UNIT III

SQL: Basic SQL, Query, union, interest, except, Nested Queries, Aggregated Operation, Null values, Embedded SQL, cursors, Database connectivity(ODBC and JDBC), Triggers and Active database, designing active databases.

Basic SQL

Structured Query Language (SQL) is the most widely used commercial relational database language. It was originally developed at IBM in the SEQUEL-XRM and System-R projects (1974-1977). Almost immediately, other vendors introduced DBMS products based on SQL, and it is now a de facto standard. SQL continues to evolve in response to changing needs in the database area. The current ANSI/ISO standard for SQL is called SQL: 1999. While not all DBMS products support the full SQL: 1999 standard yet, vendors are working toward this goal and most products already support the core features. The SQL: 1999 standard is very close to the previous standard, SQL-92.

SQL Standards Conformance: SQL: 1999 has a collection of features called Core SQL that a vendor must implement to claim conformance with the SQL: 1999 standard. It is estimated that all the major vendors can comply with Core SQL with little effort. Many of the remaining features are organized into packages.

The SQL language has several aspects to it.

- The Data Manipulation Language (DML): This subset of SQL allows users to pose queries and to insert, delete, and modify rows. Queries are the main focus of this chapter. We covered DML commands to insert, delete, and modify rows in Chapter 3.
- The Data Definition Language (DDL): This subset of SQL supports the creation, deletion, and modification of definitions for tables and views.
- *Integrity constraints* can be defined on tables, either when the table is created or later. Although the standard does not discuss indexes, commercial implementations also provide commands for creating and deleting indexes.
- Triggers and Advanced Integrity Constraints: The new SQL: 1999 standard includes support for *triggers,* which are actions executed by the DBMS whenever changes to the database meet conditions specified in the trigger. We cover triggers in this chapter. SQL allows the use of queries to specify complex integrity constraint specifications.
- Embedded and Dynamic SQL: Embedded SQL features allow SQL code to be called from a host language such as C or COBOL. Dynamic SQL features allow a query to be constructed (and executed) at run-time.
- Client-Server Execution and Remote Database Access: These commands control how a *client* application program can connect to an SQL database *server,* or access data from a database over a network.
- Transaction Management: Various commands allow a user to explicitly control aspects of how a transaction is to be executed.
- Security: SQL provides mechanisms to control users' access to data objects such as tables and views.

- Advanced features: The SQL: 1999 standard includes object-oriented features, recursive queries, decision support queries and also addresses emerging areas such as data mining, spatial data and text and XML data management.

Form of Basic SQL Query

Sailors(*sid:* integer, *sname:* string, *rating:* integer, *age:* real)

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Boats(*bid:* integer, *bname:* string, *color:* string)

bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Reserves (*sid:* integer, *bid:* integer, *day:* date)

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

The basic form of an SQL query is as follows:
SELECT [DISTINCT] select-list
FROM from-list
WHERE qualification

Every query must have a SELECT clause, which specifies columns to be retained in the result, and a FROM clause, which specifies a cross-product of tables. The optional WHERE clause specifies selection conditions on the tables mentioned in the FROM clause.

Examples of Basic SQL Queries

Find the names and ages of all sailors.

SELECT DISTINCT S.sname, S.age FROM Sailors S

If we omit the keyword DISTINCT, we would get a copy of the row *(s,a)* for each sailor with name s and age *a;* the answer would be a m*ultiset* of rows. A **multiset** is similar to a set in that it is an unordered collection of elements, but there could be several copies of each element, and the number of copies is significant-two multisets could have the same elements and yet be different because the number of copies is different for some elements. For example, {a, b, b} and {b, a, b} denote the same multiset, and differ from the multiset {a, a, b}.

Find all sailors with a rating above 7.

SELECT S.sid, S.sname, S.rating, S.age FROM Sailors AS S WHERE S.rating > 7

- The from-list in the FROM clause is a list of table names. A table name can be followed by a range variable; a range variable is particularly useful when the same table name appears more than once in the from-list.
- The select-list is a list of (expressions involving) column names of tables named in the from-list. Column names can be prefixed by a range variable.
- The qualification in the WHERE clause is a boolean combination (i.e., an expression using the logical connectives AND, OR, and NOT) of conditions of the form *expression* op *expression,* where op is one of the comparison operators {<, <=, =, <>, >=, >}.2 An *expression* is a *column* name, a *constant,* or an (arithmetic or string) expression.
- The DISTINCT keyword is optional. It indicates that the table computed as an answer to this query should not contain *duplicates,* that is, two copies of the same row. The default is that duplicates are not eliminated.

Find the names of sailors 'Who have reserved boat number 103.

SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid = R.sid AND R.bid=103

Find the sids of sailors who have reserved a red boat.

SELECT R.sid FROM Boats B, Reserves R WHERE B.bid = R.bid AND 8.color = 'red'

Find the names of sailors 'Who have reserved a red boat.

SELECT S.sname FROM Sailors S, Reserves R, Boats B

```
WHERE S.sid = R.sid AND R.bid = 13.bid AND B.color = 'red'
```

Find the colors of boats reserved by Lubber.

```
SELECT B.color FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND S.sname = 'Lubber'
```

Find the names of sailors who have Reserved at least one boat.

```
SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid = R.sid
```

Expressions and Strings in the SELECT Command

Compute increments for the ratings of persons who have sailed two different boats on the same day.

```
SELECT S.sname, S.rating+1 AS rating FROM Sailors S, Reserves R1, Reserves R2
WHERE S.sid = R1.sid AND S.sid = R2.sid
        AND R1.day = R2.day AND R1.bid <> R2.bid
```

Find the ages of sailors whose name begins and ends with B and has at least three characters.

```
SELECT S.age FROM Sailors S WHERE S.sname LIKE 'B.%B'
```

UNION, INTERSECT, AND EXCEPT

Find the names of sailors who have reserved a red or a green boat.

```
SELECT S.sname FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid
        AND (B.color = 'red' OR B.color = 'green')
```

```
SELECT S.sname FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
UNION
SELECT S2.sname
FROM Sailors S2, Boats B2, Reserves H2
WHERE S2.sid = H2.sid AND R2.bid = B2.bicl AND B2.color = 'green'
```

Find the names of sailor's that have reserved both a red and a green boat.

```
SELECT S.sname FROM Sailors S, Reserves RI, Boats BI, Reserves R2, Boats B2
WHERE S.sid = Rl.sid AND R1.bid = Bl.bid
        AND S.sid = R2.sid AND R2.bid = B2.bid
        AND B1.color='red' AND B2.color = 'green'
```

```
SELECT S.sname FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
INTERSECT
SELECT S2.sname FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

Find the sids *of all sailors who have reserved red boats but not green boats.*

```
SELECT S.sid FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
EXCEPT
SELECT S2.sid
FROM Sailors S2, Reserves R2, Boats B2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

Find all sids of sailors who have a rating of 10 or *reserved boat 104.*

```
SELECT S.sid FROM Sailors S WHERE S.rating = 10
UNION
SELECT R.sid FROM Reserves R WHERE R.bid = 104
```

Introduction to Nested Queries

One of the most powerful features of SQL is nested queries. A nested query is a query that has another query embedded within it; the embedded query is called a sub-query. The embedded query can of course be a nested query itself; thus queries that have very deeply nested structures are possible. When writing a query, we sometimes need to express a condition that refers to a table that must itself be computed. The query used to compute this subsidiary table is a sub-query and appears as part of the main query. A sub-query typically appears within the WHERE clause of a query. Sub-queries can sometimes appear in the FROM clause or the HAVING clause.

