SGBD: importer ou exporter les données, programmes.

Contenu du chapitre:

- Importer et exporter les données
- Accéder à des sources externes de données
- UDF: fonctions définies par l'utilisateur. Procédures et triggers sous postgres
- PL/SQL: les procédures sous Oracle
- Interagir avec une BD depuis un programme

Table des matières

Importer des données depuis un fichier

INSERT sert à insérer les tuples 1 par 1.

En présence de grandes quantités de données (pour charger un .csv) on fait appel à des outils de "chargement de masse" spécifiques à chaque SGBD.

```
-- PostgreSQL (COPY côté serveur, \copy côté client):
COPY Prog FROM '/user1/cinema_data.csv';
-- ou l'instruction côté client de psql:
-- attention: écrire l'instruction sur 1 ligne, sans ';' à la fin
\copy zip_codes FROM '/path/to/csv/ZIP_CODES.txt' DELIMITER ',' CSV

-- MariaDB and MySQL:
LOAD DATA INFILE '/user1/cinema_data.csv' INTO Prog;
--    ou mysqlimport pour une version multithread.

-- Oracle:
--    on peut utiliser sqlldr ou des external tables

-- SQL Server:
BULK INSERT dbname.Prog FROM  '/user1/cinema_data.csv';
```

Remarque: vous n'avez souvent (ex: au PUIO) pas les droits pour effectuer COPY (et COPY impose de placer le fichier sur serveur), mais vous pouvez faire \copy.

Importer un csv sous oracle: sqlldr

```
-- table schema: create_tables.sql:
CREATE TABLE codesPostaux (
insee varchar2(6),
nom_commune varchar2(50)
);
```

```
-- in the database (ex: from sql*plus)
SQL > @create_tables.sql;
```

```
-- control file control.txt:
LOAD DATA INFILE 'codes_postaux.csv'
TRUNCATE
INTO TABLE codesPostaux
FIELDS TERMINATED BY ';'
OPTIONALLY ENCLOSED BY '"'
TRAILING NULLCOLS
( insee ,
nom_commune
)
```

```
-- running the control.txt script (from terminal): (change xxx for your login)
jdoe@47 ~ $ sqlldr userid=C##xxxx_a/xxxx_a control=control.txt log=log.txt \
bad=bad.txt  direct=y errors=0 skip=1
```

Exporter des données dans un fichier

```
-- PostgreSQL:
COPY (SELECT Titre, Horaire FROM Prog WHERE Horaire<'20h00')
  TO '/user1/cinema_data.csv';
-- or use client side instruction from psql:
-- attention: écrire l'instruction sur 1 ligne, sans  ';' à la fin
\copy (SELECT Titre, Horaire FROM Prog WHERE Horaire<'20h00')
  TO '/user1/cinema_data.csv' CSV

-- MariaDB and MySQL:
SELECT Titre, Horaire FROM Prog WHERE Horaire<'20h00'
  INTO OUTFILE '/user1/cinema_data.csv';

  -- Oracle: voir ci-après
```

Remarque: vous n'avez pas les droits pour effectuer COPY, mais vous pouvez faire \copy.

Exporter vers un csv sous oracle

1. sous SQL Developer: plutôt facile:

   https://stackoverflow.com/questions/4168398

2. sous sqlplus on peut:
   - écrire directement les instructions SQL et utiliser spool pour écrire dans le fichier
   - ou bien utiliser une procédure PL/SQL pour générer automatiquement les instructions. Pour écrire dans un fichier, on peut utiliser utl_file, ou bien dbms_output et spool

```
-- Export d'une requête vers fichier.csv sous sqlplus: @script_export.sql
-- script_export.sql
set heading off -- si on ne veut pas d'en-tête
set feedback off -- evite le 'xxx rows selected'
set markup csv on -- sur la version 12.1 au PUIO
-- sur anciennes versions on utilisera à la place: set colsep ','  ...
spool fichier.csv
select * from ma_table;
spool off
```


Exporter/Importer la base entière (dump)

Enregistrer dans un fichier une suite d'instructions SQL permettant de recréer la base entière dans son état actuel.

```
-- PostgreSQL:
pg_dump db_name > backup-file.sql
psql db_name < backup-file.sql

/* pg_dump -t ma_table ma_base --data-only --inserts > masauvegarde.sql */

-- MariaDB and MySQL:
mysqldump db_name > backup-file.sql
mysql db_name < backup-file.sql

-- Oracle
-- Utiliser Oracle data pump (instructions: expdp, impdp)
expdp monschema TABLES=products,categ DUMPFILE=dir1:exp1.dmp NOLOGFILE=YES
impdp monschema DUMPFILE=dir1:exp1.dmp
```

Plus d'information sur Oracle data pump:

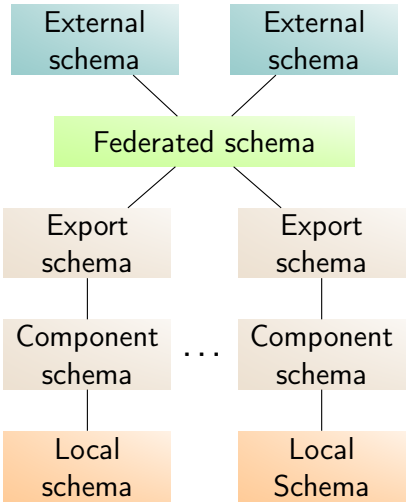
http://jaouad.developpez.com/datapump/

https://docs.oracle.com/database/122/SUTIL/oracle-data-pump.htm#SUTIL2877

Table des matières

SGBD: importer ou exporter les données, programmes.

base de donnée fédérée à 5 niveaux

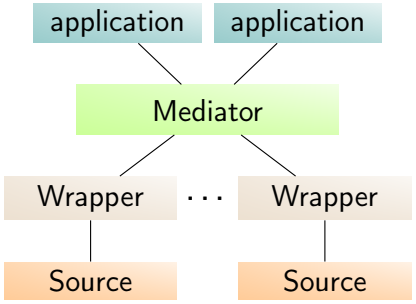


Solution pour offrir un accès commun à différentes sources autonomes.

