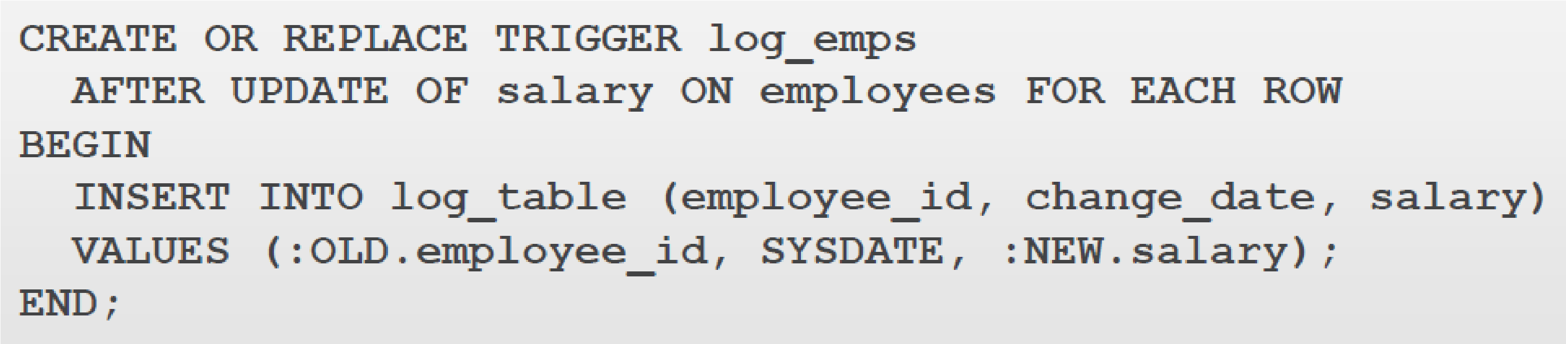
Triggers Revisited

Oracle Academy Study Materials

Row Triggers Revisited

– Example: This row trigger adds a row to the LOG_TABLE whenever an employee's salary changes.

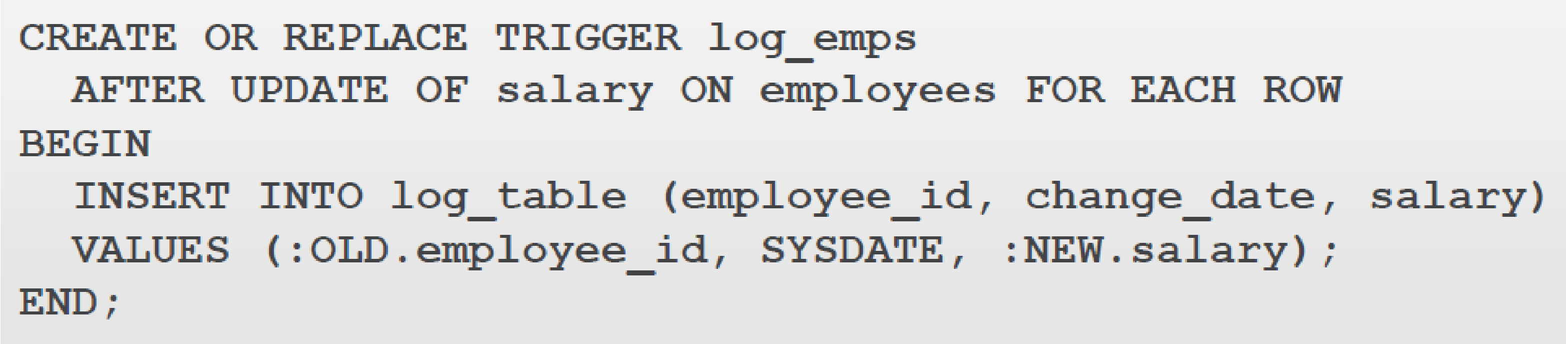


```
CREATE OR REPLACE TRIGGER log_emps
  AFTER UPDATE OF salary ON employees FOR EACH ROW
BEGIN
  INSERT INTO log_table (employee_id, change_date, salary)
  VALUES (:OLD.employee_id, SYSDATE, :NEW.salary);
END;
```

– What if there are one million employees and you give every employee a 5% salary increase?

Row Triggers Revisited

– Example: This row trigger adds a row to the LOG_TABLE whenever an employee's salary changes.



```
CREATE OR REPLACE TRIGGER log_emps
  AFTER UPDATE OF salary ON employees FOR EACH ROW
BEGIN
  INSERT INTO log_table (employee_id, change_date, salary)
  VALUES (:OLD.employee_id, SYSDATE, :NEW.salary);
END;
```

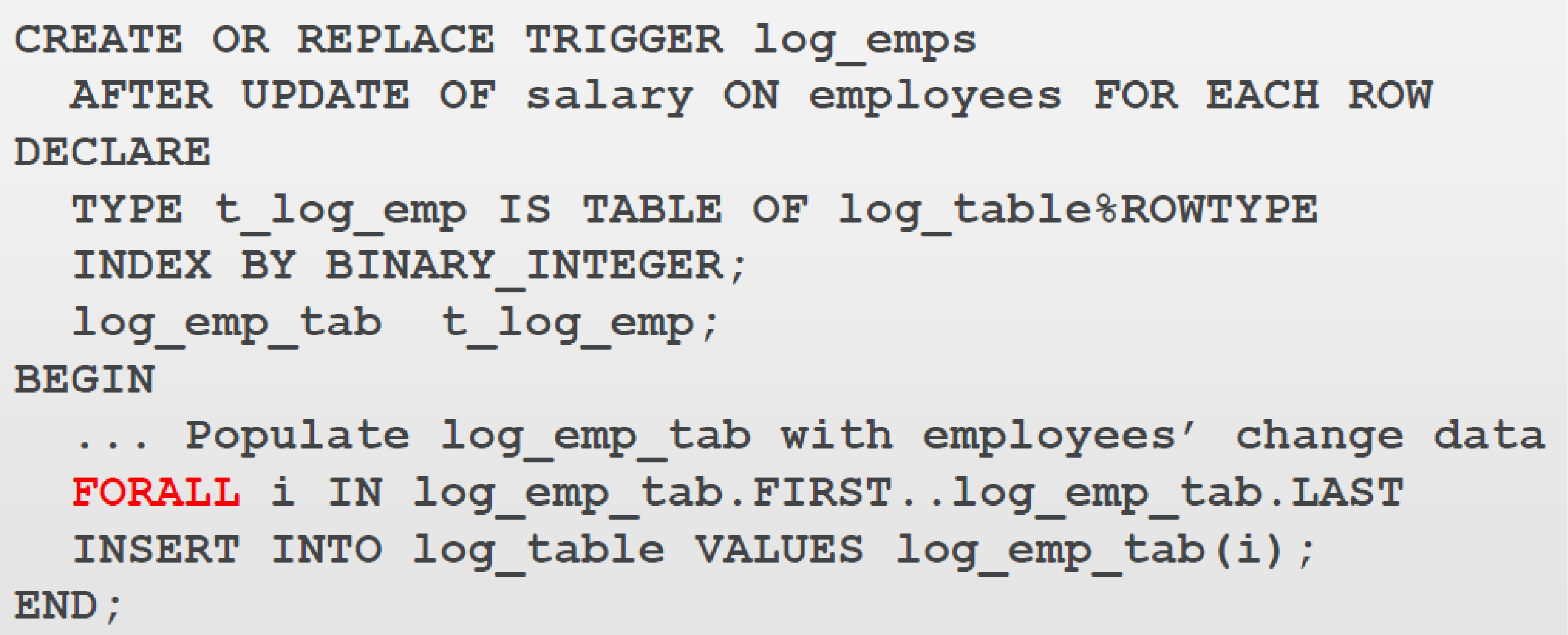
– What if there are one million employees and you give every employee a 5% salary increase?

