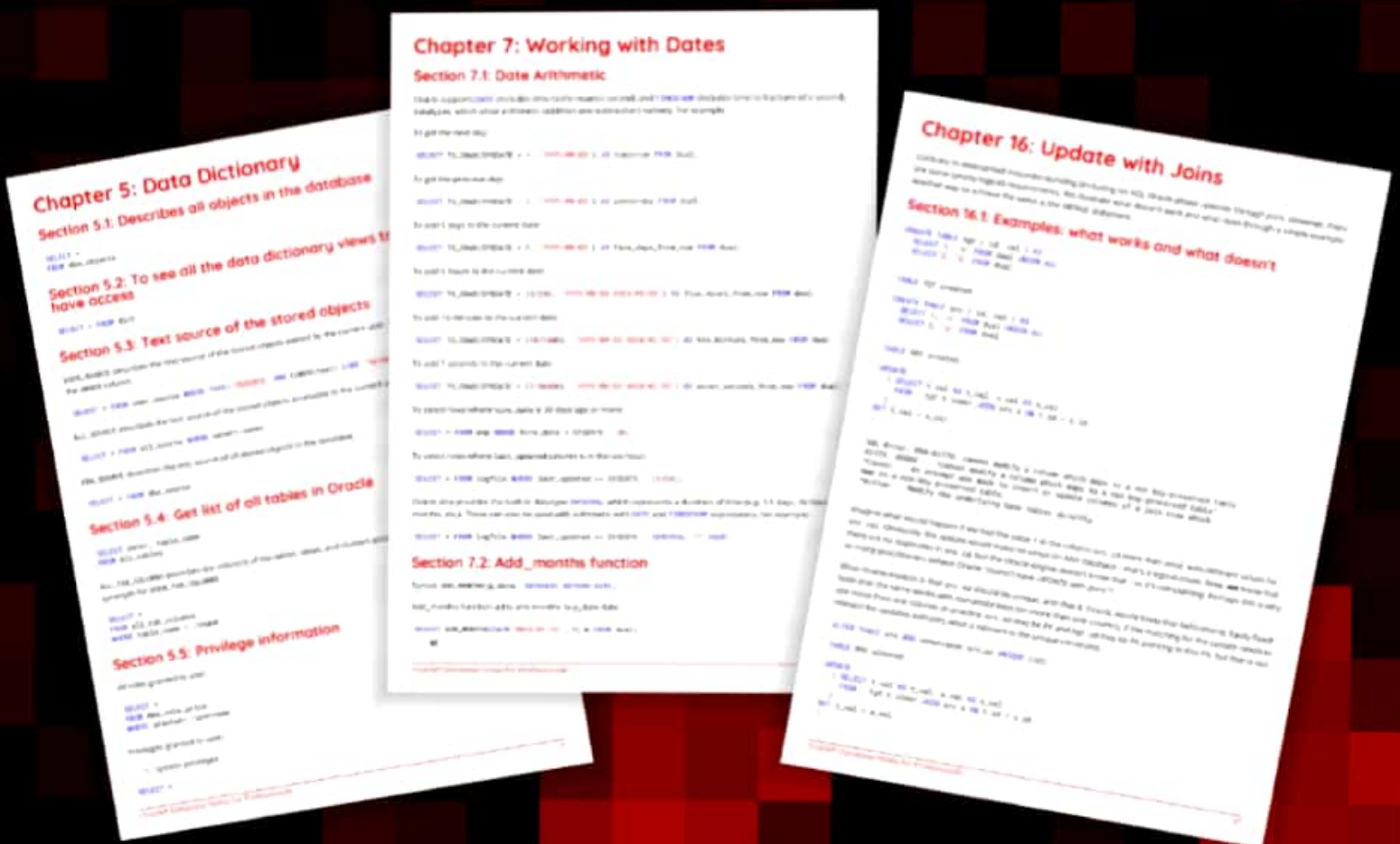
Oracle® Database

Notes for Professionals



Contents

Chapter 1: Getting started with Oracle Database

Version	Release Date
Version 1 (unreleased)	1978-01-01
Oracle V2	1979-01-01
Oracle Version 3	1983-01-01
Oracle Version 4	1984-01-01
Oracle Version 5	1985-01-01
Oracle Version 6	1988-01-01
Oracle7	1992-01-01
Oracle8	1997-07-01
Oracle8i	1999-02-01
Oracle9i	2001-06-01
Oracle 10g	2003-01-01
Oracle 11g	2007-01-01
Oracle 12c	2013-01-01

Section 1.1: Hello World

```
SELECT 'Hello world!' FROM dual;
```

In Oracle's flavor of SQL, "dual is just a convienence table". It was originally intended to double rows via a JOIN, but now contains one row with a `DUMMY` value of 'X'.

Section 1.2: SQL Query

List employees earning more than $50000 born this century. List their name, date of birth and salary, sorted alphabetically by name.

```
SELECT employee_name, date_of_birth, salary
FROM   employees
WHERE  salary > 50000
   AND date_of_birth >= DATE '2000-01-01'
ORDER BY employee_name;
```

Show the number of employees in each department with at least 5 employees. List the largest departments first.

```
SELECT department_id, COUNT(*)
FROM   employees
GROUP BY department_id
HAVING COUNT(*) >= 5
ORDER BY COUNT(*) DESC;
```

Section 1.3: Hello world! from table

Create a simple table

```
CREATE TABLE MY_table (
   what VARCHAR2(10),
   who VARCHAR2(10),
```

```
   mark VARCHAR2(10)
);
```

Insert values (you can omit target columns if you provide values for all columns)

```
INSERT INTO my_table (what, who, mark) VALUES ('Hello', 'world', '!' );
INSERT INTO my_table VALUES ('Bye bye', 'ponies', '?' );
INSERT INTO my_table (what) VALUES('Hey');
```

Remember to commit, because Oracle uses *transactions*

```
COMMIT;
```

Select your data:

```
SELECT what, who, mark FROM my_table WHERE what='Hello';
```

Section 1.4: Hello World from PL/SQL

```
/* PL/SQL is a core Oracle Database technology, allowing you to build clean, secure,
   optimized APIs to SQL and business logic. */

SET serveroutput ON

BEGIN
   DBMS_OUTPUT.PUT_LINE ('Hello World!');
END;
```

Chapter 2: Getting started with PL/SQL

Section 2.1: Hello World

```
SET serveroutput ON

DECLARE
   message CONSTANT VARCHAR2(32767):= 'Hello, World!';
BEGIN
   DBMS_OUTPUT.put_line(message);
END;
/
```

Command `SET serveroutput ON` is required in SQL*Plus and SQL Developer clients to enable the output of `DBMS_OUTPUT`. Without the command nothing is displayed.

The `END;` line signals the end of the anonymous PL/SQL block. To run the code from SQL command line, you may need to type `/` at the beginning of the first blank line after the last line of the code. When the above code is executed at SQL prompt, it produces the following result:

```
Hello, World!

PL/SQL procedure successfully completed.
```

Section 2.2: Definition of PL/SQL

PL/SQL (Procedural Language/Structured Query Language) is Oracle Corporation's procedural extension for SQL and the Oracle relational database. PL/SQL is available in Oracle Database (since version 7), TimesTen in-memory database (since version 11.2.1), and IBM DB2 (since version 9.7).

The basic unit in PL/SQL is called a block, which is made up of three parts: a declarative part, an executable part, and an exception-building part.

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <EXCEPTION handling>
END;
```

Declarations - This section starts with the keyword DECLARE. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.

Executable Commands - This section is enclosed between the keywords BEGIN and END and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed.

Exception Handling - This section starts with the keyword EXCEPTION. This section is again optional and contains exception(s) that handle errors in the program.

Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using BEGIN and END.

In anonymous block, only executable part of block is required, other parts are not nessesary. Below is example of simple anonymous code, which does not do anything but perform without error reporting.

```
BEGIN
    NULL;
END;
/
```

Missing excecutable instruction leads to an error, becouse PL/SQL does not support empty blocks. For example, excecution of code below leads to an error:

```
BEGIN
END;
/
```

Application will raise error:

```
END;
*
ERROR AT line 2:
ORA-06550: line 2, column 1:
PLS-00103: Encountered the symbol "END" WHEN expecting one OF the following:
( BEGIN CASE DECLARE EXIT FOR GOTO IF LOOP MOD NULL PRAGMA
RAISE RETURN SELECT UPDATE WHILE WITH <an identifier>
<a double-quoted delimited-identifier> <a bind variable> <<
continue CLOSE CURRENT DELETE FETCH LOCK INSERT OPEN ROLLBACK
SAVEPOINT SET SQL EXECUTE COMMIT FORALL MERGE pipe purge
```

Symbol " * " in line below keyword "END;" means, that the block which ends with this block is empty or bad constructed. Every execution block needs instructions to do, even if it does nothing, like in our example.

Section 2.3: Difference between %TYPE and %ROWTYPE

`%TYPE`: Used to declare a field with the same type as that of a specified table's column.

```
DECLARE
        vEmployeeName   Employee.Name%TYPE;
BEGIN
        SELECT Name
        INTO   vEmployeeName
        FROM   Employee
        WHERE  ROWNUM = 1;

        DBMS_OUTPUT.PUT_LINE(vEmployeeName);
END;
/
```

%ROWTYPE: Used to declare a record with the same types as found in the specified table, view or cursor (= multiple columns).

```
DECLARE
        rEmployee     Employee%ROWTYPE;
BEGIN
        rEmployee.Name := 'Matt';
        rEmployee.Age := 31;

        DBMS_OUTPUT.PUT_LINE(rEmployee.Name);
```

```
        DBMS_OUTPUT.PUT_LINE(rEmployee.Age);
END;
/
```

Section 2.4: Create or replace a view

In this example we are going to create a view.
A view is mostly used as a simple way of fetching data from multiple tables.

Example 1:
View with a select on one table.

```
CREATE OR REPLACE VIEW LessonView AS
       SELECT     L.*
       FROM       Lesson L;
```

Example 2:
View with a select on multiple tables.

```
CREATE OR REPLACE VIEW ClassRoomLessonView AS
       SELECT     C.Id,
                  C.Name,
                  L.Subject,
                  L.Teacher
       FROM       ClassRoom C,
                  Lesson L
       WHERE      C.Id = L.ClassRoomId;
```

To call this views in a query you can use a select statement.

```
SELECT * FROM LessonView;
SELECT * FROM ClassRoomLessonView;
```

Section 2.5: Create a table

Below we are going to create a table with 3 columns.
The column `Id` must be filled is, so we define it `NOT NULL`.
On the column `Contract` we also add a check so that the only value allowed is 'Y' or 'N'. If an insert in done and this column is not specified during the insert then default a 'N' is inserted.

```
CREATE TABLE Employee (
        Id              NUMBER NOT NULL,
        Name            VARCHAR2(60),
        Contract        CHAR DEFAULT 'N' NOT NULL,
        ---
        CONSTRAINT p_Id PRIMARY KEY(Id),
        CONSTRAINT c_Contract CHECK (Contract IN('Y','N'))
);
```

Section 2.6: About PL/SQL

PL/SQL stands for Procedural Language extensions to SQL. PL/SQL is available only as an "enabling technology" within other software products; it does not exist as a standalone language. You can use PL/SQL in the Oracle relational database, in the Oracle Server, and in client-side application development tools, such as Oracle Forms. Here are some of the ways you might use PL/SQL:

1. To build stored procedures. .
2. To create database triggers.
3. To implement client-side logic in your Oracle Forms application.
4. To link a World Wide Web home page to an Oracle database.

Chapter 3: Anonymous PL/SQL Block

Section 3.1: An example of an anonymous block

```
DECLARE
    -- declare a variable
    message VARCHAR2(20);
BEGIN
  -- assign value to variable
  message := 'HELLO WORLD';

  -- print message to screen
  DBMS_OUTPUT.PUT_LINE(message);
END;
/
```

Chapter 4: PL/SQL procedure

PL/SQL procedure is a group of SQL statements stored on the server for reuse. It increases the performance because the SQL statements do not have to be recompiled every time it is executed.

Stored procedures are useful when same code is required by multiple applications. Having stored procedures eliminates redundancy, and introduces simplicity to the code. When data transfer is required between the client and server, procedures can reduce communication cost in certain situations.

Section 4.1: Syntax

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] TYPE [, ...])]
{IS | AS}
  < declarations >
BEGIN
  < procedure_body >
EXCEPTION                          -- Exception-handling part begins
  <EXCEPTION handling goes here >
   WHEN exception1 THEN
       exception1-handling-statements
END procedure_name;
```

- procedure-name specifies the name of the procedure.
- [OR REPLACE] option allows modifying an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure. If no mode is specified, parameter is assumed to be of IN mode.
- In the declaration section we can declare variables which will be used in the body part.
- procedure-body contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.
- exception section will handle the exceptions from the procedure. This section is optional.

Section 4.2: Hello World

The following simple procedure displays the text "Hello World" in a client that supports DBMS_OUTPUT.

```
CREATE OR REPLACE PROCEDURE helloworld
AS
BEGIN
   DBMS_OUTPUT.put_line('Hello World!');
END;
/
```

You need to execute this at the SQL prompt to create the procedure in the database, or you can run the query below to get the same result:

```
SELECT 'Hello World!' FROM dual;
```

Section 4.3: In/Out Parameters

PL/SQL uses IN, OUT, IN OUT keywords to define what can happen to a passed parameter.

IN specifies that the parameter is read only and the value cannot be changed by the procedure.

OUT specifies the parameter is write only and a procedure can assign a value to it, but not reference the value.

IN OUT specifies the parameter is available for reference and modification.

```
PROCEDURE procedureName(x IN INT, strVar IN VARCHAR2, ans OUT VARCHAR2)
...
...
END procedureName;


procedureName(firstvar, secondvar, thirdvar);
```

The variables passed in the above example need to be typed as they are defined in the procedure parameter section.

Chapter 5: Data Dictionary

Section 5.1: Describes all objects in the database

```
SELECT *
FROM dba_objects
```

Section 5.2: To see all the data dictionary views to which you have access

```
SELECT * FROM dict
```

Section 5.3: Text source of the stored objects

USER_SOURCE describes the text source of the stored objects owned by the current user. This view does not display the OWNER column.

```
SELECT * FROM user_source WHERE TYPE='TRIGGER' AND LOWER(text) LIKE '%order%'
```

ALL_SOURCE describes the text source of the stored objects accessible to the current user.

```
SELECT * FROM all_source WHERE owner=:owner
```

DBA_SOURCE describes the text source of all stored objects in the database.

```
SELECT * FROM dba_source
```

Section 5.4: Get list of all tables in Oracle

```
SELECT owner, table_name
FROM all_tables
```

ALL_TAB_COLUMNS describes the columns of the tables, views, and clusters accessible to the current user. COLS is a synonym for USER_TAB_COLUMNS.

```
SELECT *
FROM all_tab_columns
WHERE table_name = :tname
```

Section 5.5: Privilege information

All roles granted to user.

```
SELECT *
FROM dba_role_privs
WHERE grantee= :username
```

Privileges granted to user:

1. system privileges

```
SELECT *
```

```
FROM dba_sys_privs
WHERE grantee = :username
```

2. object grants

```
SELECT *
FROM dba_tab_privs
WHERE grantee = :username
```

Permissions granted to roles.

Roles granted to other roles.

```
SELECT *
FROM role_role_privs
WHERE role IN (SELECT granted_role FROM dba_role_privs WHERE grantee= :username)
```

1. system privileges

```
SELECT *
FROM role_sys_privs
WHERE role IN (SELECT granted_role FROM dba_role_privs WHERE grantee= :username)
```

2. object grants

```
SELECT *
FROM role_tab_privs
WHERE role IN (SELECT granted_role FROM dba_role_privs WHERE grantee= :username)
```

Section 5.6: Oracle version

```
SELECT *
FROM v$version
```

Chapter 6: Dates

Section 6.1: Date Arithmetic - Difference between Dates in Days, Hours, Minutes and/or Seconds

In oracle, the difference (in days and/or fractions thereof) between two `DATE`s can be found using subtraction:

```
SELECT DATE '2016-03-23' - DATE '2015-12-25' AS difference FROM DUAL;
```

Outputs the number of days between the two dates:

```
DIFFERENCE
----------
89
```

And:

```
SELECT TO_DATE( '2016-01-02 01:01:12', 'YYYY-MM-DD HH24:MI:SS' )
         - TO_DATE( '2016-01-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS' )
         AS difference
FROM   DUAL
```

Outputs the fraction of days between two dates:

```
DIFFERENCE
----------
1.0425
```

The difference in hours, minutes or seconds can be found by multiplying this number by 24, `24*60` or `24*60*60` respectively.

The previous example can be changed to get the days, hours, minutes and seconds between two dates using:

```
SELECT TRUNC( difference                       ) AS days,
       TRUNC( MOD( difference * 24,       24 ) ) AS hours,
       TRUNC( MOD( difference * 24*60,    60 ) ) AS minutes,
       TRUNC( MOD( difference * 24*60*60, 60 ) ) AS seconds
FROM   (
  SELECT TO_DATE( '2016-01-02 01:01:12', 'YYYY-MM-DD HH24:MI:SS' )
         - TO_DATE( '2016-01-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS' )
         AS difference
  FROM   DUAL

);
```

(Note: `TRUNC()` is used rather than `FLOOR()` to correctly handle negative differences.)

Outputs:

```
DAYS HOURS MINUTES SECONDS
---- ----- ------- -------
1    1     1       12
```

The previous example can also be solved by converting the numeric difference to an interval using NUMTODSINTERVAL():

```
SELECT EXTRACT( DAY    FROM difference ) AS days,
       EXTRACT( HOUR   FROM difference ) AS hours,
       EXTRACT( MINUTE FROM difference ) AS minutes,
       EXTRACT( SECOND FROM difference ) AS seconds
FROM   (
  SELECT NUMTODSINTERVAL(
           TO_DATE( '2016-01-02 01:01:12', 'YYYY-MM-DD HH24:MI:SS' )
             - TO_DATE( '2016-01-01 00:00:00', 'YYYY-MM-DD HH24:MI:SS' ),
           'DAY'
         ) AS difference
  FROM   DUAL
);
```

Section 6.2: Setting the Default Date Format Model

When Oracle implicitly converts from a DATE to a string or vice-versa (or when TO_CHAR() or TO_DATE() are explicitly called without a format model) the NLS_DATE_FORMAT session parameter will be used as the format model in the conversion. If the literal does not match the format model then an exception will be raised.