Find the names of sailors who have reserved boat 103.

```
SELECT S.sname FROM Sailors S
WHERE S.sid IN ( SELECT R.sid FROM Reserves R WHERE R.bid = 103 )
```

Find the names of sailors 'who have reserved a red boat.

```
SELECT S.sname FROM Sailors S
WHERE S.sid IN ( SELECT R.sid FROM Reserves R
                 WHERE R. bid IN (SELECT B.bid FROM Boats B
```

```
                                    WHERE B.color = 'red'))
```

Find the names of sailors who have reserved both a red and a green boat.

```
SELECT S.sname FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
        AND S.sid IN ( SELECT S2.sid FROM Sailors S2, Boats B2, Reserves R2
                                WHERE S2.sid = R2.sid AND R2.bid = B2.bid
                                AND B2.color = 'green' )
```

Find the names of sailors who have reserved all boats.

```
SELECT S.sname FROM Sailors S
WHERE NOT EXISTS (( SELECT B.bid FROM Boats B )
EXCEPT
(SELECT R. bid FROM Reserves R
WHERE R.sid = S.sid ))
```

Note that this query is correlated--for each sailor *S,* we check to see that the set of boats reserved by *S* includes every boat. An alternative way to do this query without using EXCEPT follows:

```
SELECT S.sname FROM Sailors S
WHERE NOT EXISTS ( SELECT B.bid FROM Boats B
WHERE NOT EXISTS ( SELECT R. bid FROM Reserves R
                                WHERE R.bid = B.bid
                                        AND R.sid = S.sid ))
```

Correlated Nested Queries Set

In the nested queries seen thus far, the inner sub-query has been completely independent of the outer query. In general, the inner sub-query could depend on the row currently being examined in the outer query (in terms of our conceptual evaluation strategy). Let us rewrite the following query once more.

Find the names of sailors who have reserved boat number 103.

```
SELECT S.sname FROM Sailors S
WHERE EXISTS ( SELECT * FROM Reserves R
                        WHERE R.bid = 103
                                AND R.sid = S.sid )
```

Comparison Operators

We have already seen the set-comparison operators EXISTS, IN, and UNIQUE, along with their negated versions. SQL also supports op ANY and op ALL, where op is one of the arithmetic comparison operators {<, <=, =, <>, >=, >}. (SOME is also available, but it is just a synonym for ANY.)

Find sailors whose rating is better than some sailor called Horatio.

```
SELECT S.sid FROM Sailors S
WHERE S.rating > ANY ( SELECT S2.rating FROM Sailors S2
                       WHERE S2.sname = 'Horatio' )
```

Find sailors whose rating is better than every sailor' called Horatio.
Find the sailors with the highest rating.

```
SELECT S.sid FROM Sailors S
WHERE S.rating >= ALL ( SELECT S2.rating FROM Sailors S2 )
```

Aggregative Operators

In addition to simply retrieving data, we often want to perform some computation or summarization. As we noted earlier in this chapter, SQL allows the use of arithmetic expressions. We now consider a powerful class of constructs for computing *aggregate values* such as MIN and SUM. These features represent a significant extension of relational algebra. SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
4. MAX (A): The maximum value in the A column.
5. MIN (A): The minimum value in the A column.

Find the average age of all sailors.

```
SELECT AVG (S.age) FROM Sailors S
```

Find the average age of sailors with a rating of 10.

```
SELECT AVG (S.age) FROM Sailors S WHERE S.rating = 10
```

Find the name and age of the oldest sailor.

```
SELECT S.sname, MAX (S.age) FROM Sailors S
```

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age = ( SELECT MAX (S2.age) FROM Sailors S2 )
```

Count the number of sailors.

```
SELECT COUNT (*) FROM Sailors S
```

Count the number of different sailor names.

```
SELECT COUNT ( DISTINCT S.sname ) FROM Sailors S
```

Find the names of sailors who are older than the oldest sailor with a rating of 10.

```
SELECT S.sname FROM Sailors S
WHERE S.age > ( SELECT MAX ( S2.age ) FROM Sailors S2
                WHERE S2.rating = 10 )
```

Using ALL, this query could alternatively be written as follows:

```
SELECT S.sname FROM Sailors S
WHERE S.age > ALL ( SELECT S2.age FROM Sailors S2
                    WHERE S2.rating = 10 )
```

The **GROUP BY and HAVING Clauses**

Find the age of the youngest sailor for each rating level.

```
SELECT MIN (S.age) FROM Sailors S WHERE S.rating = i
```

```
SELECT S.rating, MIN (S.age) FROM Sailors S
GROUP BY S.rating
```

The general form of an SQL query with these extensions is:

```
SELECT [ DISTINCT] select-list
FROM from-list
WHERE 'qualification
GROUP BY grouping-list
HAVING group-qualification
```

Find the age of the youngest sailor who is eligible to vote (i.e., is at least 18 years old) for each rating level with at least one such sailors.

```
SELECT S.rating, MIN (S.age) AS minage FROM Sailors S WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```

For each red boat; find the number of reservations for this boat.

```
SELECT B.bid, COUNT (*) AS reservationcount FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = 'red'
GROUP BY B.bid
```

Find the average age of sailors for each rating level that has at least two sailors.

```
SELECT S.rating, AVG (S.age) AS avgage FROM Sailors S
GROUP BY S.rating
HAVING COUNT (*) > 1
```

Find the average age of sailors 'Who are of voting age (i.e., *at least 18 years old) for each 'rating level that has at least two sailors.*

```
SELECT S.rating, AVG ( S.age ) AS avgage
FROM Sailors S
WHERE S. age >= 18
GROUP BY S.rating
HAVING 1 < ( SELECT COUNT (*) FROM Sailors S2 WHERE S.rating = S2.rating )
```

Find the average age of sailors who are of voting age (i.e., at least 18 years old) for each rating level that has at least two such *sailors.*

```
SELECT S.rating, AVG ( S.age ) AS avgage FROM Sailors S
WHERE S. age> 18
GROUP BY S.rating
HAVING 1 < ( SELECT COUNT (*) FROM Sailors S2
                    WHERE S.rating = S2.rating AND S2.age >= 18 )
```

Find those ratings for which the average age of sailors is the minimum overall ratings.

```
SELECT S.rating
FROM Sailors S
WHERE AVG (S.age) = ( SELECT MIN (AVG (S2.age)) FROM Sailors S2
                              GROUP BY S2.rating )
```

NULL values

SQL provides special column value called *null* to use in such situations. We use *null* when the column value is either u*nknown* or *inapplicable.* Using our Sailor table definition, we might enter the row (98. *Dan, null,* 39) to represent Dan. The presence of *null* values complicates many issues, and we consider the impact of *null* values on SQL in this section.

Comparison using Null values

Consider a comparison such as *rating* = 8. If this is applied to the row for Dan, is this condition true or false'? Since Dan's rating is unknown, it is reasonable to say that this comparison should evaluate to the value unknown. In fact, this is the C::lse for the comparisons *rating*> 8 and *rating* < 8 &'3 well. Perhaps less obviously, if we compare two *null* values using <, >, =, and so on, the result is always unknown. For example, if we have *null* in two distinct rows of the sailor relation, any comparison returns unknown.

SQL also provides a special comparison operator IS NULL to test whether a column value is *null;* for example, we can say *rating* IS NULL, which would evaluate to true on the row representing Dan. We can also say *rating* IS NOT NULL, which would evaluate to false on the row for Dan.

Logical connectivity's – AND, OR and NOT

Now, what about Boolean expressions such as ra*ting* = 8 OR *age* < 40 and ra*ting* = 8 AND *age* < 40? Considering the row for Dan again, because *age* < 40, the first expression evaluates to true regardless of the value of *rating,* but what about the second? We can only say unknown.

But this example raises an important point, once we have *null* values, we must define the logical operators AND, OR, and NOT using a *three-valued* logic in which expressions evaluate to true, false, or unknown. We extend the usual interpretations of AND, OR, and NOT to cover the case when one of the arguments is unknown &., follows. The expression NOT unknown is defined to be unknown.