- schéma logique des sources adapté en schéma de composante (pour résoudre pbs d'hétérogénéité)
- Schéma exporté: portion du schéma de composante (contrôle d'accès. . .)
- Schéma fédéré intègre schéma exporté, prend en compte répartition des données. Peut être grand.
- Schéma externe implémente le contrôle d'accès, simplifie le schéma fédéré, cache les évolutions de schéma.

Fédérées: BD autonome partageant leurs données (2 à 2 : centralisation minimale)

Médiateur-Adaptateur



Solution permettant d'offrir un accès commun à différentes sources hétérogènes indépendentes.

- wrapper est l'intermédiaire entre sources et médiateur; résout pbs d'hétérogénéité (interfaces, modèle de données, sémantique)
- Médiateur intègre les données *mais schéma médiateur dérivé des applications, pas de l'intégration des sources*.

Sources de données externes dans les SGBDs

SAP Hana (SDA)

```
-- create remote access
CREATE REMOTE SOURCE HIVE1 ADAPTER "hiveodbc"
CONFIGURATION 'DSN=hive1'
WITH CREDENTIAL TYPE 'PASSWORD' USING
'user=dfuser;password=dfpass';

-- wrap remote source as virtual table
CREATE VIRTUAL TABLE "VIRTUAL_PRODUCT"
AT "HIVE1"."dflo"."dflo"."product";

-- query the data
SELECT product_name, brand_name
FROM "VIRTUAL_PRODUCT";
```

[https://openproceedings.org/2015/conf/edbt/paper-339.pdf]

Part of query execution plan can be forwarded to remote source.
Choices may impact performance.

Sources de données externes dans les SGBDs (2)

PostgreSQL

```
CREATE EXTENSION postgres_fdw; -- install extension

-- create remote access
CREATE SERVER pg_serv1
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'foo', dbname 'foodb', port '5432');

-- -- authentication credentials
CREATE USER MAPPING FOR CURRENT_USER
  SERVER pg_serv1
  OPTIONS (user 'jdoe', password 'secret1');

-- import distant schema
CREATE SCHEMA app;

IMPORT FOREIGN SCHEMA public -- or CREATE FOREIGN TABLE  vp (...) SERVER pg_serv1
  FROM SERVER pg_serv1
  INTO app;

-- query the data
SELECT product_name, brand_name
FROM app.VIRTUAL_PRODUCT;
```

[https://wiki.postgresql.org/wiki/Foreign_data_wrappers]
[https://www.postgresql.org/docs/current/postgres-fdw.html]

Table des matières

SGBD: importer ou exporter les données, programmes.

- Importer et exporter les données
- Accéder à des sources externes de données
- UDF: fonctions définies par l'utilisateur. Procédures et triggers sous postgres
- PL/SQL: les procédures sous Oracle
- Interagir avec une BD depuis un programme

Procédures

La plupart des SGBD permettent de définir des procédures.
Procédure : ensemble d'instructions, stockée dans le serveur BD.
on parle parfois de *procédure stockée* (stored procedure, or UDF)

Idée: manipuler les données (tables) dans un langage de script procédural. Ce langage de script inclut les requêtes SQL, mais ajoute les fonctionalités d'un langage impératif séquentiel: structures de contrôle (IF, boucles...) et variables.

PL/(pg)SQL utilise essentiellement

- les types atomiques,
- le type RECORD,
- et des curseurs pour parcourir le résultat d'une requête SQL.

Les scripts (fonctions, procédures) sont stockés dans le SGBD comme les autres objets. ↪ manipulés par LDD (CREATE, DROP...)

Procédures

✔ on peut donner à l'utilisateur le droit d'exécuter une procédure mais lui interdire d'accéder directement aux tables.

✔ parfois un gain d'efficacité comme la requête est stockée sur le serveur (élimine des A/R client-serveur).

✔ "portable" au sens où interne au SGBD, donc indépendant de l'OS.

✔ modularité: grouper les instructions facilite réutilisation.

✘ **pas "portable": procédures varient beaucoup avec le SGBD.**

Oracle et IBM DB2: PL/SQL
PostgreSQL : PL/pgSQL
Microsoft : Transact-SQL
syntaxe similaire pour MariaDB, MySQL.

Procédures: structure générale

```
CREATE FUNCTION mafonction(arg1 int, arg2 text, arg3 int) RETURNS integer
AS $$corps de la fonction$$
LANGUAGE plpgsql;
```

```
-- le corps étant de la forme:
[ DECLARE
    declarations de variables]
BEGIN
    instructions
END
```

```
DO $$ -- pour un bloc anonyme (hors fonction)
BEGIN
  FOR counter IN 1..5 LOOP
    RAISE NOTICE 'Counter: %', counter;
  END LOOP;
END; $$;
```

- L'instruction CREATE FUNCTION déclare les arguments et type de retour.
- les autres variables dans le corps doivent être déclarées dans le bloc DECLARE.
- Pour PostgreSQL, le corps de la fonction est du texte. L'entourer de $$ au lieu de guillemets ('corps de la fonction') évite d'échapper tout guillemet dans ce texte.

Utilisation (pour une fonction scalaire):

- On peut exécuter un bloc anonyme directement.
- SELECT mafonction(2, 'aa', 10);
- dans une autre procédure

Procédures : utilisation comme routine dans une procédure

```
CREATE FUNCTION date_jour() RETURNS char(10)
AS '
BEGIN
RETURN CAST(CURRENT_DATE AS CHAR(10));
END
'
LANGUAGE plpgsql;
```

```
-- table User(userid: int, username: text)
CREATE FUNCTION date_jour_heure() RETURNS char(26)
AS '
BEGIN
RETURN date_jour() || '' '' ||CAST(CURRENT_TIME AS CHAR(15));
END
'
LANGUAGE plpgsql;
```


