**Chapter 5: Advanced SQL**

Table of Contents

[Accessing SQL from a Programming Language 3](#_Toc49571751)

[Java Database Connectivity 3](#_Toc49571752)

[SQL Injections 8](#_Toc49571753)

[Metadata Features 9](#_Toc49571754)

[Transaction Control 9](#_Toc49571755)

[Working with Large Objects 10](#_Toc49571756)

[Procedural Constructs in SQL 11](#_Toc49571757)

[PL/SQL 13](#_Toc49571758)

[Anonymous and Named Blocks 13](#_Toc49571759)

[Conditional Statements 14](#_Toc49571760)

[Functions 15](#_Toc49571761)

[The DECODE Function 18](#_Toc49571762)

[The RANK Function 19](#_Toc49571763)

[Triggers 20](#_Toc49571764)

[Row-Level Triggers 20](#_Toc49571765)

[Statement-Level Triggers 22](#_Toc49571766)

## Accessing SQL from a Programming Language

We can access an SQL database using other programming languages. To achieve this, Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC) are used. ODBC is for languages developed by Microsoft like C, C++ and C# while JDBC is for Java. ODBC and JDBC are Application Program Interfaces (APIs), which are interfaces used by programs to interact with database servers.

Applications can be built on top of a database. For these applications to be able to access the database, the essential steps are:

* Connect with the database server
* Send SQL commands to the database server
* Fetch tuples of results one by one into program variables

### Java Database Connectivity

We will only be looking into JDBC in depth here, not ODBC. JDBC is a Java API for communicating with database systems that support SQL. It supports a variety of features for querying and updating data and retrieving query results. JDBC also supports metadata retrieval, such as queries about relations present in the database and the names and types of relation attributes.

The process for communication with the database is:

* Open a connection
* Creates a ‘statement’ object
* Execute queries using the statement object to send and fetch results
* Exception mechanism to handle errors

Below, there is an example of some Java code to send SQL queries to a database. Note that it is not essential to understand every single bit of this code. Understanding the general idea is enough.

public static void JDBCexample(String dbid, String userid, String passwd)  
*// this function needs the name of the database, the username and password*{  
 try  
 {  
 Class.forName("oracle.jdbc.driver.OracleDriver"); *// load driver  
 // create connection* Connection conn = DriverManager.getConnection(  
 "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);  
 Statement stmt = conn.createStatement(); *// create statement  
 // actual work with statement* stmt.close(); *// close statement* conn.close(); *// close connection* }  
 catch (SQLException sqle)  
 {  
 System.out.println("SQLException : " + sqle);  
 }  
}

JAVA

We have created a function that accepts the database name, the username and the password as arguments. It is a public static function, so it can be called from anywhere in the program without any object.

Inside the function, the first thing we did is to load the JDBC driver. This driver needs to be downloaded from the Oracle website and set in the PATH for the environment variables of Windows so it can be used from anywhere in our system easily. It is just a .jar file that contains a number of class files that each have some separate functions related to JDBC.

Next, we created a connection to the database we want to use. This consists of a string that is used to identify the database, the username and the password. The part of the string after the @ symbol gives the IP name of the database, the port number and the type of the database.

Finally, we create a statement to which we will pass our SQL queries and work with them. We will be examining how to do this next.

After our work is done, we need to close the statement object and the connection we made to the database.

This entire process is put inside a try block. The try and except blocks essentially allow us to catch any exceptions that may occur. Whenever we are taking any input from the user, there is always the possibility that an error will occur due to the user giving an invalid input. When it comes to connections that go outside our program, like this one, it may not be possible to know that an error will occur beforehand. The try block attempts to execute the code, and if it fails, the code in the except block is executed (which is just printing out an error in this case). This prevents the program from simply crashing when an error occurs. We are only catching one type of error here. In actual programs, there will most likely be more errors to catch.

Now let us look at examples of the actual work that can be done inside this function.

We could update our database by passing a query like this:

try  
{  
 stmt.executeUpdate(  
 "INSERT INTO instructor VALUES(77987, 'Kim', 'Physics', 98000)");  
}  
catch (SQLException sqle)  
{  
 System.out.println("Could not insert tuple. " + sqle);  
}

JAVA

Notice that the actual queries are put inside double quotes. Normally, the actual data inside these queries would be collected from the user via the user interface, using checkboxes, buttons etc. and then the query would be created by concatenating everything.

We have also used another try-except statement here. This is because, in general, we should have a separate try-except statement for each step of the program where an error might occur, instead of having a general one that encapsulates everything.

Updating the database is relatively simple compared to retrieving data. When we retrieve data, we also have to store it.

