Aggregation

DATA SCIENCE

skill academy
by testbook

Aggregation

Chapter 1: Functions (Aggregate)

Section 1.1: Conditional aggregation

Payments Table

- **Customer Payment_type Amount**

Peter	Credit	100
Peter	Credit	300
John	Credit	1000
John	Debit	500

Result:

- **Customer Credit Debit**

Peter	400	0
John	1000	50

Result:

- **Customer credit_transaction_count debit_transaction_count**

Peter	2	0
John	1	1

Section 1.2: List Concatenation

List Concatenation aggregates a column or expression by combining the values into a single string for each group.

A string to delimit each value (either blank or a comma when omitted) and the order of the values in the result can be specified. While it is not part of the SQL standard, every major relational database vendor supports it in their own way.

MySQL

```
SELECT ColumnA
        , GROUP_CONCAT(ColumnB ORDER BY ColumnB SEPARATOR ',') AS ColumnBs
FROM TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

Oracle & DB2

```
SELECT ColumnA
        , LISTAGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
FROM TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

PostgreSQL

```
SELECT ColumnA
        , STRING_AGG(ColumnB, ',' ORDER BY ColumnB) AS ColumnBs FROM TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

SQLite

without ordering:

```
SELECT ColumnA
        , GROUP_CONCAT(ColumnB, ',') AS ColumnBs FROM TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

Section 1.3: SUM

Sumfunction sum the value of all the rows in the group. If the group by clause is omitted then sums all the rows.

```
SELECT SUM(salary) AS TotalSalary

FROM employees;
```

Total Salary

2500

```
SELECT DepartmentId, SUM(salary) TotalSalary

FROM employees

GROUP BY DepartmentId;
```

DepartmentId	Total Salary
1	2000
2	500

Section 1.4: AVG()

The aggregate function AVG() returns the average of a given expression, usually numeric values in a column. Assume we have a table containing the yearly calculation of population in cities across the world. The records for New York City look similar to the ones below:

EXAMPLE TABLE

city_name	population	year
New York City	8,550,405	2015
New York City	...	...
New York City	8,000,906	2005

To select the average population of the New York City, USA from a table containing city names, population measurements, and measurement years for last ten years:

QUERY

```
select city_name, AVG(population) avg_population from city_population

where city_name = 'NEW YORK CITY';
```

Notice how measurement year is absent from the query since population is being averaged over time.

RESULTS

city_name avg_population

New York City 8,250,754

Note: *The AVG() function will convert values to numeric types. This is especially important to keep in mind when working with dates.*

Section 1.5: Count

You can count the number of rows:

```
SELECT  count(*)  TotalRows
FROM employees;
```

TotalRows

4

Or count the employees per department:

```
SELECT DepartmentId, count(*) NumEmployees
FROM employees

GROUP BY DepartmentId;
```

DepartmentId NumEmployees

1	3
2	1

You can count over a column/expression with the effect that will not count the NULLvalues:

```
SELECT count(ManagerId) mgr

FROM EMPLOYEES;
```

mgr

3

(There is one null value manager ID column)

You can also use **DISTINCT** inside of another function such as **COUNT** to only find the **DISTINCT** members of the set to perform the operation on.

For example:

```
SELECT COUNT (Continent Code) All Count
,           COUNT(DISTINCT Continent Code) Single Count
FROM Countries;
```

Will return different values. The *Single Count* will only Count individual Continents once, while the *All Count* will include duplicates.

Continent Code

OC

EU

AS

N

A

N

A

AF

AF

All Count: 7 Single Count: 5

Section 1.6: Min

Find the smallest value of column:

```
select min(age)
from employee;
```

Above example will return smallest value for column ageof employee table. Syntax:

```
SELECT MIN(column_name)

FROM table_name;
```

Section 1.7: Max

Find the maximum value of column:

```
SELECT MAX(column_name)

FROM table_name;
```

Above example will return largest value for column ageof employeetable. Syntax:

```
select max(age)

from employee;
```

Chapter 2: CASE

The CASE expression is used to implement if-then logic.

Section 2.1: Use CASE to COUNT the number of rows in a column match a condition

Use Case

CASEcan be used in conjunction with SUMto return a count of only those items matching a pre-defined condition. (This is similar to COUNTIFin Excel.)

The trick is to return binary results indicating matches, so the "1"s returned for matching entries can be summed for a count of the total number of matches.

