Administrarea Bazelor de Date
Managementul în Tehnologia Informaţiei

Sisteme Informatice şi Standarde Deschise (SISD)

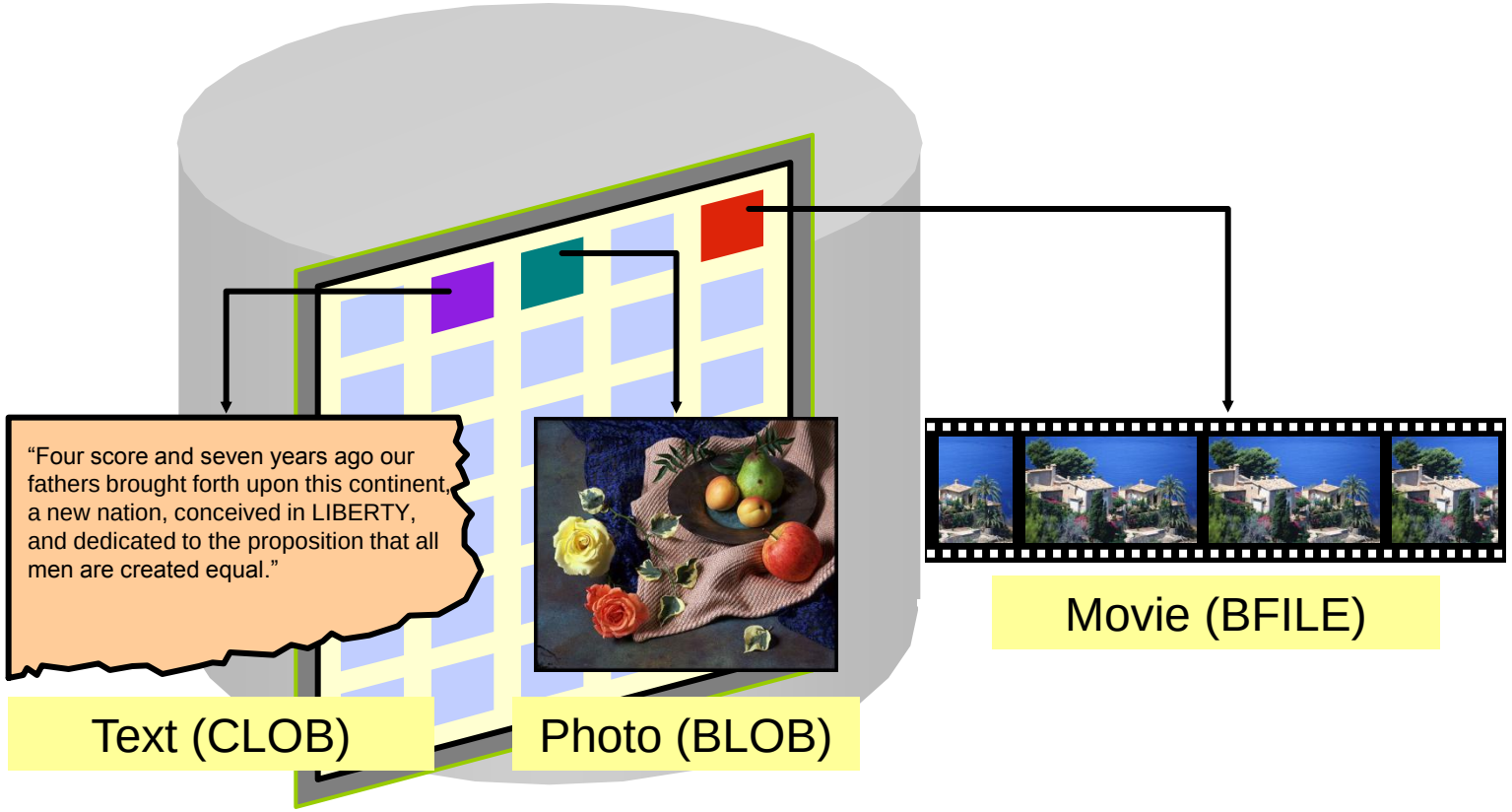
2009-2010

Curs 9

Standarde pentru programarea bazelor de date (3)

Manipulating Large Objects
What Is a LOB?

- LOBs are used to store large unstructured data such as text, graphic images, films, and sound waveforms.

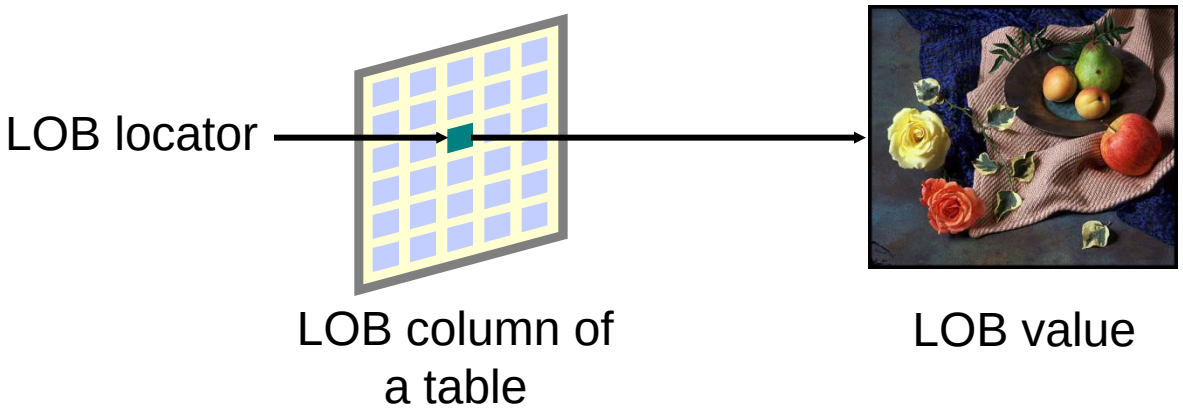


Contrasting LONG and LOB Data Types

LONG and LONG RAW	LOB
Single LONG column per table	Multiple LOB columns per table
Up to 2 GB	Up to 4 GB
SELECT returns data	SELECT returns locator
Data stored in-line	Data stored in-line or out-of-line
Sequential access to data	Random access to data