You can review this parameter using:

```
SELECT VALUE FROM NLS_SESSION_PARAMETERS WHERE PARAMETER = 'NLS_DATE_FORMAT';
```

You can set this value within your current session using:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI:SS';
```

(Note: this does not change the value for any other users.)

If you rely on the NLS_DATE_FORMAT to provide the format mask in TO_DATE() or TO_CHAR() then you should not be surprised when your queries break if this value is ever changed.

Section 6.3: Date Arithmetic - Difference between Dates in Months or Years

The difference in months between two dates can be found using the MONTHS_BETWEEN(date1, date2):

```
SELECT MONTHS_BETWEEN( DATE '2016-03-10', DATE '2015-03-10' ) AS difference FROM DUAL;
```

Outputs:

```
DIFFERENCE
----------
12
```

If the difference includes part months then it will return the fraction of the month based on there being **31** days in each month:

```
SELECT MONTHS_BETWEEN( DATE '2015-02-15', DATE '2015-01-01' ) AS difference FROM DUAL;
```

Outputs:

```
DIFFERENCE
----------
1.4516129
```

Due to `MONTHS_BETWEEN` assuming 31 days per month when there can be fewer days per month then this can result in different values for differences spanning the boundaries between months.

Example:

```
SELECT MONTHS_BETWEEN( DATE'2016-02-01', DATE'2016-02-01' - INTERVAL '1' DAY ) AS "JAN-FEB",
       MONTHS_BETWEEN( DATE'2016-03-01', DATE'2016-03-01' - INTERVAL '1' DAY ) AS "FEB-MAR",
       MONTHS_BETWEEN( DATE'2016-04-01', DATE'2016-04-01' - INTERVAL '1' DAY ) AS "MAR-APR",
       MONTHS_BETWEEN( DATE'2016-05-01', DATE'2016-05-01' - INTERVAL '1' DAY ) AS "APR-MAY"
FROM   DUAL;
```

Output:

```
JAN-FEB FEB-MAR MAR-APR APR-MAY
------- ------- ------- -------
0.03226 0.09677 0.03226 0.06452
```

The difference in years can be found by dividing the month difference by 12.

Section 6.4: Extract the Year, Month, Day, Hour, Minute or Second Components of a Date

The year, month or day components of a `DATE` data type can be found using the `EXTRACT( [ YEAR | MONTH | DAY ] FROM datevalue )`

```
SELECT EXTRACT (YEAR  FROM DATE '2016-07-25') AS YEAR,
       EXTRACT (MONTH FROM DATE '2016-07-25') AS MONTH,
       EXTRACT (DAY   FROM DATE '2016-07-25') AS DAY
FROM DUAL;
```

Outputs:

```
YEAR MONTH DAY
---- ----- ---
2016     7  25
```

The time (hour, minute or second) components can be found by either:

- Using `CAST( datevalue AS TIMESTAMP )` to convert the `DATE` to a `TIMESTAMP` and then using `EXTRACT( [ HOUR | MINUTE | SECOND ] FROM timestampvalue )`; or
- Using `TO_CHAR( datevalue, format_model )` to get the value as a string.

For example:

```
SELECT EXTRACT( HOUR   FROM CAST( datetime AS TIMESTAMP ) ) AS Hours,
       EXTRACT( MINUTE FROM CAST( datetime AS TIMESTAMP ) ) AS Minutes,
       EXTRACT( SECOND FROM CAST( datetime AS TIMESTAMP ) ) AS Seconds
FROM   (
  SELECT TO_DATE( '2016-01-01 09:42:01', 'YYYY-MM-DD HH24:MI:SS' ) AS datetime FROM DUAL
```

```
);
```

Outputs:

```
HOURS MINUTES SECONDS
----- ------- -------
9      42       1
```

Section 6.5: Generating Dates with No Time Component

All `DATE`s have a time component; however, it is customary to store dates which do not need to include time information with the hours/minutes/seconds set to zero (i.e. midnight).

Use an ANSI `DATE` literal (using ISO 8601 Date format):

```
SELECT DATE '2000-01-01' FROM DUAL;
```

Convert it from a string literal using `TO_DATE()`:

```
SELECT TO_DATE( '2001-01-01', 'YYYY-MM-DD' ) FROM DUAL;
```

(More information on the date format models can be found in the Oracle documentation.)

or:

```
SELECT TO_DATE(
         'January 1, 2000, 00:00 A.M.',
         'Month dd, YYYY, HH12:MI A.M.',
         'NLS_DATE_LANGUAGE = American'
       )
FROM   DUAL;
```

(If you are converting language specific terms such as month names then it is good practice to include the 3rd `nlsparam` parameter to the `TO_DATE()` function and specify the language to be expected.)

Section 6.6: Generating Dates with a Time Component

Convert it from a string literal using `TO_DATE()`:

```
SELECT TO_DATE( '2000-01-01 12:00:00', 'YYYY-MM-DD HH24:MI:SS' ) FROM DUAL;
```

Or use a `TIMESTAMP` literal:

```
CREATE TABLE date_table(
  date_value DATE
);

INSERT INTO date_table ( date_value ) VALUES ( TIMESTAMP '2000-01-01 12:00:00' );
```

Oracle will implicitly cast a `TIMESTAMP` to a `DATE` when storing it in a `DATE` column of a table; however you can explicitly `CAST()` the value to a `DATE`:

```
SELECT CAST( TIMESTAMP '2000-01-01 12:00:00' AS DATE ) FROM DUAL;
```

Section 6.7: The Format of a Date

In Oracle a `DATE` data type does not have a format; when Oracle sends a `DATE` to the client program (SQL/Plus, SQL/Developer, Toad, Java, Python, etc) it will send 7- or 8- bytes which represent the date.

A `DATE` which is not stored in a table (i.e. generated by `SYSDATE` and having "type 13" when using the `DUMP()` command) has 8-bytes and has the structure (the numbers on the right are the internal representation of `2012-11-26 16:41:09`):

```
BYTE VALUE                          EXAMPLE
---- ------------------------------ --------------------------------------
1    Year modulo 256                220
2    Year multiples of 256          7   (7 * 256 + 220 = 2012)
3    Month                          11
4    Day                            26
5    Hours                          16
6    Minutes                        41
7    Seconds                        9
8    Unused                         0
```

A `DATE` which is stored in a table ("type 12" when using the `DUMP()` command) has 7-bytes and has the structure (the numbers on the right are the internal representation of `2012-11-26 16:41:09`):

```
BYTE VALUE                          EXAMPLE
---- ------------------------------ --------------------------------------
1    ( Year multiples of 100 ) + 100 120
2    ( Year modulo 100 ) + 100      112 ((120-100)*100 + (112-100) = 2012)
3    Month                          11
4    Day                            26
5    Hours + 1                      17
6    Minutes + 1                    42
7    Seconds + 1                    10
```

If you want the date to have a specific format then you will need to convert it to something that has a format (i.e. a string). The SQL client may implicitly do this or you can explicitly convert the value to a string using `TO_CHAR( DATE, format_model, nls_params )`.

Section 6.8: Converting Dates to a String

Use `TO_CHAR( DATE [, format_model [, nls_params]] )`:

(Note: if a format model is not provided then the `NLS_DATE_FORMAT` session parameter will be used as the default format model; this can be different for every session so should not be relied on. It is good practice to always specify the format model.)

```
CREATE TABLE table_name (
  date_value DATE
);

INSERT INTO table_name ( date_value ) VALUES ( DATE '2000-01-01' );
INSERT INTO table_name ( date_value ) VALUES ( TIMESTAMP '2016-07-21 08:00:00' );
INSERT INTO table_name ( date_value ) VALUES ( SYSDATE );
```

Then:

```
SELECT TO_CHAR( date_value, 'YYYY-MM-DD' ) AS formatted_date FROM table_name;
```

Outputs:

```
FORMATTED_DATE
--------------
2000-01-01
2016-07-21
2016-07-21
```

And:

```
SELECT TO_CHAR(
         date_value,
         'FMMonth d yyyy, hh12:mi:ss AM',
         'NLS_DATE_LANGUAGE = French'
       ) AS formatted_date
FROM   table_name;
```

Outputs:

```
FORMATTED_DATE
----------------------------
Janvier   01 2000, 12:00:00 AM
Juillet   21 2016, 08:00:00 AM
Juillet   21 2016, 19:08:31 PM
```

Section 6.9: Changing How SQL/Plus or SQL Developer Display Dates

When SQL/Plus or SQL Developer display dates they will perform an implicit conversion to a string using the default date format model (see the Setting the Default Date Format Model example).

You can change how a date is displayed by changing the `NLS_DATE_FORMAT` parameter.

Section 6.10: Time Zones and Daylight Savings Time

The `DATE` data type does not handle time zones or changes in daylight savings time.

Either:

- use the `TIMESTAMP WITH TIME ZONE` data type; or
- handle the changes in your application logic.

A `DATE` can be stored as Coordinated Universal Time (UTC) and converted to the current session time zone like this:

```
SELECT FROM_TZ(
         CAST(
           TO_DATE( '2016-01-01 12:00:00', 'YYYY-MM-DD HH24:MI:SS' )
           AS TIMESTAMP
         ),
         'UTC'
       )
       AT LOCAL AS TIME
```

```
FROM   DUAL;
```

If you run `ALTER SESSION SET TIME_ZONE = '+01:00';` then the output is:

```
TIME
-----------------------------------
2016-01-01 13:00:00.000000000 +01:00
```

and `ALTER SESSION SET TIME_ZONE = 'PST';` then the output is:

```
TIME
-----------------------------------
2016-01-01 04:00:00.000000000 PST
```

Section 6.11: Leap Seconds

Oracle does not handle leap seconds. See My Oracle Support note `2019397.2` and `730795.1` for more details.

Section 6.12: Getting the Day of the Week

You can use `TO_CHAR( date_value, 'D' )` to get the day-of-week.

However, this is dependent on the `NLS_TERRITORY` session parameter:

```
ALTER SESSION SET NLS_TERRITORY = 'AMERICA';        -- First day of week is Sunday
SELECT TO_CHAR( DATE '1970-01-01', 'D' ) FROM DUAL;
```

Outputs 5

```
ALTER SESSION SET NLS_TERRITORY = 'UNITED KINGDOM'; -- First day of week is Monday
SELECT TO_CHAR( DATE '1970-01-01', 'D' ) FROM DUAL;
```

Outputs 4

To do this independent of the `NLS` settings, you can truncate the date to midnight of the current day (to remove any fractions of days) and subtract the date truncated to the start of the current iso-week (which always starts on Monday):

```
SELECT TRUNC( date_value ) - TRUNC( date_value, 'IW' ) + 1 FROM DUAL
```

Chapter 7: Working with Dates

Section 7.1: Date Arithmetic

Oracle supports `DATE` (includes time to the nearest second) and `TIMESTAMP` (includes time to fractions of a second) datatypes, which allow arithmetic (addition and subtraction) natively. For example:

To get the next day:

```
SELECT TO_CHAR(SYSDATE + 1, 'YYYY-MM-DD') AS tomorrow FROM dual;
```

To get the previous day:

```
SELECT TO_CHAR(SYSDATE - 1, 'YYYY-MM-DD') AS yesterday FROM dual;
```

To add 5 days to the current date:

```
SELECT TO_CHAR(SYSDATE + 5, 'YYYY-MM-DD') AS five_days_from_now FROM dual;
```

To add 5 hours to the current date:

```
SELECT TO_CHAR(SYSDATE + (5/24), 'YYYY-MM-DD HH24:MI:SS') AS five_hours_from_now FROM dual;
```

To add 10 minutes to the current date:

```
SELECT TO_CHAR(SYSDATE + (10/1440), 'YYYY-MM-DD HH24:MI:SS') AS ten_mintues_from_now FROM dual;
```

To add 7 seconds to the current date:

```
SELECT TO_CHAR(SYSDATE + (7/86400), 'YYYY-MM-DD HH24:MI:SS') AS seven_seconds_from_now FROM dual;
```

To select rows where `hire_date` is 30 days ago or more:

```
SELECT * FROM emp WHERE hire_date < SYSDATE - 30;
```

To select rows where `last_updated` column is in the last hour:

```
SELECT * FROM logfile WHERE last_updated >= SYSDATE - (1/24);
```

Oracle also provides the built-in datatype `INTERVAL` which represents a duration of time (e.g. 1.5 days, 36 hours, 2 months, etc.). These can also be used with arithmetic with `DATE` and `TIMESTAMP` expressions. For example:

```
SELECT * FROM logfile WHERE last_updated >= SYSDATE - INTERVAL '1' HOUR;
```

Section 7.2: Add_months function

Syntax: `ADD_MONTHS(p_date, INTEGER) RETURN DATE;`

Add_months function adds amt months to p_date date.

```
SELECT ADD_MONTHS(DATE'2015-01-12', 2) m FROM dual;
```

M

2015-03-12

You can also subtract months using a negative `amt`

```
SELECT ADD_MONTHS(DATE'2015-01-12', -2) m FROM dual;
```

M
2014-11-12

When the calculated month has fewer days as the given date, the last day of the calculated month will be returned.

```
SELECT TO_CHAR( ADD_MONTHS(DATE'2015-01-31', 1),'YYYY-MM-DD') m FROM dual;
```

M
2015-02-28

Chapter 8: DUAL table

Section 8.1: The following example returns the current operating system date and time

```
SELECT SYSDATE FROM dual
```

Section 8.2: The following example generates numbers between start_value and end_value

```
SELECT :start_value + LEVEL -1 n
FROM dual
CONNECT BY LEVEL <= :end_value  - :start_value + 1
```

Chapter 9: JOINS

Section 9.1: CROSS JOIN

A `CROSS JOIN` performs a join between two tables that does not use an explicit join clause and results in the Cartesian product of two tables. A Cartesian product means each row of one table is combined with each row of the second table in the join. For example, if `TABLEA` has 20 rows and `TABLEB` has 20 rows, the result would be `20*20 = 400` output rows.

Example:

```
SELECT *
FROM TABLEA CROSS JOIN TABLEB;
```

This can also be written as:

```
SELECT *
FROM TABLEA, TABLEB;
```

Here's an example of cross join in SQL between two tables:

Sample Table: TABLEA

```
+-------+---------+
| VALUE |   NAME  |
+-------+---------+
|   1   |   ONE   |
|   2   |   TWO   |
+-------+---------+
```

Sample Table: TABLEB

```
+-------+--------+
| VALUE |  NAME  |
+-------+--------+
|   3   |  THREE |
|   4   |  FOUR  |
+-------+--------+
```

Now, If you execute the query:

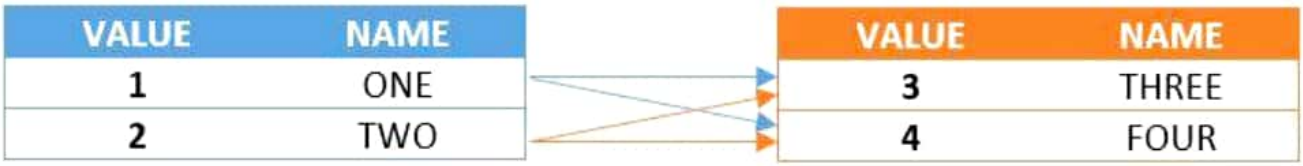
```
SELECT *
FROM TABLEA CROSS JOIN TABLEB;
```

Output:

```
+-------+--------+-------+--------+
| VALUE |  NAME  | VALUE |  NAME  |
+-------+--------+-------+--------+
|   1   |   ONE  |   3   |  THREE |
|   1   |   ONE  |   4   |  FOUR  |
|   2   |   TWO  |   3   |  THREE |
|   2   |   TWO  |   4   |  FOUR  |
```

```
+-------+--------+-------+--------+
```

This is how cross joining happens between two tables:



VALUE	NAME
1	ONE
2	TWO

VALUE	NAME
3	THREE
4	FOUR

More about Cross Join: Oracle documentation

Section 9.2: LEFT OUTER JOIN

A `LEFT OUTER JOIN` performs a join between two tables that requires an explicit join clause but does not exclude unmatched rows from the first table.

Example:

```
SELECT
      ENAME,
      DNAME,
      EMP.DEPTNO,
      DEPT.DEPTNO
FROM
      SCOTT.EMP LEFT OUTER JOIN SCOTT.DEPT
      ON EMP.DEPTNO = DEPT.DEPTNO;
```

Even though ANSI syntax is the recommended way, it is likely to encounter legacy syntax very often. Using `(+)` within a condition determines which side of the equation to be considered as *outer*.

```
SELECT
      ENAME,
      DNAME,
      EMP.DEPTNO,
      DEPT.DEPTNO
FROM
      SCOTT.EMP,
      SCOTT.DEPT
WHERE
      EMP.DEPTNO = DEPT.DEPTNO(+);
```

Here's an example of Left Outer Join between two tables:

Sample Table: EMPLOYEE

```
+-----------+---------+
|    NAME   |  DEPTNO |
+-----------+---------+
|      A    |     2   |
|      B    |     1   |
|      C    |     3   |
|      D    |     2   |
|      E    |     1   |
|      F    |     1   |
|      G    |     4   |
|      H    |     4   |
```

```
+-----------+---------+
```

Sample Table: DEPT

```
+---------+--------------+
|  DEPTNO |   DEPTNAME   |
+---------+--------------+
|    1    |  ACCOUNTING  |
|    2    |    FINANCE   |
|    5    |   MARKETING  |
|    6    |      HR      |
+---------+--------------+
```

Now, If you execute the query:

```
SELECT
      *
FROM
      EMPLOYEE LEFT OUTER JOIN DEPT
      ON EMPLOYEE.DEPTNO = DEPT.DEPTNO;
```

Output:

```
+-----------+---------+---------+--------------+
|    NAME   |  DEPTNO |  DEPTNO |   DEPTNAME   |
+-----------+---------+---------+--------------+
|     F     |    1    |    1    |  ACCOUNTING  |
|     E     |    1    |    1    |  ACCOUNTING  |
|     B     |    1    |    1    |  ACCOUNTING  |
|     D     |    2    |    2    |    FINANCE   |
|     A     |    2    |    2    |    FINANCE   |
|     C     |    3    |         |              |
|     H     |    4    |         |              |
|     G     |    4    |         |              |
+-----------+---------+---------+--------------+
```

Section 9.3: RIGHT OUTER JOIN

A `RIGHT OUTER JOIN` performs a join between two tables that requires an explicit join clause but does not exclude unmatched rows from the second table.

Example:

```
SELECT
      ENAME,
      DNAME,
      EMP.DEPTNO,
      DEPT.DEPTNO
FROM
      SCOTT.EMP RIGHT OUTER JOIN SCOTT.DEPT
      ON EMP.DEPTNO = DEPT.DEPTNO;
```

As the unmatched rows of `SCOTT.DEPT` are included, but unmatched rows of `SCOTT.EMP` are not, the above is equivalent to the following statement using `LEFT OUTER JOIN`.