Given this table ItemSales, let's say you want to learn the total number of items that have been categorized as "Expensive":

ID	ITEM	PRICE	PRICE RATING
1	100	34.5	Expensive
2	145	2.3	Cheap
3	100	34.5	Expensive
4	100	34.5	Expensive
5	145	10	Affordable

Query

```
SELECT
    COUNT(Id) AS ItemsCount,
    SUM ( CASE
            WHEN PriceRating = 'Expensive' THEN 1
            ELSE 0
          END
        ) AS ExpensiveItemsCount
FROM ItemSales
```

Results:

Items Count	5
Expensive Items Count	3

Alternative:

```
SELECT
    COUNT(Id) as ItemsCount,
    SUM (
        CASE PriceRating
            WHEN 'Expensive' THEN
            1 ELSE 0
        END
       ) AS ExpensiveItemsCount
FROM ItemSales
```

Section 2.2: Searched CASE in SELECT (Matches a boolean expression)

The *searched* CASE returns results when a *boolean* expression is TRUE.

(This differs from the simple case, which can only check for equivalency with an input.)

```
SELECT Id, ItemId, Price,

CASE WHEN Price < 10 THEN 'CHEAP'

     WHEN  Price  <  20  THEN  'AFFORDABLE'
     ELSE 'EXPENSIVE'

  END  AS  PriceRating
FROM ItemSales
```

ID	ITEM	PRICE	PRICE RATING
1	100	34.5	Expensive
2	145	2.3	Cheap
3	100	34.5	Expensive
4	100	34.5	Expensive
5	145	10	Affordable

Section 2.3: CASE in a clause ORDER BY

We can use 1,2,3.. to determine the type of order:

```
SELECT     *     FROM
DEPT ORDER BY

CASE DEPARTMENT

    WHEN    'MARKETING'    THEN
        1 WHEN 'SALES' THEN 2

    WHEN   'RESEARCH'   THEN   3
    WHEN  'INNOVATION'  THEN  4
    ELSE         5

    END,
    CITY
```

ID	REGION	CITY	DEPARTMENT	EMPLOYEES NUMBER
12	New England	Boston	Marketing	9
15	West	San Francisco	Marketing	12
9	Midwest	Chicago	Sales	8
14	Mid Atlantic	New York	Sales	12
5	West	Los Angeles	Research	11
10	Mid Atlantic	Philadelphia	Research	13
4	Midwest	Chicago	Innovation	11
2	Midwest	Detroit	Human Resource	9

Section 2.4: Shorthand CASE in SELECT

CASE's shorthand variant evaluates an expression (usually a column) against a series of values. This variant is a bit shorter, and saves repeating the evaluated expression over and over again. The ELSE clause can still be used, though:

```
SELECT Id, ItemId, Price,
    CASE Price WHEN 5    THEN 'CHEAP'
        WHEN 15 THEN 'AFFORDABLE' ELSE  'EXPENSIVE'
    END as Price Rating
FROM Item Sales
```

A word of caution. It's important to realize that when using the short variant the entire statement is evaluated at each WHEN. Therefore the following statement:

```
SELECT
    CASE ABS(CHECKSUM(NEWID())) % 4
        WHEN 0 THEN 'Dr'
        WHEN 1 THEN 'Master'
        WHEN 2 THEN 'Mr' WHEN 3
        THEN 'Mrs'
END
```

may produce a NULLresult. That is because at each WHENNEWID()is being called again with a new result. Equivalent to:

```
SELECT

    CASE

        WHEN ABS(CHECKSUM(NEWID())) % 4 = 0 THEN
        'Dr'  WHEN  ABS(CHECKSUM(NEWID()))  %  4  =  1
        THEN 'Master' WHEN ABS(CHECKSUM(NEWID()))
        %   4   =   2   THEN   'Mr'   WHEN   ABS(CHECKSUM
        (NEWID())) % 4 = 3 THEN'Mrs'

END
```

Therefore, it can miss all the WHENcases and result as NULL.