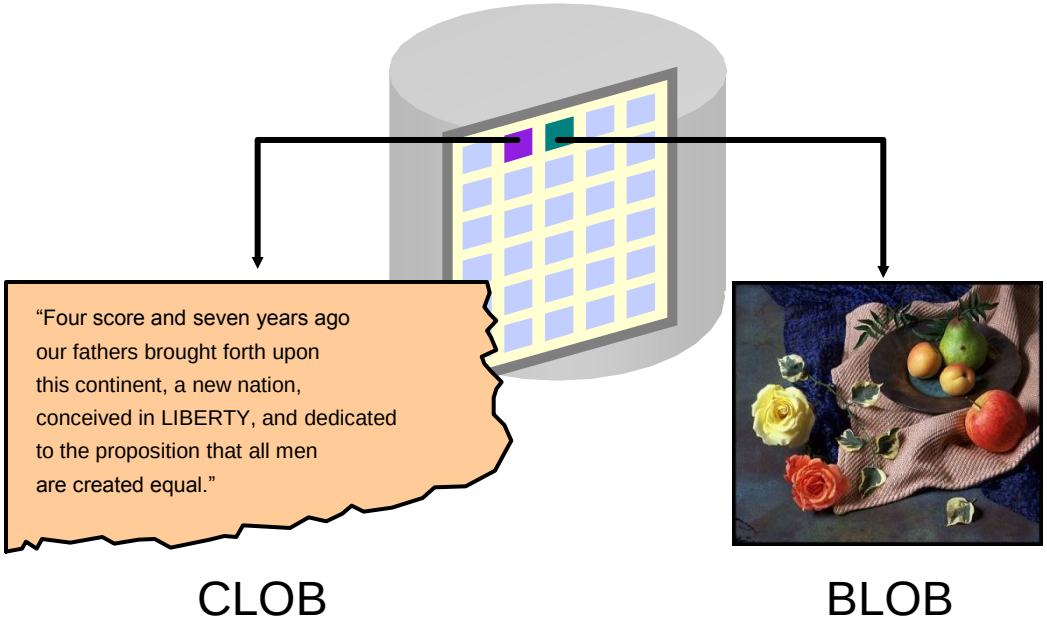
Anatomy of a LOB

- The LOB column stores a locator to the LOB's value.



Internal LOBs

- The LOB value is stored in the database.

