WST 212 - Databases

University of Pretoria
Faculty of Natural and Agricultural Sciences
Department of Statistics

Presented by: Iena Derks

1 Databases

In this section we will focus on data storage, management, and information retrieval.

1.1 Required packages

The following packages are required for the database section of WST 212:

```
library(sqldf)
library(lubridate)
library(RH2)
```

Please ensure these packages are installed and load successfully.

Reminder – You can install packages using the following:

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

1.2 Introduction to Databases

1.2.1 Learning Objectives:

You must be able to:

- Define a database.
- Discuss the importance of databases.

- Define a relational database.
- List the advantages of using a database management system.
- Define terminology used in relational databases.
- List reasons for data collected not being in usable formats.

Advances in technology have contributed to the generation of large-scale data in nearly all fields of science, giving rise to challenges concerned with storing and manipulating data. Data sets can be stored in various formats using different software as well as relational databases. During your studies, you will most likely be exposed to smaller data sets that easily fit into your computer's memory. However, most real-world applications make use of data sets that are too large for your computer to handle (think here of a data set containing tweets/social media posts). In cases like these, we would like to store data outside of R and organising it in a database is helpful. By storing data in a database, we can retrieve only those parts or chunks required for the analysis – and not the entire data set.

The first universal Data Base Management System (DBMS) was called the Integrated Data Store. In the late 1960s, International Business Machines (IBM) developed the Information Management System (IMS) DBMS used even today in many major installations. In 1970 a new data representation framework called the relational data model was proposed. The benefits were widely recognised, and the use of DBMSs for managing corporate data became standard practice.

The Structured Query Language (SQL) query language for relational databases, developed as part of IBMs System R project, is now the standard query language. Specialised systems have been developed by numerous organisations for creating data warehouses, consolidating data from several databases, and for carrying out specialised analysis.

Why do we need DBMS?

Consider the scenario where a company has a large collection of data on employees, departments, products, sales, etc. Several employees access the data simultaneously. Questions about the data must be answered quickly, changes made to the data by different users must be applied consistently, and access to certain parts of the data must be restricted. We can try to manage the data by storing it in operating system files.

What drawbacks would this approach have?

By storing data in a DBMS rather than a collection of operating system files, we can use the DBMS's features to manage the data in a robust and efficient manner.

Advantages of using a DBMS to manage data:

- Data Independence
- Efficient Data Access
- Data Integrity and Security
- Data Administration
- Concurrent Access and Crash Recovery
- Reduced Application Development Time

Data must be stored in such a manner that allows quick queries of the data, sorting of the data and manipulating of data in several ways to effectively and efficiently analyse the data and ensure data security.
These systems (DBMS) are important links in the creation and management of data.

- A **database** is defined as "a structured set of data held in a computer, especially one that is accessible in various ways".
- In a **relational database**, each relation is a set of rows. Each row is a list of attributes, which represents a single item in the database. Each row in a relation (table) shares the same attributes (columns) where each attribute has a well-defined data type.
- **Structural Query Language (SQL)** is a programming language used to manipulate and retrieve data in a relational database.

A large part of statistical analysis consists of first getting the data into a form that will allow effective analysis. The two main reasons why data will not already be in a usable format is:

- Large data sets consists of so many records and characteristics, it will not always be efficient to store everything in one table.
- Statistical analysis will often explore the use of more than one data set, with different characteristics and very often different origins. The analyst will need to accurately combine these data sets and transform them into the required form to start the statistical investigation.