Section 2.5: CASE use for NULL values ordered last

in this way '0' representing the known values are ranked first, '1' representing the NULL values are sorted by the last:

```
 SELECT ID

        ,REGION

        ,CITY

        ,DEPARTMENT

        ,EMPLOYEES_NUMBER

    FROM DEPT

    ORDER BY

    CASE  WHEN  REGION  IS  NULL
    THEN 1 ELSE 0

    END, REGION
```

ID	REGION	CITY	DEPARTMENT	EMPLOYEES NUMBER
14	Mid Atlantic	New York	Sales	12
9	Midwest	Chicago	Sales	8

12	New England	Boston	Marketing	9
5	West	Los Angeles	Research	11
15	Null	San Francisco	Marketing	12
4	Null	Chicago	Innovation	11
2	Null	Detroit	Human Resource	9

Section 2.6: CASE in ORDER BY clause to sort records by lowest value of 2 columns

Imagine that you need sort records by lowest value of either one of two columns. Some databases could use a non- aggregated MIN()or LEAST()function for this (... ORDER BY MIN(Date1, Date2)), but in standard SQL, you have to use a CASEexpression.

The CASEexpression in the query below looks at the Date1and Date2columns, checks which column has the lower value, and sorts the records depending on this value.

Sample data

ID	DATE 1	DATE 2
1	2017-01-01	2017-01-31
2	2017-01-31	2017-01-03
3	2017-01-31	2017-01-02
4	2017-01-06	2017-01-31
5	2017-01-31	2017-01-05
6	2017-01-04	2017-01-31

Query

```
SELECT  Id,  Date1,  Date2
FROM YourTable

ORDER BY CASE

    WHEN COALESCE(Date1, '1753-01-01') < COALESCE(Date2, '1753-01-01') THEN Date1

    ELSE Date2 END
```

Results

ID	DATE 1	DATE 2
1	**2017-01-01**	2017-01-31
3	2017-01-31	**2017-01-02**
2	2017-01-31	**2017-01-03**
6	**2017-01-04**	2017-01-31
5	2017-01-31	**2017-01-05**
6	**2017-01-06**	2017-01-31

Explanation

As you see row with Id = 1is first, that because Date1have lowest record from entire table 2017-01-01, row where Id = 3is second that because Date2equals to 2017-01-02that is second lowest value from table and so on.

So we have sorted records from 2017-01-01to 2017-01-06ascending and no care on which one column Date1 or Date2are those values.

Chapter 3: Filter results using WHERE and HAVING

Section 3.1: Use BETWEEN to Filter Results

The following examples use the Item Sales and Customers sample databases.

Note: *The BETWEEN operator is inclusive.*

Using the BETWEEN operator with Numbers:

```
SELECT * From ItemSales

WHERE Quantity BETWEEN 10 AND 17
```

This query will return all ItemSalesrecords that have a quantity that is greater or equal to 10 and less than or equal to 17. The results will look like:

ID	SALE DATE	ITEM ID	QUANTITY PRICE
1	2013-07-01 100	10	34.5
4	2013-07-23 100	15	34.5
5	2013-07-24 145	10	34.5

Using the BETWEEN operator with Date Values:

```
SELECT * From ItemSales

WHERE SaleDate BETWEEN '2013-07-11' AND '2013-05-24'
```

This query will return all ItemSalesrecords with a SaleDatethat is greater than or equal to July 11, 2013 and less than or equal to May 24, 2013.

ID	SALE DATE	ITEM ID	QUANTITY PRICE
3	2013-07-11 100	20	34.5
4	2013-07-23 100	15	34.5
5	2013-07-24 145	10	34.5

NOTE : *When comparing datetime values instead of dates, you may need to convert the datetime values into a date values, or add or subtract 24 hours to get the correct results.*

Using the BETWEEN operator with Text Values:

Live example: SQL fiddle

This query will return all customers whose name alphabetically falls between the letters 'D' and 'L'. In this case, Customer #1 and #3 will be returned. Customer #2, whose name begins with a 'M' will not be included.

ID	F Name	L Name
1	William	Jones
3	Richard	Davis

Section 3.2: Use HAVING with Aggregate Functions

Unlike the WHEREclause, HAVINGcan be used with aggregate functions.

Note : *An aggregate function is a function where the values of multiple rows are grouped together as input on certain criteria to form a single value of more significant meaning or measurement (Wikipedia).*

Common aggregate functions include COUNT(), SUM(), MIN(), and MAX(). This example uses the Car Table from the Example Databases.

```
SELECT CustomerId, COUNT(Id) AS [Number of Cars]
FROM Cars

GROUP BY CustomerId
HAVING COUNT(Id) > 1
```

This query will return the CustomerIdand Number of Carscount of any customer who has more than one car. In this case, the only customer who has more than one car is Customer #1.