OR of two arguments evaluates to true if either argument evaluates to true, and to unknown if one argument evaluates to false and the other evaluates to unknown. (If both arguments are false, of course, OR evaluates to false.) AND of two arguments evaluates to false if either argument evaluates to false, and to unknown if one argument evaluates to unknown and the other evaluates to true or unknown. (If both arguments are true, AND evaluates to true.)

Impact on SQL Constructs

Boolean expressions arise in many contexts in SQL, and the impact of *null* values must be recognized. For example, the qualification in the WHERE clause eliminates rows (in the cross-product of tables named in the FROM clause) for which the qualification does not evaluate to true. Therefore, in the presence of *null* values, any row that evaluates to false or unknown is eliminated. Eliminating rows that evaluate to unknown h&') a subtle but significant impact on queries, especially nested queries involving EXISTS or UNIQUE.

Another issue in the presence of *null* values is the definition of when two rows in a relation instance are regarded as *duplicates.* The SQL definition is that two rows are duplicates if corresponding columns are either equal, or both contain *null.* Contrast this definition with the

fact that if we compare two *null* values using =, the result is unknown! In the context of duplicates, this comparison is implicitly treated as true, which is an anomaly.

As expected, the arithmetic operations +, -, *, and / all return null if one of their arguments is *null.* However, nulls can cause some unexpected behavior with aggregate operations. COUNT(*) handles *'null* values just like other values; that is, they get counted. All the other aggregate operations (COUNT, SUM, AVG, MIN, MAX, and variations using DISTINCT) simply discard *null* values--thus SUM cannot be understood as just the addition of all values in the (multi)set of values that it is applied to; a preliminary step of discarding all *null* values must also be accounted for. As a special case, if one of these operators-other than COUNT-is applied to *only* null values, the result is again *null.*

Outer Joins

Some interesting variants of the join operation that rely on *null* values, called outer joins, are supported in SQL. Consider the join of two tables, say Sailors Reserves. Tuples of Sailors that do not match some row in Reserves according to the join condition c do not appear in the result. In an outer join, on the other hand, Sailor rows without a matching Reserves row appear exactly once in the result, with the result columns inherited from Reserves assigned *null* values.

In fact, there are several variants of the outer join idea. In a **left outer join**, Sailor rows without a matching Reserves row appear in the result, but not vice versa. In a **right outer join**, Reserves rows without a matching Sailors row appear in the result, but not vice versa. In a **full outer join**, both Sailors and Reserves rows without a match appear in the result. (Of course, rows with a match always appear in the result, for all these variants, just like the usual joins, sometimes called *inner* joins.)

SQL allows the desired type of join to be specified in the FROM clause. For example, the following query lists *(sid, bid)* pairs corresponding to sailors and boats they have reserved:

```
SELECT S.sid, R.bid FROM Sailors S NATURAL LEFT OUTER JOIN Reserves R
```

The NATURAL keyword specifies that the join condition is equality on all common attributes (in this example, *sid),* and the WHERE clause is not required (unless we want to specify additional, non-join conditions). On the instances of Sailors and Reserves shown above, this query computes the result shown below.

sid	*bid*
22	101
31	*null*
58	103

Disallowing NULL values

We can disallow *null* values by specifying NOT NULL as part of the field definition; for example, *sname* CHAR(20) NOT NULL. In addition, the fields in a primary key are not allowed to take on *null* values. Thus, there is an implicit NOT NULL constraint for every field listed in a PRIMARY KEY constraint.

Complex Integrity Constraints in SQL

Constraints over a Single Table

We can specify complex constraints over a single table using table constraints, which have the form CHECK *conditional-expression.* For example, to ensure that *rating* must be an integer in the range 1 to 10, we could use:

```
CREATE TABLE Sailors ( sid INTEGER, sname CHAR(10), rating INTEGER,
age REAL, PRIMARY KEY (sid),
CHECK (rating >= 1 AND rating <= 10 ))
```

To enforce the constraint that Interlake boats cannot be reserved, we could use:

```
CREATE TABLE Reserves (sid INTEGER, bid INTEGER, day DATE,
FOREIGN KEY (sid) REFERENCES Sailors
FOREIGN KEY (bid) REFERENCES Boats
CONSTRAINT noInterlakeRes CHECK ( 'Interlake' <> ( SELECT B.bname
FROM Boats B WHERE B.bid = Reserves.bid )))
```

When a row is inserted into Reserves or an existing row is modified, the *conditional expression* in the CHECK constraint is evaluated. If it evaluates to false, the command is rejected.

Domain Constraints and Distinct Types

A user can define a new domain using the CREATE DOMAIN statement, which uses CHECK constraints.

```
CREATE DOMAIN ratingval INTEGER DEFAULT 1
CHECK ( VALUE >= 1 AND VALUE <= 10 )
```

INTEGER is the underlying, or *source,* type for the domain ratingval, and every ratingval value must be of this type. Values in ratingval are further restricted by using a CHECK constraint; in defining this constraint, we use the keyword VALUE to refer to a value in the domain. By using this facility, we can constrain the values that belong to a domain using the full power of SQL queries. Once a domain is defined, the name of the domain can be used to restrict column values in a table; we can use the following line in a schema

declaration, for example:

rating ratingval

The optional DEFAULT keyword is used to associate a default value with a domain.

If the domain ratingval is used for a column in some relation and no value is entered for this column in an inserted tuple, the default value 1 associated with ratingval is used.

SQL's support for the concept of a domain is limited in an important respect. For example, we can define two domains called SailorId and BoatId, each using INTEGER as the underlying type. The intent is to force a comparison of a SailorId value with a BoatId value to always fail (since they are drawn from different domains); however, since they both have the same base type, INTEGER, the comparison will succeed in SQL. This problem is addressed through the introduction of distinct types in SQL:1999:

CREATE TYPE ratingtype AS INTEGER

This statement defines a new *distinct* type called ratingtype, with INTEGER as its source type. Values of type ratingtype can be compared with each other, but they cannot be compared with values of other types. In particular, ratingtype values are treated as being distinct from values of the source type, INTEGER--we cannot compare them to integers or combine them with integers (e.g., add an integer to a ratingtype value). If we want to define operations on the new type, for example, an *average* function, we must do so explicitly; none of the existing operations on the source type carryover.

Assertions: ICs over Several Tables

Table constraints are associated with a single table, although the conditional expression in the CHECK clause can refer to other tables. Table constraints are required to hold *only* if the associated table is nonempty. Thus, when a constraint involves two or more tables, the table constraint mechanism is sometimes cumbersome and not quite what is desired. To cover such situations, SQL supports the creation of assertions, which are constraints not associated with anyone table.

As an example, suppose that we wish to enforce the constraint that the number of boats plus the number of sailors should be less than 100. (This condition might be required, say, to qualify as a 'small' sailing club.) We could try the following table constraint:

```
CREATE TABLE Sailors ( sid INTEGER, sname CHAR ( 10) , rating INTEGER,
age REAL, PRIMARY KEY (sid),
CHECK ( rating >= 1 AND rating <= 10)
CHECK ( ( SELECT COUNT (S.sid) FROM Sailors S ) + ( SELECT COUNT (B. bid)
FROM Boats B ) < 100 ))
```

This solution suffers from two drawbacks. It is associated with Sailors, although it involves Boats in a completely symmetric way. More important, if the Sailors table is empty, this constraint is defined (as per the semantics of table constraints) to always hold, even if we have more than 100 rows in Boats! We could extend this constraint specification to check that Sailors is nonempty, but this approach becomes cumbersome. The best solution is to create an assertion, as follows:
CREATE ASSERTION smallClub
CHECK ((SELECT COUNT (S.sid) FROM Sailors S) + (SELECT COUNT (B. bid) FROM Boats B) < 100

PL/SQL:
PL/SQL stands for procedural language-standard query language. It is a significant member of Oracle programming tool set which is extensively used to code server side programming. Similar to SQL language PL/SQL is also a case-insensitive programming language.

