|  |  |
| --- | --- |
| **Keyword** | **Explanation** |
| **SELECT** | Used to state which columns to query. Use \* for all |
| **FROM** | Declares which table/view etc to select from |
| **WHERE** | Introduces a condition |
| **=** | Used for comparing a value to a specified input |
| **LIKE** | Special operator used with the WHERE clause to search for a specific pattern in a column |
| **GROUP BY** | Arranges identical data into groups |
| **HAVING** | Specifies that only rows where aggregate values meet the specified conditions should be returned. Used because the WHERE keyword cannot be used with aggregate functions |
| **INNER JOIN** | Returns all rows where key record of one table is equal to key records of another |
| **LEFT JOIN** | Returns all rows from the ‘left’ (1st) table with the matching rows in the right (2nd) |
| **RIGHT JOIN** | Returns all rows from the ‘right’ (2nd) table with the matching rows in the left (1st) |
| **FULL OUTER JOIN** | Returns rows that match either in the left or right table |

**Reporting aggregate functions**

In database management, an aggregate function is a function where the values of multiples rows are grouped to form a single value. They are useful for reporting and some examples of common aggregate functions can be found below:

|  |  |
| --- | --- |
| **Function** | **Explanation** |
| **COUNT** | Return the number of rows in a certain table/view |
| **SUM** | Accumulate the values |
| **AVG** | Returns the average for a group of values |
| **MIN** | Returns the smallest value of the group |
| **MAX** | Returns the largest value of the group |

**Querying data from a table**

A database table is a set of data elements (values) stored in a model of vertical columns and horizontal rows. Use any of the below to query a table in SQL:

|  |  |
| --- | --- |
| **SQL** | **Explanation** |
| **SELECT c1 FROM t** | Select data in column c1 from a table named t |
| **SELECT \* FROM t** | Select all rows and columns from a table named t |
| **SELECT c1 FROM t**  **WHERE c1 = ‘test’** | Select data in column c1 from a table named t where the value in c1 = ‘test’ |
| **SELECT c1 FROM t**  **ORDER BY c1 ASC (DESC)** | Select data in column c1 from a table name t and order by c1, either in ascending (default) or descending order |
| **SELECT c1 FROM t**  **ORDER BY c1LIMIT n OFFSET offset** | Select data in column c1 from a table named t and skip offset of rows and return the next n rows |
| **SELECT c1, aggregate(c2)**  **FROM t**  **GROUP BY c1** | Select data in column c1 from a table named t and group rows using an aggregate function |
| **SELECT c1, aggregate(c2)**  **FROM t**  **GROUP BY c1HAVING condition** | Select data in column c1 from a table named t and group rows using an aggregate function and filter these groups using ‘HAVING’ clause |

**Querying data from multiple tables**

As well as querying from a single table, SQL gives you the ability to query data from multiple tables:

|  |  |
| --- | --- |
| **SQL** | **Explanation** |
| **SELECT c1, c2**  **FROM t1**  **INNER JOIN t2 on condition** | Select columns c1 and c2 from a table named t1 and perform an inner join between t1 and t2 |
| **SELECT c1, c2**  **FROM t1**  **LEFT JOIN t2 on condition** | Select columns c1 and c2 from a table named t1 and perform a left join between t1 and t2 |
| **SELECT c1, c2**  **FROM t1**  **RIGHT JOIN t2 on condition** | Select columns c1 and c2 from a table named t1 and perform a right join between t1 and t2 |
| **SELECT c1, c2**  **FROM t1**  **FULL OUTER JOIN t2 on condition** | Select columns c1 and c2 from a table named t1 and perform a full outer join between t1 and t2 |
| **SELECT c1, c2**  **FROM t1**  **CROSS JOIN t2** | Select columns c1 and c2 from a table named t1 and produce a Cartesian product of rows in tables |
| **SELECT c1, c2**  **FROM t1, t2** | Same as above - Select columns c1 and c2 from a table named t1 and produce a Cartesian product of rows in tables |
| **SELECT c1, c2**  **FROM t1 A**  **INNER JOIN t2 B on condition** | Select columns c1 and c2 from a table named t1 and joint it to itself using an INNER JOIN clause |

**Using SQL Operators**

SQL operators are reserved words or characters used primarily in an SQL statement where clause to perform operations:

|  |  |
| --- | --- |
| SQL | Explanation |
| **SELECT c1 FROM t1**  **UNION [ALL]**  **SELECT c1 FROM t2** | Select column c1 from a table named t1 and column c1 from a table named t2 and combine the rows from these two queries |
| **SELECT c1 FROM t1**  **INTERSECT**  **SELECT c1 FROM t2** | Select column c1 from a table named t1 and column c1 from a table named t2 and return the intersection of two queries |
| **SELECT c1 FROM t1**  **MINUS**  **SELECT c1 FROM t2** | Select column c1 from a table named t1 and column c1 from a table named t2 and subtract the 2nd result set from the 1st |
| **SELECT c1 FROM t**  **WHERE c1 [NOT] LIKE pattern** | Select column c1 from a table named t and query the rows using pattern matching % |
| **SELECT c1 FROM t**  **WHERE c1 [NOT] in test\_list** | Select column c1 from a table name t and return the rows that are (or are not) in test\_list |
| **SELECT c1 FROM t**  **WHERE c1 BETWEEN min AND max** | Select column c1 from a table named t and return the rows where c1 is between min and max |
| **SELECT c1 FROM t**  **WHERE c1 IS [NOT] NULL** | Select column c1 from a table named t and check if the values are NULL or not |

**Data modification**

Data modification is a key part of SQL, giving the ability to not only add and delete data, but modify existing records:

|  |  |
| --- | --- |
| **SQL** | **Explanation** |
| **INSERT INTO t(column\_list)**  **VALUES(value\_list)** | Insert one row into a table named t |
| **INSERT INTO t(column\_list)**  **VALUES (value\_list), (value\_list), …** | Insert multiple rows into a table named t |
| **INSERT INTO t1(column\_list)**  **SELECT column\_list FROM t2** | Insert rows from t2 into a table named t1 |
| **UPDATE tSET c1 = new\_value** | Update a new value in table t in the column c1 for all rows |
| **UPDATE tSET c1 = new\_value, c2 = new\_value**  **WHERE condition** | Update values in column c1 and c2 in table t that match the condition |
| **DELETE FROM t** | Delete all the rows from a table named t |
| **DELETE FROM tWHERE condition** | Delete all rows from that a table named t that match a certain condition |

**Views**

A view is a virtual table that is a result of a query. They can be extremely useful and are often used as a security mechanism, letting users access the data through the view, rather than letting them access the underlying base table:

|  |  |
| --- | --- |
| **SQL** | **Explanation** |
| **CREATE VIEW view1 AS**  **SELECT c1, c2**  **FROM t1**  **WHERE condition** | Create a view, comprising of columns c1 and c2 from a table named t1 where a certain condition has been met. |

**Indexes**

An index is used to speed up the performance of queries by reducing the number of database pages that have to be visited:

|  |  |
| --- | --- |
| **SQL** | **Explanation** |
| **CREATE INDEX index\_nameON t(c1, c2)** | Create an index on columns c1 and c2 of the table t |
| **CREATE UNIQUE INDEX index\_name**  **ON t(c3, c4)** | Create a unique index on columns c3 and c4 of the table t |
| **DROP INDEX index\_name** | Drop an index |