The results will look like:

Customer ID	Number of Cars
1	2

Section 3.3: WHERE clause with NULL/NOT NULL values

```
SELECT *

FROM Employees

WHERE ManagerId IS NULL
```

This statement will return all Employee records where the value of the ManagerIdcolumn is NULL. The result will be:

Id	FName	LName	Phone Number	Manager Id	Department Id
1	James	Smith	1234567890	NULL	1

```
SELECT *

FROM Employees

WHERE ManagerId IS NOT NULL
```

This statement will return all Employee records where the value of the ManagerIdis *not* NULL. The result will be:

Id	FName	LName	Phone Number	Manager Id	Department Id
2	John	Johnson	2468101214	1	1

3	Michael	Williams	1357911131	1	2
4	Johnathon	Smith	1212121212	2	1

***Note:** The same query will not return results if you change the WHERE clause to WHERE ManagerId = NULLor WHERE ManagerId <> NULL.*

Section 3.4: Equality

This statement will return all the rows from the table Employees.

ID	F Name	L Name	Phone Number	Manager ID	Department ID	Salary	Hire Date	Created Date	Modified Date
1	James	Smith	1234567890	Null	1	1000	01-01-2002	01-01-2002	01-01-2002
2	John	Johnson	2468101214	1	1	400	23-03-2005	23-03-2005	01-01-2002
3	Michael	Williams	1357911131	1	2	600	12-05-2009	12-05-2009	Null
4	Johnathon	Smith	1212121212	2	1	500	24-07-2016	24-07-2016	01-01-2002

Using a WHEREat the end of your SELECTstatement allows you to limit the returned rows to a condition. In this case, where there is an exact match using the =sign:

```
SELECT * FROM Employees WHERE DepartmentId = 1
```

Will only return the rows where the DepartmentIdis equal to 1:

ID	F Name	L Name	Phone Number	Manager ID	Department ID	Salary	Hire Date	Created Date	Modified Date
1	James	Smith	1234567890	Null	1	1000	01-01-2002	01-01-2002	01-01-2002
3	John	Johnson	2468101214	1	1	400	23-03-2005	23-03-2005	01-01-2002
4	Johnathon	Smith	1212121212	2	1	500	24-07-2016	24-07-2016	01-01-2002

Section 3.5: The WHERE clause only returns rows that match its criteria

Steam has a games under $10 section of their store page. Somewhere deep in the heart of their systems, there's probably a query that looks something like:

```
SELECT *

FROM Items

WHERE Price < 10
```

Section 3.6: AND and OR

You can also combine several operators together to create more complex WHERE conditions. The following examples use the Employees table:

ID	F Name	L Name	Phone Number	Manager ID	Department ID	Salary	Hire Date	Created Date	Modified Date
1	James	Smith	1234567890	Null	1	1000	01-01-2002	01-01-2002	01-01-2002
2	John	Johnson	2468101214	1	1	400	23-03-2005	23-03-2005	01-01-2002
3	Michael	Williams	1357911131	1	2	600	12-05-2009	12-05-2009	Null
4	Johnathon	Smith	1212121212	2	1	500	24-07-2016	24-07-2016	01-01-2002

AND

```
SELECT * FROM Employees WHERE DepartmentId = 1 AND ManagerId = 1
```

Will return:

ID	F Name	L Name	Phone Number	Manager ID	Department ID	Salary	Hire Date	Created Date	Modified Date
2	John	Johnson	2468101214	1	1	400	23-03-2005	23-03-2005	01-01-2002

OR

```
SELECT * FROM Employees WHERE DepartmentId = 2 OR ManagerId = 2
```

Will return:

ID	F Name	L Name	Phone Number	Manager ID	Department ID	Salary	Hire Date	Created Date	Modified Date
3	Michael	Williams	1357911131	1	2	600	12-05-2009	12-05-2009	Null
4	Johnathon	Smith	1212121212	2	1	500	24-07-2016	24-07-2016	01-01-2002

Section 3.7: Use IN to return rows with a value contained in a list

This example uses the Car Table from the Example Databases.

```
SELECT *

FROM Cars

WHERE TotalCost IN (100, 200, 300)
```

This query will return Car #2 which costs 200 and Car #3 which costs 100. Note that this is equivalent to using multiple clauses with OR, e.g.:

```
SELECT *

FROM Cars

WHERE TotalCost = 100 OR TotalCost = 200 OR TotalCost = 300
```

Section 3.8: Use LIKE to find matching strings and substrings

See full documentation on LIKE operator.