Blocks

Generally a program written in PL/SQL language is divided into blocks. We can say blocks are basic programming units in PL/SQL programming language.

PL/SQL Blocks contain set of instructions for oracle to execute, display information to the screen, write data to file, call other programs, manipulate data and many more.

Does Blocks supports DDL statements?

Yes, PL/SQL blocks support all DML statements and using Native Dynamic SQL (NDS) or they can run DDL statements using the build in DBMS_SQL package.

Types of PL/SQL Blocks

There are two types of blocks in PL/SQL

1. Anonymous Block
2. Named Block

Anonymous Block

As the title suggests these anonymous blocks do not have any names as a result they cannot be stored in database and referenced later.

Named Block

On the other hand Named PL/SQL blocks are the one that have names and are used when creating subroutines such as procedures, functions and packages. These subroutines then can be stored in the database and referenced by their name later.

Both type of PL/SQL blocks are further divided into 3 different sections which are:

1. The Declaration Section
2. The Execution Section and
3. The Exception-handling Section

The Execution Section is the only mandatory section of block whereas Declaration and Exception Handling sections are optional.

Basic prototype of Anonymous PL/SQL Block

DECLARE
Declaration
BEGIN
Executable
Exception
Exception
***END**;*

Declaration Section

This is the first section of PL/SQL block which contains definition of PL/SQL identifiers such as variables, Constants, cursors and so on. You can say this is the place where all local variables used in the program are defined and documented.

Example 1

```
DECLARE
Var_first_name VARCHAR2(30);
Var_last_name   VARCHAR2(30);
Con_flag       CONSTANT       NUMBER:=0;
```

The above example shows declaration section of an anonymous block. It begins with keyword ***declare*** and contains two variables var_first_name and var_last_name and one constant con_flag. Notice that semicolon terminates each declaration.

Execution Section

This section contains executable statements that allow you to manipulate the variables that have been declared in the declaration section. The content of this section must be complete to allow the block to compile. By complete I mean complete set of instruction for the PL/SQL engine must be between BEGIN and END keyword.

The execution Section of any PL/SQL block always begins with the Keyword BEGIN and ends with the Keyword END.

This is the only mandatory section in PL/SQL block. This section supports all DML commands and SQL*PLUS built-in functions and using Native Dynamic SQL (NDS) or using DMBS_SQL built-in package it also supports DDL commands.

Example 2

```
BEGIN
SELECT first_name, last_name INTO var_first_name,
var_last_name
FROM employees WHERE employee_id =100;
DBMS_OUTPUT.PUT_LINE
('Employee Name '||var_first_name||' '||var_last_name);
END;
```

This is very simple program where I fetched the value of first name and last name column from employees table where employee id is 100 and stored it into the variable var_first_name and var_last_name which we declared in our first example.

Exception-Handling Section

This is the last section of PL/SQL block which is optional like the declaration block. This section contains statements that are executed when a runtime error occurs within the block.

Runtime error occurs while the program is running and cannot be detected by the PL/SQL compiler. When a runtime error occurs, controlled is pass to the exception handling section of the block the error is evaluated and specific exception is raised.

Example 3

```
EXCEPTION
WHEN NO_DATA_FOUND THEN      DBMS_OUTPUT.PUT_LINE ('No Employee Found with '||employee_id);
```

Cursors:
Cursor is a pointer to a memory area called context area. This context area is a memory region inside the Process Global Area or PGA assigned to hold the information about the processing of a SELECT statement or DML Statement such as INSERT, DELETE, UPDATE or MERGE

What is the Context Area?

Let's dig a little deeper and see what the context area is?

The context area is a special memory region inside the Process Global Area or PGA which helps oracle server in processing an SQL statement by holding the important information about that statement.

This information includes:

- Rows returned by a query.
- Number of rows processed by a query.
- A pointer to the parsed query in the shared pool.

Using cursor you can control the context area as it is a pointer to the same.

Advantages of Cursors

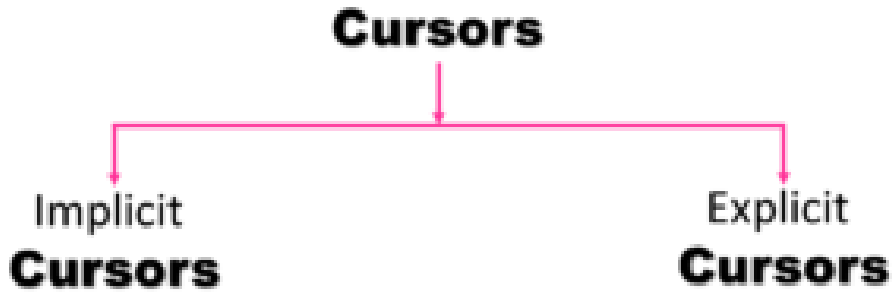
There are two main advantages of a cursor:

1. Cursor is names thus you can reference it in your program whenever you want.
2. Cursor allows you to fetch and process rows returned by a SELECT statement by a row at a time.

Types of cursors in oracle database:

There are two types of cursors in oracle database:

1. Implicit cursor
2. Explicit cursor



Implicit Cursors in Oracle Database

As the name suggests these are the cursors which are automatically created by the oracle server every time an SQL DML statement is executed. User cannot control the behavior of these cursors. Oracle server creates an implicit cursor in the background for any PL/SQL block which executes an SQL statement as long as an explicit cursor does not exist for that SQL statement.

Oracle server associates a cursor with every DML statement. Each of the Update & Delete statements has cursors which are responsible to identify those set of rows that are affected by the operation. Also the implicit cursor fulfills the need of a place for an Insert statement to receive the data that is to be inserted into the database.

Explicit Cursor in oracle database

In contrast to implicit cursors, we have explicit cursors. Explicit cursors are user defined cursors which means user has to create these cursors for any statement which returns more than one row of data. Unlike implicit cursor user has full control of explicit cursor. An explicit cursor can be generated only by naming the cursor in the declaration section of the PL/SQL block.

Advantages of Explicit Cursor:

1. Since Explicit cursors are user defined hence they give you more programmatic control on your program.

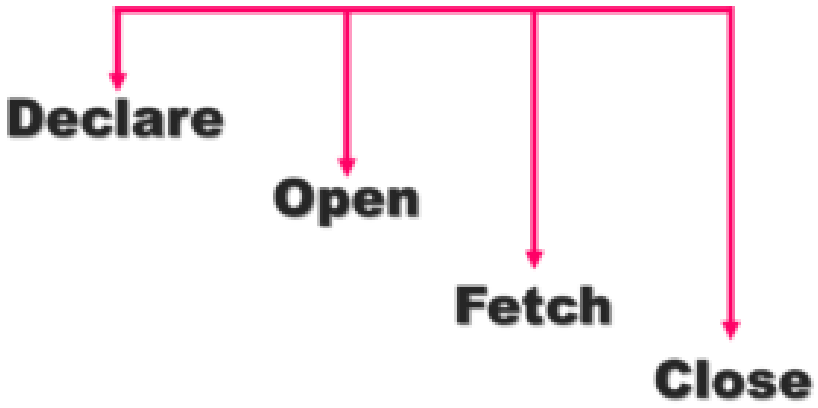
2. Explicit cursors are more efficient as compared to implicit cursors as in latters case it is hard to track data errors.

Steps for creating an Explicit Cursor

To create an explicit cursor you need to follow 4 steps. These 4 steps are:

- Declare
- Open
- Fetch
- Close

In case of implicit cursors oracle server performs all these steps automatically for you.



Declare: How To Declare a Database Cursor?

Declaring a cursor means initializing a cursor into memory. You define explicit cursor in declaration section of your PL/SQL block and associate it with the SELECT statement.

Syntax

```
CURSOR cursor_name IS select_statement;
```

Open: How to Open a Database Cursor?