**Stored procedure**

A stored procedure is a set of SQL statements with an assigned name that can then be easily reused and share by multiple programs:

|  |  |
| --- | --- |
| **SQL** | **Explanation** |
| **CREATE PROCEDURE procedure\_name**  **@variable AS datatype = value**  **AS**  **-- Comments**  **SELECT \* FROM tGO** | Create a procedure called procedure\_name, create a local variable and then select from table t |

**Triggers**

A trigger is a special type of stored procedure that automatically executes when a user tries to modify data through a DML event (data manipulation language). A DML event is an INSERT, UPDATE or DELETE statement on a table or view:

|  |  |
| --- | --- |
| **SQL** | **Explanation** |
| **CREATE OR MODIFY TRIGGER trigger\_name**  **WHEN EVENT**  **ON table\_name TRIGGER\_TYPE**  **EXECUTE stored\_procedure** | WHEN:   * **BEFORE** – invoke before the event occurs * **AFTER** – invoke after the event occurs   EVENT:   * **INSERT** – invoke for insert * **UPDATE** – invoke for update * **DELETE** – invoke for delete   TRIGGER\_TYPE:   * **FOR EACH ROW** * **FOR EACH STATEMENT** |
| **DROP TRIGGER trigger\_name** | Delete a specific trigger |

# SQLServer TSQL CheatSheet

|  |  |
| --- | --- |
| **DECLARE and SET Varibales**  DECLARE @Mojo int  SET @Mojo = 1  SELECT @Mojo = Column FROM Table WHERE id=1  **IF / ELSE IF / ELSE Statement**  IF @Mojo < 1  BEGIN  PRINT 'Mojo Is less than 1'  END  ELSE IF @Mojo = 1  BEGIN  PRINT 'Mojo Is 1'  END  ELSE  BEGIN  PRINT 'Mojo greater than 1'  END  **CASE Statement**  SELECT Day = CASE  WHEN (DateAdded IS NULL) THEN 'Unknown'  WHEN (DateDiff(day, DateAdded, getdate()) = 0) THEN 'Today'  WHEN (DateDiff(day, DateAdded, getdate()) = 1) THEN 'Yesterday'  WHEN (DateDiff(day, DateAdded, getdate()) = -1) THEN 'Tomorrow'  ELSE DATENAME(dw , DateAdded)  END  FROM Table  **Add A Column**  ALTER TABLE YourTableName ADD [ColumnName] [int] NULL;  **Rename a Column**  EXEC sp\_rename 'TableName.OldColName', 'NewColName','COLUMN';  **Rename a Table**  EXEC sp\_rename 'OldTableName', 'NewTableName';  **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)  **DROP a Column**  ALTER TABLE TableName DROP COLUMN ColumnName; | **Single Line Comments**  SET @mojo = 1 **--THIS IS A COMMENT**  **Multi-Line Comments**  /\* This is a comment  that can span  multiple lines  \*/  **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  **While Loop**  DECLARE @i int  SET @i = 0  WHILE (@i < 10)  BEGIN  SET @i = @i + 1  PRINT @i  IF (@i >= 10)  BREAK  ELSE  CONTINUE  END  **CREATE a Table**  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)  );  **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) |

Determine if object exists

IF OBJECT\_ID('product.Model', 'U') **IS** **NOT** **NULL**

add Check Constraint

**ALTER** **TABLE** dbo.MyTable

**ADD** **CONSTRAINT** CHK\_dbo\_MyTable\_Value

**CHECK** (VALUE > 0.00)

T-SQL Processing Order

1. From
2. Where
3. Group by
4. Having
5. Select
6. Order By

Window Function

**SELECT** PurchaseOrderID, ItemCode, subtotal,

ROW\_NUMBER() OVER ( PARTITION **BY** PurchaseOrderID **ORDER** **BY** ItemCode) **AS** rownum,

**SUM**(subtotal) OVER (PARTITION **BY** PurchoseOrderID) **AS** purchaseOrderTotal

**FROM** pur.PurchaseOrderDetail

Casting

**SELECT** **CAST**('12345' **AS** NUMERIC(12,2))

Case Expressions

**SELECT** Vendor\_Code =

**CASE**

**WHEN** VendorItemCode **IS** **NULL** **THEN** ''

**WHEN** LEN(VendorItemCode) > 10 **THEN** **LEFT**(VendorItemCode, 10)

**ELSE** VendorItemCode

**END**

**FROM** pur.PurchaseOrderDetail

String Functions

**select** LEN('sql server') *-- 10*

**select** CHARINDEX('e','sql server') *-- 6*

**select** PATINDEX('%serv%', 'sql server') *-- 5*

**select** **REPLACE**('sql server', 'sql', 'cookie') *-- cookie server*

**select** REPLICATE('sql', 4) *-- sqlsqlsqlsql*

**select** STUFF('sql server', 1, 0, 'Microsoft ') *-- Microsoft sql server*

Wildcards

**SELECT** ManufacturerCode

**FROM** product.Model

**WHERE** ManufacturerCode

**LIKE** 'PG-42%' *--PG-42445-01 PG-42600-02*

**LIKE** '%G-42%'*-- PG-42445-01 RG-42900-03*

**LIKE** 'RG-\_\_\_\_\_-\_\_' *-- RG-85000-01 RG-42900-03*

**LIKE** 'RG-[8-9]\_\_\_\_-\_\_' *-- RG-85000-01, RG-95000-01*

**LIKE** '[O-Z]G%' *-- RG, PG, but not AG, FG, etc.*

Date Functions

**select** GETDATE() *-- 2014-01-17 07:45:59.730*

**select** DATEADD(**year**, 1, getdate()) *--2015-01-17 07:45:59.730*

**select** DATEADD(**month**, 1, getdate())*-- 2014-02-17 07:45:59.730*

**select** DATEADD(**day**, 1, getdate()) *-- 2014-01-18 07:45:59.730*

**select** DATEDIFF(**year**, '20130101', '20131024') *-- 0*

**select** DATEDIFF(**month**, '20130101', '20131024') *-- 9*

**select** DATEDIFF(**day**, '20130101', '20131024') *-- 296*

**select** DATEPART(**year**, getdate()) *-- 2014*

**select** DATEPART(**month**, getdate()) *-- 1*

**select** DATEPART(**day**, getdate()) *-- 17*

**select** **YEAR**(GETDATE()) *-- 2014*

**select** **MONTH**(GETDATE()) *-- 1*

**select** **DAY**(getdate()) *-- 17*

**select** DATENAME(**month**, getdate()) *-- January*

**select** DATENAME(**DAY**, GETDATE()) *-- 17*

**select** ISDATE('20130101') - 1

**select** ISDATE('20139999') - 0

Metadata Queries

USE BikeStore

**GO**

**SELECT** **SCHEMA\_NAME**(SCHEMA\_ID) **AS** table\_schema\_name, name **AS** **table\_name** **FROM** sys.tables **ORDER** **BY** table\_schema\_name, **table\_name**

**SELECT** name

**FROM** sys.columns

**WHERE** OBJECT\_ID = OBJECT\_ID('product.Category')

**SELECT** \* **FROM** INFORMATION\_SCHEMA.TABLES

**SELECT** \* **FROM** INFORMATION\_SCHEMA.TABLE\_PRIVILEGES *-- table specific privileges granted to accounts*

**SELECT** \* **FROM** INFORMATION\_SCHEMA.VIEW\_TABLE\_USAGE *-- tables referenced by views*

**EXEC** sys.sp\_tables

**EXEC** sys.sp\_help @objname = N'product.Model' *-- returns general info about the object*

**SELECT** SERVERPROPERTY('ProductLevel') *-- current value is 'SP1'*

**SELECT** SERVERPROPERTY('Edition') *-- Standard Edition (64-bit)*

**SELECT** @@**VERSION** *-- Microsoft SQL Server 2008 R2 (SP1) - 10.50.2550.0 (X64) Jun 11 2012 16:41:53 Copyright (c) Microsoft Corporation Standard Edition (64-bit) on Windows NT 6.1 <X64> (Build 7601: Service Pack 1) (Hypervisor)*

