—————————————————————

SQL command line

————————————————————————————

https://www.sqlshack.com/working-sql-server-command-line-sqlcmd/

https://www.databasejournal.com/ms-sql/top-10-transact-sql-statements-a-sql-server-dba-should-know/

Server Level Transact-SQL Statements a SQL Server DBA should know

T-SQL Statement 1

The following T-SQL statement retrieves information such as Hostname, Current instance name, Edition, Server type, ServicePack and version number from current SQL Server connection. ‘Edition’ will give information on a 32 bit or 64 bit architecture and ‘Productlevel’ gives information about what service pack your SQL Server is on. It also displays if the current SQL Server is a clustered server. Refer Fig 1.1

SELECT

SERVERPROPERTY('MachineName') as Host,

SERVERPROPERTY('InstanceName') as Instance,

SERVERPROPERTY('Edition') as Edition, /\*shows 32 bit or 64 bit\*/

SERVERPROPERTY('ProductLevel') as ProductLevel, /\* RTM or SP1 etc\*/

Case SERVERPROPERTY('IsClustered') when 1 then 'CLUSTERED' else

'STANDALONE' end as ServerType,

@@VERSION as VersionNumber



**Fig 1.1**

T-SQL Statement 2

Server level configuration controls some of the features and performance of SQL Server. It is also important for a SQL Server DBA to know the server level configuration information. The following SQL Statement will give all of the information related to Server level configuration. Refer Fig 1.2

SELECT \* from sys.configurations order by NAME

If you are using SQL Server 2000, you can execute the following command instead.

SP\_CONFIGURE 'show advanced options',1

go

RECONFIGURE with OVERRIDE

go

SP\_CONFIGURE

go

Table

Description automatically generated with medium confidence

**Fig 1.2**

T-SQL Statement 3

Security is a very important aspect that a DBA should know about. It is also important to know which login has a [sysadmin](https://www.webopedia.com/TERM/S/system_administrator.html) or security admin server level role. The following SQL Command will show information related to the security admin server role and system admin server role. Refer Fig 1.3

SELECT l.name, l.denylogin, l.isntname, l.isntgroup, l.isntuser

FROM master.dbo.syslogins l

WHERE l.sysadmin = 1 OR l.securityadmin = 1

Table

Description automatically generated

**Fig 1.**3

T-SQL Statement 4

Another important bit of information that you need to know as a DBA is all of the traces that are enabled. The following T-SQL statement will list all of the trace flags that are enabled gloabally on the server. Refer Fig 1.4

DBCC TRACESTATUS(-1);

The following T-SQL statement will list all the trace flags that are enabled on the current sql server connection. Refer Fig 1.4

DBCC TRACESTATUS();

Graphical user interface, application, table

Description automatically generated

**Fig 1.4**

Database Level Transact-SQL Statements a SQL Server DBA should know

T-SQL Statement 5

The following T-SQL statement gives information on the database names, their compatibility level and also the recovery model and their current status. The result from this T-SQL Statement will help you to determine if there is any compatibility level update necessary. When upgrading from an older version to new version, the compatibility level of the database may not be in the desired level. The following statement will help you to list all of the database names with compatibilty level. It also lists the online/offline status of the database as well as helping the DBA to see if any update to recovery model is necessary. Refer Fig 1.5

SELECT name,compatibility\_level,recovery\_model\_desc,state\_desc FROM sys.databases

Table

Description automatically generated

**Fig 1.5**

If you are using SQL Server 2000, you could execute the following T-SQL Statement. Refer Fig 1.6

SELECT name,cmptlevel,DATABASEPROPERTYEX(name,'Recovery')AS RecoveryModel,

DATABASEPROPERTYEX(name,'Status') as Status FROM sysdatabases

Table

Description automatically generated

**Fig 1.6**

T-SQL Statement 6

The next level of information related to database that is needed is the location of the database. The following T-SQL Statement provides the logical name and the physical location of the data/log files of all the databases available in the current SQL Server instance. Referg Fig 1.7

SELECT db\_name(database\_id) as DatabaseName,name,type\_desc,physical\_name FROM sys.master\_files

Graphical user interface, text, application

Description automatically generated

**Fig 1.7**

If you are using SQL Server 2000, you could execute the following T-SQL Statement. Refer Fig 1.8

SELECT db\_name(dbid) as DatabaseName,name,filename FROM master.dbo.sysaltfiles

Graphical user interface, text, application

Description automatically generated

**Fig 1.8**

T-SQL Statement 7

A database may contain filegroups other than the primary file group. The following T-SQL Statement gets executed in each database on the server and displays the file groups related results. Refer Fig 1.9

EXEC master.dbo.sp\_MSforeachdb @command1 = 'USE [?] SELECT \* FROM sys.filegroups'

