Using SQL Guide

division of
Finance

quick reference guide

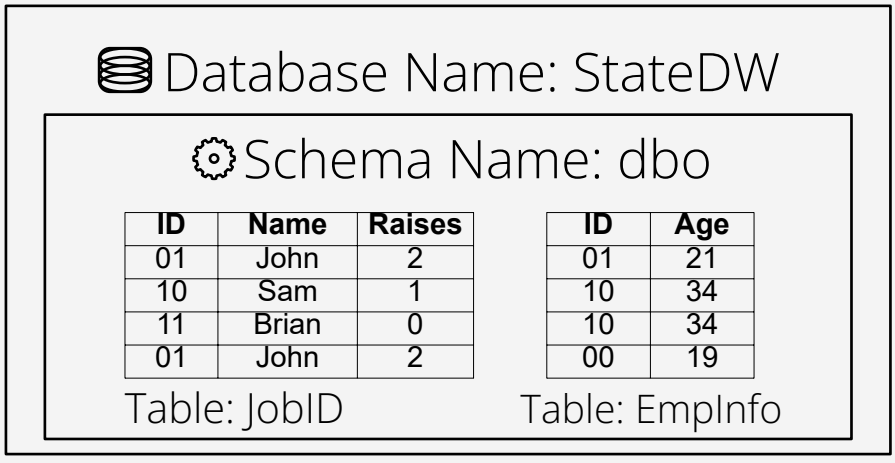
Instructions

You may have heard of others using something called **sequel** or **SQL**.
You might have wondered what it is how to use it. Those are the questions that this QRG are meant to answer.

What is SQL?

SQL stands for **structured query language**; it is the standard language of databases. A **database** is a computer on which data is stored and from which data can be retrieved, usually in the form of **data tables**. Those tables are organized in folders within the database called **schemas**. Each table has rows of data that pertain to a particular **data field**, or table column.
SQL leverages the names of databases, schemas, data tables, and data fields in those tables.

Let's look at a visual representation of a database below:



The database above, named **StateDW**, has a schema called **dbo** that contains two tables, each with particular data fields and data values within those data fields. Here is an example of a SQL query on our database:

```
select Name, Raises
from StateDW.dbo.JobID as jid
where jid.Name='John'
```

This SQL statement follows a specific structure. First, we use the **select** clause to list the data fields or columns we want to return, then we use the **from** clause to specify the database, its schema, and the table in which we can find the data fields mentioned.

We can also give that directory of **database.schema.table** an alias, or a nickname, using the **as** operator. In this case, we named our directory **jid** so that every time we want to refer to this directory, we can simply use that nickname. For example, at the end of our SQL statement we use the **where** clause to list a condition - we want to narrow the results of the query to only the rows where the **Name** column in the **jid** directory is 'John'.

This query would return the following table:

Name	Raises
John	2
John	2

Notice that this is the JobID table, in the dbo schema, in the StateDW database where the Name is **John**. Notice how only the data fields we mentioned were returned, not the ID data field. Also, note how there are two rows in our table that have the same information, just as it existed in the original table.

Let's say that we want to deduplicate our results - we can use the **distinct** command.

Here is how we can deduplicate data using the **distinct** command.

```
select distinct Name, Raises
from StateDW.dbo.JobID as jid
where jid.Name='John'
```

This returns:

Name	Raises
John	2

We can actually perform aggregations over a given group. For example, if wanted to know the average age of our employees. We could run the following SQL statement.

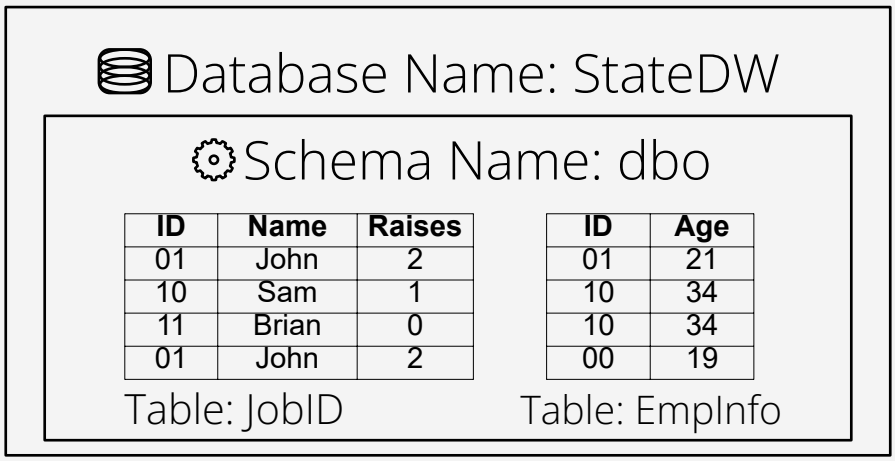
```
select AVG(Age) as  avg_age
from StateDW.dbo.EmpInfo as emp
```

This returns:

avg_age
27

Note how we gave the data field a nickname and how the only value returned is the average over the age column in the EmpInfo table. However, we know that there were two instances of the same employee ID, so our average is over-emphasizing the employee whose ID is **10**. Before we do anything else, let's find out who that employee is.