This example uses the Employees Table from the Example Databases.

```
SELECT *

FROM Employees

WHERE FName LIKE 'John'
```

This query will only return Employee #1 whose first name matches 'John' exactly.

```
SELECT *

FROM Employees

WHERE FName like 'John%'
```

Adding %allows you to search for a substring:

John% - will return any Employee whose name begins with 'John', followed by any amount of characters

%John - will return any Employee whose name ends with 'John', proceeded by any amount of characters

%John% - will return any Employee whose name contains 'John' anywhere within the value

In this case, the query will return Employee #2 whose name is 'John' as well as Employee #4 whose name is 'Johnathon'.

Section 3.9: Use HAVING to check for multiple conditions in a group

Orders Table

Customer ID	Product ID	Quantity	Price
1	2	5	100
1	3	2	200
1	4	1	500
2	1	4	50
3	5	6	700

To check for customers who have ordered both - ProductID 2 and 3, HAVING can be used

```
select customerId from orders

where productID in (2,3) group by customerId

having count(distinct productID) = 2
```

Return value:

Customer Id

1

The query selects only records with the productIDs in questions and with the HAVING clause checks for groups having 2 productIds and not just one.

Another possibility would be

```
select customerId

from orders

group by customerId

having sum(case when productID = 2 then 1 else 0 end) >0

and sum(case when productID = 3 then 1 else 0 end) >0
```

This query selects only groups having at least one record with productID 2 and at least one with productID 3.

Chapter 4: UNION / UNION ALL

UNION keyword in SQL is used to combine to ***SELECT*** statement results with out any duplicate. In order to use UNION and combine results both SELECT statement should have same number of column with same data type in same order, but the length of column can be different.

Section 4.1: Basic UNION ALL query

```
CREATE TABLE HR_EMPLOYEES (

    PersonID int, LastName VARCHAR(30),

    FirstName VARCHAR(30), Position VARCHAR(30)

);

CREATE TABLE FINANCE_EMPLOYEES (

    PersonID INT, LastName VARCHAR(30),

    FirstName VARCHAR(30), Position VARCHAR(30)

);
```

Let's say we want to extract the names of all the managersfrom our departments.

Using a UNIONwe can get all the employees from both HR and Finance departments, which hold the position of a Manager

```
SELECT
    FirstName, LastName
FROM
    HR_EMPLOYEES WHERE
    Position = 'manager' UNION ALL
SELECT
    FirstName, LastName
FROM
    FINANCE_EMPLOYEES WHERE
    Position = 'manager'
```

The UNIONstatement removes duplicate rows from the query results. Since it is possible to have people having the same Name and position in both departments we are using UNION ALL, in order not to remove duplicates.

If you want to use an alias for each output column, you can just put them in the first select statement, as follows:

```
SELECT
    FirstName as 'First Name', LastName as 'Last Name'
FROM
    HR_EMPLOYEES WHERE
    Position = 'manager' UNION ALL
SELECT
    FirstName, LastName
FROM
    FINANCE_EMPLOYEES WHERE
    Position = 'manager'
```

Section 4.2: Simple explanation and Example

In simple terms:

- UNIONjoins 2 result sets while removing duplicates from the result set
- UNIONALLjoins 2 result sets without attempting to remove duplicates

***Note** : One mistake many people make is to use a UNION when they do not need to have the duplicates removed. The additional performance cost against large results sets can be very significant.*

When you might need UNION

Suppose you need to filter a table against 2 different attributes, and you have created separate non-clustered indexes for each column.

A UNIONenables you to leverage both indexes while still preventing duplicates.

```
SELECT C1, C2, C3 FROM Table1 WHERE C1 = @Param1 UNION

SELECT C1, C2, C3 FROM Table1 WHERE C2 = @Param2
```

This simplifies your performance tuning since only simple indexes are needed to perform these queries optimally.

You may even be able to get by with quite a bit fewer non-clustered indexes improving overall write performance against the source table as well.

