5 Integrity Constraints and Triggers

5.1 Integrity Constraints

In Section 1 we have discussed three types of integrity constraints: not null constraints, primary keys, and unique constraints. In this section we introduce two more types of constraints that can be specified within the **create table** statement: *check constraints* (to restrict possible attribute values), and *foreign key constraints* (to specify interdependencies between relations).

5.1.1 Check Constraints

Often columns in a table must have values that are within a certain range or that satisfy certain conditions. Check constraints allow users to restrict possible attribute values for a column to admissible ones. They can be specified as column constraints or table constraints. The syntax for a check constraint is

[**constraint** <name>] **check**(<condition>)

If a **check** constraint is specified as a column constraint, the condition can only refer that column.

Example: The name of an employee must consist of upper case letters only; the minimum salary of an employee is 500; department numbers must range between 10 and 100:

```
create table EMP
    ( ...,
      ENAME      varchar2(30) constraint check_name
                              check(ENAME = upper(ENAME) ),
      SAL        number(5,2) constraint check_sal  check(SAL >= 500),
      DEPTNO     number(3) constraint check_deptno
                              check(DEPTNO between 10 and 100) );
```

If a **check** constraint is specified as a table constraint, <condition> can refer to all columns of the table. Note that only simple conditions are allowed. For example, it is not allowed to refer to columns of other tables or to formulate queries as check conditions. Furthermore, the functions **sysdate** and **user** cannot be used in a condition. In principle, thus only simple attribute comparisons and logical connectives such as **and**, **or**, and **not** are allowed. A check condition, however, can include a not null constraint:

SAL **number**(5,2) **constraint** check_sal **check**(SAL **is not null and** SAL >= 500),

Without the **not null** condition, the value *null* for the attribute SAL would not cause a violation of the constraint.

Example: At least two persons must participate in a project, and the project's start date must be before the project's end date:


```
create table PROJECT
    ( ...,
      PERSONS      number(5) constraint check_pers check (PERSONS > 2),
      ...,
      constraint dates_ok check(PEND > PSTART) );
```

In this table definition, check_pers is a column constraint and dates_ok is a table constraint.

The database system automatically checks the specified conditions each time a database modification is performed on this relation. For example, the insertion

insert into EMP **values**(7999,'SCOTT','CLERK',7698,'31-OCT-94',450,10);

causes a constraint violation

ORA-02290: check_constraint (CHECK_SAL) violated

and the insertion is rejected.

5.1.2 Foreign Key Constraints

A foreign key constraint (or referential integrity constraint) can be specified as a column constraint or as a table constraint:

[**constraint** <name>] [**foreign key** (<column(s)>)]
references <table>[(<column(s)>)]
[**on delete cascade**]

This constraint specifies a column or a list of columns as a foreign key of the referencing table. The referencing table is called the *child-table*, and the referenced table is called the *parent-table*. In other words, one cannot define a referential integrity constraint that refers to a table R before that table R has been created.

The clause **foreign key** has to be used in addition to the clause **references** if the foreign key includes more than one column. In this case, the constraint has to be specified as a table constraint. The clause **references** defines which columns of the parent-table are referenced. If only the name of the parent-table is given, the list of attributes that build the primary key of that table is assumed.

Example: Each employee in the table EMP must work in a department that is contained in the table DEPT:

```
create table EMP
    ( EMPNO      number(4) constraint pk_emp primary key,
      ...,
      DEPTNO     number(3) constraint fk_deptno references DEPT(DEPTNO) );
```

The column DEPTNO of the table EMP (child-table) builds the foreign key and references the primary key of the table DEPT (parent-table). The relationship between these two tables is illustrated in Figure 2. Since in this table definition the referential integrity constraint includes

only one column, the clause **foreign key** is not used. It is very important that a foreign key must refer to the complete primary key of a parent-table, not only a subset of the attributes that build the primary key !