```
SELECT
      ENAME,
      DNAME,
      EMP.DEPTNO,
      DEPT.DEPTNO
FROM
      SCOTT.DEPT RIGHT OUTER JOIN SCOTT.EMP
      ON DEPT.DEPTNO = EMP.DEPTNO;
```

Here's an example of Right Outer Join between two tables:

Sample Table: EMPLOYEE

```
+-----------+---------+
|    NAME   |  DEPTNO |
+-----------+---------+
|     A     |    2    |
|     B     |    1    |
|     C     |    3    |
|     D     |    2    |
|     E     |    1    |
|     F     |    1    |
|     G     |    4    |
|     H     |    4    |
+-----------+---------+
```

Sample Table: DEPT

```
+---------+-------------+
|  DEPTNO |   DEPTNAME  |
+---------+-------------+
|    1    |  ACCOUNTING |
|    2    |    FINANCE  |
|    5    |   MARKETING |
|    6    |      HR     |
+---------+-------------+
```

Now, If you execute the query:

```
SELECT
      *
FROM
      EMPLOYEE RIGHT OUTER JOIN DEPT
      ON EMPLOYEE.DEPTNO = DEPT.DEPTNO;
```

Output:

```
+-----------+---------+---------+-------------+
|    NAME   |  DEPTNO |  DEPTNO |   DEPTNAME  |
+-----------+---------+---------+-------------+
|     A     |    2    |    2    |    FINANCE  |
|     B     |    1    |    1    |  ACCOUNTING |
|     D     |    2    |    2    |    FINANCE  |
|     E     |    1    |    1    |  ACCOUNTING |
|     F     |    1    |    1    |  ACCOUNTING |
|           |         |    5    |   MARKETING |
|           |         |    6    |      HR     |
```

```
+-----------+---------+---------+-------------+
```

Oracle (+) syntax equivalent for the query is:

```
SELECT *
FROM EMPLOYEE, DEPT
WHERE EMPLOYEE.DEPTNO(+) = DEPT.DEPTNO;
```

Section 9.4: FULL OUTER JOIN

A `FULL OUTER JOIN` performs a join between two tables that requires an explicit join clause but does not exclude unmatched rows in either table. In other words, it returns all the rows in each table.

Example:

```
SELECT
      *
FROM
      EMPLOYEE FULL OUTER JOIN DEPT
      ON EMPLOYEE.DEPTNO = DEPT.DEPTNO;
```

Here's an example of Full Outer Join between two tables:

Sample Table: EMPLOYEE

```
+-----------+---------+
|    NAME   |  DEPTNO |
+-----------+---------+
|     A     |    2    |
|     B     |    1    |
|     C     |    3    |
|     D     |    2    |
|     E     |    1    |
|     F     |    1    |
|     G     |    4    |
|     H     |    4    |
+-----------+---------+
```

Sample Table: DEPT

```
+---------+--------------+
|  DEPTNO |   DEPTNAME   |
+---------+--------------+
|    1    |  ACCOUNTING  |
|    2    |    FINANCE   |
|    5    |   MARKETING  |
|    6    |      HR      |
+---------+--------------+
```

Now, If you execute the query:

```
SELECT
      *
FROM
      EMPLOYEE FULL OUTER JOIN DEPT
```

```
    ON EMPLOYEE.DEPTNO = DEPT.DEPTNO;
```

Output

```
+-----------+---------+---------+--------------+
|    NAME   |  DEPTNO |  DEPTNO |   DEPTNAME   |
+-----------+---------+---------+--------------+
|     A     |    2    |    2    |    FINANCE   |
|     B     |    1    |    1    |  ACCOUNTING  |
|     C     |    3    |         |              |
|     D     |    2    |    2    |    FINANCE   |
|     E     |    1    |    1    |  ACCOUNTING  |
|     F     |    1    |    1    |  ACCOUNTING  |
|     G     |    4    |         |              |
|     H     |    4    |         |              |
|           |         |    6    |      HR      |
|           |         |    5    |   MARKETING  |
+-----------+---------+---------+--------------+
```

Here the columns that do not match has been kept NULL.

Section 9.5: ANTIJOIN

An antijoin returns rows from the left side of the predicate for which there are no corresponding rows on the right side of the predicate. It returns rows that fail to match (NOT IN) the subquery on the right side.

```
SELECT * FROM employees
   WHERE department_id NOT IN
   (SELECT department_id FROM departments
       WHERE location_id = 1700)
   ORDER BY last_name;
```

Here's an example of Anti Join between two tables:

Sample Table: EMPLOYEE

```
+-----------+---------+
|    NAME   |  DEPTNO |
+-----------+---------+
|     A     |    2    |
|     B     |    1    |
|     C     |    3    |
|     D     |    2    |
|     E     |    1    |
|     F     |    1    |
|     G     |    4    |
|     H     |    4    |
+-----------+---------+
```

Sample Table: DEPT

```
+---------+--------------+
|  DEPTNO |   DEPTNAME   |
+---------+--------------+
|    1    |  ACCOUNTING  |
|    2    |    FINANCE   |
```

```
|    5    |   MARKETING  |
|    6    |      HR      |
+---------+--------------+
```

Now, If you execute the query:

```
SELECT
      *
FROM
      EMPLOYEE WHERE DEPTNO NOT IN
      (SELECT DEPTNO FROM DEPT);
```

Output:

```
+-----------+---------+
|    NAME   |  DEPTNO |
+-----------+---------+
|     C     |    3    |
|     H     |    4    |
|     G     |    4    |
+-----------+---------+
```

The output shows that only the rows of EMPLOYEE table, of which DEPTNO were not present in DEPT table.

Section 9.6: INNER JOIN

An INNER JOIN is a JOIN operation that allows you to specify an explicit join clause.

Syntax

TableExpression [INNER] JOIN TableExpression { ON booleanExpression | USING clause }

You can specify the join clause by specifying ON with a boolean expression.

The scope of expressions in the ON clause includes the current tables and any tables in outer query blocks to the current SELECT. In the following example, the ON clause refers to the current tables:

```
-- Join the EMP_ACT and EMPLOYEE tables
-- select all the columns from the EMP_ACT table and
-- add the employee's surname (LASTNAME) from the EMPLOYEE table
-- to each row of the result
SELECT SAMP.EMP_ACT.*, LASTNAME
 FROM SAMP.EMP_ACT JOIN SAMP.EMPLOYEE
 ON EMP_ACT.EMPNO = EMPLOYEE.EMPNO
-- Join the EMPLOYEE and DEPARTMENT tables,
-- select the employee number (EMPNO),
-- employee surname (LASTNAME),
-- department number (WORKDEPT in the EMPLOYEE table and DEPTNO in the
-- DEPARTMENT table)
-- and department name (DEPTNAME)
-- of all employees who were born (BIRTHDATE) earlier than 1930.
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
 FROM SAMP.EMPLOYEE JOIN SAMP.DEPARTMENT
 ON WORKDEPT = DEPTNO
 AND YEAR(BIRTHDATE) < 1930

-- Another example of "generating" new data values,
```

```
-- using a query which selects from a VALUES clause (which is an
-- alternate form of a fullselect).
-- This query shows how a table can be derived called "X"
-- having 2 columns "R1" and "R2" and 1 row of data
SELECT *
FROM (VALUES (3, 4), (1, 5), (2, 6))
AS VALUESTABLE1(C1, C2)
JOIN (VALUES (3, 2), (1, 2),
(0, 3)) AS VALUESTABLE2(c1, c2)
ON VALUESTABLE1.c1 = VALUESTABLE2.c1
-- This results in:
-- C1          |C2          |C1          |2
-- -----------------------------------------------
-- 3           |4           |3           |2
-- 1           |5           |1           |2


-- List every department with the employee number and
-- last name of the manager

SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
FROM DEPARTMENT INNER JOIN EMPLOYEE
ON MGRNO = EMPNO

-- List every employee number and last name
-- with the employee number and last name of their manager
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
FROM EMPLOYEE E INNER JOIN
DEPARTMENT INNER JOIN EMPLOYEE M
    ON MGRNO = M.EMPNO
    ON E.WORKDEPT = DEPTNO
```

Section 9.7: JOIN

The `JOIN` operation performs a join between two tables, excluding any unmatched rows from the first table. From Oracle 9i forward, the `JOIN` is equivalent in function to the `INNER JOIN`. This operation requires an explicit join clause, as opposed to the `CROSS JOIN` and `NATURAL JOIN` operators.

Example:

```
SELECT t1.*,
       t2.DeptId
  FROM table_1 t1
  JOIN table_2 t2 ON t2.DeptNo = t1.DeptNo
```

Oracle documentation:

- 10g
- 11g
- 12g

Section 9.8: SEMIJOIN

A semijoin query can be used, for example, to find all departments with at least one employee whose salary exceeds 2500.

```
SELECT * FROM departments
   WHERE EXISTS
```

```
    (SELECT 1 FROM employees
        WHERE departments.department_id = employees.department_id
        AND employees.salary > 2500)
    ORDER BY department_name;
```

This is more efficient than the full join alternatives, as inner joining on employees then giving a where clause detailing that the salary has to be greater than 2500 could return the same department numerous times. Say if the Fire department has n employees all with salary 3000, `SELECT * FROM departments, employees` with the necessary join on ids and our where clause would return the Fire department n times.

Section 9.9: NATURAL JOIN

NATURAL JOIN requires no explitic join condition; it builds one based on all the fields with the same name in the joined tables.

```
CREATE TABLE tab1(id NUMBER,  descr VARCHAR2(100));
CREATE TABLE tab2(id NUMBER,  descr VARCHAR2(100));
INSERT INTO tab1 VALUES(1, 'one');
INSERT INTO tab1 VALUES(2, 'two');
INSERT INTO tab1 VALUES(3, 'three');
INSERT INTO tab2 VALUES(1, 'ONE');
INSERT INTO tab2 VALUES(3, 'three');
```

The join will be done on the fields ID and DESCR, common to both the tables:

```
SQL> SELECT *
  2  FROM tab1
  3        NATURAL JOIN
  4       tab2;

ID DESCR
---------- ----------
3 three
```

Columns with different names will not be used in the JOIN condition:

```
SQL> SELECT *
  2  FROM (SELECT id AS id, descr AS descr1 FROM tab1)
  3       NATURAL JOIN
  4       (SELECT id AS id, descr AS descr2 FROM tab2);

ID DESCR1     DESCR2
---------- ---------- ----------
1 one        ONE
3 three      three
```

If the joined tables have no common columns, a JOIN with no conditions will be done:

```
SQL> SELECT *
  2  FROM (SELECT id AS id1, descr AS descr1 FROM tab1)
  3       NATURAL JOIN
  4       (SELECT id AS id2, descr AS descr2 FROM tab2);

ID1 DESCR1           ID2 DESCR2
---------- ---------- ---------- ----------
```

```
1 one                  1 ONE
2 two                  1 ONE
3 three                1 ONE
1 one                  3 three
2 two                  3 three
3 three                3 three
```

Chapter 10: Handling NULL values

A column is NULL when it has no value, regardless of the data type of that column. A column should never be compared to NULL using this syntax a `= NULL` as the result would be UNKNOWN. Instead use a `IS NULL` or a `IS NOT NULL` conditions. NULL is not equal to NULL. To compare two expressions where null can happen, use one of the functions described below. All operators except concatenation return NULL if one of their operand is NULL. For instance the result of `3 * NULL + 5` is null.

Section 10.1: Operations containing NULL are NULL, except concatenation

```
SELECT 3 * NULL + 5, 'Hello ' || NULL || 'world'    FROM DUAL;
```

3*NULL+5	'HELLO'||NULL||'WORLD'
(null)	Hello world

Section 10.2: NVL2 to get a different result if a value is null or not

If the first parameter is NOT NULL, NVL2 will return the second parameter. Otherwise it will return the third one.

```
SELECT NVL2(NULL, 'Foo', 'Bar'), NVL2(5, 'Foo', 'Bar') FROM DUAL;
```

NVL2(NULL,'FOO','BAR')	NVL2(5,'FOO','BAR')
Bar	Foo

Section 10.3: COALESCE to return the first non-NULL value

```
SELECT COALESCE(a, b, c, d, 5) FROM
    (SELECT NULL A, NULL b, NULL c, 4 d FROM DUAL);
```

COALESCE(A,B,C,D,5)
4

In some case, using COALESCE with two parameters can be faster than using NVL when the second parameter is not a constant. NVL will always evaluate both parameters. COALESCE will stop at the first non-NULL value it encounters. It means that if the first value is non-NULL, COALESCE will be faster.

Section 10.4: Columns of any data type can contain NULLs

```
SELECT 1 NUM_COLUMN, 'foo' VARCHAR2_COLUMN FROM DUAL
UNION ALL
SELECT NULL, NULL FROM DUAL;
```

NUM_COLUMN	VARCHAR2_COLUMN
1	foo
(null)	(null)

Section 10.5: Empty strings are NULL

```
SELECT 1 a, '' b FROM DUAL;
```

A	B
1	(null)

Section 10.6: NVL to replace null value

```
SELECT a column_with_null, NVL(a, 'N/A') column_without_null FROM
  (SELECT NULL a FROM DUAL);
```

COLUMN_WITH_NULL	COLUMN_WITHOUT_NULL
(null)	N/A

NVL is useful to compare two values which can contain NULLs :

```
SELECT
    CASE WHEN a = b THEN 1 WHEN a <> b THEN 0 ELSE -1 END comparison_without_nvl,
    CASE WHEN NVL(a, -1) = NVL(b, -1) THEN 1 WHEN NVL(a, -1) <> NVL(b, -1) THEN 0 ELSE -1 END
comparison_with_nvl
  FROM
    (SELECT NULL a, 3 b FROM DUAL
     UNION ALL
     SELECT NULL, NULL FROM DUAL);
```

COMPARISON_WITHOUT_NVL	COMPARISON_WITH_NVL
-1	0
-1	1

Chapter 11: String Manipulation

Section 11.1: INITCAP

The `INITCAP` function converts the case of a string so that each word starts with a capital letter and all subsequent letters are in lowercase.

```
SELECT INITCAP('HELLO mr macdonald!') AS NEW FROM dual;
```

Output

```
NEW
-------------------
Hello Mr Macdonald!
```

Section 11.2: Regular expression

Let's say we want to replace only numbers with 2 digits: regular expression will find them with `(\d\d)`

```
SELECT REGEXP_REPLACE ('2, 5, and 10 are numbers in this example', '(\d\d)', '#')
FROM dual;
```

Results in:

```
'2, 5, and # are numbers in this example'
```

If I want to swap parts of the text, I use `\1`, `\2`, `\3` to call for the matched strings:

```
SELECT REGEXP_REPLACE ('swap around 10 in that one ', '(.*)(\d\d )(.*)', '\3\2\1\3')
FROM dual;
```

Section 11.3: SUBSTR

`SUBSTR` retrieves part of a string by indicating the starting position and the number of characters to extract

```
SELECT SUBSTR('abcdefg',2,3) FROM DUAL;
```

returns:

```
bcd
```

To count from the end of the string, `SUBSTR` accepts a negative number as the second parameter, e.g.

```
SELECT SUBSTR('abcdefg',-4,2) FROM DUAL;
```

returns:

```
de
```

To get the last character in a string: `SUBSTR(mystring,-1,1)`

Section 11.4: Concatenation: Operator || or concat() function

The Oracle SQL and PL/SQL `||` operator allows you to concatenate 2 or more strings together.

Example:

Assuming the following `customers` table:

```
 id  firstname    lastname
---  -----------  ----------
1  Thomas       Woody
```

Query:

```
SELECT firstname || ' ' || lastname || ' is in my database.' AS "My Sentence"
  FROM customers;
```

Output:

```
My Sentence
--------------------------------
Thomas Woody is in my database.
```

Oracle also supports the standard SQL `CONCAT(str1, str2)` function:

Example:

Query:

```
SELECT CONCAT(firstname, ' is in my database.') FROM customers;
```

Output:

```
Expr1
--------------------------------
Thomas is in my database.
```

Section 11.5: UPPER

The UPPER function allows you to convert all lowercase letters in a string to uppercase.

```
SELECT UPPER('My text 123!') AS result FROM dual;
```

Output:

```
RESULT
------------
MY TEXT 123!
```

Section 11.6: LOWER

LOWER converts all uppercase letters in a string to lowercase.

