PROCESSING SINGLE TABLES

Four data manipulation language commands are used in SQL. We have talked briefly about three of them (UPDATE, INSERT, and DELETE) and have seen several examples of the fourth, **SELECT**.

- Although the UPDATE, INSERT, and DELETE com mands allow modification of the data in the tables, it is the SELECT command, with its various clauses, that allows users to query the data contained in the tables and ask many different questions or create ad hoc queries.

Be careful when writing SELECT queries, here are some ways to ensure that your queries are doing what they are intended to do…

- Before running queries against a large production database, always test them carefully on a small test set of data to be sure that they are returning the correct results.
- In addition to checking the query results manually, it is often possible to parse queries into smaller parts, examine the results of these simpler queries, and then recombine them.
    - This will ensure that they act together in the expected way.

Clauses of the SELECT Statement

Most SQL data retrieval statements include the following three clauses:

SELECT	Lists the columns (including expressions involving columns) from base tables, derived tables, or views to be projected into the table that will be the result of the command. (That's the technical way of saying it lists the data you want to display.)
FROM	Identifies the tables, derived tables, or views from which columns can be chosen to appear in the result table and includes the tables, derived tables, or views needed to join tables to process the query.
WHERE	Includes the conditions for row selection within the items in the FROM clause and the conditions between tables, derived tables, or views for joining. Because SQL is considered a set manipulation language, the WHERE clause is important in defining the set of rows being manipulated.

- The first two clauses are required, and the third is necessary when only certain table rows are to be retrieved or multiple tables are to be joined.
- Example:

Query: Which products have a standard price of less than $275?

```
SELECT ProductDescription, ProductStandardPrice
  FROM Product_T
      WHERE ProductStandardPrice < 275;
```

Result:

PRODUCTDESCRIPTION	PRODUCTSTANDARDPRICE
End Table	175
Computer Desk	250
Coffee Table	200

- Every SELECT statement returns a result table (a set of rows) when it executes.
    - So, SQL is consistent—tables in, tables out of every query.
    - This becomes important with more complex queries because we can use the result of one query (a table) as part of another query (e.g., we can

include a SELECT statement as one of the elements in the FROM clause, creating a derived table, which we illustrate later in this chapter).

- Two special key words can be used along with the list of columns to display: DISTINCT and *.
  - If the user does not wish to see duplicate rows in the result, SELECT DISTINCT may be used.
    - In the preceding example, if the other computer desk carried by Pine Valley Furniture also had a cost of $250, the results of the query would have had duplicate rows. SELECT DISTINCT ProductDescription would display a result table without the duplicate rows.
  - SELECT *, where * is used as a wildcard to indicate all columns, displays all columns from all the items in the FROM clause.
- If there is any ambiguity in an SQL command, you must indicate exactly from which table, derived table, or view the requested data are to come.
  - For example, in Figure 5-3 CustomerID is a column in both Customer_T and Order_T. When you own the database being used (i.e., the user created the tables) and you want CustomerID to come from Customer_T, specify it by asking for Customer_T.CustomerID. If you want CustomerID to come from Order_T, then ask for Order_T.CustomerID.
  - When you are allowed to use data created by someone else, you must also specify the owner of the table by adding the owner's user ID. Now a request to SELECT the CustomerID from Customer_T may look like this: .Customer_T.CustomerID.
- establish aliases for data names that will then be used for the rest of the query
  - they are widely implemented and aid in readability and simplicity in query construction.
  - Example:

Query: What is the address of the customer named Home Furnishings? Use an alias, Name, for the customer name. (The AS clauses are bolded for emphasis only.)

```
SELECT CUST.CustomerName AS Name, CUST.CustomerAddress
  FROM Customer_T AS Cust
    WHERE Name = 'Home Furnishings';
```

Result:

NAME	CUSTOMERADDRESS
Home Furnishings	1900 Allard Ave.

    - Notice that we alias CUST.CustomerName as NAME and then use it below
    - Notice that the column header prints as Name rather than CustomerName and that the table alias may be used in the SELECT clause even though it is not defined until the FROM clause.
      - Using an alias is a good way to make column headings more readable.
- Many RDBMSs have other proprietary SQL clauses to improve the display of data.

- For example, Oracle has the COLUMN clause of the SELECT statement, which can be used to change the text for the column heading, change alignment of the column heading, reformat the column value, or control wrapping of data in a column, among other properties.