We can view a relational database as follows:

- Relational database: Collection of tables (also called relations)
- Table:
  - Collection of rows (also called tuples or records)
  - Each row in a table contains a set of columns (also called fields or attributes)
  - Each column has a type:
    * String: VARCHAR(20)
    * Integer: INTEGER
    * Floating-point: FLOAT, DOUBLE
    * Date or time: DATE, TIME, DATETIME
- Primary key: Provides a unique identifier for each row (need not be present, but usually is in practice)
- Schema: The structure of the database, including for each table,
  - The table name
  - The names and types of its columns
  - Various optional additional information (constraints, etc.)
  - The number of rows in a table is not part of the schema

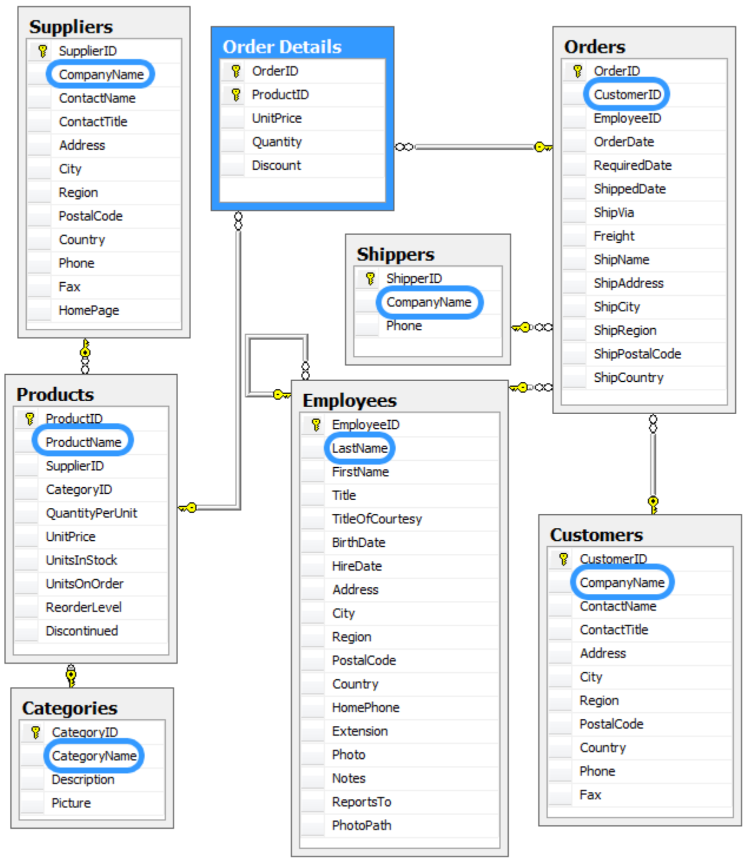

Figure 1: Illustration of a Database

1.3 Basic Queries

1.3.1 Learning Objectives:

You must be able to:

- Identify key syntax of the SQL procedure.
- List and apply key features of the SQL procedure.
- List and apply key features of the SELECT statement.
- List and apply SQL procedure statements.
- Display columns directly from a table.
- Display columns calculated from other columns in a query.
- Calculate columns conditionally using the CASE expression.
- Apply date functions in an SQL query.
- Select a subset of rows in a query.
- Display a query's results in a specific order.
- Apply functions to create summary queries.
- Group data and produce summary statistics for each group.
- Subset a query on summarised values.

1.3.2 Overview of the SQL Procedure

SQL Query Syntax:

Operations in SQL are performed using individual queries without loops or control statements. Although there are various SQL statements (as described on the next slide), the **SELECT** statement is probably one of the most commonly used SQL statements. We use the SELECT statement to return results from the database in question. Although this might seem easy, the syntax of the SELECT statement could be very complex:

```
SELECT object_item <,...object_item>
  FROM <list_of_tables>
  <WHERE sql_expression>
  <GROUP BY object_item <,...object_item>>
  <HAVING sql_expression>
  <ORDER BY order_by_item <DESC> <,...order_by_item>>
```

Most of the clauses in the SELECT statement are optional – indicated here by <. . . >.

SQL Statements:

- SELECT: Identifies columns to be selected.
- CREATE: Build new tables, views, or indexes.
- DESCRIBE: Displays table attributes or view definitions.
- INSERT: Adds rows of data to tables.

- RESET: Adds to or changes SQL options without revoking the procedure.

Clauses for the SELECT Statement:

- FROM: Specifies data source.
- WHERE: Specifies data that meets certain conditions.
- GROUP BY: Groups data into categories.
- HAVING: Specifies groups that meet certain conditions.
- ORDER BY: Specifies an order for the data.

