6 Using Explicit Cursors

✅Distinguish between implicit and explicit cursors and use SQL cursor attributes

Cursors overview and implicit: 6.2 - 6.2.1.4

Cursor overview

- a cursor is a pointer to a private sql area that stores information about processing a specific `SELECT` or DML statement
- > Note: the cursors that this topic explains are session cursors. A **session cursor** lives in a session memory until the session ends, when it ceases to exist
- a cursor that is constructed and managed by plsql is an **implicit cursor**
- a cursor that you construct and manage is an **explicit cursor**
- you can get information about any session cursor from its attributes (which you can reference in procedural statements, but not in sql statements)
- to list the session cursors that each user session currently has opened and parsed, query the dynamic performance view `V$OPEN_CURSOR`
- the number of cursors that a session can have open simultaneously is determined by:
  - the amount of memory available to the session
  - the value of the initialization parameter `OPEN_CURSORS`
- > Note: generally, plsql parses an explicit cursor only the first time the session opens it and parses a sql statement (creating an implicit cursor) only the first time the statement runs. All parsed sql statements are cached. A sql statement is reparsed only if it is aged out of the cache by a new sql statement. Although you must close an explicit cursor before you can reopen it, plsql need not reparse the associated query. If you close and immediately reopen an explicit cursor, plsql does not reparse the associated query

Implicit cursors

- an **implicit cursor** is a session cursor that is constructed and managed by plsql
- plsql opens an implicit cursor every time you run a `SELECT` or DML statement
- you cannot control an implicit cursor, but you can get information from its attributes
- the syntax of an implicit cursor attribute value is SQL*attribute* (therefore, an implicit cursor is also called a **SQL cursor**)
- SQL*attribute* always refers to the most recently run `SELECT` or DML statement. If no such statement has run, the value of SQL*attribute* is `NULL`
- an implicit cursor closes after its associated statement runs; however, its attribute values remain available until another `SELECT` or DML statement runs
- the most recently run `SELECT` or DML statement might be in a different scope. To save an attribute value for later use, assign it to a local variable immediately. Otherwise, other operations, such as subprogram invocations, might change the value of the attribute before you can test it
- the implicit cursor attributes are:

- `SQL%ISOPEN`: is the cursor open? always returns `FALSE` because an implicit cursor always closes after its associated statement runs
- `SQL%FOUND`: were any rows affected? returns:
  - `NULL` if no `SELECT` or DML statement has run
  - `TRUE` if a `SELECT` statement returned one or more rows or a DML statement affected one or more rows
  - `FALSE` otherwise
- `SQL%NOTFOUND`: were no rows affected? returns the opposite of `SQL%FOUND` is not useful with the plsql `SELECT INTO` statement, because:
  - if the `SELECT INTO` statement returns no rows, plsql raises the predefined exception `NO_DATA_FOUND` immediately, before you can check `SQL%NOTFOUND`
  - a `SELECT INTO` statemnet that invokes a sql aggregate function always returns a value (possibly `NULL`). After such a statemnet, the `SQL%NOTFOUND` attribute is always `FALSE`, so checking it is unnecessary
- `SQL%ROWCOUNT`: how many rows were affected? returns:
  - `NULL` if no `SELECT` or DML statement has run
  - otherwise, the number of rows returned by a `SELECT` statement or affected by a DML statement (an `INTEGER`)

  If a `SELECT INTO` statement without `BULK COLLECT` clause returns multiple rows, plsql raises the predefined exception `TOO_MANY_ROWS` and `SQL%ROWCOUNT` returns 1, not the actual number of rows that satisfy the query. The value of `SQL%ROWCOUNT` attribute is unrelated to the state of a transaction, therefore:
  - when a transaction rolls back to a savepoint, the value of `SQL%ROWCOUNT` is not restored to the value it had before the savepoint
  - when an autonomous transaction ends, `SQL%ROWCOUNT` is not restored to the original value in the parent transaction
- `SQL%BULK_ROWCOUNT`
- `SQL%BULK_EXCEPTIONS`

✅Declare and control explicit cursors, use simple loops and cursor FOR loops to fetch data

Explicit cursors: 6.2.2 - 6.2.2.5, 6.2.2.7

Control cursors and cursor FOR LOOP: 6.3.2 - 6.3.4

Explicit cursors

- An **explicit cursor** is a named pointer to a private SQL area that stores information for processing a specific query or DML statement—typically, one that returns or affects multiple rows
- You can use an explicit cursor to retrieve the rows of a result set one at a time