**SELECT** DATABASEPROPERTYEX('enterprise', 'Collation') *-- SQL\_Latin1\_General\_CP1\_CI\_AS*

**SELECT** OBJECTPROPERTY(OBJECT\_ID('product.PartNumber), 'TableHasPrimaryKey') -- 1

CROSS JOIN

**CROSS** **JOIN**

**DECLARE** @digits **TABLE** (digit INT)

**INSERT** **INTO** @digits (digit) **values** (1),(2), (3)

**SELECT** d2.digit, d1.digit *-- returns 9 record result*

**FROM** @digits d1

**CROSS** **JOIN** @digits d2

Derived Tables

**SELECT** SalesYear, **SUM**(LineTotal) **AS** TotalSold

**FROM** (

**SELECT** **YEAR**(DateSold) **AS** SalesYear, LineTotal

**FROM** sales.SkuSales

) SalesByYear

**GROUP** **BY** SalesYear

**ORDER** **BY** TotalSold **DESC**

Common Table Expression’s / CTE’s

**WITH** <CTEname> (<column1>, <column2>) *-- column list optional*

**as**

(

<subquery>

)

<**outer** query>

**WITH** SalesByYearCTE ( SalesYear, LineTotal ) *-- list of columns optional*

**AS**

(

**SELECT** **YEAR**(DateSold) **AS** SalesYear, LineTotal

**FROM** sales.SkuSales

) *-- to add more CTE's, add a comma here*

**SELECT** SalesYear, **SUM**(LineTotal) **AS** TotalSold

**FROM** SalesByYearCTE

**GROUP** **BY** SalesYear

**ORDER** **BY** TotalSold **DESC**

Recursive CTE

useful for querying tables that are self-referencing

**declare** @Location\_Table **table**

(

Location\_ID int,

Location\_Name varchar(25),

Location\_Parent int **null**

)

**insert** **into** @Location\_Table (Location\_ID, Location\_Name, Location\_Parent)

**select** 1, 'United States', **null** **union** **all**

**select** 2, 'Iowa', 1 **union** **all**

**select** 3, 'South Dakota', 1 **union** **all**

**select** 4, 'Minnesota', 1 **union** **all**

**select** 5, 'Nebraska', 1 **union** **all**

**select** 6, 'Orange City', 2 **union** **all**

**select** 7, 'Sioux Center', 2 **union** **all**

**select** 8, 'Hospers', 2 **union** **all**

**select** 9, 'Sioux Falls', 3 **union** **all**

**select** 10, 'Brookings', 3 **union** **all**

**select** 11, '102 Michigan Ave SW', 6 **union** **all**

**select** 12, '412 4th St SE', 6 **union** **all**

**select** 13, 'Utility Room', 11 **union** **all**

**select** 14, 'Kitchen', 11 **union** **all**

**select** 15, 'Chest Freezer', 13;

**with** Location\_CTE **as**

(

*-- anchor*

**select** location\_ID, location\_Name, Location\_Parent

**from** @Location\_Table **where** Location\_ID = 2 *-- iowa*

**union** **all**

*-- recurse*

**select** lt.Location\_ID, lt.Location\_Name, lt.Location\_Parent

**from** Location\_CTE lc

**join** @Location\_Table lt

**on** lt.Location\_Parent = lc.Location\_ID

)

**select** Location\_ID, Location\_Name,

(**select** Location\_Name **from** @Location\_Table **where** Location\_ID = Location\_CTE.Location\_Parent) **as** location\_parent\_name

**from** Location\_CTE;

Correlated Queries

*-- inside WHERE clause*

**SELECT** TOP 1000 BillOfMaterialsID, BillOfMaterialsRevisionID, Quantity

**FROM** manufacturing.BillOfMaterials bom\_outer

**WHERE** BillOfMaterialsID =

(

**SELECT** TOP 1 BillOfMaterialsID

**FROM** manufacturing.BillOfMaterial bom\_inner

**WHERE** bom\_inner.BillOfMaterialsRevisionID = bom\_outer.BillOfMaterialsRevisionID *-- outer reference*

**ORDER** **BY** Quantity **DESC**

)

*-- inside SELECT clause*

**SELECT** BillOfMaterialsID, BillOfMaterialsRevisionID, MaterialID, Quantity,

(Amount / (**SELECT** **SUM**(Quantity) **FROM** manufacturing.BillOfMaterials bom\_inner **WHERE** bom\_inner.BillOfMaterialsRevisionID = bom\_outer.BillOfMaterialsRevisionID) ) \* 100

**AS** percent\_of\_recipe

**FROM** manufacturing.BillOfMaterialsl bom\_outer

**WHERE** BillOfMaterialsRevisionID = 10004

EXISTS

**SELECT** mfg.Name

**FROM** product.Manufacturer mfg

**WHERE** **NOT** **EXISTS** ( **SELECT** \* **FROM** pur.PurchaseOrder po **WHERE** po.ManufacturerID = mfg.ManufacturerID)

Views

**CREATE** **VIEW** PUR.ViewManufacturersWithPurchases

**AS**

**SELECT** mfg.Name

**FROM** product.Manufacturer mfg

**WHERE** **NOT** **EXISTS** ( **SELECT** \* **FROM** pur.PurchaseOrder po **WHERE** po.ManufacturerID = mfg.ManufacturerID)

**GO**

**SELECT** \* **FROM** PUR.ViewManufacturersWithPurchases

Apply Operator

*-- Query 5 most recent orders for each active product*

**SELECT** a\_left.PartNumberID, p\_right.DateSold, p\_right.LineTotal

**FROM** product.PartNumber a\_left

**CROSS** APPLY *-- replace with OUTER APPLY to include products from left that do not have any batches*

(

**SELECT** TOP(5) PartNumberID, DateSold, LineTotal

**FROM** sales.SkuSales **AS** p\_inner

**WHERE** p\_inner.PartNumberID = a\_left.PartNumberID

**ORDER** **BY** DateSold **DESC**

) **AS** p\_right

**WHERE** a\_left.StatusID = 1

**ORDER** **BY** a\_left.PartNumberID, p\_right.DateSold **DESC**

SET Operators

<query 1> *-- column names define output columns*

[**Set** **Operation**] *-- specifying ALL will return duplicates*

<query 2>

**UNION** [**ALL**] - **return** **rows** **from** **both** queries - **using** **all** will **return** **rows** **from** **both** queries

even if they **are** duplicates

**INTERSECT** - **return** **rows** if they appear **in** **both** queries

**EXCEPT** - **return** **rows** if they appear **in** the 1st query but **not** the 2nd query

PIVOT

Pivot = rows-> columns

Pivot Process

1. Group

2. Spread

3. Aggregate

**SELECT** ...

**FROM** <source\_table\_or\_table\_expression>

PIVOT(<agg\_func>(<aggregation\_element>)