The *sqldf* package in R

- The *sqldf* package allows users to run SQL statements within R.
- The *sqldf* package creates an opportunity to work directly with SQL statements on an R data frame.
- Typically passed a single argument which is an SQL SELECT statement where the tables are R data frames.
- Behind the scenes:
  - Database is automatically created
  - Data frames are automatically loaded into the database
  - The specified SQL statement is performed
  - Result is read back into R
  - Database is deleted
- Syntax is the same regardless of the software used to implement the query.
- Slight variations may occur, however, the overall structure remains the same.

To implement this in R, we first need to "call" the *sqldf* package and then "wrap" the procedure in the *sqldf()* function:

```
library(sqldf)

sqldf(SELECT object_item
        FROM table)
```

1.3.3 Specifying columns

1.3.3.1 Querying columns in a table If we would like to retrieve all of the data contained in a given table, we could use the asterisk (*) in the SELECT statement:

```
SELECT *
  FROM table
```

Alternatively, we can specify the variables/expressions in the SELECT statement:

```
SELECT var1, var2
  FROM table
```

Example

Make use of the **crashes** table in the **Highway** database. Find the name of the road being studied, the year in which the observation was taken and the number of crashes on the road during that year.

```
ex <- sqldf("SELECT Road, Year, N_Crashes
              FROM crashes")
```

Alter the above SQL query to provide only the information for crashes in the year 1995 and order from lowest to highest based on the number of crashes.

```
ex <- sqldf("SELECT Road, Year, N_Crashes
              FROM crashes
              WHERE Year = 1995
              ORDER BY N_Crashes")
```

1.3.3.2 Calculated Columns

The SQL procedure allows you to modify the query such that a new column is created, for example, our table consists of var1 and var2 and we are interested in the ratio of var1 and var2:

```
SELECT var1, var2, var2/var1
  FROM table
```

A useful operator in SQL is the **AS** operator, which is used to assign an *alias* to the new column in the result set, for example, if we wanted the alias of the ratio calculation to be "ratio":

```
SELECT var1, var2, var2/var1 AS ratio
  FROM table
```

Note that the use of the AS operator (and as such, the assignment of an alias to a new column) is optional. If an alias is not assigned to the new column in the result set, the column name will simply appear blank.

Example

Make use of the **crashes** table in the **Highway** database. Find the name of the road being studied, the year in which the observation was taken and the number of crashes on the road during that year and a new **Average_Num_Crashes** column which contains the average number of crashes per month for the year for each road.

```
ex <- sqldf("SELECT Road, Year, N_Crashes,
              N_Crashes/12 AS Average_Num_Crashes
              FROM crashes")
```

New columns can also be calculated conditionally using the **CASE** expression. For example, if you have a Bonus column but the percentage should be conditional on the position of the employee then the CASE expression can be used when creating the new calculated column.

Generic syntax for CASE expression:

```
SELECT object_item <,...object_item>
    CASE <case_operand>
      WHEN when_condition THEN result_expression
      <WHEN when_condition THEN result_expression>
      <ELSE result-expression>
    END <AS column>
  FROM <table>
```

Example

Consider a database containing email addresses. Create a new column that groups the domains into different express groups.

- If the email domain belongs to Gmail, Yahoo or Hotmail it should be in group 1
- All other domains should be in group 0

```
ex <- sqldf("SELECT E_MAIL,
              CASE
                WHEN E_MAIL like '%gmail%' THEN 1
                WHEN E_MAIL like '%yahoo%' THEN 1
                WHEN E_MAIL like '%hotmail%' THEN 1
              else 0
              END express
              FROM email_data")
```

1.3.3.3 Working with Dates in SQL Data sets often contain date variables, for example, date of purchase or date of last update. There are several built-in R functions available to process date variables, such as **as.Date()**. However, the *as.Date()* function can only handle dates (and not time). A useful package for working with dates (and times) is the **lubridate** package. This package is a more user-friendly functionality to process date and time (as opposed to other packages). *sqldf* requires date – and time – data to be in the correct format as well as correct class/data type. The *lubridate* package allows us to manipulate date (and time) data.