Whenever oracle server comes across an 'Open Cursor' Statement the following four steps take place automatically.

1. All the variables including bind variables in the WHERE clause of a SELECT statement are examined.
2. Based on the values of the variables, the active set is determined, and the PL/SQL engine executes the query for that cursor. Variables are examined at cursor open time.
3. The PL/SQL engine identifies the active set of data.
4. The active set pointer sets to the first row.

Syntax

```
OPEN cursor_name;
```

Fetch: How to retrieve data from cursor?

The process of retrieving the data from the cursor is called fetching. Once the cursor is declared and opened then you can retrieve the data from it. Let's see how.

Syntax

```
FETCH cursor_name INTO PL/SQL variable;

Or

FETCH cursor_name INTO PL/SQL record;
```

What happens when you execute fetch command of a cursor?

1. The use of a FETCH command is to retrieve one row at a time from the active set. This is usually done inside a loop. The values of each row in the active set can then be stored in the corresponding variables or PL/SQL record one at a time, performing operations on each one successively.
2. After completion of each FETCH, the active set pointer is moved forward to the next row. Therefore, each FETCH returns successive rows of the active set, until the entire set is returned. The last FETCH does not assign values to the output variables as they still contain their previous values.

Close: How To Close a Database Cursor?

Once you are done working with your cursor it's advisable to close it. As soon as the server comes across the closing statement of a cursor it will relinquish all the resources associated with it.

Syntax

```
CLOSE cursor_name;
```

Basic Programming Structure of the Cursor

Here is the basic programming structure of the cursor in oracle database.

```
DECLARE

        CURSOR cursor_name IS select_statement;

BEGIN

        OPEN cursor_name;

        FETCH cursor_name INTO PL/SQL variable [PL/SQL record];

CLOSE cursor_name;

END;

/
```

Cursor Attributes

Oracle provides four attributes which work in correlation with cursors. These attributes are:

- %FOUND
- %NOTFOUND
- %ISOPEN
- %ROWCOUNT

First three attributes 'Found', 'NotFound' and 'IsOpen' are Boolean attributes whereas the last one 'RowCount' is a numeric attribute.

Let's quickly take a look at all these attributes.

%Found

Cursor attribute 'Found' is a Boolean attribute which returns TRUE if the previous FETCH command returned a row otherwise it returns FALSE.

%NotFound

'Not Found' cursor attribute is also a Boolean attribute which returns TRUE only when previous FETCH command of the cursor did not return a row otherwise this attribute will return FALSE.

%IsOpen

Again ‘Is Open’ Cursor attribute is a Boolean attribute which you can use to check whether your cursor is open or not. It returns TRUE if the cursor is open otherwise it returns FALSE.

%RowCount

Row count cursor attribute is a numeric attribute which means it returns a numeric value as a result and that value will be the number of records fetched from a cursor at that point in time.

So, this is the detailed blog on cursors in oracle database. Hope it will help you with your exams as well as with your job interview. Do make sure to share this article on your social media or

Triggers:

Triggers are named PL/SQL blocks which are stored in the database. We can also say that they are specialized stored programs which execute implicitly when a triggering event occurs. This means we cannot call and execute them directly instead they only get triggered by events in the database.

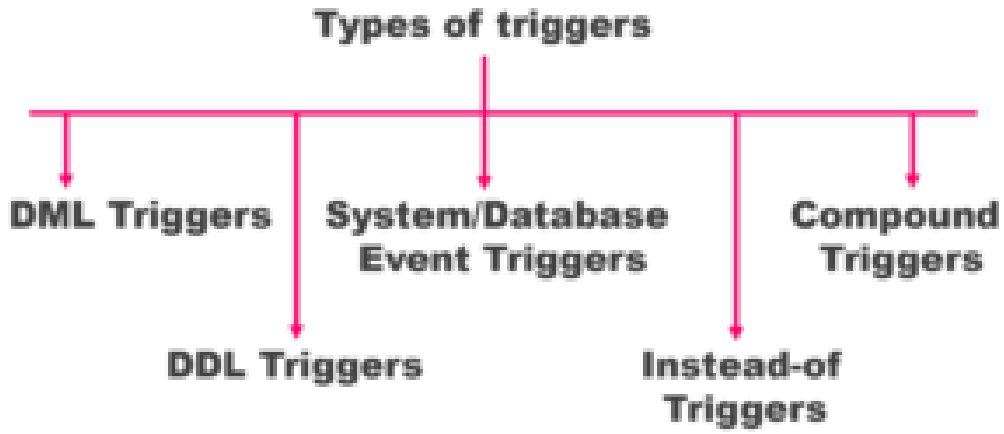
Events Which Fires the Database Triggers

These events can be anything such as

1. **A DML Statement** – An Update, Insert or Delete statement executing on any table of your database. You can program your trigger to execute either BEFORE or AFTER executing your DML statement. For example, you can create a trigger which will get fired *Before* the Update. Similarly, you can create a trigger which will get triggered after the execution of your INSERT DML statement.
2. **A DDL Statement** – Next type of triggering statement can be a DDL Statement such as CREATE or ALTER. These triggers can also be executed either BEFORE or AFTER the execution of your DDL statement. These triggers are generally used by DBAs for auditing purposes. And they really come in handy when you want to keep an eye on the various changes on your schema. For instance, who created the object or which user. Just like some cool spy tricks.
3. **A system event**. – Yes, you can create a trigger on a system event. And by a system event, I mean shut down or startup of your database.
4. **A User Events** – Another type of triggering event can be User Events such as log off or log on onto your database. You can create a trigger which will either execute before or after the event. Furthermore, it will record the information such as time of event occur, the username who created it.

Types of Database Triggers

There are 5 types of triggers in the Oracle database. 3 of them are based on the triggering event which are discussed in the previous section.



1. **Data Manipulation Language Triggers or DML triggers**

As the name suggests these are the triggers which depend on DML statements such as Update, Insert or Delete. They get fired either before or after them. Using DML trigger you can control the behavior of your DML statements. You can audit, check, replace or save values before they are changed. Automatic Increment of your Numeric primary key is one of the most frequent tasks of these types of triggers.

2. **Data Definition Language Triggers or DDL triggers.**

Again as the name suggests these are the type of triggers which are created over DDL statements such as CREATE or ALTER. They get fired either before or after the execution of your DDL statements. Using this type of trigger you can monitor the behavior and force rules on your DDL statements.

3. **System or Database Event triggers.**

Third type of triggers is system or database triggers. These are the type of triggers which come into action when some system event occurs such as database log on or log off. You can use these triggers for auditing purposes. For example, keeping an eye on information of system access like say who connects with your database and when. Most of the time System or Database Event triggers work as Swiss Knife for DBAs and help them in increasing the security of the data.

4. **Instead-of Trigger**

This is a type of trigger which enables you to stop and redirect the performance of a DML statement. Often this type of trigger helps you in managing the way you write to non-updatable views. You can also see the application of business rules by INSTEAD OF triggers where they insert, update or delete rows directly in tables that are defining updatable views. Alternatively, sometimes the INSTEAD OF triggers are also seen inserting, updating or deleting rows in designated tables that are otherwise unrelated to the view.

5. **Compound triggers**

These are multi-tasking triggers that act as both statement as well as row-level triggers when the data is inserted, updated or deleted from a table. You can capture information at four timing points using this trigger:

- before the firing statement;

- prior to the change of each row from the firing statement;
- post each row changes from the firing statement;
- after the firing statement.

All these types of triggers can be used to audit, check, save and replace the values. Even before they are changed right when there is a need to take action at the statement as well as at row event levels.

