Import and export data in a DBMS: get data in, push data out

Chapter content:

- Import and export data in a DBMS
- Federated DB, and external tabless
- UDF: user defined functions
- Interact with DBMS from a program

Table of content

Import data

`INSERT` helps insert tuples 1 after another.

When loading huge amounts of data (ex: load a huge .csv) we use bulk loading tools specific to each DBMS.

```
-- PostgreSQL:
COPY Prog FROM '/user1/cinema_data.csv';
-- or use client side instruction from psql:
-- beware: write instruction on a single line, without final ';'
\copy zip_codes FROM '/path/to/csv/ZIP_CODES.txt' DELIMITER ',' CSV

-- MariaDB and MySQL:
LOAD DATA INFILE '/user1/cinema_data.csv' INTO Prog;
--   or use mysqlimport for multithread version.

-- Oracle:
--   may use sqlldr program, or external tables

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

Remark: generally you (ex: PUIO) you do not have enough rights to use COPY, but you can still use \copy.

Import a csv under 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 '"' 无视双引号
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
```

Export data into a file

```
-- PostgreSQL:
COPY (SELECT Titre, Horaire FROM Prog WHERE Horaire<'20h00')
  TO '/user1/cinema_data.csv';
-- or use client side instruction from psql:
-- beware: write instruction on a single line, without final ';'
\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: see next
```

Remark: again, generally you will use \copy instead of COPY.

Export a .csv under oracle

1. with SQL Developer, it's rather easy:

   https://stackoverflow.com/questions/4168398

2. with sqlplus one may:
   - write directly SQL instructions and rely on `spool` to write into the file
   - or rely on some PL/SQL procedure to generate automatically the instructions. To write into a file, one may use utl_file, or dbms_output with spool

```
-- Export  a query into fichier.csv under sqlplus: @script_export.sql
-- script_export.sql
set heading off -- unless we want a header
set feedback off -- to avoid 'xxx rows selected'
set markup csv on -- for version 12.1 at PUIO
-- on former versions, use instead: set colsep ','  ...
spool fichier.csv
select * from ma_table;
spool off
```

Export/Import the whole database (dump)

Saves into a file a sequence of SQL instructions that allows to reconstruct the whole database.

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

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

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

More on Oracle data pump:

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

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

Table of content

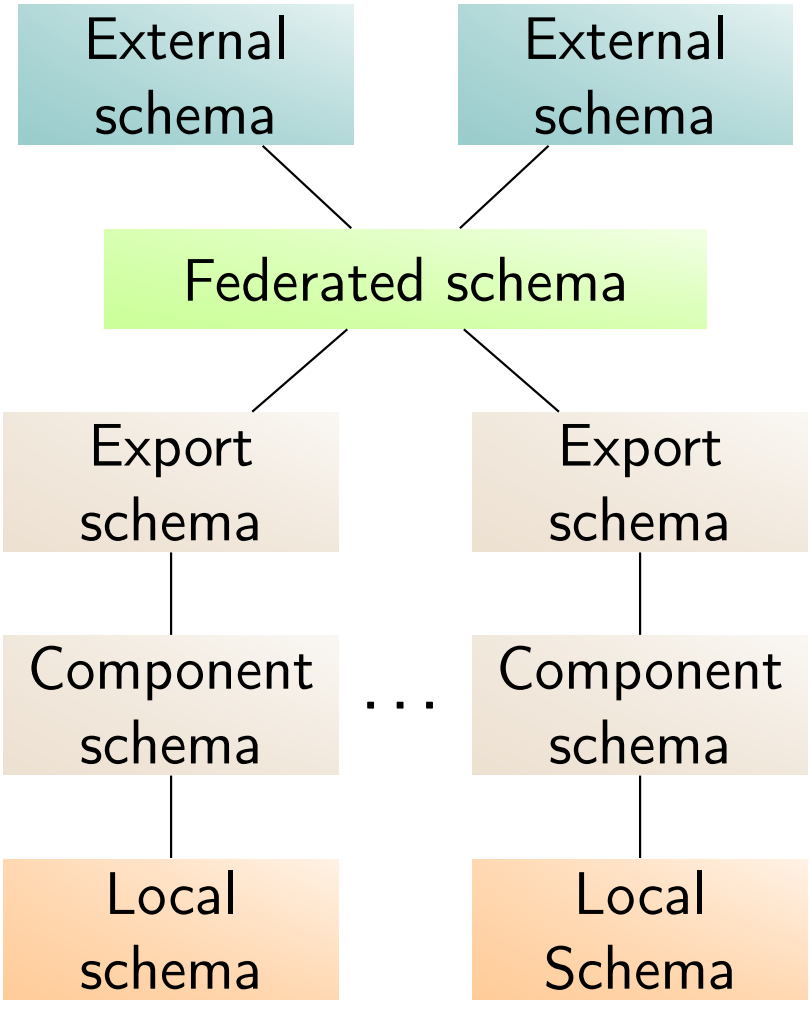
2018-2019

Import and export data in a DBMS: get data in, push data out

- Import and export data in a DBMS
- Federated DB, and external tabless
- UDF: user defined functions
- Interact with DBMS from a program

5 levels federated DB

联合的



Solution to provide common access to autonomous sources.

- logical schema of source is mapped into component schema (solves heterogeneity issues)
- Export schema is a portion of component schema (access control. . .)
- Federated schema integrates export schemas, is aware of data repartition. Might be large.
- External schema implements access control, simplifies federated schema, hides schema evolutions.

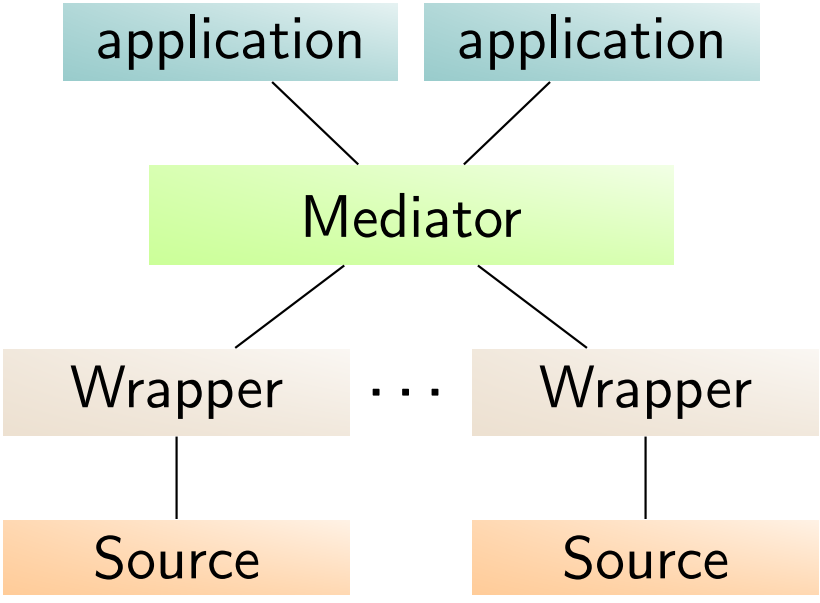
Metadata helps!

Federated: autonomous DB sharing their data (pairwise) (minimal centralization)

自主的

Mediator-Wrapper

中介 封装



Solution to provide common access to heterogeneous independent sources (not relying on metadata).

- wrapper is buffer level between sources and mediator; solves heterogeneity issues (interfaces, data model, semantics)
- Mediator integrates the data *but mediator schema derived from applications and not from source integration*.

Remote data sources in DBMS

远程操作

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.

Remote data sources in DBMS (2)

PostgreSQL

```
CREATE EXTENSION postgres_fdw;

-- create remote access
CREATE SERVER hive1
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'foo', dbname 'foodb', port '5432');

-- -- authentication credentials
CREATE USER MAPPING FOR CURRENT_USER
  SERVER hive1
  OPTIONS (user 'jdoe', password 'secret1');

-- import distant schema
CREATE SCHEMA app;

IMPORT FOREIGN SCHEMA public -- or CREATE FOREIGN TABLE  vp (...) SERVER hive1
  FROM SERVER hive1
  INTO app;

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

Table of content

Procedures

Most DBMS allow to define procedures.
Procedure : set of instructions, stored on a DBMS server.
one sometimes speaks of *stored procedure*

Use cursors to iterate through the result of an SQL query.

- ✔ allows to grant some user the right to execute a procedure without authorizing direct access to the tables
- ✔ sometimes better performance as the query is stored on the server side
- ✔ "portable" in some way: specific to DBMS, but cross-OS.
- ✔ modules grouping instructions facilitate re-use

- ✘ **not so "portable": procedures are DBMS-specific.**
- ✘ old language

Oracle and IBM DB2: PL/SQL
PostgreSQL : PL/pgSQL
Microsoft : Transact-SQL
similar syntax for MariaDB, MySQL.

PL/SQL: bloc structure

Remark: PL/SQL is an old language: 1991. Ada-inspired syntax (hence Pascal-inspired).

Bloc anonyme