When you might need UNION ALL

Suppose you still need to filter a table against 2 attributes, but you do not need to filter duplicate records (either because it doesn't matter or your data wouldn't produce any duplicates during the union due to your data model design).

```
SELECT C1 FROM Table1 UNION ALL

SELECT C1 FROM Table2
```

This is especially useful when creating Views that join data that is designed to be physically partitioned across multiple tables (maybe for performance reasons, but still wants to roll-up records).

Since the data is already split, having the database engine remove duplicates adds no value and just adds additional processing time to the queries.

Chapter 5: SQL Group By vs Distinct

Section 5.1: Difference between GROUP BY and DISTINCT

GROUP BYis used in combination with aggregation functions. Consider the following table:

Order ID	User ID	Store Name	Order Value	Order Date
1	43	Store A	25	20-03-2016
2	57	Store B	50	22-03-2016
3	43	Store A	30	25-03-2016
4	82	Store C	10	26-03-2016
5	21	Store A	45	29-03-2016

The query below uses GROUP BYto perform aggregated calculations.

```
SELECT

storeName,

COUNT(*) AS total_nr_orders,

COUNT(DISTINCT userId) AS nr_unique_customers, AVG(orderValue) AS average_order_value, MIN
(orderDate) AS first_order,

    MAX(orderDate) AS lastOrder FROM

    orders GROUP BY

storeName;
```

and will return the following information

Store Name	Total Order	Unique Customers	Average Order Value	First Order - Last Order
Store A	3	2	33.3	20-03-2016 29-03-2016
Store B	1	1	50	22-03-2016 22-03-2016
Store C	1	1	10	26-03-2016 26-03-2016

While DISTINCTis used to list a unique combination of distinct values for the specified columns.

```
SELECT DISTINCT

storeName, userId

FROM

orders;
```

Chapter 6: Order of Execution

Section 6.1: Logical Order of Query Processing in SQL

```
/*(8)*/          SELECT /*9*/ DISTINCT /*11*/ TOP

/*(1)*/          FROM

/*(3)*/          JOIN

/*(2)*/          ON

/*(4)*/          WHERE

/*(5)*/          GROUP BY

/*(6)*/          WITH {CUBE | ROLLUP}

/*(7)*/          HAVING

/*(10)*/ ORDER BY

/*(11)*/ LIMIT
```

The order in which a query is processed and description of each section.

VT stands for 'Virtual Table' and shows how various data is produced as the query is processed

1. FROM: A Cartesian product (cross join) is performed between the first two tables in the FROM clause, and as a result, virtual table VT1 is generated.
2. ON: The ON filter is applied to VT1. Only rows for which the is TRUE are inserted to VT2.

3. OUTER (join): If an OUTER JOIN is specified (as opposed to a CROSS JOIN or an INNER JOIN), rows from the preserved table or tables for which a match was not found are added to the rows from VT2 as outer rows, generating VT3. If more than two tables appear in the FROM clause, steps 1 through 3 are applied repeatedly between the result of the last join and the next table in the FROM clause until all tables are processed.
4. WHERE: The WHERE filter is applied to VT3. Only rows for which the is TRUE are inserted to VT4.
5. GROUP BY: The rows from VT4 are arranged in groups based on the column list specified in the GROUP BY clause. VT5 is generated.
6. CUBE | ROLLUP: Supergroups (groups of groups) are added to the rows from VT5, generating VT6.
7. HAVING: The HAVING filter is applied to VT6. Only groups for which the is TRUE are inserted to VT7.
8. SELECT: The SELECT list is processed, generating VT8.
9. DISTINCT: Duplicate rows are removed from VT8. VT9 is generated.
10. ORDER BY: The rows from VT9 are sorted according to the column list specified in the ORDER BY clause. A cursor is generated (VC10).
11. TOP: The specified number or percentage of rows is selected from the beginning of VC10. Table VT11 is generated and returned to the caller. LIMIT has the same functionality as TOP in some SQL dialects such as Postgres and Netezza.

Chapter 7: Clean Code in SQL

How to write good, readable SQL queries, and example of good practices.

Section 7.1: Formatting and Spelling of Keywords and Names

Table/Column Names

Two common ways of formatting table/column names are CamelCaseand snake_case:

```
SELECT FirstName, LastName FROM Employees

WHERE Salary > 500;
```

```
SELECT first_name, last_name FROM employees

WHERE salary > 500;
```

Names should describe what is stored in their object. This implies that column names usually should be singular. Whether table names should use singular or plural is a heavily discussed question, but in practice, it is more common to use plural table names.