**FOR** <spreading\_element>

**IN** (<list\_of\_target\_columns>)) **AS** <result\_table\_alias>

**WITH** ProductSalesYearCTE

**AS**

(

**SELECT** PartNumberID, [1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12]

**FROM**

(

**SELECT** PartnUmberID, LineTotal, **MONTH**(DateSold) **AS** SalesMonth

**FROM** sales.SkuSales

**WHERE** **YEAR**(DateSold) = 2016

) **AS** SalesInYear

PIVOT(**SUM**(LineTotal) **FOR** SalesMonth **IN** ([1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12])) **AS** P

)

**SELECT** PartNumberID, COALESCE([1], 0) **AS** January, COALESCE([2], 0) **AS** February, COALESCE([3], 0) **AS** March, COALESCE([4], 0) **AS** April

, COALESCE([5], 0)**AS** May, COALESCE([6], 0) **AS** June, COALESCE([7], 0) **AS** July, COALESCE([8], 0) **AS** August

, COALESCE([9], 0) **AS** September, COALESCE([10], 0) **AS** October, COALESCE([11], 0) **AS** November, COALESCE([12], 0) **AS** December

**FROM** ProductSalesYearCTE

**ORDER** **BY** PartNumberID

UNPIVOT

unpivot - columns->rows ```sql

SELECT ...

FROM <source\_table\_or\_table\_expression>

UNPIVOT(<target\_col\_to\_hold\_source\_col\_values>

FOR <target\_col\_to\_hold\_source\_col\_names> IN(<list\_of\_source\_columns>)) AS

<result\_table\_alias> ```

**select** PartNumberID, SalesMonth, LineTotal

**from** sales.ViewProductSalesYear

unpivot (LineTotal **for** SalesMonth **in** (January, February, March, April, May, June, July, August,September,October,November,December) ) **as** U

Select from VALUES

**SELECT** \* **FROM**

(

**VALUES** (1,2), (2,3), (3,4), (4,5)

) **AS** t (field\_1, field2)

**SELECT** \*

**FROM** ( **VALUES**

(10003, '20090213', 4, 'B'),

(10004, '20090214', 1, 'A'),

(10005, '20090213', 1, 'C'),

(10006, '20090215', 3, 'C') )

**AS** O(orderid, orderdate, empid, custid);

Grouping Sets

**SELECT** PartNumberID, Store, **SUM**(LineTotal) **AS** TotalSold

**FROM** sales.SkuSales

**WHERE** **YEAR**(DateSold) = 2013

**group** **by**

**GROUPING** **sets**

(

(), *-- total 2013 Sales*

(Store), *-- total 2013 sales by store*

(store,PartNumberID) *-- total 2013 sales by Store + Part Number*

);

*-- ROLLUP clause*

*-- will return*

*-- 1) all Sales*

*-- 2) Each Store*

*-- 3) Each Store + SKU*

*-- 4) Does not return just the SKU because PartNumberID listed after Store in rollup clause*

**SELECT** PartNumberID, Store, **SUM**(LineTotal) **AS** TotalSold

**FROM** sales.SkuSales

**WHERE** **YEAR**(DateSold) = 2013

**GROUP** **BY** **ROLLUP**( Store, PartNumberID); *-- Left-right order important!*

Group by cube

- total sales **at** **each** store, **and** **for** **each** sku

- PartNumberID Store

**null** **null** - total 2013 sales

<value> **null** - total 2013 sales **for** sku (**any** store)

**null** <value> - total 2013 sales **at** store

<value> <value> - total 2013 sales **for** sku **at** store

**SELECT** PartNumberID, Store, **SUM**(LineTotal) **AS** TotalSold

**FROM** sales.SkuSales

**WHERE** **YEAR**(DateSold) = 2013

**GROUP** **BY** **CUBE**(Store, PartNumberID)

**ORDER** **BY** PartNumberID

Insert from Select

**INSERT** **INTO** tempdb.dbo.ModelListing ( ManufacturerCode, Name)

**SELECT** ManufacturerCode, Name

**FROM** product.Model

Insert from Sproc

**INSERT** **INTO** tempdb.dbo.ModelListing

**EXEC** product.ModelList

Select into a new table

*-- note that ModelID is an identity column, and will be created as such*

**SELECT** ModelID, ManufacturerCode, Name

**INTO** tempdb.dbo.Model

**FROM** product.Model

Bulk Insert

test file:

1,Brian,2003

2,Kit,2006

3,Dean, 2007

4,Ryan,2010

USE tempdb;

if OBJECT\_ID('dbo.SSBulk\_Insert\_Test','U') **is** **not** **null** **drop** **table** dbo.SSBulk\_Insert\_Test;

**create** **table** dbo.SSBulk\_Insert\_Test

(

id int,

person varchar(25),

year\_started int

)

BULK **INSERT** dbo.SSBulk\_Insert\_Test **FROM** 'c:**\t**emp**\S**SBulk\_Insert\_Test.txt' *-- MUST be releative to the server*

**WITH**

(

DATAFILETYPE = 'char',

FIELDTERMINATOR = ',',

ROWTERMINATOR = '**\n**'

);

Last Identity

SCOPE\_IDENTITY() *-- @@identity is legacy, do not use*

*-- scope\_identity() will be null if no inserts in the current session*

*-- do not use the following as replacement for scope\_identity()*

**select** IDENT\_CURRENT('product.Model')

DELETE

**SELECT** TOP 100 \*

**INTO** tempdb.dbo.Model

**FROM** product.Model

*-- Does not update identity seed -*

**DELETE** **FROM** tempdb.dbo.Model *-- entire table*

**DELETE** **FROM** tempdb.dbo.MOdel **WHERE** ModelID = 10000

*-- resets identity seed*

**TRUNCATE** **TABLE** tempdb.dbo.Model

*-- delete based on join*

**DELETE** **FROM** modelTemp

**FROM** tempdb.dbo.Model modelTemp

**JOIN** product.Manufacturer mfg

**ON** modelTemp.ManufacturerID = mfg.ManufacturerID

**WHERE** mfg.MaufacturerID = 10000

UPDATE based on join

**DECLARE** @table1 **TABLE** (date\_field DATE, value\_field VARCHAR(25))

**INSERT** **INTO** @table1 (date\_field, value\_field) **VALUES**

('4/1/2012', 'intial value 1'), ('4/2/2012', 'intial value 2'),

('4/3/2012', 'intial value 3'), ('4/4/2012', 'intial value 4')

**SELECT** \* **FROM** @table1

**DECLARE** @table2 **TABLE** (date\_field DATE, value\_field VARCHAR(25))

**INSERT** **INTO** @table2 (date\_field, value\_field) **VALUES**

('4/1/2012', 'modified value 1'),('4/2/2012', 'modified value 2'),

('4/3/2012', 'modified value 3'),('4/4/2012', 'modified value 4')