```
[ DECLARE
    variable declarations]
BEGIN
 -- function body
    instructions
[ EXCEPTIONS
  error handling ]
END;
```

```
-- to make dbms_output.put_line
-- display text on screen:
-- set serveroutput on
DECLARE
  texte VARCHAR2 (40) := 'Hello World!';
BEGIN
  DBMS_OUTPUT.put_line (texte);
END;
/
```

equivalent to:

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

PL/SQL under sql*plus

Particulars for 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).
- to debug in case of compilation failure, type: `show errors`

Functions (UDF)

```
CREATE OR REPLACE FUNCTION salutation (nom VARCHAR2)
    RETURN VARCHAR2 -- a function must have a return type
IS -- ou  AS :  no difference between the 2
BEGIN
   RETURN 'Hello ' || nom;
END; -- compilation fails if DBMS has another salutation object
```

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

can be used in a block, other function/procedure, external program, SQL query:

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

Procedures

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

```
-- parameters have mode (see usage on next slide):
-- IN (input - by default), OUT (output) or IN OUT (read/write)
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;
/
```

↪ a procedure may have 0,1, or several OUT parameters

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

Procedures : usage

Usage: in PL/SQL block (anonymous block, or procedure. . .)

```
EXECUTE DBMS_OUTPUT.put_line('aa' || 'b')
-- EXECUTE or EXEC equivalent to:
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;
/
```

Procedures : 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;
-- one may assign multiple variables at once:
-- SELECT COUNT(*) into total, MIN(prix_place) INTO prix FROM concert
```

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

Query SELECT must return a single result (neither 0 nor several).

Procedures : types inferred from data

```
-- 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; --  so we do not have to fix size
v_complete ville%rowtype; --  so we do not have to fix columns
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;
```

Procedures : cursors

Used to handle (iterate) a set of lines: the result of a SELECT query.

Cursor handling involves 4 steps:

0. declare cursor
1. open cursor (instruction OPEN)

   builds execution plan, initializes program. The data returned by cursor matches the state of the database at the time of this OPEN instruction.

2. iterate through the cursor (with FETCH instructions)

   Updates may occur on the DB between FETCH ops, by the program defininf the cursor or by others. But the result will not be affected. In terms of results: as if the cursore had executed the SELECT query during OPEN instruction. In practice, though, DBMS may choose not to materialize, in order to return the first tuples faster and to avoid storing lots of data.

3. close the cursor (CLOSE instruction)

Procedures: cursor attributes (implicit or explicit)

PL/SQL creates implicit cursor implicite for each SELECT INTO, UPDATE, or DELETE.

Implicit cursors attributes:

sql%rowcount
#lines affected

sql%found
true iff $\geq$ 1 line affected

sql%notfound
the opposite

```
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;
```

Attributes for explicit cursor c:

c%ISOPEN: true iff cursor is open.

c%rowcount: #tuples fetched till current state.

(⚠ numbers of FETCH, not the number of tuples in SELECT)

c%found: true iff last FETCH returned a tuple.

