LEARNING
Oracle Database

#oracle

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: oracle-database

It is an unofficial and free Oracle Database ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Oracle Database.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Oracle Database

Remarks

Oracle is a relational database management system (RDBMS) originally built by Larry Ellison, Bob Miner, and Ed Oates in the late 70s. It was intended to be compatible with IBM's System R.

Versions

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

Examples

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'.

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';
```

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;
```

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;
```

Read Getting started with Oracle Database online: https://riptutorial.com/oracle/topic/558/getting-started-with-oracle-database

Chapter 2: Anonymous PL/SQL Block

Remarks

Since they are unnamed, anonymous blocks cannot be referenced by other program units.

Examples

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;
/
```

Read Anonymous PL/SQL Block online: https://riptutorial.com/oracle/topic/6451/anonymous-pl-sql-block

Chapter 3: Autonomous Transactions

Remarks

Typical use cases for autonomous transaction are.

1. For building any kind of logging framework like the error logging framework explained in the above example.
2. For auditing DML operations in triggers on tables irrespective of the final status of the transaction (COMMIT or ROLLBACK).

Examples

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 PLSQL 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
```

Read Autonomous Transactions online: https://riptutorial.com/oracle/topic/6103/autonomous-transactions

Chapter 4: constraints

Examples

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;
```

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;
```

Read constraints online: https://riptutorial.com/oracle/topic/6040/constraints

Chapter 5: Creating a Context

Syntax

- CREATE [OR REPLACE] CONTEXT namespace USING [schema.]package;
- CREATE [OR REPLACE] CONTEXT namespace USING [schema.]package INITIALIZED EXTERNALLY;
- CREATE [OR REPLACE] CONTEXT namespace USING [schema.]package INITIALIZED GLOBALLY;
- CREATE [OR REPLACE] CONTEXT namespace USING [schema.]package ACCESSED GLOBALLY;

Parameters

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.

Remarks

Oracle documentation (12cR1): http://docs.oracle.com/database/121/SQLRF/statements_5003.htm

Examples

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
```

Read Creating a Context online: https://riptutorial.com/oracle/topic/2088/creating-a-context

Chapter 6: Data Dictionary

Remarks

The data dictionary views, also known as catalog views, let you monitor the state of the database in real time:

The views prefixed with USER_, ALL_, and DBA_, show information about schema objects that are owned by you (USER_), accessible by you (ALL_) or accessible by a user with SYSDBA privilege (DBA_). For example, the view ALL_TABLES shows all tables that you have privileges on.

The V$ views show performance-related information.

The _PRIVS views show privilege information for different combinations of users, roles, and objects.

Oracle documentation: Catalog Views / Data Dictionary Views

Examples

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
```

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
```

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)
```

Oracle version

```
select *
from v$version
```

Describes all objects in the database.

```
select *
from dba_objects
```

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

```
select * from dict
```

Read Data Dictionary online: https://riptutorial.com/oracle/topic/7347/data-dictionary

Chapter 7: Data Pump

Introduction

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

Examples

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

Step 3/6 : Create directory

```
create or replace directory DATAPUMP_REMOTE_DIR as '/oracle/scripts/expimp';
```

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
```

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>
```

---

1. 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

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>
```

Read Data Pump online: https://riptutorial.com/oracle/topic/9391/data-pump

Chapter 8: Database Links

Examples

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

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.

Read Database Links online: https://riptutorial.com/oracle/topic/3859/database-links

Chapter 9: Dates

Examples

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.)

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;
```

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 )`.

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
```

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.

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.

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
);
```

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.

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
```

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
```

Leap Seconds

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

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
```

Read Dates online: https://riptutorial.com/oracle/topic/2087/dates

Chapter 10: Delimiting keywords or special characters

Examples

Delimit the table or column name with special characters

Select * from firm's_address;

Select * from "firm's_address";

Delimiting table or column name which is a reserved word as well

Say you have a table named table or you want to create a table with name which is also a keyword, You have to include the name table in pair of double quotes "table"

Select * from table; Above query will fail with syntax error, where as below query will run fine.

Select * from "table";

Read Delimiting keywords or special characters online: https://riptutorial.com/oracle/topic/6553/delimiting-keywords-or-special-characters

Chapter 11: Different ways to update records

Syntax

- UPDATE table-Name [[AS] correlation-Name] SET column-Name = Value [, column-Name = Value}]* [WHERE clause]
- UPDATE table-Name SET column-Name = Value [, column-Name = Value]* WHERE CURRENT OF

Examples

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);
```

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;
```

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';
```

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);

--

select * from table01 a
where
       not exists
       (
         select 'x' from table02 b where a.code = b.codetable01
       );

select * from table01 a where a.code not in (select b.codetable01 from table02 b);
```

Read Different ways to update records online: https://riptutorial.com/oracle/topic/4193/different-ways-to-update-records

Chapter 12: DUAL table

Remarks

`DUAL` table has one column `DUMMY`, defined to be `VARCHAR2(1)` and only one row with a value `x`.

