

SQL and R

Unit V

Data Analysis Using R Programming

SQL

- SQL stands for Structured Query Language.
- SQL queries are used for interacting with a database.
- Using SQL queries we can access and manipulate data stored in the database.
- With the help of SQL queries, we can create, read, update and delete data on databases and perform lots more operations on the database.

C R U D

SQL : R Database Interface (DBI)

- The DBI package helps connecting R to database management systems (DBMS).
- DBI separates the connectivity to the DBMS into a “front-end” and a “back-end”.
- The interface defines a small set of classes and methods similar in spirit to Perl's DBI, Java's JDBC, Python's DB-API, and Microsoft's ODBC. It supports the following operations:
 - connect/disconnect to the DBMS
 - create and execute statements in the DBMS
 - extract results/output from statements
 - error/exception handling
 - information (meta-data) from database objects
 - transaction management (optional)

SQL : R Database Interface (DBI)

- Most users who want to access a database do not need to install DBI directly. It will be installed automatically when you install one of the database backends:
- RPostgres for PostgreSQL,
- RMariaDB for MariaDB or MySQL,
- RSQLite for SQLite,
- odbc for databases that you can access via ODBC,
- bigrquery,

R Database Interface (DBI) Installation

- You can install the released version of DBI from [CRAN](https://cran.r-project.org/web/packages/DBI/index.html) with:

```
install.packages("DBI")
```

The following example illustrates some of the DBI capabilities:

✓ #Connection using DBI

✓ library(DBI)

✓ # Create an ephemeral in-memory RSQLite database

✓ con <- dbConnect(RSQLite::SQLite(), dbname = ":memory:")

✓ # Listing of tables in DB

✓ dbListTables(con)

✓ # Create of tables in DB

✓ dbWriteTable(con, "mtcars", mtcars)

✓ dbListTables(con)

✓ OUTPUT

✓ [1] "mtcars"

```
dbListFields(con, "mtcars")
```

```
[1] "mpg" "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am" "gear" "carb"
```

```
dbReadTable(con, "mtcars")
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
1	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
2	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
3	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
4	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
5	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
6	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
7	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
8	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2

You can fetch all results:

saw
`res <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")`

`dbFetch(res)`

```
#>      mpg  cyl  disp  hp drat   wt  qsec vs  am  gear  carb
#> 1  22.8    4  108.0  93  3.85  2.320 18.61  1  1    4    1
#> 2  24.4    4  146.7  62  3.69  3.190 20.00  1  0    4    2
#> 3  22.8    4  140.8  95  3.92  3.150 22.90  1  0    4    2
#> 4  32.4    4   78.7  66  4.08  2.200 19.47  1  1    4    1
#> 5  30.4    4   75.7  52  4.93  1.615 18.52  1  1    4    2
#> 6  33.9    4   71.1  65  4.22  1.835 19.90  1  1    4    1
#> 7  21.5    4  120.1  97  3.70  2.465 20.01  1  0    3    1
#> 8  27.3    4   79.0  66  4.08  1.935 18.90  1  1    4    1
#> 9  26.0    4  120.3  91  4.43  2.140 16.70  0  1    5    2
#> [ reached 'max' / getOption("max.print") -- omitted 2 rows ]
```

`dbClearResult(res)`

Or a chunk at a time

```
res <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")
```

```
while (!dbHasCompleted(res)) {
```

```
  chunk <- dbFetch(res, n = 5)
```

```
  print(nrow(chunk))
```

```
}
```

```
#> [1] 5
```

```
#> [1] 5
```

```
#> [1] 1
```

```
dbClearResult(res)
```

```
dbDisconnect(con)
```

sqldf package

- We can also use sqldf package to run SQL queries in R.
- Sqldf is a convenient R tool that allows the execution of SQL operations on R data frames.
- The databases MySQL, PostgreSQL, H2, and SQLite can all be used with sqldf.
- We can run SQL queries in R using sqldf package.



```
install.packages("sqldf")
```

sqldf package

- Now after the installation of sqldf package we have to import it to use its functionalities.



library(sqldf)

Reading data using SQL query

- we are going to use the **SQL select command** to view the data of the data frame.
- First, we import the sqldf library, then read the CSV file and store it into a variable “df” as a data frame.
- Select Statement