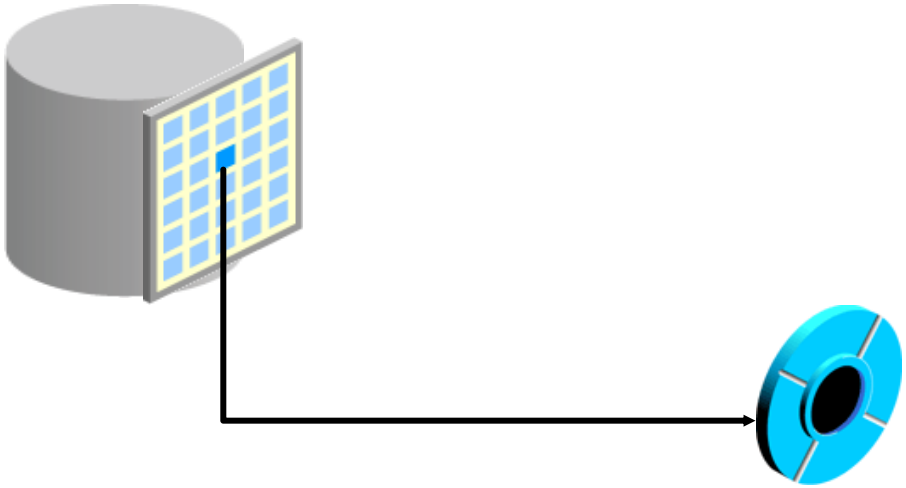


Managing Internal LOBs

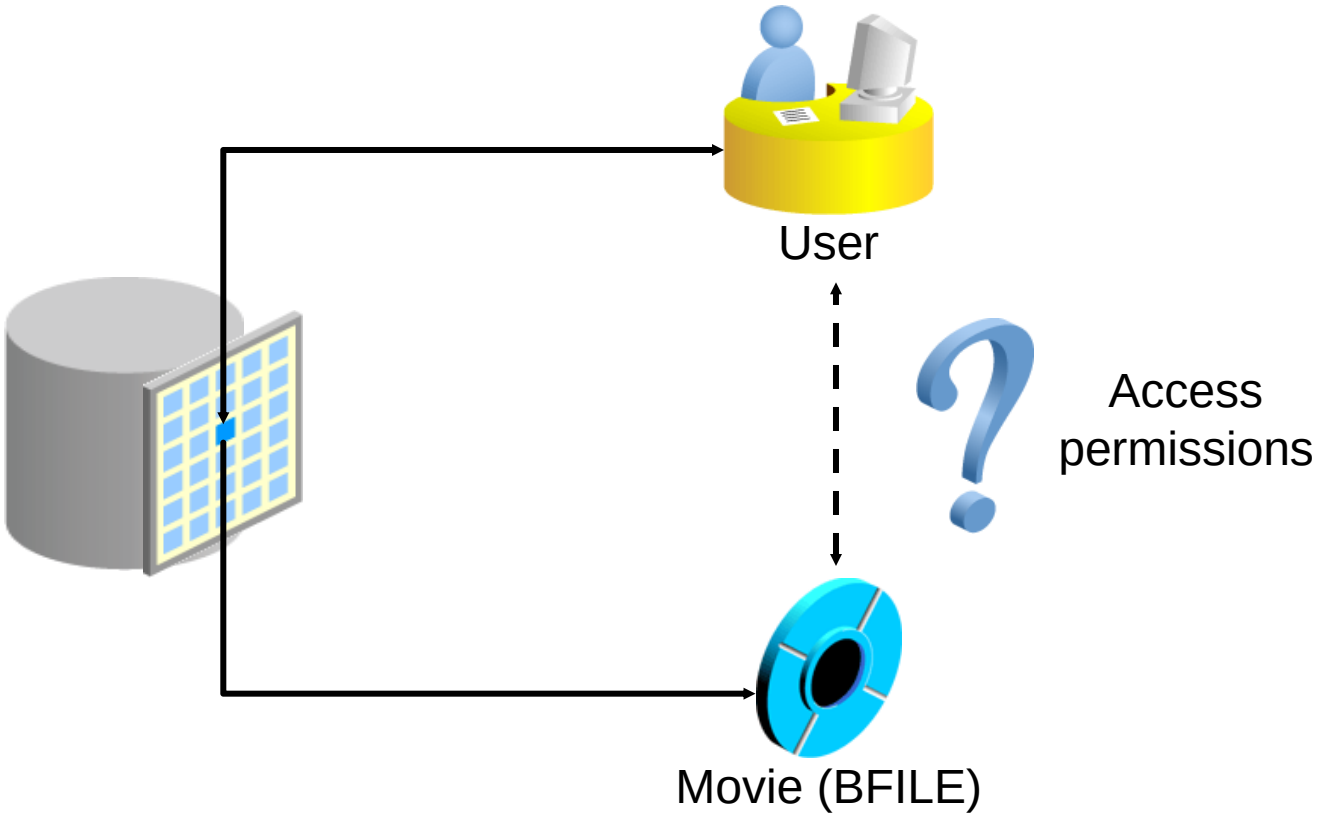
- To interact fully with LOB, file-like interfaces are provided in:
  - PL/SQL package DBMS_LOB
  - Oracle Call Interface (OCI)
  - Oracle Objects for object linking and embedding (OLE)
  - Pro*C/C++ and Pro*COBOL precompilers
  - JDBC (Java Database Connectivity)
- The Oracle server provides some support for LOB management through SQL.

What Are BFILEs?

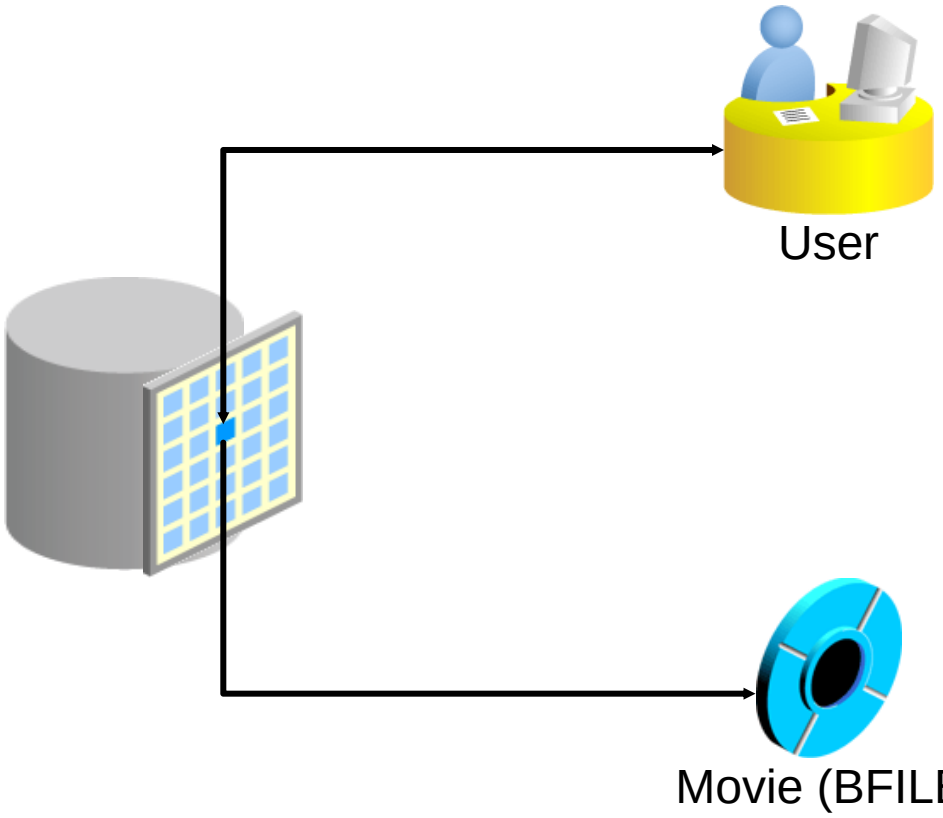
- The BFILE data type supports an external or file-based large object as:
  - Attributes in an object type
  - Column values in a table



Securing BFILEs



A New Database Object: DIRECTORY



Guidelines for Creating DIRECTORY Objects

- Do not create DIRECTORY objects on paths with database files.
- Limit the number of people who are given the following system privileges:
  - CREATE ANY DIRECTORY
  - DROP ANY DIRECTORY
- All DIRECTORY objects are owned by SYS.
- Create directory paths and properly set permissions before using the DIRECTORY object so that the Oracle server can read the file.

