DBS211

Week 3 - Introduction to SQL (Single Table DML Statements)

Table of Contents

Suggested Additional Reading Materials

- Text - Chapter 16 (SQL Data Manipulation Language)
  (Up to end of "Built-in Functions", ignoring grouping, count and having sections)
- W3Schools - SELECT
- W3Schools - SELECT DISTINCT
- W3Schools - SELECT WHERE
- W3Schools - SELECT AND, OR, NOT
- W3Schools - SELECT ORDER BY
- W3Schools - NULL Values
- W3Schools - INSERT
- W3Schools - UPDATE
- W3Schools - DELETE

Welcome to Week 3

Week 3 is where we move from from theory of design and database concepts to hand's on real programming work. With the introduction of SQL and the coding environment. This is a heavy week and takes a lot of time, so get started early.

By the end of this week, you will be able to:

- define SQL and describe how it is used to communicate with various database management systems
- list and outline the application of the 4 primary sub-languages of SQL in real-world scenarios
- create basic data query statements requesting specific information from single tables while filtering and sorting the data
- demonstrate knowledge of the term CRUD
- be able to write and execute basic DML statements to insert, update and delete data within a relational database

SQL and SQL Standardization

What is SQL? SQL stands for Structured Query Language SQL lets you access and manipulate databases.

SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987

What Can SQL do? SQL can:

- execute queries against a database
- retrieve data from a database
- insert records in a database
- update records in a database
- delete records from a database
- create new databases
- create new tables in a database
- create stored procedures in a database
- create views in a database
- set permissions on tables, procedures, and views

SQL is a Standard - BUT.... Although SQL is an ANSI/ISO standard, there are different versions of the SQL language. However, they all DBMSs support at least the major commands (such as SELECT, UPDATE, DELETE, INSERT, CREATE) in a similar manner. It shoudl also be noted that most of the DBMSs also have their own proprietary extensions in addition to the SQL standard! This course uses Oracle as it's DBMS, but will try to remain agnostic with respect to the teaching and learning of the SQL language to maximize the learner's ability to apply their learned knwoledge to a different DBMS in the future.

Sub-Languages of SQL

There are several sub-categories of SQL that play specific roles in the querying and manipulation of databases.

DML	Data	This set of SQL statements are used to query or change the data stored in the database. DML common

	Manipulation Language	statements include `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. These commands generally only impact rows of data in existing tables. These commands are about "What" data is stored in the database.
DDL	Data Definition Language	This set of SQL statements are used to create or modify the structure of the database. "How" is data stored in the database? These commands generally only impact tables and columns and other stored objects in the database. The main statements included in DDL are `CREATE`, `CREATE OR REPLACE`, and `ALTER`.
TCL	Transaction Control Language	TCL is a subset of statements that assist in controlling one or more SQL DML statements that need to either completely succeed or entirely fail. A transaction must be complete or not happen at all. Common commands include `BEGIN`, `COMMIT`, `SAVEPOINT` and `ROLLBACK`
PL/SQL	Procedural Language extensions to SQL	PL/SQL is an advanced compiled procedural language extension to SQL that allows for complex processes to be performed on data or databases. PL/SQL contains many of the operations that we would find in most programming languages, such as loops, functions, variables, conditional statements, etc. This topic is not covered until DBS501 or DBS511.

The Basics of SELECT

The `SELECT` statement is used to retrieve data from a database and is most ofter referred to as a **Query**. In this lesson, we will concentrate on retrieving data from a single table only.

You can think of a table as an entity (example: movies), and each row as a specific instance of that entity type (example: Avatar, Star Wars, Toy Story). Therefore, the columns represent the various properties of those entities (i.e. Title, Director, Year, Length).

The basic form of the SELECT statement, at this point in the course is:

```
SELECT <field list comma separated>
FROM <source name>
WHERE <one or more comparison expressions>
ORDER BY <field list comma separated>;
```

The SELECT and FROM parts of the staement are required, the WHERE and ORDER BY parts are optional.

Order of Execution

When we look at the basic SELECT statement, we might assume that it is either executed all at once, due to its' semi-colon syntax, or it is executed from top to bottom, like most sequential applications run. However, this is not the case in SQL. There is an order to the execution of the statement that must be understood, as it has consequences to how the statement can be used.

For the four basic parts of the statement in this lesson, the order of execution is:

1. FROM
2. WHERE
3. SELECT
4. ORDER BY

In English, we might say "FROM this source, WHERE something is true, SELECT these fields and calculations, and ORDER BY the output by these fields."