ResultSet rset = stmt.executeQuery(  
 "SELECT dept\_name, AVG(salary) FROM instructor GROUP BY dept\_name");  
while (rset.next())  
{  
 System.out.println(rset.getString("dept\_name") + " " + rset.getFloat(2));  
}

JAVA

Here, we are storing the results from the query in a variable and while there are still results available, we are printing them one tuple at a time.

The print statement is interesting, since we are essentially having to search for each attribute separately depending on the type of the attribute. There is another method to retrieve attributes though. Instead of using rset.getString("dept\_name"), we could have used rset.getString(1). This would retrieve the first attribute from the results, assuming it is a string. We used this second method for the second attribute we printed.

Also notice that we are using a different method, executeQuery as opposed to executeUpdate.

Things get even more complicated if there is a possibility for a value to be NULL, since then, we have to check for that as well.

int a = rset.getInt("a");  
if (rset.wasNull()) Systems.out.println("Got null value");

JAVA

Notice that this part of the program did not need its own try-except block, since we are not taking any input for the data directly into the program, which means there is much less scope for things to do wrong.

### SQL Injections

When taking inputs from the user to create queries, we should never concatenate strings with the inputs to create the queries, such as what is done below:

ResultSet rset = stmt.executeQuery(  
 "SELECT \* FROM instructor WHERE name = " + name);

JAVA

This is because the user could potentially give some input that would wreak havoc on our database or steal data. For example, it is entirely possible to give an input like X; UPDATE instructor SET salary = salary + 10000 which would make the actual query:

ResultSet rset = stmt.executeQuery(  
 "SELECT \* FROM instructor WHERE name = X;" +  
 "UPDATE instructor SET salary = salary + 10000");

JAVA

This sort of an attack is called an SQL injection attack, and it was a huge problem at a time. Now however, programs are developed so as to prevent such attacks from happening. One easy way to prevent such an attack is to use prepared statements. This is essentially an SQL statement that has some question marks in it that are replaced later on.

PreparedStatement pStmt = conn.prepareStatement(  
 "INSERT INTO instructor VALUES(?, ?)");  
pStmt.setString(1, "88877");

JAVA

This prevents an SQL injection attack since, if the input is not in the right format, the query will fail anyways. There is no scope to add statements to the original query.

### Metadata Features

We can retrieve metadata information about our database using queries like this one:

ResultSetMetaData rsmd = rs.getMetaData();  
for(int i = 1; i <= rsmd.getColumnCount(); i++) {  
 System.out.println(rsmd.getColumnName(i));  
 System.out.println(rsmd.getColumnTypeName(i));  
}

JAVA

### Transaction Control

By default, each SQL statement is treated as a separate transaction and committed automatically. This can be a bad idea for some situations where atomicity must be maintained while executing several statements together. In these situations, it is possible to turn off the automatic commits and make the commits or rollbacks manually.

conn.setAutoCommit(false);  
*// turn off auto commits  
// some statements that are checked to see if they all succeeded*if () *// statements all succeeded*  
 conn.commit();  
else *// something went wrong* conn.rollback();  
conn.setAutoCommit(true);

JAVA

### Working with Large Objects

The functions getBlob() and getClob() allow us to retrieve large objects. The data from these objects can be read using the getBytes() function.

To update a large object, we need to associate it with an open stream:

blob.setBlob(int parameterIndex, InputStream inputStream);

JAVA

## Procedural Constructs in SQL

Procedural constructs use something called modular coding. Generally, when we write code, instead of doing all our work in one go, we divide it into parts by using functions. This allows us to keep things much more organized and reuse code.

So far, we have been using pure SQL. SQL also provides a module language, which permits us to create procedures. These procedures allow us to use everything from if else statements to for loops. This means we can store these procedures in our database and then call them when we need. It also allows external applications to operate on the database using the procedures without knowing the internal details of the database. We can even create the procedures in other languages and then use it with our SQL database. There are some object-oriented aspects to modular SQL, but that is not covered here.

To be more accurate, procedures are not functions. Functions also exist in SQL. The main difference is that a function must have a RETURN statement, while a procedure must not (even though it can return values in other ways, as we will see later on).

*/\* this function counts the number of instructors in some department \*/*CREATE OR REPLACE FUNCTION dept\_count (dept\_name VARCHAR2)  
*/\* function name and parameter name with type \*/* RETURNS INTEGER  
 */\* return type \*/* BEGIN  
 DECLARE d\_count INTEGER;  
 */\* create an integer and send count into the integer \*/* SELECT COUNT(\*) INTO d\_count  
 FROM instructor  
 WHERE instructor.dept\_name = dept\_name  
 RETURN d\_count;  
 END

SQL

If we wanted to create a procedure instead of a function, we would just replace the word FUNCTION with the word PROCEDURE.