- an **explicit cursor** is a session cursor that you construct and manage. You must declare and define an explicit cursor, giving it a name and associating it with a query (typically, the query returns multiple rows). Then you can process the query result set in either of these ways:
    - open the explicit cursor (with the `OPEN` statement), fetch rows from the result set (with the `FETCH` statement), and close the explicit cursor (with the `CLOSE` statement)
    - use the explicit cursor in a cursor `FOR LOOP` statement
- you cannot assign a value to an explicit cursor, use it in an expression, or use it as a formal subprogram parameter or host variable. You *can* do those things with a cursor variable
- unlike an implicit cursor, you can reference an explicit cursor or cursor variable by its name. Therefore, an explicit cursor or cursor variable is called a **named cursor**

Declaring and defining explicit cursors

- you can either declare an explicit cursor first and then define it later in the same block, subprogram, or package, or declare and define it at the same time
- explicit cursor declaration and definition are also called **cursor specification** and **cursor body**, respectively
- > Note: an explicit cursor declared in a package specification is affected by the AUTHID clause of the package
- an expicit cursor declaration, which only declares a cursor, has this syntax: `CURSOR cursor_name [parameter_list] RETURN return_type;`
- an explicit cursor definition has this syntax: `CURSOR cursor_name [parameter_list] [RETURN return_type] IS select_statement;` select_statement cannot have a `WITH` clause
- parameter_list: `parameter_name [IN] datatype [{:= | DEFAULT} expression]`
- return_type: `{{db_table_or_view | cursor | cursor_variable} % ROWTYPE | record % TYPE | record_type}`
- if you declared the cursor earlier, then the explicit cursor definition defines it; otherwise, it both declares and defines it

Opening and closing explicit cursors

- after declaring and defining an explicit cursor, you can open it with the `OPEN` statement, which does the following:
    1. allocates database resources to process the query
    2. processes the query; that is: a. identifies the result set (if the query references variables or cursor parameters, their values affect the result set) b. if the query has a `FOR UPDATE` clause, locks the rows of the result set
    3. positions the cursor before the first row of the result set
    - syntax: `OPEN cursor [ (actual_cursor_parameter [,...]) ] ;`
- you close an open explicit cursor with the `CLOSE` statement, thereby allowing its resources to be reused
    - after closing a cursor, you cannot fetch records from its result set or reference its attributes. If you try, plsql raises the predefined exception `INVALID_CURSOR`
    - you can reopen a closed cursor. You must close an explicit cursor before you try to reopen it. Otherwise, plsql raises the predefined exception `CURSOR_ALREADY_OPEN`
    - after closing a cursor variable, you can reopen it with the `OPEN FOR` statement. You need not close a cursor variable before reopening it

- syntax: `CLOSE {cursor | cursor_variable | :host_cursor_variable} ;`

Fetching data with explicit cursors

- after opening an explicit cursor, you can fetch the rows of the query result set with the `FETCH` statement
- full syntax:

```
FETCH {cursor | cursor_variable | :host_cursor_variable}
{
    INTO {variable [, ...] | record} |
    BULK COLLECT INTO {collection [,...] | :host_array [,...]} [LIMIT
numeric_expression]
} ;
```

- if you try to fetch from an explicit cursor or cursor variable before opening it or after closing it, plsql raises the predefined exception `INVALID_CURSOR`
- the into_clause is either a list of variables or a single record variable. For each column that the query returns, the variable list or record must have a corresponding type-compatible variable or field. The `%TYPE and %ROWTYPE` attributes are useful for declaring variables and records for use in `FETCH` statements
- the `FETCH` statement retrieves the current row of the result set, stores the column values of that row into the variables or record, and advances the cursor to the next row
- typically, you use the `FETCH` statement inside a `LOOP` statement, which you exit when the `FETCH` statement runs out of rows. To detect this exit condition, use the cursor attribute `%NOTFOUND`. Plsql does not raise an exception when a `FETCH` statement returns no rows

Variables in explicit cursor queries

- an explicit cursor query can reference any variable in its scope. When you open an explicit cursor, plsql evaluates any variables in the query and uses those values when identifying the result set. Changing the values of the variables later does not change the result set
- to change the result set, you must close the cursor, change the value of the variable, and then open the cursor again