> We will refer back to the Order of Execution many times through this course when the lesson is impacted by the order of execution.

SELECT

Movies				
Id	**Title**	**Director**	**Year**	**Length_minutes**
1	Toy Story	John Lasseter	1995	81
2	A Bug's Life	John Lasseter	1998	95
3	Toy Story 2	John Lasseter	1999	93
4	Monsters, Inc.	Pete Docter	2001	92

Id	Title	Director	Year	Length_minutes
5	Finding Nemo	Andrew Stanton	2003	107
6	The Incredibles	Brad Bird	2004	116
7	Cars	John Lasseter	2006	117
8	Ratatouille	Brad Bird	2007	115

The basic query has 2 mandatory parts, the SELECT part and the source, FROM, part and is written as follows:

```
SELECT <column1>, <column2>, <columnN>
FROM <tablename>;
```

For example using the above table:

```
SELECT id, title, director
FROM movies;
```

A few points to note:

- SQL code is not case sensitive, but we often use case as a style guide for code readability
  - Keywords are entered in upper case
  - User defined words are entered in lower case
  - Carriage returns are used before each of the main components of an SQL statement
- column and table names must be spelled exactly
- column and table names must be contained inside double quotes (") if there is a space in the name. (this is a bad design practice)
- the columns are output in the same order as listed in the SELECT portion of the statements

- although we almost always include a semi-colon (;) at the end of each statement, it is only required if there is another statement in the same file afterwards.

The results of this query are a table of rows and columns including only those columns requested in the statement.

If we wanted to retrieve an exact copy of ALL the data stored in a table, we use the * shorthand. This retrieves all columns from the data table and the columns are presented in the same order as the way the table was created in the first place.

```
SELECT * FROM movies;
```

practice exercises

The Dual Table

There are many times when you need to output something that is not in a table. Oracle has a built in table for just this purpose called ***dual***. The FROM part of the SELECT statement is required, but you do not want to choose an existing table as it would load it into memory unnecessarily. Therefore the dual table is minimalistic in nature for this purpose.

```
SELECT * FROM dual;
```

outputs a single row, single column table with the value 'X' in it.

```
SELECT 2 * 7 AS number FROM dual
```

is an example where no table is needed, we just need to value.

> The dual table is used most when dealing with dates, especially when dates are relative to the current date: today, yesterday, tomorrow, next Monday, next week, last week, this year, etc.

Aliases (Field and Table Aliases)

Aliases are a feature of SQL that allows you to rename fields and tables inside the SQL statement for a variety of purposes. Some of these purposes include:

- Having the name of columns different than the field name in the output.
- If the output is a new calculated value, it needs an appropriate name
- when joining multiple tables, the same field name may exist in more than one table, so we need an identifier
- can be used to same significant typing and make the SQL code much cleaner and easier to read

Quotations in Oracle SQL

Quotes are utilized differently in many programming languages and even used differently in different versions of SQL. Oracle uses the most strict implementation of quotes and is stardardized to work in mosts versions of SQL.

Single Quotes: are used to defined string values. For example: Inserting a person's name, would require the name to be in single quotes `'Name'`

Double Quotes: are used to define or use an object. For example: If a column name has a space in it, which is bad practice, you would have to use double quotes around the name to use it or define it. Aliases can have spaces in them, if they are to be in a printed report, but are required to be defined using doubel quotes.

> HINT: It is bad practice to ever have a space in an object, even if it is an alias. In modern technology, creating reports directly in the database for printing is becoming obsolete. Most of the time, the database is feeding query results to software which has a user interface. This process often means columns are referred to by name in the software and spaces in the column names makes this extremely difficult in many programming languages. Spaces should only ever be used when the output is being directly viewed by the recipient of the report or it is being printed.

Field Aliases

Aliases can be applied to fields in several ways. Here are a few examples:

```
SELECT firstname AS first, lastname AS last
FROM employees;
```

FIRST	LAST

```
SELECT firstname || ' ' || lastname AS name
FROM employees;
```

NAME

```
SELECT productCode, price, quantityordered AS quantity, price * quantityordered
FROM orders;
```

PRODUCTCODE	PRICE	QUANTITY	[PRICE * QUANTITYORDERED]

Without the alias, the column name for the calculation becomes the calculation itself.

```
SELECT productCode, price, quantityordered AS quantity, price * quantityordered AS subtotal
FROM orders;
```

By using an appropriate alias, the column name is much more readable and usable

PRODUCTCODE	PRICE	QUANTITY	SUBTOTAL

The `AS` keyword is optional. It is a good idea for readability though.