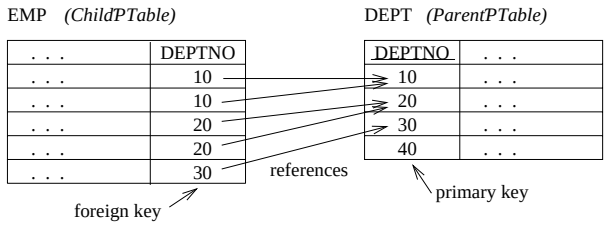

Figure 2: Foreign Key Constraint between the Tables `EMP` and `DEPT`

In order to satisfy a foreign key constraint, each row in the child-table has to satisfy one of the following two conditions:

- the attribute value (list of attribute values) of the foreign key must appear as a primary key value in the parent-table, or
- the attribute value of the foreign key is *null* (in case of a composite foreign key, at least one attribute value of the foreign key is *null*)

According to the above definition for the table `EMP`, an employee must not necessarily work in a department, i.e., for the attribute `DEPTNO` the value *null* is admissible.

Example: Each project manager must be an employee:

```
create table PROJECT
    ( PNO      number(3) constraint prj_pk primary key,
      PMGR     number(4) not null
                         constraint fk_pmgr references  EMP,
      ...);
```

Because only the name of the parent-table is given (`DEPT`), the primary key of this relation is assumed. A foreign key constraint may also refer to the same table, i.e., parent-table and child-table are identical.

Example: Each manager must be an employee:

```
create table EMP
    ( EMPNO    number(4) constraint emp_pk primary key,
      ...
      MGR      number(4) not null
                         constraint fk_mgr references EMP,
      ...
    );
```


5.1.3 More about Column- and Table Constraints

If a constraint is defined within the **create table** command or added using the **alter table** command (compare Section 1.5.5), the constraint is automatically enabled. A constraint can be disabled using the command

alter table <table> **disable**
constraint <name> | **primary key** | **unique**[<column(s)>]
[**cascade**];

To disable a primary key, one must disable all foreign key constraints that depend on this primary key. The clause **cascade** automatically disables foreign key constraints that depend on the (disabled) primary key.

Example: Disable the primary key of the table `DEPT` and disable the foreign key constraint in the table `EMP`:

alter table `DEPT` **disable primary key cascade**;

In order to enable an integrity constraint, the clause **enable** is used instead of **disable**. A constraint can only be enabled successfully if no tuple in the table violates the constraint. Otherwise an error message is displayed. Note that for enabling/disabling an integrity constraint it is important that you have named the constraints.

In order to identify those tuples that violate an integrity constraint whose activation failed, one can use the clause **exceptions into** `EXCEPTIONS` with the **alter table** statement. `EXCEPTIONS` is a table that stores information about violating tuples.[3] Each tuple in this table is identified by the attribute `ROWID`. Every tuple in a database has a pseudo-column `ROWID` that is used to identify tuples. Besides the rowid, the name of the table, the table owner as well as the name of the violated constraint are stored.

Example: Assume we want to add an integrity constraint to our table `EMP` which requires that each manager must earn more than 4000:

```
alter table EMP add constraint manager_sal
          check(JOB != 'MANAGER' or SAL >= 4000)
exceptions into EXCEPTIONS;
```

If the table `EMP` already contains tuples that violate the constraint, the constraint cannot be activated and information about violating tuples is automatically inserted into the table `EXCEPTIONS`.

Detailed information about the violating tuples can be obtained by joining the tables `EMP` and `EXCEPTIONS`, based on the join attribute `ROWID`:

```
select EMP.*, CONSTRAINT from EMP, EXCEPTIONS
where EMP.ROWID = EXCEPTIONS.ROW_ID;
```

[3] Before this table can be used, it must be created using the SQL script `utlexcept.sql` which can be found in the directory `$ORACLE_HOME/rdbms/admin`.

Tuples contained in the query result now can be modified (e.g., by increasing the salary of managers) such that adding the constraint can be performed successfully. Note that it is important to delete "old" violations from the relation `EXCEPTIONS` before it is used again.

If a table is used as a reference of a foreign key, this table can only be dropped using the command **drop table** <table> **cascade constraints**;. All other database objects that refer to this table (e.g., triggers, see Section 5.2) remain in the database system, but they are not valid.

Information about integrity constraints, their status (enabled, disabled) etc. is stored in the data dictionary, more precisely, in the tables `USER_CONSTRAINTS` and `USER_CONS_CONSTRAINTS`.