– The row trigger will automatically execute one million times, inserting one row each time.

– This will be very slow.

Row Triggers Revisited

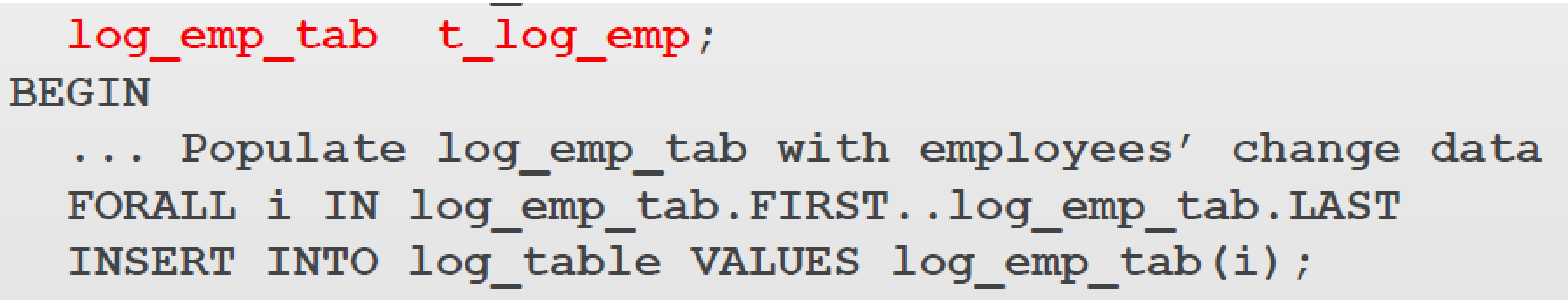
- Can we use FORALL in our trigger?



```
CREATE OR REPLACE TRIGGER log_emps
  AFTER UPDATE OF salary ON employees FOR EACH ROW
DECLARE
  TYPE t_log_emp IS TABLE OF log_table%ROWTYPE
  INDEX BY BINARY_INTEGER;
  log_emp_tab  t_log_emp;
BEGIN
  ... Populate log_emp_tab with employees’ change data
  FORALL i IN log_emp_tab.FIRST..log_emp_tab.LAST
  INSERT INTO log_table VALUES log_emp_tab(i);
END;
```

Row Triggers Revisited

– It is a row trigger - Trigger variables lose scope at the end of each execution of the trigger.



```
   log_emp_tab  t_log_emp;
BEGIN
   ... Populate log_emp_tab with employees' change data
   FORALL i IN log_emp_tab.FIRST..log_emp_tab.LAST
   INSERT INTO log_table VALUES log_emp_tab(i);
```

– So each time the row trigger is fired, all the data already collected in LOG_EMP_TAB is lost.

– To avoid losing this data, we need a trigger that fires only once –a statement trigger.

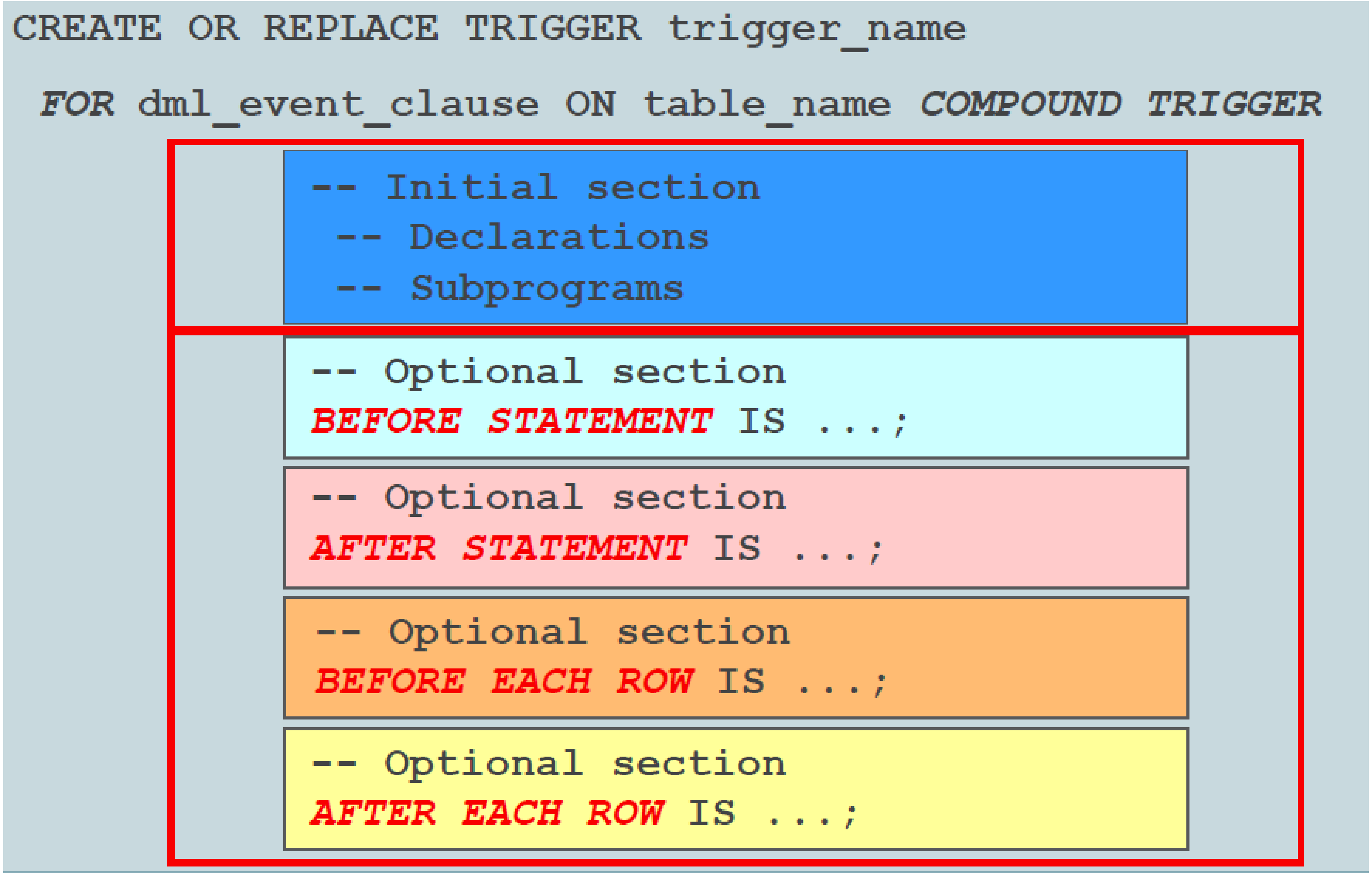
– But to reference column values from each row (using :OLD and :NEW) we need a row trigger.