Graphical user interface, text, application

Description automatically generated

**Fig 1.9**

Backup Level Transact-SQL Statements a SQL Server DBA should know

T-SQL Statement 8

Backup of a database is bread and butter for database administrators. The following T-SQL Statement lists all of the databases in the server and the last day the backup happened. This will help the database administrators to check the backup jobs and also to make sure backups are happening for all the databases. Refer Fig 1.10

SELECT db.name,

case when MAX(b.backup\_finish\_date) is NULL then 'No Backup' else convert(varchar(100),

MAX(b.backup\_finish\_date)) end AS last\_backup\_finish\_date

FROM sys.databases db

LEFT OUTER JOIN msdb.dbo.backupset b ON db.name = b.database\_name AND b.type = 'D'

WHERE db.database\_id NOT IN (2)

GROUP BY db.name

ORDER BY 2 DESC

Table

Description automatically generated

**Fig 1.10**

T-SQL Statement 9

The next level of information that is important for a SQL Server database administrator to know is the location of all the backup files. You don’t want the backups to go to the local drive or to an OS drive. The following T-SQL statement gets all the information related to the current backup location from the msdb database. Refer Fig 1.11

SELECT Distinct physical\_device\_name FROM msdb.dbo.backupmediafamily

Graphical user interface, application, table

Description automatically generated

**Fig 1.11**

Process Level Transact-SQL Statements a SQL Server DBA should know

T-SQL Statement 10

Last but not least, is the information related to current processes and connection related information. From the beginning, SQL Server database administrators used sp\_who and sp\_who2 to check the current users, process and session information. These statements also provided information related to cpu, memory and blocking information related to the sessions. Refer Fig 1.12. Also, search the internet for sp\_who3. You can find many articles related to sp\_who3.

sp\_who

sp\_who2

Table, Excel

Description automatically generated

**Fig 1.12**

====================================================================================================

=======================================================================================================

**To create a Table**

create table risk\_clos\_rank(

id\_num int IDENTITY(1,1) NOT NULL,

username nvarchar(100),

datetime\_of\_decision DATETIME

);

CREATE TABLE TheNameOfYourTable (

ID INT NOT NULL IDENTITY(1,1),

DateAdded DATETIME DEFAULT(getdate()) NOT NULL,

Description VARCHAR(100) NULL,

IsGood BIT DEFAULT(0) NOT NULL,

TotalPrice MONEY NOT NULL,

CategoryID int NOT NULL REFERENCES Categories(ID),

PRIMARY KEY (ID)

);

**To create a copy of table( doesnt create constraints like primary key, not null , indexes ect)**

SELECT \* INTO NewTable FROM OldTable

Eg. SELECT \* INTO clos\_ext\_bkup FROM clos\_ext;

**To create a copy of table with its data (create and insert)**

SELECT expressions INTO new\_table FROM tables [WHERE conditions];

SELECT employee\_id AS contact\_id, last\_name, first\_name INTO contacts FROM employees WHERE employee\_id < 1000;

The format of new\_table is determined by evaluating the expressions in the select list. The columns in new\_table are created in the order specified by the select list. Each column in new\_table has the same name, data type, nullability, and value as the corresponding expression in the select list.

**Inserting Data from another table ( only insert)**

INSERT INTO Table (col1, col2, col3) SELECT col1, col2, col3 FROM other\_table WHERE sql = 'cool'

INSERT INTO contacts (contact\_id, last\_name, first\_name) SELECT employee\_id, last\_name, first\_name FROM employees WHERE employee\_id <= 100;

**Inserting Multiple values**

INSERT INTO table1 (First, Last)

VALUES

('Fred', 'Smith'),

('John', 'Smith'),

('Michael', 'Smith'),

('Robert', 'Smith');

**To add a column**

ALTER TABLE table\_name ADD column\_1 column-definition,column\_2 column-definition,column\_n column\_definition;

alter table risk\_user\_approval\_tree add lineusr nvarchar(100);

ALTER TABLE table ADD columnname BIT CONSTRAINT Constraint\_name DEFAULT 0 WITH VALUES

**To add a auto increment**

ALTER TABLE 'tableName' ADD 'NewColumn' INT IDENTITY(1,1);

**To add a column with computed value**

ALTER TABLE dbo.Products ADD RetailValue AS (QtyAvailable \* UnitPrice \* 1.5);

**To delete/drop a column**

ALTER TABLE table\_name DROP COLUMN column\_name;

**To drop a table**

DROP TABLE tablename;

**To modify a column**

ALTER TABLE table\_name ALTER COLUMN column\_name column\_type;

**To update a row**

UPDATE clos\_customer\_master SET Prev = 'Reactivation' WHERE Prev = 'Reactivate';

**To update a row from select clause**

UPDATE table SET Col1 = i.Col1, Col2 = i.Col2 FROM ( SELECT ID, Col1, Col2 FROM other\_table) i WHERE i.ID = table.ID;

The subquery results are substituted into the outer query. As we need table object in outer query, we need to make an alias of inner query.