The following packages are required:

```
library(lubridate)
library(RH2)
```

Example

Consider the **housing** database. The table contains a column **date** which contains the year, month and day (yyyy/mm/dd) information in one column. Create a report which displays the city, sales, volume, month (in a new column labelled **month**) and year (in a new column labelled **year**) .

```
housing$date <- as.Date(housing$date, format = "%Y/%m/%d")

ex <- sqldf("SELECT city, sales, volume,
              month(date) AS month,
              year(date) AS year
              FROM housing")
```

1.3.4 Specifying Rows

1.3.4.1 Selecting a Subset of Rows To set a condition the data must meet before being selected (limiting the rows which are returned), the **WHERE** clause can be used. The WHERE clause can combine comparison operators, logical operators and/or special operators. Only one WHERE clause is required in a SELECT statement to specify multiple subsetting criteria. For example, suppose we would like to extract all columns for the rows of a table where var1 is greater than 15 and var2 is less than var1, then:

```
SELECT *
  FROM table
  WHERE var1 > 15 AND var2 < var1
```

NOTE: A calculated column that has an alias cannot be called in the WHERE clause. In other words, the WHERE clause cannot access the "new" variables created in the SELECT statement.

Example

Make use of the **crashes** table in the **Highway** database. Display a list of roads where the number of crashes exceeds 35.

```
ex <- sqldf("SELECT Road, N_Crashes
              FROM crashes
              WHERE N_Crashes > 35")
```

Example

Modify the report obtained from the **housing.csv** database (ex1f) to only include the details for the year 2001.

```
ex <- sqldf("SELECT city, sales, volume,
              month(date) AS month,
              year(date) AS year
              FROM housing
              WHERE year(date) = '2001'")
```

1.3.5 Ordering Rows

It is often required to sort the results in ascending or descending order, based on one or more columns. In SQL, the **ORDER BY** clause is used to ensure data is brought into R in the appropriate order. For example, suppose we want to sort the results according to var1:

```
SELECT *
  FROM table
  ORDER BY var1
```

- Default order is **ascending**
- The keyword **DESC** must follow the column name in the clause if a **descending** order is required.
- More than one column may be specified in the ORDER BY clause, however, the first column specified will determine the major sort order.

Example

Use the **crashes** table in the **Highway** database to find the name of the road being studied, the year in which the observation was taken and the number of crashes on the road during that year. The report must display the number of crashes from lowest to highest.

```
ex <- sqldf("SELECT Road, Year, N_Crashes
              FROM crashes
              ORDER BY N_Crashes")
```

How would the query change to obtain the same report but with the number of crashes from highest to lowest?

1.3.6 Summarising Data

Often we do not require all of the details contained in a table and instead, we can obtain a summary of the information. Here, the **GROUP BY** clause, in conjunction with aggregation

SQL	Description
AVG	Returns the mean value
COUNT	Returns the number of non-missing values
MAX	Returns the largest value
MIN	Returns the smallest value
SUM	Returns the sum of the values
sum(is.na())	Counts the number of missing values
std	Returns the standard deviation
var	Returns the variance

functions, can be used to produce a data summary from a database. A few of the most commonly used aggregate functions are listed below,

For example, suppose we would like to display the means of *var1*, from our database table, broken down by the *type*:

```
SELECT type, AVG(var1) AS mean
  FROM table
  GROUP BY type
```

Notice here the "grouping variable" is included in the SELECT statement. We need to specify this, as SQL does not do this automatically.