A fun thing about functions is that, while declaring variables, we do not need to explicitly state what the data type is. Instead, we can perform dynamic type casting by writing something like employee.salary%TYPE. This would give the variable the same data type that we used for that particular attribute, which makes it very useful in situations where we might not know what data type to use or it may change.

If we wanted to select more than one attribute and store the results in other variables, we would list the variables one after another in the same fashion, separated by commas.

Another problem with the given function is that we might get a NULL value as a result, which is a problem. Instead of just returning the value like we did, we should return NVL(d\_count, -1). This is a function that will return -1 if the value of d\_count is NULL.

In SQL, we can use the function or procedure like this:

SELECT dept\_count FROM dual;

SQL

And in Java we call it like this:

CallableStatement cStmt1 = conn.prepareCall("{? = call some function(?)}");  
CallableStatement cStmt2 = conn.prepareCall("{call some procedure(?,?)}");

JAVA

## PL/SQL

SQL is extremely strict about the way we write code, which can make it uncomfortable, or sometimes even frustrating, to write code that fulfils our intentions. Fortunately, all the databases in the world support some simple logic manipulation such as if-else statements, loops etc. In Oracle, this is called Programmable SQL, or PL/SQL. Essentially, it allows a programming environment inside the database unit.

### Anonymous and Named Blocks

A block is the smallest meaningful grouping of code. It is a unit of code that provides execution and scoping boundaries for variable declarations and exception handling.

The first block we will look at is the anonymous block. It does not have a name and thus cannot be called from anywhere. They simply serve as containers that execute PL/SQL statements. They are not stored in the server.

*-- this is a comment*SET SERVEROUTPUT ON  
*-- output is now visible*BEGIN  
 DBMS\_OUTPUT.PUT\_LINE('Hello World!');  
END;

*-- this symbol is used at the end to execute the code*  
/

SQL

Anonymous blocks are normally used for debugging purposes by programmers in the development phase.

Named blocks on the other hand, are stored in the database and can be reused. The functions and procedures we have already seen were named blocks. The entire code for a function or a procedure is stored on the database, with comments and all.

### Conditional Statements

BEGIN  
 IF 1=2  
 THEN DBMS\_OUTPUT.PUT\_LINE('True');  
 ELSIF 1=(2/2)  
 THEN DBMS\_OUTPUT.put\_line ('Kinda True');  
 ELSE DBMS\_OUTPUT.put\_line ('False');  
END IF;  
END;  
/

SQL

Notice the spelling for ELSIF.

### Functions

We have already gone over functions, but here are a few more points.

* The basic structure of a function looks like this:

CREATE OR REPLACE FUNCTION function\_name(parameter\_name parameter\_type)  
 RETURNS return\_type  
 IS  
 local\_variable\_name INT := 0;  
  
 BEGIN  
 *-- executable statements* EXCEPTION  
 *-- exception handling statements* END;

SQL

* Functions have two types of parameters, IN parameters that are just normal parameters, and OUT parameters which are another way to return values from the function.
* In the parameter type and return type, precision is not allowed. This means VARCHAR2(20) is invalid but VARCHAR2 is valid.
* Notice the := symbol in the local variable declaration. This symbol is used for initialization.
* When writing the executable statements, if we use a SELECT statement, we also need to use an INTO statement. This is because the results of the SELECT statement need to be stored in a variable for us to be able to work with it.

SELECT SUM(unit\_price \* quantity), COUNT(quantity)  
INTO local\_variable\_name, another\_local\_variable  
FROM ordered\_items  
WHERE status = 'Shipped';

RETURN local\_variable\_name;

SQL

There are three ways in which we can call a function. One way, is to call it readily using a fixed parameter.

SELECT some\_function('CSE') FROM dual;

SQL

Note that dual is a dummy table. It does not actually exist and is simply used to make SQL statements valid when there is actually no table to select anything from.

Another way is to print the output of the function without a parameter.

DBMS\_OUTPUT.PUT\_LINE('Sales: ' || get\_total\_sales);

SQL

Note that the || sign is for concatenation.

The third way is to call the function on the result of some query we are running.