The Syntax Of Database Trigger

```
CREATE [OR REPLACE] TRIGGER Ttrigger_name

{BEFORE|AFTER} Triggering_event ON table_name

[FOR EACH ROW]

[FOLLOWS another_trigger_name]

[ENABLE/DISABLE]

[WHEN condition]

DECLARE

  declaration statements

BEGIN

  executable statements

EXCEPTION

  exception-handling statements

END;
```

DDL Triggers:

DDL triggers are the triggers which are created over DDL statements such as CREATE, DROP or ALTER. Using this type of trigger you can monitor the behavior and force rules on your DDL statements.

In order to proceed ahead and start writing the trigger first we need a table in which we can journal the auditing information created by the trigger.

```
CREATE TABLE schema_audit

 (

   ddl_date       DATE,

   ddl_user       VARCHAR2(15),

   object_created VARCHAR2(15),

   object_name    VARCHAR2(15),

   ddl_operation  VARCHAR2(15)

  );
```

In case of schema/user auditing using DDL trigger creates this table in the same schema which you are auditing and in case of Database auditing using DDL trigger create this table in sys or system schema (sys or system both schemas can be used to perform database auditing).

DDL Trigger for Schema Auditing

First you need to log on to the database using the schema which you want to audit. For example suppose you want to create the DDL trigger to audit the HR schema then log on to your database using the HR schema.

Then Write, Execute and Compile the below trigger.

```
CREATE OR REPLACE TRIGGER hr_audit_tr

AFTER DDL ON SCHEMA

BEGIN

   INSERT INTO schema_audit VALUES (

sysdate,

sys_context('USERENV','CURRENT_USER'),

ora_dict_obj_type,

ora_dict_obj_name,

ora_sysevent);
```

```
END;
```

/

If you will notice carefully the second line of the code (“AFTER DDL ON SCHEMA”) indicates that this trigger will work on the schema in which it is created. On successful compilation this trigger will insert the respective information such as the date when the DDL is executed, username who executed the DDL, type of database object created, name of the object given by the user at the time of its creation and the type of DDL into the table which we created earlier.

DDL Trigger for Database Auditing.

Similar to the schema auditing with some minor changes in the above trigger you can audit your database too. But for that first you need to logon to the database using either SYS user or SYSTEM user.

After doing that you have to create the above shown table under the same user so that your trigger can dump the auditing data without any read and write errors.

```
CREATE OR REPLACE TRIGGER db_audit_tr

AFTER DDL ON DATABASE

BEGIN

   INSERT INTO schema_audit VALUES (

sysdate,

sys_context('USERENV','CURRENT_USER'),

ora_dict_obj_type,

ora_dict_obj_name,

ora_sysevent);

END;

/
```

If you notice the second line of this code carefully then you will find that we have replaced the **keyword Schema** with the **keyword Database** which indicates that this trigger will work for the whole database and will perform the underlying work.

To create a trigger on database we require **ADMINISTER DATABASE TRIGGER** system privilege. All the administrative users such as sys or system already has these privileges by default that is the reason we created this database auditing DDL trigger using these users. Though you can create the same trigger with any user by granting the same privileges to them but that is not advisable because of your database security reasons.

Data Manipulation Language (DML) Triggers.

As the name suggests these are the triggers which execute on DML events or say depend on DML statements such as Update, Insert or Delete. Using DML trigger you can control the behavior of your DML statements.

Since the theory has already been discussed in the previous tutorial hence I won't bore you further. You can refer to the previous tutorial "Introduction to Triggers" anytime.

Examples

In order to demonstrate the creation process of DML trigger we need to first create a table.

```
CREATE TABLE superheroes (

 sh_name VARCHAR2 (15)

);
```

I have created this table with the name SUPERHEROES which has only one column sh_name with data type varchar2 and data width 15. Now I will write a DML trigger which will work on this table.

So the table is created. Now let's do some examples which will help you in understanding the concepts more clearly.

```
SET SERVEROUTPUT ON;
```

Example 1. Before Insert Trigger

In the first example we will see how to create a trigger over Insert DML. This trigger will print a user defined message every time a user inserts a new row in the superheroes table.

```
CREATE OR REPLACE TRIGGER bi_Superheroes

BEFORE INSERT ON superheroes

FOR EACH ROW

ENABLE

DECLARE

  v_user  VARCHAR2 (15);

BEGIN

 SELECT user INTO v_user FROM dual;

DBMS_OUTPUT.PUT_LINE('You Just Inserted a Row Mr.'|| v_user);
```

```
END;

/
```

On successfully compiling, this trigger will show you a string along with the user name who performed the “Insert” DML on superheroes table. Thus you check this trigger by Inserting a row in Superheroes table.

```
INSERT INTO superheroes VALUES ('Ironman');
```

Example 2: Before Update Trigger.

Update Trigger is the one which will execute either before or after Update DML. The creation process of an Update trigger is the same as that of Insert Trigger. You just have to replace Keyword INSERT with UPDATE in the 2nd Line of the above example.

```
CREATE OR REPLACE TRIGGER bu_Superheroes

BEFORE UPDATE ON superheroes

FOR EACH ROW

ENABLE

DECLARE

  v_user  VARCHAR2 (15);

BEGIN

 SELECT user INTO v_user FROM dual;

 DBMS_OUTPUT.PUT_LINE('You Just Updated a Row Mr.'|| v_user);

END;

/
```

On successfully compiling, this trigger will print a user defined string with the username of the user who updated the row. You can check this trigger by writing an update DML on the superheroes table.

```
UPDATE superheroes SET SH_NAME = ‘Superman’ WHERE SH_NAME='Ironman';
```

Example 3: Before Delete Trigger

Similar to Insert and Update DML you can write a trigger over Delete DML. This trigger will execute either before or after a user deletes a row from the underlying table.

```
CREATE OR REPLACE TRIGGER bu_Superheroes
```

```
BEFORE DELETE ON superheroes

FOR EACH ROW

ENABLE

DECLARE

 v_user  VARCHAR2 (15);

BEGIN

 SELECT user INTO v_user FROM dual;

 DBMS_OUTPUT.PUT_LINE('You Just Deleted a Row Mr.'|| v_user);

END;

/
```

You can check the working of this trigger by executing a DELETE DML on underlying table which is superheroes.

```
DELETE FROM superheroes WHERE sh_name = 'Superman';
```

Above three examples showed you 3 different triggers for 3 different DML events on one table. Don't you think that if we can cover all these 3 events in just 1 trigger then it will be a great relief? If you think so then my dear friend I have some good news for you. Let me show you how we can achieve this feat.

INSERT, UPDATE, DELETE All in One DML Trigger Using IF-THEN-ELSIF

```
CREATE OR REPLACE TRIGGER tr_superheroes

BEFORE INSERT OR DELETE OR UPDATE ON superheroes

FOR EACH ROW

ENABLE

DECLARE

 v_user  VARCHAR2(15);

BEGIN


 SELECT

   user INTO v_user FROM dual;
```

```
IF INSERTING THEN
  DBMS_OUTPUT.PUT_LINE('one line inserted by '||v_user);
ELSIF DELETING THEN
  DBMS_OUTPUT.PUT_LINE('one line Deleted by '||v_user);
ELSIF UPDATING THEN
  DBMS_OUTPUT.PUT_LINE('one line Updated by '||v_user);
END IF;
END;
/
```

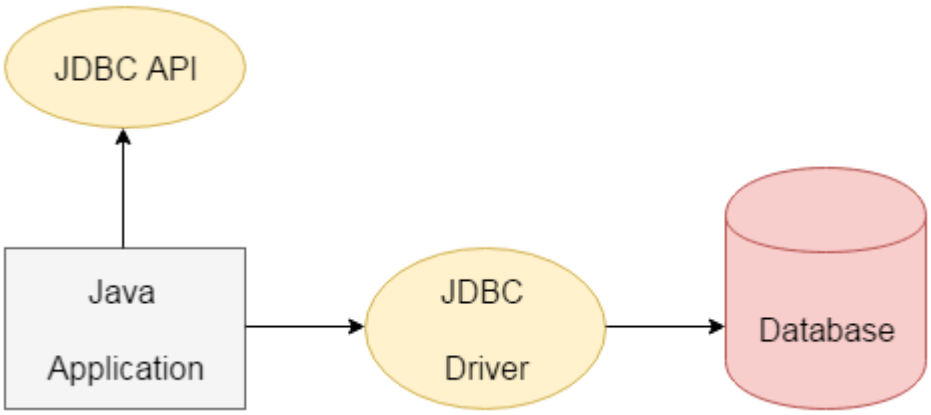
Java JDBC Tutorial

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver

We have discussed the above four drivers in the next chapter.