Procédures : affectations, conditions

```
CREATE FUNCTION prix_ttc(prixHT real, category text) RETURNS real
AS $$
DECLARE
taux_normal real := .2;
taux_reduit real := .1;
prix real := prixHT;
BEGIN
  IF category = 'alimentaire' OR category = 'livre' THEN
    prix := prixHT * (1+taux_reduit);
  ELSE
    prix := prixHT * (1+taux_normal);
  END IF;
  -- on aurait aussi pu utiliser CASE WHEN condition THEN ... ELSE ... END CASE
  RETURN prix;
END
$$ LANGUAGE plpgsql;
```


Procédures : paramètres IN et OUT

```
CREATE OR REPLACE FUNCTION benefices(prix_coutant real, prix_vente real,
  deduction real, OUT benefice_avant_promo real, OUT benefice real)
AS $$
DECLARE
BEGIN
  benefice_avant_promo := prix_vente - prix_coutant;
  benefice := benefice_avant_promo - deduction;
END
$$ LANGUAGE plpgsql;
-- OUT permet plusieurs retours. Pour fonction sans aucun retour: RETURNS void
```

Les paramètres IN (défaut) sont les arguments passés en entrées, les paramètres de sortie (OUT) servent à retourner les valeurs calculées:

```
SELECT * FROM benefices(8, 10.5, 0.5);
```

```
 benefice_avant_promo | benefice
----------------------+----------
                  2.5 |        2
(1 row)
```

Une fonction pgSQL peut ainsi avoir 0,1, ou plusieurs paramètres en entrée comme en sortie.

Procédures : SELECT INTO

```
-- table User(userid: int, username: text)
CREATE FUNCTION get_userid(username text) RETURNS int
AS $$
DECLARE
userid int;
BEGIN
  SELECT users.userid INTO userid
    FROM users WHERE users.username = get_userid.username;
  RETURN userid;
END
$$ LANGUAGE plpgsql;
```

La requête SELECT devrait retourner un seul résultat (pas 0 ni plusieurs).

Si vous êtes curieux de savoir ce qui se passe sinon:

https://www.postgresql.org/docs/current/static/plpgsql-statements.html#plpgsql-statements-sql-onerow

Procédures : exceptions

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN -- same as: WHEN SQLSTATE '22012' THEN
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;
```

https://www.postgresql.org/docs/current/plpgsql-control-structures.html#PLPGSQL-ERROR-TRAPPING

Procédures : exceptions

```
CREATE FUNCTION fusionne_base(cle INT, donnee TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        -- commençons par tenter la mise à jour de la clé
        UPDATE base SET b = donnee WHERE a = cle;
        IF found THEN
            RETURN;
        END IF;

        -- si elle n'est pas dispo, tentons l'insertion de la clé
        -- si quelqu'un essaie d'insérer la même clé en même temps,
        -- il y aura une erreur pour violation de clé unique
        BEGIN
            INSERT INTO base(a,b) VALUES (cle, donnee);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- ne rien faire, et tente de nouveau la mise à jour
        END;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

https://www.postgresql.org/docs/current/plpgsql-control-structures.html#PLPGSQL-ERROR-TRAPPING

Procédures : types déduits des données

```
CREATE FUNCTION prix_a_payer() RETURNS ventes.prix%TYPE
AS $$
DECLARE
prixTTC ventes.prix%TYPE;
prixTTC2 prixTTC%TYPE;
r ventes%ROWTYPE;
BEGIN
  SELECT * INTO r FROM ventes WHERE id=135;
  prixTTC := r.prixHT * 1.1;
  prixTTC2 := r.prixHT * 1.05;
  RETURN prixTTC - r.reduction;
END
$$ LANGUAGE plpgsql;
```


Procédures : les variables de type curseur

Sert à manipuler (parcourir) un ensemble de lignes: le résultat d'un SELECT.

L'utilisation d'un curseur se déroule en 4 étapes:

0. déclarer le curseur
1. ouvrir le curseur (instruction OPEN)

   Construit le plan d'exécution, et initialise le programme. Le résultat qui sera lu par le curseur correspond à l'état de la base à l'instant de cette instruction OPEN.
2. parcourir le curseur (en itérant des instructions FETCH)

   Des mises à jour peuvent être réalisées sur la base entre les FETCH, par le programme exécutant le curseur ou par d'autres. Mais le résultat parcouru par le curseur ne sera pas affecté. Du point de vue du résultat tout se passe comme si le curseur exécutait la requête SELECT au moment du OPEN, même si en pratique le SGBD peut préférer éviter de matérialiser ainsi le résultat, pour pouvoir retourner les premiers tuples plus vite et pour éviter de stocker un résultat volumineux.
3. fermer le curseur (instruction CLOSE)

Procédures : les curseurs

```
-- table Vente (produit int, montant number) ayant 10 lignes
CREATE FUNCTION ca_produit(OUT montant1 real, OUT montant2 real,
  OUT montant_restant real) AS $$
DECLARE
  c CURSOR FOR SELECT produit, montant FROM Vente ORDER BY montant DESC
  nb int := 0;
  prod int := 0;
  valeur_courante real := 0;
BEGIN
montant_restant := 0;
  OPEN c; -- le curseur calcule la requête
    FETCH c INTO prod, montant1; -- enregistre la 1ère valeur de c dans montant1
    FETCH c INTO prod, montant2; -- enregistre la 2è valeur de c dans montant2
    LOOP -- boucle en pgSQL
      EXIT WHEN nb > 10; -- exit quitte la boucle (la plus interne si imbriquée)
      FETCH c INTO prod, valeur_courante; -- récupère la valeur suivante de c
      montant_restant := montant_restant + valeur_courante;
      nb := nb+1;
    END LOOP;
  CLOSE c;
END
$$ LANGUAGE plpgsql;
```


Procédures : les curseurs

```
CREATE OR REPLACE FUNCTION show_cine(refcursor) RETURNS refcursor AS $$
BEGIN -- pas de déclaration: le curseur aura le nom passé en argument ($1)
  OPEN $1 FOR SELECT nom, adresse FROM Cine; -- le curseur récupère le résultat
  RETURN $1;
END;
$$ LANGUAGE plpgsql;
```

```
BEGIN;
SELECT show_cine('mon_nom');
FETCH ALL IN mon_nom;
COMMIT;
```


Renvoyer une table

```
CREATE FUNCTION get_name(a text) RETURNS SETOF integer AS
$BODY$
BEGIN
    RETURN QUERY SELECT id from person WHERE nom = a;
    IF NOT FOUND THEN
        RAISE EXCEPTION '% est inconnu au bataillon', $1;
    END IF;

    RETURN;
 END
$BODY$
LANGUAGE plpgsql;

SELECT * from get_name('Jean');
```

[https://www.postgresql.org/docs/current/plpgsql-control-structures.html#PLPGSQL-STATEMENTS-RETURNING]

Instructions dynamiques : EXECUTE

Principe de SQL dynamique: paramétrer des instructions SQL au niveau du programme.