```
SELECT LOWER('HELLO World123!') text FROM dual;
```

Outputs:

text
hello world123!

Section 11.7: LTRIM / RTRIM

`LTRIM` and `RTRIM` remove characters from the beginning or the end (respectively) of a string. A set of one or more characters may be supplied (default is a space) to remove.

For example,

```
SELECT LTRIM('<===>HELLO<===>', '=<>')
      ,RTRIM('<===>HELLO<===>', '=<>')
FROM dual;
```

Returns:

```
HELLO<===>
<===>HELLO
```

Chapter 12: IF-THEN-ELSE Statement

Section 12.1: IF-THEN

```
DECLARE
v_num1 NUMBER(10);
v_num2 NUMBER(10);

BEGIN
  v_num1 := 2;
  v_num2 := 1;

  IF v_num1 > v_num2 THEN
     DBMS_OUTPUT.put_line('v_num1 is bigger than v_num2');
  END IF;
END;
```

Section 12.2: IF-THEN-ELSE

```
DECLARE
v_num1 NUMBER(10);
v_num2 NUMBER(10);

BEGIN
  v_num1 := 2;
  v_num2 := 10;

  IF v_num1 > v_num2 THEN
     DBMS_OUTPUT.put_line('v_num1 is bigger than v_num2');
  ELSE
    DBMS_OUTPUT.put_line('v_num1 is NOT bigger than v_num2');
  END IF;
END;
```

Section 12.3: IF-THEN-ELSIF-ELSE

```
DECLARE
v_num1 NUMBER(10);
v_num2 NUMBER(10);

BEGIN
  v_num1 := 2;
  v_num2 := 2;

  IF v_num1 > v_num2 THEN
     DBMS_OUTPUT.put_line('v_num1 is bigger than v_num2');
  ELSIF v_num1 < v_num2 THEN
    DBMS_OUTPUT.put_line('v_num1 is NOT bigger than v_num2');
  ELSE
    DBMS_OUTPUT.put_line('v_num1 is EQUAL to v_num2');
  END IF;
END;
```

Chapter 13: Limiting the rows returned by a query (Pagination)

Section 13.1: Get first N rows with row limiting clause

The FETCH clause was introduced in Oracle 12c R1:

```
SELECT   val
FROM     mytable
ORDER BY val DESC
FETCH FIRST 5 ROWS ONLY;
```

An example without FETCH that works also in earlier versions:

```
SELECT * FROM (
   SELECT   val
   FROM     mytable
   ORDER BY val DESC
) WHERE ROWNUM <= 5;
```

Section 13.2: Get row N through M from many rows (before Oracle 12c)

Use the analytical function row_number():

```
WITH t AS (
  SELECT col1
  , col2
  , ROW_NUMBER() over (ORDER BY col1, col2) rn
  FROM TABLE
)
SELECT col1
, col2
FROM t
WHERE rn BETWEEN N AND M; -- N and M are both inclusive
```

Oracle 12c handles this more easily with OFFSET and FETCH.

Section 13.3: Get N numbers of Records from table

We can limit no of rows from result using rownum clause

```
SELECT * FROM
(
  SELECT val FROM  mytable
) WHERE rownum<=5
```

If we want first or last record then we want order by clause in inner query that will give result based on order.

Last Five Record :

```
SELECT * FROM
(
    SELECT val FROM  mytable ORDER BY val DESC
```

```
) WHERE rownum<=5
```

First Five Record

```
SELECT * FROM
(
    SELECT val FROM  mytable ORDER BY val
) WHERE rownum<=5
```

Section 13.4: Skipping some rows then taking some

In Oracle 12g+

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS FETCH NEXT 20 ROWS ONLY;
```

In earlier Versions

```
SELECT Id,
   Col1
 FROM (SELECT Id,
           Col1,
           ROW_NUMBER() over (ORDER BY Id) RowNumber
      FROM TableName)
WHERE RowNumber BETWEEN 21 AND 40
```

Section 13.5: Skipping some rows from result

In Oracle 12g+

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 5 ROWS;
```

In earlier Versions

```
SELECT Id,
   Col1
 FROM (SELECT Id,
           Col1,
           ROW_NUMBER() over (ORDER BY Id) RowNumber
      FROM TableName)
WHERE RowNumber > 20
```

Section 13.6: Pagination in SQL

```
SELECT val
FROM   (SELECT val, ROWNUM AS rnum
        FROM   (SELECT val
                FROM   rownum_order_test
                ORDER BY val)
        WHERE ROWNUM <= :upper_limit)
WHERE  rnum >= :lower_limit ;
```

Chapter 14: Recursive Sub-Query Factoring using the WITH Clause (A.K.A. Common Table Expressions)

Section 14.1: Splitting a Delimited String

Sample Data:

```
CREATE TABLE table_name ( VALUE VARCHAR2(50) );

INSERT INTO table_name ( VALUE ) VALUES ( 'A,B,C,D,E' );
```

Query:

```
WITH items ( list, item, lvl ) AS (
  SELECT VALUE,
         REGEXP_SUBSTR( VALUE, '[^,]+', 1, 1 ),
         1
  FROM   table_name
UNION ALL
  SELECT VALUE,
         REGEXP_SUBSTR( VALUE, '[^,]+', 1, lvl + 1 ),
         lvl + 1
  FROM   items
  WHERE  lvl < REGEXP_COUNT( VALUE, '[^,]+' )
)
SELECT * FROM items;
```

Output:

```
LIST      ITEM LVL
--------- ---- ---
A,B,C,D,E    A   1
A,B,C,D,E    B   2
A,B,C,D,E    C   3
A,B,C,D,E    D   4
A,B,C,D,E    E   5
```

Section 14.2: A Simple Integer Generator

Query:

```
WITH generator ( VALUE ) AS (
  SELECT 1 FROM DUAL
UNION ALL
  SELECT VALUE + 1
  FROM   generator
  WHERE  VALUE < 10
)
SELECT VALUE
FROM   generator;
```

Output:

Chapter 15: Different ways to update records

Section 15.1: Update using Merge

Using Merge

```
MERGE INTO
      TESTTABLE
USING
      (SELECT
            T1.ROWID AS RID,
            T2.TESTTABLE_ID
       FROM
                TESTTABLE T1
            INNER JOIN
                MASTERTABLE T2
            ON TESTTABLE.TESTTABLE_ID = MASTERTABLE.TESTTABLE_ID
       WHERE ID_NUMBER=11)
ON
      ( ROWID = RID )
WHEN MATCHED
THEN
    UPDATE SET TEST_COLUMN= 'Testvalue';
```

Section 15.2: Update Syntax with example

Normal Update

```
UPDATE
      TESTTABLE
SET
      TEST_COLUMN= 'Testvalue',TEST_COLUMN2= 123
WHERE
      EXISTS
          (SELECT MASTERTABLE.TESTTABLE_ID
           FROM MASTERTABLE
           WHERE ID_NUMBER=11);
```

Section 15.3: Update Using Inline View

Using Inline View (If it is considered updateable by Oracle)

Note: If you face a non key preserved row error add an index to resolve the same to make it update-able

```
UPDATE
      (SELECT
            TESTTABLE.TEST_COLUMN AS OLD,
            'Testvalue' AS NEW
       FROM
                TESTTABLE
            INNER JOIN
                MASTERTABLE
            ON TESTTABLE.TESTTABLE_ID = MASTERTABLE.TESTTABLE_ID
       WHERE ID_NUMBER=11) T
SET
```

```
      T.OLD        = T.NEW;
```

Section 15.4: Merge with sample data

```
DROP TABLE table01;
DROP TABLE table02;

CREATE TABLE table01 (
       code int,
       name VARCHAR(50),
       old int
);

CREATE TABLE table02 (
       code int,
       name VARCHAR(50),
       old int
);

truncate TABLE table01;
INSERT INTO table01 VALUES (1, 'A', 10);
INSERT INTO table01 VALUES (9, 'B', 12);
INSERT INTO table01 VALUES (3, 'C', 14);
INSERT INTO table01 VALUES (4, 'D', 16);
INSERT INTO table01 VALUES (5, 'E', 18);

truncate TABLE table02;
INSERT INTO table02 VALUES (1, 'AA', NULL);
INSERT INTO table02 VALUES (2, 'BB', 123);
INSERT INTO table02 VALUES (3, 'CC', NULL);
INSERT INTO table02 VALUES (4, 'DD', NULL);
INSERT INTO table02 VALUES (5, 'EE', NULL);

SELECT * FROM table01 a ORDER BY 2;
SELECT * FROM table02 a ORDER BY 2;

--

MERGE INTO table02 a USING (
      SELECT b.code, b.old FROM table01 b
) c ON (
  a.code = c.code
)
WHEN matched THEN UPDATE SET a.old = c.old
;

--

SELECT a.*, b.* FROM table01 a
inner JOIN table02 b ON a.code = b.codetable01;

SELECT * FROM table01 a
WHERE
       EXISTS
       (
         SELECT 'x' FROM table02 b WHERE a.code = b.codetable01
       );

SELECT * FROM table01 a WHERE a.code IN (SELECT b.codetable01 FROM table02 b);

--
```

Chapter 16: Update with Joins

Contrary to widespread misunderstanding (including on SO), Oracle allows updates through joins. However, there are some (pretty logical) requirements. We illustrate what doesn't work and what does through a simple example. Another way to achieve the same is the MERGE statement.

Section 16.1: Examples: what works and what doesn't

```
CREATE TABLE tgt ( id, val ) AS
  SELECT 1, 'a' FROM dual UNION ALL
  SELECT 2, 'b' FROM dual
;

TABLE TGT created.

CREATE TABLE src ( id, val ) AS
  SELECT 1, 'x' FROM dual UNION ALL
  SELECT 2, 'y' FROM dual
;

TABLE SRC created.

UPDATE
  ( SELECT t.val AS t_val, s.val AS s_val
    FROM   tgt t inner JOIN src s ON t.id = s.id
  )
SET t_val = s_val
;


SQL Error: ORA-01779: cannot modify a column which maps to a non key-preserved table
01779. 00000 -  "cannot modify a column which maps to a non key-preserved table"
*Cause:    An attempt was made to insert or update columns of a join view which
map to a non-key-preserved table.
*Action:   Modify the underlying base tables directly.
```

Imagine what would happen if we had the value `1` in the column `src.id` more than once, with different values for `src.val`. Obviously, the update would make no sense (in ANY database - that's a logical issue). Now, **we** know that there are no duplicates in `src.id`, but the Oracle engine doesn't know that - so it's complaining. Perhaps this is why so many practitioners believe Oracle "doesn't have UPDATE with joins"?

What Oracle expects is that `src.id` should be unique, and that it, Oracle, would know that beforehand. Easily fixed! Note that the same works with composite keys (on more than one column), if the matching for the update needs to use more than one column. In practice, `src.id` may be PK and `tgt.id` may be FK pointing to this PK, but that is not relevant for updates with join; what *is* relevant is the unique constraint.

```
ALTER TABLE src ADD constraint src_uc UNIQUE (id);

TABLE SRC altered.

UPDATE
  ( SELECT t.val AS t_val, s.val AS s_val
    FROM   tgt t inner JOIN src s ON t.id = s.id
  )
SET t_val = s_val
;
```

```
2 rows updated.

SELECT * FROM tgt;

ID  VAL
--  ---
 1  x
 2  y
```

The same result could be achieved with a MERGE statement (which deserves its own Documentation article), and I personally prefer MERGE in these cases, but the reason is not that "Oracle doesn't do updates with joins." As this example shows, Oracle *does* do updates with joins.

Chapter 17: Functions

Section 17.1: Calling Functions

There are a few ways to use functions.

Calling a function with an assignment statement

```
DECLARE
    x NUMBER := functionName(); --functions can be called in declaration section
BEGIN
    x := functionName();
END;
```

Calling a function in IF statement

```
IF functionName() = 100 THEN
    NULL;
END IF;
```

Calling a function in a SELECT statement

```
SELECT functionName() FROM DUAL;
```

Chapter 18: Statistical functions

Section 18.1: Calculating the median of a set of values

The MEDIAN function since Oracle 10g is an easy to use aggregation function:

```
SELECT MEDIAN(SAL)
FROM EMP
```

It returns the median of the values

Works on `DATETIME` values too.

> The result of MEDIAN is computed by first ordering the rows. Using N as the number of rows in the group, Oracle calculates the row number (RN) of interest with the formula RN = (1 + (0.5*(N-1)). The final result of the aggregate function is computed by linear interpolation between the values from rows at row numbers CRN = CEILING(RN) and FRN = FLOOR(RN).

Since Oracle 9i you can use PERCENTILE_CONT which works the same as MEDIAN function with percentile value defaults to 0.5

```
SELECT PERCENTILE_CONT(.5) WITHIN GROUP(ORDER BY SAL)
FROM EMP
```

Chapter 19: Window Functions

Section 19.1: Ratio_To_Report

Provides the ratio of the current rows value to all the values within the window.

```
--Data
CREATE TABLE Employees (Name VARCHAR2(30), Salary NUMBER(10));
INSERT INTO Employees VALUES ('Bob',2500);
INSERT INTO Employees VALUES ('Alice',3500);
INSERT INTO Employees VALUES ('Tom',2700);
INSERT INTO Employees VALUES ('Sue',2000);
--Query
SELECT Name, Salary, RATIO_TO_REPORT(Salary) OVER () AS Ratio
FROM Employees
ORDER BY Salary, Name, Ratio;
--Output
```

```
NAME                           SALARY      RATIO
------------------------------ ---------- ----------
Sue                                  2000 .186915888
Bob                                  2500  .23364486
Tom                                  2700 .252336449
Alice                                3500 .327102804
```

Chapter 20: Creating a Context

Parameter	Details
`OR REPLACE`	Redefine an existing context namespace
namespace	Name of the context - this is the namespace for calls to `SYS_CONTEXT`
schema	Owner of the package
package	Database package that sets or resets the context attributes. Note: the database package doesn't have to exist in order to create the context.
`INITIALIZED`	Specify an entity other than Oracle Database that can set the context.
`EXTERNALLY`	Allow the OCI interface to initialize the context.
`GLOBALLY`	Allow the LDAP directory to initialize the context when establishing the session.
`ACCESSED GLOBALLY`	Allow the context to be accessible throughout the entire instance - multiple sessions can share the attribute values as long as they have the same Client ID.

Section 20.1: Create a Context

```
CREATE CONTEXT my_ctx USING my_pkg;
```

This creates a context that can only be set by routines in the database package `my_pkg`, e.g.:

```
CREATE PACKAGE my_pkg AS
  PROCEDURE set_ctx;
END my_pkg;

CREATE PACKAGE BODY my_pkg AS
  PROCEDURE set_ctx IS
  BEGIN
    DBMS_SESSION.set_context('MY_CTX','THE KEY','Value');
    DBMS_SESSION.set_context('MY_CTX','ANOTHER','Bla');
  END set_ctx;
END my_pkg;
```

Now, if a session does this:

```
my_pkg.set_ctx;
```

It can now retrieve the value for the key thus:

```
SELECT SYS_CONTEXT('MY_CTX','THE KEY') FROM dual;

VALUE
```

Chapter 21: Splitting Delimited Strings

Section 21.1: Splitting Strings using a Hierarchical Query

Sample Data:

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- Multiple items in the list
SELECT 2, 'e'       FROM DUAL UNION ALL -- Single item in the list
SELECT 3, NULL      FROM DUAL UNION ALL -- NULL list
SELECT 4, 'f,,g'    FROM DUAL;          -- NULL item in the list
```

Query:

```
SELECT t.id,
       REGEXP_SUBSTR( list, '([^,]*)(,|$)', 1, LEVEL, NULL, 1 ) AS VALUE,
       LEVEL AS lvl
FROM   table_name t
CONNECT BY
       id = PRIOR id
AND    PRIOR SYS_GUID() IS NOT NULL
AND    LEVEL < REGEXP_COUNT( list, '([^,]*)(,|$)' )
```

Output:

```
        ID ITEM           LVL
---------- ------- ----------
1 a                 1
1 b                 2
1 c                 3
1 d                 4
2 e                 1
3 (NULL)            1
4 f                 1
4 (NULL)            2
4 g                 3
```

Section 21.2: Splitting Strings using a PL/SQL Function

PL/SQL Function:

```
CREATE OR REPLACE FUNCTION split_String(
  i_str    IN  VARCHAR2,
  i_delim  IN  VARCHAR2 DEFAULT ','
) RETURN SYS.ODCIVARCHAR2LIST DETERMINISTIC
AS
  p_result       SYS.ODCIVARCHAR2LIST := SYS.ODCIVARCHAR2LIST();
  p_start        NUMBER(5) := 1;
  p_end          NUMBER(5);
  c_len CONSTANT NUMBER(5) := LENGTH( i_str );
  c_ld  CONSTANT NUMBER(5) := LENGTH( i_delim );
BEGIN
  IF c_len > 0 THEN
    p_end := INSTR( i_str, i_delim, p_start );
    WHILE p_end > 0 LOOP
      p_result.EXTEND;
```

```
      p_result( p_result.COUNT ) := SUBSTR( i_str, p_start, p_end - p_start );
      p_start := p_end + c_ld;
      p_end := INSTR( i_str, i_delim, p_start );
    END LOOP;
    IF p_start <= c_len + 1 THEN
      p_result.EXTEND;
      p_result( p_result.COUNT ) := SUBSTR( i_str, p_start, c_len - p_start + 1 );
    END IF;
  END IF;
  RETURN p_result;
END;
/
```

Sample Data:

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- Multiple items in the list
SELECT 2, 'e'       FROM DUAL UNION ALL -- Single item in the list
SELECT 3, NULL      FROM DUAL UNION ALL -- NULL list
SELECT 4, 'f,,g'    FROM DUAL;          -- NULL item in the list
```

Query:

```
SELECT t.id,
       v.column_value AS VALUE,
       ROW_NUMBER() OVER ( PARTITION BY id ORDER BY ROWNUM ) AS lvl
FROM   table_name t,
       TABLE( split_String( t.list ) ) (+) v
```

Output:

```
        ID ITEM           LVL
---------- ------- ----------
1 a                1
1 b                2
1 c                3
1 d                4
2 e                1
3 (NULL)           1
4 f                1
4 (NULL)           2
4 g                3
```

Section 21.3: Splitting Strings using a Recursive Sub-query Factoring Clause

Sample Data:

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- Multiple items in the list
SELECT 2, 'e'       FROM DUAL UNION ALL -- Single item in the list
SELECT 3, NULL      FROM DUAL UNION ALL -- NULL list
SELECT 4, 'f,,g'    FROM DUAL;          -- NULL item in the list
```

Query:

```
WITH bounds ( id, list, start_pos, end_pos, lvl ) AS (
```

```
  SELECT id, list, 1, INSTR( list, ',' ), 1 FROM table_name
UNION ALL
  SELECT id,
         list,
         end_pos + 1,
         INSTR( list, ',', end_pos + 1 ),
         lvl + 1
  FROM   bounds
  WHERE  end_pos > 0
)
SELECT id,
       SUBSTR(
         list,
         start_pos,
         CASE end_pos
           WHEN 0
           THEN LENGTH( list ) + 1
           ELSE end_pos
         END - start_pos
       ) AS item,
      lvl
FROM   bounds
ORDER BY id, lvl;
```

Output:

```
        ID ITEM           LVL
---------- ------- ----------
1 a                1
1 b                2
1 c                3
1 d                4
2 e                1
3 (NULL)           1
4 f                1
4 (NULL)           2
4 g                3
```

Section 21.4: Splitting Strings using a Correlated Table Expression

Sample Data:

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- Multiple items in the list
SELECT 2, 'e'       FROM DUAL UNION ALL -- Single item in the list
SELECT 3, NULL      FROM DUAL UNION ALL -- NULL list
SELECT 4, 'f,,g'    FROM DUAL;          -- NULL item in the list
```

Query:

```
SELECT t.id,
       v.COLUMN_VALUE AS VALUE,
       ROW_NUMBER() OVER ( PARTITION BY id ORDER BY ROWNUM ) AS lvl
FROM   table_name t,
       TABLE(
         CAST(
           MULTISET(
```

```
            SELECT REGEXP_SUBSTR( t.list, '([^,]*)(,|$)', 1, LEVEL, NULL, 1 )
            FROM   DUAL
            CONNECT BY LEVEL < REGEXP_COUNT( t.list, '[^,]*(,|$)' )
          )
          AS SYS.ODCIVARCHAR2LIST
        )
      ) v;
```

Output:

```
        ID ITEM           LVL
---------- ------- ----------
1 a                1
1 b                2
1 c                3
1 d                4
2 e                1
3 (NULL)           1
4 f                1
4 (NULL)           2
4 g                3
```

Section 21.5: Splitting Strings using CROSS APPLY (Oracle 12c)

Sample Data:

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- Multiple items in the list
SELECT 2, 'e'       FROM DUAL UNION ALL -- Single item in the list
SELECT 3, NULL      FROM DUAL UNION ALL -- NULL list
SELECT 4, 'f,,g'    FROM DUAL;          -- NULL item in the list
```

Query:

```
SELECT t.id,
       REGEXP_SUBSTR( t.list, '([^,]*)($|,)', 1, l.lvl, NULL, 1 ) AS item,
       l.lvl
FROM   table_name t
       CROSS APPLY
       (
         SELECT LEVEL AS lvl
         FROM   DUAL
         CONNECT BY LEVEL <= REGEXP_COUNT( t.list, ',' ) + 1
       ) l;
```

Output:

```
        ID ITEM           LVL
---------- ------- ----------
1 a                1
1 b                2
1 c                3
1 d                4
2 e                1
3 (NULL)           1
4 f                1
4 (NULL)           2
4 g                3
```

Section 21.6: Splitting Strings using XMLTable and FLWOR expressions

This solution uses the ora:tokenize XQuery function that is available from Oracle 11.

