Running SQL in R

Cheng Peng

Contents

1 Introducion

To run SQL clauses in R, we need to use several R libraries (installed and loaded in the above R setup code chunk). There are different ways to run SQL queries in R. We only introduce one of them that runs the basic SQL code.

1.1 Connect R to Existing Database

If there is an existing database, the following code connects R to the database.

```
con <- DBI::dbConnect(drv = odbc::odbc(),
                      Driver = "driver_name",
                      Server = "server_url",
                      Database = "database_name",
                      user = "user", #optional
                      password = "password") #optional
```

1.2 Create A Database to Run SQL Queries in R

This short note shows the three basic steps to run SQL in R using R Markdown starting with a set of relational tables.

1. Load relational data tables as usual to R.
2. Create a SQLite (relational) database that contains these relational tables.
3. Create an R code chunk and connect to the created database using Chunk options.

2 Create SQLite Database with R

If modeling requires a data set that contains information from multiple relational data tables, we need to perform data management to aggregate the required information from different data tables. We can load the different data sets in different formats using appropriate R functions.

As an example, We use three ecological survey data sets to create a database.

```
#Load the sample data
plots <- read.csv("https://pengdsci.github.io/datasets/AnimalSurvey/plots.csv")
species <- read.csv("https://pengdsci.github.io/datasets/AnimalSurvey/species.csv")
surveys <- read.csv("https://pengdsci.github.io/datasets/AnimalSurvey/surveys.csv")
```

We next explore the relationship between the tables.

```
summary(plots)
```

```
    plot_id        plot_type        
 Min.   : 1.00   Length:24          
 1st Qu.: 6.75   Class :character  
 Median :12.50   Mode  :character  
 Mean   :12.50                     
 3rd Qu.:18.25                     
 Max.   :24.00                     
```

```
summary(species)
```

```
  species_id           genus             species             taxa          
 Length:54          Length:54          Length:54          Length:54         
 Class :character   Class :character   Class :character   Class :character  
 Mode  :character   Mode  :character   Mode  :character   Mode  :character  
```

```
summary(surveys)
```

```
       X           record_id         month             day       
 Min.   :    1   Min.   :    1   Min.   : 1.000   Min.   : 1.00  
 1st Qu.: 8888   1st Qu.: 8888   1st Qu.: 4.000   1st Qu.: 9.00  
 Median :17775   Median :17775   Median : 6.000   Median :16.00  
 Mean   :17775   Mean   :17775   Mean   : 6.478   Mean   :15.99  
 3rd Qu.:26662   3rd Qu.:26662   3rd Qu.:10.000   3rd Qu.:23.00  
 Max.   :35549   Max.   :35549   Max.   :12.000   Max.   :31.00  

      year         plot_id      species_id            sex           
 Min.   :1977   Min.   : 1.0   Length:35549       Length:35549      
 1st Qu.:1984   1st Qu.: 5.0   Class :character   Class :character  
 Median :1990   Median :11.0   Mode  :character   Mode  :character  
 Mean   :1990   Mean   :11.4                                        
 3rd Qu.:1997   3rd Qu.:17.0                                        
 Max.   :2002   Max.   :24.0

 hindfoot_length     weight
 Min.   : 2.00   Min.   :  4.00
 1st Qu.:21.00   1st Qu.: 20.00
 Median :32.00   Median : 37.00
 Mean   :29.29   Mean   : 42.67
 3rd Qu.:36.00   3rd Qu.: 48.00
 Max.   :70.00   Max.   :280.00
 NA's   :4111    NA's   :3266
```

The relational table `surveys` has attributes **species_id** and **plot_id** that connect relational tables `plots` and `species`. The primary and foreign keys are depicted in the following figure.

```r
include_graphics("img/RelationshipBetweenTables.png")
```