Ex: définir *lors de l'exécution du programme* les noms de tables ou colonnes affectées (ou les procédures à exécuter, les conditions, le type des variables)...

La chaîne de caractères d'une instruction dynamique est évaluée à l'exécution et non à la compilation.

```
CREATE FUNCTION destructeur_table(nom text) RETURNS void AS $$
BEGIN
  EXECUTE 'DROP TABLE ' || nom;
END;
$$ LANGUAGE plpgsql;
-- une instruction dynamique s'exécute avec EXECUTE
```

⚠ Éviter d'utiliser SQL dynamique lorsque ce n'est pas nécessaire.

Instructions dynamiques

```
-- pg_catalog.pg_tables(schemaname: name, tablename: name, ...) liste les tables
-- quote_ident renvoie la chaîne entre guillemets, et échappe ceux de la chaîne
DO $$ DECLARE
  r RECORD;
BEGIN
  -- si le schéma sur lequel on veut effectuer l'opération n'est pas le schéma cou
  -- il faut remplacer current_schema() par  'nom_du_schema_souhaite'
  -- *et* modifier les instructions 'DROP...' en conséquence.
  FOR r IN (SELECT tablename FROM pg_tables WHERE schemaname = current_schema())
  LOOP
    EXECUTE 'DROP TABLE IF EXISTS ' || quote_ident(r.tablename) || ' CASCADE';
  END LOOP;
END $$;
```

effet assez proche de:

```
DROP SCHEMA public CASCADE;
CREATE SCHEMA public;
GRANT ALL ON SCHEMA public TO postgres;
GRANT ALL ON SCHEMA public TO public;
COMMENT ON SCHEMA public IS 'standard public schema';
```

[https://stackoverflow.com/questions/3327312/drop-all-tables-in-postgresql]

Instructions dynamiques: paramètres de substitution

Instruction compilée une seule fois, avant que les paramètres soient évalués.

- ✔ réduit la vulnérabilité aux injections SQL
- ✔ peut améliorer la performance
- ✔ plus besoin d'échapper les paramètres string

```
-- t doit avoir 2 colonnes (éventuellement plus)
CREATE FUNCTION insert_generique(t text) RETURNS void AS $$
DECLARE
ordre text;
BEGIN
  ordre := 'INSERT INTO ' || quote_ident(t) || ' VALUES ($1, $2)';
  for i in 1..100 loop
    EXECUTE ordre USING i,2*i;
    end loop;
END;$$
LANGUAGE plpgsql;
```

Le paramètre doit être un littéral ($\simeq$ valeur de colonne). Pas nom de table, col...

En fait il est préférable d'utiliser la fonction `format` (voir la doc) pour régler le problème du formattage de la chaîne de caractère t décrivant la table

Droits d'exécution

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
BEGIN
        SELECT  (pwd = $2) INTO passed
        FROM    pwds
        WHERE   username = $1;

        RETURN passed;
END;
$$  LANGUAGE plpgsql
    SECURITY DEFINER
    -- Set a secure search_path: trusted schema(s), then 'pg_temp'.
    SET search_path = admin, pg_temp;
```

With `SECURITY DEFINER`, s'exécute avec les droits de celui qui l'a créé ⚠.

Sinon (par défaut: `SECURITY INVOKER`), s'exécute avec les droits de l'utilisateur courant.

Les déclencheurs (🇬🇧 triggers)

Des procédures se déclenchant lors d'un évènement

```
CREATE TABLE emp (empname text, salary integer, last_date timestamp,
  last_user text);
CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
  BEGIN
    IF NEW.salary < 0 THEN
      RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
    END IF;
    -- Remember who changed the payroll when
    NEW.last_date := current_timestamp;
    NEW.last_user := current_user;
    RETURN NEW;
  END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
  FOR EACH ROW EXECUTE PROCEDURE emp_stamp();

insert into emp values('john',3,null,'aa')
```

```
 empname | salary |          last_date          | last_user
---------+--------+-----------------------------+----------
 john    |      3 | 2020-10-14 00:09:49.546118 | bgroz
```

Les déclencheurs/triggers (triggers), suite

Très expressifs, mais:

⚠ **Impopulaires:**

- Durs à maintenir: pour chaque nouvelle action sur la BD, on devra s'assurer qu'il n'y a pas d'effets de bords liés aux triggers.
- Risque d'incohérences si le trigger effectue des actions qu'on ne peut pas annuler.
- Impact sur la performance dans une grande BD.

Définir des fonctions dans un autre langage: PL/Python

```
-- install extension
CREATE EXTENSION plpython;

CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

– plpython uses default python version on postgres, currently python2 – better use extension plpython2u, plpython3u

[https://www.postgresql.org/docs/current/plpython.html]

Bibliographie

https://www.postgresql.org/docs/current/static/plpgsql.html

https://mariadb.com/kb/en/library/create-procedure/

http://initd.org/psycopg/docs/usage.html

http://sys.bdpedia.fr/files/cbd-sys.pdf

Un peu orienté "Oracle":
https://perso.limsi.fr/anne/cours/

Table des matières

PL/SQL: structure des blocs

Remarque: PL/SQL est un vieux language: 1991. Syntaxe inspirée par Ada (et par conséquent Pascal).