Managing BFILEs

- The DBA or the system administrator:
  1. Creates an OS directory and supplies files
  2. Creates a `DIRECTORY` object in the database
  3. Grants the `READ` privilege on the `DIRECTORY` object to appropriate database users
- The developer or the user:
  4. Creates an Oracle table with a column defined as a `BFILE` data type
  5. Inserts rows into the table using the `BFILENAME` function to populate the `BFILE` column
  6. Writes a PL/SQL subprogram that declares and initializes a `LOB` locator, and reads `BFILE`

Preparing to Use BFILEs

1. Create an OS directory to store the physical data files.

```
mkdir /temp/data_files
```

2. Create a DIRECTORY object by using the CREATE DIRECTORY command.

```
CREATE DIRECTORY data_files
                AS '/temp/data_files';
```

3. Grant the READ privilege on the DIRECTORY object to appropriate users.

```
GRANT READ ON DIRECTORY data_files
   TO SCOTT, MANAGER_ROLE, PUBLIC;
```

Populating BFILE Columns with SQL

- Use the BFILENAME function to initialize a BFILE column. The function syntax is:

```
FUNCTION BFILENAME(directory_alias IN
VARCHAR2, filename IN VARCHAR2)
RETURN BFILE;
```

- Example:
  - Add a BFILE column to a table.

```
ALTER TABLE employees ADD video BFILE;
```

  - Update the column using the BFILENAME function.

```
UPDATE employees
  SET video = BFILENAME('DATA_FILES',
'King.avi')
WHERE employee_id = 100;
```

Populating a BFILE Column with PL/SQL

```
CREATE PROCEDURE set_video(
  dir_alias VARCHAR2, dept_id NUMBER) IS
  filename VARCHAR2(40);
  file_ptr BFILE;
  CURSOR emp_csr IS
    SELECT first_name FROM employees
    WHERE department_id = dept_id FOR UPDATE;
BEGIN
  FOR rec IN emp_csr LOOP
    filename := rec.first_name || '.gif';
    file_ptr := BFILENAME(dir_alias, filename);
    DBMS_LOB.FILEOPEN(file_ptr);
    UPDATE employees SET video = file_ptr
      WHERE CURRENT OF emp_csr;
    DBMS_OUTPUT.PUT_LINE('FILE: ' || filename ||
     ' SIZE: ' || DBMS_LOB.GETLENGTH(file_ptr));
    DBMS_LOB.FILECLOSE(file_ptr);
  END LOOP;
END set_video;
```

Using DBMS_LOB Routines with BFILEs

- The DBMS_LOB.FILEEXISTS function can verify if the file exists in the OS. The function:
  - Returns 0 if the file does not exist
  - Returns 1 if the file does exist

```
CREATE FUNCTION get_filesize(file_ptr BFILE)
RETURN NUMBER IS
  file_exists BOOLEAN;
  length NUMBER:= -1;
BEGIN
  file_exists :=
DBMS_LOB.FILEEXISTS(file_ptr)=1;
  IF file_exists THEN
    DBMS_LOB.FILEOPEN(file_ptr);
    length := DBMS_LOB.GETLENGTH(file_ptr);
    DBMS_LOB.FILECLOSE(file_ptr);
  END IF;
  RETURN length;
END;
/
```

Migrating from LONG to LOB

- Oracle Database 10*g* enables migration of LONG columns to LOB columns.
  - Data migration consists of the procedure to move existing tables containing LONG columns to use LOBs:

```
ALTER TABLE [<schema>.] <table_name>
   MODIFY (<long_col_name> {CLOB | BLOB | NCLOB}
```

  - Application migration consists of changing existing LONG applications for using LOBs.

Migrating from LONG to LOB

- Implicit conversion: From LONG (LONG RAW) or a VARCHAR2(RAW) variable to a CLOB (BLOB) variable, and vice versa
- Explicit conversion:
  - TO_CLOB() converts LONG, VARCHAR2, and CHAR to CLOB.
  - TO_BLOB() converts LONG RAW and RAW to BLOB.
- Function and procedure parameter passing:
  - CLOBs and BLOBs as actual parameters
  - VARCHAR2, LONG, RAW, and LONG RAW are formal parameters, and vice versa.
- LOB data is acceptable in most of the SQL and PL/SQL operators and built-in functions.

DBMS_LOB Package

- Working with LOBs often requires the use of the Oracle-supplied DBMS_LOB package.
- DBMS_LOB provides routines to access and manipulate internal and external LOBs.
- Oracle Database 10*g* enables retrieving LOB data directly using SQL without a special LOB API.
- In PL/SQL, you can define a VARCHAR2 for a CLOB and a RAW for a BLOB.

DBMS_LOB Package

- Modify LOB values:
  APPEND, COPY, ERASE, TRIM, WRITE, LOADFROMFILE
- Read or examine LOB values:
  GETLENGTH, INSTR, READ, SUBSTR
- Specific to BFILEs:
  FILECLOSE, FILECLOSEALL, FILEEXISTS, FILEGETNAME, FILEISOPEN, FILEOPEN

DBMS_LOB Package

- NULL parameters get NULL returns.
- Offsets:
  - BLOB, BFILE: Measured in bytes
  - CLOB, NCLOB: Measured in characters
- There are no negative values for parameters.