By default, output column names are in ALL CAPS. If a case senstive column name is required or a column name with a space then the use of double quotes is needed.

```
SELECT firstname AS "FirstName", lastname AS LastName
FROM employees;
```

FirstName	LASTNAME

```
SELECT firstname || ' ' || lastname AS "Employee Name"
FROM employees;
```

In the above example, the name `FirstName` will appear as it is in the quotes, but `LastName` will appear as `LASTNAME`. Also note that the space between the first and last names is defined using single quotes while the aliases, which are objects, are defined using double quotes.

Table Aliases

Aliases can also be applied to data sources as well, regardless if they are tables, views, or user-defined objects.

```
SELECT firstname, lastname FROM employees e WHERE employeeid = 123;
```

Table aliases can then be used in other parts of the statement to refer to fields more directly

```
SELECT e.firstname, e.lastname FROM employees e WHERE employeeid = 123;
```

In this example, the prefix `e` is used to absolutey declare the firstname and lastname fields. This is important when more than one table may be referenced and multiple fields with the same name are available.

Calculated Values and Single-Line Functions

> NOTE: ALL calculated values should be defined with an alias. Otherwise, the name of the column will be the calculation forula itself.

In SQL it is almost always required to manipulate the data to calculate new values. This is how information can be retrieved from data.

```
SELECT productCode, price, quantityordered AS quantity, price * quantityordered AS subtotal
FROM orders;
```

The column defined as `subtotal` does not exist in the source table. It is a **calculated value.** The output may look something like this

PRODUCTCODE	PRICE	QUANTITY	SUBTOTAL
123	1.25	3	3.75
456	5.95	2	11.90

Calculated fields are not always mathematical in nature.

> Any output value that is not directly taken from a single field wil lbe a calcualted value.

```
SELECT firstname || ' ' || lastname AS "Employee Name"
FROM employees;
```

Employee Name
John Smith
Mary Jones

The one and only column in this query is a calculated field. String concatenation changes the values of the original fields, and therefore result in calculated fields.

Single-Row Functions

There are many functions available for use within SQL. Most functions are standardized across DBMSs, but there are slight differences in some functions. We will concentrate on the oracle version of the functions for this course. A fairly complete list of functions are available at https://docs.oracle.com/database/121/SQLRF/functions002.htm#SQLRF51178.

There are three main categories for functions:

Numeric Functions are functions involved in mathematical calculations. Some of the most common include `ROUND`, `SQRT`, and `MOD`

```
SELECT studentID, mark, maxmarks, round(100 * mark / maxmarks,2) AS grade
FROM studentMarks
```

STUDENTID	MARK	MAXMARKS	GRADE
1	27	53	45.28
2	33	53	62.26
3	48	53	90.57

String Functions involve manipulating strings to create custom output. Some common examples are: `CONCAT`, `UPPER`, `LENGTH` and `SUBSTR`.

```
SELECT  firstname, lastname,
        CONCAT(firstname, lastname) AS name,
        UPPER(firstname) as first,
        LENGTH(lastname) as lenlast,
        SUBSTR(lastname, 2, 3) as 2l
FROM students;
```

FIRSTNAME	LASTNAME	NAME	FIRST	LENLAST	2L

FIRSTNAME	LASTNAME	NAME	FIRST	LENLAST	2L
John	Smith	JohnSmith	JOHN	5	mit
Mary	Johnston	MaryJohnston	MARY	8	ohn

Date Functions are used to manipulate dates and perform calculations with respect to dates. The formatting of dates are amongst the most common, but also age, length of time etc are also important. Some important data functions include `TO_DATE`, `MONTHS_BETWEEN`, `ADD_MONTHS`, `NEXT_DAY`, `SYSDATE`, and `EXTRACT`.

```
SELECT sysdate FROM dual;
```

would output a single-row, single-column, table, know as a scalar table, with the current date displayed.

```
SELECT MONTHS_BETWEEN(to_date('05082020','mmddyyyy'), to_date('09222020','mmddyyyy')) NumMonths FROM dual;
```

would output the value `-4.45161290322580645161290322580645161129`. Note the negative value, so the function expects the last date first for positive values. Using the `ROUND(#,2)` function outside the `MONTHS_BETWEEN` function would produce `-4.45`.