– But a single trigger cannot be both a row trigger and a statement trigger at the same time.

Compound Trigger

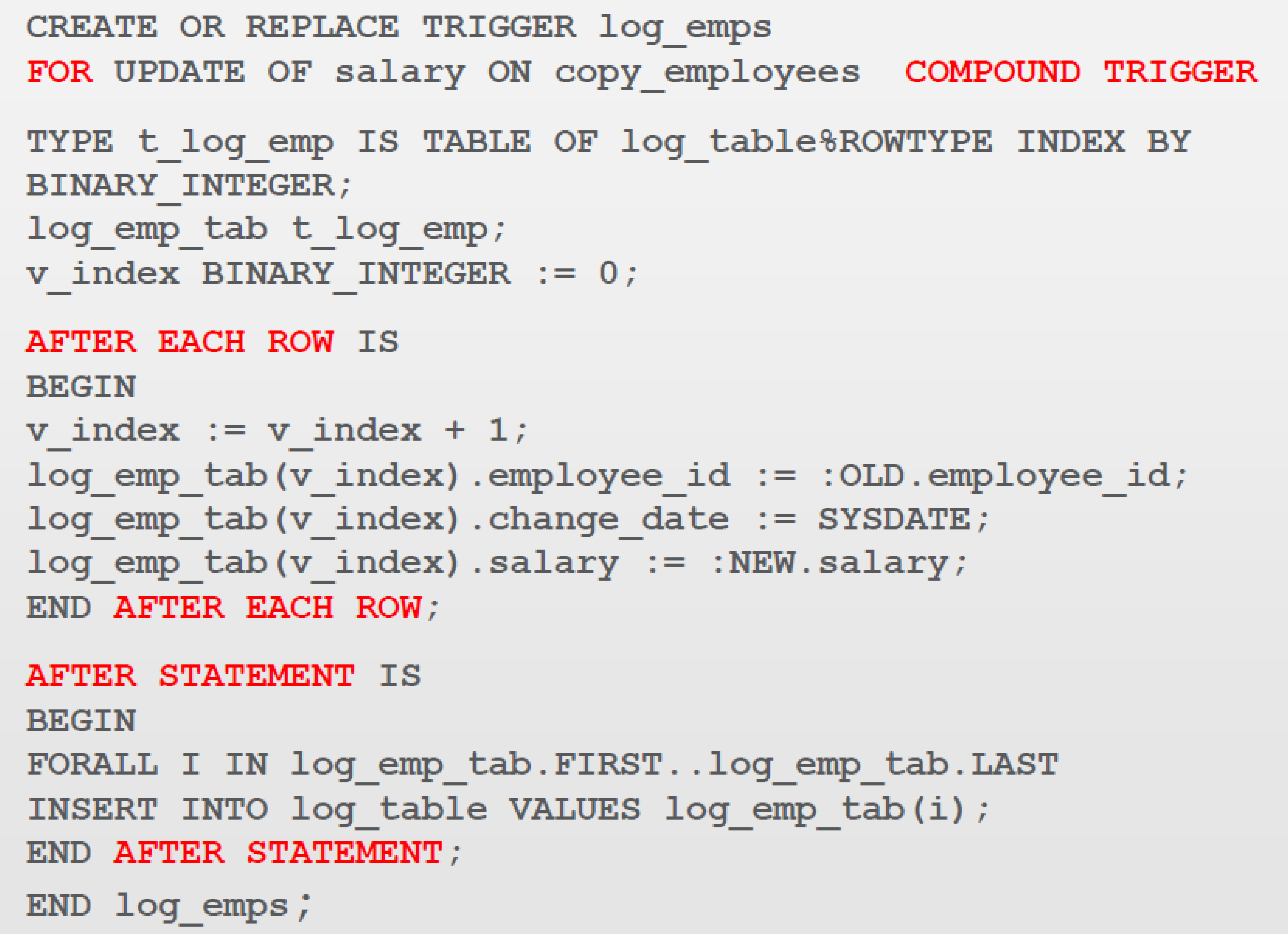
- A **compound trigger** is a single trigger that can include actions for each of the four possible timing points: before the triggering statement, before each row, after each row, and after the triggering statement.
- A Compound Trigger has a declaration section and a section for each of its timing points.
- You do not have to include all the timing points, just the ones you need.
- The scope of Compound Trigger variables is the whole trigger, so they retain their scope throughout the whole execution.

Compound Trigger Structure



```
CREATE OR REPLACE TRIGGER trigger_name
 FOR dml_event_clause ON table_name COMPOUND TRIGGER
 -- Initial section
  -- Declarations
  -- Subprograms
 -- Optional section
 BEFORE STATEMENT IS ...;
 -- Optional section
 AFTER STATEMENT IS ...;
 -- Optional section
 BEFORE EACH ROW IS ...;
 -- Optional section
 AFTER EACH ROW IS ...;
```