- When you use the SELECT clause to pick out the columns for a result table, the columns can be rearranged so that they will be ordered differently in the result table than in the original table.
  - In fact, they will be displayed in the same order as they are included in the SELECT statement.
  - Example

Query: List the unit price, product name, and product ID for all products in the Product table.

```
SELECT ProductStandardPrice, ProductDescription, ProductID
FROM Product_T;
```

Result:

PRODUCTSTANDARDPRICE	PRODUCTDESCRIPTION	PRODUCTID
175	End Table	1
200	Coffee Table	2
375	Computer Desk	3
650	Entertainment Center	4
325	Writer's Desk	5
750	8-Drawer Desk	6
800	Dining Table	7
250	Computer Desk	8

Using Expressions

Basically you can do mathematical expressions on applicable attributes in a table

The basic SELECT . . . FROM . . . WHERE clauses can be used with a single table in a number of ways. You can create expressions, which are mathematical manipulations of the data in the table, or take advantage of stored functions, such as SUM or AVG, to manipulate the chosen rows of data from the table. Mathematical manipulations can be constructed by using the + for addition, – for subtraction, * for multiplication, and / for division. These operators can be used with any numeric columns. Expressions are computed for each row of the result table, such as displaying the difference between the standard price and unit cost of a product, or they can involve computations of columns and functions, such as standard price of a product multiplied by the amount of that product sold on a particular order (which would require summing OrderedQuantities). Some systems also have an operand called modulo, usually indicated by %. A modulo is the integer remainder that results from dividing two integers. For example, 14 % 4 is 2 because 14/4 is 3, with a remainder of 2. The SQL standard supports year-month and

day-time intervals, which makes it possible to perform date and time arithmetic (e.g., to calculate someone's age from today's date and a person's birthday).

Perhaps you would like to know the current standard price of each product and its future price if all prices were increased by 10 percent. Here are the query and the results.

Query: What are the standard price and standard price if increased by 10 percent for every product?

```
SELECT ProductID, ProductStandardPrice, ProductStandardPrice*1.1 AS
Plus10Percent
  FROM Product_T;
```

Result:

PRODUCTID	PRODUCTSTANDARDPRICE	PLUS10PERCENT
2	200.0000	220.00000
3	375.0000	412.50000
1	175.0000	192.50000
8	250.0000	275.00000
7	800.0000	880.00000
5	325.0000	357.50000
4	650.0000	715.00000
6	750.0000	825.00000

The precedence rules for the order in which complex expressions are evaluated are the same as those used in other programming languages and in algebra. Expressions in parentheses will be calculated first. When parentheses do not establish order, multiplication and division will be completed first, from left to right, followed by addition and subtraction, also left to right. To avoid confusion, use parentheses to establish order. Where parentheses are nested, the innermost calculations will be completed first.

Using Functions

Standard SQL has a wide variety of mathematical, string and date manipulation, and other functions.

Mathematical	MIN, MAX, COUNT, SUM, ROUND (to round up a number to a specific number of decimal places), TRUNC (to truncate insignificant digits), and MOD (for modular arithmetic)
String	LOWER (to change to all lowercase), UPPER (to change to all capital letters), INITCAP (to change to only an initial capital letter), CONCAT (to concatenate), SUBSTR (to isolate certain character positions), and COALESCE (finding the first not NULL values in a list of columns)
Date	NEXT_DAY (to compute the next date in sequence), ADD_MONTHS (to compute a date a given number of months before or after a given date), and MONTHS_BETWEEN (to compute the number of months between specified dates)
Analytical	TOP (find the top n values in a set, e.g., the top 5 customers by total annual sales)

- Example: Perhaps you want to know the average standard price of all inventory items. To get the overall average value, use the AVG stored function. We can name the resulting expression with an alias, AveragePrice.

  Query: What is the average standard price for all products in inventory?

  ```
  SELECT AVG (ProductStandardPrice) AS AveragePrice
  FROM Product_T;
  ```

  Result:

  AVERAGEPRICE
  440.625

- Stored functions include, ANY, AVG, COUNT, EVERY, GROUPING, MAX, MIN, SOME, and SUM. SQL:2008 added LN, EXP, POWER, SQRT, FLOOR, CEILING,

and WIDTH_BUCKET.