Sample Data:

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- Multiple items in the list
SELECT 2, 'e'       FROM DUAL UNION ALL -- Single item in the list
SELECT 3, NULL      FROM DUAL UNION ALL -- NULL list
SELECT 4, 'f,,g'    FROM DUAL;          -- NULL item in the list
```

Query:

```
SELECT t.id,
       x.item,
       x.lvl
FROM   table_name t,
       XMLTABLE(
         'let $list := ora:tokenize(.,","),
              $cnt := count($list)
          for $val at $r in $list
          where $r < $cnt
          return $val'
       PASSING list||','
       COLUMNS
         item VARCHAR2(100) PATH '.',
         lvl FOR ORDINALITY
       ) (+) x;
```

Output:

```
        ID ITEM           LVL
---------- ------- ----------
1 a                1
1 b                2
1 c                3
1 d                4
2 e                1
3 (NULL)      (NULL)
4 f                1
4 (NULL)           2
4 g                3
```

Section 21.7: Splitting Delimited Strings using XMLTable

Sample Data:

```
CREATE TABLE table_name ( id, list ) AS
SELECT 1, 'a,b,c,d' FROM DUAL UNION ALL -- Multiple items in the list
SELECT 2, 'e'       FROM DUAL UNION ALL -- Single item in the list
SELECT 3, NULL      FROM DUAL UNION ALL -- NULL list
SELECT 4, 'f,,g'    FROM DUAL;          -- NULL item in the list
```

Query:

```
SELECT t.id,
       SUBSTR( x.item.getStringVal(), 2 ) AS item,
       x.lvl
FROM   table_name t
       CROSS JOIN
       XMLTABLE(
         ( '"#' || REPLACE( t.list, ',', '","#' ) || '"' )
         COLUMNS item XMLTYPE PATH '.',
                 lvl  FOR ORDINALITY
       ) x;
```

(Note: the # character is appended to facilitate extracting `NULL` values; it is later removed using `SUBSTR( item, 2 )`. If `NULL` values are not required then you can simplify the query and omit this.)

Output:

```
        ID ITEM           LVL
---------- ------- ----------
1 a                1
1 b                2
1 c                3
1 d                4
2 e                1
3 (NULL)           1
4 f                1
4 (NULL)           2
4 g                3
```

Chapter 22: Collections and Records

Section 22.1: Use a collection as a return type for a split function

It's necessary to declare the type; here `t_my_list`; a collection is a `TABLE OF something`

```
CREATE OR REPLACE TYPE t_my_list AS TABLE OF VARCHAR2(100);
```

Here's the function. Notice the `()` used as a kind of constructor, and the `COUNT` and `EXTEND` keywords that help you create and grow your collection;

```
CREATE OR REPLACE
FUNCTION cto_table(p_sep IN VARCHAR2, p_list IN VARCHAR2)
  RETURN t_my_list
AS
--- this function takes a string list, element being separated by p_sep
--                                                          as separator
  l_string VARCHAR2(4000) := p_list || p_sep;
  l_sep_index PLS_INTEGER;
  l_index PLS_INTEGER := 1;
  l_tab t_my_list     := t_my_list();
BEGIN
  LOOP
    l_sep_index := INSTR(l_string, p_sep, l_index);
    EXIT
  WHEN l_sep_index = 0;
    l_tab.EXTEND;
    l_tab(l_tab.COUNT) := TRIM(SUBSTR(l_string,l_index,l_sep_index - l_index));
    l_index            := l_sep_index + 1;
  END LOOP;
  RETURN l_tab;
END cto_table;
/
```

Then you can see the content of the collection with the `TABLE()` function from SQL; it can be used as a list inside a SQL `IN ( ..)` statement:

```
SELECT * FROM A_TABLE
 WHERE A_COLUMN IN ( TABLE(cto_table('|','a|b|c|d')) )
--- gives the records where A_COLUMN in ('a', 'b', 'c', 'd') --
```

Chapter 23: Object Types

Section 23.1: Accessing stored objects

```
CREATE SEQUENCE test_seq START WITH 1001;

CREATE TABLE test_tab
(
   test_id  INTEGER,
   test_obj base_type,
   PRIMARY KEY (test_id)
);

INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, base_type(1,'BASE_TYPE'));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, base_type(2,'BASE_TYPE'));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, mid_type(3, 'MID_TYPE',SYSDATE - 1));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, mid_type(4, 'MID_TYPE',SYSDATE + 1));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, leaf_type(5, 'LEAF_TYPE',SYSDATE - 20,'Maple'));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.NEXTVAL, leaf_type(6, 'LEAF_TYPE',SYSDATE + 20,'Oak'));
```

Returns object reference:

```
SELECT test_id
      ,test_obj
  FROM test_tab;
```

Returns object reference, pushing all to subtype

```
SELECT test_id
      ,TREAT(test_obj AS mid_type) AS obj
  FROM test_tab;
```

Returns a string descriptor of each object, by type

```
SELECT test_id
      ,TREAT(test_obj AS base_type).to_string() AS to_string -- Parenthesis are needed after the
function name, or Oracle will look for an attribute of this name.
  FROM test_tab;
```

Section 23.2: BASE_TYPE

Type declaration:

```
CREATE OR REPLACE TYPE base_type AS OBJECT
(
   base_id     INTEGER,
   base_attr   VARCHAR2(400),
   null_attr   INTEGER, -- Present only to demonstrate non-default constructors
   CONSTRUCTOR FUNCTION base_type
   (
      i_base_id INTEGER,
```

```
      i_base_attr VARCHAR2
   ) RETURN SELF AS RESULT,
   MEMBER FUNCTION get_base_id RETURN INTEGER,
   MEMBER FUNCTION get_base_attr RETURN VARCHAR2,
   MEMBER PROCEDURE set_base_id(i_base_id INTEGER),
   MEMBER PROCEDURE set_base_attr(i_base_attr VARCHAR2),
   MEMBER FUNCTION to_string RETURN VARCHAR2
) INSTANTIABLE NOT FINAL
```

Type body:

```
CREATE OR REPLACE TYPE BODY base_type AS
   CONSTRUCTOR FUNCTION base_type
   (
      i_base_id INTEGER,
      i_base_attr VARCHAR2
   ) RETURN SELF AS RESULT
   IS
   BEGIN
      self.base_id := i_base_id;
      self.base_attr := i_base_attr;
      RETURN;
   END base_type;

   MEMBER FUNCTION get_base_id RETURN INTEGER IS
   BEGIN
      RETURN self.base_id;
   END get_base_id;

   MEMBER FUNCTION get_base_attr RETURN VARCHAR2 IS
   BEGIN
      RETURN self.base_attr;
   END get_base_attr;

   MEMBER PROCEDURE set_base_id(i_base_id INTEGER) IS
   BEGIN
      self.base_id := i_base_id;
   END set_base_id;

   MEMBER PROCEDURE set_base_attr(i_base_attr VARCHAR2) IS
   BEGIN
      self.base_attr := i_base_attr;
   END set_base_attr;

   MEMBER FUNCTION to_string RETURN VARCHAR2 IS
   BEGIN
      RETURN 'BASE_ID ['||self.base_id||']; BASE_ATTR ['||self.base_attr||']';
   END to_string;
END;
```

Section 23.3: MID_TYPE

Type declaration:

```
CREATE OR REPLACE TYPE mid_type UNDER base_type
(
   mid_attr DATE,
   CONSTRUCTOR FUNCTION mid_type
   (
      i_base_id   INTEGER,
```

```
      i_base_attr VARCHAR2,
      i_mid_attr  DATE
   ) RETURN SELF AS RESULT,
   MEMBER FUNCTION get_mid_attr RETURN DATE,
   MEMBER PROCEDURE set_mid_attr(i_mid_attr DATE),
   OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2
) INSTANTIABLE NOT FINAL
```

Type body:

```
CREATE OR REPLACE TYPE BODY mid_type AS
   CONSTRUCTOR FUNCTION mid_type
   (
      i_base_id   INTEGER,
      i_base_attr VARCHAR2,
      i_mid_attr  DATE
   ) RETURN SELF AS RESULT
   IS
   BEGIN
      self.base_id := i_base_id;
      self.base_attr := i_base_attr;
      self.mid_attr := i_mid_attr;
      RETURN;
   END mid_type;

   MEMBER FUNCTION get_mid_attr RETURN DATE IS
   BEGIN
      RETURN self.mid_attr;
   END get_mid_attr;

   MEMBER PROCEDURE set_mid_attr(i_mid_attr DATE) IS
   BEGIN
      self.mid_attr := i_mid_attr;
   END set_mid_attr;

   OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2
   IS
   BEGIN
      RETURN (SELF AS base_type).to_string || '; MID_ATTR [' || self.mid_attr || ']';
   END to_string;
END;
```

Section 23.4: LEAF_TYPE

Type declaration:

```
CREATE OR REPLACE TYPE leaf_type UNDER mid_type
(
   leaf_attr VARCHAR2(1000),
   CONSTRUCTOR FUNCTION leaf_type
   (
      i_base_id   INTEGER,
      i_base_attr VARCHAR2,
      i_mid_attr  DATE,
      i_leaf_attr VARCHAR2
   ) RETURN SELF AS RESULT,
   MEMBER FUNCTION get_leaf_attr RETURN VARCHAR2,
   MEMBER PROCEDURE set_leaf_attr(i_leaf_attr VARCHAR2),
   OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2
) INSTANTIABLE FINAL
```

Type Body:

```
CREATE OR REPLACE TYPE BODY leaf_type AS
   CONSTRUCTOR FUNCTION leaf_type
   (
      i_base_id   INTEGER,
      i_base_attr VARCHAR2,
      i_mid_attr  DATE,
      i_leaf_attr VARCHAR2
   ) RETURN SELF AS RESULT
   IS
   BEGIN
      self.base_id := i_base_id;
      self.base_attr := i_base_attr;
      self.mid_attr := i_mid_attr;
      self.leaf_attr := i_leaf_attr;
      RETURN;
   END leaf_type;

   MEMBER FUNCTION get_leaf_attr RETURN VARCHAR2 IS
   BEGIN
      RETURN self.leaf_attr;
   END get_leaf_attr;

   MEMBER PROCEDURE set_leaf_attr(i_leaf_attr VARCHAR2) IS
   BEGIN
      self.leaf_attr := i_leaf_attr;
   END set_leaf_attr;

   OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2 IS
   BEGIN
      RETURN (SELF AS mid_type).to_string || '; LEAF_ATTR [' || self.leaf_attr || ']';
   END to_string;
END;
```

Chapter 24: Loop

Section 24.1: Simple Loop

```
DECLARE
v_counter NUMBER(2);

BEGIN
  v_counter := 0;
  LOOP
    v_counter := v_counter + 1;
    DBMS_OUTPUT.put_line('Line number' || v_counter);

    EXIT WHEN v_counter = 10;
  END LOOP;
END;
```

Section 24.2: WHILE Loop

The WHILE loop is executed untill the condition of end is fulfilled. Simple example:

```
DECLARE
v_counter NUMBER(2); --declaration of counter variable

BEGIN
  v_counter := 0; --point of start, first value of our iteration

  WHILE v_counter < 10 LOOP --exit condition

    DBMS_OUTPUT.put_line('Current iteration of loop is ' || v_counter); --show current iteration
number in dbms script output
    v_counter := v_counter + 1; --incrementation of counter value, very important step

  END LOOP; --end of loop declaration
END;
```

This loop will be executed untill current value of variable v_counter will be less than ten.

The result:

```
CURRENT iteration OF LOOP IS 0
CURRENT iteration OF LOOP IS 1
CURRENT iteration OF LOOP IS 2
CURRENT iteration OF LOOP IS 3
CURRENT iteration OF LOOP IS 4
CURRENT iteration OF LOOP IS 5
CURRENT iteration OF LOOP IS 6
CURRENT iteration OF LOOP IS 7
CURRENT iteration OF LOOP IS 8
CURRENT iteration OF LOOP IS 9
```

The most important thing is, that our loop starts with '0' value, so first line of results is 'Current iteration of loop is 0'.

Section 24.3: FOR Loop

Loop FOR works on similar rules as other loops. FOR loop is executed exact number of times and this number is

known at the beginning - lower and upper limits are directly set in code. In every step in this example, loop is increment by 1.

Simple example:

```
DECLARE
v_counter NUMBER(2); --declaration of counter variable

BEGIN
  v_counter := 0; --point of start, first value of our iteration, execute of variable

  FOR v_counter IN 1..10 LOOP --The point, where lower and upper point of loop statement is declared
- in this example, loop will be executed 10 times, start with value of 1

    DBMS_OUTPUT.put_line('Current iteration of loop is ' || v_counter); --show current iteration
number in dbms script output

  END LOOP; --end of loop declaration
END;
```

And the result is:

```
CURRENT iteration OF LOOP IS 1
CURRENT iteration OF LOOP IS 2
CURRENT iteration OF LOOP IS 3
CURRENT iteration OF LOOP IS 4
CURRENT iteration OF LOOP IS 5
CURRENT iteration OF LOOP IS 6
CURRENT iteration OF LOOP IS 7
CURRENT iteration OF LOOP IS 8
CURRENT iteration OF LOOP IS 9
CURRENT iteration OF LOOP IS 10
```

Loop FOR has additional property, which is working in reverse. Using additional word 'REVERSE' in declaration of lower and upper limit of loop allow to do that. Every execution of loop decrement value of v_counter by 1.

Example:

```
DECLARE
v_counter NUMBER(2); --declaration of counter variable

BEGIN
  v_counter := 0; --point of start

  FOR v_counter IN REVERSE 1..10 LOOP

    DBMS_OUTPUT.put_line('Current iteration of loop is ' || v_counter); --show current iteration
number in dbms script output

  END LOOP; --end of loop declaration
END;
```

And the result:

```
CURRENT iteration OF LOOP IS 10
CURRENT iteration OF LOOP IS 9
CURRENT iteration OF LOOP IS 8
CURRENT iteration OF LOOP IS 7
CURRENT iteration OF LOOP IS 6
```

Chapter 25: Cursors

Section 25.1: Parameterized "FOR loop" Cursor

```
DECLARE
  CURSOR c_emp_to_be_raised(p_sal emp.sal%TYPE) IS
    SELECT * FROM emp WHERE  sal < p_sal;
BEGIN
  FOR cRowEmp IN c_emp_to_be_raised(1000) LOOP
    DBMS_OUTPUT.Put_Line(cRowEmp .eName ||' ' ||cRowEmp.sal||'... should be raised ;)');
  END LOOP;
END;
/
```

Section 25.2: Implicit "FOR loop" cursor

```
BEGIN
  FOR x IN (SELECT * FROM emp WHERE sal < 100) LOOP
    DBMS_OUTPUT.Put_Line(x.eName ||' '||x.sal||'... should REALLY be raised :D');
  END LOOP;
END;
/
```

- First advantage is there is no tedious declaration to do (think of this horrible "CURSOR" thing you had in previous versions)
- second advantage is you first build your select query, then when you have what you want, you immediately can access the fields of your query (`x.<myfield>`) in your PL/SQL loop
- The loop opens the cursor and fetches one record at a time for every loop. At the end of the loop the cursor is closed.
- Implicit cursors are faster because the interpreter's work grows as the code gets longer. The less code the less work the interpreter has to do.

Section 25.3: Handling a CURSOR

- Declare the cursor to scan a list of records
- Open it
- Fetch current record into variables (this increments position)
- Use `%notfound` to detect end of list
- Don't forget to close the cursor to limit resources consumption in current context

--

```
DECLARE
  CURSOR curCols IS -- select column name and type from a given table
         SELECT column_name, data_type FROM all_tab_columns WHERE table_name='MY_TABLE';
  v_tab_column all_tab_columns.column_name%TYPE;
  v_data_type all_tab_columns.data_type%TYPE;
  v_ INTEGER := 1;
BEGIN
  OPEN curCols;
  LOOP
     FETCH curCols INTO v_tab_column, v_data_type;
     IF curCols%notfound OR v_ > 2000 THEN
       EXIT;
     END IF;
```

```
      DBMS_OUTPUT.put_line(v_||':Column '||v_tab_column||' is of '|| v_data_type||' Type.');
      v_:= v_ + 1;
   END LOOP;

   -- Close in any case
   IF curCols%ISOPEN THEN
      CLOSE curCols;
   END IF;
END;
/
```

Section 25.4: Working with SYS_REFCURSOR

SYS_REFCURSOR can be used as a return type when you need to easily handle a list returned not from a table, but more specifically from a function:

function returning a cursor

```
CREATE OR REPLACE FUNCTION list_of (required_type_in IN VARCHAR2)
   RETURN SYS_REFCURSOR
IS
   v_ SYS_REFCURSOR;
BEGIN
   CASE required_type_in
      WHEN 'CATS'
      THEN
         OPEN v_ FOR
           SELECT nickname FROM (
                SELECT 'minou' nickname FROM dual
      UNION ALL SELECT 'minâ'           FROM dual
      UNION ALL SELECT 'minon'          FROM dual
           );
      WHEN 'DOGS'
      THEN
         OPEN v_ FOR
              SELECT dog_call FROM (
                SELECT 'bill'   dog_call FROM dual
      UNION ALL SELECT 'nestor'          FROM dual
      UNION ALL SELECT 'raoul'           FROM dual
           );
   END CASE;
   -- Whit this use, you must not close the cursor.
   RETURN v_;
END list_of;
/
```

and how to use it:

```
DECLARE
   v_names   SYS_REFCURSOR;
   v_        VARCHAR2 (32767);
BEGIN
   v_names := list_of('CATS');
   LOOP
      FETCH v_names INTO v_;
      EXIT WHEN v_names%NOTFOUND;
      DBMS_OUTPUT.put_line(v_);
   END LOOP;
   -- here you close it
   CLOSE v_names;
END;
/
```

Chapter 26: Sequences

Parameter	Details
schema	schema name
increment by	interval between the numbers
start with	first number needed
maxvalue	Maximum value for the sequence
nomaxvalue	Maximum value is defaulted
minvalue	minimum value for the sequence
nominvalue	minimum value is defaulted
cycle	Reset to the start after reaching this value
nocycle	Default
cache	Preallocation limit
nocache	Default
order	Guarantee the order of numbers
noorder	default

Section 26.1: Creating a Sequence: Example

Purpose

Use the CREATE SEQUENCE statement to create a sequence, which is a database object from which multiple users may generate unique integers. You can use sequences to automatically generate primary key values.

When a sequence number is generated, the sequence is incremented, independent of the transaction committing or rolling back. If two users concurrently increment the same sequence, then the sequence numbers each user acquires may have gaps, because sequence numbers are being generated by the other user. One user can never acquire the sequence number generated by another user. After a sequence value is generated by one user, that user can continue to access that value regardless of whether the sequence is incremented by another user.

Sequence numbers are generated independently of tables, so the same sequence can be used for one or for multiple tables. It is possible that individual sequence numbers will appear to be skipped, because they were generated and used in a transaction that ultimately rolled back. Additionally, a single user may not realize that other users are drawing from the same sequence.

After a sequence is created, you can access its values in SQL statements with the CURRVAL pseudocolumn, which returns the current value of the sequence, or the NEXTVAL pseudocolumn, which increments the sequence and returns the new value.

Prerequisites

To create a sequence in your own schema, you must have the CREATE SEQUENCE system privilege.

To create a sequence in another user's schema, you must have the CREATE ANY SEQUENCE system privilege.

Creating a Sequence: Example The following statement creates the sequence customers_seq in the sample schema oe. This sequence could be used to provide customer ID numbers when rows are added to the customers table.