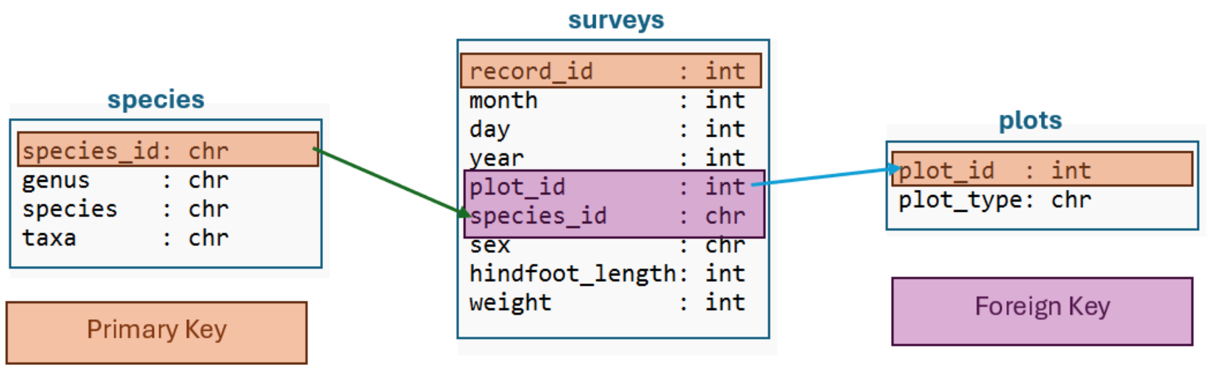


Figure 1: The primary and foreign keys of the three relational tables.

Next, we create a SQLit database using R libraries and then remove the data frames from the working directory.

```r
#Create database
con <- dbConnect(drv = SQLite(),
                 dbname = ":memory:")

#store sample data in the database
dbWriteTable(conn = con,
             name = "plots",
             value = plots)

dbWriteTable(conn = con,
             name = "species",
             value = species)

dbWriteTable(conn = con,
             name = "surveys",
             value = surveys)
```

```r
#remove the local data from the environment
rm(plots, species, surveys)
```

We can use the table view function `tbl()` to view the relational data tables in the database.

```r
tbl(src = con,          #  the source if the database connection profile
    c("surveys"))       #  the name of the table to preview
```

```
# Source:   table<surveys> [?? x 10]
# Database: sqlite 3.45.2 [:memory:]
       X record_id month   day  year plot_id species_id sex   hindfoot_length
   <int>     <int> <int> <int> <int>   <int> <chr>      <chr>           <int>
 1     1         1     7    16  1977       2 NL         M                  32
 2     2         2     7    16  1977       3 NL         M                  33
 3     3         3     7    16  1977       2 DM         F                  37
 4     4         4     7    16  1977       7 DM         M                  36
 5     5         5     7    16  1977       3 DM         M                  35
 6     6         6     7    16  1977       1 PF         M                  14
 7     7         7     7    16  1977       2 PE         F                  NA
 8     8         8     7    16  1977       1 DM         M                  37
 9     9         9     7    16  1977       1 DM         F                  34
10    10        10     7    16  1977       6 PF         F                  20
# i more rows
# i 1 more variable: weight <int>
```

```r
tbl(src = con, "species")
```

```
# Source:   table<species> [?? x 4]
# Database: sqlite 3.45.2 [:memory:]
   species_id genus            species         taxa
   <chr>      <chr>            <chr>           <chr>
 1 AB         Amphispiza       bilineata       Bird
 2 AH         Ammospermophilus harrisi         Rodent
 3 AS         Ammodramus       savannarum      Bird
 4 BA         Baiomys          taylori         Rodent
 5 CB         Campylorhynchus  brunneicapillus Bird
 6 CM         Calamospiza      melanocorys     Bird
 7 CQ         Callipepla       squamata        Bird
 8 CS         Crotalus         scutalatus      Reptile
 9 CT         Cnemidophorus    tigris          Reptile
10 CU         Cnemidophorus    uniparens       Reptile
# i more rows
```

```r
tbl(src = con, "plots")
```

```
# Source:   table<plots> [?? x 2]
# Database: sqlite 3.45.2 [:memory:]
   plot_id plot_type
     <int> <chr>
 1       1 Spectab exclosure
 2       2 Control
 3       3 Long-term Krat Exclosure
 4       4 Control
 5       5 Rodent Exclosure
 6       6 Short-term Krat Exclosure
 7       7 Rodent Exclosure
 8       8 Control
 9       9 Spectab exclosure
10      10 Rodent Exclosure
# i more rows
```