Iterating through cursors

```
-- table Sales (product int, amount number) with 10 lines
CREATE PROCEDURE affiche(amount1 OUT NUMBER, amount2 OUT NUMBER,
  remaining_amount  OUT NUMBER)
AS
  CURSOR c IS SELECT product, amount FROM Sales ORDER BY amount DESC;
  current_value NUMBER := 0;
  prod int := 0;
BEGIN
  remaining_amount := 0;
  OPEN c; -- cursor computes query
  FETCH c INTO prod, amount1; -- records first value of c into amount1
  FETCH c INTO prod, amount2;
  DBMS_OUTPUT.put_line(c%rowcount); -- affiche 2
  LOOP -- PL/SQL loop
    FETCH c INTO prod, current_value; -- get the next value into c
    EXIT WHEN c%notfound; -- exits the (most internal) loop
    remaining_amount := remaining_amount + current_value;
  END LOOP;
  CLOSE c;
END;
```

⚠ *FETCH does not modify amount1 if Sales has not enough lines (no error raised either): use cursor attributes if you want to check.*

Inferred type for cursor

```
-- table Sales (product int, amount number ... )
CREATE PROCEDURE affiche(amount1 IN OUT NUMBER)
AS
  CURSOR c IS SELECT product, amount FROM Sales ORDER BY amount DESC;
  line c%rowtype; -- type inferred from cursor
BEGIN
  OPEN c;
  FETCH c INTO line;
  amount1 := amount1 + line.amount;
  CLOSE c;
END;
```

cursors managed automatically by FOR loop

```
CREATE PROCEDURE affiche_products AS
  CURSOR c IS SELECT product, amount FROM Sales ORDER BY amount DESC;
BEGIN
  FOR line IN c LOOP
    DBMS_OUTPUT.put_line(line.product || ' , ' || line.amount);
  END LOOP;
END;
-- advantage: no need to declare line, nor write FETCH, OPEN, CLOSE
```

And for a simple use case:

```
CREATE PROCEDURE affiche_products AS
BEGIN
  FOR line IN (SELECT product, amount FROM Sales ORDER BY amount DESC) LOOP
    DBMS_OUTPUT.put_line(line.product || ' , ' || line.amount);
  END LOOP;
END;
```

Cursor variables

```
CREATE FUNCTION products_list(amount_max Sales.amount%type)
RETURN SYS_REFCURSOR AS
  c SYS_REFCURSOR; -- builds dynamic cursor: not a real one.
BEGIN
  OPEN c FOR SELECT product, amount FROM Sales
    WHERE amount < amount_max ORDER BY amount DESC;
  RETURN c;
END;

CREATE PROCEDURE affiche(v_cur IN SYS_REFCURSOR) AS
  v_a   VARCHAR2(10);
  v_b   VARCHAR2(10);
BEGIN
  LOOP
    -- dynamic cursors can only be handled with FETCH
    -- they do not support "automatic" handling with a 'FOR' loop
    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(products_list(30.4))
```

Dynamic SQL

Principle: let program assign the parameters of SQL instructions.

Ex: define *at running time* the name of tables or columns that must be processed (or the procedures that must be executed, conditions, variable type)...

The string defining a dynamic instruction is evaluated while running program, instead of 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;
/
-- a dynamic instruction is run with EXECUTE IMMEDIATE
-- Note that we do not write a  ';'
```

 Avoid dynamic SQL unless necessary.

Dynamic SQL: substitution parameters

Instruction compiled single time, before parameters are evaluated.

- ✔ prevents SQL injection
- ✔ may improve performance
- ✔ spares us the task of escaping string parameters

```
-- t must have 2 columns (may have more)
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;
-- only way to pass a NULL parameter: use a non-initialized variable.
```

Parameter must be a literal ($\simeq$ column value). No table name, col...

If the parameters of a DML or SELECT are unknown, use DBMS_SQL instead of native PL/SQL

Dynamic SQL: cursor

If cursor order is unknown; use dynamic cursor `sys_refcursor`:

```
CREATE PROCEDURE affiche_gen (t VARCHAR2, a VARCHAR2, b VARCHAR2) is
  texte VARCHAR2(100);
  cur sys_refcursor; -- declares the dynamic cursor cur
  x1 VARCHAR2(10);
  x2 int;
BEGIN
  texte := 'select '||a||','||b||' from '||t; --cursor determined at runtime
  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;
/
-- not reliable : user may input anything in 'a'
```

Execution rights

```
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;
/
```

With `authid current_user` a procedure, function or package runs with the rights of its user.

Otherwise, run with its owner's rights.

Other PL/SQL features

- create a package

```
--  Interface: specifies types/public functions
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;
```

```
-- interface body: implementation.
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 ;`
- datatypes `RECORD, VARRAY, TABLE, OBJECT...`
- error handling
- labeling blocks or instructions: `«label» instruction...`

Differences with PL/pgSQL

Under PostgreSQL PL/pgSQL:

- blocks are delimited with special symbol (ex: `$$`)
- no distinction between function and procedure (use `FUNCTION` in all cases)
- syntax for `FOUND` diagnostics, `ROWCOUNT`: no cursor attribute.
- variations on cursors syntax (automatic variant, dynamic cursors)
- Dynamic SQL:
  - substitution parameters are numbered `$1, $2...`,
  - use `EXECUTE` instead of `EXECUTE IMMEDIATE`
- no "packages" for PostgreSQL.