Compound Trigger Example

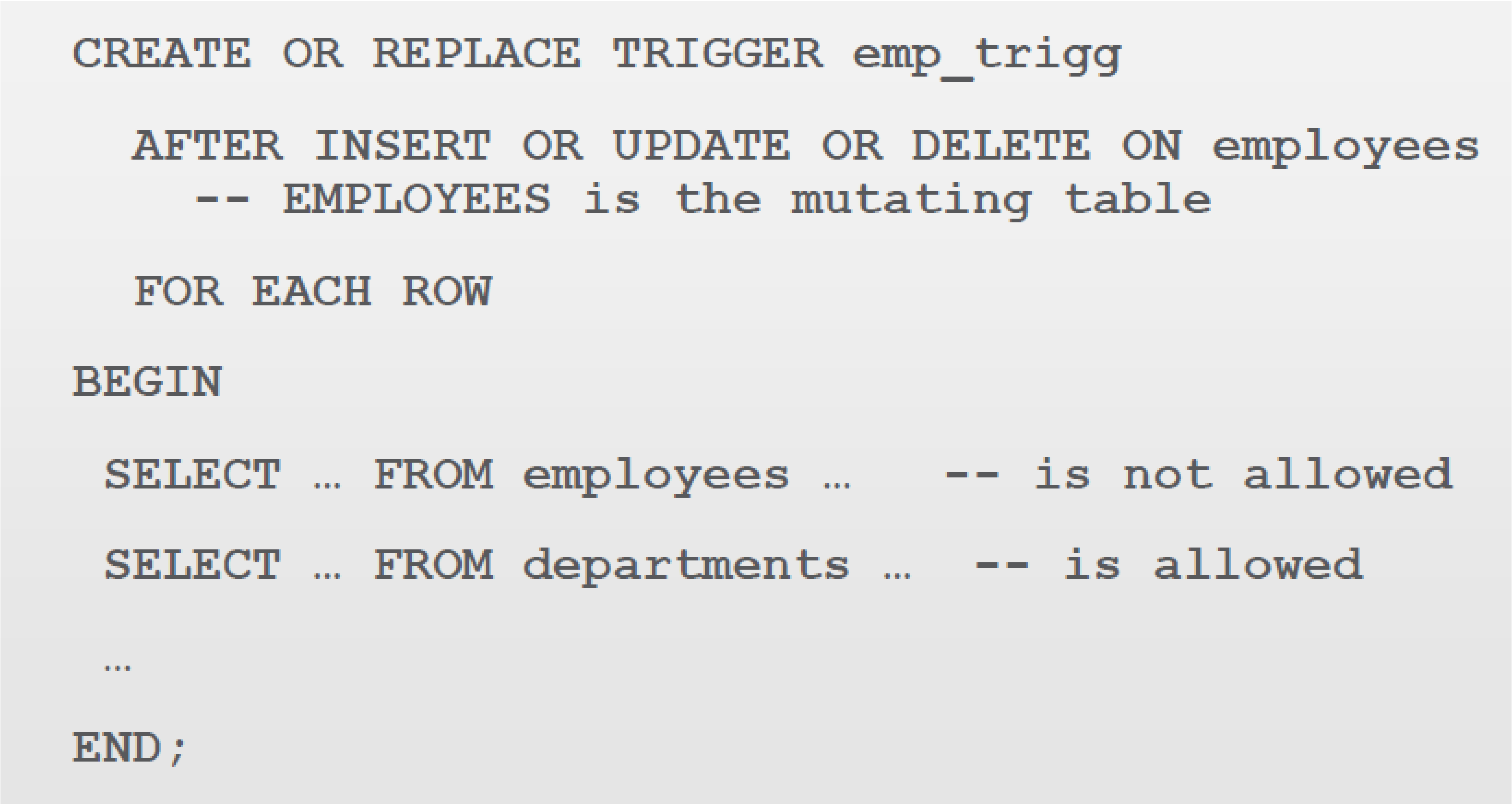


```
CREATE OR REPLACE TRIGGER log_emps
FOR UPDATE OF salary ON copy_employees  COMPOUND TRIGGER

TYPE t_log_emp IS TABLE OF log_table%ROWTYPE INDEX BY
BINARY_INTEGER;
log_emp_tab t_log_emp;
v_index BINARY_INTEGER := 0;

AFTER EACH ROW IS
BEGIN
v_index := v_index + 1;
log_emp_tab(v_index).employee_id := :OLD.employee_id;
log_emp_tab(v_index).change_date := SYSDATE;
log_emp_tab(v_index).salary := :NEW.salary;
END AFTER EACH ROW;

AFTER STATEMENT IS
BEGIN
FORALL I IN log_emp_tab.FIRST..log_emp_tab.LAST
INSERT INTO log_table VALUES log_emp_tab(i);
END AFTER STATEMENT;

END log_emps;
```

Mutating Tables and Row Triggers

- A mutating table is a table that is currently being modified by a DML statement.
- A row trigger cannot SELECT from a mutating table, because it would see an inconsistent set of data (the data in the table would be changing while the trigger was trying to read it).
- However, a row trigger can SELECT from a different table if needed.
- This restriction does not apply to DML statement triggers, only to DML row triggers.
- To avoid mutating table errors:
  - A row-level trigger must not query or modify a mutating table.
  - A statement-level trigger must not query or modify a mutating table if the trigger is fired as the result of a CASCADE delete.
  - Reading and writing data using triggers is subject to certain rules. The restrictions apply only to row triggers, unless a statement trigger is fired as a result of ON DELETE CASCADE.

Mutating Tables and Row Triggers

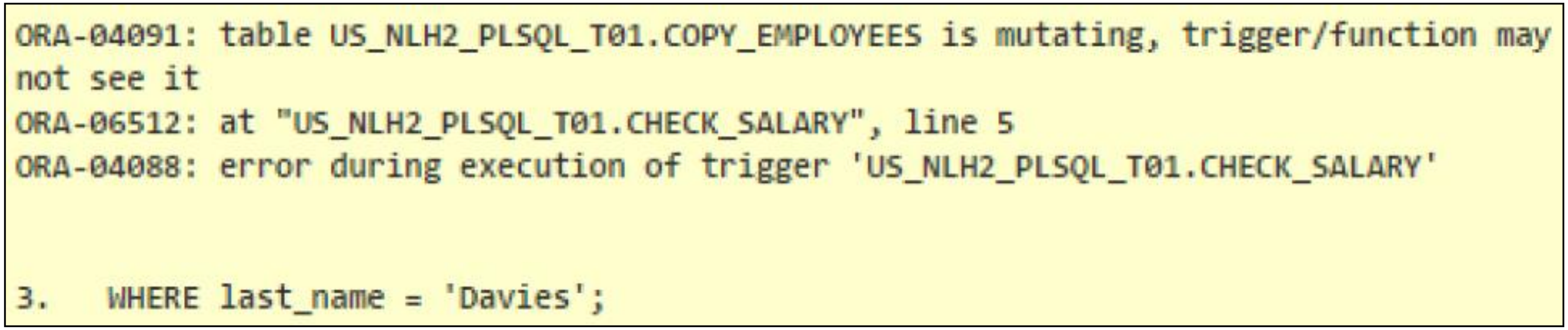


```
CREATE OR REPLACE TRIGGER emp_trigg

  AFTER INSERT OR UPDATE OR DELETE ON employees
    -- EMPLOYEES is the mutating table

  FOR EACH ROW

BEGIN

 SELECT … FROM employees …    -- is not allowed

 SELECT … FROM departments …  -- is allowed

 …

END;
```

Mutating Tables and Row Triggers – Example

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE INSERT OR UPDATE OF salary, job_id ON
employees
  FOR EACH ROW
DECLARE
  v_minsalary employees.salary%TYPE;
  v_maxsalary employees.salary%TYPE;
BEGIN
 SELECT MIN(salary), MAX(salary)
 INTO v_minsalary, v_maxsalary
 FROM employees
 WHERE job_id = :NEW.job_id;
  IF :NEW.salary < v_minsalary OR
   :NEW.salary > v_maxsalary THEN
   RAISE_APPLICATION_ERROR(-20505,'Out of range');
  END IF;
END;
```

```
UPDATE employees
  SET salary = 3400
  WHERE last_name = 'Davies';
```



```
ORA-04091: table US_NLH2_PLSQL_T01.COPY_EMPLOYEES is mutating, trigger/function may
not see it
ORA-06512: at "US_NLH2_PLSQL_T01.CHECK_SALARY", line 5
ORA-04088: error during execution of trigger 'US_NLH2_PLSQL_T01.CHECK_SALARY'



3.   WHERE last_name = 'Davies';
```

The Status of a Trigger

A trigger is in either of two distinct modes:

- **Enabled**: The trigger runs its trigger action if a triggering statement is issued and the trigger restriction (if any) evaluates to true (default).
- **Disabled**: The trigger does not run its trigger action, even if a triggering statement is issued and the trigger restriction (if any) would evaluate to true.

- Creating a Disabled Trigger:

```
CREATE OR REPLACE TRIGGER mytrg
  BEFORE INSERT ON mytable FOR EACH ROW
  DISABLE
BEGIN
  :New.ID := my_seq.Nextval;
. . .
END;
```

Changing the Status of Triggers

Why would we disable a trigger?

1. To improve performance when loading very large amounts ofdata into the database.
   For example, imagine a trigger defined as …
   AFTER INSERT ON bigtable FOR EACH ROW….
   Now someone (maybe the DBA) inserts 10 million rows into BIGTABLE. This row trigger will fire 10 milliontimes, slowing down the data load considerably.
2. We may disable a trigger when it references a database object that is currently unavailable due to a failed network connection, disk crash, offline data file, or offline table space.

Managing Triggers

Managing Triggers Using the ALTER and DROP SQL Statements