When explicit cursor queries need column aliases

- when an explicit cursor query includes a virtual column (an expression), that column must have an alias if either of the following is true:
  - you use the cursor to fetch into a record that was declared with `%ROWTYPE`
  - you want to reference the virtual column in your program
- e.g. `CURSOR c1 IS SELECT id, salary+commission total, salary*bonus_pct bonus FROM ...` here `salary+commission` and `salary*bonus_pct` are virtual columns, and must have an alias if above is true

Explicit cursor attributes

- the syntax for the value of an explicit cursor attribute is *cursor_name* immediately followed by *attribute* (e.g. `c1%ISOPEN`)
- the explicit cursor attributes are: `%ISOPEN, %FOUND, %NOTFOUND, %ROWCOUNT`
- if an explicit cursor is not open, referencing any attribute except `%ISOPEN` raises the predefined exception `INVALID_CURSOR`
- **%ISOPEN: is the cursor open?**
  - returns `TRUE` if its explicit cursor is open; `FALSE` otherwise
  - useful for:
    - checking that an explicit cursor is not already open before you try to open it

      > if you try to open an explicit cursor that is already open, plsql raises the predefined exception `CURSOR_ALREADY_OPEN`. You must close an explicit cursor before you can reopen it (this does not apply to cursor variables)

    - checking that an explicit cursor is open before you try to close it
- **%FOUND: has a row been fetched?**
  - returns:
    - `NULL` after the explicit cursor is opened but before the first fetch
    - `TRUE` if the most recent fetch from the explicit cursor returned a row
    - `FALSE` otherwise
  - `%FOUND` is useful for determining whether there is a fetched row to process
- **%NOTFOUND: has no row been fetched?**
  - `NULL` after the explicit cursor is opened but before the first fetch
  - `FALSE` if the most recent fetch from the explicit cursor returned a row
  - `TRUE` otherwise
  - `%NOTFOUND` is useful for exiting a loop when `FETCH` fails to return a row
- **%ROWCOUNT how many rows were fetched?**
  - returns:
    - zero after the explicit cursor is opened but before the first fetch
    - otherwise, the number of rows fetched (an `INTEGER`)

Control cursors and cursor FOR LOOP

Cursor FOR LOOP

- the cursor `FOR LOOP` statement lets you run a `SELECT` statement and then immediately loop through the rows of the result set
- this statement can use either an implicit or explicit cursor (but not a cursor variable)
- if you use the `SELECT` statement only in the cursor `FOR LOOP` statement,
  - then specify the `SELECT` statement inside the cursor `FOR LOOP` statement e.g. `FOR rec IN (select_statement) LOOP ...`
  - this form of the cursor `FOR LOOP` statement uses an implicit cursor, and is called an **implicit cursor FOR LOOP statement**.
  - because the implicit cursor is internal to the statement, you cannot reference it with the name `SQL`
- if you use the `SELECT` statement multiple times in the same plsql unit,
  - then define an explicit cursor for it and specify that cursor in the cursor `FOR LOOP` statement.
  - this form of the cursor `FOR LOOP` statement is called an **explicit cursor FOR LOOP statement**.

    - you can use the same explicit cursor elsewhere in the same plsql unit
- the cursor `FOR LOOP` implicitly declares its loop index as a `%ROWTYPE` record variable of the type that its cursor returns
    - this record is local to the loop and exists only during loop execution.
    - statements inside the loop can reference the record and its fields.
    - they can reference virtual columns only by aliases
    - statements outside the loop cannot reference record
    - after the cursor `FOR LOOP` statement runs, record is undefined
- after declaring the loop index record variable, the `FOR LOOP` statement opens the specified cursor
    - with each iteration of the loop, the `FOR LOOP` statement fetches a row from the result set and stores it in the record
    - when there are no more rows to fetch, the cursor `FOR LOOP` statement closes the cursor
    - the cursor also closes if a statement inside the loop transfers control outside the loop or if plsql raises an exception

Processing query result sets with explicit cursors, OPEN, FETCH, and CLOSE

- for full control over query result set processing, declare explicit cursors and manage them with the statements `OPEN, FETCH, and CLOSE`
- this result set processing technique is more complicated than the others, but it is also more flexible, e.g. you can:
    - process multiple result sets in parallel, using multiple cursors
    - process multiple rows in a single loop iteration, skip rows, or split the processing into multiple loops
    - specify the query in one plsql unit but retrieve the rows in another