**To add a primary key**

ALTER TABLE table\_name ADD CONSTRAINT constraint\_name PRIMARY KEY (column1, column2, ... column\_n);

**To find the name of constraints**

SELECT \* FROM INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS WHERE TABLE\_NAME = 'tablename'

**To find name of Primary key constraint**

SELECT name FROM sys.key\_constraints WHERE type = 'PK' AND OBJECT\_NAME(parent\_object\_id) = N'CLOS\_ext';

**Drop primary key**

ALTER TABLE table\_name DROP CONSTRAINT constraint\_name;

**To rename a column (alter command doesnt work here)**

sp\_rename 'table\_name.old\_column\_name', 'new\_column\_name', 'COLUMN';

sp\_rename 'cl\_ff\_docm.WINAME', 'WI\_NAME', 'COLUMN';

**To rename a table**

sp\_rename 'old\_table\_name', 'new\_table\_name';

**To top 10% of records**

SELECT TOP(10) \* FROM CLOS\_EXT

**To find when a table was altered**

SELECT [name] , create\_date, modify\_date FROM sys.tables;

**To find which table contains a given column**

SELECT \* FROM INFORMATION\_SCHEMA.COLUMNS;

SELECT OBJECT\_SCHEMA\_NAME (c.object\_id) SchemaName,

o.Name AS Table\_Name,

c.Name AS Field\_Name,

t.Name AS Data\_Type,

t.max\_length AS Length\_Size,

t.precision AS Precision

FROM sys.columns c

INNER JOIN sys.objects o ON o.object\_id = c.object\_id

LEFT JOIN sys.types t on t.user\_type\_id = c.user\_type\_id

WHERE o.type = 'U'

-- and o.Name = 'YourTableName'

ORDER BY o.Name, c.Name

**To find which table has which constraint and on which column.**

Select \* from INFORMATION\_SCHEMA.KEY\_COLUMN\_USAGE;

**Selcting based on case**

SELECT CASE

WHEN <test> THEN <returnvalue>

WHEN <othertest> THEN <returnthis>

ELSE <returndefaultcase>

END AS <newcolumnname>

FROM <table>

Eg.

SELECT ProductNumber, Name, "Price Range" =

CASE

WHEN ListPrice = 0 THEN 'Mfg item - not for resale'

WHEN ListPrice < 50 THEN 'Under $50'

WHEN ListPrice >= 50 and ListPrice < 250 THEN 'Under $250'

WHEN ListPrice >= 250 and ListPrice < 1000 THEN 'Under $1000'

ELSE 'Over $1000'

END

FROM Production.Product

ORDER BY ProductNumber ;

**Adding row numbers to the result** //here we are creating

SELECT ROW\_NUMBER() OVER(ORDER BY name ASC) AS Row#, name,

recovery\_model\_desc

FROM sys.databases

WHERE database\_id < 5;

**While Loop**

DECLARE @MaxCount INTEGER

DECLARE @Count INTEGER

DECLARE @Txt VARCHAR(MAX)

SET @Count = 1

SET @Txt = ''

SET @MaxCount = (SELECT MAX(RowID) FROM ConcatenationDemo)

WHILE @Count<=@MaxCount

BEGIN

IF @Txt!=''

SET @Txt=@Txt+',' + (SELECT Txt FROM ConcatenationDemo

WHERE RowID=@Count)

ELSE

SET @Txt=(SELECT Txt FROM ConcatenationDemo WHERE RowID=@Count)

SET @Count += 1

END

SELECT @Txt AS Txt

DECLARE @i int

SET @i = 0

WHILE (@i < 10)

BEGIN

SET @i = @i + 1

PRINT @i

IF (@i >= 10)

BREAK

ELSE

CONTINUE

END

**Try / Catch Statements**

BEGIN TRY

-- try / catch requires SQLServer 2005

-- run your code here

END TRY

BEGIN CATCH

PRINT 'Error Number: ' + str(error\_number())

PRINT 'Line Number: ' + str(error\_line())

PRINT error\_message()

-- handle error condition

END CATCH

**To get date in DD/MM/YYYY format**

SELECT CONVERT(varchar, GETDATE(), 103);

**To get all foreign keys refrencing a given table**

EXEC sp\_fkeys 'TableName'

**To get datatype, size of columns of a table**

EXEC sp\_columns CLOS\_EXT;

**To get empty string after concatenation of a string with NULL**

When **SET CONCAT\_NULL\_YIELDS\_NULL** is ON, concatenating a null value with a string yields a NULL result.