`DUAL` table is automatically created in `SYS` schema when database is created. You can access it from any schema.

You can not change `DUAL` table.

You can use `DUAL` table to call any function from SQL statement. It is useful because it has only one row and oracle optimizer knows everything about it.

Examples

The following example returns the current operating system date and time

```
select sysdate from dual
```

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
```

Read DUAL table online: https://riptutorial.com/oracle/topic/7328/dual-table

Chapter 13: Dynamic SQL

Introduction

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.

Remarks

Some important remarks:

1. Never use string concatenation to add values to query, use parameters instead. This is wrong:

   ```
   execute immediate 'select value from my_table where id = ' ||
        id_valiable into result_variable;
   ```

   And this is right:

   ```
   execute immediate 'select value from my_table where id = :P '
       using id_valiable into result_variable;
   ```

   There are two reasons for this. The first is the security. String concatenation allows to make SQL injection. In the query below, if a variable will contain value `1 or 1 = 1`, the `UPDATE` statement will update all lines in the table:

   ```
   execute immediate 'update my_table set value = ''I have bad news for you'' where id = '
   || id;
   ```

   The second reason is performance. Oracle will parse query without parameters every time when it executes, while query with parameter will be parsed only once in the session.

2. Note, that when the database engine executes a DDL statement, it executes implicit commit before.

Examples

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!');
```

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;
/
```

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
  execute immediate query_text using value, id;
end;
/
```

Execute DDL statement

This code creates the table:

```
begin
  execute immediate 'create table my_table (id number, column_value varchar2(100))';
end;
/
```

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;
/
```

Read Dynamic SQL online: https://riptutorial.com/oracle/topic/10905/dynamic-sql

Chapter 14: Error logging

Examples

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;
```

Read Error logging online: https://riptutorial.com/oracle/topic/3505/error-logging

Chapter 15: Handling NULL values

Introduction

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.

Remarks

NULL can't appear in columns restricted by a PRIMARY KEY or a NOT NULL constraint. (Exception is a new constraint with NOVALIDATE clause)

Examples

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)

Empty strings are NULL

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

A	B
1	(null)

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

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

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

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.

Read Handling NULL values online: https://riptutorial.com/oracle/topic/8183/handling-null-values

Chapter 16: Hierarchical Retrieval With Oracle Database 12C

Introduction

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

Examples

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.

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;
```

Read Hierarchical Retrieval With Oracle Database 12C online: https://riptutorial.com/oracle/topic/8777/hierarchical-retrieval-with-oracle-database-12c

Chapter 17: Hints

Parameters

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.

Examples

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.

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
```

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;
```

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
```

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 |
--------------------------------------------------------------------------------
```

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.

Read Hints online: https://riptutorial.com/oracle/topic/1490/hints

Chapter 18: Indexes

Introduction

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

Examples

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.

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.

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.

Read Indexes online: https://riptutorial.com/oracle/topic/9978/indexes

Chapter 19: JOINS

Examples

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  |
+-------+--------+
```

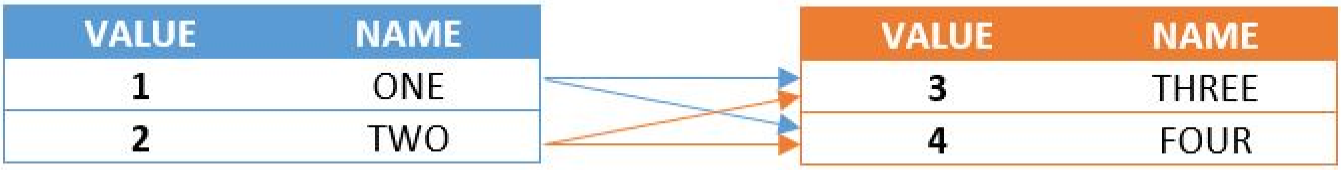
Now, If you execute the query:

```
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

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
```

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    |         |              |
+-----------+---------+---------+--------------+
```

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;
```

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.

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.

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.

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

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
```

Read JOINS online: https://riptutorial.com/oracle/topic/4192/joins

Chapter 20: level query

Remarks

level clause is responsible for generating N number of dummy records based on some specific condition.

Examples

Generate N Number of records

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

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

Read level query online: https://riptutorial.com/oracle/topic/6548/level-query

Chapter 21: Limiting the rows returned by a query (Pagination)

Examples

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;
```

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 ;
```

this way we can paginate the table data , just like web serch page

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
```

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`.