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



The current version of JDBC is 4.3. It is the stable release since 21st September, 2017. It is based on the X/Open SQL Call Level Interface. The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

A list of popular *classes* of JDBC API are given below:

- DriverManager class
- Blob class
- Clob class
- Types class

Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database

2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

What is API

API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.

JDBC Driver

1. JDBC Drivers

1. JDBC-ODBC bridge driver
2. Native-API driver
3. Network Protocol driver
4. Thin driver

JDBC Driver is a software component that enables java application to interact with the database. T
JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because thin driver.

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the v of your database instead of the JDBC-ODBC Bridge.

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

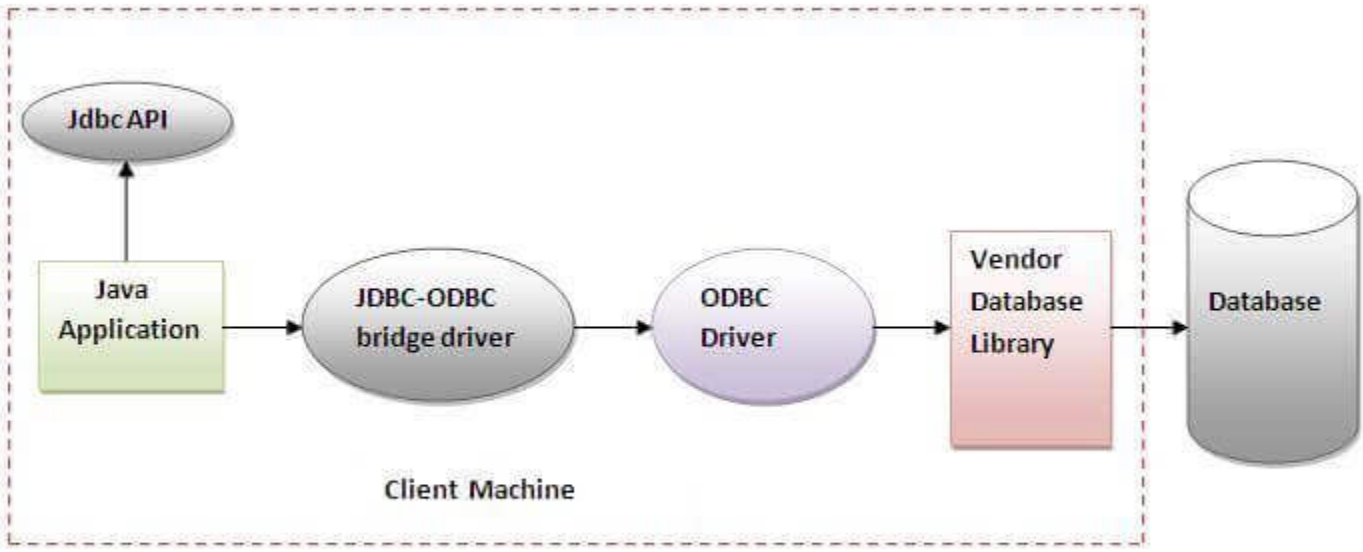


Figure- JDBC-ODBC Bridge Driver

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

2) Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

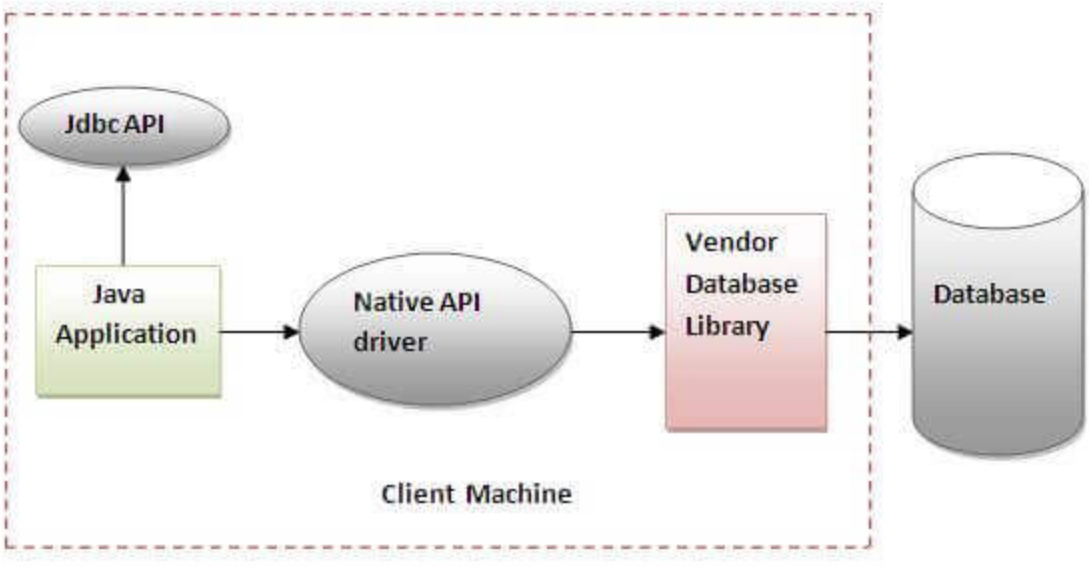


Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

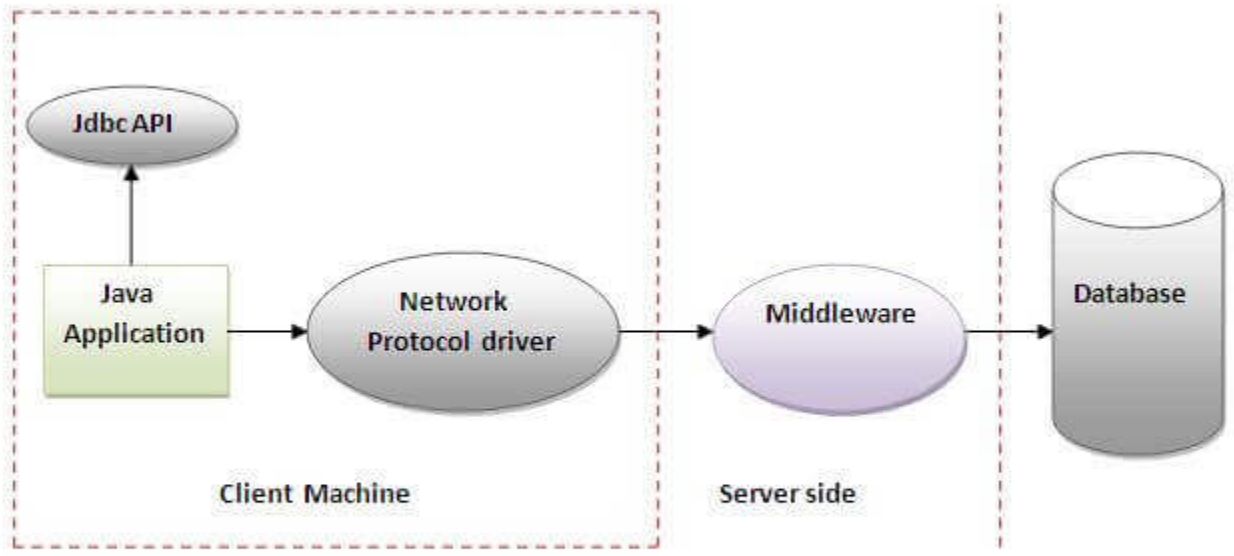


Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

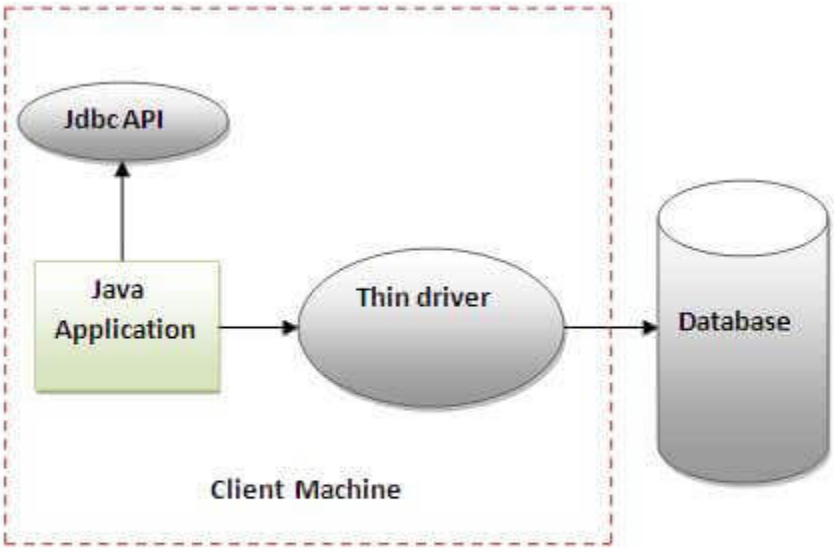


Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

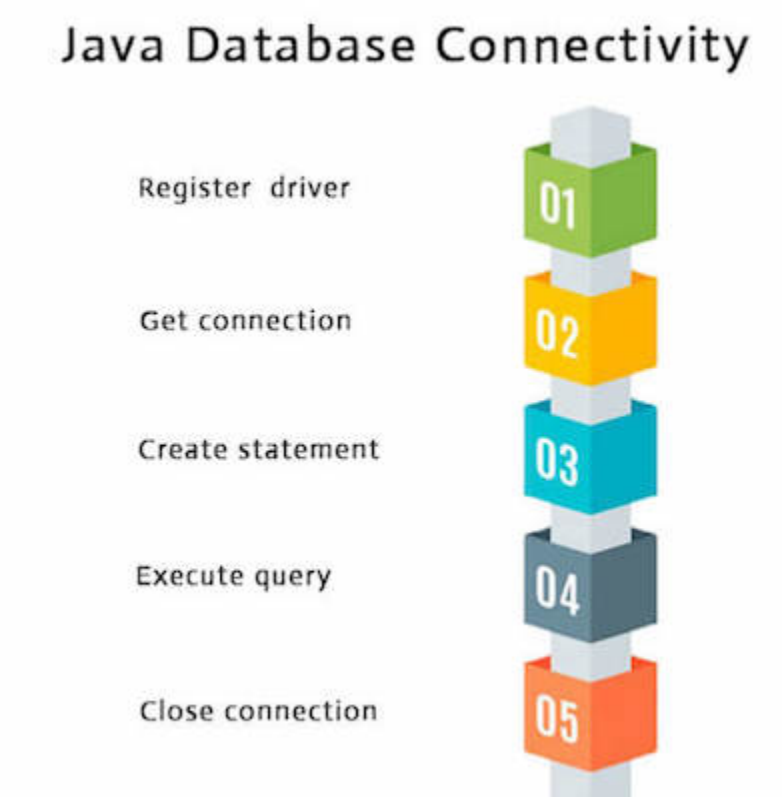
- Drivers depend on the Database.

Java Database Connectivity with 5 Steps

1. 5 Steps to connect to the database in java
1. Register the driver class
2. Create the connection object
3. Create the Statement object
4. Execute the query
5. Close the connection object

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection



1) Register the driver class

The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of forName() method

1. **public static void** forName(String className)**throws** ClassNotFoundException

Example to register the OracleDriver class

Here, Java program is loading oracle driver to esteblish database connection.

```
1. Class.forName("oracle.jdbc.driver.OracleDriver");
```

2) Create the connection object

The **getConnection()** method of DriverManager class is used to establish connection with the database.

Syntax of getConnection() method

```
1. 1) public static Connection getConnection(String url)throws SQLException
2. 2) public static Connection getConnection(String url,String name,String password)
3. throws SQLException
```

Example to establish connection with the Oracle database

```
1. Connection con=DriverManager.getConnection(
2. "jdbc:oracle:thin:@localhost:1521:xe","system","password");
```

3) Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax of createStatement() method

```
1. public Statement createStatement()throws SQLException
```

Example to create the statement object

```
1. Statement stmt=con.createStatement();
```

4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax of executeQuery() method

```
1. public ResultSet executeQuery(String sql)throws SQLException
```

Example to execute query

```
1. ResultSet rs=stmt.executeQuery("select * from emp");