> The `TO_DATE` function is necessary when hard coding dates to ensure dates are what they were intended. '05082020' could be interpreted as May 8, or August 5, pending which country you live in (Canada and the USA would be different) and more importantly what language keyboard was selected when the operating system was installed. Most Canadian computers are setup with USA keyboards. By using the TO_DATE function, your are explicitly declaring the date and it will work using any international standard.

```
SELECT firstname, birthDate, EXTRACT(month FROM birthDate) as Mon, EXTRACT(year FROM birthDate) as YR
FROM employees;
```

FIRSTNAME	BIRTHDATE	MON	YR

FIRSTNAME	BIRTHDATE	MON	YR
John	12-Feb-72	2	1972
Mary	23-Sep-76	9	1976

The extract function can be used in many queries to obtain values through date manipulation.

Filtering Data (WHERE)

Although we are now able to extract data from a table in the database, it is rare that all the data will always be output. If you run a store for 20 years, you would not want to output all the sales information if you just need to know how many sales were made today, or yesterday. Therefore, the majority of SQL SELECT statements will include a `WHERE` component.

```
SELECT firstname, birthDate
FROM employees
WHERE EXTRACT(month FROM birthDate) = 5;
```

returns all employees who are born in the month of May, regardless of the year they were born.

Comparison and Logical Operators

Symbol / Text	Meaning	Example
=	exactly equal to (strings are case senstive)	studID = 990123456
<	less than (numerically, alphabetically, chronologically	salary < 5000
<=	less than or equal to	numProducts <= 10
>	greater than	salary > 5000

Symbol / Text	Meaning	Example
>=	great than or equal to	numProducts >= 2
<> !=	not equal to	name != 'Bob'
IN and NOT IN	In a subset of values.	productID IN (12, 45, 67, 65)
ANY	used with another comparator and a subset of values	productID > ANY(12,45,67,76)
ALL	used with another comparator and a subset of values	salary >= ALL(2000, 3000, 4000)
BETWEEN and NOT BETWEEN	used to determine if a value is between 2 other values	salary BETWEEN (2000 and 5000)
LIKE and NOT LIKE	similar to =, but enables the use of wildcards	name LIKE 'B%'
IS NULL and IS NOT NULL	determines is the value is NULL or not	middlename IS NULL
AND	determines if two expressions are both true or not	salary => 2000 AND salary <= 5000
OR	determines if one or the other expression are true	studID = 990123456 OR studID = 9906543321

Wildcards

Wild cards allow queries to return results regardless of exact matches. This is extremely powerful in searches and filtering operations. Wild cards will be interpreted as exact matches if used with any comparator other than `LIKE`.

Key Character	Description
%	The percent wildcard specifies that any characters can appear in multiple positions represented by the wildcard.

Key Character	Description
_	The underscore wildcard specifies a single position in which any character can occur.

Examples

LIKE Operator	Description
WHERE LOWER(CustomerName) LIKE 'a%'	Finds any values that starts with "a"
WHERE LOWER(CustomerName) LIKE '%a'	Finds any values that ends with "a"
WHERE LOWER(CustomerName) LIKE '%or%'	Finds any values that have "or" in any position
WHERE LOWER(CustomerName) LIKE '_r%'	Finds any values that have "r" in the second position
WHERE LOWER(CustomerName) LIKE 'a_%_%'	Finds any values that starts with "a" and are at least 3 characters in length
WHERE LOWER(ContactName) LIKE 'a%o'	Finds any values that starts with "a" and ends with "o"
WHERE LOWER(productName) LIKE '_at'	Would find any 3 letter name ending in at (hat, cat, bat, pat, sat, mat, etc....)
WHERE LOWER(productName) = '_at'	would not likely find any results, as the product name would have to be eaxctly '_at' the wildcard is only recognized with the LIKE comparison operator.

Order of Precendence

Like the sstandard order of operations that we learned in beginner mathematics, this needs to be extended to include logical operators as well. Here are a few points to get familiar with order of precendence.