```
df <- iris
```

```
head(df)
```

Reading data using SQL query

Importing sqldf library

library(sqldf)

#Creating Dataframe

df <- iris

Reading data from data frame using SQL select query

sqldf("select * from df")

all rows

Example

- In this example, we are going to read a particular column from the data frame by selecting a particular column using the SQL query

`sqldf("select [sepal.length], [sepal.width], [Species] from df")`

- we are using order by clause in SQL select statement due to which our data is displayed by sorting the data of the "sepal_length" column

`sqldf("select * from df order by sepal_length")`

Example

 `sqldf("select max(sepal_length) from df")`

 `sqldf("select min(sepal_length) from df")`

WHERE

- Conditional statements can be added via WHERE

 `sqldf("select * from df where species='Iris-virginica')"`

WHERE

- Both **AND** and **OR** are valid, along with paranthese to affect order of operations.

sqldf('SELECT * FROM rock **WHERE** (peri > 5000 **AND** shape < .05) **OR**
perm > 1000')

Example

Deleting rows where species is Iris-virginica

```
df<-sqldf(c("delete from df where species='Iris-  
virginica'", "select * from df"))
```

```
print("After delete rows")
```

Displaying data frame

```
sqldf("select * from df")
```

Example

```
# Update species name from Iris-versicolor to versicolor
print("Before update")
sqldf("select * from df")
# Updating values
df <- sqldf(c("update df set species='versicolor'
              where species='Iris-versicolor'",
              "select * from df"))
print("After update")
sqldf("select * from df")
```

IN

- `%in%`, is used to check if an element exists within a vector, list, or other data structure.
- It returns a logical vector indicating whether a match was found

- # Check value in a Vector

- `67 %in% c(2,5,8,23,67,34)` FALSE

#Check one vector elements in another vector

- `vec1 <- c(2,5,8,23,67,34)`

- `vec2 <- c(1,2,8,34)`

- `vec2 %in% vec1`

IN

```
> # Sequence of characters
```

```
> x <- LETTERS[5:10]
```

```
> y <- LETTERS[2:7]
```

```
> y %in% x
```

```
[1] FALSE FALSE FALSE TRUE TRUE TRUE
```

IN

```
> ## Check if any value from vector present in another vector
```

```
> x <- 1:10
```

```
> y <- 5:20
```

```
> any(x %in% y)
```

```
[1] TRUE
```

IN

```
> ## check if all values from vector present in another vector
```

```
> x <- 1:5
```

```
> y <- 1:20
```

```
> all(x %in% y)
```

```
[1] TRUE
```

IN

```
> # Check values from one vector present in another vector
> # Return Index
> a <- c('A', 'B', 'C', 'D', 'E')
> b <- c('C', 'D')
> which(a %in% b)
[1] 3 4
```

1 based indexing

WHERE --IN

- WHERE IN is used similar to R's %in%. It also supports NOT

```
> sqldf("SELECT * FROM BOD WHERE Time in (1,7)")
```

	Time	demand
--	------	--------

1	1	8.3
---	---	-----

2	7	19.8
---	---	------

```
> sqldf("SELECT * FROM BOD WHERE Time not in (1,7)")
```

	Time	demand
--	------	--------

1	2	10.3
---	---	------

2	3	19.0
---	---	------

3	4	16.0
---	---	------

4	5	15.6
---	---	------

WHERE --LIKE

- LIKE can be thought of as a weak regular expression command.
- It only allows the single wildcard % which matches any number of characters.
- For example, to extract the data where the feed ends and does not end with "bean"



```
sqldf('SELECT * FROM chickwts WHERE feed LIKE "%bean" LIMIT 5')
```

The image shows a SQL query on a blue background. A green highlight is under the word 'WHERE'. Blue hand-drawn scribbles are over the words 'WHERE', 'feed', and 'LIKE'. Another blue scribble is on the left side of the query.



```
sqldf('SELECT * FROM chickwts WHERE feed NOT LIKE "%bean" LIMIT 5')
```

The image shows a SQL query on a blue background. Blue hand-drawn scribbles are over the words 'WHERE', 'feed', 'NOT', and 'LIKE'. Another blue scribble is on the left side of the query.

Sorting the data using SQL query (Order by Clause)

- To order variables, use the syntax

`ORDER BY var1 {ASC/DESC}, var2 {ASC/DESC}`

`sqlidf("SELECT * FROM Orange ORDER BY age ASC, circumference DESC
LIMIT 5")`

LIMIT

- To control the number of results returned, use LIMIT #.

 `sqldf('SELECT * FROM iris LIMIT 10')`

Aggregated data

- Select statements can create aggregated data using AVG, MEDIAN, MAX, MIN, and SUM as functions in the list of variables to select.
- The GROUP BY statement can be added to aggregate by groups.



```
sqlldr("SELECT AVG(circumference) FROM Orange")
```



```
sqlldr("SELECT max(circumference) AND min(circumference) FROM Orange")
```



```
sqlldr("SELECT tree, AVG(circumference) AS meancirc FROM Orange GROUP BY tree")
```