For example, SELECT 'abc' + NULL yields NULL.

When SET CONCAT\_NULL\_YIELDS\_NULL is OFF, concatenating a null value with a string yields the string itself (the null value is treated as an empty string).

For example, SELECT 'abc' + NULL yields abc.

**To compile without executing**

SET NOEXEC ON;

When SET NOEXEC is ON, SQL Server compiles each batch of Transact-SQL statements but does not execute them.

**Updating data from another table**

UPDATE table SET Col1 = i.Col1, Col2 = i.Col2 FROM ( SELECT ID, Col1, Col2 FROM other\_table) i WHERE i.ID = table.ID

**Check if column exists in table**

IF EXISTS(SELECT 1 FROM sys.columns

WHERE Name = N'columnName'

AND Object\_ID = Object\_ID(N'schemaName.tableName'))

BEGIN

-- Column Exists

END

**Converting Multi row data into a comma separated string**

DECLARE @Names VARCHAR(8000)

SELECT @Names = COALESCE(@Names + ', ', '') +

ISNULL(Name, 'N/A')

FROM People

**Nvarchar** allows storing of unicode data

**To remove duplicate rows**

select distinct \* into t2 from t1;

delete from t1;

insert into t1 select \* from t2;

drop table t2;

**Check if the table exists**

IF (EXISTS (

SELECT \*

FROM INFORMATION\_SCHEMA.TABLES

WHERE TABLE\_SCHEMA = 'TheSchema'

AND TABLE\_NAME = 'TheTable')

)

BEGIN

--Do Stuff

END

**Find tables with given column name**

select \* from INFORMATION\_SCHEMA.COLUMNS

where COLUMN\_NAME like '%clientid%'

order by TABLE\_NAME

**Find all user tables**

SELECT \* FROM INFORMATION\_SCHEMA.TABLES WHERE TABLE\_TYPE='BASE TABLE'

**Allows explicit values to be inserted into the identity column of a table.**

SET IDENTITY\_INSERT dbo.Tool ON

The **DBCC CHECKIDENT** management command is used to reset identity counter. Example:

DBCC CHECKIDENT ('[TestTable]', RESEED, 0);

GO

**DECLARE and SET Varibales**

DECLARE @Mojo int

SET @Mojo = 1

SELECT @Mojo = Column FROM Table WHERE id=1;

**Add a Foreign Key**

ALTER TABLE Products WITH CHECK ADD CONSTRAINT [FK\_Prod\_Man] FOREIGN KEY(ManufacturerID) REFERENCES Manufacturers (ID);

**Add a NULL Constraint**

ALTER TABLE TableName ALTER COLUMN ColumnName int NOT NULL;

**Set Default Value for Column**

ALTER TABLE TableName ADD CONSTRAINT DF\_TableName\_ColumnName DEFAULT 0 FOR ColumnName;

**Create an Index**

CREATE INDEX IX\_Index\_Name ON Table(Columns)

**Check Constraint**

ALTER TABLE TableName ADD CONSTRAINT CK\_CheckName CHECK (ColumnValue > 1)

**Single Line Comments**

SET @mojo = 1 --THIS IS A COMMENT

**Multi-Line Comments**

/\* This is a comment

that can span

multiple lines

\*/

**User Defined Function**

CREATE FUNCTION dbo.DoStuff(@ID int)

RETURNS int

AS

BEGIN

DECLARE @result int

IF @ID = 0

BEGIN

RETURN 0

END

SELECT @result = COUNT(\*)

FROM table WHERE ID = @ID

RETURN @result

END

GO

SELECT dbo.DoStuff(0);

**Pivot - To convert rows into columns**

SELECT Wi\_name, Often, Sometimes, Never, NA

FROM

(

SELECT Wi\_name, Past\_due, 'Selected' T, '' F

FROM NG\_CA\_MISCELLANEOUS\_DETAILS

) P1

PIVOT

(

MAX(T) for Past\_due IN ([Often], [Sometimes], [Never],[NA])

)

P2 ORDER BY WI\_NAME;

**WITH (NOLOCK)**

is the equivalent of using READ UNCOMMITED as a transaction isolation level. While it can prevent reads being deadlocked by other.

**Finding the last identity inserted into a table**

* **@@IDENTITY** returns the last identity value generated for any table in the current session, across all scopes. You need to be careful here, since it's across scopes. You could get a value from a trigger, instead of your current statement.
* **SCOPE\_IDENTITY()** returns the last identity value generated for any table in the current session and the current scope. Generally what you want to use.
* **IDENT\_CURRENT('tableName')** returns the last identity value generated for a specific table in any session and any scope. This lets you specify which table you want the value from, in case the two above aren't quite what you need (very rare). You could use this if you want to get the current IDENTITY value for a table that you have not inserted a record into.