Bloc anonyme

```
[ DECLARE
    declarations de variables]
BEGIN
 -- le corps de la fonction
    instructions
[ EXCEPTIONS
  traitement des erreurs ]
END;
```

```
-- pour que dbms_output.put_line
-- affiche le texte à l'écran:
-- set serveroutput on
DECLARE
  texte VARCHAR2 (40) := 'Hello World!';
BEGIN
  DBMS_OUTPUT.put_line (texte);
END;
/
```

équivalent à:

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


PL/SQL sous sql*plus

Spécificités propres à SQL*Plus:

- `set serveroutput on/off`: will/will not display the ouput generated by package DBMS_OUTPUT
- "/" on a blank line executes whatever is in current buffer (needed after a procedure. After a usual SQL statement, would execute statement twice).
- pour débuguer en cas d'échec à la compilation, tapper: `show errors`

Fonctions (UDF)

```
CREATE OR REPLACE FUNCTION salutation (nom VARCHAR2)
    RETURN VARCHAR2 -- une fonction doit avoir un type retour
IS -- ou  AS :  pas de différence entre les 2
BEGIN
   RETURN 'Hello ' || nom;
END; -- compilation échoue si la BD a un autre objet salutation
```

```
CREATE FUNCTION square(original NUMBER)
RETURN NUMBER
AS
   original_squared NUMBER; -- déclaration de variable
BEGIN
   original_squared := original * original;
   RETURN original_squared;
END;
/
```

Utilisable dans bloc, autre fonction/procédure, programme externe, requête SQL:

```
SELECT * FROM t WHERE column_a = square(column_b);
INSERT INTO t VALUES (1,4,salutation('jean dupont'));
```


Procédures

```
CREATE OR REPLACE PROCEDURE salutation (nom VARCHAR2)
IS -- or AS
BEGIN
   DBMS_OUTPUT.put_line (nom);
END;
```

```
-- parameters have mode (voir utilisation sur le transparent suivant):
-- IN (input - by default), OUT (output) or IN OUT (lecture-écriture)
CREATE PROCEDURE split_name (
   phrase IN VARCHAR2, first OUT VARCHAR2, last OUT VARCHAR2) IS
BEGIN
   first := SUBSTR(phrase, 1, INSTR(phrase, ' ')-1);
   last := SUBSTR(phrase, INSTR(phrase, ' ')+1);
   IF first = 'John' THEN
      DBMS_OUTPUT.PUT_LINE('That is a common first name.');
   END IF;
END;
/
```

↪ une procédure peut avoir 0,1, ou plusieurs paramètres en mode OUT

⚠ Subtilités: `http://www.dba-oracle.com/t_plsql_function_restrictions.htm`

Procédures : utilisation

Utilisation: dans un bloc PL/SQL (bloc anonyme, ou procédure...)

```
EXECUTE DBMS_OUTPUT.put_line('aa' || 'b')
-- EXECUTE ou EXEC est équivalent à:
BEGIN
  DBMS_OUTPUT.put_line('aa' || 'b');
END;
/
```

```
-- prints 'Bonjour M. Girard'
DECLARE
  fullname varchar2(21);
  prenom varchar2(10);
  nom varchar2(10);
BEGIN
  fullname := 'Jean Girard';
  split_name(fullname, prenom, nom);
  DBMS_OUTPUT.put_line('Bonjour ' || 'M. ' || nom);
END;
/
```


Procédures : SELECT INTO

```
-- table concert(id int, id_place int, prix_place DECIMAL(4,2))
CREATE FUNCTION get_nbplace(concert_id int) RETURN int
AS
total int;
BEGIN
  SELECT count(*) INTO total FROM concert c WHERE c.id = concert_id;
  RETURN total;
END;
-- on peut aussi assigner plusieurs variables à la fois:
-- SELECT COUNT(*) into total, MIN(prix_place) INTO prix FROM concert
```

```
SELECT get_nbplace(1) FROM DUAL;
-- revient à:
BEGIN
  DBMS_OUTPUT.put_line(get_nbplace(1));
END;
```

La requête SELECT devrait retourner un seul résultat (pas 0 ni plusieurs).

Procédures : types déduits des données

```
-- table ville(nom varchar2(10), maire varchar2(20),...)
CREATE FUNCTION get_maire(nom IN ville.nom%type) RETURN varchar2
AS
v_maire ville.maire%type; --  évite de fixer la taille
v_complete ville%rowtype; --  évite de fixer les colonnes
BEGIN
  SELECT * INTO v_complete
  FROM ville
  WHERE ville.nom = get_maire.nom; -- we qualify ambiguous parameters
  v_maire := v_complete.maire;
  RETURN v_maire;

  EXCEPTION
  WHEN TOO_MANY_ROWS
    THEN RETURN 'plusieurs maires pour cette ville';
  WHEN NO_DATA_FOUND
    THEN RETURN 'pas de maire pour cette ville';
  WHEN OTHERS
    THEN raise_application_error(
    -20011, 'exception inconnue dans la fonction get_maire'
    );
END;
```


Procédures : les curseurs

Sert à manipuler (parcourir) un ensemble de lignes: le résultat d'un SELECT.

L'utilisation d'un curseur se déroule en 4 étapes:

0. déclarer le curseur
1. ouvrir le curseur (instruction OPEN)

   Construit le plan d'exécution, et initialise le programme. Le résultat qui sera lu par le curseur correspond à l'état de la base à l'instant de cette instruction OPEN.
2. parcourir le curseur (en itérant des instructions FETCH)

   Des mises à jour peuvent être réalisées sur la base entre les FETCH, par le programme exécutant le curseur ou par d'autres. Mais le résultat parcouru par le curseur ne sera pas affecté. Du point de vue du résultat tout se passe comme si le curseur exécutait la requête SELECT au moment du OPEN, même si en pratique le SGBD peut préférer éviter de matérialiser ainsi le résultat, pour pouvoir retourner les premiers tuples plus vite et pour éviter de stocker un résultat volumineux.
3. fermer le curseur (instruction CLOSE)

Procédures : attributs des curseurs (implicites ou explicites)

PL/SQL crée un curseur implicite pour chaque SELECT INTO, UPDATE, ou DELETE.