5.2 Triggers

5.2.1 Overview

The different types of integrity constraints discussed so far provide a *declarative* mechanism to associate "simple" conditions with a table such as a primary key, foreign keys or domain constraints. Complex integrity constraints that refer to several tables and attributes (as they are known as assertions in the SQL standard) cannot be specified within table definitions. *Triggers*, in contrast, provide a procedural technique to specify and maintain integrity constraints. Triggers even allow users to specify more complex integrity constraints since a trigger essentially is a PL/SQL procedure. Such a procedure is associated with a table and is automatically called by the database system whenever a certain modification (*event*) occurs on that table. Modifications on a table may include **insert**, **update**, and **delete** operations (Oracle 7).

5.2.2 Structure of Triggers

A trigger definition consists of the following (optional) components:

- *trigger name*
  create [**or replace**] **trigger** <trigger name>
- *trigger time point*
  before | **after**
- *triggering event(s)*
  insert or update [**of** <column(s)>] **or delete on** <table>
- *trigger type* (optional)
  for each row
- *trigger restriction* (only for **for each row** triggers !)
  when (<condition>)
- *trigger body*
  <PL/SQL block>

The clause **replace** re-creates a previous trigger definition having the same <trigger name>. The name of a trigger can be chosen arbitrarily, but it is a good programming style to use

a trigger name that reflects the table and the event(s), e.g., `upd_ins_EMP`. A trigger can be invoked **before** or **after** the triggering event. The *triggering event* specifies before (after) which operations on the table <table> the trigger is executed. A single event is an **insert**, an **update**, or a **delete**; events can be combined using the logical connective **or**. If for an **update** trigger no columns are specified, the trigger is executed after (before) <table> is updated. If the trigger should only be executed when certain columns are updated, these columns must be specified after the event **update**. If a trigger is used to maintain an integrity constraint, the triggering events typically correspond to the operations that can violate the integrity constraint.

In order to program triggers efficiently (and correctly) it is essential to understand the difference between a *row level trigger* and a *statement level trigger*. A row level trigger is defined using the clause **for each row**. If this clause is not given, the trigger is assumed to be a statement trigger. A row trigger executes once for each row after (before) the event. In contrast, a statement trigger is executed once after (before) the event, independent of how many rows are affected by the event. For example, a row trigger with the event specification **after update** is executed once for each row affected by the update. Thus, if the update affects 20 tuples, the trigger is executed 20 times, for each row at a time. In contrast, a statement trigger is only executed once.

When combining the different types of triggers, there are twelve possible trigger configurations that can be defined for a table:

event	trigger time point		trigger type	
	before	**after**	statement	row
insert	X	X	X	X
update	X	X	X	X
delete	X	X	X	X

Figure 3: Trigger Types

Row triggers have some special features that are not provided by statement triggers:

Only with a row trigger it is possible to access the attribute values of a tuple before and after the modification (because the trigger is executed once for each tuple). For an **update** trigger, the old attribute value can be accessed using **:old**.<column> and the new attribute value can be accessed using **:new**.<column>. For an **insert** trigger, only **:new**.<column> can be used, and for a **delete** trigger only **:old**.<column> can be used (because there exists no old, respectively, new value of the tuple). In these cases, **:new**.<column> refers to the attribute value of <column> of the inserted tuple, and **:old**.<column> refers to the attribute value of <column> of the deleted tuple. In a row trigger thus it is possible to specify comparisons between old and new attribute values in the PL/SQL block, e.g., "**if :old**.`SAL` < **:new**.`SAL` **then** ...". If for a row trigger the trigger time point **before** is specified, it is even possible to modify the new values of the row, e.g., **:new**.`SAL` := **:new**.`SAL` $*$ 1.05 or **:new**.`SAL` := **:old**.`SAL`. Such modifications are not possible with **after** row triggers. In general, it is advisable to use a **after** row trigger if the new row is not modified in the PL/SQL block. Oracle then can process

these triggers more efficiently. Statement level triggers are in general only used in combination with the trigger time point **after**.