Processing query result sets with subqueries

- if you process a query result set by looping through it and running another query for each row, then you can improve performance by removing the second query from inside the loop and making it a subquery of the first query
- while an ordinary subquery is evaluated for each table, a **correlated subquery** is evaluated for each row

✅Declare and use cursors with parameters

Explicit cursors accept parameters: 6.2.2.6

Cursor variables: 6.4

Cursor expression: 6.5

Cursors with parameters

- you can create an explicit cursor that has formal parameters, and then pass different actual parameters to the cursor each time you open it
- in the cursor query, you can use a formal cursor parameter anywhere that you can use a constant
- outside the cursor query, you cannot reference formal cursor parameters

- example:

```
CURSOR c (job VARCHAR2, max_sal NUMBER) IS
  SELECT name, (salary - max_sal) pay
  FROM employees
  WHERE job_id = job;
```

- > Tip: to avoid confusion, use different names for formal and actual cursor parameters

- when you create an explicit cursor with formal parameters, you can specify default values for them
    - when a formal parameter has a default value, its corresponding actual parameter is optional
    - if you open the cursor without specifying the actual parameter, then the formal parameter has its default value

- if you add formal parameters to a cursor, and you specify default values for the added parameters, then you need not change existing references to the cursor

Cursor variables

- a cursor variable is like an explicit cursor, except that:
    - it is not limited to one query you can open a cursor variable for a query, process the result set, and then use the cursor variable for another query
    - you can assign a value to it
    - you can use it in an expression
    - it can be a subprogram parameter you can use cursor variables to pass query result sets between plsql stored subprograms and their clients
    - it cannot accept parameters you cannot pass parameters to a cursor variable, but you can pass whole queries to it. The queries can include variables
- a cursor variable has this flexibility because it is a pointer; that is, its value is the address of an item, not the item itself
- before you can reference a cursor variable, you must make it point to a sql work area, either by opening it or by assigning it the value of an open plsql cursor variable or open host cursor variable

- > Note: cursor variables and explicit cursors are not interchangeable, you cannot use one where the other is expected, e.g. you cannot reference a cursor variable in a cursor `FOR LOOP` statement

- other restrictions:
    - you cannot use a cursor variable in a cursor `FOR LOOP` statement
    - you cannot declare a cursor variable in a package specification that is, a package cannot have a public cursor variable (that can be referenced from outside the package)
    - you cannot store the value of a cursor variable in a collection or database column
    - you cannot use comparison operators to test cursor variables for equality, inequality, or nullity
    - using a cursor variable in a server to server remote procedure call (RPC) causes an error. However, you can use a cursor variable in a server-to-server RPC if the remote database is a non-Oracle database accessed through a procedural gateway

Creating cursor variables

- to create a cursor variable, either declare a variable of the predefined type `SYS_REFCURSOR` or define a `REF CURSOR` type and then declare a variable of that type
- > Note: informally, a cursor variable is sometimes called a `REF CURSOR`
- basic syntax of a `REF CURSOR` type definition is: `TYPE type_name IS REF CURSOR [RETURN return_type]`
- if you specify return_type, then the `REF CURSOR` type and cursor variables of that type are **strong**, if not, they are **weak**. `SYS_REFCURSOR` and cursor variables of that type are weak
  - with a strong cursor variable, you can associate only queries that return the specfied type.
  - with a weak cursor variable, you can associate any query
- weak cursor variables are more error-prone than strong ones, but they are also more flexible
  - weak `REF CURSOR` types are interchangeable with each other and with the predefined type `SYS_REFCURSOR`
  - you can assign the value of a weak cursor variable to any other weak cursor variable
- you can assign the value of a strong cursor variable to another strong cursor variable only if both cursor variables have the same type (not merely the same return type)

Opening and closing cursor variables

- after declaring a cursor variable, you can open it with the `OPEN FOR` statement, which does the following:

  1. associated the cursor variable with a query The query can include placeholders for bind variables, whose values you specify in the `USING` clause of the `OPEN FOR` statement
  2. allocates database resources to process the query
  3. processes the query; that is: a. identifies the result set (if the query references variables, their values affect the result set) b. if the query has a `FOR UPDATE` clause, locks the rows of the result set
  4. positions the cursor before the first row of the result set