1. multiplication and division (left to right)
2. addition, subtraction, concatenation, negatives (left to right)
3. comparison operators (=, <, <=, >, >=, <>, !=, !<, !>)

4. NOT logical operators
5. AND logical operators
6. OR, IN, LIKE, ALL, BETWEEN, ANY
7. =

This ordering can always be manipulated by using brackets. This is often needed to ensure calcualtions are performed as intended.

```
4 + 3 * 2                  -- returns 10 as the * happens first
(4 + 3) * 2                -- returns 14
true AND false OR false    -- returns false
true OR false AND false    -- return true
(true OR false) AND false  -- returns false
NOT false AND true         -- returns true, but important to note the NOT false results in true AND true
```

Filtering and Aliases

When applying filtering your query, it is possible to use table aliases in the WHERE part of the statement, but it is NOT possible to use field aliases. This is because the SELECT statement does not execute from the top to the bottom. Therefore, the timing of the creation of the alias will impact the ability to reference it or not.

Table aliases are usable in the WHERE part of the statement as they are defined during the FROM part and therefore exist at the time WHERE executes.

However, fields aliases have not been defined at the time WHERE executes and therefore are not accessible.

```
SELECT firstname || ' ' || lastname AS fullname
FROM employees
WHERE upper(fullname) LIKE 'B%';
```

will fail. ***fullname*** does not yet exist when the WHERE statement executes, which results in an error. This statment will have to be rewritten using the original fields:

```
SELECT firstname || ' ' || lastname AS fullname
FROM employees
WHERE upper(firstname) LIKE 'B%';
```

Another Example

```
SELECT studentID, mark, maxmarks, round(100 * mark / maxmarks,2) AS grade
FROM studentMarks
WHERE grade >= 50;    -- will fail

-- needs to be:
SELECT studentID, mark, maxmarks, round(100 * mark / maxmarks,2) AS grade
FROM studentMarks
WHERE round(100 * mark / maxmarks,2) >= 50;
```

Sorting Data (ORDER BY)

Image trying to find your friend in the Toronto (6.1 million people) phone book if it was sorted in the order in which the data was entered and not alphabetical. It would take you forever to find the exact record you are looking for. Therefore, it is very important that query results are sorted in a logical way such that it is easy to navigate the results.

Oracle enforces a primary key integrity constraint by creating a unique index on the primary key. This index is automatically created by Oracle when the constraint is enabled; no action is required by the issuer of the `CREATE TABLE` or `ALTER TABLE` statement to create the index.

In these cases, output from an SQL command is automatically sorted by the primary key unless otherwise specified. In SQL, this is accomplished through the `ORDER BY` part of the `SELECT` statement.

```
SELECT firstname, lastname
FROM employees
ORDER BY lastname;
```

sorts the data alphabetically by last name from A to Z. This sorting can happen in both ascending `ASC` and descending `DESC` order: the default is ascending if not specified.

```
SELECT firstname, lastname
FROM employees
ORDER BY lastname DESC;
```

sorts the data alphabetically by last name from Z to A.

There are three ways to sort things and it is important to know the differences:

alphabetically	sorts A to Z or Z to A
numerically	sorts -9 to 0 to 9
chronologically	sorts the earliest date to the latest date

Sorting Using More Than One Field

When you sort by more than one field, it is important to understand that the second field is only ever even considered when the first field has exact duplicates.

```
SELECT firstname, lastname
FROM employees
ORDER BY lastname, firstname;
```

sorts the entire list by lastname period. If and only if there are duplicates in the lastname field, then only the duplicates are sorted secondarily by firstname.

If sorting is performed with many fields indicated, the probability that the later ones will ever get used drastically decreases. Each level of sorting will only be performed if ALL levels before it exactly match. For example: if two people have the same lastname, same firstname, same birthdate, and live in the same city, then maybe postal code might be used to sort them.

Sorting and Aliases

When using aliases in the ORDER BY part of the statement, there are two consideration;

1. Order of Execution will have no affect on ORDER BY and aliases, as all aliases would have already been defined at the time ORDER BY executes. Therefore, both table and field aliases are eligable for use in the ORDER BY portion.

