

3 Writing SQL in PL/SQL

✓ Create PL/SQL executable blocks using DML and transaction control statements

DML, static SQL: 6 - 6.1.1

INSERT extension

UPDATE extension

DELETE extension

Transaction processing and control: 6.6 - 6.6.6.1

Static SQL

- static sql is plsql feature that allows sql syntax directly in a plsql statement
- these are the plsql static sql statement, which have the same syntax as the corresponding sql statements, except as noted:
 - **SELECT**, plsql syntax: **SELECT INTO** (see below chapter)
 - DML statements:
 - **INSERT**, has plsql extension
 - **UPDATE**, has plsql extension
 - **DELETE**, has plsql extension
 - **MERGE**
 - TCL statements:
 - **COMMIT**
 - **ROLLBACK**
 - **SAVEPOINT**
 - **SET TRANSACTION**
 - **LOCK TABLE**
- a plsql static sql statement can have a plsql identifier wherever its sql counterpart can have a placeholder for a bind variable. The plsql identifier must identify either a variable or a formal parameter
- to use plsql identifiers for table names, column names, and so on, use the **EXECUTE IMMEDIATE** statement

DML statements in PL/SQL

- **INSERT** statement extension
 - the plsql extension to the sql **INSERT** statement lets you specify a record name in the `values_clause` of the `single_table_insert` instead of specifying a column list in the `insert_into_clause`
 - effectively, this form of the **INSERT** statement inserts the record into the table; actually it adds a row to the table and gives each column of the row the value of the corresponding record field
 - syntax: **INSERT INTO dml_table_expression_clause [t_alias] VALUES record**

- record must represent a row of the item explained by `dml_table_expression_clause`. That is, for every column of the row, the record must have a field with a compatible data type. If a column has a **NOT NULL** constraint, then its corresponding field cannot have a **NULL** value
- **UPDATE** statement extension
 - plsql extends the `update_set_clause` and `where_clause` of the sql **update** statement as follows:
 - in the `update_set_clause`, you can specify a record (**SET ROW =** instead of **SET column_name =**). For each selected row, the update statement updates each column with the value of the corresponding record field
 - in the `where_clause`, you can specify a **CURRENT OF** clause, which restricts the update statement to the current row of the specified cursor
 - syntax: **UPDATE ... SET ROW = record ... WHERE CURRENT OF for_update_cursor**
 - `for_update_cursor`: name of a **FOR UPDATE** cursor; an explicit cursor associated with a **FOR SELECT UPDATE** statement
- **DELETE** statement extension
 - plsql extension to the `where_clause` of the sql **DELETE** statement lets you specify a **CURRENT OF clause** which restricts the **DELETE** statement to the current row of the specified cursor

Transaction processing and control

- **transaction processing** is an oracle database feature that lets multiple users work on the database concurrently and ensures that each user sees a consistent version of data and that all changes are applied in the right order
- a **transaction** is a sequence of one or more sql statements that oracle treats as a unit: either all of the statements are performed or none of them are
- different users can write to the same data structures without harming each other's data or coordinating with each other, because oracle locks data structures automatically
- you rarely must write extra code to prevent problems with multiple users accessing data concurrently, but if you need to you can manually override the default locking mechanism
- **COMMIT** statement
 - the **COMMIT** statement ends the current transaction, making its changes permanent and visible to other users
 - a transaction can span multiple blocks and a block can contain multiple transactions
 - the **WRITE** clause of the **COMMIT** statement specifies the priority with which oracle writes to the redo log the information that the commit operation generates
 - the default plsql commit behaviour for nondistributed transactions is **BATCH NOWAIT** if the **COMMIT_LOGGING** and **COMMIT_WAIT** database initialization parameters have not been set
- **ROLLBACK** statement
 - the **ROLLBACK** statement ends the current transaction and undoes any changes made during that transaction
 - if you make a mistake, such as deleting the wrong row from a table, a rollback restores the original data. if you cannot finish a transaction because a sql statement fails or plsql raises an exception, a rollback lets you take corrective action and perhaps start over
- **SAVEPOINT** statement