3 Running SQL Queries in R

To use SQL in RMarkdown, we need the following chunk options:

- sql
- connection = "database-name"
- output.var = "output-dataset-name"

If we create a data view only, we simply ignore option `output.var =`

Following are few examples of SQL queries based on the animal survey data tables in the database.

3.1 Basic SQL Operators

While working with databases, we use SQL queries to manipulate the data and retrieve the desired result. This manipulation of data is achieved through various **SQL operators**. An operator is a keyword in SQL that helps us access the data and returns the result based on the operator's functionality. SQL provides us with many such operators to ease the process of data manipulation. In this subsection, we will look at three major types of operators in SQL.

Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations such as addition, subtraction, division, and multiplication. These operators usually accept numeric operands.

Operator	Operation	Description
+	Addition	Adds operands on either side of the operator
-	Subtraction	Subtracts the right-hand operand from the left-hand operand
*	Multiplication	Multiplies the values on each side
/	Division	Divides left-hand operand by right-hand operand
%	Modulus	Divides left-hand operand by right-hand operand and returns the remainder

Comparison Operators

Comparison operators in SQL are used to check the equality of two expressions. It checks whether one expression is identical to another. Comparison operators are generally used in the **WHERE clause** of a SQL query. The result of a comparison operation may be TRUE, FALSE, or UNKNOWN. When one or both the expression is NULL, then the operator returns UNKNOWN.

Operator	Operation	Description
=	Equal to	Checks if both operands have equal value, if yes, then returns TRUE
>	Greater than	Checks if the value of the left-hand operand is greater than the right-hand operand or not
<	Less than	Returns TRUE if the value of the left-hand operand is less than the value of the right-hand operand
>=	Greater than or equal to	It checks if the value of the left-hand operand is greater than or equal to the value of the right-hand operand, if yes, then returns TRUE
<=	Less than or equal to	Examines if the value of the left-hand operator is less than or equal to the right-hand operand
<> or !=	Not equal to	Checks if values on either side of the operator are equal or not. Returns TRUE if values are not equal

Logical Operators

Logical operators are those operators that take two expressions as operands and return TRUE or False as output.

Operator	Description
ALL	Compares a value to all other values in a set
AND	Returns the records if all the conditions separated by AND are TRUE
ANY	Compares a specific value to any other values in a set
SOME	Compares a value to each value in a set. It is similar to ANY operator
LIKE	It returns the rows for which the operand matches a specific pattern
IN	Used to compare a value to a specified value in a list
BETWEEN	Returns the rows for which the value lies between the mentioned range
NOT	Used to reverse the output of any logical operator
EXISTS	Used to search a row in a specified table in the database
OR	Returns the records for which any of the conditions separated by OR is true
NULL	Returns the rows where the operand is NULL

The logical and comparison operators are usually used with conditional queries in which we can specify a conditional expression in a **SELECT statement WHERE clause** which specifies that only those rows for which the conditional expression is true are to be retrieved. The syntax for the **SELECT statement** containing the **WHERE clause** is as follows.

```
select_statement:
        SELECT ... FROM table_name
```

```
            WHERE conditional_expression
```

The following is a simple example. We want to create a subset such that `species_id` equals “DM” and `weight > 0`.

```
SELECT *
FROM surveys
WHERE species_id LIKE "DM" AND weight > 0;
```

3.2 Subsetting and Duplicating Data

1. Extract year, month, and day from `surveys` table

```
SELECT
  surveys.year, surveys.month, surveys.Day
FROM
 surveys   -- pointer is not needed since it is in the database
WHERE
  surveys.species_id IN ('NL', 'DM') AND
  surveys.sex = 'M'
```