Attributs des curseurs implicites:

sql%rowcount
#lignes affectées

sql%found
true ssi ≥ 1 ligne affectée

sql%notfound
l'inverse

```
DECLARE
v_maire ville.maire%type;
BEGIN
  UPDATE ville SET maire='Hidalgo' WHERE nom='Paris';
  IF SQL%FOUND THEN
    dbms_output.put_line(SQL%ROWCOUNT);
  ELSIF SQL%NOTFOUND THEN RETURN 'pas de ville sélectionnée';
  END IF;
END;
```

Attributs d'un curseur explicite c:

c%ISOPEN: true ssi curseur est ouvert.

c%rowcount: #tuples parcourus jusque l'état courant
(⚠ nombres de FETCH, pas le nombre d'éléments dans le résultat du SELECT)

c%found: true si le dernier FETCH a renvoyé une ligne

Parcours de curseurs

```
-- table Vente (produit int, montant number) ayant 10 lignes
CREATE PROCEDURE affiche(montant1 OUT NUMBER, montant2 OUT NUMBER,
  montant_restant  OUT NUMBER)
AS
  CURSOR c IS SELECT produit, montant FROM Vente ORDER BY montant DESC;
  valeur_courante NUMBER := 0;
  prod int := 0;
BEGIN
  montant_restant := 0;
  OPEN c; -- le curseur calcule la requête
  FETCH c INTO prod, montant1; -- enregistre la 1ère valeur de c dans montant1
  FETCH c INTO prod, montant2;
  DBMS_OUTPUT.put_line(c%rowcount); -- affiche 2
  LOOP -- boucle en PL/SQL
    FETCH c INTO prod, valeur_courante; -- récupère la valeur suivante de c
    EXIT WHEN c%notfound; -- exit quitte la boucle (la plus interne)
    montant_restant := montant_restant + valeur_courante;
  END LOOP;
  CLOSE c;
END;
```

⚠ *FETCH ne modifie pas montant1 s'il n'y a pas assez de lignes dans vente (et pas d'erreur non plus): utiliser les attributs du curseur si on veut vérifier.*

Type dérivé du curseur

```
-- table Vente (produit int, montant number ... )
CREATE PROCEDURE affiche(montant1 IN OUT NUMBER)
AS
  CURSOR c IS SELECT produit, montant FROM Vente ORDER BY montant DESC;
  ligne c%rowtype; -- type dérivé du curseur
BEGIN
  OPEN c;
  FETCH c INTO ligne;
  montant1 := montant1 + ligne.montant;
  CLOSE c;
END;
```


Curseurs gérés automatiquement par boucle FOR

```
CREATE PROCEDURE affiche_produits AS
  CURSOR c IS SELECT produit, montant FROM Vente ORDER BY montant DESC;
BEGIN
  FOR ligne IN c LOOP
    DBMS_OUTPUT.put_line(ligne.produit || ' , ' || ligne.montant);
  END LOOP;
END;
-- intérêt: pas besoin de déclarer ligne, ni d'écrire les FETCH, OPEN, CLOSE
```

Et même pour un usage simple:

```
CREATE PROCEDURE affiche_produits AS
BEGIN
  FOR ligne IN (SELECT produit, montant FROM Vente ORDER BY montant DESC) LOOP
    DBMS_OUTPUT.put_line(ligne.produit || ' , ' || ligne.montant);
  END LOOP;
END;
```

Cursor variables

```
CREATE FUNCTION liste_produits(montant_max Vente.montant%type)
RETURN SYS_REFCURSOR AS
  c SYS_REFCURSOR; -- crée un curseur dynamique: pas un véritable curseur.
BEGIN
  OPEN c FOR SELECT produit, montant FROM Vente
    WHERE montant < montant_max ORDER BY montant DESC;
  RETURN c;
END;

CREATE PROCEDURE affiche(v_cur IN SYS_REFCURSOR) AS
  v_a   VARCHAR2(10);
  v_b   VARCHAR2(10);
BEGIN
  LOOP
    -- les curseurs dynamiques se parcourent exclusivement par FETCH
    -- on ne peut pas utiliser la version 'automatique' par boucle 'FOR'
    FETCH v_cur INTO v_a, v_b;
    EXIT WHEN v_cur%NOTFOUND;
    dbms_output.put_line(v_a || ' ' || v_b);
  END LOOP;
  CLOSE v_cur;
END;

EXEC affiche(liste_produits(30.4))
```


SQL dynamique

Principe: paramétrer des instructions SQL au niveau du programme.

Ex: définir *lors de l'exécution du programme* les noms de tables ou colonnes affectées (ou les procédures à exécuter, les conditions, le type des variables)...

La chaîne de caractères d'une instruction dynamique est évaluée à l'exécution et non à la compilation.

```
-- desc user_objects: OBJECT_NAME, OBJECT_TYPE...
DECLARE
ordre varchar2(200);
BEGIN
  FOR liste_func IN (SELECT object_name
    FROM user_objects WHERE object_type='FUNCTION') LOOP
      ordre := 'DROP FUNCTION '|| liste_func.object_name;
      EXECUTE IMMEDIATE ordre;
  END LOOP;
END;
/
-- une instruction dynamique s'exécute avec EXECUTE IMMEDIATE
-- Noter que l'ordre dynamique s'écrit sans  ';'
```

⚠ Éviter d'utiliser SQL dynamique lorsque ce n'est pas nécessaire.

SQL dynamique: paramètres de substitution

Instruction compilée une seule fois, avant que les paramètres soient évalués.

- ✔ réduit la vulnérabilité aux injections SQL
- ✔ peut améliorer la performance
- ✔ plus besoin d'échapper les paramètres string

```
-- t doit avoir 2 colonnes (éventuellement plus)
CREATE PROCEDURE insert_generique(t varchar2) IS
ordre varchar2(100);
b int;
BEGIN
  ordre := 'INSERT INTO ' || t || ' VALUES (:a, :b)'; -- noms arbitraires
  for i in 1..100 loop
    EXECUTE IMMEDIATE ordre USING i,b;
    end loop;
END;
-- le seul moyen pour passer un paramètre NULL: une variable non initialisée.
```

Le paramètre doit être un littéral ($\simeq$ valeur de colonne). Pas nom de table, col...

