usually don't need everything from the tables that we do join, so we can choose what we keep. Let's create a new table that has just the information we need.

```sql
CREATE TABLE joinedtable2 AS
SELECT awards.employee, awards.award, students.occupation, students.classification
FROM students
JOIN awards
ON students.employee = awards.employee
```

Firstly, notice that we use aliasing to help make referring to tables easier. That is, in the third and fourth lines, we put "`a`" and "`b`" after each table to give it that alias. We can then use "`a`" and "`b`" whenever we refer to either table, which makes the `SELECT` statement easier.

Along those lines, notice that we specify which table each variable was from. If the column name is unique between the two tables (i.e. both tables don't have a column with the same name), then you don't need to specify the table as we've done. However, if they aren't unique and both tables have a variable with that name, you need to specify which one you want.

Finally, we've made the table easier to read by changing the name of the variable in the new table, using `AS` in the `SELECT` part of the query.

- Dropping Tables

Conversely, you can also drop, or delete, tables. We created a table in the previous section that we won't need, so let's drop it.

```sql
DROP TABLE joinedtable1
```

```sql
DROP TABLE joinedtable2
```

You might be tempted to avoid dropping tables since it seems relatively harmless to simply not use the table anymore without dropping them. However, it is important to keep databases clean and consider the amount of space each table takes up.