- COUNT, MIN, MAX, SUM, and AVG of specified columns in the column list of a SELECT command may be used to specify that the resulting answer table is to contain aggregated data instead of row-level data.
  - This means we are only getting one thing returned rather than eevery row returned since these functions use up all the rows to return 1 thing back
  - Example:

Query: How many different items were ordered on order number 1004?

```
SELECT COUNT (*)
  FROM OrderLine_T
    WHERE OrderID = 1004;
```

Result:

COUNT (*)
2

    - The difference between COUNT(*) and COUNT is that COUNT(*) counts all rows selected by a query, regardless of if the row has a null value. Whereas COUNT tallies only rows that contain values.
  - Example of an error:

Query: How many different items were ordered on order number 1004, and what are they?

```
SELECT ProductID, COUNT (*)
  FROM OrderLine_T
    WHERE OrderID = 1004;
```

In Oracle, here is the result.

Result:

```
ERROR at line 1:
ORA-00937: not a single-group group function
```

    - We get an error here because we are trying to print out both the productID and a count(*) but that isnt possible
      - To do this we would have to build two queries.
      - In most implementations, SQL cannot return both a row value and a set value (creating one value out of many); users must run two separate queries, one that returns row information and one that returns set information.
  - Example of another error: A similar issue arises if we try to find the difference between the standard price of each product and the overall average standard price (which we calculated above). You might think the query would be:

```
SELECT ProductStandardPrice – AVG(ProductStandardPrice)
  FROM Product_T;
```

    - However, again we have mixed a column value with an aggregate (AVG(ProductStandardPrice)), which will cause an error.
    - A correct way would be to..

Query: Display for each product the difference between its standard price and the overall average standard price of all products.

```
SELECT ProductStandardPrice – PriceAvg AS Difference
  FROM Product_T, (SELECT AVG(ProductStandardPrice) AS PriceAvg
    FROM Product_T);
```

Result:

```
DIFFERENCE
   –240.63
    –65.63
   –265.63
   –190.63
    359.38
   –115.63
    209.38
    309.38
```

- A query inside of a query, notice also that the inside determines what we can use on the outside

- SUM and AVG can only be used with numeric columns. COUNT, COUNT (*), MIN, and MAX can be used with any data type. Using MIN on a text column, for example, will find the lowest value in the column, the one whose first column is closest to the beginning of the alphabet. SQL implementations interpret the order of the alphabet differently. For example, some systems may start with A–Z, then a–z, and then 0–9 and special characters. Others treat upper- and lowercase letters as being equivalent. Still others start with some special characters, then proceed to numbers, letters, and other special characters. Here is the query to ask for the first ProductName in Product_T alphabetically, which was done using the AMERICAN character set in Oracle 12c.
  - More about other functions
  - Example:

Query: Alphabetically, what is the first product name in the Product table?

```
SELECT MIN (ProductDescription)
  FROM Product_T;
```

Result:

```
MIN(PRODUCTDESCRIPTION)
8-Drawer Desk
```

- The result demonstrates that numbers are sorted before letters in this character set.

Using Wildcards

The use of the asterisk (*) as a wildcard in a SELECT statement has been previously shown. Wildcards may also be used in the WHERE clause when an exact match is not possible. Here, the key word LIKE is paired with wildcard characters and usually a string containing the characters that are known to be desired matches. The wildcard character % is used to represent any collection of characters. Thus, using LIKE '%Desk' when searching ProductDescription will find all different types of desks carried by Pine Valley Furniture Company. The underscore (_) is used as a wildcard character to represent exactly one character rather than any collection of characters. Thus, using LIKE '_-drawer' when searching ProductName will find any products with specified drawers, such as 3-, 5-, or 8-drawer dressers.