A single query can contain multiple aggregate statistics, for example, if we were to display the number of observations for each *type* (COUNT()) as well as the mean of var1 (AVG()):

```
SELECT type, COUNT(*), AVG(var1) AS mean
  FROM table
  GROUP BY type
```

Example

Make use of the **Orange** database. Write a query that reports the average circumference of the trees.

```
ex <- sqldf("SELECT AVG(circumference)
              FROM Orange")
```

Now, write a query to create a report that calculates the average circumference for each tree type.

```
ex <- sqldf("SELECT tree, AVG(circumference) AS meancirc
              FROM Orange
              GROUP BY tree")
```

How would you modify the query to display only the results for trees with an average circumference larger than 110?

```r
ex <- sqldf("SELECT tree, AVG(circumference) AS meancirc
              FROM Orange
              GROUP BY tree
              HAVING meancirc > 110")
```

1.4 SQL Joins

1.4.1 Learning Objectives:

You must be able to:

- Identify different ways to combine data horizontally from multiple tables.
- Distinguish between inner and outer SQL joins.
- Apply inner joins as follows:
  - Join two or more tables on matching columns.
  - Qualify column names to identify specific columns.
  - Use a table alias to simplify code.
- Apply outer joins as follows:
  - Join two tables on matching columns and include non-matching rows from one table (Apply LEFT join correctly).

Database servers allow us to effectively join together multiple tables, based on common values (of columns) shared within the tables. This is also possible using R functions, however, it may be more efficient to use the database server for "merging" or joining together these tables. In SQL we use **joins** to combine tables **horizontally**. A join involves matching data from one row in one table with a corresponding row in another table. A join can be performed on one or more columns. There are two main types of joins:

- **Inner join**: Returns only matching rows.
- **Outer join**: Returns all rows from one or both the tables being joined (includes matching and non-matching rows.)

1.4.2 Inner Joins

The most common way of joining two tables is through an **INNER JOIN**. With an INNER JOIN, only those observations that have common values of the variable used to "merge" (or join) will be retained in the output. In other words, we use an INNER JOIN when we want to "merge" (or join) two or more tables horizontally and display only the matching rows.

Syntax

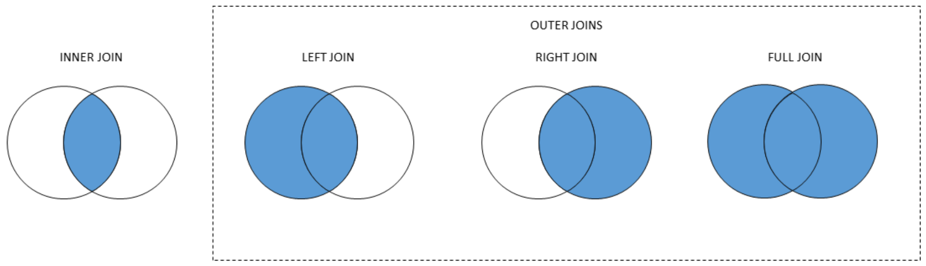

Figure 2: SQL Joins

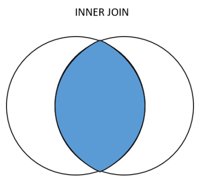

Figure 3: Inner Join

```
SELECT object_item <,...object_item>
  FROM table_name <<AS> alias>
    INNER JOIN table_name <<AS> alias>
  ON join_condition(s)
  <other clauses>
```

Example

Make use of the **crashes** table and the **roads** table from the **Highway** database. Write a report that displays the Year, Road, Number of Crashes, Volume, District and Length for roads that have all the information.

```
ex <- sqldf("SELECT a.*, b.District, b.Length
              FROM crashes AS a
              INNER JOIN roads AS b
              ON a.Road = b.Road")
```

1.4.3 Outer Joins

When using an INNER JOIN, rows that are unmatched from either table are not returned (or displayed) in the final result set, and as such are "lost". **Outer joins** can be used to prevent the loss of data since the content of both tables is integrated and unmatched rows in one or both tables can be returned. In other words, if we want to "merge" (or join) two or more tables horizontally to include all rows an *outer join* is used.