- the **SAVEPOINT** statement names and marks the current point in the processing of a transaction
- savepoints let you roll back part of a transaction instead of the whole transaction. The number of active savepoints for each session is unlimited
- when you rollback to a savepoint, any savepoints marked after that savepoint are erased. The savepoint to which you roll back is not erased. A simple rollback or commit erases all savepoints
- if you mark a savepoint in a recursive subprogram, new instances of the **SAVEPOINT** statement run at each level in the recursive descent, but you can only roll back to the most recently marked savepoint
- savepoint names are undeclared identifiers. Reusing a savepoint name in a transaction moves the savepoint from its old position to the current point in the transaction, which means that a rollback to the savepoint affects only the current part of the transaction
- **Implicit rollbacks**
 - before running an **INSERT**, **UPDATE**, **DELETE** or **MERGE** statement, the database marks an implicit savepoint (unavailable to you). If the statement fails, the database rolls back to the savepoint
 - usually, just the failed sql statement is rolled back, not the whole transaction. If the statement raises an unhandled exception, the host environment determines what is rolled back
 - the database can also roll back single sql statements to break deadlocks. The database signals an error to a participating transaction and rolls back the current statement in that transaction
 - before running an sql statement, the database must parse it, that is, examine it to ensure it follows syntax rules and refers to valid schema objects. Errors detected while running a sql statement cause a rollback, but errors detected while parsing the statement do not
 - if you exit a stored subprogram with an unhandled exception, plsql does not assign values to out parameters and does not do any rollback
- **SET TRANSACTION** statement
 - you use the **SET TRANSACTION** statement to begin a read-only or read-write transaction, establish an isolation level, or assign your current transaction to a specified rollback segment
 - read-only transactions are useful for running multiple queries while other users update the same tables
 - during a read only transaction, all queries refer to the same snapshot of the database, providing a multi-table, multi-query, read consistent view. Other users can continue to query or update data as usual. A commit or rollback ends the transaction
 - the **SET TRANSACTION** statement must be the first sql statement in a read only transaction and can appear only once in a transaction. If you set a transaction to **READ ONLY**, subsequent queries see only changes committed before the transaction began. The use of **READ ONLY** does not affect other users or transactions
 - only the **SELECT**, **OPEN**, **FETCH**, **CLOSE**, **LOCK TABLE**, **COMMIT**, **ROLLBACK** statements are allowed in a read-only transaction. Queries cannot be **FOR UPDATE**
- **Overriding default locking**
 - by default, oracle locks data structures automatically, which lets different applications write to the same data structures without harming each other's data or coordinating with each other
 - if you must have exclusive access to data during a transaction, you can override default locking with these sql statements:
 - **LOCK TABLE**, which explicitly locks entire tables
 - **SELECT** with the **FOR UPDATE** clause (**SELECT FOR UPDATE**), which explicitly locks specific rows of a table

- **LOCK TABLE** statement
 - the **LOCK TABLE** statement explicitly locks one or more tables in a specified lock mode so that you can share or deny access to them
 - the lock mode determines what other locks can be placed on a table at the same time, but only one user at a time can acquire an exclusive lock. While other users can insert, delete, or update rows in that table
 - a table lock never prevents other users from querying a table, and a query never acquires a table lock. Only if two different transactions try to modify the same row does one transaction wait for the other to complete. The **LOCK TABLE** statement lets you specify how long to wait for another transaction to complete
 - table locks are released when the transaction that acquired them is either committed or rolled back

✓ Make use of the INTO clause to hold the values returned by a SQL statement

Processing query result with SELECT INTO: 6.3.1

SELECT INTO syntax

- using an implicit cursor, the **SELECT INTO** statement retrieves values from one or more database tables (as the sql **SELECT** statement does) and stores them in variables (which the sql **SELECT** statement does not do)
- **Handling single-row result sets**
 - if you expect the query to return only one row, then use the **SELECT INTO** statement to store values from that row in either one or more scalar variables, or one record variable
 - if the query might return multiple rows, but you care about only the nth row, then restrict the result set to that row with the clause **WHERE ROWNUM = n**
- **Handling large multiple-row result sets**
 - if you must assign a large quantity of table data to variables, oracle recommends using the **SELECT INTO** statement with the **BULK COLLECT** clause
 - this statement retrieves an entire result set into one or more collection variables
- **syntax and semantics:**

```
SELECT [{DISTINCT | UNIQUE | ALL}] select_list
      { INTO {variable,... | record}
        | BULK COLLECT INTO {collection,... | :host_array} }
FROM rest_of_statement;
```

- specify **DISTINCT** or **UNIQUE** if you want the database to return only one copy of each set of duplicate rows selected, they are synonymous
- you cannot specify **DISTINCT** if the select_list contains LOB columns

- **ALL** is the default and causes the database to return all rows selected, including copies of duplicates
- if the **SELECT INTO** statement returns no rows, plsql raises the predefined exception **NO_DATA_FOUND**. To guard against this exception, select the result of the aggregate function **COUNT(*)** which returns a single value even if no rows match the condition
- if the **SELECT INTO** statement returns more rows than one, plsql raises the predefined exception **TOO_MANY_ROWS**, to prevent this and you know that the result is more than one row, use the bulk collect into clause with collection variables