DBMS_LOB.READ and DBMS_LOB.WRITE

```
PROCEDURE READ (
 lobsrc IN BFILE|BLOB|CLOB ,
 amount IN OUT BINARY_INTEGER,
offset IN INTEGER,
buffer OUT RAW|VARCHAR2 )
```

```
PROCEDURE WRITE (
lobdst IN OUT BLOB|CLOB,
amount IN OUT BINARY_INTEGER,
offset IN INTEGER := 1,
buffer IN RAW|VARCHAR2 )  -- RAW for BLOB
```

Initializing LOB Columns Added to a Table

- Create the table with columns using the LOB type, or add the LOB columns using ALTER TABLE.

```
ALTER TABLE employees
     ADD (resume CLOB, picture BLOB);
```

- Initialize the column LOB locator value with the DEFAULT option or DML statements using:
  - EMPTY_CLOB() function for a CLOB column
  - EMPTY_BLOB() function for a BLOB column

```
CREATE TABLE emp_hiredata (
   employee_id  NUMBER(6),
   full_name    VARCHAR2(45),
   resume       CLOB DEFAULT EMPTY_CLOB(),
   picture      BLOB DEFAULT EMPTY_BLOB());
```

Populating LOB Columns

- Insert a row into a table with LOB columns:

```
INSERT INTO emp_hiredata
  (employee_id, full_name, resume, picture)
 VALUES (405, 'Marvin Ellis', EMPTY_CLOB(), NULL);
```

- Initialize a LOB using the EMPTY_BLOB() function:

```
UPDATE emp_hiredata
 SET resume = 'Date of Birth: 8 February 1951',
    picture = EMPTY_BLOB()
 WHERE employee_id = 405;
```

- Update a CLOB column:

```
UPDATE emp_hiredata
 SET resume = 'Date of Birth: 1 June 1956'
 WHERE employee_id = 170;
```

Updating LOB by Using DBMS_LOB in PL/SQL

```
DECLARE
  lobloc CLOB;     -- serves as the LOB locator
  text   VARCHAR2(50) := 'Resigned = 5 June 2000';
  amount NUMBER ;  -- amount to be written
  offset INTEGER;  -- where to start writing
BEGIN
  SELECT resume INTO lobloc FROM emp_hiredata
  WHERE employee_id = 405 FOR UPDATE;
  offset := DBMS_LOB.GETLENGTH(lobloc) + 2;
  amount := length(text);
  DBMS_LOB.WRITE (lobloc, amount, offset, text);
  text := ' Resigned = 30 September 2000';
  SELECT resume INTO lobloc FROM emp_hiredata
  WHERE employee_id = 170 FOR UPDATE;
  amount := length(text);
  DBMS_LOB.WRITEAPPEND(lobloc, amount, text);
  COMMIT;
END;
```

Selecting CLOB Values by Using SQL

```
SELECT employee_id, full_name , resume -- CLOB
FROM emp_hiredata
WHERE employee_id IN (405, 170);
```

EMPLOYEE_ID	FULL_NAME	RESUME
405	Marvin Ellis	Date of Birth: 8 February 1951 Resigned = 5 June 2000
170	Joe Fox	Date of Birth: 1 June 1956 Resigned = 30 September 2000

Selecting CLOB Values by Using DBMS_LOB

- DBMS_LOB.SUBSTR (lob, amount, start_pos)
- DBMS_LOB.INSTR (lob, pattern)

```
SELECT DBMS_LOB.SUBSTR (resume, 5, 18),
       DBMS_LOB.INSTR (resume,' = ')
FROM   emp_hiredata
WHERE  employee_id IN (170, 405);
```

DBMS_LOB.SUBSTR(RESUME,5,18)	DBMS_LOB.INSTR(RESUME,'=')
Febru	40
June	36

Selecting CLOB Values in PL/SQL

```
SET LINESIZE 50 SERVEROUTPUT ON FORMAT
WORD_WRAP
DECLARE
  text VARCHAR2(4001);
BEGIN
 SELECT resume INTO text
 FROM emp_hiredata
 WHERE employee_id = 170;
 DBMS_OUTPUT.PUT_LINE('text is: '|| text);
END;
```

text is: Date of Birth: 1 June 1956 Resigned = 30 September 2000

PL/SQL procedure successfully completed.

Removing LOBs

- Delete a row containing LOBs:

```
DELETE
FROM  emp_hiredata
WHERE employee_id = 405;
```

- Disassociate a LOB value from a row:

```
UPDATE emp_hiredata
SET resume = EMPTY_CLOB()
WHERE employee_id = 170;
```

Temporary LOBs

- Temporary LOBs:
  - Provide an interface to support creation of LOBs that act like local variables
  - Can be BLOBs, CLOBs, or NCLOBs
  - Are not associated with a specific table
  - Are created using the DBMS_LOB.CREATETEMPORARY procedure
  - Use DBMS_LOB routines
- The lifetime of a temporary LOB is a session.
- Temporary LOBs are useful for transforming data in permanent internal LOBs.