Using Comparison Operators

- These are all the comparison operators in sql:

TABLE 5-3 Comparison Operators in SQL

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
!=	Not equal to

- Example:

Query: Which orders have been placed since 10/24/2018?

```
SELECT OrderID, OrderDate
  FROM Order_T
    WHERE OrderDate > '24-OCT-2018';
```

Result:

ORDERID	ORDERDATE
1007	27-OCT-18
1008	30-OCT-18
1009	05-NOV-18
1010	05-NOV-18

  - We are doing this on a Date type

- Example:

```
SELECT ProductDescription, ProductFinish
  FROM Product_T
    WHERE ProductFinish != 'Cherry';
```

Result:

PRODUCTDESCRIPTION	PRODUCTFINISH
Coffee Table	Natural Ash
Computer Desk	Natural Ash
Entertainment Center	Natural Maple
8-Drawer Desk	White Ash
Dining Table	Natural Ash
Computer Desk	Walnut

Using Null Values

Columns that are defined without the NOT NULL clause may be empty, and this may be a significant fact for an organization. You will recall that a null value means that a column is missing a value; the value is not zero or blank or any special code—there simply is no value. We have already seen that functions may produce different results when null values are present than when a column has a value of zero in all qualified rows. It is not uncommon, then, to first explore whether there are null values before deciding how to write other commands, or it may be that you simply want to see data about table rows where there are missing values. For example, before undertaking a postal mail advertising campaign, you might want to pose the following query.

Query: Display all customers for whom we do not know their postal code.

```
SELECT * FROM Customer_T WHERE CustomerPostalCode IS NULL;
```

Result:

Fortunately, this query returns 0 rows in the result in our sample database, so we can mail advertisements to all our customers because we know their postal codes. The term IS NOT NULL returns results for rows where the qualified column has a non-null value. This allows us to deal with rows that have values in a critical column, ignoring other rows.

Using Boolean Operators

Some complex questions can be answered by adjusting the WHERE clause further.
The Boolean or logical operators AND, OR, and NOT can be used to good purpose:

AND	Joins two or more conditions and returns results only when all conditions are true.
OR	Joins two or more conditions and returns results when any conditions are true.
NOT	Negates an expression.

- If multiple Boolean operators are used in an SQL statement, NOT is evaluated first, then AND, then OR.
- Example:

Query A: List product name, finish, and standard price for all desks and all tables that cost more than $300 in the Product table.

```
SELECT ProductDescription, ProductFinish, ProductStandardPrice
  FROM Product_T
    WHERE ProductDescription LIKE '%Desk'
      OR ProductDescription LIKE '%Table'
      AND ProductStandardPrice > 300;
```

Result:

PRODUCTDESCRIPTION	PRODUCTFINISH	PRODUCTSTANDARDPRICE
Computer Desk	Natural Ash	375
Writer's Desk	Cherry	325
8-Drawer Desk	White Ash	750
Dining Table	Natural Ash	800
Computer Desk	Walnut	250

  - Notice that only the tables are being filtered by the ProductStandardPrice > 300.
  - With this query (illustrated in Figure 5-8), the AND will be processed first, returning all tables with a standard price greater than $300. Then the part of the query before the OR is processed, returning all desks, regardless of cost. Finally the results of the two parts of the query are combined (OR), with the final result of all desks along with all tables with standard price greater than $300.
  - If we had wanted to return only desks and tables costing more than $300, we should have put parentheses after the WHERE and before the AND..

Query B: List product name, finish, and standard price for all desks and tables in the PRODUCT table that cost more than $300.

```
SELECT ProductDescription, ProductFinish, ProductStandardPrice
  FROM Product_T;
  WHERE (ProductDescription LIKE '%Desk'
    OR ProductDescription LIKE '%Table')
    AND ProductStandardPrice > 300;
```

The results follow. Only products with unit price greater than $300 are included.

Result:

PRODUCTDESCRIPTION	PRODUCTFINISH	PRODUCTSTANDARDPRICE
Computer Desk	Natural Ash	375
Writer's Desk	Cherry	325
8-Drawer Desk	White Ash	750
Dining Table	Natural Ash	800

- The parenthesis helps sql understand that the things being OR'd are together and then we AND those things by ProductStandardPrice > 300

- This example illustrates why SQL is considered a set-oriented, not a record-oriented, language. (C, Java, and Cobol are examples of record-oriented languages because they must process one record, or row, of a table at a time.)

Using Ranges for Qualification

The comparison operators < and > are used to establish a range of values. The key words BETWEEN and NOT BETWEEN can also be used. For example, to find products with a standard price between $200 and $300, the following query could be used.

Query: Which products in the Product table have a standard price between $200 and $300?

```
SELECT ProductDescription, ProductStandardPrice
  FROM Product_T
    WHERE ProductStandardPrice > = 200 AND ProductStandardPrice < = 300;
```

Result:

PRODUCTDESCRIPTION	PRODUCTSTANDARDPRICE
Coffee Table	200
Computer Desk	250

The same result will be returned by the following query.