```
-- Disable or reenable a database trigger:

ALTER TRIGGER trigger_name DISABLE | ENABLE;
```

```
-- Disable or reenable all triggers for a table:

ALTER TABLE table_name DISABLE | ENABLE ALL TRIGGERS;
```

```
-- Recompile a trigger for a table:

ALTER TRIGGER trigger_name COMPILE;
```

```
-- Remove a trigger from the database:

DROP TRIGGER trigger_name;
```

Note: All triggers on a table are removed when the table is removed.

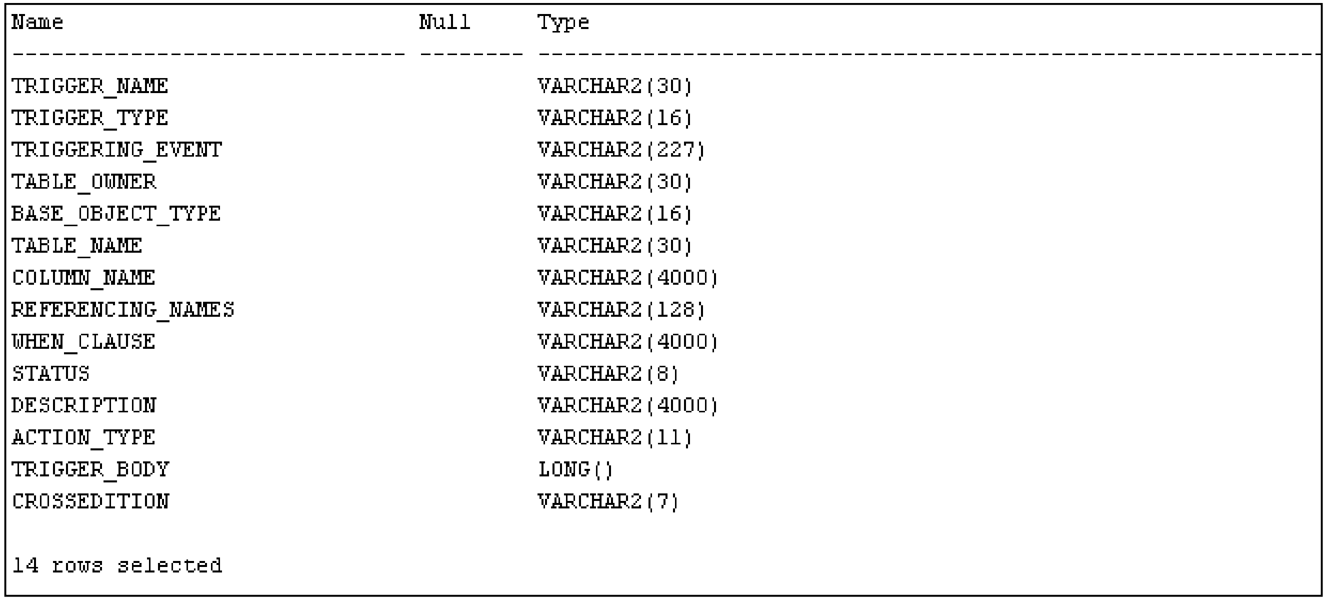
Viewing Trigger Information

You can view the following trigger information:

Data Dictionary View	Description
`USER_OBJECTS`	Displays object information
`USER/ALL/DBA_TRIGGERS`	Displays trigger information
`USER_ERRORS`	Displays PL/SQL syntax errors for a trigger

Using USER_TRIGGERS

```
DESCRIBE user_triggers
```



```
Name                           Null     Type
------------------------------ -------- ------------------------------------------------------------
TRIGGER_NAME                            VARCHAR2(30)
TRIGGER_TYPE                            VARCHAR2(16)
TRIGGERING_EVENT                        VARCHAR2(227)
TABLE_OWNER                             VARCHAR2(30)
BASE_OBJECT_TYPE                        VARCHAR2(16)
TABLE_NAME                              VARCHAR2(30)
COLUMN_NAME                             VARCHAR2(4000)
REFERENCING_NAMES                       VARCHAR2(128)
WHEN_CLAUSE                             VARCHAR2(4000)
STATUS                                  VARCHAR2(8)
DESCRIPTION                             VARCHAR2(4000)
ACTION_TYPE                             VARCHAR2(11)
TRIGGER_BODY                            LONG()
CROSSEDITION                            VARCHAR2(7)