```
CREATE SEQUENCE customers_seq
START WITH     1000
INCREMENT BY   1
NOCACHE
```

Chapter 27: Indexes

Here I will explain different index using example, how index increase query performance, how index decrease DML performance etc

Section 27.1: b-tree index

```
CREATE INDEX ord_customer_ix ON orders (customer_id);
```

By default, if we do not mention anything, oracle creates an index as a b-tree index. But we should know when to use it. B-tree index stores data as binary tree format. As we know that, index is a schema object which stores some sort of entry for each value for the indexed column. So, whenever any search happens on those columns, it checks in the index for the exact location of that record to access fast. Few points about indexing:

- To search for entry in the index, some sort of binary search algorithm used.
- When **data cardinality is high, b-tree** index is perfect to use.
- Index makes DML slow, as for each record, there should be one entry in the index for indexed column.
- So, if not necessary, we should avoid creating index.

Section 27.2: Bitmap Index

```
CREATE BITMAP INDEX
emp_bitmap_idx
ON index_demo (gender);
```

- Bitmap index is used when **data cardinality is low.**
- Here, **Gender** has value with low cardinality. Values are may be Male, Female & others.
- So, if we create a binary tree for this 3 values while searching it will have unnecessary traverse.
- In bitmap structures, a two-dimensional array is created with one column for every row in the table being indexed. Each column represents a distinct value within the bitmapped index. This two-dimensional array represents each value within the index multiplied by the number of rows in the table.
- At row retrieval time, Oracle decompresses the bitmap into the RAM data buffers so it can be rapidly scanned for matching values. These matching values are delivered to Oracle in the form of a Row-ID list, and these Row-ID values may directly access the required information.

Section 27.3: Function Based Index

```
CREATE INDEX first_name_idx ON user_data (UPPER(first_name));
```

```
SELECT *
FROM   user_data
WHERE  UPPER(first_name) = 'JOHN2';
```

- Function based index means, creating index based on a function.
- If in search (where clause), frequently any function is used, it's better to create index based on that function.
- Here, in the example, for search, **Upper()** function is being used. So, it's better to create index using upper function.

Chapter 28: Hints

Parameters	Details
Degree of Parallelism (DOP)	It is the number of parallel connection/processes which you want your query to open up. It is usually 2, 4, 8, 16 so on.
Table Name	The name of the table on which parallel hint will be applied.

Section 28.1: USE_NL

Use Nested Loops.

Usage : `use_nl(A B)`

This hint will ask the engine to use nested loop method to join the tables A and B. That is row by row comparison. The hint does not force the order of the join, just asks for NL.

```
SELECT /*+use_nl(e d)*/ *
FROM Employees E
JOIN Departments D ON E.DepartmentID = D.ID
```

Section 28.2: APPEND HINT

"Use DIRECT PATH method for inserting new rows".

The APPEND hint instructs the engine to use direct path load. This means that the engine will not use a conventional insert using memory structures and standard locks, but will write directly to the tablespace the data. Always creates new blocks which are appended to the table's segment. This will be faster, but have some limitations:

- You cannot read from the table you appended in the same session until you commmit or rollback the transaction.
- If there are triggers defined on the table Oracle will not use direct path(it's a different story for sqlldr loads).
- others

Example.

```
INSERT /*+append*/ INTO Employees
SELECT *
FROM Employees;
```

Section 28.3: Parallel Hint

Statement-level parallel hints are the easiest:

```
SELECT /*+ PARALLEL(8) */ first_name, last_name FROM employee emp;
```

Object-level parallel hints give more control but are more prone to errors; developers often forget to use the alias instead of the object name, or they forget to include some objects.

```
SELECT /*+ PARALLEL(emp,8) */ first_name, last_name FROM employee emp;
```

```
SELECT /*+ PARALLEL(table_alias,Degree of Parallelism) */ FROM table_name table_alias;
```

Let's say a query takes 100 seconds to execute without using parallel hint. If we change DOP to 2 for same query,

then *ideally* the same query with parallel hint will take 50 second. Similarly using DOP as 4 will take 25 seconds.

In practice, parallel execution depends on many other factors and does not scale linearly. This is especially true for small run times where the parallel overhead may be larger than the gains from running in multiple parallel servers.

Section 28.4: USE_HASH

Instructs the engine to use hash method to join tables in the argument.

Usage : `use_hash(TableA [TableB] ... [TableN])`

As explained in many places, "in a HASH join, Oracle accesses one table (usually the smaller of the joined results) and builds a hash table on the join key in memory. It then scans the other table in the join (usually the larger one) and probes the hash table for matches to it."

It is preferred against Nested Loops method when the tables are big, no indexes are at hand, etc.

Note: The hint does not force the order of the join, just asks for HASH JOIN method.

Example of usage:

```
SELECT /*+use_hash(e d)*/ *
FROM Employees E
JOIN Departments D ON E.DepartmentID = D.ID
```

Section 28.5: FULL

The FULL hint tells Oracle to perform a full table scan on a specified table, no matter if an index can be used.

```
CREATE TABLE fullTable(id) AS SELECT LEVEL FROM dual CONNECT BY LEVEL < 100000;
CREATE INDEX idx ON fullTable(id);
```

With no hints, the index is used:

```
SELECT COUNT(1) FROM fullTable f WHERE id BETWEEN 10 AND 100;
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    13 |     3   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE   |      |     1 |    13 |            |          |
|*  2 |   INDEX RANGE SCAN| IDX  |     2 |    26 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

FULL hint forces a full scan:

```
SELECT /*+ full(f) */ COUNT(1) FROM fullTable f WHERE id BETWEEN 10 AND 100;
```

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |     1 |    13 |    47   (3)| 00:00:01 |
|   1 |  SORT AGGREGATE    |           |     1 |    13 |            |          |
|*  2 |   TABLE ACCESS FULL| FULLTABLE |     2 |    26 |    47   (3)| 00:00:01 |
-------------------------------------------------------------------------------
```

Section 28.6: Result Cache

Oracle (*11g and above*) allows the SQL queries to be cached in the SGA and reused to improve performance. It queries the data from cache rather than database. Subsequent execution of same query is faster because now the data is being pulled from cache.

```
SELECT /*+ result_cache */ NUMBER FROM main_table;
```

Output -

```
Number
------
1
2
3
4
5
6
7
8
9
10

Elapsed: 00:00:02.20
```

If I run the same query again now, the time to execute will reduce since the data is now fetched from cache which was set during the first execution.

Output -

```
Number
------
1
2
3
4
5
6
7
8
9
10

Elapsed: 00:00:00.10
```

Notice how the elapsed time reduced from **2.20 seconds** to **0.10 seconds**.

> Result Cache holds the cache until the data in database is updated/altered/deleted. Any change will release the cache.

Chapter 29: Packages

Section 29.1: Define a Package header and body with a function

In this example we define a package header and a package body wit a function.
After that we are calling a function from the package that return a return value.

Package header:

```
CREATE OR REPLACE PACKAGE SkyPkg AS

       FUNCTION GetSkyColour(vPlanet IN VARCHAR2)
       RETURN VARCHAR2;

END;
/
```

Package body:

```
CREATE OR REPLACE PACKAGE BODY SkyPkg AS

        FUNCTION GetSkyColour(vPlanet IN VARCHAR2)
        RETURN VARCHAR2
        AS
               vColour VARCHAR2(100) := NULL;
        BEGIN
               IF vPlanet = 'Earth' THEN
                       vColour := 'Blue';
               ELSIF vPlanet = 'Mars' THEN
                       vColour := 'Red';
               END IF;

               RETURN vColour;
        END;

END;
/
```

Calling the function from the package body:

```
DECLARE
        vColour VARCHAR2(100);
BEGIN
        vColour := SkyPkg.GetSkyColour(vPlanet => 'Earth');
        DBMS_OUTPUT.PUT_LINE(vColour);
END;
/
```

Section 29.2: Overloading

Functions and procedures in packages can be overloaded. The following package **TEST** has two procedures called **print_number**, which behave differently depending on parameters they are called with.

```
CREATE OR REPLACE PACKAGE TEST IS
  PROCEDURE print_number(p_number IN INTEGER);
```

```
  PROCEDURE print_number(p_number IN VARCHAR2);
END TEST;
/
CREATE OR REPLACE PACKAGE BODY TEST IS

  PROCEDURE print_number(p_number IN INTEGER) IS
  BEGIN
    DBMS_OUTPUT.put_line('Digit: ' || p_number);
  END;

  PROCEDURE print_number(p_number IN VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.put_line('String: ' || p_number);
  END;

END TEST;
/
```

We call both procedures. The first with integer parameter, the second with varchar2.

```
SET serveroutput ON;
-- call the first procedure
exec test.print_number(3);
-- call the second procedure
exec test.print_number('three');
```

The output of the above script is:

```
SQL>
Digit: 3
PL/SQL procedure successfully completed
String: three
PL/SQL procedure successfully completed
```

Restrictions on Overloading

Only local or packaged subprograms, or type methods, can be overloaded. Therefore, you cannot overload standalone subprograms. Also, you cannot overload two subprograms if their formal parameters differ only in name or parameter mode

Section 29.3: Package Usage

Packages in PL/SQL are a collection of procedures, functions, variables, exceptions, constants, and data structures. Generally the resources in a package are related to each other and accomplish similar tasks.

Why Use Packages

- Modularity
- Better Performance/ Funtionality

Parts of a Package

Specification - Sometimes called a package header. Contains variable and type declarations and the signatures of the functions and procedures that are in the package which are **public** to be called from outside the package.

Package Body - Contains the code and **private** declarations.

Chapter 30: Exception Handling

Oracle produces a variety of exceptions. You may be surprised how tedious it can be to have your code stop with some unclear message. To improve your PL/SQL code's ability to get fixed easily it is necessary to handle exceptions at the lowest level. Never hide an exception "under the carpet", unless you're here to keep your piece of code for you only and for no one else to maintain.

The predefined errors.

Section 30.1: Syntax

The general syntax for exception section:

```
DECLARE
    declaration Section
BEGIN
    some statements

EXCEPTION
    WHEN exception_one THEN
        DO something
    WHEN exception_two THEN
        DO something
    WHEN exception_three THEN
        DO something
    WHEN OTHERS THEN
        DO something
END;
```

An exception section has to be on the end of the PL/SQL block. PL/SQL gives us the opportunity to nest blocks, then each block may have its own exception section for example:

```
CREATE OR REPLACE PROCEDURE nested_blocks
IS
BEGIN
    some statements
    BEGIN
        some statements

    EXCEPTION
        WHEN exception_one THEN
            DO something
    END;
EXCEPTION
    WHEN exception_two THEN
        DO something
END;
```

If exception will be raised in the nested block it should be handled in the inner exception section, but if inner exception section does not handle this exception then this exception will go to exception section of the external block.

Section 30.2: User defined exceptions

As the name suggest user defined exceptions are created by users. If you want to create your own exception you have to:

1. Declare the exception
2. Raise it from your program
3. Create suitable exception handler to catch him.

Example

I want to update all salaries of workers. But if there are no workers, raise an exception.

```
CREATE OR REPLACE PROCEDURE update_salary
IS
    no_workers EXCEPTION;
    v_counter NUMBER := 0;
BEGIN
    SELECT COUNT(*) INTO v_counter FROM emp;
    IF v_counter = 0 THEN
        RAISE no_workers;
    ELSE
        UPDATE emp SET salary = 3000;
    END IF;

    EXCEPTION
        WHEN no_workers THEN
            raise_application_error(-20991,'We don''t have workers!');
END;
/
```

What does it mean `RAISE`?

Exceptions are raised by database server automatically when there is a need, but if you want, you can raise explicitly any exception using `RAISE`.

Procedure `raise_application_error(error_number,error_message);`

- error_number must be between -20000 and -20999
- error_message message to display when error occurs.

Section 30.3: Internally defined exceptions

An internally defined exception doesn't have a name, but it has its own code.

When to use it?

If you know that your database operation might raise specific exceptions those which don't have names, then you can give them names so that you can write exception handlers specifically for them. Otherwise, you can use them only with `OTHERS` exception handlers.

Syntax

```
DECLARE
    my_name_exc EXCEPTION;
    PRAGMA exception_init(my_name_exc,-37);
BEGIN
    ...
EXCEPTION
    WHEN my_name_exc THEN
        DO something
END;
```

`my_name_exc EXCEPTION;` that is the exception name declaration.

`PRAGMA exception_init(my_name_exc,-37);` assign name to the error code of internally defined exception.

Example

We have an emp_id which is a primary key in emp table and a foreign key in dept table. If we try to remove emp_id when it has child records, it will be thrown an exception with code -2292.

```
CREATE OR REPLACE PROCEDURE remove_employee
IS
    emp_exception EXCEPTION;
    PRAGMA exception_init(emp_exception,-2292);
BEGIN
    DELETE FROM emp WHERE emp_id = 3;
EXCEPTION
    WHEN emp_exception THEN
        DBMS_OUTPUT.put_line('You can not do that!');
END;
/
```

> Oracle documentation says: "An internally defined exception with a user-declared name is still an internally defined exception, not a user-defined exception."

Section 30.4: Predefined exceptions

Predefined exceptions are internally defined exceptions but they have names. Oracle database raise this type of exceptions automatically.

Example

```
CREATE OR REPLACE PROCEDURE insert_emp
IS
BEGIN
    INSERT INTO emp (emp_id, ename) VALUES ('1','Jon');

EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        DBMS_OUTPUT.put_line('Duplicate value on index!');
END;
/
```

Below are examples exceptions name with theirs codes:

Exception Name	Error Code
NO_DATA_FOUND	-1403
ACCESS_INTO_NULL	-6530
CASE_NOT_FOUND	-6592
ROWTYPE_MISMATCH	-6504
TOO_MANY_ROWS	-1422
ZERO_DIVIDE	-1476

Full list of exception names and their codes on Oracle web-site.

Section 30.5: Define custom exception, raise it and see where it comes from

To illustrate this, here is a function that has 3 different "wrong" behaviors

- the parameter is completely stupid: we use a user-defined expression
- the parameter has a typo: we use Oracle standard NO_DATA_FOUND error
- another, but not handled case

Feel free to adapt it to your standards:

```
DECLARE
  this_is_not_acceptable EXCEPTION;
  PRAGMA EXCEPTION_INIT(this_is_not_acceptable, -20077);
  g_err VARCHAR2 (200) := 'to-be-defined';
  w_schema all_tables.OWNER%TYPE;

  PROCEDURE get_schema( p_table IN VARCHAR2, p_schema OUT VARCHAR2)
  IS
    w_err VARCHAR2 (200) := 'to-be-defined';
  BEGIN
    w_err := 'get_schema-step-1:';
    IF (p_table = 'Delivery-Manager-Is-Silly') THEN
      RAISE this_is_not_acceptable;
    END IF;
    w_err := 'get_schema-step-2:';
    SELECT owner INTO p_schema
      FROM all_tables
     WHERE table_name LIKE(p_table||'%');
  EXCEPTION
  WHEN NO_DATA_FOUND THEN
    -- handle Oracle-defined exception
     DBMS_OUTPUT.put_line('[WARN]'||w_err||'This can happen. Check the table name you entered.');
  WHEN this_is_not_acceptable THEN
    -- handle your custom error
     DBMS_OUTPUT.put_line('[WARN]'||w_err||'Please don''t make fun of the delivery manager.');
  WHEN OTHERS THEN
     DBMS_OUTPUT.put_line('[ERR]'||w_err||'unhandled exception:'||SQLERRM);
     RAISE;
  END Get_schema;

BEGIN
  g_err := 'Global; first call:';
  get_schema('Delivery-Manager-Is-Silly', w_schema);
  g_err := 'Global; second call:';
  get_schema('AAA', w_schema);
  g_err := 'Global; third call:';
  get_schema('', w_schema);
  g_err := 'Global; 4th call:';
  get_schema('Can''t reach this point due to previous error.', w_schema);

EXCEPTION
  WHEN OTHERS THEN
     DBMS_OUTPUT.put_line('[ERR]'||g_err||'unhandled exception:'||SQLERRM);
  -- you may raise this again to the caller if error log isn't enough.
--  raise;
END;
/
```

Giving on a regular database:

```
[WARN]get_schema-step-1:Please don't make fun of the delivery manager.
[WARN]get_schema-step-2:This can happen. Check the table name you entered.
[ERR]get_schema-step-2:unhandled exception:ORA-01422: exact fetch returns more than requested
number of rows
[ERR]Global; third call:unhandled exception:ORA-01422: exact fetch returns more than requested
number of rows
```

Remember that exception are here to handle *rare* cases. I saw applications who raised an exception at every access, just to ask for the user password, saying "not connected"... so much computation waste.

Section 30.6: Handling connexion error exceptions

Each standard Oracle error is associated with an error number. It's important to anticipate what could go wrong in your code. Here for a connection to another database, it can be:

- `-28000` account is locked
- `-28001` password expired
- `-28002` grace period
- `-1017` wrong user / password

Here is a way to test what goes wrong with the user used by the database link:

```
DECLARE
  v_dummy NUMBER;
BEGIN
  -- testing db link
  EXECUTE IMMEDIATE 'select COUNT(1) from dba_users@pass.world' INTO v_dummy ;
  -- if we get here, exception wasn't raised: display COUNT's result
  DBMS_OUTPUT.put_line(v_dummy||' users on PASS db');

EXCEPTION
  -- exception can be referred by their name in the predefined Oracle's list
    WHEN LOGIN_DENIED
    THEN
        DBMS_OUTPUT.put_line('ORA-1017 / USERNAME OR PASSWORD INVALID, TRY AGAIN');
    WHEN OTHERS
    THEN
  -- or referred by their number: stored automatically in reserved variable SQLCODE
        IF  SQLCODE = '-2019'
        THEN
          DBMS_OUTPUT.put_line('ORA-2019 / Invalid db_link name');
        ELSIF SQLCODE = '-1035'
        THEN
          DBMS_OUTPUT.put_line('ORA-1035 / DATABASE IS ON RESTRICTED SESSION, CONTACT YOUR DBA');

        ELSIF SQLCODE = '-28000'
        THEN
          DBMS_OUTPUT.put_line('ORA-28000 / ACCOUNT IS LOCKED. CONTACT YOUR DBA');
        ELSIF SQLCODE = '-28001'
        THEN
          DBMS_OUTPUT.put_line('ORA-28001 / PASSWORD EXPIRED. CONTACT YOUR DBA FOR CHANGE');
        ELSIF SQLCODE  = '-28002'
        THEN
          DBMS_OUTPUT.put_line('ORA-28002 / PASSWORD IS EXPIRED, CHANGED IT');
        ELSE
   -- and if it's not one of the exception you expected
          DBMS_OUTPUT.put_line('Exception not specifically handled');
          DBMS_OUTPUT.put_line('Oracle Said'||SQLCODE||':'||SQLERRM);
```

```
        END IF;
END;
/
```

Section 30.7: Exception handling

1. What is an exception?

   Exception in PL/SQL is an error created during a program execution.

   We have three types of exceptions:

   - Internally defined exceptions
   - Predefined exceptions
   - User-defined exceptions

2. What is an exception handling?

   Exception handling is a possibility to keep our program running even if appear runtime error resulting from for example coding mistakes, hardware failures.We avoid it from exiting abruptly.

Chapter 31: Error logging

Section 31.1: Error logging when writing to database

Create Oracle error log table ERR$_EXAMPLE for existing EXAMPLE table:

```
EXECUTE DBMS_ERRLOG.CREATE_ERROR_LOG('EXAMPLE', NULL, NULL, NULL, TRUE);
```

Make writing operation with SQL:

```
INSERT INTO EXAMPLE (COL1) VALUES ('example')
LOG ERRORS INTO ERR$_EXAMPLE reject limit unlimited;
```

Chapter 32: Database Links

Section 32.1: Creating a database link

```
CREATE DATABASE LINK dblink_name
CONNECT TO remote_username
IDENTIFIED BY remote_password
USING 'tns_service_name';
```

The remote DB will then be accessible in the following way:

```
SELECT * FROM MY_TABLE@dblink_name;
```

To test a database link connection without needing to know any of the object names in the linked database, use the following query:

```
SELECT * FROM DUAL@dblink_name;
```

To explicitly specify a domain for the linked database service, the domain name is added to the `USING` statement. For example:

```
USING 'tns_service_name.WORLD'
```

If no domain name is explicitly specified, Oracle uses the domain of the database in which the link is being created.

Oracle documentation for database link creation:

- 10g: https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_5005.htm
- 11g: https://docs.oracle.com/cd/B28359_01/server.111/b28310/ds_concepts002.htm
- 12g: https://docs.oracle.com/database/121/SQLRF/statements_5006.htm#SQLRF01205

Section 32.2: Create Database Link

Let we assume we have two databases "ORA1" and "ORA2". We can access the objects of "ORA2" from database "ORA1" using a database link.

Prerequisites: For creating a private Database link you need a `CREATE DATABASE LINK` privilege. For creating a private Database link you need a `CREATE PUBLIC DATABASE LINK` privilege.

*Oracle Net must be present on both the instances.

How to create a database link:

From ORA1:

```
SQL> CREATE <public> DATABASE LINK ora2 CONNECT TO user1 IDENTIFIED BY pass1 USING <tns name OF
ora2>;
```

Database link created.

Now that we have the DB link set up, we can prove that by running the following from ORA1:

```
SQL> SELECT name FROM V$DATABASE@ORA2; -- should return ORA2
```

You can also access the DB Objects of "ORA2" from "ORA1", given the user `user1` has the `SELECT` privilege on those objects on ORA2 (such as TABLE1 below):

```
SELECT COUNT(*) FROM TABLE1@ORA2;
```

Pre-requistes:

- Both databases must be up and running (opened).
- Both database listeners must be up and running.
- TNS must be configured correctly.
- User user1 must be present in ORA2 database, password must be checked and verified.
- User user1 must have at least the `SELECT` privilege, or any other required to access the objects on ORA2.

Chapter 33: Table partitioning

Partitioning is a functionality to split tables and indexes into smaller pieces. It is used to improve performance and to manage the smaller pieces individually. The partition key is a column or a set of columns that defines in which partition each row is going to be stored. Partitioning Overview in official Oracle documentation

Section 33.1: Select existing partitions

Check existing partitions on Schema

```
SELECT * FROM user_tab_partitions;
```

Section 33.2: Drop partition

```
ALTER TABLE table_name DROP PARTITION partition_name;
```

Section 33.3: Select data from a partition

Select data from a partition

```
SELECT * FROM orders PARTITION(partition_name);
```

Section 33.4: Split Partition

Splits some partition into two partitions with another high bound.

```
ALTER TABLE table_name SPLIT PARTITION old_partition
    AT (new_high_bound) INTO (PARTITION new_partition TABLESPACE new_tablespace,
    PARTITION old_partition)
```

Section 33.5: Merge Partitions

Merge two partitions into single one

```
ALTER TABLE table_name
  MERGE PARTITIONS first_partition, second_partition
  INTO  PARTITION  splitted_partition TABLESPACE new_tablespace
```

Section 33.6: Exchange a partition

Exchange/convert a partition to a non-partitioned table and vice versa. This facilitates a fast "move" of data between the data segments (opposed to doing something like "insert...select" or "create table...as select") as the operation is DDL (the partition exchange operation is a data dictionary update without moving the actual data) and not DML (large undo/redo overhead).

Most basic examples :

1. Convert a non-partitioned table (table "B") to a partition (of table "A") :

Table "A" doesn't contain data in partition "OLD_VALUES" and table "B" contains data

```
ALTER TABLE "A" EXCHANGE PARTITION "OLD_VALUES" WITH TABLE "B";
```

Result : data is "moved" from table "B" (contains no data after operation) to partition "OLD_VALUES"

2. Convert a partition to a non-partitioned table :

Table "A" contains data in partition "OLD_VALUES" and table "B" doesn't contain data

```
ALTER TABLE "A" EXCHANGE PARTITION "OLD_VALUES" WITH TABLE "B";
```

Result : data is "moved" from partition "OLD_VALUES" (contains no data after operation) to table "B"

Note : there is a quite a few additional options, features and restrictions for this operation

Further info can be found on this link ---> "https://docs.oracle.com/cd/E11882_01/server.112/e25523/part_admin002.htm#i1107555" (section "Exchanging Partitions")

Section 33.7: Hash partitioning

This creates a table partitioned by hash, in this example on store id.

```
CREATE TABLE orders (
    order_nr NUMBER(15),
    user_id VARCHAR2(2),
    order_value NUMBER(15),
    store_id NUMBER(5)
)
PARTITION BY HASH(store_id) PARTITIONS 8;
```

You should use a power of 2 for the number of hash partitions, so that you get an even distribution in partition size.

Section 33.8: Range partitioning

This creates a table partitioned by ranges, in this example on order values.

```
CREATE TABLE orders (
    order_nr NUMBER(15),
    user_id VARCHAR2(2),
    order_value NUMBER(15),
    store_id NUMBER(5)
)
PARTITION BY RANGE(order_value) (
    PARTITION p1 VALUES LESS THAN(10),
    PARTITION p2 VALUES LESS THAN(40),
    PARTITION p3 VALUES LESS THAN(100),
    PARTITION p4 VALUES LESS THAN(MAXVALUE)
);
```

Section 33.9: List partitioning

This creates a table partitioned by lists, in this example on store id.

```
CREATE TABLE orders (
    order_nr NUMBER(15),
    user_id VARCHAR2(2),
    order_value NUMBER(15),
    store_id NUMBER(5)
)
```

```
PARTITION BY LIST(store_id) (
    PARTITION p1 VALUES (1,2,3),
    PARTITION p2 VALUES(4,5,6),
    PARTITION p3 VALUES(7,8,9),
    PARTITION p4 VALUES(10,11)
);
```

Section 33.10: Truncate a partition

```
ALTER TABLE table_name TRUNCATE PARTITION partition_name;
```

Section 33.11: Rename a partition

```
ALTER TABLE table_name RENAME PARTITION p3 TO p6;
```

Section 33.12: Move partition to different tablespace

```
ALTER TABLE table_name
MOVE PARTITION partition_name TABLESPACE tablespace_name;
```

Section 33.13: Add new partition

```
ALTER TABLE table_name
ADD PARTITION new_partition VALUES LESS THAN(400);
```

Chapter 34: Oracle Advanced Queuing (AQ)

Section 34.1: Simple Producer/Consumer

Overview

Create a queue that we can send a message to. Oracle will notify our stored procedure that a message has been enqueued and should be worked. We'll also add some subprograms we can use in an emergency to stop messages from being deqeued, allow dequeuing again, and run a simple batch job to work through all of the messages.

These examples were tested on Oracle Database 12c Enterprise Edition Release 12.1.0.2.0 - 64bit Production.

Create Queue

We will create a message type, a queue table that can hold the messages, and a queue. Messages in the queue will be dequeued first by priority then be their enqueue time. If anything goes wrong working the message and the dequeue is rolled-back AQ will make the message available for dequeue 3600 seconds later. It will do this 48 times before moving it an exception queue.

```
CREATE TYPE message_t AS object
   (
   sender  VARCHAR2 ( 50 ),
   message VARCHAR2 ( 512 )
   );
/
-- Type MESSAGE_T compiled
BEGIN DBMS_AQADM.create_queue_table(
     queue_table        => 'MESSAGE_Q_TBL',
     queue_payload_type => 'MESSAGE_T',
     sort_list          => 'PRIORITY,ENQ_TIME',
     multiple_consumers =>  FALSE,
     compatible         => '10.0.0');
  END;
/
-- PL/SQL procedure successfully completed.
BEGIN DBMS_AQADM.create_queue(
     queue_name          => 'MESSAGE_Q',
     queue_table         => 'MESSAGE_Q_TBL',
     queue_type          =>  0,
     max_retries         =>  48,
     retry_delay         =>  3600,
     dependency_tracking =>  FALSE);
  END;
/
-- PL/SQL procedure successfully completed.
```

Now that we have a place to put the messages lets create a package to manage and work messages in the queue.

```
CREATE OR REPLACE PACKAGE message_worker_pkg
IS
   queue_name_c CONSTANT VARCHAR2(20) := 'MESSAGE_Q';

   -- allows the workers to process messages in the queue
   PROCEDURE enable_dequeue;

   -- prevents messages from being worked but will still allow them to be created and enqueued
```

```
   PROCEDURE disable_dequeue;

   -- called only by Oracle Advanced Queueing.  Do not call anywhere else.
   PROCEDURE on_message_enqueued (context        IN RAW,
                                  reginfo        IN sys.aq$_reg_info,
                                  descr          IN sys.aq$_descriptor,
                                  payload        IN RAW,
                                  payloadl       IN NUMBER);

   -- allows messages to be worked if we missed the notification (or a retry
   -- is pending)
   PROCEDURE work_old_messages;

END;
/

CREATE OR REPLACE PACKAGE BODY message_worker_pkg
IS
   -- raised by Oracle when we try to dequeue but no more messages are ready to
   -- be dequeued at this moment
   no_more_messages_ex          EXCEPTION;
   PRAGMA exception_init (no_more_messages_ex,
                          -25228);

   -- allows the workers to process messages in the queue
   PROCEDURE enable_dequeue
   AS
   BEGIN
      DBMS_AQADM.start_queue (queue_name => queue_name_c, dequeue => TRUE);
   END enable_dequeue;

   -- prevents messages from being worked but will still allow them to be created and enqueued
   PROCEDURE disable_dequeue
   AS
   BEGIN
      DBMS_AQADM.stop_queue (queue_name => queue_name_c, dequeue => TRUE, enqueue => FALSE);
   END disable_dequeue;

   PROCEDURE work_message (message_in IN OUT NOCOPY message_t)
   AS
   BEGIN
      DBMS_OUTPUT.put_line ( message_in.sender || ' says ' || message_in.message );
   END work_message;

   -- called only by Oracle Advanced Queueing.  Do not call anywhere else.

   PROCEDURE on_message_enqueued (context        IN RAW,
                                  reginfo        IN sys.aq$_reg_info,
                                  descr          IN sys.aq$_descriptor,
                                  payload        IN RAW,
                                  payloadl       IN NUMBER)
   AS
      PRAGMA autonomous_transaction;
      dequeue_options_l      DBMS_AQ.dequeue_options_t;
      message_id_l           RAW (16);
      message_l              message_t;
      message_properties_l   DBMS_AQ.message_properties_t;
   BEGIN
      dequeue_options_l.msgid         := descr.msg_id;
      dequeue_options_l.consumer_name := descr.consumer_name;
      dequeue_options_l.wait          := DBMS_AQ.no_wait;
      DBMS_AQ.dequeue (queue_name           => descr.queue_name,
```

```
                        dequeue_options      => dequeue_options_l,
                        message_properties   => message_properties_l,
                        payload              => message_l,
                        msgid                => message_id_l);
      work_message (message_l);
      COMMIT;
   EXCEPTION
      WHEN no_more_messages_ex
      THEN
         -- it's possible work_old_messages already dequeued the message
         COMMIT;
      WHEN OTHERS
      THEN
         -- we don't need to have a raise here.  I just wanted to point out that
         -- since this will be called by AQ throwing the exception back to it
         -- will have it put the message back on the queue and retry later
         RAISE;
   END on_message_enqueued;

   -- allows messages to be worked if we missed the notification (or a retry
   -- is pending)
   PROCEDURE work_old_messages
   AS
      PRAGMA autonomous_transaction;
      dequeue_options_l      DBMS_AQ.dequeue_options_t;
      message_id_l           RAW (16);
      message_l              message_t;
      message_properties_l   DBMS_AQ.message_properties_t;
   BEGIN
      dequeue_options_l.wait       := DBMS_AQ.no_wait;
      dequeue_options_l.navigation := DBMS_AQ.first_message;

      WHILE (TRUE) LOOP -- way out is no_more_messages_ex
         DBMS_AQ.dequeue (queue_name           => queue_name_c,
                          dequeue_options      => dequeue_options_l,
                          message_properties   => message_properties_l,
                          payload              => message_l,
                          msgid                => message_id_l);
         work_message (message_l);
         COMMIT;
      END LOOP;
   EXCEPTION
      WHEN no_more_messages_ex
      THEN
         NULL;
   END work_old_messages;
END;
```

Next tell AQ that when a message is enqueued to MESSAGE_Q (and committed) notify our procedure it has work to do. AQ will start up a job in its own session to handle this.

```
BEGIN
  DBMS_AQ.register (
     sys.aq$_reg_info_list (
        sys.aq$_reg_info (USER || '.' || message_worker_pkg.queue_name_c,
                          DBMS_AQ.namespace_aq,
                          'plsql://' || USER || '.message_worker_pkg.on_message_enqueued',
                          HEXTORAW ('FF'))),
     1);
  COMMIT;
END;
```

Start Queue and Send a Message

```
DECLARE
   enqueue_options_1      DBMS_AQ.enqueue_options_t;
   message_properties_1   DBMS_AQ.message_properties_t;
   message_id_1           RAW (16);
   message_1              message_t;
BEGIN
   -- only need to do this next line ONCE
   DBMS_AQADM.start_queue (queue_name => message_worker_pkg.queue_name_c, enqueue => TRUE , dequeue
=> TRUE);

   message_1 := NEW message_t ( 'Jon', 'Hello, world!' );
   DBMS_AQ.enqueue (queue_name           => message_worker_pkg.queue_name_c,
                    enqueue_options      => enqueue_options_1,
                    message_properties   => message_properties_1,
                    payload              => message_1,
                    msgid                => message_id_1);
   COMMIT;
END;
```

Chapter 35: constraints

Section 35.1: Update foreign keys with new value in Oracle

Suppose you have a table and you want to change one of this table primary id. you can use the following scrpit. primary ID here is "PK_S"

```
BEGIN
  FOR i IN (SELECT a.table_name, c.column_name
              FROM user_constraints a, user_cons_columns c
             WHERE a.CONSTRAINT_TYPE = 'R'
               AND a.R_CONSTRAINT_NAME = 'PK_S'
               AND c.constraint_name = a.constraint_name) LOOP


        EXECUTE IMMEDIATE 'update ' || i.table_name || ' set ' || i.column_name ||
                    '=to_number(''1000'' || ' ||  i.column_name || ') ';



  END LOOP;

END;
```

Section 35.2: Disable all related foreign keys in oracle

Suppose you have the table T1 and it has relation with many tables and its primary key constraint name is "pk_t1" you want to disable these foreign keys you can use:

```
BEGIN
    FOR I IN (SELECT table_name, constraint_name FROM user_constraint t WHERE
r_constraint_name='pk_t1') LOOP

EXECUTE IMMEDIATE ' alter table ' || I.table_name || ' disable constraint ' || i.constraint_name;

   END LOOP;
END;
```

Chapter 36: Autonomous Transactions

Section 36.1: Using autonomous transaction for logging errors

The following procedure is a generic one which will be used to log all errors in an application to a common error log table.

```
CREATE OR REPLACE PROCEDURE log_errors
(
  p_calling_program IN VARCHAR2,
  p_error_code IN INTEGER,
  p_error_description IN VARCHAR2
)
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO error_log
  VALUES
  (
  p_calling_program,
  p_error_code,
  p_error_description,
  SYSDATE,
  USER
  );
  COMMIT;
END log_errors;
```

The following anonymous PL/SQL block shows how to call the log_errors procedure.