Query: Which products in the PRODUCT table have a standard price between $200 and $300?

```
SELECT ProductDescription, ProductStandardPrice
  FROM Product_T
    WHERE ProductStandardPrice BETWEEN 200 AND 300;
```

Result: Same as previous query.

Adding NOT before BETWEEN in this query will return all the other products in Product_T because their prices are less than $200 or more than $300.

Using Distinct Values

Sometimes when returning rows that don't include the primary key(rows that are not the primary key since every primary key is different thus no duplicates ever), duplicate rows will be returned.

- Example of a query that has many duplicates:

Query: What order numbers are included in the OrderLine table?

```
SELECT OrderID
  FROM OrderLine_T;
```

ORDERID
1002
1003
1004
1004
1005
1006
1006
1006
1007
1007
1008
1008
1009
1009
1010

18 rows selected.

- Do we really need the redundant OrderIDs in this result? If we add the key word DISTINCT, then only 1 occurrence of each OrderID will be returned, 1 for each of the 10 orders represented in the table.

Query: What are the distinct order numbers included in the OrderLine table?

```
SELECT DISTINCT OrderID
  FROM OrderLine_T;
```

Result:

ORDERID
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010

10 rows selected.

- Here we note that the keyword DISTINCT makes it so that every record with that ORDERID is only shown once

- DISTINCT and its counterpart, ALL, can be used only once in a SELECT statement. It comes after SELECT and before any columns or expressions are listed.
  - The DISTINCT keyword vs ALL keyword
    - ALL keyword returns all the record including duplicates
- If a SELECT statement projects more than one column, only rows that are identical for every column will be eliminated.
  - That is say we query based on two attributes and we want only distinct records. How will the query distinguish duplicates.
  - Example: Thus, if the previous statement also includes OrderedQuantity (another attribute), 14 rows are returned (now more rows are returned) because there are now only 4 duplicate rows rather than 8 (we have less duplicates since OrderedQuantity is also in play now).
    - For example, both items ordered on OrderID 1004 were for 2 items, so the second pairing of 1004 and 2 will be eliminated.
      - The OrderID 1004 had two records with OrderedQuantity of 2

thus we only keep one of those records

- For clarity, we could also have 1 record with OrderID 1004 with OrderedQuantity of 1 which in this query would also be included even though it's the same ID the quanity is different thus its still different based on our query

```
SELECT DISTINCT OrderID, OrderedQuantity
  FROM OrderLine_T;
```

Result:

ORDERID	ORDEREDQUANTITY
1001	1
1001	2
1002	5
1003	3
1004	2
1005	4
1006	1
1006	2
1007	2
1007	3
1008	3
1009	2
1009	3
1010	10

14 rows selected.

Using IN and NOT IN with Lists

To match a list of values, consider using IN.

Query: List all customers who live in warmer states.

```
SELECT CustomerName, CustomerCity, CustomerState
  FROM Customer_T
    WHERE CustomerState IN ('FL', 'TX', 'CA', 'HI');
```

Result:

CUSTOMERNAME	CUSTOMERCITY	CUSTOMERSTATE
Contemporary Casuals	Gainesville	FL
Value Furniture	Plano	TX
Impressions	Sacramento	CA
California Classics	Santa Clara	CA
M and H Casual Furniture	Clearwater	FL
Seminole Interiors	Seminole	FL
Kaneohe Homes	Kaneohe	HI

7 rows selected.

IN is particularly useful in SQL statements that use subqueries, which will be covered in Chapter 6. The use of IN is also very consistent with the set nature of SQL. Very simply, the list (set of values) inside the parentheses after IN can be literals, as illustrated here, or can be a SELECT statement with a single result column, the result of which will be plugged in as the set of values for comparison. In fact, some SQL programmers always use IN, even when the set in parentheses after IN includes only one item. Similarly, any "table" of the FROM clause can be itself a derived table defined by including a SELECT statement in parentheses in the FROM clause (as we saw earlier, with the query about the difference between the standard price of each product and the average standard price of all products). The ability to include a SELECT statement anyplace within SQL where a set is involved is a very powerful and useful feature of SQL, and, of course, totally consistent with SQL being a set-oriented language, as illustrated in Figures 5-8 and 5-9.