14 rows selected
```

```
SELECT trigger_type, trigger_body
FROM user_triggers
WHERE trigger_name = 'SECURE_EMP';
```

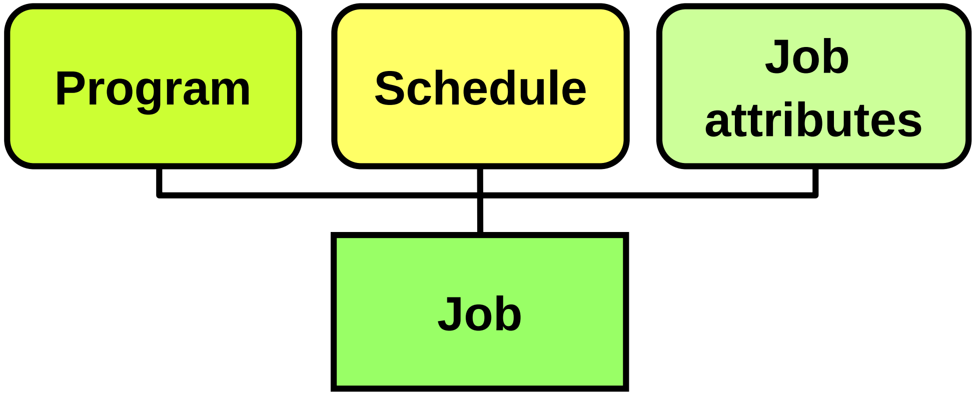
Triggers – final remark

- You should not create a trigger to do something that can easily be done in another way, such as by a check constraint or by suitable object privileges.
- But sometimes you must create a trigger because there is no other way to do what is needed.

Database Programmability

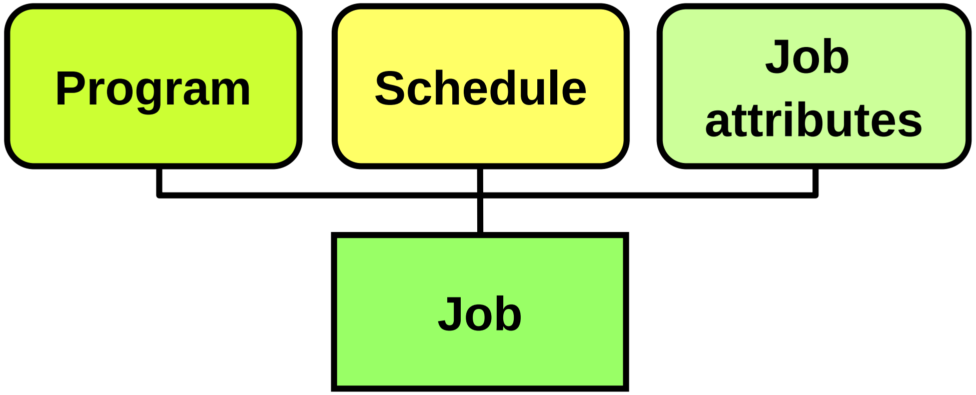
Automating Tasks with the Scheduler

Scheduler



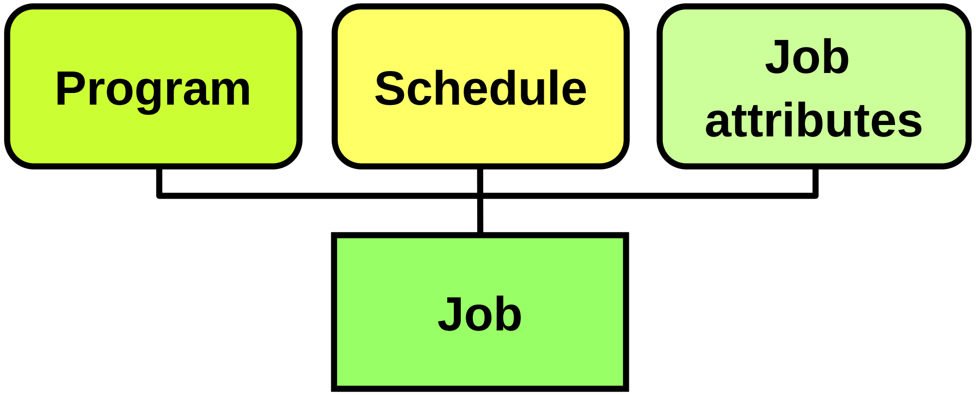
- A job has two mandatory components:
  - “what” action needs to be done, and
  - the time or schedule “when” the action occurs.
- The “what” is expressed in the command region and the job attributes: the job_type and the job_action parameters.
- The “when” is expressed in a schedule, which can be based on time or events, or be dependent on the outcome of other jobs.

Scheduler



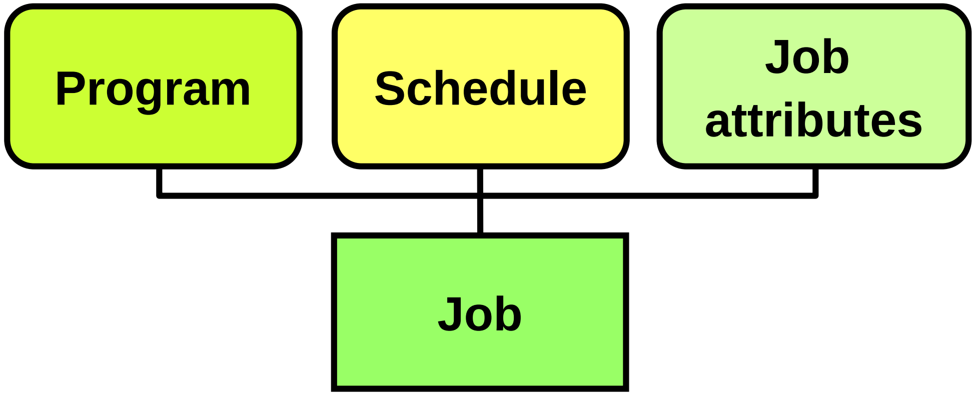
- The Scheduler uses the following basic components:
  - a job,
  - a schedule,
  - a program.

Scheduler



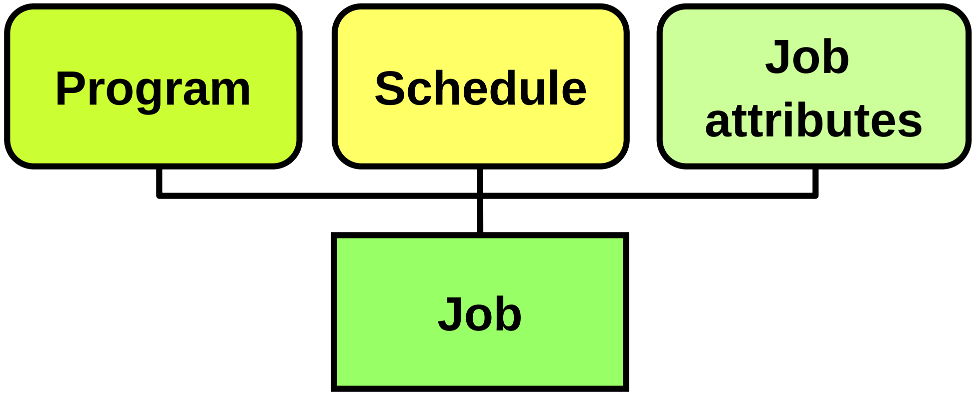
- A **job** specifies what needs to be executed.
- It could be as an example a PL/SQL procedure.