@@IDENTITY and SCOPE\_IDENTITY will return the last identity value generated in any table in the current session.

However, SCOPE\_IDENTITY returns the value only within the current scope; @@IDENTITY is not limited to a specific scope. That is, if there was a second IDENTITY inserted based on a trigger after your insert, it would not be reflected in SCOPE\_IDENTITY, only the insert you performed.

IDENT\_CURRENT is not limited by scope and session; it is limited to a specified table. IDENT\_CURRENT returns the identity value generated for a specific table in any session and any scope. For more information, see IDENT\_CURRENT.

Identity doesn’t guarantee uniqueness. If you want that, make a PK or add a unique index.

=====================================================

===================================================

**Query 1: Ranking the Data**

Management likes to see rankings for absolutely anything: products sold, salaries, employees per department, money earned per any imaginable segment – they’ll always ask to see it ranked. To show you an example of how to rank things in SQL, I’ll use the **sales** table. It has the following columns:

* product - The name of the product.
* product\_price - The price of the product.
* items\_sold - The number of items sold.

The idea is to calculate the revenue for every product and rank it using the RANK() function. The code below will solve this task:

|  |
| --- |
| SELECT  product,          product\_price,          items\_sold,          product\_price \* items\_sold AS revenue,          RANK() OVER (ORDER BY product\_price \* items\_sold DESC) AS revenue\_rank  FROM sales; |

The above code selects all columns from the table. To get the revenue per product, you need to multiply the price by the items sold. This is exactly what the query will do, and the result will be shown in the new column *revenue*. Finally, there’s the RANK() function. Here, this function will rank all rows by the new column revenue (defined by product\_price \* items\_sold). The rank will be shown in the new column *revenue\_rank*.

And here’s the result:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **product** | **product\_price** | **items\_sold** | **revenue** | **revenue\_rank** |
| 14HA34OrtegaYGasset | 44.12 | 6547 | 288,853.64 | 1 |
| TAL578UmbertoEcoFriendly | 100 | 547 | 54,700.00 | 2 |
| 44HRZ890Sartrade | 52.87 | 800 | 42,296.00 | 3 |
| FX312AdornoForHome | 12 | 3254 | 39,048.00 | 4 |
| H618T4DeBeauvoirForAll | 47.59 | 813 | 38,690.67 | 5 |
| H16GRSocratesYoghurt | 1.5 | 14587 | 21,880.50 | 6 |
| 67GHZ4Marximum | 9.99 | 1789 | 17,872.11 | 7 |
| H618T4HobbesSolutions | 7.77 | 147 | 1,142.19 | 8 |
| 14HX13Arendt | 12.47 | 47 | 586.09 | 9 |
| MT657GombrowiczExtra | 4 | 12 | 48.00 | 10 |

RANK() is only one way to rank data in SQL. You can see the other ways in this [article](https://learnsql.com/blog/common-sql-window-functions-ranking-functions/). And, if you want to do a deep dive into the subject, here’s our [complete guide to ranking functions](https://learnsql.com/blog/how-to-rank-rows-in-sql/).

**Query 2: Calculating Delta Values**

Along with ranking, [calculating delta values](https://learnsql.com/blog/difference-between-two-rows-in-sql/) is probably one of data analysts’ most common tasks. This is usually required when calculating day-to-day, month-to-month, quarter-to-quarter, or year-to-year changes. Be it revenue, costs, price changes, volume changes, or anything else imaginable, you’ll need to calculate the difference between the numbers. To do that, an advanced SQL query with the LAG() function is what you need. This function is used to retrieve the data from a previous row. Let me show you how it works using the **revenue** table. The table consists of two rows:

* month - The month of the year.
* revenue - The revenue for that month.

Your task is to calculate the difference between each month’s revenue and the previous month (i.e. the monthly revenue delta). How would you do that? If you know the LAG() function, it’s very easy. Here’s the code:

|  |
| --- |
| SELECT  month,          revenue,          revenue - LAG(revenue, 1) OVER (ORDER BY month) AS monthly\_delta  FROM revenue; |

The principle for calculating the delta is to deduct the previous month from the current month. The code above does exactly that. First, it selects the columns month and *revenue*. After that, it deducts the amount of the current month’s revenue from the previous month. This is defined by the LAG() function. The values we put in the function’s parenthesis (revenue, 1) indicate the value in the column *revenue* will be deducted from the previous value in that column. That’s why there’s the number 1; it defines how many rows the function goes back to perform the operation.

In theory, LAG(revenue) will do the same, since going one row back is the default for the LAG() function. However, I wanted to show you this explicitly. It’s easier to understand and you’ll know what to do when you need to go back more than one row.