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
```

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
```

Read Limiting the rows returned by a query (Pagination) online: https://riptutorial.com/oracle/topic/4300/limiting-the-rows-returned-by-a-query--pagination-

Chapter 22: Oracle Advanced Queuing (AQ)

Remarks

- Never use DDL or DML against tables created by `dbms_aqadm.create_queue_table`. Only use dbms_aqadm and dbms_aq to work with these tables. Oracle may make several supporting tables, indexes, etc that you will not be aware of. Manually running DDL or DML against the table may lead you to a scenario where Oracle Support will need you to drop and recreate the table and queues to resolve the situation.

- It's strongly recommended you do not use `dbms_aq.forever` for a wait option. This has caused issues in the past as Oracle may start scheduling an excessive number of worker jobs to work the queues that are unnecessary (see Oracle Doc ID 2001165.1).

- It's recommended you do not set the AQ_TM_PROCESSES parameter in version 10.1 and later. Especially avoid setting this to zero since this will disable the QMON background job that is necessary to maintain the queues. You can reset this value to the Oracle default using the following command and restarting the database. `alter system reset aq_tm_processes scope=spfile sid='*';`

Examples

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_l      dbms_aq.enqueue_options_t;
   message_properties_l   dbms_aq.message_properties_t;
   message_id_l           raw (16);
   message_l              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_l := new message_t ( 'Jon', 'Hello, world!' );
   dbms_aq.enqueue (queue_name           => message_worker_pkg.queue_name_c,
                    enqueue_options      => enqueue_options_l,
                    message_properties   => message_properties_l,
                    payload              => message_l,
                    msgid                => message_id_l);
   commit;
```

```
end;
```

Read Oracle Advanced Queuing (AQ) online: https://riptutorial.com/oracle/topic/4362/oracle-advanced-queuing--aq-

Chapter 23: Oracle MAF

Examples

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

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

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

To call a javaScript function

```

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

"function" is the desired js function to be invoked

Read Oracle MAF online: https://riptutorial.com/oracle/topic/6352/oracle-maf

Chapter 24: Real Application Security

Introduction

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

Examples

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)

Read Real Application Security online: https://riptutorial.com/oracle/topic/10864/real-application-security

Chapter 25: Recursive Sub-Query Factoring using the WITH Clause (A.K.A. Common Table Expressions)

Remarks

Recursive sub-query factoring is available in Oracle 11g R2.

Examples

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:

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

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
```

Read Recursive Sub-Query Factoring using the WITH Clause (A.K.A. Common Table Expressions) online: https://riptutorial.com/oracle/topic/3506/recursive-sub-query-factoring-using-the-with-clause--a-k-a--common-table-expressions-

Chapter 26: Sequences

Syntax

- CREATE SEQUENCE SCHEMA.SEQUENCE { INCREMENT BY INTEGER | START WITH INTEGER | MAXVALUE INTEGER | NOMAXVALUE INTEGER | MINVALUE INTEGER | NOMINVALUE INTEGER | CYCLE INTEGER | NOCYCLE INTEGER | CACHE | NOCACHE | ORDER | NOODER }

Parameters

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

Examples

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
NOCYCLE;
```

The first reference to customers_seq.nextval returns 1000. The second returns 1001. Each subsequent reference will return a value 1 greater than the previous reference.

Read Sequences online: https://riptutorial.com/oracle/topic/3709/sequences

Chapter 27: Splitting Delimited Strings

Examples

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
```

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
```

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
```

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
```

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
```

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
```

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
```

Read Splitting Delimited Strings online: https://riptutorial.com/oracle/topic/1968/splitting-delimited-strings

Chapter 28: Statistical functions

Examples

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
```

VARIANCE

Variance measures how far a set numbers is spread out from it's mean. From practical perspective it is squared distance from its mean (center) - the bigger the number the farther the point is.

The following example would return variance of salary values

```
SELECT name, salary, VARIANCE(salary) "Variance"
FROM employees
```

STDDEV

STDDEV returns the sample standard deviation of expr, a set of numbers. You can use it as both an aggregate and analytic function. It differs from STDDEV_SAMP in that STDDEV returns zero when it has only 1 row of input data, whereas STDDEV_SAMP returns null.

Oracle Database calculates the standard deviation as the square root of the variance defined for the VARIANCE aggregate function.

This function takes as an argument any numeric datatype or any nonnumeric datatype that can be implicitly converted to a numeric datatype. The function returns the same datatype as the numeric datatype of the argument.

If you specify DISTINCT, then you can specify only the query_partition_clause of the analytic_clause. The order_by_clause and windowing_clause are not allowed.