Outer joins are classified into three types:

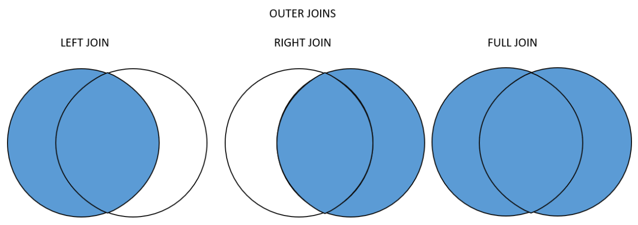

Figure 4: Outer Join

- **Left join**: Returns all matching and non-matching rows from the *left* table and all matching rows from the *right* table.
- **Right join**: Returns all matching and non-matching rows from the *right* table and all matching rows from the *left* table.
- **Full join**: Returns all matching and non-matching rows from both tables.

Syntax

```
SELECT object_item <,...object_item>
  FROM table_name <<AS> alias>
    LEFT|RIGHT|FULL JOIN table_name <<AS> alias>
  ON join_condition(s)
  <other clauses>
```

The **ON** clause specifies the criteria for the outer join.

NOTE: Currently the *sqldf* package does not support RIGHT or FULL outer joins. Hence, the focus of this course will be on INNER and LEFT joins.

Example

Make use of the **crashes** table and the **roads** table from the **Highway** database. Write a report that displays the Year, Road, Number of Crashes, Volume, District and Length for all roads which have had crashes.

```
ex <- sqldf("SELECT a.*, b.District, b.Length
              FROM crashes AS a
              LEFT JOIN roads AS b
              ON a.Road = b.Road")
```

Example with a Double Join

Make use of the **crashes** table, the **roads** table, and the **works** table from the **Highway** database. **Highway** database. You would like a report of the crashes on roads that had or are having roadworks done. The report should include the Year, Road, Number of Crashes, Volume, District and Length.

```
ex <- sqldf("SELECT a.*, b.District, b.Length
              FROM crashes AS a
              INNER JOIN roads AS b
              ON a.Road = b.Road
              INNER JOIN works AS c
              ON a.Road = c.Road")
```

1.5 SQL Subqueries

1.5.1 Learning Objectives:

You must be able to:

- Define a subquery.
- Differentiate between correlated and non-correlated subqueries.
- Subset data based on values returned from a subquery.
- Define an in-line view.
- Create and use in-line views correctly.
- Use in-line views and subqueries to simplify coding a complex query.
- Combine subqueries and joins for complex queries.

As the name implies, a **subquery** is a query that resides within an outer query, in other words, it is nested inside another query. Subqueries are also known as *nested queries*, *inner queries* and *sub-selects*. A subquery must be resolved before the outer query can be resolved. An example of subquery usage is when we want to return all products whose *list_price* is greater than the average *list_price* for all products. Here, you would first calculate the average price and then use this to compare against each product's price:

```
SELECT product_id, name, list_price
  FROM products
  WHERE list_price > (SELECT AVG(list_price)
                        FROM products)
```

A subquery:

- Returns values to be used in the outer query's WHERE or HAVING clause.
- Returns a single column.
- Can return multiple or a single value.

There are two types of subqueries:

- **Non-correlated subquery** : A self-contained query – it executes independently of the outer query.

- **Correlated subquery** : Requires a value (or values) to be passed to it by the outer query before it can be successfully resolved.

NOTE: In this course we will focus on non-correlated subqueries.

1.5.2 Noncorrelated subqueries

Syntax for Noncorrelated Subqueries

```
SELECT object_item <,...object_item>
  FROM table_name <<AS> alias>
  WHERE sql_expression ... (SELECT object_item
                  FROM table)
```

```
SELECT object_item <,...object_item>
  FROM table_name <<AS> alias>
  GROUP BY object_item
  HAVING sql_expression ... (SELECT object_item
                  FROM table)
```

Example

Make use of the **staff** table from the **company** database. This database contains salary and job title information on employees at a specific company. Create a report that shows the job titles with an average salary greater than the average salary of the company.