The code chunk defined the output of the query as an R data frame (*Caution: not a relational table stored in the SQLite database*) with the name **YMD**. We check the first few records from the data frame

```
head(YMD)
```

```
  year month day
1 1977     7  16
2 1977     7  16
3 1977     7  16
4 1977     7  16
5 1977     7  16
6 1977     7  16
```

If we simply create a data view without saving it as an R data frame, we simply ignore the option `output.var="YMD"` in the **SQL code chunk**.

```
SELECT
  surveys.year, surveys.month, surveys.Day
FROM
 surveys  -- pointer is not needed since it is in the database
WHERE
  surveys.species_id IN ('NL', 'DM') AND
  surveys.sex = 'M'
```

Table 4: Displaying records 1 - 10

year	month	day
1977	7	16
1977	7	16
1977	7	16
1977	7	16
1977	7	16
1977	7	16
1977	7	16
1977	7	17
1977	7	17

year	month	day
1977	7	17

2. Duplicate data and rename it

```
SELECT
  surveys.*
FROM
 surveys
```

Note that **surveys** is a relation data table in the SQLite database and duplicated data is not in the SQLite database but an R data frame in the working directory.

Caution: in the SQL code chunk, query statements only work with SQL database. They don't work for R data frames in the working directory. The following code doesn't work because the data set `SurveyCopy` is not in the SQLite database.

```
SELECT
  SurveyCopy.*
FROM
 SurveyCopy
```

In other words, if we want to query the data table (data frame) `SurveyCopy`, we need to add it to the database `con` defined earlier.

```
# Store sample data in the database
dbWriteTable(conn = con,
             name = "SurveyCopy",
             value = SurveyCopy)
## Remove SurveyCopy in the working directory
rm(SurveyCopy)
```

We can use `tbl()` to view the newly added table in the SQLite database `con`.

```
tbl(src = con, c("SurveyCopy"))
```

```
# Source:   table<SurveyCopy> [?? x 10]
# Database: sqlite 3.45.2 [:memory:]
       X record_id month   day  year plot_id species_id sex   hindfoot_length
   <int>     <int> <int> <int> <int>   <int> <chr>      <chr>           <int>
 1     1         1     7    16  1977       2 NL         M                  32
 2     2         2     7    16  1977       3 NL         M                  33
 3     3         3     7    16  1977       2 DM         F                  37
 4     4         4     7    16  1977       7 DM         M                  36
 5     5         5     7    16  1977       3 DM         M                  35
 6     6         6     7    16  1977       1 PF         M                  14
 7     7         7     7    16  1977       2 PE         F                  NA
 8     8         8     7    16  1977       1 DM         M                  37
 9     9         9     7    16  1977       1 DM         F                  34
10    10        10     7    16  1977       6 PF         F                  20
# i more rows
# i 1 more variable: weight <int>
```

In R, we can use `dbListTables()` to view relational tables in the database.

```
dbListTables(con)
```

```
[1] "SurveyCopy" "plots"      "species"    "surveys"
```

We now can query the relational table `SurveyCopy` in the database `con` using SQL clause as usual.

3. Create a table view (i.e., no data set will be created and saved)

```
SELECT
  surveys.year, surveys.month, surveys.Day
FROM
 surveys
WHERE
  surveys.species_id = 'NL' AND
  surveys.sex = 'M'
```

3.3 Define A New Variable

1. Define a new variable with simple arithmetic operations

```
SELECT
    surveys.plot_id,
    surveys.species_id,
    surveys.sex,
    surveys.weight,
    surveys.weight/100 AS wt_kilo   -- should not the pointer in front of
                                    -- the name of the new variable
FROM
   surveys
```

2. Define new variables using string functions in SQL

```
SELECT surveys.*,
       surveys.species_id||'-'||surveys.sex AS newKey
FROM surveys
```