Adding prefixes or suffixes like tblor colreduces readability, so avoid them. However, they are sometimes used to avoid conflicts with SQL keywords, and often used with triggers and indexes (whose names are usually not mentioned in queries).

Keywords

SQL keywords are not case sensitive. However, it is common practice to write them in upper case.

Section 7.2: Indenting

There is no widely accepted standard. What everyone agrees on is that squeezing everything into a single line is bad:

```
SELECT d.Name, COUNT(*) AS Employees FROM Departments AS d JOIN Employees AS e ON d.ID = e.DepartmentID WHERE d.Name != 'HR' HAVING COUNT(*) > 10 ORDER BY COUNT(*) DESC;
```

At the minimum, put every clause into a new line, and split lines if they would become too long otherwise:

```
SELECT d.Name,

          COUNT(*) AS Employees FROM Departments AS d

JOIN Employees AS e ON d.ID = e.DepartmentID WHERE d.Name != 'HR'

HAVING COUNT(*) > 10 ORDER BY COUNT(*) DESC;
```

Sometimes, everything after the SQL keyword introducing a clause is indented to the same column:

```
SELECT        d.Name,

             COUNT(*) AS Employees FROM    Departments AS d

JOIN    Employees AS e ON d.ID = e.DepartmentID WHERE  d.Name != 'HR'

HAVING        COUNT(*) > 10

ORDER BY COUNT(*) DESC;
```

(This can also be done while aligning the SQL keywords right.)

Another common style is to put important keywords on their own lines:

```
SELECT
    d.Name,
    COUNT(*) AS Employees FROM
    Departments AS d
JOIN
    Employees AS e
    ON d.ID = e.DepartmentID WHERE
    d.Name != 'HR' HAVING
    COUNT(*) > 10 ORDER BY
    COUNT(*) DESC;
```

Vertically aligning multiple similar expressions improves readability:

```
SELECT Model,
       EmployeeID FROM Cars
WHERE CustomerID = 42
  AND Status = 'READY';
```

Using multiple lines makes it harder to embed SQL commands into other programming languages. However, many languages have a mechanism for multi-line strings, e.g., @"..."in C#, """..."""in Python, or R"(...)"in C++.

Section 7.3: SELECT *

SELECT *returns all columns in the same order as they are defined in the table.

When using SELECT *, the data returned by a query can change whenever the table definition changes. This increases the risk that different versions of your application or your database are incompatible with each other.

Furthermore, reading more columns than necessary can increase the amount of disk and network I/O. So you should always explicitly specify the column(s) you actually want to retrieve:

```
--SELECT *                                   don't

SELECT ID, FName, LName, PhoneNumber   -- do

FROM Emplopees;
```

(When doing interactive queries, these considerations do not apply.)

However, SELECT *does not hurt in the subquery of an EXISTS operator, because EXISTS ignores the actual data anyway (it checks only if at least one row has been found). For the same reason, it is not meaningful to list any specific column(s) for EXISTS, so SELECT *actually makes more sense:

```
 -- list departments where nobody was hired recently
 SELECT ID,

          Name

 FROM Departments

 WHERE NOT EXISTS (SELECT *

                   FROM Employees

                   WHERE DepartmentID = Departments.ID AND HireDate >= '2015-01-01');
```

Section 7.4: Joins

Explicit joins should always be used; implicit joins have several problems:

The join condition is somewhere in the WHERE clause, mixed up with any other filter conditions. This makes it harder to see which tables are joined, and how.

Due to the above, there is a higher risk of mistakes, and it is more likely that they are found later.
In standard SQL, explicit joins are the only way to use outer joins:

```
SELECT d.Name,

         e.Fname  ||  e.LName  AS  EmpName
FROM  Departments AS d

LEFT JOIN Employees  AS e ON d.ID = e.DepartmentID;
```

Explicit joins allow using the USING clause:

```
SELECT RecipeID,
 Recipes.Name,
        COUNT(*)   AS    NumberOfIngredients
FROM  Recipes
LEFT JOIN Ingredients USING (RecipeID);
```

(This requires that both tables use the same column name.

USING automatically removes the duplicate column from the result, e.g., the join in this query returns a single RecipeIDcolumn.)