- you need not close a cursor variable before reopening it (that is, using it in another `OPEN FOR` statement).
  - after you reopen a cursor variable, the query previously associated with it is lost
- when you no longer need a cursor variable, close it with the `CLOSE` statement, thereby allowing its resources to be reused
  - after closing a cursor variable, you cannot fetch records from its result set or reference its attributes. If you try, plsql raises the predefined exception `INVALID_CURSOR`
- you can reopen a closed cursor variable
- `OPEN FOR` syntax and semantics:

```
OPEN {cursor_variable | :host_cursor_variable} FOR {select_statement |
dynamic_string}
```

```
[USING {IN | OUT | IN OUT} bind_argument [,...] ] ;
```

dynamic_string is a string literal, string variable, or string expression of the data type `CHAR, VARCHAR2, or CLOB`, which represents a sql `SELECT` statement Use using_clause if and only if select_statement or dynamic_sql_stmt includes placeholders for bind variables

Fetching data with cursor variables

- after opening a cursor variable, you can fetch the rows of the query resutl set with the `FETCH` statement
- the return type of the cursor variable must be compatible with the into_clause of the `FETCH` statement
    - if the cursor variable is strong, plsql catches incompatibility at compile time
    - if the cursor variable is weak, plsql catches incompatibility at run time, raising the predefined exception `ROWTYPE_MISMATCH` before the first fetch

Assigning values to cursor variables

- you can assign to a plsql cursor variable the value of another plsql cursor variable or host cursor variable
- syntax: `target_cursor_variable := source_cursor_variable;`
- if source_cursor_variable is open, then after the assignment, target_cursor_variable is also open. The two cursor variables point to the same sql work area
- if source_cursor_variable is not open, opening target_cursor_variable after the assignment does not open source_cursor_variable

Variables is cursor variable queries

- the query associated with a cursor variable can reference any variable in its scope
- when you open a cursor variable with the `OPEN FOR` statement, plsql evaluates any variables in the query and uses those values when identifying the result set
- changing the values of the variables later does not change the result set
- to change the result set, you must change the value of the variable and then open the cursor variable again for the same query

Querying a collection

- you can query a collection if all of the following are true:
    - the data type of the collection was either created at schema level or declared in a package specification
    - the data type of the collection element is either a scalar data type, a user-defined type, or a record type
- in the query `FROM` clause, the collection appears in table_collection_expression as the argument of the `TABLE` operator
- > Note: in sql contexts, you cannot use a function whose return type was declared in a package specification
- Example

```
CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS
  TYPE rec IS RECORD(f1 NUMBER, f2 VARCHAR2(30));
  TYPE mytab IS TABLE OF rec INDEX BY pls_integer;
END;
/
DECLARE
  v1 pkg.mytab;  -- collection of records
  v2 pkg.rec;
  c1 SYS_REFCURSOR;
BEGIN
  v1(1).f1 := 1;
  v1(1).f2 := 'one';
  OPEN c1 FOR SELECT * FROM TABLE(v1);
  FETCH c1 INTO v2;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE('Values in record are ' || v2.f1 || ' and ' ||
v2.f2);
END;
/
-- result:
Values in record are 1 and one
```

Cursor variable attributes

- a cursor variable has the same attributes as an explicit cursor
- the syntax for the value of a cursor variable attribute is cursor_variable_name immediately followed by attribute (e.g. cv%ISOPEN)
- if a cursor variable is not open, referencing any attribute except `%ISOPEN` raises the predefined exception `INVALID_CURSOR`

Cursor variables as subprogram parameters

- you can use a cursor variable as a subprogram parameter, which makes it useful for passing query results between subprograms, e.g.
  - you can open a cursor variable in one subprogram and process it in a different subprogram
  - in a multilanguage application, a plsql subprogram can use a cursor variable to return a result set to a subprogram written in a different language

> Note: the invoking and invoked subprograms must be in the same database instance. You cannot pass or return cursor variables to subprograms invoked through database links

- when declaring a cursor variable as the formal parameter of a subprogram:
  - if the subprogram opens or assigns a value to the cursor variable, then the parameter mode must be `IN OUT`
  - if the subprogram only fetches from, or closes, the cursor variable, then the parameter mode can be either `IN` or `IN OUT`
- corresponding formal and actual cursor variable parameters must have compatible return types. Otherwise plsql raises the predefined exception `ROWTYPE_MISMATCH`