Si on ne connaît pas la liste/le type des attributs d'un SELECT ou DML, on utilisera le paquet DBMS_SQL au lieu de native PL/SQL

SQL dynamique: curseur

Si l'ordre du curseur n'est pas connu à l'avance, utiliser un curseur dynamique `sys_refcursor`:

```
CREATE PROCEDURE affiche_gen (t VARCHAR2, a VARCHAR2, b VARCHAR2) is
  texte VARCHAR2(100);
  cur sys_refcursor; -- déclare le curseur dynamique cur
  x1 VARCHAR2(10);
  x2 int;
BEGIN
  texte := 'select '||a||','||b||' from '||t; --curseur déterminé à l'exécution
  dbms_output.put_line(texte);
  open cur for texte;
  LOOP
    FETCH cur INTO x1, x2;
    EXIT WHEN cur%NOTFOUND;
      dbms_output.put_line(x1||', '||x2);
  END LOOP;
END;
/
-- pas très fiable : l'utilisateur pourrait entrer n'importe quoi dans a
```

Droits d'exécution

```
CREATE OR REPLACE PROCEDURE create_log_table
-- use AUTHID CURRENT_USER to execute with the privileges and
-- schema context of the calling user
  AUTHID CURRENT_USER
AS
    tabname VARCHAR2(30);
BEGIN
    tabname := 'log_table_' || currentdate;
    EXECUTE IMMEDIATE 'CREATE TABLE ' || tabname;
END;
/
```

Avec `authid current_user` une procédure, fonction ou un paquet s'exécute avec les droits de celui qui les utilise.

Sinon, ils s'exécutent par défaut avec les droits de leur propriétaire (`owner`).

Autres éléments du langage PL/SQL

- créer un paquet

```
--  l'interface: spécifie les types/fonctions publiques
CREATE OR REPLACE PACKAGE emp_actions AS
   TYPE EmpRecTyp IS RECORD (emp_id INT, salary REAL);
   CURSOR desc_salary RETURN EmpRecTyp;
   PROCEDURE hire_employee (ename  VARCHAR2, sal NUMBER);
END emp_actions;
```

```
-- corps de l'interface: l'implémentation.
CREATE OR REPLACE PACKAGE BODY emp_actions AS
   CURSOR desc_salary RETURN EmpRecTyp IS
      SELECT empno, sal FROM emp ORDER BY sal DESC;
   PROCEDURE hire_employee (ename  VARCHAR2, sal NUMBER) IS
   BEGIN
      INSERT INTO emp VALUES (empno_seq.NEXTVAL, ename, sal);
   END hire_employee;
END emp_actions;
```

- constant keyword: `prix constant int :=2 ;`
- types de données `RECORD, VARRAY, TABLE, OBJECT...`
- gestion des erreurs
- étiqueter des blocs ou instructions: `«étiquette» instruction...`

Différences avec PL/pgSQL

Sous PL/pgSQL:

- blocs sont délimités pas un symbole spécial (ex: `$$`)
- pas de distinction entre fonction et procédure (on utilise `FUNCTION` dans tous les cas)
- syntaxe différente pour diagnostiquer `FOUND`, `ROWCOUNT`: pas d'attribut de curseur.
- variations sur la syntaxe des curseurs (variante automatique, curseurs dynamiques)
- SQL dynamique:
  - les paramètres de substitutions sont numérotés `$1, $2...`,
  - on utilise l'instruction `EXECUTE` au lieu de `EXECUTE IMMEDIATE`
- pas de "paquets" sous PostgreSQL.

PL/pgSQL: exemple

```
-- table Vente (produit int, montant number) ayant 10 lignes
CREATE FUNCTION ca_produit(OUT montant1 real, OUT montant2 real,
  OUT montant_restant real) AS $$
DECLARE
  c CURSOR FOR SELECT produit, montant FROM Vente ORDER BY montant DESC
  nb int := 0;
  prod int := 0;
  valeur_courante real := 0;
BEGIN
montant_restant := 0;
  OPEN c; -- le curseur calcule la requête
    FETCH c INTO prod, montant1; -- enregistre la 1ère valeur de c dans montant1
    FETCH c INTO prod, montant2; -- enregistre la 2è valeur de c dans montant2
    LOOP -- boucle en pgSQL
      EXIT WHEN nb > 10; -- exit quitte la boucle (la plus interne si imbriquée)
      FETCH c INTO prod, valeur_courante; -- récupère la valeur suivante de c
      montant_restant := montant_restant + valeur_courante;
      nb := nb+1;
    END LOOP;
  CLOSE c;
END
$$ LANGUAGE plpgsql;
```

Bibliographie

http://www.oracle.com/technetwork/database/features/plsql/index.html

https://docs.oracle.com/database/121/LNPLS/toc.htm

https://www.postgresql.org/docs/current/static/plpgsql.html

https://mariadb.com/kb/en/library/create-procedure/

https://perso.limsi.fr/anne/cours/

http://sys.bdpedia.fr/files/cbd-sys.pdf

Table des matières

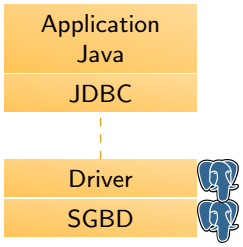
Interfaces entre SGBD et programmes: principe

Le langage de progr. fournit une (plusieurs) api pour interroger les BDs.
Pour garantir la portabilité du code, le langage définit souvent une interface (= API = passerelle) *indépendante du SGBD* entre le langage et le Driver du SGBD.
Le SGBD propose un pilote (driver), i.e., logiciel qui implémente l'API.
Les méthodes définies par l'API servent à:

- ouvrir une session sur le SGBD
- envoyer les instructions du programme au SGBD
- renvoyer au programme les résultat de la requête.
- fermer la session

Ex d'interfaces:

- JDBC pour Java
- Python DB API
- PDO pour PHP
- SQLAPI++ (payant) pour C++



L'interface JDBC

Interagir avec la BD depuis un programme JAVA

```
import java.sql.*;

public class test {
  public static void main(String[] args) {
    try {
      Class.forName("org.postgresql.Driver"); // needed if JDBC <4.0
      String url = "jdbc:postgresql://host:port/database";
      Connection conn = DriverManager.getConnection(url, "mylogin", "mypaswd");

      Statement st = conn.createStatement();
      ResultSet res = st.executeQuery("select titre, salle from prog");
      while(res.next()){
        System.out.print(res.getString("titre") + "\t" + res.getInt("salle"));
      }
      st.close();
      conn.close();
    }                    // bad idea: what if exception raised? better solution:
    catch (Exception e) {try-with-ressource:    https://stackoverflow.com/questions/2225221
      e.printStackTrace();
    }
  }
}
//Oracle PUIO: url = "jdbc:oracle:thin:@tp-oracle.ups.u-psud.fr:1522:dbinfo";
// Class.forName("oracle.jdbc.driver.OracleDriver"); est superflu
```

JDBC (2): l'interface Statement

```
...
  // établir une connexion (session) en créant un objet Connection
  // En fait il est recommandé d'utiliser Datasource plutôt que DriverManager
  Connection conn = DriverManager.getConnection(url, user, passwd);


  // crée un objet Statement
  Statement st = conn.createStatement();

  // exécute un Select
  ResultSet res = st.executeQuery("select titre, salle from prog");
  while(res.next()){
    System.out.print(res.getString("titre") + "\t" + res.getInt("salle")});
  }

  // exécute un Update, Delete, Insert, or DDL. Retourne le nb lignes affectées
  int j = st.executeUpdate("delete from prog where titre='Kagemusha'")
  System.out.print("nombre de lignes mises à jour:" + j);

  // on pourrait aussi envisager st.execute:
  // retourne true ssi le premier objet renvoyé par la requête est un ResultSet
...
```


JDBC (3): l'interface PreparedStatement

- hérite de `Statement`
- l'état est envoyé au SGBD où il est compilé et pourra être réutilisé en modifiant les valeurs de paramètres (attention: paramètre est toujours une valeur de colonne).
- même sans paramètre, compiler peut gagner du temps (plusieurs exécutions).

```
...
  // création des objets preparedStatement
  String ordreSQL = "INSERT INTO Prog VALUES(ugc bercy, The Godfather, ?, ?)";
  String ordreSQL3 = "SELECT * FROM Prog WHERE Heure < ?";
  preparedStatement st = conn.prepareStatement(ordreSQL);
  preparedStatement st2 = conn.prepareStatement("SELECT * FROM Prog");

  st.setInt(1,8);
  st.setString(2,"20h00");
  int j = st.executeUpdate();  // insère (ugc bercy, The Godfather, 8, 20h00)

  st.setInt(1,7);
  st.setString(2,"21h00");
  st.executeUpdate();

  ResultSet res = st2.executeQuery();
  while (res.next()) {System.out.println(res.getString(1));...}
...
```


JDBC (4) l'interface CallableStatement

- hérite de `PreparedStatement`
- permet d'exécuter une fonction/procédure stockée

```
...
  // crée des objets preparedStatement
  String call_string = "? = salutation(?)";
  callableStatement cs1 = conn.prepareCall(call_string);
  callableStatement cs2 = conn.prepareCall("split_name(?,?,?)");ioo
  callableStatement cs3 = conn.prepareCall("begin ? := salutation(?); end;");

  cs1.registerOutParameter(1,java.sql.Types.VARCHAR2);
  cs1.setString(2,"John");
  cs1.execute();
  String res = cs1.getString(1); // "Hello John"
...
```


L'interface JDBC (4): pour aller plus loin...

- autres méthode de l'objet connexion: accéder aux métadonnées (schémas...)
- différents types de curseurs (options permettant navigation, modification)
- différents types de drivers JDBC
- table des conversions entre types java et types BD
- transactions
- codes d'erreurs

L'interface PDO pour PHP (aperçu)

```php
<?php
try
{
  $url = 'mysql:host=localhost;dbname=test;charset=utf8';
  $conn = new PDO($url, 'mylogin', 'mypaswd');
}
catch(Exception $e)
{
  die('Erreur : '.$e->getMessage());
}

$res = $conn->query('SELECT * FROM Prog');
while ($donnees = $reS->fetch())
{
  echo $donnees['titre'] . '<br>';
}
$res->closeCursor();

$req = $bdd->prepare('SELECT titre FROM Prog WHERE nom_Cine=? AND heure >= ?');
$req->execute(array($_GET['nom_cine'], $_GET['heure']));
?>
```


Python DB API avec cx_oracle (aperçu)

```python
import cx_Oracle

# Try to connect
try:
    dsnStr = cx_Oracle.makedsn("tp-oracle.ups.u-psud.fr", "1522", "dbinfo")
    con = cx_Oracle.connect(user="c##bgroz_a", password="bgroz_a", dsn=dsnStr)
except:
    print "I am unable to connect to the database."

cur = conn.cursor()
try:
    cur.execute("""SELECT * from bar""")
except:
    print "I can't SELECT from bar"

rows = cur.fetchall()
print "\nRows: \n"
for row in rows:
    print "   ", row[1]
```


Python DB API avec psycopg2

```python
import psycopg2

# Try to connect
try:
    conn=psycopg2.connect("host='url' dbname='g_a' user='g_a' password='pwd")
except:
    print "I am unable to connect to the database."

cur = conn.cursor()
try:
    cur.execute("""SELECT * from bar""")
except:
    print "I can't SELECT from bar"

rows = cur.fetchall()
print "\nRows: \n"
for row in rows:
    print "   ", row[1]
```


Bibliographie

https://en.wikipedia.org/wiki/Java_Database_Connectivity

https://docs.oracle.com/javase/tutorial/jdbc/basics/

https://docs.oracle.com/javase/9/docs/api/java/sql/package-summary.html

http://initd.org/psycopg/docs/usage.html

https://perso.limsi.fr/anne/cours/