- You can specify the program (what) and schedule (when) as part of the job definition, or you can use an existing program or schedule instead.
- You can use arguments for a job to customize its run-time behavior.

Scheduler



- A **schedule** specifies when and how many times a job is executed.
- A schedule can be based on time or an event, or a combination of the two.
- You can store the schedule for a job separately and then use the same schedule for multiple jobs.

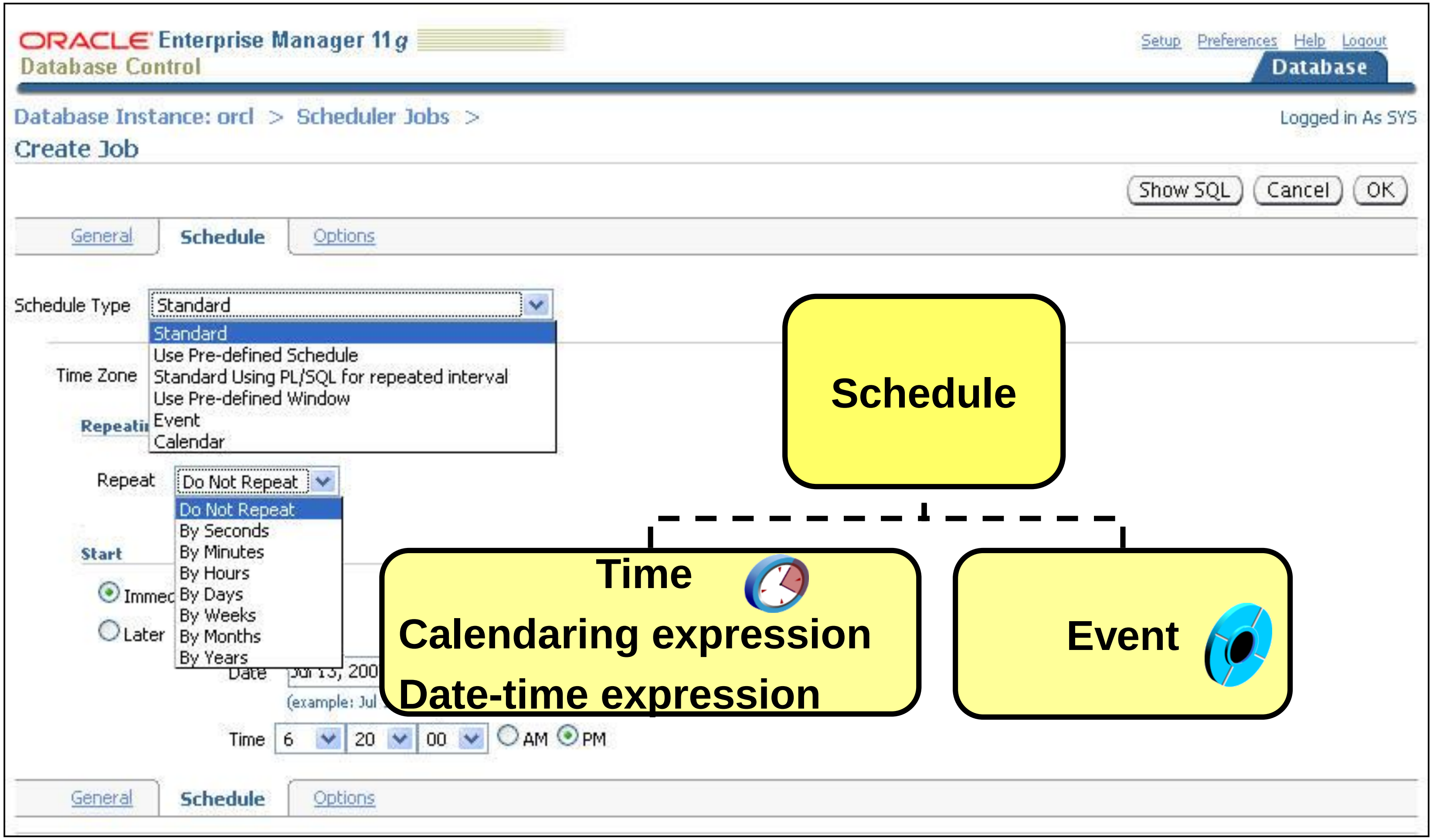
Scheduler



- A **program** is a collection of metadata about a particular executable, script, or procedure.
- An automated job executes some task. Using a program enables you to modify the job task, or the “what,” without modifying the job itself.
- You can define arguments for a program, enabling users to modify the run-time behavior of the task.

Scheduler

Using a Time-Based or Event-Based Schedule



ORACLE Enterprise Manager 11g
Database Control

Database Instance: orcl > Scheduler Jobs >

Logged in As SYS

Create Job

Schedule Type: Standard

Repeat: Do Not Repeat

Start: ● Immed ○ Later

Date: Jul 13, 200

(example: Jul

Time: 6 : 20 : 00 ○ AM ● PM

Scheduler

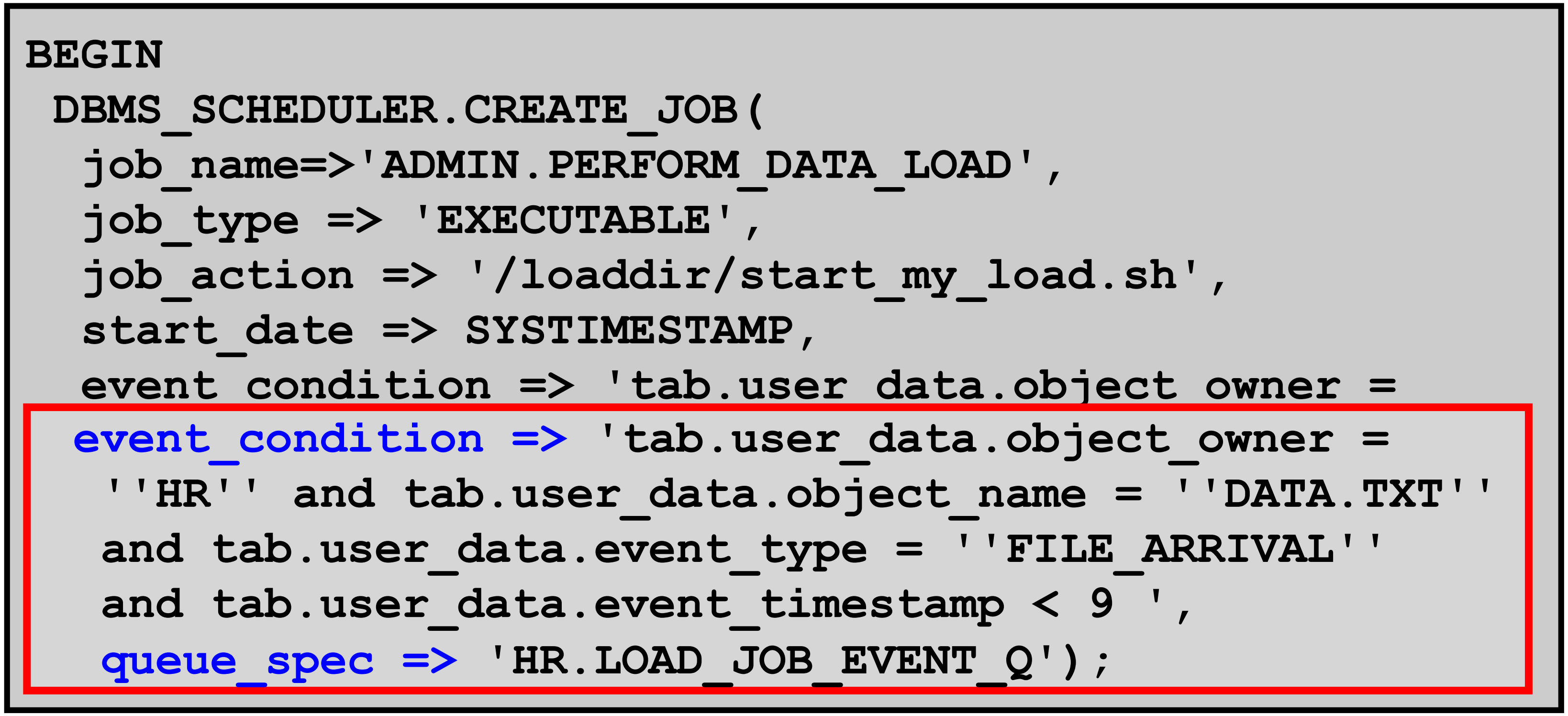
Creating a Time-Based Job

Example: Create a job that calls a backup script every night at 11:00, starting tonight.