SELECT id, get\_cgpa(id) FROM students;

SQL

Example:

CREATE OR REPLACE FUNCTION get\_status(PID IN NUMBER)  
 RETURNS VARCHAR2  
 IS  
  
 v\_status VARCHAR2(20);  
 v\_cgpa students.cgpa%TYPE;  
  
 BEGIN  
  
 SELECT cgpa INTO v\_cgpa  
 FROM students  
 WHERE sid = PID;  
  
 IF v\_cgpa BETWEEN 3.7 AND 4.0  
 THEN v\_status := 'Studious';  
 ELSE v\_status := 'Not Studious';  
 END IF;  
  
 RETURN v\_status;  
 END;  
/

SQL

SELECT id, get\_status(id) AS status FROM students;

SQL

Since functions are stored on the database, we can even retrieve the entire function from the database if we needed to.

SELECT text FROM user\_source WHERE name = 'GET\_STATUS' ORDER BY line;

SQL

## The DECODE Function

DECODE compares an expression to a set of search values and returns a result if they are found to be equal. If no matches are found, a default result is returned. If no default value is given, NULL is returned.

The DECODE function is similar to a switch statement from traditional programming languages. It is not available in all databases.

Consider a scenario where an integer 0 or 1 has been used to specify whether a particular student is an undergraduate student or a postgraduate student. We can print the data for the students like this:

SELECT sid, name, cgpa, student\_status FROM students;

SQL

This will of course print the 1s and 0s under the student\_status attribute. However, an end program may want to avoid this and print the actual strings undergraduate or postgraduate, but without changing the underlying data. This is where the DECODE function comes in.

SELECT sid, name, cgpa, DECODE(student\_status, 0, 'Undergraduate', 1, 'Postgraduate') AS status FROM students;

SQL

## The RANK Function

The RANK function is an analytical function that calculates the rank of a value in a set of values. Essentially, it assigns a rank for the records based on some value. If there are equal values, the highest value is assigned to all the records that have the equal value and the corresponding number of ranks is skipped after that. E.g. if 3 people have an equal score and they all qualify for 3rd place, they will all be given the rank 3, the ranks 4 and 5 will be skipped, and the person who comes next will be given the rank 6.

The query below assigns a rank to the students based on their CGPA, with the highest CGPA getting the top rank. Note that the query also orders the results based on the CGPA (and thus rank).

SELECT sid, name, cgpa, RANK() OVER (ORDER BY cgpa DESC) AS position FROM students;

SQL

## Triggers

A trigger is an event-oriented process. It is similar to a function, in that it executes some code, but it is not something that can be called. It occurs automatically whenever some event, such as an insertion, update or deletion, occurs. For example, clicking on a button that causes some event is a type of trigger.

To design a trigger mechanism, we must specify the condition under which the trigger will be executed and the actions to be taken when the trigger is executed. Since triggers are event-oriented, they are not explicitly called like functions, but are automatically fired whenever the condition is met. In fact, it is not even possible to call a trigger manually.

Triggers are of two types, system-level triggers and DML triggers. For now, we will be ignoring system-level triggers. DML triggers can be further subdivided into two categories, row-level triggers and statement-level triggers.

### Row-Level Triggers

Some examples of row-level triggers are:

* An account holder withdrawing money from their account must have a balance greater than the withdrawal amount.
* A new student should have an automatically generated student ID.
* When there is an update to a table, then time of the update should automatically be recorded.

All three of these examples have something in common. They are all executed for each possible record’s change. For example, if we add 5 new students to our database at the same time, student IDs will be generated for all 5 of them individually. Note that the trigger executes when we make a commit.

Triggers that behave in this manner are called row-level triggers. They are executed for each applicable change in each record. The number of rows determine the number of executions.

CREATE OR REPLACE TRIGGER trigger\_name  
 BEFORE | AFTER  
 INSERT OR DELETE OR UPDATE OF column1, column2,  
 ON table\_name  
 FOR EACH ROW  
 REFERENCING OLD AS old\_name  
 NEW AS new\_name  
 WHEN (conditions)  
 DECLARE  
 BEGIN  
  
 EXCEPTION  
  
 END;

SQL

This trigger is executed before or after there is an update on either column1 or column2 of table\_name. The keywords FOR EACH ROW indicates that it is a row-level trigger. The old data is referenced as old\_name and the new data is referenced as new\_name. Note that the code is incomplete since the actual work of the trigger is not defined here, only the declaration. This code may seem a little confusing, but will not be discussed further here.

### Statement-Level Triggers

Statement-level triggers work in the same way as row-level triggers, except that they are executed just one time when the change occurs, no matter how many rows are affected. Thus, if we add 5 new students to our database, a statement-level trigger would only execute once.

The syntax for a statement-level trigger is the same as for a row-level trigger, except that the line FOR EACH ROW is removed.