PL/pgSQL: example

```
-- table Sales (product int, amount number) ayant 10 lines
CREATE FUNCTION ca_product(OUT amount1 real, OUT amount2 real,
  OUT remaining_amount real) AS $$
DECLARE
  c CURSOR FOR SELECT product, amount FROM Sales ORDER BY amount DESC
  nb int := 0;
  prod int := 0;
  current_value real := 0;
BEGIN
remaining_amount := 0;
  OPEN c; -- cursor computes query
    FETCH c INTO prod, amount1; -- records 1st value of c into amount1
    FETCH c INTO prod, amount2; --  records 2nd value of c into amount2
    LOOP -- pgSQL loop
      EXIT WHEN nb > 10; -- exits from the loop (internal one if nested)
      FETCH c INTO prod, current_value; -- gets next value of c
      remaining_amount := remaining_amount + current_value;
      nb := nb+1;
    END LOOP;
  CLOSE c;
END
$$ LANGUAGE plpgsql;
```

References

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 of content

2018-2019

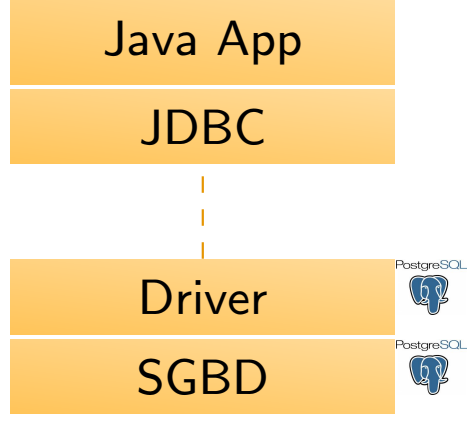
Connecting a program to DBMS: mainnideas

The prog. language provides API(s) to query DBMS.
To make code portable, language often defines API *independent from DBMS* between language and the DBMS Driver.
DBMS provides driver, i.e., lsoftware implementing l'API.
API methods allow to:

- open DBMS session
- send program instructions to DBMS
- return query results to program
- close the session

Some famous interfaces:

- JDBC pour Java
- Python DB API
- PDO pour PHP
- SQLAPI++ (not free) for C++



The JDBC interface

Manipulate a DBMS from 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): interface Statement

```
...
  // build a connection (session) by creating Connection object
  // Actually, should rather use Datasource instead of DriverManager
  Connection conn = DriverManager.getConnection(url, user, passwd);


  // create Statement object
  Statement st = conn.createStatement();

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

  // executes an Update, Delete, Insert, or DDL. Returns nb affected lines
  int j = st.executeUpdate("delete from prog where titre='Kagemusha'")
  System.out.print("nombre de lines mises à jour:" + j);

  // we could consider st.execute:
  // returns true first object returned by query is ResultSet
...
```

JDBC (3): Interface PreparedStatement

- inherits from `Statement`
- state is sent to DBMS where it is compiled and can be reused with other parameter values (attention: parameter must be column value).
- even without parameter, compiling can improve performance (if severals executions).

```
...
  // build preparedStatement objects
  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();  // inserts (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) Interface CallableStatement

- inherits from `PreparedStatement`
- allows to execute function/procedure

```
...
  // build preparedStatement object
  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"
...
```

Interface JDBC (4): further details...

- other methods for connection object: access metadata (schemas...)
- multiple kinds of cursors (navigation options, modification)
- multiple driver types for JDBC
- see some conversion table btw. Java types and SQL types
- transactions
- error codes

Interface PDO for PHP (overview)

```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 with cx_oracle (overview)

```
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 with psycopg2

```
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]
```

References

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/