By looking at the EmpInfo table, we don't know the name of employee 10. When we look at the JobID table, we see that employee 10 is Sam. However, it is not very convenient to have to go back and forth between tables to compare the information between one table and another. We would like all that information to be in one place! For this we can use, a **join**!



There are 4 types of joins - left join, right join, inner join, and outer join. Here is an example of a **left join**.

```
select jid.ID, jid.Name, jid.Raises, emp.ID, emp.Age

from StateDW.dbo.JobID as jid

left join StateDW.dbo.EmpInfo as emp
on jid.ID=emp.ID
```

This returns:

ID	Name	Raises	ID	Age
01	John	2	01	21
10	Sam	1	10	34
10	Sam	1	10	34
11	Brian	0	11	-
01	John	2	01	21

Note that all the information in the table listed on the **from** statement was preserved; this is the **left** table. Notice how there is two instances of **Sam** now because, though it appears on the left table only once, it matches to both instances from the **right** table - this matching is being performed on a common data field, **on** the **ID** data field that both tables contain. Also note how Brian has incomplete information; this is because, though Brian exists in the left table, his employee ID **11** does not exist in the EmpInfo table (the **right table** listed in the **join** statement).

Left Table (in **from** line)

ID	Name	Raises
01	John	2
10	Sam	1
11	Brian	0
01	John	2

Right Table (in **join** line)

ID	Age
01	21
10	34
10	34
00	19

ID	Name	Raises	ID	Age
01	John	2	01	21
10	Sam	1	10	34
10	Sam	1	10	34
11	Brian	0	11	-
01	John	2	01	21

A **right** join simply is the inverse of a **left** join - here is the SQL below:

```
select jid.ID, jid.Name, jid.Raises, emp.ID, emp.Age

from StateDW.dbo.JobID as jid

right join StateDW.dbo.EmpInfo as emp
on jid.ID=emp.ID
```

The right join would return the following:

Left Table (in **from** line)

ID	Name	Raises
01	John	2
10	Sam	1
11	Brian	0
01	John	2

Right Table (in **join** line)

ID	Age
01	21
10	34
10	34
00	19

ID	Name	Raises	ID	Age
01	John	2	01	21
01	John	2	01	21
10	Sam	1	10	34
10	Sam	1	10	34
00	-	-	00	19

Note that there are two instances of **John** because the right table's **ID** 01 matches on two occasions with the left table's. Also note how the **ID** 00 did not find a match when we are doing a right join because, though the ID 00 exists in the right table, it does not exist in the left table, so it is preserved in the final output without any information added from the left table. Lastly, note how ID 11, since it doesn't exist in the right table, though it exists in the left table, was not brought into the final results. This is how the left vs right join works - we keep all the data from the our join side (the right table in a right join or the left table in a left join), we match on every occasion where there is a match between both tables, and we drop any data for which there was no match from the non-join side (the left table in a right join or the right table in a left join).

The last two joins are even easier to understand: an inner join returns only the rows where there is a match between both tables, and an outer join returns all the rows from both tables, matching where possible. On the next page, you will see some examples.

```
select jid.ID, jid.Name, jid.Raises, emp.ID, emp.Age

from StateDW.dbo.JobID as jid

inner join StateDW.dbo.EmpInfo as emp
on jid.ID=emp.ID
```

This join would return the following:

Left Table (in **from** line)

ID	Name	Raises
01	John	2
10	Sam	1
11	Brian	0
01	John	2

Right Table (in **join** line)

ID	Age
01	21
10	34
10	34
00	19

ID	Name	Raises	ID	Age
01	John	2	01	21
01	John	2	01	21
10	Sam	1	10	34
10	Sam	1	10	34

Note that only the rows that were able to find a match on employee ID between the tables were returned - a row for each match.

Now let's do the outer join!

```
select jid.ID, jid.Name, jid.Raises, emp.ID, emp.Age

from StateDW.dbo.JobID as jid

outer join StateDW.dbo.EmpInfo as emp
on jid.ID=emp.ID
```

This join would return the following:

Left Table (in **from** line)

ID	Name	Raises
01	John	2
10	Sam	1
11	Brian	0
01	John	2

Right Table (in **join** line)

ID	Age
01	21
10	34
10	34
00	19

ID	Name	Raises	ID	Age
01	John	2	01	21
10	Sam	1	10	34
10	Sam	1	10	34
11	Brian	0	11	-
01	John	2	01	21
00	-	-	00	19

Note that every possible match was made and that all the data from both tables was preserved!