2. Aliases are often used on calculated fields and this creates another gotcha you need to be aware of. Some built-in, or single-row, functions actualy change the data type of the output. This means that the sorting may not occur as expected as there may be different algorithms for alphabetical, numeric and chronological sorting.

```
SELECT ordernumber, TO_CHAR(orderdate, 'mon dd, yyyy') AS dtOrder, status
FROM orders
WHERE extract(year FROM orderdate) = 2004
ORDER BY dtOrder;
```

This query will output all the orders in 2004, but the order will be incorrect. The `TO_CHAR()` function converts the orderdate from a data type to a strig type. This results in the sorting being alphabetical rather than chronological: meaning all April orders are shown first, followed by August etc. Obviously we want to show January orders first, followed by February. This is achieved by using the original field in the ORDER BY part to keep the data type as a date type and maintaining the chronological ordering.

```
SELECT ordernumber, TO_CHAR(orderdate, 'mon dd, yyyy') AS dtOrder, status
FROM orders
```

```
WHERE extract(year FROM orderdate) = 2004
ORDER BY orderdate;
```

Limiting Rows Returned

Sometimes data resultant sets are very large, but we might only be interested in the last few records, or the first few records. We can use the OFFSET and FETCH commands to determine which rows in a seet are to be returned.

If we wanted to see the first 10 orders in the database that were received we could write:

```
SELECT * FROM orders
ORDER BY orderdate
FETCH NEXT 10 ROWS ONLY;
```

If we wanted to see the last 10 orders:

```
SELECT * FROM orders
ORDER BY orderdate DESC
FETCH NEXT 10 ROWS ONLY;
```

If we want to skip a few records to se only some in the middle somewhere, we can use the OFFSET clause.

```
SELECT * FROM orders
ORDER BY orderdate DESC
OFFSET 5 ROWS
FETCH NEXT 10 ROWS ONLY;
```

skipping the last 5 orders, this query shows the 10 orders just before the 5 that were skipped

ROWNUM

Rownum is a builtin value in all query results that indicates the Rown Number of the results. Therefore you can use ROWNUM to limit the number of rows returned of return exact rows.

```
SELECT * FROM orders
WHERE rownum <= 5;
    -- returns the first 5 rows
SELECT * FROM orders
WHERE rownum BETWEEN 10 and 20
    -- returns the row 10 through 20 (11 rows)
```

You can also use ROWNUM in calculations or to display. If you were building a ranking results, than you would want 1, 2, 3 to represent position.

```
SELECT rownum num, rownum*rownum AS sqr, rownum*rownum*rownum AS cub
FROM orders
WHERE rownum < 10;
```

CRUD

CRUD is a database related term that is one of those terms that almost every IT professional should be aware of. CRUD is a DML term that stands for Create, Read, Update and Delete. These terms refer to `INSERT`, `SELECT`, `UPDATE` and `DELETE` keywords in SQL. Create means Creating a new row of data, selecting data, updating existing data and deleting existing data. Do not confuse this Create with the `CREATE` you will learn in Weeks 4 and 5.

Insert, Update and Delete statements are also know as writable or action queries.

Writing Data

For the remainder of week 3 we will move away from Selection and read only statements and focus on writing to the database. This will include inserting new data rows into the tables, updating existing data rows in tables, and deleting existing rows of data from the database.

INSERT - Inserting New Data Rows

Inserting new rows of data into a database is accomplished using an `INSERT` statement. In SQL there are 4 forms of the `INSERT` statement with different features and requirements.

Standard INSERT Form

The standard `INSERT` statement is formed as:

```
INSERT INTO <tablename> ( <fieldlist comma separated> )
    VALUES ( <Value list comma separated> );

-- example
INSERT INTO offices (officecode, phone, city, addressline1, state, country, postalcode, territory)
    VALUES (8, '+1 905 555 1212', 'Toronto', '200 Young St. N.', 'ON', 'Canada', 'M4A3A1', 'NA');
```

Notes:

- only those fields that are required (not null) need to be present
- the field list can be in any order
- the number of values must match the number of fields
- the values must be in the same order as the fields they match.
- single quotes are used to indicate string values
- the `TO_DATE()` function should alwasy be used if hard coded dates are being entered.

Short-Cut INSERT FORM

The short-cut form of the INSERT statement is:

```
INSERT INTO <tablename> ( <Value list comma separated> );

-- example
INSERT INTO offices VALUES (8, 'Toronto', '+1 905 555 1212', '200 Young St. N., NULL, 'ON', 'Canada', 'M4A3A1', 'NA);
```

Notes:

- ALL fields values must be included, regardless if they are NULL or not.
- the values MUST be in the same order as the table was created
- the number of values must match the number of fields, even if the Primary Key is an autonumbering sequence
- use the value NULL for fields that are not required and a value is not available