The following example returns the standard deviation of the salaries in the sample **hr.employees** table:

Where hr is Schema and employees is a table name.

```
SELECT STDDEV(salary) "Deviation"
FROM employees;

Deviation
----------
3909.36575
```

The query in the following example returns the cumulative standard deviation of the salaries in Department 80 in the sample table hr.employees, ordered by hire_date:

```
SELECT last_name, salary,
STDDEV(salary) OVER (ORDER BY hire_date) "StdDev"
FROM employees
WHERE department_id = 30;

LAST_NAME                     SALARY     StdDev
------------------------- ---------- ----------
Raphaely                       11000          0
Khoo                            3100 5586.14357
Tobias                          2800  4650.0896
```

Read Statistical functions online: https://riptutorial.com/oracle/topic/2283/statistical-functions

Chapter 29: String Manipulation

Examples

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.
```

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!
```

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!
```

LOWER

LOWER converts all uppercase letters in a string to lowercase.

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

Outputs:

text
hello world123!

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;
```

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)`

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
```

Read String Manipulation online: https://riptutorial.com/oracle/topic/1518/string-manipulation

Chapter 30: Table partitioning

Introduction

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

Remarks

Partitioning is an extra cost option and only available for the Enterprise Edition.

Examples

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.

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)
);
```

Select existing partitions

Check existing partitions on Schema

```
SELECT * FROM user_tab_partitions;
```

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)
);
```

Drop partition

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

Select data from a partition

Select data from a partition

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

Truncate a partition

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

Rename a partition

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

Move partition to different tablespace

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

Add new partition

```
ALTER TABLE table_name
```

```
ADD PARTITION new_partition VALUES LESS THAN(400);
```

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)
```

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
```

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")

Read Table partitioning online: https://riptutorial.com/oracle/topic/3955/table-partitioning

Chapter 31: Update with Joins

Introduction

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.

Examples

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.

Read Update with Joins online: https://riptutorial.com/oracle/topic/8061/update-with-joins

Chapter 32: Window Functions

Syntax

- Ratio_To_Report (expr) OVER (query_partition_clause)

Examples

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
```

Read Window Functions online: https://riptutorial.com/oracle/topic/6669/window-functions

Chapter 33: Working with Dates

Examples

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;
```

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 substract 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

Read Working with Dates online: https://riptutorial.com/oracle/topic/768/working-with-dates

Credits

S. No	Chapters	Contributors
1	Getting started with Oracle Database	Community, J. Chomel, Jeffrey Kemp, Jon Ericson, Kevin Montrose, Mark Stewart, Sanjay Radadiya, Steven Feuerstein, tonirush
2	Anonymous PL/SQL Block	Jon Heller, Skynet, Zohar Elkayam
3	Autonomous Transactions	phonetic_man
4	constraints	SSD
5	Creating a Context	Jeffrey Kemp
6	Data Dictionary	Mark Stewart, Pancho, Slava Babin
7	Data Pump	Vidya Thotangare
8	Database Links	carlosb, Daniel Langemann, g00dy, kasi
9	Dates	carlosb, MT0, Roman, tonirush
10	Delimiting keywords or special characters	dev
11	Different ways to update records	nimour pristou, Nogueira Jr, SriniV
12	DUAL table	Slava Babin
13	Dynamic SQL	Dmitry
14	Error logging	zygimantus
15	Handling NULL values	Dalex, JeromeFr
16	Hierarchical Retrieval With Oracle Database 12C	Muntasir, Vahid
17	Hints	Aleksej, Florin Ghita, Jon Heller, Mark Stewart, Pirate X

18	Indexes	smshafiqulislam
19	JOINS	Aleksej, B Samedi, Bakhtiar Hasan, Daniel Langemann, Erkan Haspulat, Pranav Shah, Robin James, SriniV, Sumner Evans
20	level query	Sanjay Radadiya, TechEnthusiast
21	Limiting the rows returned by a query (Pagination)	Ahmed Mohamed, Martin Schapendonk, Matas Vaitkevicius, Sanjay Radadiya, tonirush, trincot
22	Oracle Advanced Queuing (AQ)	Jon Theriault
23	Oracle MAF	Anand Raj
24	Real Application Security	Ben H
25	Recursive Sub-Query Factoring using the WITH Clause (A.K.A. Common Table Expressions)	B Samedi, MT0
26	Sequences	Pranav Shah, SriniV
27	Splitting Delimited Strings	Arkadiusz Łukasiewicz, MT0
28	Statistical functions	Evgeniy K., Matas Vaitkevicius, ppeterka, Pranav Shah
29	String Manipulation	carlosb, Eric B., Florin Ghita, Francesco Serra, J. Chomel, J.Hudler, Jeffrey Kemp, Mark Stewart, SriniV, Thunder, walen, zhliu03
30	Table partitioning	BobC, carlosb, ivanzg, JeromeFr, Kamil Islamov, Stephen Leppik, tonirush
31	Update with Joins	mathguy
32	Window Functions	Leigh Riffel
33	Working with Dates	David Aldridge, Florin Ghita, Jeffrey Kemp, Mark Stewart, tonirush, zygimantus