Since we have duplicate rows, let's use the distinct function on this **outer** join.

```
select distinct jid.ID, jid.Name, jid.Raises, emp.ID, emp.Age

from StateDW.dbo.JobID as jid

outer join StateDW.dbo.EmpInfo as emp
on jid.ID=emp.ID
```

This join would return the following:

Left Table (in **from** line)

ID	Name	Raises
01	John	2
10	Sam	1
11	Brian	0
01	John	2

Right Table (in **join** line)

ID	Age
01	21
10	34
10	34
00	19

ID	Name	Raises	ID	Age
01	John	2	01	21
10	Sam	1	10	34
11	Brian	0	11	-
00	-	-	00	19

Since we don't need both the jid.ID and the emp.ID columns (by definition they must be matching) we can exclude listing one of them from our select statement.

```
select distinct jid.ID, jid.Name, jid.Raises, emp.Age

from StateDW.dbo.JobID as jid

outer join StateDW.dbo.EmpInfo as emp
on jid.ID=emp.ID
```

This join would return the following:

ID	Name	Raises	Age
01	John	2	21
10	Sam	1	34
11	Brian	0	-
00	-	-	19

Now, we can see all the information from both of our tables in one place!

Another useful tool in SQL is to perform operations on groups. Take the payroll table below:

ID	Name	Pay	Date
01	John	100	01-01-25
01	John	200	02-01-25
00	Sam	300	01-01-25
00	Sam	400	02-01-25

If we wanted to see how much each person and/or ID has been paid in total, over all dates, we could produce the following query:

```
select ID, Name, sum(Pay) as Total_Pay

from Table

group by ID, Name
```

This would return the following:

ID	Name	Total_Pay
01	John	300
00	Sam	700

The **group by** operator allows us to specify the fields over which we can find groups, in this case the ID and Name are the fields by which we want to create unique groups. Anything that is not listed in the **group by** clause but still listed in the **select** statement is assumed to be a data field over which we will perform aggregations. For example, in our query we aggregated over the Pay field using the **SUM** operator since it was excluded in the **group by** clause while still being mentioned in the **select** statement.

What is important to remember is that SQL follows a given structure, where each command goes in a certain order.

Here is the general order:

select [data fields]

from [data table] **as** [alias]

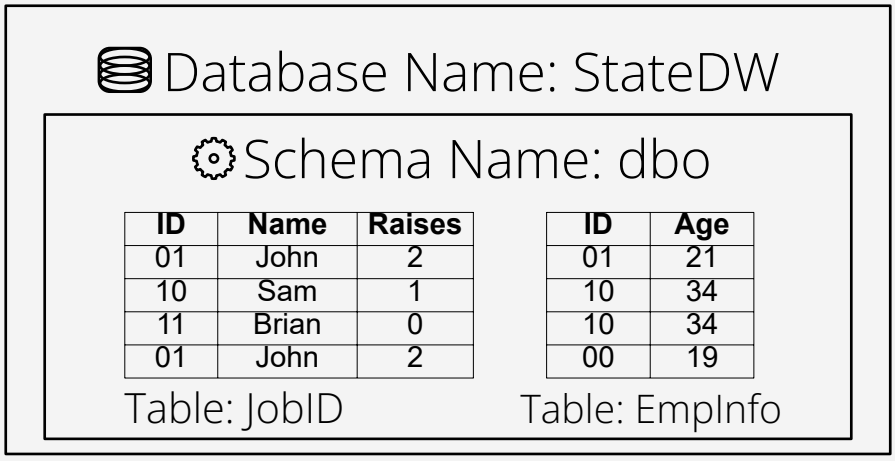
join [other data table] **on** [matching data fields]

group by [data field for groups]

where [filtering condition]

Let's do one last query now that we know the basics!

Here is our database we initially were working with. If you remember, when we computed the average age of our EmpInfo table, that average was overemphasizing employee 10, which is Sam. This last challenge is to calculate the true average of our employees without double counting employees who are duplicated in the EmpInfo table. To do this, we are going to use a **subquery**. A **subquery** effectively queries a table, and then does a query on top of that table. Take the example below:



select AVG(Age) **as** avg_age

from

(**select distinct** *

from StateDW.dbo.EmpInfo

) **as** sub

This returns:

avg_age
24.67

Typically, we pull a database **table** in the **from** clause, but this time we are pulling a manufactured table, a **subquery**, in the **from** clause. Note how that subquery is surrounded by parentheses; its results function as a table! In any **select** statement, an asterisk can be used as a wildcard to pull back the data from **all** datafields in the table; in this case, we are pulling back all the **distinct** data in the table, thus creating a deduplicated table. On this deduplicated table, nicknamed **sub**, we calculate the average age. Note how we don't use a **group by**, since we want to compute the aggregation over the whole dataset, not over specific groups.

Congrats! Now you know the basics of how to interpret and create SQL!