Deltas need to be calculated sequentially, not by random months; that’s why there’s ORDER BY month. The delta will be shown in the new column *monthly\_delta*. Run the code and you’ll get the resulting table:

|  |  |  |
| --- | --- | --- |
| **month** | **revenue** | **monthly\_delta** |
| 01/2019 | 12587.14 | NULL |
| 02/2019 | 478456.88 | 465869.74 |
| 03/2019 | 312588 | -165868.88 |
| 04/2019 | 518387.66 | 205799.66 |
| 05/2019 | 222222.22 | -296165.44 |
| 06/2019 | 588954.48 | 366732.26 |
| 07/2019 | 358981 | -229973.48 |
| 08/2019 | 678841.54 | 319860.54 |
| 09/2019 | 1547895.82 | 869054.28 |
| 10/2019 | 1647895.82 | 100000 |
| 11/2019 | 912541.26 | -735354.56 |
| 12/2019 | 984784.52 | 72243.26 |

**Query 3: Calculating Running Totals**

Running totals (also known as cumulative sums) are extensively used in data analysis. They are usually used with time series data to see how certain performance indicators are (or will be) developing over time. Like other advanced SQL concepts, running totals have a very broad practical use. They’re used to monitor sales, revenues, costs, profit, and budgets. Here’s an [article](https://learnsql.com/blog/what-is-a-running-total-and-how-to-compute-it-in-sql/) that nicely explains running totals and how to calculate them in SQL.

For now, I’ll show you how a cumulative sum works for budgets. Let’s use a table very imaginatively named **budget**. It consists of these columns:

* month - The month of the cash flow.
* client - The client name.
* cash\_flow - The budgeted cash flow.

There are three clients. The budget contains monthly projections of the yearly cash flow that your company will collect from them. You need to calculate the cumulative cash flow for each client. You might already intuitively know you have to use the SUM() function, but with some kind of twist. Here’s how to calculate running totals:

|  |
| --- |
| SELECT  month,          client,          cash\_flow,          SUM (cash\_flow) OVER (PARTITION BY client ORDER BY month) AS running\_total    FROM budget; |

This is a simple little query that does wonders! It selects the columns *month*, *client*, and *cash\_flow*. To calculate the running total, you have to summarize the cash flows. This is defined by SUM (cash\_flow).

However, you’re not interested in having running totals on a table level. You need to somehow make it summarize cash flows month by month for the first client, then reset and start again for the second client. To do that, you need OVER (PARTITION BY client ORDER BY month). Here the partition is defined by the column client, which means that every data set is defined by the different clients. Also, the operation will be performed only within the partition, not on the whole table. That way, you get a running total for every client separately.

Of course, the cash flows have to be summarized sequentially; that’s why it’s ordered by the *month* column. The running total will appear in the new column *running\_total*.

Here’s the resulting table:

|  |  |  |  |
| --- | --- | --- | --- |
| **month** | **client** | **cash\_flow** | **running\_total** |
| 01/2020 | Claudio Gaudio | 75564.38 | 75564.38 |
| 02/2020 | Claudio Gaudio | 12894.45 | 88458.83 |
| 03/2020 | Claudio Gaudio | 75564.38 | 164023.21 |
| 04/2020 | Claudio Gaudio | 12894.45 | 176917.66 |
| 05/2020 | Claudio Gaudio | 743541.12 | 920458.78 |
| 06/2020 | Claudio Gaudio | 325558.45 | 1246017.23 |
| 07/2020 | Claudio Gaudio | 390278.63 | 1636295.86 |
| 08/2020 | Claudio Gaudio | 22008.12 | 1658303.98 |
| 09/2020 | Claudio Gaudio | 85000 | 1743303.98 |
| 10/2020 | Claudio Gaudio | 42840.55 | 1786144.53 |
| 11/2020 | Claudio Gaudio | 85612.34 | 1871756.87 |
| 12/2020 | Claudio Gaudio | 412000 | 2283756.87 |
| 01/2020 | Gabriele Pappardelle | 49000 | 49000 |
| 02/2020 | Gabriele Pappardelle | 18480.26 | 67480.26 |
| 03/2020 | Gabriele Pappardelle | 127850.5 | 195330.76 |
| 04/2020 | Gabriele Pappardelle | 327000.5 | 522331.26 |
| 05/2020 | Gabriele Pappardelle | 500000 | 1022331.26 |
| 06/2020 | Gabriele Pappardelle | 0 | 1022331.26 |
| 07/2020 | Gabriele Pappardelle | 0 | 1022331.26 |
| 08/2020 | Gabriele Pappardelle | 1000000 | 2643324.72 |
| 08/2020 | Gabriele Pappardelle | 620993.46 | 2643324.72 |
| 09/2020 | Gabriele Pappardelle | 0 | 2643324.72 |
| 10/2020 | Gabriele Pappardelle | 500000 | 3143324.72 |
| 11/2020 | Gabriele Pappardelle | 500000 | 3643324.72 |
| 12/2020 | Gabriele Pappardelle | 500000 | 4143324.72 |
| 01/2020 | Tony Pepperoni | 10000 | 10000 |
| 02/2020 | Tony Pepperoni | 10000 | 20000 |
| 03/2020 | Tony Pepperoni | 10000 | 30000 |
| 04/2020 | Tony Pepperoni | 0 | 30000 |
| 05/2020 | Tony Pepperoni | 0 | 30000 |
| 06/2020 | Tony Pepperoni | 25787 | 55787 |
| 07/2020 | Tony Pepperoni | 32000 | 87787 |
| 08/2020 | Tony Pepperoni | 25787 | 113574 |
| 09/2020 | Tony Pepperoni | 0 | 113574 |
| 10/2020 | Tony Pepperoni | 18000 | 131574 |
| 11/2020 | Tony Pepperoni | 67450.5 | 199024.5 |
| 12/2020 | Tony Pepperoni | 1000 | 200024.5 |