Creating a Temporary LOB

- PL/SQL procedure to create and test a temporary LOB:

```
CREATE OR REPLACE PROCEDURE is_templob_open(
  lob IN OUT BLOB, retval OUT INTEGER) IS
BEGIN
  -- create a temporary LOB
  DBMS_LOB.CREATETEMPORARY (lob, TRUE);
  -- see if the LOB is open: returns 1 if open
  retval := DBMS_LOB.ISOPEN (lob);
  DBMS_OUTPUT.PUT_LINE (
    'The file returned a value...' || retval);
  -- free the temporary LOB
  DBMS_LOB.FREETEMPORARY (lob);
END;
/
```

Design Considerations for PL/SQL Code

Standardizing Constants and Exceptions

- Constants and exceptions are typically implemented using a bodiless package (that is, in a package specification).
  - Standardizing helps to:
    - Develop programs that are consistent
    - Promote a higher degree of code reuse
    - Ease code maintenance
    - Implement company standards across entire applications
  - Start with standardization of:
    - Exception names
    - Constant definitions

Standardizing Exceptions

- Create a standardized error-handling package that includes all named and programmer-defined exceptions to be used in the application.

```
CREATE OR REPLACE PACKAGE error_pkg IS
  fk_err          EXCEPTION;
  seq_nbr_errEXCEPTION;
  PRAGMA EXCEPTION_INIT (fk_err, -2292);
  PRAGMA EXCEPTION_INIT (seq_nbr_err, -2277);
  ...
END error_pkg;
/
```

Standardizing Exception Handling

- Consider writing a subprogram for common exception handling to:
  - Display errors based on `SQLCODE` and `SQLERRM` values for exceptions
  - Track run-time errors easily by using parameters in your code to identify:
    - The procedure in which the error occurred
    - The location (line number) of the error
    - `RAISE_APPLICATION_ERROR` using stack trace capabilities, with the third argument set to `TRUE`

Standardizing Constants

- For programs that use local variables whose values should not change:
  - Convert the variables to constants to reduce maintenance and debugging
  - Create one central package specification and place all constants in it

```
CREATE OR REPLACE PACKAGE constant_pkg IS
  c_order_received CONSTANT VARCHAR(2) := 'OR';
  c_order_shipped  CONSTANT VARCHAR(2) := 'OS';
  c_min_sal        CONSTANT NUMBER(3)  := 900;
  ...
END constant_pkg;
```

Local Subprograms

- A local subprogram is a `PROCEDURE` or `FUNCTION` defined in the declarative section.
- The local subprogram must be defined at the end of the declarative section.

```
CREATE PROCEDURE employee_sal(id NUMBER) IS
   emp employees%ROWTYPE;
   FUNCTION tax(salary VARCHAR2) RETURN NUMBER IS
   BEGIN
     RETURN salary * 0.825;
   END tax;
BEGIN
   SELECT * INTO emp
   FROM EMPLOYEES WHERE employee_id = id;
   DBMS_OUTPUT.PUT_LINE('Tax: '||tax(emp.salary));
END;
```

Definer's Rights Versus Invoker's Rights

- Definer's rights:
  - Used prior to Oracle8*i*
  - Programs execute with the privileges of the creating user.
  - User does not require privileges on underlying objects that the procedure accesses. User requires privilege only to execute a procedure.
- Invoker's rights:
  - Introduced in Oracle8*i*
  - Programs execute with the privileges of the calling user.
  - User requires privileges on the underlying objects that the procedure accesses.

Specifying Invoker's Rights

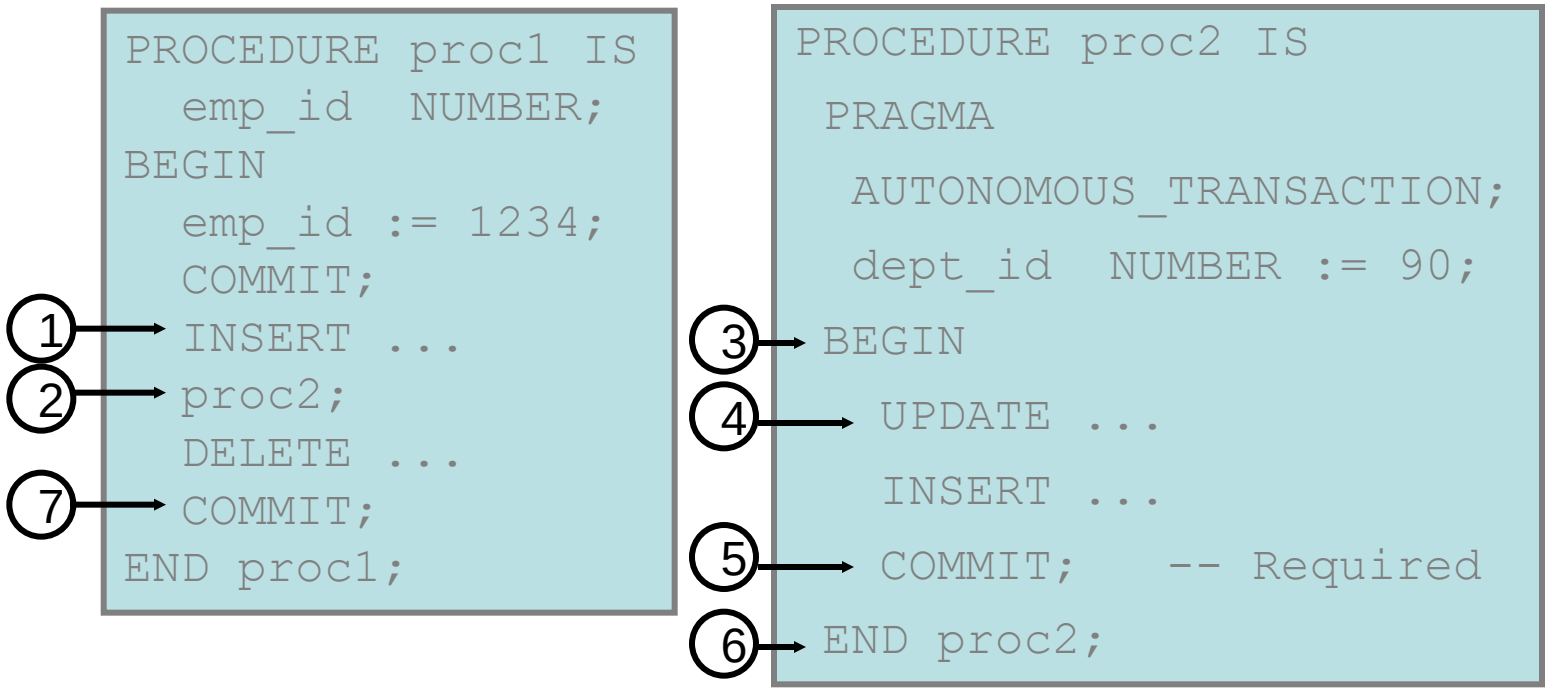
- Set `AUTHID` to `CURRENT_USER`:

```
CREATE OR REPLACE PROCEDURE add_dept(
  id NUMBER, name VARCHAR2) AUTHID
   CURRENT_USER IS
BEGIN
  INSERT INTO departments
  VALUES (id,name,NULL,NULL);
END;
```