3. Define new variables with aggregated information

```
SELECT surveys.species_id,
       COUNT(surveys.species_id) AS species_ctr
FROM surveys
GROUP BY surveys.species_id
HAVING species_ctr > 10
```

3.4 Sorting Variables

1. Sort data based on the summarized statistics of a variable

Summary functions are restricted to the SELECT and HAVING clauses only;

```
SELECT surveys.species_id
FROM surveys
GROUP BY surveys.species_id
ORDER BY COUNT(surveys.species_id);
```

2. Sort data based on a new variable defined using summarized statistics of a variable.

```
/* create a table view*/
SELECT surveys.species_id AS subtotal,
       COUNT(*)
FROM surveys
GROUP BY surveys.species_id
ORDER BY subtotal;
```

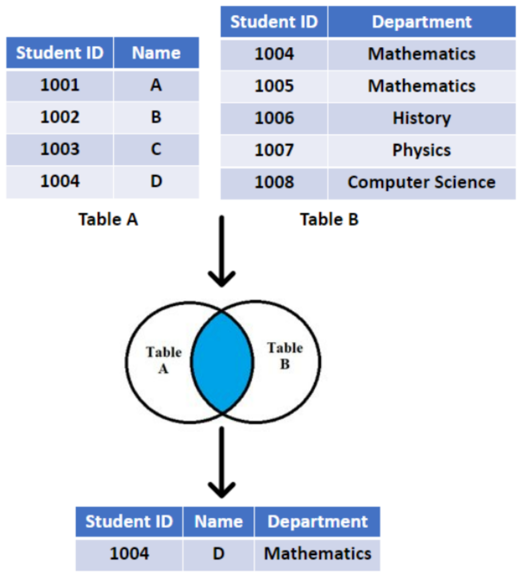
4 SQL Joins

SQL JOIN clause is used to query and access data from multiple tables by establishing logical relationships between them. It can access data from multiple tables simultaneously using common key values shared across different tables. This section briefly introduces commonly used join operations to merge tables using the common key(s). In each of the major `join` operations, we use a visual illustration followed by an example.

1. Inner Join

Inner joins combine records from two tables whenever there are matching values in a field common to both tables.

```
include_graphics("img/inner-join.png")
```



Student ID	Name
1001	A
1002	B
1003	C
1004	D

Table A

Student ID	Department
1004	Mathematics
1005	Mathematics
1006	History
1007	Physics
1008	Computer Science

Table B

Student ID	Name	Department
1004	D	Mathematics

Figure 2: Illustration of inner join.

```
SELECT *
FROM surveys AS A
INNER JOIN species AS B    -- by default, JOIN means INNER JOIN
ON A.species_id = B.species_id;
```

2. Left Join

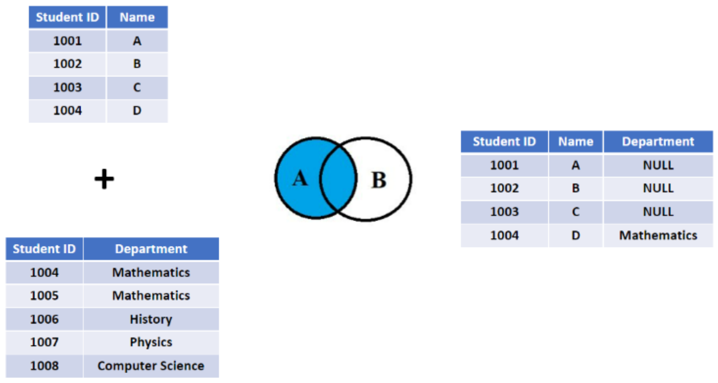
Left Join or Left Outer Join in SQL combines two or more tables, where the first table is returned wholly; but, only the matching record(s) are retrieved from the consequent tables. If zero (0) records are matched in the consequent tables, the join will still return a row in the result, but with NULL in each column from the right table.

```
include_graphics("img/left-join.png")
```

```
SELECT *
FROM surveys AS A
LEFT JOIN species AS B
ON A.species_id = B.species_id;
```