I’ve used window functions in the last three examples. If you want to learn more about this topic, a good way is the [Window Functions course](https://learnsql.com/course/window-functions), one of our advanced SQL courses.

Something that could also be very helpful, especially if you’re new to window functions or use them only occasionally, is this [SQL Window Functions Cheat Sheet](https://learnsql.com/blog/sql-window-functions-cheat-sheet/). I’ll be using it next time I write about window functions, for sure!

**Query 4: Creating a Report Based on Multiple Conditions**

One of data analysts’ main tasks is making data more friendly for other users. By giving them data in a form that they can easily use, we make their jobs easier. To create useful reports, a data analyst has to combine business input with their knowledge of the data. One of the tools that can help you in achieving that is a CASE statement, which is another advanced SQL concept.

To give you an example, let’s imagine the following scenario. You’re working for a bank and you’re asked by your colleagues to create a report. There’s a table called **debt** that shows the bank’s clients and details about their debt. The table consists of the following columns:

* client - The name of the client.
* date\_due - The day the debt became due.
* amount\_due - The amount of the debt that is due.

What you need to do is create a report as of 30.4.2020. You somehow need to calculate the number of days due as of the reporting date. Also, you need to allocate the client to a certain time bucket, according to the number of the days their account is due.

The query is just below. Don’t be afraid – I’ll analyze it for you. It’s not as scary as it looks!

|  |
| --- |
| SELECT  client,          date\_due,          amount\_due,          DATEDIFF ('2020-04-30', date\_due) AS days\_due,          CASE      WHEN  DATEDIFF ('2020-04-30', date\_due) <= 30 THEN '0-30 days'      WHEN  DATEDIFF ('2020-04-30', date\_due) > 30 AND DATEDIFF ('2020-04-30', date\_due) <=90 THEN '31-90 days'      WHEN  DATEDIFF ('2020-04-30', date\_due) > 90 AND DATEDIFF ('2020-04-30', date\_due) <=180 THEN '91-180 days'      WHEN  DATEDIFF ('2020-04-30', date\_due) > 180 AND DATEDIFF ('2020-04-30', date\_due) <=365 THEN '181-365 days'      ELSE '> 365 days'  END AS time\_bucket    FROM debt; |

First, you need to specify the SELECT part of the query. I’ve selected the existing columns *client*, *date\_due*, and *amount\_due*.

Next, you have to calculate the days due. You do that by subtracting the due date from the reporting date. This is exactly what I did with DATEDIFF ('2020-04-30', date\_due) AS days\_due. I’ve used the DATEDIFF() function to calculate the required difference. When using this function, you first have to specify which dates you want to subtract. In our case, it’s the reporting date and the date due. Next, you have to specify how you want the result to be shown, i.e. in years, months, or days. You need days in this case, so you put day as the last value in DATEDIFF().

Now comes the exciting part – creating the conditions I’ve used in the CASE statement. This statement opens with CASE and finishes with END. In between, you need to define the conditions that will create the report your colleagues want. For this, you’ll use WHEN and THEN.

Let’s say that the first bucket of the days due is 0-30 days. The first condition in the CASE statement is WHEN DATEDIFF ('2020-04-30', date\_due) <= 30 THEN '0-30 days'. Since you need to allocate clients to a time bucket according to the days due, this part of the code does exactly that. It reads like this: if the difference between the reporting date and the due date is less than or equal to 30 days, then this client will be allocated to the time bucket 0-30 days.