We can break this query down into three steps:

1. Calculate the average salary of the company.
2. Calculate the average salary for each job title.
3. Compare the two and keep only the job titles for which the average salary is greater than the value calculated in step 1.

```
ex <- sqldf("SELECT Employee_Job_Title, AVG(Salary) AS MeanSalary
              FROM staff
              GROUP BY Employee_Job_Title
              HAVING AVG(Salary) > (SELECT AVG(Salary)
                                     FROM staff)")
```

Note: The subquery is enclosed in brackets and will first execute followed by the outer query.

Example

Consider the **addresses** table and **info** table from the **company** database. The CEO sends a birthday card to each employee having a birthday in February. Create a report listing the names, cities and countries of employees with birthdays in February.

Keyword	Signifies
=ANY(20,30,40)	=20 or =30 or =40
>ANY(20,30,40)	>20
>ALL(20,30,40)	>40
<ALL(20,30,40)	<20

```
info$Employee_Birth_Date <- as.Date(info$Employee_Birth_Date,
                                   format = "%Y/%m/%d")

ex <- sqldf("SELECT Employee_ID, Employee_Name, City, Country
              FROM addresses
              WHERE Employee_ID in (SELECT Employee_ID
                                    FROM info
                                    WHERE month(Employee_Birth_Date) = 2)
              ORDER BY Employee_ID")
```

Correlated Subqueries - Example for Illustration purposes only

Note the code below illustrates a correlated subquery. This query is not stand alone and requires information from the main/outer query to execute.

```
ex <- sqldf("SELECT Employee_ID, AVG(Salary) as MeanSalary
              FROM employee
              where 'AU' = (SELECT Country
                           FROM supervisors
                           WHERE employee$Employee_ID = supervisors$Employee_ID")
```

Subqueries that return multiple values

If a subquery returns multiple values and the "equal" operator is used an error is generated. For subqueries that return multiple values the "IN" operator or a comparison operator with the keywords "ANY" or "ALL" must be used.

The "ANY" keyword is true for any of the values returned by the subquery whereas the "ALL" keyword is true for all the values returned by the subquery.

1.5.3 In-line Views

An in-line view is a query expression (SELECT statement) that resides in a FROM clause. It acts as a temporary table in a query. In-line views are mostly used to simplify complex queries. An in-line view can consist of any valid SQL query, however, it may not contain an ORDER BY clause. An in-line view can be assigned an alias if it were a table.

Hint: always test in-line views independently while building a complex query to avoid errors.

Syntax

```
SELECT object_item <,...object_item>
  FROM table_name <<AS> alias>
  GROUP BY object_item
  HAVING sql_expression ... (SELECT object_item
                  FROM table)
```

Example

Make use of the Company database. List all active Sales Department employees who have annual salaries significantly lower (less than 95%) than the average salary for everyone with the same job title. We can break this query down into the following steps:

1. Calculate the average salaries of all active employees in the Sales department, grouped by their title.
2. Calculate the average salary for each job title.
3. Compare the two and keep only the job titles for which the average salary is greater than the value calculated in step 1.
4. Match each employee to a job title group and compare the employee's salary to the group's average to determine whether it is less than 95.

```
ex <- sqldf("SELECT Job_Title, AVG(Salary) AS Job_Avg
              FROM payroll AS p
              INNER JOIN organisation AS o
              ON p.Employee_ID = o.Employee_ID
              WHERE Employee_Term_date is NULL and Department = 'Sales'
              GROUP BY Job_Title")
```

```
ex <- sqldf("SELECT Employee_Name, emp.Employee_Job_Title,
              Employee_Annual_Salary, Job_Avg
              FROM (SELECT Job_Title, AVG(p.Salary) AS Job_Avg
                     FROM payroll AS p
                     INNER JOIN organisation AS o
                     ON p.Employee_ID = o.Employee_ID
                     WHERE Employee_Term_Date is NULL AND Department = 'Sales'
                     GROUP BY Job_Title) AS job
              INNER JOIN salesstaff AS emp
              ON emp.Employee_Job_Title = job.Job_Title
              WHERE Employee_Annual_Salary < (Job_Avg*0.95)
              ORDER BY Job_Title, Employee_Name")
```