In a trigger definition the **when** clause can only be used in combination with a **for each row** trigger. The clause is used to further restrict when the trigger is executed. For the specification of the condition in the **when** clause, the same restrictions as for the **check** clause hold. The only exceptions are that the functions **sysdate** and **user** can be used, and that it is possible to refer to the old/new attribute values of the actual row. In the latter case, the colon ":" must not be used, i.e., only **old**.<attribute> and **new**.<attribute>.

The trigger body consists of a PL/SQL block. All SQL and PL/SQL commands except the two statements **commit** and **rollback** can be used in a trigger's PL/SQL block. Furthermore, additional **if** constructs allow to execute certain parts of the PL/SQL block depending on the triggering event. For this, the three constructs **if inserting**, **if updating**[('<column>')], and **if deleting** exist. They can be used as shown in the following example:

```
create or replace trigger emp_check
after insert or delete or update on EMP
for each row
begin
   if inserting then
      <PL/SQL block>
   end if;
   if updating then
      <PL/SQL block>
   end if;
   if deleting then
      <PL/SQL block>
   end if;
end;
```

It is important to understand that the execution of a trigger's PL/SQL block builds a part of the transaction that contains the triggering event. Thus, for example, an **insert** statement in a PL/SQL block can cause another trigger to be executed. Multiple triggers and modifications thus can lead to a cascading execution of triggers. Such a sequence of triggers terminates successfully if (1) no exception is raised within a PL/SQL block, and (2) no declaratively specified integrity constraint is violated. If a trigger raises an exception in a PL/SQL block, all modifications up to the beginning of the transaction are rolled back. In the PL/SQL block of a trigger, an exception can be raised using the statement **raise_application_error** (see Section 4.1.5). This statement causes an implicit **rollback**. In combination with a row trigger, **raise_application_error** can refer to old/new values of modified rows:

```
raise_application_error(−20020, 'Salary increase from ' || to_char(:old.SAL) || ' to '
                                  to_char(:new.SAL) || ' is too high');  or

raise_application_error(−20030, 'Employee Id ' ||
                               to_char(:new .EMPNO) || ' does not exist.');
```


5.2.3 Example Triggers

Suppose we have to maintain the following integrity constraint: "The salary of an employee different from the president cannot be decreased and must also not be increased more than 10%. Furthermore, depending on the job title, each salary must lie within a certain salary range.

We assume a table SALGRADE that stores the minimum (MINSAL) and maximum (MAXSAL) salary for each job title (JOB). Since the above condition can be checked for each employee individually, we define the following row trigger:

trig1.sql

```
create or replace trigger check_salary_EMP
after insert or update of SAL, JOB on EMP
for each row
when (new.JOB != 'PRESIDENT') − − trigger restriction
declare
  minsal, maxsal SALGRADE.MAXSAL%TYPE;
begin
  − − retrieve minimum and maximum salary for JOB
  select MINSAL, MAXSAL into minsal, maxsal from SALGRADE
  where JOB = :new.JOB;
  − − If the new salary has been decreased or does not lie within the salary range,
  − − raise an exception
  if (:new.SAL < minsal or :new.SAL > maxsal) then
    raise_application_error(-20225, 'Salary range exceeded');
  elsif (:new.SAL < :old.SAL) then
    raise_application_error(-20230, 'Salary has been decreased');
  elsif (:new.SAL > 1.1 * :old.SAL) then
    raise_application_error(-20235, 'More than 10% salary increase');
  end if;
end;
```

We use an **after** trigger because the inserted or updated row is not changed within the PL/SQL block (e.g., in case of a constraint violation, it would be possible to restore the old attribute values).

Note that also modifications on the table SALGRADE can cause a constraint violation. In order to maintain the complete condition we define the following trigger on the table SALGRADE. In case of a violation by an **update** modification, however, we do not raise an exception, but restore the old attribute values.

trig2.sql

```
create or replace trigger check_salary_SALGRADE
before update or delete on SALGRADE
for each row
when (new.MINSAL > old.MINSAL
      or new.MAXSAL < old.MAXSAL)
      – – only restricting a salary range can cause a constraint violation
declare
  job_emps number(3) := 0;
begin
  if deleting then – – Does there still exist an employee having the deleted job ?
    select count(*) into job_emps from EMP
    where JOB = :old.JOB;
    if job_emps != 0 then
      raise_application_error(-20240, ' There still exist employees with the job ' ||
                                        :old.JOB);
    end if ;
  end if ;
  if updating then
    – – Are there employees whose salary does not lie within the modified salary range ?
    select count(*) into job_emps from EMP
    where JOB = :new.JOB
      and SAL not between :new.MINSAL and :new.MAXSAL;
    if job_emps != 0 then – – restore old salary ranges
      :new.MINSAL := :old.MINSAL;
      :new.MAXSAL := :old.MAXSAL;
    end if ;
  end if ;
end;
```

In this case a **before** trigger must be used to restore the old attribute values of an updated row.