- to pass a cursor variable parameter between subprograms in different plsql units, define the `REF CURSOR` type of the parameter in a package. When the type is in a package, multiple subprograms can use it. One subprogram can declare a formal parameter of that type, and other subprograms can declare variables of that type and pass them to the first subprogram

Cursor variables as host variables

- you can use a cursor variable as a host variable, which makes it useful for passing query results between plsql stored subprograms and their clients
- when a cursor variable is a host variable, plsql and the client (the host environment) share a pointer to the sql work area that stores the result set
- to use a cursor variable as a host variable, declare the cursor variable in the host environment and then pass it as an input host variable (bind variable) to plsql
- host cursor variables are compatible with any query return type (like weak plsql cursor variables)
- a sql work area remains accessible while any cursor variable points to it, even if you pass the value of a cursor variable from one scope to another
- if you have a plsql engine on the client side, calls from client to server impose no restrictions. e.g. you can declare a cursor variable on the client side, open and fetch from it on the server side, and continue to fetch from it on the client side

Cursor expressions

- a `CURSOR` expression returns a nested cursor
- syntax: `CURSOR ( subquery )`
- you can use a `CURSOR` expression in a `SELECT` statement that is not a subquery or pass it to a function that accepts a cursor variable parameter, e.g. `CURSOR c1 IS SELECT id, CURSOR (SELECT name FROM ...) FROM ... FETCH c1 INTO v_id, c_my_ref_cur`
- you cannot use a cursor expression with an implicit cursor

✅Lock rows with the FOR UPDATE clause and reference the current row with the WHERE CURRENT OF clause

cursors FOR UPDATE, WHERE CURRENT OF: 6.6.6.2 - 6.6.6.3

SELECT FOR UPDATE clause and FOR UPDATE cursors

- the `SELECT` statement with the `FOR UPDATE` clause selects the rows of the result set and locks them.
- `SELECT FOR UPDATE` lets you base an update on the existing values in the rows, because it ensures that no other user can change those values before you update them
- you can also use `SELECT FOR UPDATE` to lock rows that you do not want to update
- by default, the `SELECT FOR UPDATE` statement waits until the requested row lock is acquired. To change this behaviour, use the `NOWAIT, WAIT or SKIP LOCKED` clause of the `SELECT FOR UPDATE` statement
- when `SELECT FOR UPDATE` is associated with an explicit cursor, the cursor is called `FOR UPDATE cursor`. Only a `FOR UPDATE` cursor can appear in the `CURRENT OF` clause of an `UPDATE or DELETE` statement

- the `CURRENT OF` clause, a plsql extension to the `WHERE` clause of the sql statements `UPDATE` and `DELETE` restricts the statement to the current row of the cursor.
- when `SELECT FOR UPDATE` queries multiple tables, it locks only rows whose columns appear in the `FOR UPDATE` clause
- example:

```
DECLARE
  CURSOR c1 IS SELECT * FROM emp FOR UPDATE OF chara ORDER BY id;
  crecord emp%ROWTYPE;
BEGIN
  OPEN c1;
  FETCH c1 INTO crecord; -- row where id = 1
  FETCH c1 INTO crecord; -- row where id = 2
  UPDATE emp SET chara = 'hellow' WHERE CURRENT OF c1;
  -- will update the row where id = 2
  COMMIT; -- releases locks (same with ROLLBACK), cannot fetch from FOR
UPDATE cursor when rows unlocked
  FETCH c1 INTO crecord; -- statement fails (use ROWID workaround), data
still committed
  CLOSE c1;
END;
```

Simulating CURRENT OF clause with ROWID pseudocolumn

- the rows of the result set are locked when you open a `FOR UPDATE` cursor, not as they are fetched
- the rows are unlocked when you commit or roll back the transaction
- after the rows are unlocked, you cannot fetch from the `FOR UPDATE` cursor `ORA-01002 fetch out of sequence`
- the workaround is to simulate the `CURRENT OF` clause with the `ROWID` pseudocolumn
- select the rowid of each row into a `UROWID` variable and use the rowid to identify the current row during subsequent updates and deletes (to print the value of a `UROWID` variable, convert it to `VARCHAR2` using the `ROWIDTOCHAR` function)

> Caution: because no `FOR UPDATE` clause locks the fetched rows, other users might unintentionally overwrite your changes

> Note: the extra space needed for read consistency is not released until the cursor is closed, which can slow down processing for large updates