```
BEGIN
   DELETE FROM dept WHERE deptno = 10;
EXCEPTION
   WHEN OTHERS THEN
      log_errors('Delete dept',SQLCODE, SQLERRM);
      RAISE;
END;

SELECT * FROM error_log;

CALLING_PROGRAM    ERROR_CODE    ERROR_DESCRIPTION
 ERROR_DATETIME         DB_USER
DELETE dept        -2292         ORA-02292: integrity constraint violated - child RECORD found
 08/09/2016             APEX_PUBLIC_USER
```

Chapter 37: Oracle MAF

Section 37.1: To get value from Binding

```
ValueExpression ve = AdfmfJavaUtilities.getValueExpression(<binding>, String.class);
String <variable_name> = (String) ve.getValue(AdfmfJavaUtilities.getELContext());
```

Here "binding" indicates the EL expression from which the value is to be get.

"variable_name" the parameter to which the value from the binding to be stored

Section 37.2: To set value to binding

```
ValueExpression ve = AdfmfJavaUtilities.getValueExpression(<binding>, String.class);
ve.setValue(AdfmfJavaUtilities.getELContext(), <value>);
```

Here "binding" indicates the EL expression to which the value is to be stored.

"value" is the desired value to be add to the binding

Section 37.3: To invoke a method from binding

```
AdfELContext adfELContext = AdfmfJavaUtilities.getAdfELContext();
MethodExpression me;
me = AdfmfJavaUtilities.getMethodExpression(<binding>, Object.class, NEW Class[] { });
me.invoke(adfELContext, NEW Object[] { });
```

"binding" indicates the EL expression from which a method to be invoked

Section 37.4: To call a javaScript function

```
AdfmfContainerUtilities.invokeContainerJavaScriptFunction(AdfmfJavaUtilities.getFeatureId(),
<function>, NEW Object[] {
                                                          });
```

"function" is the desired js function to be invoked

Chapter 38: level query

Section 38.1: Generate N Number of records

```
SELECT ROWNUM NO FROM DUAL CONNECT BY LEVEL <= 10
```

Section 38.2: Few usages of Level Query

/* This is a simple query which can generate a sequence of numbers. The following example generates a sequence of numbers from 1..100 */

```
SELECT LEVEL FROM dual CONNECT BY LEVEL <= 100;
```

/*The above query is useful in various scenarios like generating a sequence of dates from a given date. The following query generates 10 consecutive dates */

```
SELECT TO_DATE('01-01-2017','mm-dd-yyyy')+level-1 AS  dates FROM dual CONNECT BY LEVEL <= 10;
```

01-JAN-17
02-JAN-17
03-JAN-17
04-JAN-17
05-JAN-17
06-JAN-17
07-JAN-17
08-JAN-17
09-JAN-17
10-JAN-17

Chapter 39: Hierarchical Retrieval With Oracle Database 12C

You can use hierarchical queries to retrieve data based on a natural hierarchical relationship between rows in a table

Section 39.1: Using the CONNECT BY Caluse

```
SELECT E.EMPLOYEE_ID,E.LAST_NAME,E.MANAGER_ID FROM HR.EMPLOYEES E
CONNECT BY PRIOR E.EMPLOYEE_ID = E.MANAGER_ID;
```

The `CONNECT BY` clause to define the relationship between employees and managers.

Section 39.2: Specifying the Direction of the Query From the Top Down

```
SELECT E.LAST_NAME|| ' reports to ' ||
PRIOR E.LAST_NAME "Walk Top Down"
FROM HR.EMPLOYEES E
START WITH E.MANAGER_ID IS NULL
CONNECT BY PRIOR E.EMPLOYEE_ID = E.MANAGER_ID;
```

Chapter 40: Data Pump

Following are the steps to create a data pump import/export:

Section 40.1: Monitor Datapump jobs

Datapump jobs can be monitored using

1. data dictionary views:

```
    SELECT * FROM dba_datapump_jobs;
    SELECT * FROM DBA_DATAPUMP_SESSIONS;
    SELECT username,opname,target_desc,sofar,totalwork,message FROM V$SESSION_LONGOPS WHERE username
= 'bkpadmin';
```

2. Datapump status:

- Note down the job name from the import/export logs or data dictionary name and
- Run **attach** command:
- type status in Import/Export prompt

```
impdp <bkpadmin>/<bkp123> attach=<SYS_IMPORT_SCHEMA_01>
Import> status
```

Press press **CTRL+C** to come out of Import/Export prompt

Section 40.2: Step 3/6 : Create directory

```
CREATE OR REPLACE directory DATAPUMP_REMOTE_DIR AS '/oracle/scripts/expimp';
```

Section 40.3: Step 7 : Export Commands

Commands:

```
expdp <bkpadmin>/<bkp123>  parfile=<EXP.par>
```

*Please replace the data in <> with appropriate values as per your environment. You can add/modify parameters as per your requirements. In the above example all the remaining parameters are added in parameter files as stated below: *

- Export Type : **User Export**
- Export entire schema
- Parameter file details [say exp.par] :

```
schemas=<schema>
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>.dmp
logfile=exp_<dbname>_<schema>.LOG
```

- Export Type : **User Export for large schema**
- Export entire schema for large datasets: Here the export dump files will be broken down and compressed. Parallelism is used here *(Note : Adding parallelism will increase the CPU load on server)*

- Parameter file details [say exp.par] :

```
schemas=<schema>
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>_%U.dmp
logfile=exp_<dbname>_<schema>.LOG
compression = ALL
parallel=5
```

- Export Type : **Table Export** [Export set of tables]
- Parameter file details [say exp.par] :

```
tables= tname1, tname2, tname3
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>.dmp
logfile=exp_<dbname>_<schema>.LOG
```

Section 40.4: Step 9 : Import Commands

Prerequisite:

- Prior to user import it is a good practice to drop the schema or table imported.

Commands:

```
impdp <bkpadmin>/<bkp123>  parfile=<imp.par>
```

*Please replace the data in <> with appropriate values as per your environment. You can add/modify parameters as per your requirements. In the above example all the remaining parameters are added in parameter files as stated below: *

- Import Type : **User Import**
- Import entire schema
- Parameter file details [say imp.par] :

```
schemas=<schema>
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>.dmp
logfile=imp_<dbname>_<schema>.LOG
```

- Import Type : **User Import for large schema**
- Import entire schema for large datasets: Parallelism is used here *(Note : Adding parallelism will increase the CPU load on server)*
- Parameter file details [say imp.par] :

```
schemas=<schema>
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>_%U.dmp
logfile=imp_<dbname>_<schema>.LOG
parallel=5
```

- Import Type : **Table Import** [Import set of tables]

- Parameter file details [say imp.par] :

```
tables= tname1, tname2, tname3
directory= DATAPUMP_REMOTE_DIR
dumpfile=<dbname>_<schema>.dmp
logfile=exp_<dbname>_<schema>.LOG
TABLE_EXISTS_ACTION= <APPEND /SKIP /TRUNCATE /REPLACE>
```

Section 40.5: Datapump steps

Source Server [Export Data]	Target Server [Import Data]
1. Create a datapump folder that will contain the export dump files	4. Create a datapump folder that will contain the import dump files
2. Login to database schema that will perform the export.	5. Login to database schema that will perform the import.
3. Create directory pointing to step 1.	6. Create directory pointing to step 4.
7. Run Export Statements.	
8. Copy/SCP the dump files to Target Server.	
	9. Run Import statements
	10. check data ,compile invalid objects and provide related grants

Section 40.6: Copy tables between different schemas and tablespaces

```
 expdp <bkpadmin>/<bkp123> directory=DATAPUMP_REMOTE_DIR dumpfile=<customer.dmp>


 impdp <bkpadmin>/<bkp123> directory=DATAPUMP_REMOTE_DIR dumpfile=<customer.dmp>
remap_schema=<source schema>:<target schema> remap_tablespace=<source tablespace>:<target
tablespace>
```

Chapter 41: Bulk collect

Section 41.1: Bulk data Processing

local collections are not allowed in select statements. Hence the first step is to create a schema level collection. If the collection is not schema level and being used in SELECT statements then it would cause "PLS-00642: local collection types not allowed in SQL statements"

```
CREATE OR REPLACE TYPE table1_t IS OBJECT (
a_1 INTEGER,
a_2 VARCHAR2(10)
);
```

--Grant permissions on collection so that it could be used publically in database

```
GRANT EXECUTE ON table1_t TO PUBLIC;
CREATE OR REPLACE TYPE table1_tbl_typ IS TABLE OF table1_t;
GRANT EXECUTE ON table1_tbl_typ TO PUBLIC;
```

--fetching data from table into collection and then loop through the collection and print the data.

```
DECLARE
 table1_tbl table1_tbl_typ;
BEGIN
 table1_tbl := table1_tbl_typ();
  SELECT table1_t(a_1,a_2)
  BULK COLLECT INTO table1_tbl
  FROM table1 WHERE ROWNUM<10;

 FOR rec IN (SELECT a_1 FROM TABLE(table1_tbl))--table(table1_tbl) won't give error)
 LOOP
   DBMS_OUTPUT.put_line('a_1'||rec.a_1);
   DBMS_OUTPUT.put_line('a_2'||rec.a_2);
 END LOOP;
END;
/
```

Chapter 42: Real Application Security

Oracle Real Application Security was introduced in Oracle 12c. It summarize many Security Topics like User-Role-Model, Access Control, Application vs. Database, End-User-Security or Row- and Column Level Security

Section 42.1: Application

To associate an Application with something in the Database there are three main parts:

Application Privilege: An Application Privilege describes Privileges like `SELECT`, `INSERT`, `UPDATE`, `DELETE`, ... Application Privileges can be summarized as an Aggregate Privilege.

```
XS$PRIVILEGE(
    name=>'privilege_name'
    [, implied_priv_list=>XS$NAME_LIST('"SELECT"', '"INSERT"', '"UPDATE"', '"DELETE"')]
)

XS$PRIVILEGE_LIST(
    XS$PRIVILEGE(...),
    XS$PRIVILEGE(...),
    ...
);
```

Application User:

Simple Application User:

```
BEGIN
    SYS.XS_PRINCIPAL.CREATE_USER('user_name');
END;
```

Direct Login Application User:

```
BEGIN
    SYS.XS_PRINCIPAL.CREATE_USER(name => 'user_name', schema => 'schema_name');
END;

BEGIN
    SYS.XS_PRINCIPAL.SET_PASSWORD('user_name', 'password');
END;
CREATE PROFILE prof LIMIT
    PASSWORD_REUSE_TIME 1/4440
    PASSWORD_REUSE_MAX 3
    PASSWORD_VERIFY_FUNCTION Verify_Pass;

BEGIN
    SYS.XS_PRINCIPAL.SET_PROFILE('user_name', 'prof');
END;

BEGIN
    SYS.XS_PRINCIPAL.GRANT_ROLES('user_name', 'XSONNCENT');
END;
```

(optional:)

```
BEGIN
    SYS.XS_PRINCIPAL.SET_VERIFIER('user_name', '6DFF060084ECE67F', XS_PRINCIPAL.XS_SHA512");
```

```
END;
```

Application Role:

Regular Application Role:

```
DECLARE
    st_date TIMESTAMP WITH TIME ZONE;
    ed_date TIMESTAMP WITH TIME ZONE;
BEGIN
    st_date := SYSTIMESTAMP;
    ed_date := TO_TIMESTAMP_TZ('2013-06-18 11:00:00 -5:00','YYYY-MM-DD HH:MI:SS');
    SYS.XS_PRINCIPAL.CREATE_ROLE
        (name => 'app_regular_role',
        enabled => TRUE,
        start_date => st_date,
        end_date => ed_date);
END;
```

Dynamic Application Role: (gets enabled dynamical based on the authenatication state)

```
BEGIN
    SYS.XS_PRINCIPAL.CREATE_DYNAMIC_ROLE
        (name => 'app_dynamic_role',
        duration => 40,
        scope => XS_PRINCIPAL.SESSION_SCOPE);
END;
```

Predefined Application Roles:

Regular:

- `XSPUBLIC`
- `XSBYPASS`
- `XSSESSIONADMIN`
- `XSNAMESPACEADMIN`
- `XSPROVISIONER`
- `XSCACHEADMIN`
- `XSDISPATCHER`

Dynamic: (depended on the authentication state of application user)

- `DBMS_AUTH`: (direct-logon or other database authentication method)
- `EXTERNAL_DBMS_AUTH`: (direct-logon or other database authentication method and user is external)
- `DBMS_PASSWD`: (direct-logon with password)
- `MIDTIER_AUTH`: (authentication through middle tier application)
- `XSAUTHENTICATED`: (direct or middle tier application)
- `XSSWITCH`: (user switched from proxy user to application user)

Chapter 43: Assignments model and language

Section 43.1: Assignments model in PL/SQL

All programming languages allow us to assign values to variables. Usually, a value is assigned to variable, standing on left side. The prototype of the overall assignment operations in any contemporary programming language looks like this:

```
left_operand assignment_operand right_operand instructions_of_stop
```

This will assign right operand to the left operand. In PL/SQL this operation looks like this:

```
left_operand := right_operand;
```

Left operand **must be always a variable**. Right operand can be value, variable or function:

```
SET serveroutput ON
DECLARE
  v_hello1 VARCHAR2(32767);
  v_hello2 VARCHAR2(32767);
  v_hello3 VARCHAR2(32767);
  FUNCTION hello RETURN VARCHAR2 IS BEGIN RETURN 'Hello from a function!'; END;
BEGIN
   -- from a value (string literal)
  v_hello1 := 'Hello from a value!';
   -- from variable
  v_hello2 := v_hello1;
  -- from function
  v_hello3 := hello;

  DBMS_OUTPUT.put_line(v_hello1);
  DBMS_OUTPUT.put_line(v_hello2);
  DBMS_OUTPUT.put_line(v_hello3);
END;
/
```

When the code block is executed in SQL*Plus the following output is printed in console:

```
Hello from a value!
Hello from a value!
Hello from a function!
```

There is a feature in PL/SQL that allow us to assign "from right to the left". It's possible to do in SELECT INTO statement. Prototype of this instrunction you will find below:

```
SELECT [ literal | column_value ]

INTO local_variable

FROM [ table_name | aliastable_name ]

WHERE comparison_instructions;
```

This code will assign character literal to a local variable:

```
SET serveroutput ON
DECLARE
  v_hello VARCHAR2(32767);
BEGIN
  SELECT 'Hello world!'
  INTO v_hello
  FROM dual;

  DBMS_OUTPUT.put_line(v_hello);
END;
/
```

When the code block is executed in SQL*Plus the following output is printed in console:

```
Hello world!
```

Asignment "from right to the left" **is not a standard**, but it's valuable feature for programmers and users. Generally it's used when programmer is using cursors in PL/SQL - this technique is used, when we want to return a single scalar value or set of columns in the one line of cursor from SQL cursor.

Further Reading:

- Assigning Values to Variables

Chapter 44: Triggers

Introduction:

Triggers are a useful concept in PL/SQL. A trigger is a special type of stored procedure which does not require to be explicitly called by the user. It is a group of instructions, which is automatically fired in response to a specific data modification action on a specific table or relation, or when certain specified conditions are satisfied. Triggers help maintain the integrity, and security of data. They make the job convenient by taking the required action automatically.

Section 44.1: Before INSERT or UPDATE trigger

```
CREATE OR REPLACE TRIGGER CORE_MANUAL_BIUR
  BEFORE INSERT OR UPDATE ON CORE_MANUAL
  FOR EACH ROW
BEGIN
  IF inserting THEN
    -- only set the current date if it is not specified
    IF :NEW.created IS NULL THEN
      :NEW.created := SYSDATE;
    END IF;
  END IF;

  -- always set the modified date to now
  IF inserting OR updating THEN
    :NEW.modified := SYSDATE;
  END IF;
END;
/
```

```
  EXECUTE IMMEDIATE query_text USING VALUE, id;
END;
/
```

Section 45.4: Execute DDL statement

This code creates the table:

```
BEGIN
  EXECUTE IMMEDIATE 'create table my_table (id number, column_value varchar2(100))';
END;
/
```

Section 45.5: Execute anonymous block

You can execute anonymous block. This example shows also how to return value from dynamic SQL:

```
DECLARE
  query_text VARCHAR2(1000) := 'begin :P_OUT := cos(:P_IN); end;';
  in_value NUMBER := 0;
  out_value NUMBER;
BEGIN
  EXECUTE IMMEDIATE query_text USING OUT out_value, IN in_value;
  DBMS_OUTPUT.put_line('Result of anonymous block: ' || TO_CHAR(out_value));
END;
/
```

Chapter 45: Dynamic SQL

Dynamic SQL allows you to assemble an SQL query code in the runtime. This technique has some disadvantages and have to be used very carefully. At the same time, it allows you to implement more complex logic. PL/SQL requires that all objects, used in the code, have to exist and to be valid at compilation time. That's why you can't execute DDL statements in PL/SQL directly, but dynamic SQL allows you to do that.

Section 45.1: Select value with dynamic SQL

Let's say a user wants to select data from different tables. A table is specified by the user.

```
FUNCTION get_value(p_table_name VARCHAR2, p_id NUMBER) RETURN VARCHAR2 IS
   VALUE VARCHAR2(100);
 BEGIN
   EXECUTE IMMEDIATE 'select column_value from ' || p_table_name ||
                     ' where id = :P' INTO VALUE USING p_id;
   RETURN VALUE;
 END;
```

Call this function as usual:

```
DECLARE
  table_name VARCHAR2(30) := 'my_table';
  id NUMBER := 1;
BEGIN
  DBMS_OUTPUT.put_line(get_value(table_name, id));
END;
```

Table to test:

```
CREATE TABLE my_table (id NUMBER, column_value VARCHAR2(100));
INSERT INTO my_table VALUES (1, 'Hello, world!');
```

Section 45.2: Insert values in dynamic SQL

Example below inserts value into the table from the previous example:

```
DECLARE
  query_text VARCHAR2(1000) := 'insert into my_table(id, column_value) values (:P_ID, :P_VAL)';
  id NUMBER := 2;
  VALUE VARCHAR2(100) := 'Bonjour!';
BEGIN
  EXECUTE IMMEDIATE query_text USING id, VALUE;
END;
/
```

Section 45.3: Update values in dynamic SQL

Let's update table from the first example:

```
DECLARE
  query_text VARCHAR2(1000) := 'update my_table set column_value = :P_VAL where id = :P_ID';
  id NUMBER := 2;
  VALUE VARCHAR2(100) := 'Bonjour le monde!';
BEGIN
```