**UPDATE** t1

**SET** t1.value\_field = t2.value\_field

**FROM** @table1 t1

**JOIN** @table2 **AS** t2

**ON** t1.date\_field = t2.date\_field

MERGE

MERGE **INTO** <destination **table**> **as** dest

**using** <**source** **table**> **as** **source**

**on** dest.**key** = **source**.**key**

**WHEN** MATCHED **THEN**

**UPDATE** **SET**

dest.val1 = **source**.val1,

dest.val2 = **source**.val2

**WHEN** **NOT** MATCHED **THEN**

**INSERT** (**key**, val1, val2)

**VALUES** (**source**.**key**, **source**.val1, **source**.val2)

OUTPUT clause

**DECLARE** @temp\_table **TABLE** (ManufacturerCode VARCHAR(50))

**INSERT** **INTO** @temp\_table (ManufacturerCode)

**SELECT** ManufacturerCode

**FROM** product.ManufacturerCode

**WHERE** ManufacturerCode **LIKE** 'PG-%';

*-- same as above, but with optional OUTPUT clause*

**INSERT** **INTO** @temp\_table (ManufacturerCode)

**OUTPUT** inserted.ManufacturerCode *-- each displayed field must be preceded by inserted.<field name>*

**SELECT** ManufacturerCode

**FROM** product.ManufacturerCode

**WHERE** ManufacturerCode **LIKE** 'PG-%';

*-- delete*

**DELETE** **FROM** @temp\_table

**OUTPUT** deleted.ManufacturerCode

**WHERE** ManufacturerCode **LIKE** 'PG-%'

*-- use output clause to view changed values (there is no 'updated' value per se)*

**update** @temp\_table

**set** ManufacturerCode = **SUBSTRING**(ManufacturerCode, 1,4)

**output**

deleted.ManufacturerCode **as** prior\_value,

inserted.ManufacturerCode **as** new\_value

Transactions

**BEGIN** TRAN *-- if not specified, each statement runs as an implicit transaction*

*-- Statement #1*

*-- Statement #2*

...

*-- Statement N*

**COMMIT** TRAN *-- All statements since BEGIN TRAN committed to database*

**ROLLBACK** *-- ALL statements since BEGIN TRAN are canceled, no data will be changed*

Detailed Transaction Template

*-- http://stackoverflow.com/questions/2073737/nested-stored-procedures-containing-try-catch-rollback-pattern/2074139#2074139*

**SET** XACT\_ABORT, NOCOUNT **ON**

**DECLARE** @starttrancount INT

**BEGIN** TRY

**SELECT** @starttrancount = @@TRANCOUNT

IF @starttrancount = 0

**BEGIN** TRANSACTION

*-- Do work, call nested sprocs*

IF @starttrancount = 0

**COMMIT** TRANSACTION

**END** TRY

**BEGIN** CATCH

IF XACT\_STATE() <> 0 **AND** @starttrancount = 0

**ROLLBACK** TRANSACTION

**DECLARE** @error INT, @message VARCHAR(4000)

**SET** @error = ERROR\_NUMBER()

**SET** @message = ERROR\_MESSAGE()

RAISERROR('<Sproc Name> : %d: %s', 16, 1, @error, @message)

**END** CATCH

Locking

- Exclusive Locks

- generated by update operations

- no other session can update or read an item that has an exclusive lock

- Shared Locks

- generated by selects

- Isolation levels affect how sql interacts with shared locks:

READ UNCOMMITTED - SELECT does not generate shared locks - dirty reads

READ COMMITTED (default) - SELECT requires shared locks

REPEATABLE READ - shared lock open for the entire transaction

SERIALIZABLE - locks range of keys returned to prevent phantom records

SNAPSHOT - reads previous row reivsion stored in tempdb

READ COMMITTED SNAPSHOT - Gets the last committed version of the row that was available when the statement started

- can be set at database level or transaction level:

SET TRANSACTION ISOLATION LEVEL <name>

Variables

**DECLARE** @i **AS** INT;

**SET** @i = 10;

**DECLARE** @i **AS** INT = 10; *-- only works in SS2008 or higher*

**DECLARE** @ManufacturerCode VARCHAR(25)

**DECLARE** @Name VARCHAR(100)

*-- select into variables*

**SELECT** @ManufacturerCode = ManufacturerCode,

@Name = Name

**FROM** product.Model

**WHERE** ModelID = 10000

Flow Control

*-- single statements*

IF <expression>

*-- will execute if expression is true*

**ELSE**

*-- will execute if expression is false or unknown*

*-- Multiple statements*

IF <expression>

**BEGIN**

**END**

**ELSE**

**BEGIN**

**END**

WHILE <expression

**begin**

**end**

*-- FOR replacement*

**DECLARE** @i **AS** INT;

**SET** @i = 1

WHILE @i <= 10

**BEGIN**

PRINT @i;

**SET** @i = @i + 1;

**END**;

Cursors

**DECLARE** @ModelID INT,

@ManufacturerCode VARCHAR(25)

**DECLARE** **C** **CURSOR** FAST\_FORWARD */\* read only, forward only \*/* **FOR**

**SELECT** ModelID, ManufacturerCode

**FROM** Product.Model

**ORDER** **BY** ModelID

**OPEN** **C**

**FETCH** **NEXT** **FROM** **C** **INTO** @ModelID, @ManufacturerCode;

WHILE @@FETCH\_STATUS = 0

**BEGIN**

*-- do work*

**FETCH** **NEXT** **FROM** **C** **INTO** @ModelID, @ManufacturerCode;

**END**

**CLOSE** **C**;

**DEALLOCATE** **C**;

Min Key approach

**DECLARE** @ModelID INT

**DECLARE** @ManufacturerCode VARCHAR(25)

**SET** @ModelID = (**SELECT** **MIN**(ModelID) **FROM** product.model)

WHILE @ModelID **IS** **NOT** **NULL**

**BEGIN**

**SELECT** @ManufacturerCode = ManufacturerCode

**FROM** product.model

**WHERE** ModelID = @ModelID

*-- do work*

**SET** @ModelID = (**SELECT** **MIN**(ModelID) **FROM** product.model **WHERE** ModelID > @ModelID)

**END**

Temporary Tables

**CREATE** **TABLE** #TempTable

(

ModelID INT,

ManufacturerCode VARCHAR(25)

)

**CREATE** **TABLE** ##GlobalTempTable

(

ModelID INT,

ManufacturerCode VARCHAR(25)

)

Table Variables

**DECLARE** @TempTable **TABLE**

(

ModelID INT,

ManufacturerCode VARCHAR(25)

);

Table Types

IF TYPE\_ID('dbo.OrderTotalsByYear') **IS** **NOT** **NULL**

**DROP** **TYPE** dbo.OrderTotalsByYear;

**CREATE** **TYPE** dbo.OrderTotalsByYear **AS** **TABLE**

(

orderyear INT **NOT** **NULL** **PRIMARY** **KEY**,

qty INT **NOT** **NULL**

);

**DECLARE** @MyOrderTotalsByYear **AS** dbo.OrderTotalsByYear;

Dynamic SQL

*-- EXEC*

**DECLARE** @**sql** **AS** VARCHAR(100);

**SET** @**sql** = 'PRINT **''**This message was printed by a dynamic SQL batch.**''**;';

**EXEC**(@**sql**);

*-- sp\_execute - supports params*

**DECLARE** @**sql** NVARCHAR(**MAX**);

**SET** @**sql** = N'select \* from product.model

where Name LIKE **''**%**''** + @Name + **''**%**''** AND CategoryId = @CategoryId'

**EXEC** sp\_executesql

@stmt = @**sql**,

@params = N'@Name as varchar(100), @CategoryId as int',

@Name = 'John Deere',

@CategoryId = 2

User Defined Functions

Scalar

**CREATE** **FUNCTION** dbo.fn\_age

(

@birthdate **AS** DATETIME,

@eventdate **AS** DATETIME

)

**RETURNS** INT *-- <-- Defines as scalar*

**AS**

**BEGIN**

**RETURN**

DATEDIFF(**year**, @birthdate, @eventdate)

- **CASE** **WHEN** 100 \* **MONTH**(@eventdate) + **DAY**(@eventdate)