- When used with stand-alone functions, procedures, or packages:
  - Names used in queries, DML, Native Dynamic SQL, and `DBMS_SQL` package are resolved in the invoker's schema
  - Calls to other packages, functions, and procedures are resolved in the definer's schema

Autonomous Transactions

- Are independent transactions started by another main transaction.
- Are specified with `PRAGMA AUTONOMOUS_TRANSACTION`



Features of Autonomous Transactions

- Autonomous transactions:
  - Are independent of the main transaction
  - Suspend the calling transaction until it is completed
  - Are not nested transactions
  - Do not roll back if the main transaction rolls back
  - Enable the changes to become visible to other transactions upon a commit
  - Are demarcated (started and ended) by individual subprograms, and not nested or anonymous PL/SQL block

Using Autonomous Transactions

- Example:

```
PROCEDURE bank_trans(cardnbr NUMBER,loc NUMBER) IS
BEGIN
  log_usage (cardnbr, loc);
  INSERT INTO txn VALUES (9001, 1000,...);
END bank_trans;
```

```
PROCEDURE log_usage (card_id NUMBER, loc NUMBER)
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO usage
  VALUES (card_id, loc);
  COMMIT;
END log_usage;
```

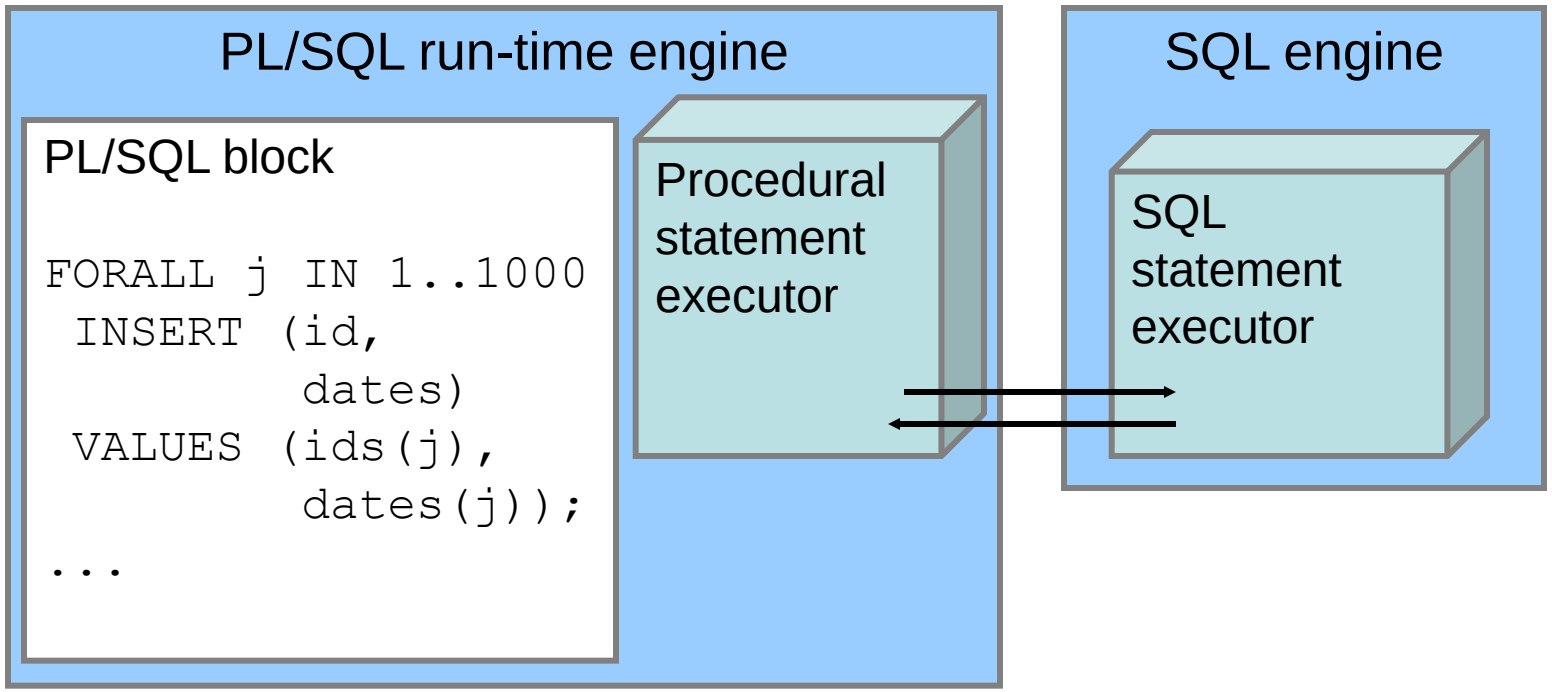
RETURNING Clause

- The RETURNING clause:
  - Improves performance by returning column values with INSERT, UPDATE, and DELETE statements
  - Eliminates the need for a SELECT statement

```
CREATE PROCEDURE update_salary(emp_id NUMBER) IS
  name    employees.last_name%TYPE;
  new_sal employees.salary%TYPE;
BEGIN
  UPDATE employees
    SET salary = salary * 1.1
  WHERE employee_id = emp_id
  RETURNING last_name, salary INTO name, new_sal;
END update_salary;
/
```