3. Right Join

The Right Join query in SQL returns all rows from the right table, even if there are no matches in the left table. In short, a right join returns all the values from the right table, plus matched values from the left table



Student ID	Name
1001	A
1002	B
1003	C
1004	D

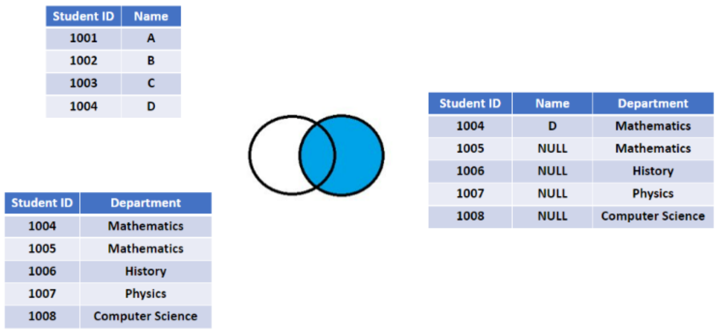
Student ID	Department
1004	Mathematics
1005	Mathematics
1006	History
1007	Physics
1008	Computer Science

Student ID	Name	Department
1001	A	NULL
1002	B	NULL
1003	C	NULL
1004	D	Mathematics

Figure 3: Illustration of left join.

or NULL in case of no matching join predicate.

```
include_graphics("img/right-join.png")
```



Student ID	Name
1001	A
1002	B
1003	C
1004	D

Student ID	Department
1004	Mathematics
1005	Mathematics
1006	History
1007	Physics
1008	Computer Science

Student ID	Name	Department
1004	D	Mathematics
1005	NULL	Mathematics
1006	NULL	History
1007	NULL	Physics
1008	NULL	Computer Science

Figure 4: Illustration of right join.

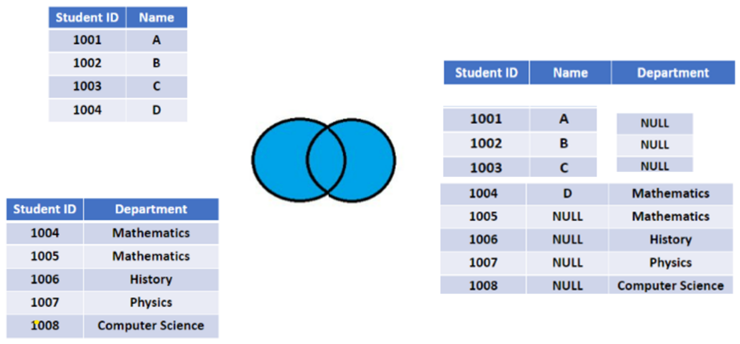
```sql
SELECT *
FROM surveys AS A
RIGHT JOIN species AS B
ON A.species_id = B.species_id;
```

4. Full Join

SQL Full Join creates a new table by joining two tables as a whole. The joined table contains all records from both tables and fills NULL values for missing matches on either side. In short, full join is a type of outer join that combines the resulting sets of both left and right joins.

```
include_graphics("img/full-join.png")
```

```sql
SELECT *
FROM surveys AS A
FULL JOIN species AS B
ON A.species_id = B.species_id;
```



Student ID	Name
1001	A
1002	B
1003	C
1004	D

Student ID	Department
1004	Mathematics
1005	Mathematics
1006	History
1007	Physics
1008	Computer Science

Student ID	Name	Department
1001	A	NULL
1002	B	NULL
1003	C	NULL
1004	D	Mathematics
1005	NULL	Mathematics
1006	NULL	History
1007	NULL	Physics
1008	NULL	Computer Science

Figure 5: Illustration of full join.

5 SQL Aggregation Functions

SQL is a good starting point for high-level data analysis. Many data analysis packages and languages have their own interface to read data from different SQL-based database systems. In fact, any real-life data analysis starts from an RDBMS, and the basic analysis and report generation is done on the SQL platform of that RDBMS itself. A good preview of data within the RDBMS platform itself helps analysts to get a fast high-level analysis in a different platform. The most commonly used aggregation functions include **MAX(), MIN(), AVG(),SUM()** and **COUNT()**. The following are basic examples that involve some of these aggregation functions.