< 100 \* **MONTH**(@birthdate) + **DAY**(@birthdate)

**THEN** 1 **ELSE** 0

**END**

**END**

**GO**

Table Value

**ALTER** **FUNCTION** [dbo].[Split\_MultiValue\_Parameter]

( @delimitedString VARCHAR(**MAX**)

,@**delimiter** VARCHAR(1)

)

**RETURNS** @**Table** **TABLE** (VALUE VARCHAR(100)) *-- <-- Defines as table*

**AS**

**BEGIN**

**DECLARE** @tempString VARCHAR(**MAX**);

**SET** @tempString = **ISNULL**(@delimitedString,'') + @**Delimiter**;

WHILE LEN(@tempString) > 0

**BEGIN**

**INSERT** **INTO** @**Table**

**SELECT** **SUBSTRING**(@tempString,1,

CHARINDEX(@**Delimiter**,@tempString)-1);

**SET** @tempString = **RIGHT**(@tempString,

LEN(@tempString)-CHARINDEX(@**Delimiter**,@tempString)) ;

**END**

**RETURN**;

**END**

Stored Procedures

**CREATE** **PROCEDURE** product.ModelList

**AS**

**BEGIN**

**SET** NOCOUNT **ON**;

**SELECT** ModelId, Name, ManufacturerCode, CategoryId, Description

**FROM** product.Model

**END**

Triggers

DML - Data Modification

**ALTER** **TRIGGER** product.trProductPartNumberDateModified **ON** product.PartNumber **FOR** **UPDATE** **AS**

**DECLARE** @PartNumberID INT;

**SELECT** @PartNumberID = PartNumberId **FROM** Inserted;

**UPDATE** Product.PartNumber

**SET** DateModified = GETDATE()

**WHERE** PartNumberID = @PartNumberID;

Structural modification

**CREATE** **TRIGGER** [DDL\_Notify]

**ON** **DATABASE**

**FOR** DROP\_TABLE, ALTER\_TABLE,CREATE\_TABLE, DROP\_FUNCTION, ALTER\_FUNCTION, CREATE\_FUNCTION,

DROP\_PROCEDURE, ALTER\_PROCEDURE, CREATE\_PROCEDURE

**AS**

*-- actions*

Try / Catch

**BEGIN** TRY

**TRUNCATE** **TABLE** tempdbo.dbo.DoesNotExist

*-- following statement will not execute*

**select** top 1 \* **from** product.model

**END** TRY

**BEGIN** CATCH

*-- 4701 Cannot find the object "DoesNotExist" because it does not exist or you do not have permissions.*

**SELECT** ERROR\_NUMBER(), ERROR\_MESSAGE()

**END** CATCH

Template for basic table + relationships

**CREATE** **TABLE** tempdb.dbo.temp1

(

Temp1\_pk INT **IDENTITY**(1,1) **NOT** **NULL**

**CONSTRAINT** PK\_dbo\_Temp1 **PRIMARY** **KEY** CLUSTERED,

VALUE VARCHAR(25)

)

**GO**

**CREATE** **TABLE** tempdb.dbo.temp2

(

Temp2\_pk INT **IDENTITY**(1,1) **NOT** **NULL**

**CONSTRAINT** pk\_dbo\_temp2 **PRIMARY** **KEY** CLUSTERED,

Temp1\_fk INT **NOT** **NULL**

**CONSTRAINT** FK\_dbo\_Temp2\_dbo\_Temp1 **FOREIGN** **KEY** (Temp1\_fk)

**REFERENCES** dbo.temp1(Temp1\_pk),

VALUE VARCHAR(25) **NULL**

)