Inserting Multiple Rows in One Statement

It is possible to insert more than one row manually in a single statement. It is important to note that this statement has a variety of forms in different DBMSs, so if you are familiar with SQL Server, MySQL or other database, Oracle's for is quite different.

```
INSERT ALL
    INTO <tablename> VALUES ( <value list comma separated >)
    INTO <tablename> VALUES ( <value list comma separated >)
    INTO <tablename> VALUES ( <value list comma separated >)
    SELECT <some field name> FROM <table name>;
    -- note the field and table names in the SELECT do NOT need to from the same table.

-- Example:
INSERT ALL
    INTO offices VALUES (8, 'Toronto', '+1 416 555 1111', '200 Young St. N., NULL, 'ON', 'Canada', 'M4A3A1', 'NA)
    INTO offices VALUES (9, 'Oshawa', '+1 905 555 2222', '155 Simcoe. S., NULL, 'ON', 'Canada', 'N2L3G4', 'NA)
```

```
    INTO offices VALUES (10, 'Montreal', '+1 268 555 3333', '1245 Rue Lavac, NULL, 'QC', 'Canada', 'K3S2H4', 'NA)
    SELECT * FROM dual;
```

notes:

- that there are no commas between the multiple INTO parts
- the standard and short-cut forms can both be used, but typically the shortcut method is used to reduce repetition
- The SELECT statement at the end does not need to be relevent to the data being inserted. Using the dual table uses a minimal footprint for processing and memory.

Inserting Multiple Rows From another Table

It is possible to insert data into a table using data from another table. Use this feature with caution as it absolutely will be creating redundent and repetitive data if used incorrectly. This is typically only used when creating tables through migration of an old database to a new database or importing data into live tables from temporary tables.

```
INSERT INTO <tablename> (<fieldlist>)
    SELECT <fieldlist>
    FROM <tablename>
    WHERE <condition>          -- optional
    ORDER BY <fieldlist>;      -- optional
```

Notes:

- the fieldlist in the SELECT must match the fieldlist in the INSERT (not same names, but smae meaning)
- the order of the two fieldlists (SELECT and INSERT) must be the same
- you can choose not to include the fieldlist in the INSERT line, but then must have all fields in the right order in the SELECT line
- The WHERE clause is optional
- The ORDER BY is also optional and not typically needed unless there are autonumbering sequences involved

UPDATE - Updating Existing Data

Updating existing rows is possible through the `UPDATE` statement. The basic form of `UPDATE` is:

```
UPDATE <tablename>
    SET <field> = <value>,
        <field> = <value>,
        <field> = <value>,
WHERE <condition>;   -- optional,
```

The update statement can update a single field, multiple fields base on the SET part and can update several rows at the same time based on the WHERE part. The optional WHERE part is used in most cases as it is important to very cleary indicate which rows are to be updated. For intance:

```
UPDATE employees SET lastname = 'Smith';   -- BAD, updates ALL employees
```

would update every employee in the companies lastname to 'Smith' rather than the one woman who got married on the weekend. OOPS!

```
UPDATE employees SET lastname = 'Smith'
WHERE lastname = 'Jones';   -- BAD, updates all people with last name Jones to Smith
```

If the woman is changing her name from Jones to Smith, this statement may seem reasonable, but there is no guarantee that there is not more than one employee with the last name Jones. This statement would change all the Jone's to Smith's. OOPS!

> Using the PRIMARY KEY is the best way to clearly indicate the exact row that needs to be updated as by definition it is unique.

```
UPDATE employees SET lastname = 'Smith'
WHERE employeeId = 343;   -- BEST, Primary Key is unique
```

DELETE - Deleting Existing Data

Although the most dangerous of the CRUD statements, the DELETE statement is also the simplist. The basic form of the DELETE statement is:

```
DELETE FROM <tablename>
WHERE <condition> -- optional
```

There is no fieldlist or complexities in the delete statement. It's intention is to delete entire ros and all values in the row(s). If a WHERE clause is not included, this statement will delete all rows from the table indicated; potential disaster. Therefore, it is highly recommend that in most cases, the WHERE clause is indicated and uses a clear indicator field, such as the Primary Key.

Summary

Well that was a long week with lots of things to learn. It is extermly important that you learn these concepts well as they are the foundation for the next 4 weeks and 6 weeks of DBS311 in addition to the foundation of database communication for computer programmers.

Practice Exercises

- Intro to SELECT
- Filtering and Sorting Query
- Limiting results using Fetch, offset and rownum
- Inserting, Updating, and Deleting Rows