```
BEGIN
 DBMS_SCHEDULER.CREATE_JOB(
  job_name=>'HR.DO_BACKUP',
  job_type => 'EXECUTABLE',
  job_action =>
'/home/usr/dba/rman/nightly_incr.sh',
  start_date=> SYSDATE,
  repeat_interval=>'FREQ=DAILY;BYHOUR=23',
                /* next night at 11:00 PM */
  comments => 'Nightly incremental backups');
END;
/
```

Creating an Event-Based Job

Example: Create a job that runs if a batch load data file arrives on the file system before 9:00 AM.



```
BEGIN
 DBMS_SCHEDULER.CREATE_JOB(
  job_name=>'ADMIN.PERFORM_DATA_LOAD',
  job_type => 'EXECUTABLE',
  job_action => '/loaddir/start_my_load.sh',
  start_date => SYSTIMESTAMP,
  event condition => 'tab.user data.object owner =
  event_condition => 'tab.user_data.object_owner =
   ''HR'' and tab.user_data.object_name = ''DATA.TXT''
   and tab.user_data.event_type = ''FILE_ARRIVAL''
   and tab.user_data.event_timestamp < 9 ',
   queue_spec => 'HR.LOAD_JOB_EVENT_Q');
```

Event-Based Scheduling

- Event types:
  - User- or application-generated events
  - Scheduler-generated events
- Events raised by Scheduler jobs:
  - JOB_STARTED
  - JOB_SUCCEEDED
  - JOB_FAILED
  - JOB_BROKEN
  - JOB_COMPLETED
  - JOB_STOPPED
  - JOB_SCH_LIM_REACHED
  - JOB_DISABLED
  - JOB_CHAIN_STALLED
  - JOB_ALL_EVENTS
  - JOB_RUN_COMPLETED
  - JOB_OVER_MAX_DUR
- Example of raising an event:

```
DBMS_SCHEDULER.SET_ATTRIBUTE('hr.do_backup',
 'raise_events', DBMS_SCHEDULER.JOB_FAILED);
```

Database Programmability

Designing & TUNING PL/SQL Code

Guidelines for Cursor Design

- Fetch into a record when fetching from a cursor - this way you do not need to declare individual variables, and you reference only the values that you want to use. Additionally, you can automatically use the structure of the SELECT column list.

```
DECLARE
  CURSOR cur_cust IS
    SELECT customer_id, cust_last_name, cust_email
    FROM customers
    WHERE credit_limit = 1200;
  v_cust_record    cur_cust%ROWTYPE;
BEGIN
  OPEN cur_cust;
  LOOP
    FETCH cur_cust INTO v_cust_record;
...
```

Guidelines for Cursor Design

- Create cursors with parameters – they increase the flexibility and reusability of cursors, because you can pass different values to the WHERE clause when you open a cursor, rather than hard-code a value for the WHERE clause.

```
CREATE OR REPLACE PROCEDURE cust_pack
 (p_crd_limit_in NUMBER, p_acct_mgr_in NUMBER)
IS
  v_credit_limit NUMBER := 1500;
   CURSOR cur_cust (p_crd_limit NUMBER, p_acct_mgr NUMBER)
     IS
     SELECT customer_id, cust_last_name, cust_email
     FROM customers
     WHERE credit_limit = p_crd_limit AND account_mgr_id = p_acct_mgr;
BEGIN
   OPEN cur_cust(p_crd_limit_in, p_acct_mgr_in);
...
   CLOSE cur_cust;
...
   OPEN cur_cust(v_credit_limit, 145);
...
END;
```

Guidelines for Cursor Design

Reference implicit cursor attributes immediately after the SQL statement executes.

```
BEGIN
  UPDATE customers
    SET   credit_limit = p_credit_limit
    WHERE customer_id  = p_cust_id;
  get_avg_order(p_cust_id);  -- procedure call
  IF SQL%NOTFOUND THEN
       ...
```

Guidelines for Cursor Design

Simplify coding with cursor `FOR` loops.

```
CREATE OR REPLACE PROCEDURE cust_pack
 (p_crd_limit_in NUMBER, p_acct_mgr_in NUMBER)
IS
  v_credit_limit NUMBER := 1500;
   CURSOR cur_cust (p_crd_limit NUMBER, p_acct_mgr NUMBER)
     IS
     SELECT customer_id, cust_last_name, cust_email
     FROM customers
     WHERE credit_limit = p_crd_limit
     AND   account_mgr_id = p_acct_mgr;
BEGIN
  FOR cur_rec IN cur_cust (p_crd_limit_in, p_acct_mgr_in)
  LOOP         -- implicit open and fetch
  ...
  END LOOP;    -- implicit close
  ...
END;
```

Tuning PL/SQL Code

You can tune your PL/SQL code by:

- Identifying the data type and constraint issues
  - Data type conversion
  - The `NOT NULL` constraint
- Writing smaller executable sections of code
- Comparing SQL with PL/SQL
- Rephrasing conditional statements

Avoiding Implicit Data Type Conversion

- PL/SQL performs implicit conversions between structurally different data types.
- Example: When assigning a `PLS_INTEGER` variable to a `NUMBER` variable

```
DECLARE
  n NUMBER;
BEGIN
  n := n + 15;    -- converted
  n := n + 15.0;  -- not converted
  ...
END;
```

Avoiding Implicit Data Type Conversion

- The NOT NULL constraint incurs a small performance cost. Therefore, use it with care.

```
PROCEDURE calc_m IS
 m NUMBER NOT NULL:=0;
 a NUMBER;
 b NUMBER;
BEGIN
  ...
 m := a + b;
  ...
END;
```

```
PROCEDURE calc_m IS
m NUMBER; -- no
       -- constraint
a NUMBER;
b NUMBER;
BEGIN
       ...
   m := a + b;
  IF m IS NULL THEN
        -- raise error
  END IF;
END;
```

Avoiding Implicit Data Type Conversion

- The NOT NULL constraint incurs a small performance cost. Therefore, use it with care.

```
PROCEDURE calc_m IS
 m NUMBER NOT NULL:=0;
 a NUMBER;
 b NUMBER;
BEGIN
 ...
 m := a + b;
 ...
END;
```

```
PROCEDURE calc_m IS
m NUMBER; -- no
       -- constraint
a NUMBER;
b NUMBER;
BEGIN
       ...
   m := a + b;
  IF m IS NULL THEN
        -- raise error
  END IF;
END;
```

Modularizing Your Code

- Limit the number of lines of code between a `BEGIN` and `END` to about a page or 60 lines of code.
- Use packaged programs to keep each executable section small.
- Use local procedures and functions to hide logic.
- Use a function interface to hide formulas and business rules.

Comparing SQL with PL/SQL

- Some simple set processing is markedly faster than the equivalent PL/SQL.

```
BEGIN
  INSERT INTO inventories2
    SELECT product_id, warehouse_id
    FROM main_inventories;
END;
```

- Avoid using procedural code when it may be better to use SQL.

```
...FOR I IN 1..5600 LOOP
     counter := counter + 1;
     SELECT product_id, warehouse_id
       INTO v_p_id, v_wh_id
       FROM big_inventories WHERE v_p_id = counter;
     INSERT INTO inventories2 VALUES(v_p_id, v_wh_id);
   END LOOP;...
```

Rephrasing Conditional Control Statements

- If your business logic results in one condition being true, use the `ELSIF` syntax for mutually exclusive clauses:

```
IF v_acct_mgr = 145 THEN
  process_acct_145;
END IF;
IF v_acct_mgr = 147 THEN
  process_acct_147;
END IF;
IF v_acct_mgr = 148 THEN
  process_acct_148;
END IF;
IF v_acct_mgr = 149 THEN
  process_acct_149;
END IF;
```

```
IF v_acct_mgr = 145
THEN
  process_acct_145;
ELSIF v_acct_mgr = 147
THEN
  process_acct_147;
ELSIF v_acct_mgr = 148
THEN
  process_acct_148;
ELSIF v_acct_mgr = 149
THEN
  process_acct_149;
END IF;
```