The next time bucket is 31-90 days, and this is the part of the code that defines it:

WHEN DATEDIFF ('2020-04-30', date\_due) > 30 AND DATEDIFF ('2020-04-30', date\_due) <=90 THEN '31-90 days'

It’s not that complicated, right?

The same principle works for the remaining two time buckets: 91-180 days and 181-365 days. Every debt that has been due for more than 365 days belongs to the 365 days time bucket. This is defined by ELSE '> 365 days'. This simply defines the criteria for reporting: if the value is this, do this; if it’s not, do this. Essentially, it’s a more complex version of the IF statement.

Please note there’s a more elegant way to write this code: I could’ve declared a variable containing the value ‘2020-04-30’ instead of writing ‘2020-04-30’ manually everywhere in the code. However, I didn’t want to confuse you if you’re not familiar with variables.

Also, regarding the DATEDIFF() function, note that I’ve used the MySQL function and syntax. Depending on the database engine you’re using, it’s possible that you’ll have to adapt the syntax accordingly.

All those time buckets will be shown in the new column *time\_bucket*. Since you want your data to look nice, you’ll order your table by *days\_due* ascending. Run the code and you’ll get a nice table. And probably a free coffee from your colleagues!

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **client** | **date\_due** | **amount\_due** | **days\_due** | **time\_bucket** |
| GreatCompany | 2019-12-31 | 10000 | 121 | 91-180 days |
| WeAreTheBest | 2020-04-15 | 2000 | 15 | 0-30 days |
| AlmostBankrupt | 2019-06-30 | 150000 | 305 | 181-365 days |
| WeWontPay | 2019-01-15 | 870000 | 471 | > 365 days |
| AllAboutMoney | 2020-01-15 | 5000 | 106 | 91-180 days |
| YouTalkinToMe | 2019-08-31 | 78000 | 243 | 181-365 days |
| BigLebowski | 2020-01-31 | 42000 | 90 | 31-90 days |
| MilesSmiles | 2019-11-30 | 78000 | 152 | 91-180 days |
| PanthelyaSolutions | 2019-10-31 | 7000 | 182 | 181-365 days |
| PurplePrince | 2019-12-31 | 500 | 121 | 91-180 days |

Exciting, isn’t it? If you want more excitement like this, [Creating Basic SQL Reports](https://learnsql.com/course/sql-basic-reporting) is for you! There you’ll learn more about CASE WHEN and the nuances of GROUP BY.

**Query 5: Adding Subtotals to a Report**

A very common request is to show subtotals and totals in the same report. The ROLLUP clause makes this much easier. It’s an extension of a GROUP BY clause. It allows you to add subtotals and grand totals to your data.

Here’s how to use ROLLUP. You have the table **warehouse** with the following columns:

* warehouse - The name of the warehouse.
* brand - The product’s brand.
* product - The product’s name.
* quantity - The quantity of this product in the warehouse.

There are two different brands with five products between them. And there are two warehouses. Your task is to calculate the total product quantity for both brands in both warehouses. You also need the grand total of all the products in both warehouses. And finally, you need to do everything in one table with one query. How would you manage it? The code is:

|  |
| --- |
| SELECT  warehouse,          brand,          SUM (quantity) AS sum\_product    FROM warehouse  GROUP BY ROLLUP (warehouse, brand); |

First, you select the columns *warehouse and brand* from the table. You also want the sum of the column *quantity*, which will be shown in the new table **sum\_product**. What’s the next step? This is when ROLLUP comes in! It’s used to get totals for multiple data grouping levels. The GROUP BY ROLLUP (warehouse, brand) part will do exactly that. It will group the data by the *warehouse* and *brand* columns. After that, it will sum the data according to each grouping. The result is:

|  |  |  |
| --- | --- | --- |
| **warehouse** | **brand** | **sum\_product** |
| Amsterdam | Brando | 1105 |
| Amsterdam | Ostap | 62934 |
| Amsterdam | NULL | 64039 |
| Berlin | Brando | 67356 |
| Berlin | Ostap | 13451 |
| Berlin | NULL | 80807 |
| NULL | NULL | 144846 |

The table contains totals for the Brando and Ostap brands in the Amsterdam and Berlin warehouses and a grand total. The subtotal for both products in the Amsterdam warehouse is shown in the first row with the NULL *brand* value. It amounts to 64 039, the sum of the two previous rows.

Next, you can see the totals for both brands in the Berlin warehouse. After that, there’s another line with a NULL *brand* value; this is actually the Berlin subtotal amounting to 80 807. The last row shows the grand total of all products in all warehouses, which is 144 846.

Why are there NULL values in some rows? Because SQL doesn’t know how to name brands and warehouses when they’re grouped and a subtotal or grand total is shown. To find more fun details about other GROUP BY extensions, check out our [GROUP BY Extensions course](https://learnsql.com/course/sql-group-by-extensions).