1. Average

```sql
SELECT A.species_id,
       A.sex,
       AVG(A.weight) as mean_wgt
FROM surveys AS A
JOIN species AS B
ON A.species_id = B.species_id
WHERE taxa = 'Rodent' AND A.sex IS NOT NULL
GROUP BY A.species_id, A.sex;    -- sorted by two variables
```

2. Sample size

```sql
SELECT COUNT(*)
FROM surveys
```

6 Subqueries

Subqueries (also known as inner queries or nested queries) are a tool for performing operations in multiple steps. For example, if you wanted to take the sums of several columns, and then average all of those values, you'd need to do each aggregation in a distinct step. Subqueries can be used in several places within a query.

6.1 Subquery in SELECT Clause

In the following example, we want to define a new attribute, the relative percentage of taxonomic groups in the data set in the relational table `surveys`. The feature `taxa` is not in the `surveys` table. We need to join tables `surveys` and `species` to obtain the distribution of `taxa` in the `surveys` table. To this end, we need to know the frequency of each taxonomic group and the size of the relational table `surveys`.

We could use the following queries to find the sample size and the sizes of each taxon group.

1. **Total Sample Size**

```
SELECT COUNT(*) FROM surveys
```

2. **Calculating Taxon Group Frequencies**

The group totals are calculated in the following code.

```
SELECT B.taxa,
       COUNT(*)
FROM surveys AS A  -- A is an alias of `surveys` table
INNER JOIN species AS B -- inner join
ON A.species_id = B.species_id
-- this finds the group frequencies
GROUP BY taxa;
```

This is not efficient. We can use the following nested query to efficiently find the relative frequency table.

```
SELECT B.taxa,
       100.0*COUNT(*)/(SELECT COUNT(*) FROM surveys)  AS Percentage
FROM surveys AS A
JOIN species AS B
ON A.species_id = B.species_id
GROUP BY taxa;
```

The above query produces the following relative frequency table.

```
kable(subSQLinSELECT)
```

taxa	Percentage
Bird	1.2658584
Rabbit	0.2109764
Reptile	0.0393823
Rodent	96.3374497

6.2 Subquery in FROM Clause

As an example, we create a subset with features `record_id`, `year`, `plot_id`, `species_id`, `sex`, `hindfoot_length`, `weight` with the condition that `species_id = "DM"` and `sex = "M"`. The following query that contains a sub-query can do the trick.

```
SELECT sub_survey.record_id,
       sub_survey.year,
       sub_survey.plot_id,
       sub_survey.species_id,
       sub_survey.sex,
       sub_survey.hindfoot_length,
       sub_survey.weight
  FROM (
        SELECT *
         FROM surveys
         WHERE species_id = 'DM'
       ) sub_survey  -- the name of a relational table defined by the subquery
 WHERE sub_survey.sex = 'M'
```

The above subset can also be generated using the following simple query.

```
SELECT record_id,
       year,
       plot_id,
       species_id,
       sex,
       hindfoot_length,
       weight
  FROM surveys
 WHERE species_id = 'DM' AND sex = 'M';
```

6.3 Subquery in WHERE Clause

This example shows the way to create a subset of surveys `record_id, plot_id, species_id, year, sex, hindfoot_length, weight` for the earliest year.

```
SELECT record_id,
       plot_id,
       species_id,
       year,
       sex,
       hindfoot_length,
       weight
  FROM surveys
 WHERE year = (SELECT MIN(year)
                FROM surveys
              )
```

6.4 Subquery in JOIN Clause

In the following example, we use the subquery to find the frequency count in each species group from the `surveys` table and add the frequency to the `species` table.

```
SELECT *
  FROM species
  JOIN ( SELECT COUNT(species_id) AS speciesfreq,
                species_id
           FROM surveys
           GROUP BY species_id
       ) subq
    ON species.species_id = subq.species_id
 ORDER BY subq.species_id DESC
```