Bulk Binding

- Binds whole arrays of values in a single operation, rather than using a loop to perform a `FETCH`, `INSERT`, `UPDATE`, and `DELETE` operation multiple times



Using Bulk Binding

- Keywords to support bulk binding:
  - The `FORALL` keyword instructs the PL/SQL engine to bulk bind input collections before sending them to the SQL engine.

```
FORALL index IN lower_bound .. upper_bound
  [SAVE EXCEPTIONS]
  sql_statement;
```

  - The `BULK COLLECT` keyword instructs the SQL engine to bulk bind output collections before returning them to the PL/SQL engine.

```
... BULK COLLECT INTO
      collection_name[,collection_name] ...
```

Bulk Binding FORALL: Example

```
CREATE PROCEDURE raise_salary(percent NUMBER) IS
  TYPE numlist IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
  id  numlist;
BEGIN
 id(1) := 100; id(2) := 102;
 id(3) := 106; id(3) := 110;
 -- bulk-bind the PL/SQL table
  FORALL i IN id.FIRST .. id.LAST
    UPDATE employees
      SET salary = (1 + percent/100) * salary
      WHERE manager_id = id(i);
END;
/
```

```
EXECUTE raise_salary(10)
```

Using BULK COLLECT INTO with Queries

- The `SELECT` statement has been enhanced to support `BULK COLLECT INTO` syntax. For example:

```
CREATE PROCEDURE get_departments(loc NUMBER) IS
  TYPE dept_tabtype IS
    TABLE OF departments%ROWTYPE;
  depts dept_tabtype;
BEGIN
  SELECT * BULK COLLECT INTO depts
  FROM departments
  WHERE location_id = loc;
  FOR I IN 1 .. depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(depts(i).department_name
     ||' '||depts(i).department_name);
  END LOOP;
END;
```

Using BULK COLLECT INTO with Cursors

- The `FETCH` statement has been enhanced to support `BULK COLLECT INTO` syntax. For example:

```
CREATE PROCEDURE get_departments(loc NUMBER) IS
  CURSOR dept_csr IS SELECT * FROM departments
                     WHERE location_id = loc;
  TYPE dept_tabtype IS TABLE OF dept_csr%ROWTYPE;
  depts dept_tabtype;
BEGIN
  OPEN dept_csr;
  FETCH dept_csr BULK COLLECT INTO depts;
  CLOSE dept_csr;
 FOR I IN 1 .. depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(depts(i).department_name
     ||' '||depts(i).department_name);
  END LOOP;
END;
```

Using BULK COLLECT INTO with a RETURNING Clause

- Example:

```
CREATE PROCEDURE raise_salary(rate NUMBER) IS
   TYPE emplist IS TABLE OF NUMBER;
   TYPE numlist IS TABLE OF employees.salary%TYPE
     INDEX BY BINARY_INTEGER;
   emp_ids  emplist := emplist(100,101,102,104);
   new_sals numlist;
BEGIN
  FORALL i IN emp_ids.FIRST .. emp_ids.LAST
    UPDATE employees
      SET commission_pct = rate * salary
    WHERE employee_id = emp_ids(i)
    RETURNING salary BULK COLLECT INTO new_sals;
  FOR i IN 1 .. new_sals.COUNT LOOP ...
END;
```

Using the NOCOPY Hint

- The NOCOPY hint:
  - Is a request to the PL/SQL compiler to pass OUT and IN OUT parameters by reference rather than by value
  - Enhances performance by reducing overhead when passing parameters

```
DECLARE
  TYPE emptabtype IS TABLE OF employees%ROWTYPE;
  emp_tab emptabtype;
  PROCEDURE populate(tab IN OUT NOCOPY emptabtype)
  IS BEGIN ... END;
BEGIN
  populate(emp_tab);
END;
/
```

Effects of the NOCOPY Hint

- If the subprogram exits with an exception that is not handled:
  - You cannot rely on the values of the actual parameters passed to a `NOCOPY` parameter
  - Any incomplete modifications are not “rolled back”
- The remote procedure call (RPC) protocol enables you to pass parameters only by value.

NOCOPY Hint Can Be Ignored

- The NOCOPY hint has no effect if:
  - The actual parameter:
    - Is an element of an index-by table
    - Is constrained (for example, by scale or NOT NULL)
    - And formal parameter are records, where one or both records were declared by using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ
    - Requires an implicit data type conversion
  - The subprogram is involved in an external or remote procedure call

PARALLEL_ENABLE Hint

- The PARALLEL_ENABLE hint:
  - Can be used in functions as an optimization hint

```
CREATE OR REPLACE FUNCTION f2 (p1 NUMBER)
  RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
  RETURN p1 * 2;
END f2;
```

  - Indicates that a function can be used in a parallelized query or parallelized DML statement

Exam's quizzes

- **1.** Ce este un LOB? Câte tipuri de LOB există?
- **2.** Ce este un BFILE și când sunt folosite?
- **3.** Ce probleme apar la folosirea LOB-urilor în combinație cu triggere și cursoare.