2.
3. while(rs.next()){
4. System.out.println(rs.getInt(1)+" "+rs.getString(2));
5. }
```

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method

```
1. public void close()throws SQLException
```

Example to close connection

```
1. con.close();
```

Java Database Connectivity with Oracle

To connect java application with the oracle database, we need to follow 5 following steps. In this example, we are using Oracle 10g as the database. So we need to know following information for the oracle database:

1. **Driver class:** The driver class for the oracle database is **oracle.jdbc.driver.OracleDriver**.
2. **Connection URL:** The connection URL for the oracle10G database is **jdbc:oracle:thin:@localhost:1521:xe** where jdbc is the API, oracle is the database, thin is the driver, localhost is the server name on which oracle is running, we may also use IP address, 1521 is the port number and XE is the Oracle service name. You may get all these information from the tnsnames.ora file.
3. **Username:** The default username for the oracle database is **system**.
4. **Password:** It is the password given by the user at the time of installing the oracle database.

Create a Table

Before establishing connection, let's first create a table in oracle database. Following is the SQL query to create a table.

```
1. create table emp(id number(10),name varchar2(40),age number(3));
```

Example to Connect Java Application with Oracle database

In this example, we are connecting to an Oracle database and getting data from **emp** table. Here, **system** and **oracle** are the username and password of the Oracle database.

```
1. import java.sql.*;
2. class OracleCon{
3. public static void main(String args[]){
4. try{
5. //step1 load the driver class
6. Class.forName("oracle.jdbc.driver.OracleDriver");
7.
8. //step2 create  the connection object
9. Connection con=DriverManager.getConnection(
10. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
11.
12. //step3 create the statement object
13. Statement stmt=con.createStatement();
14.
15. //step4 execute query
16. ResultSet rs=stmt.executeQuery("select * from emp");
17. while(rs.next())
18. System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
19.
20. //step5 close the connection object
21. con.close();
22.
23. }catch(Exception e){ System.out.println(e);}
24.
25. }
26. }
```

The above example will fetch all the records of emp table.

To connect java application with the Oracle database ojdbc14.jar file is required to be loaded.

download the jar file ojdbc14.jar

Two ways to load the jar file:

1. paste the ojdbc14.jar file in jre/lib/ext folder
2. set classpath

1) paste the ojdbc14.jar file in JRE/lib/ext folder:

Firstly, search the ojdbc14.jar file then go to JRE/lib/ext folder and paste the jar file here.

2) set classpath:

There are two ways to set the classpath:

- temporary
- permanent

How to set the temporary classpath:

Firstly, search the ojdbc14.jar file then open command prompt and write:

1. C:>set classpath=c:\folder\ojdbc14.jar;.;

How to set the permanent classpath:

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to ojdbc14.jar by appending ojdbc14.jar;.; as C:\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar;.;