**GO**

Identity Insert

**SET** IDENTITY\_INSERT product.Model **ON**

**INSERT** **INTO** product.Model (ModelID, Name)

**VALUES** (12345, 'Test')

**SET** IDENTITY\_INSERT product.Model **OFF**

SQL CHEAT SHEET created by Tomi Mester I originally created this cheat sheet for my SQL course and workshop participants.\* But I have decided to open-source it and make it available for everyone who wants to learn SQL. It's designed to give you a meaningful structure but also to let you add your own notes (that's why the empty boxes are there). It starts from the absolute basics (SELECT \* FROM table\_name;) and guides you to the intermediate level (JOIN, HAVING, subqueries). I added everything that you will need as a data analyst/ scientist. The ideal use case of this cheat sheet is that you print it in color and keep it next to you while you are learning and practicing SQL on your computer. Enjoy! Cheers, Tomi Mester \*The workshops and courses I mentioned: Online SQL tutorial (free): data36.com/sql-tutorial Live SQL workshop: data36.com/sql-workshop Practice SQL - an online SQL course for practicing: data36.com/practice-sql SQL CHEAT SHEET BASE QUERY SELECT \* FROM table\_name; This query returns every column and every row of the table called table\_name. SELECT \* FROM table\_name LIMIT 10; It returns every column and the first 10 rows from table\_name. SELECTING SPECIFIC COLUMNS SELECT column1, column2, column3 FROM table\_name; This query returns every row of column1, column2 and column3 from table\_name. DATA TYPES IN SQL In SQL we have more than 40 different data types. But these seven are the most important ones: 1. Integer. A whole number without a fractional part. E.g. 1, 156, 2012412 2. Decimal. A number with a fractional part. E.g. 3.14, 3.141592654, 961.1241250 3. Boolean. A binary value. It can be either TRUE or FALSE. 4. Date. Speaks for itself. You can also choose the format. E.g. 2017-12-31 5. Time. You can decide the format of this, as well. E.g. 23:59:59 6. Timestamp. The date and the time together. E.g. 2017-12-31 23:59:59 7. Text. This is the most general data type. But it can be alphabetical letters only, or a mix of letters and numbers and any other characters. E.g. hello, R2D2, Tomi, 124.56.128.41 CREATED BY TOMI MESTER | DATA36.COM 1 [your notes] SQL CHEAT SHEET FILTERING (the WHERE CLAUSE) SELECT \* FROM table\_name WHERE column1 = 'expression'; "Horizontal filtering." This query returns every column from table\_name - but only those rows where the value in column1 is 'expression'. Obviously this can be something other than text: a number (integer or decimal), date or any other data format, too. ADVANCED FILTERING Comparison operators help you compare two values. (Usually a value that you define in your query and values that exist in your SQL table.) Mostly, they are mathematical symbols, with a few exceptions: CREATED BY TOMI MESTER | DATA36.COM 2 Comparison operator What does it mean? = Equal to <> Not equal to != Not equal to < Less than <= Less than or equal to > Greater than >= Greater than or equal to LIKE ‘%expression%’ Contains ‘expression’ IN (‘exp1’, ‘exp2’, ‘exp3’) Contains any of ‘exp1’, ‘exp2’, or ‘exp3’ SQL CHEAT SHEET A few examples: SELECT \* FROM table\_name WHERE column1 != 'expression'; This query returns every column from table\_name, but only those rows where the value in column1 is NOT 'expression'. SELECT \* FROM table\_name WHERE column2 >= 10; It returns every column from table\_name, but only those rows where the value in column2 is greater or equal to 10. SELECT \* FROM table\_name WHERE column3 LIKE ‘%xzy%’; It returns every column from table\_name, but only those rows where the value in column3 contains the 'xyz' string. MULTIPLE CONDITIONS You can use more than one condition to filter. For that, we have two logical operators: OR, AND. SELECT \* FROM table\_name WHERE column1 != ‘expression’ AND column3 LIKE ‘%xzy%’; This query returns every column from table\_name, but only those rows where the value in column1 is NOT ‘expression’ AND the value in column3 contains the 'xyz' string. SELECT \* FROM table\_name WHERE column1 != ‘expression’ OR column3 LIKE ‘%xzy%’; This query returns every column from table\_name, but only those rows where the value in column1 is NOT ‘expression’ OR the value in column3 contains the 'xyz' string. CREATED BY TOMI MESTER | DATA36.COM 3 SQL CHEAT SHEET PROPER FORMATTING You can use line breaks and indentations for nicer formatting. It won't have any effect on your output. Be careful and put a semicolon at the end of the query though! SELECT \* FROM table\_name WHERE column1 != 'expression' AND column3 LIKE '%xzy%' LIMIT 10; SORTING VALUES SELECT \* FROM table\_name ORDER BY column1; This query returns every row and column from table\_name, ordered by column1, in ascending order (by default). SELECT \* FROM table\_name ORDER BY column1 DESC; This query returns every row and column from table\_name, ordered by column1, in descending order. UNIQUE VALUES SELECT DISTINCT(column1) FROM table\_name; It returns every unique value from column1 from table\_name. CREATED BY TOMI MESTER | DATA36.COM 4 SQL CHEAT SHEET CORRECT KEYWORD ORDER SQL is extremely sensitive to keyword order. So make sure you keep it right: 1. SELECT 2. FROM 3. WHERE 4. ORDER BY 5. LIMIT SQL FUNCTIONS FOR AGGREGATION In SQL, there are five important aggregate functions for data analysts/scientists: • COUNT() • SUM() • AVG() • MIN() • MAX() A few examples: SELECT COUNT(\*) FROM table\_name WHERE column1 = 'something'; It counts the number of rows in the SQL table in which the value in column1 is 'something'. SELECT AVG(column1) FROM table\_name WHERE column2 > 1000; It calculates the average (mean) of the values in column1, only including rows in which the value in column2 is greater than 1000. CREATED BY TOMI MESTER | DATA36.COM 5 SQL CHEAT SHEET SQL GROUP BY The GROUP BY clause is usually used with an aggregate function (COUNT, SUM, AVG, MIN, MAX). It groups the rows by a given column value (specified after GROUP BY) then calculates the aggregate for each group and returns that to the screen. SELECT column1, COUNT(column2) FROM table\_name GROUP BY column1; This query counts the number of values in column2 - for each group of unique column1 values. SELECT column1, SUM(column2) FROM table\_name GROUP BY column1; This query sums the number of values in column2 - for each group of unique column1 values. SELECT column1, MIN(column2) FROM table\_name GROUP BY column1; This query finds the minimum value in column2 - for each group of unique column1 values. SELECT column1, MAX(column2) FROM table\_name GROUP BY column1; This query finds the maximum value in column2 - for each group of unique column1 values. CREATED BY TOMI MESTER | DATA36.COM 6 SQL CHEAT SHEET SQL ALIASES You can rename columns, tables, subqueries, anything. SELECT column1, COUNT(column2) AS number\_of\_values FROM table\_name GROUP BY column1; This query counts the number of values in column2 - for each group of unique column1 values. Then it renames the COUNT(column2) column to number\_of\_values. SQL JOIN You can JOIN two (or more) SQL tables based on column values. SELECT \* FROM table1 JOIN table2 ON table1.column1 = table2.column1; This joins table1 and table2 values - for every row where the value of column1 from table1 equals the value of column1 from table2. Detailed explanation here: https://data36.com/sql-join-data-analysis-tutorial-ep5/ CREATED BY TOMI MESTER | DATA36.COM 7 SQL CHEAT SHEET SQL HAVING The execution order of the different SQL keywords doesn't allow you to filter with the WHERE clause on the result of an aggregate function (COUNT, SUM, etc.). This is because WHERE is executed before the aggregate functions. But that's what HAVING is for: SELECT column1, COUNT(column2) FROM table\_name GROUP BY column1 HAVING COUNT(column2) > 100; This query counts the number of values in column2 - for each group of unique column1 values. It returns only those results where the counted value is greater than 100. Detailed explanation and examples here: https://data36.com/sql-data-analysisadvanced-tutorial-ep6/ CORRECT KEYWORD ORDER AGAIN SQL is extremely sensitive to keyword order. So make sure you keep it right: 1. SELECT 2. FROM 3. JOIN (ON) 4. WHERE 5. GROUP BY 6. HAVING 7. ORDER BY 8. LIMIT CREATED BY TOMI MESTER | DATA36.COM 8 SQL CHEAT SHEET SUBQUERIES You can run SQL queries within SQL queries. (Called subqueries.) Even queries within queries within queries. The point is to use the result of one query as an input value of another query. Example: SELECT COUNT(\*) FROM (SELECT column1, COUNT(column2) AS inner\_number\_of\_values FROM table\_name GROUP BY column1) AS inner\_query WHERE inner\_number\_of\_values > 100; The inner query counts the number of values in column2 - for each group of unique column1 values. Then the outer query uses the inner query's results and counts the number of values where inner\_number\_of\_values are greater than 100. (The result is one number.) Detailed explanation here: https://data36.com/sql-data-analysis-advancedtutorial-ep6/ CREATED BY TOMI MESTER | DATA36.COM 9 SQL CHEAT SHEET CREATED BY Tomi Mester from Data36.com Tomi Mester is a data analyst and researcher. He worked for Prezi, iZettle and several smaller companies as an analyst/consultant. He’s the author of the Data36 blog where he writes posts and tutorials on a weekly basis about data science, ABtesting, online research and data coding. He’s an O’Reilly author and presenter at TEDxYouth, Barcelona E-commerce Summit, Stockholm Analyticsdagarna and more. WHERE TO GO NEXT Find company workshops, online tutorials and online video courses on my website: https://data36.com Subscribe to my Newsletter list for useful stuff like this: https://data36.com/newsletter Online SQL tutorial (free): data36.com/sql-tutorial Live SQL workshop: data36.com/sql-workshop Practice SQL - an online SQL course for practicing: data36.com/practice-sql CREATED BY TOMI MESTER | DATA36.COM 10

## Querying**data** from a table

Query data in columns c1, c2 from a table

|  |  |
| --- | --- |
| 1 | SELECT c1, c2 FROM t; |

Query all rows and columns from a table

|  |  |
| --- | --- |
| 1 | SELECT \* FROM t; |

Query data and filter rows with a condition

|  |  |
| --- | --- |
| 1  2 | SELECT c1, c2 FROM t  WHERE condition; |

Query distinct rows from a table

|  |  |
| --- | --- |
| 1  2 | SELECT DISTINCT c1 FROM t  WHERE condition; |

Sort the result set in ascending or descending order

|  |  |
| --- | --- |
| 1  2 | SELECT c1, c2 FROM t  ORDER BY c1 ASC [DESC]; |

Skip *offset* of rows and return the next n rows

|  |  |
| --- | --- |
| 1  2  3 | SELECT c1, c2 FROM t  ORDER BY c1  LIMIT n OFFSET offset; |

Group rows using an aggregate function

|  |  |
| --- | --- |
| 1  2  3 | SELECT c1, aggregate(c2)  FROM t  GROUP BY c1; |

Filter groups using HAVING clause

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT c1, aggregate(c2)  FROM t  GROUP BY c1  HAVING condition; |

## Querying**from** multiple tables

Inner join t1 and t2

|  |  |
| --- | --- |
| 1  2  3 | SELECT c1, c2  FROM t1  INNER JOIN t2 ON condition; |