The resulting table is shown in the following.

```
kable(subSQLinJOIN)
```

species_id	genus	species	taxa	speciesfreq	species_id
ZL	Zonotrichia	leucophrys	Bird	2	ZL
US	Sparrow	sp.	Bird	4	US
UR	Rodent	sp.	Rodent	10	UR
UP	Pipilo	sp.	Bird	8	UP
UL	Lizard	sp.	Reptile	4	UL
SU	Sceloporus	undulatus	Reptile	5	SU
ST	Spermophilus	tereticaudus	Rodent	1	ST
SS	Spermophilus	spilosoma	Rodent	248	SS

species_id	genus	species	taxa	speciesfreq	species_id
SO	Sigmodon	ochrognathus	Rodent	43	SO
SH	Sigmodon	hispidus	Rodent	147	SH
SF	Sigmodon	fulviventer	Rodent	43	SF
SC	Sceloporus	clarki	Reptile	1	SC
SA	Sylvilagus	audubonii	Rabbit	75	SA
RX	Reithrodontomys	sp.	Rodent	2	RX
RO	Reithrodontomys	montanus	Rodent	8	RO
RM	Reithrodontomys	megalotis	Rodent	2609	RM
RF	Reithrodontomys	fulvescens	Rodent	75	RF
PX	Chaetodipus	sp.	Rodent	6	PX
PU	Pipilo	fuscus	Bird	5	PU
PP	Chaetodipus	penicillatus	Rodent	3123	PP
PM	Peromyscus	maniculatus	Rodent	899	PM
PL	Peromyscus	leucopus	Rodent	36	PL
PI	Chaetodipus	intermedius	Rodent	9	PI
PH	Perognathus	hispidus	Rodent	32	PH
PG	Pooecetes	gramineus	Bird	8	PG
PF	Perognathus	flavus	Rodent	1597	PF
PE	Peromyscus	eremicus	Rodent	1299	PE
PC	Pipilo	chlorurus	Bird	39	PC
PB	Chaetodipus	baileyi	Rodent	2891	PB
OX	Onychomys	sp.	Rodent	12	OX
OT	Onychomys	torridus	Rodent	2249	OT
OL	Onychomys	leucogaster	Rodent	1006	OL
NL	Neotoma	albigula	Rodent	1252	NL
DX	Dipodomys	sp.	Rodent	40	DX
DS	Dipodomys	spectabilis	Rodent	2504	DS
DO	Dipodomys	ordii	Rodent	3027	DO
DM	Dipodomys	merriami	Rodent	10596	DM
CV	Crotalus	viridis	Reptile	1	CV
CU	Cnemidophorus	uniparens	Reptile	1	CU
CT	Cnemidophorus	tigris	Reptile	1	CT
CS	Crotalus	scutalatus	Reptile	1	CS
CQ	Callipepla	squamata	Bird	16	CQ
CM	Calamospiza	melanocorys	Bird	13	CM
CB	Campylorhynchus	brunneicapillus	Bird	50	CB
BA	Baiomys	taylori	Rodent	46	BA
AS	Ammodramus	savannarum	Bird	2	AS
AH	Ammospermophilus	harrisi	Rodent	437	AH
AB	Amphispiza	bilineata	Bird	303	AB

7 Concluding Remarks

We have introduced the basic SQL clauses, the basic join operations, and the sub-queries in this note. But it is by no means considered a complete tutorial for SQL programming. This note intends to help you get started with the basic SQL coding outside a DBMS and expand your technical Vocabulary so you can talk confidently with data professionals in the future.

Since we don't have DBMS to practice SQL, some of the advanced SQL tasks such as control flow, user-defined SQL functions, etc. can not be performed using PROC SQL in SAS and R libraries. You can learn these advanced techniques quickly once you are comfortable with the basic concepts covered in this note.