Left join t1 and t1

|  |  |
| --- | --- |
| 1  2  3 | SELECT c1, c2  FROM t1  LEFT JOIN t2 ON condition; |

Right join t1 and t2

|  |  |
| --- | --- |
| 1  2  3 | SELECT c1, c2  FROM t1  RIGHT JOIN t2 ON condition; |

Perform full outer join

|  |  |
| --- | --- |
| 1  2  3 | SELECT c1, c2  FROM t1  FULL OUTER JOIN t2 ON condition; |

Produce a Cartesian product of rows in tables

|  |  |
| --- | --- |
| 1  2  3 | SELECT c1, c2  FROM t1  CROSS JOIN t2; |

Another way to perform cross join

|  |  |
| --- | --- |
| 1  2 | SELECT c1, c2  FROM t1, t2; |

Join t1 to itself using INNER JOIN clause

|  |  |
| --- | --- |
| 1  2  3 | SELECT c1, c2  FROM t1 A  INNER JOIN t2 B ON condition; |

## **Using SQL Operators**

Combine rows from two queries

|  |  |
| --- | --- |
| 1  2  3 | SELECT c1, c2 FROM t1  UNION [ALL]  SELECT c1, c2 FROM t2; |

Return the intersection of two queries

|  |  |
| --- | --- |
| 1  2  3 | SELECT c1, c2 FROM t1  INTERSECT  SELECT c1, c2 FROM t2; |

Subtract a result set from another result set

|  |  |
| --- | --- |
| 1  2  3 | SELECT c1, c2 FROM t1  MINUS  SELECT c1, c2 FROM t2; |

Query rows using pattern matching %, \_

|  |  |
| --- | --- |
| 1  2 | SELECT c1, c2 FROM t1  WHERE c1 [NOT] LIKE pattern; |

Query rows in a list

|  |  |
| --- | --- |
| 1  2 | SELECT c1, c2 FROM t  WHERE c1 [NOT] IN value\_list; |

Query rows between two values

|  |  |
| --- | --- |
| 1  2 | SELECT c1, c2 FROM t  WHERE  c1 BETWEEN low AND high; |

Check if values in a table is NULL or not

|  |  |
| --- | --- |
| 1  2 | SELECT c1, c2 FROM t  WHERE  c1 IS [NOT] NULL; |

## **Managing tables**

Create a new table with three columns

|  |  |
| --- | --- |
| 1  2  3  4  5 | CREATE TABLE t (       id INT PRIMARY KEY,       name VARCHAR NOT NULL,       price INT DEFAULT 0  ); |

Delete the table from the database

|  |  |
| --- | --- |
| 1 | DROP TABLE t ; |

Add a new column to the table

|  |  |
| --- | --- |
| 1 | ALTER TABLE t ADD column; |

Drop column c from the table

|  |  |
| --- | --- |
| 1 | ALTER TABLE t DROP COLUMN c ; |

Add a constraint

|  |  |
| --- | --- |
| 1 | ALTER TABLE t ADD constraint; |

Drop a constraint

|  |  |
| --- | --- |
| 1 | ALTER TABLE t DROP constraint; |

Rename a table from t1 to t2

|  |  |
| --- | --- |
| 1 | ALTER TABLE t1 RENAME TO t2; |

Rename column c1 to c2

|  |  |
| --- | --- |
| 1 | ALTER TABLE t1 RENAME c1 TO c2 ; |

Remove all data in a table

|  |  |
| --- | --- |
| 1 | TRUNCATE TABLE t; |

## Using SQL**constraints**

Set c1 and c2 as a primary key

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE TABLE t(      c1 INT, c2 INT, c3 VARCHAR,      PRIMARY KEY (c1,c2)  ); |

Set c2 column as a foreign key

|  |  |
| --- | --- |
| 1  2  3  4  5 | CREATE TABLE t1(      c1 INT PRIMARY KEY,      c2 INT,      FOREIGN KEY (c2) REFERENCES t2(c2)  ); |

Make the values in c1 and c2 unique

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE TABLE t(      c1 INT, c1 INT,      UNIQUE(c2,c3)  ); |

Ensure c1 > 0 and values in c1 >= c2

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE TABLE t(    c1 INT, c2 INT,    CHECK(c1> 0 AND c1 >= c2)  ); |

Set values in c2 column not NULL

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE TABLE t(       c1 INT PRIMARY KEY,       c2 VARCHAR NOT NULL  ); |

## **Modifying** Data

Insert one row into a table

|  |  |
| --- | --- |
| 1  2 | INSERT INTO t(column\_list)  VALUES(value\_list); |

Insert multiple rows into a table

|  |  |
| --- | --- |
| 1  2  3 | INSERT INTO t(column\_list)  VALUES (value\_list),         (value\_list), …; |

Insert rows from t2 into t1

|  |  |
| --- | --- |
| 1  2  3 | INSERT INTO t1(column\_list)  SELECT column\_list  FROM t2; |

Update new value in the column c1 for all rows

|  |  |
| --- | --- |
| 1  2 | UPDATE t  SET c1 = new\_value; |

Update values in the column c1, c2 that match the condition

|  |  |
| --- | --- |
| 1  2  3  4 | UPDATE t  SET c1 = new\_value,          c2 = new\_value  WHERE condition; |

Delete all data in a table

|  |  |
| --- | --- |
| 1 | DELETE FROM t; |

Delete subset of rows in a table

|  |  |
| --- | --- |
| 1  2 | DELETE FROM t  WHERE condition; |

## **Managing Views**

Create a new view that consists  of c1 and c2

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE VIEW v(c1,c2)  AS  SELECT c1, c2  FROM t; |

Create a new view with check option

|  |  |
| --- | --- |
| 1  2  3  4  5 | CREATE VIEW v(c1,c2)  AS  SELECT c1, c2  FROM t;  WITH [CASCADED | LOCAL] CHECK OPTION; |

Create a recursive view

|  |  |
| --- | --- |
| 1  2  3  4  5 | CREATE RECURSIVE VIEW v  AS  select-statement -- anchor part  UNION [ALL]  select-statement; -- recursive part |

Create a temporary view

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE TEMPORARY VIEW v  AS  SELECT c1, c2  FROM t; |

Delete a view

|  |  |
| --- | --- |
| 1 | DROP VIEW view\_name; |

## Managing indexes

Create an index on c1 and c2 of the t table

|  |  |
| --- | --- |
| 1  2 | CREATE INDEX idx\_name  ON t(c1,c2); |

Create a unique index on c3, c4 of the t table

|  |  |
| --- | --- |
| 1  2 | CREATE UNIQUE INDEX idx\_name  ON t(c3,c4) |

Drop an index

|  |  |
| --- | --- |
| 1 | DROP INDEX idx\_name; |

## Managing**triggers**

Create or modify a trigger

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE OR MODIFY TRIGGER trigger\_name  WHEN EVENT  ON table\_name TRIGGER\_TYPE  EXECUTE stored\_procedure; |

**WHEN**

* **BEFORE**– invoke before the event occurs
* **AFTER**– invoke after the event occurs

**EVENT**

* **INSERT**– invoke for INSERT
* **UPDATE**– invoke for UPDATE
* **DELETE**– invoke for DELETE

**TRIGGER\_TYPE**

* **FOR EACH ROW**
* **FOR EACH STATEMENT**

Delete a specific trigger

|  |  |
| --- | --- |
| 1 | DROP TRIGGER trigger\_name; |