Suppose we furthermore have a column BUDGET in our table DEPT that is used to store the budget available for each department. Assume the integrity constraint requires that the total of all salaries in a department must not exceed the department's budget. Critical operations on the relation EMP are insertions into EMP and updates on the attributes SAL or DEPTNO.

trig3.sql

```
create or replace trigger check_budget_EMP
after insert or update of SAL, DEPTNO on EMP
declare
  cursor DEPT_CUR is
    select DEPTNO, BUDGET from DEPT;
  DNO       DEPT.DEPTNO%TYPE;
  ALLSAL    DEPT.BUDGET%TYPE;
  DEPT_SAL number;
begin
  open DEPT_CUR;
  loop
    fetch DEPT_CUR into DNO, ALLSAL;
    exit when DEPT_CUR%NOTFOUND;
    select sum(SAL) into DEPT_SAL from EMP
    where DEPTNO = DNO;
    if DEPT_SAL > ALLSAL then
      raise_application_error(-20325, 'Total of salaries in the department ' ||
                        to_char(DNO) || ' exceeds budget');
    end if;
  end loop;
  close DEPT_CUR;
end;
```

In this case we use a statement trigger on the relation EMP because we have to apply an aggregate function on the salary of all employees that work in a particular department. For the relation DEPT, we also have to define a trigger which, however, can be formulated as a row trigger.

5.2.4 Programming Triggers

For programmers, row triggers are the most critical type of triggers because they include several restrictions. In order to ensure read consistency, ORACLE performs an exclusive lock on the table at the beginning of an **insert**, **update**, or **delete** statement. That is, other users cannot access this table until modifications have been successfully completed. In this case, the table currently modified is said to be a *mutating* table. The only way to access a mutating table in a trigger is to use **:old**.<column> and **:new**.<column> in connection with a row trigger.

Example of an erroneous row trigger:

```
create trigger check_sal_EMP
after update of SAL on EMP
for each row
```

```
declare
  sal_sum number;
begin
  select sum(SAL) into sal_sum from EMP;
  ...;
end;
```

For example, if an **update** statement of the form **update** EMP **set** SAL = SAL ∗ 1.1 is executed on the table EMP, the above trigger is executed once for each modified row. While the table is being modified by the update command, it is not possible to access all tuples of the table using the **select** command, because it is locked. In this case we get the error message

```
ORA-04091: table EMP is mutating, trigger may not read or modify it
ORA-06512: at line 4
ORA-04088: error during execution of trigger 'CHECK_SAL_EMP'
```

The only way to access the table, or more precisely, to access the modified tuple, is to use **:old**.<column> and **:new**.<column>.

It is recommended to follow the rules below for the definition of integrity maintaining triggers:

identify operations and tables that are critical for the integrity constraint
  for each such table check
    if constraint can be checked at row level **then**
      if checked rows are modified in trigger **then**
        use **before** row trigger
      else use **after** row trigger
    else
    use **after** statement trigger

Triggers are not exclusively used for integrity maintenance. They can also be used for

- Monitoring purposes, such as the monitoring of user accesses and modifications on certain sensitive tables.
- Logging actions, e.g., on tables:

```
create trigger LOG_EMP
after insert or update or delete on EMP
begin
  if inserting then
    insert into EMP_LOG values(user, 'INSERT', sysdate);
  end if;
  if updating then
    insert into EMP_LOG values(user, 'UPDATE', sysdate);
  end if;
  if deleting then
    insert into EMP_LOG values(user, 'DELETE', sysdate);
  end if;
end;
```

  By using a row trigger, even the attribute values of the modified tuples can be stored in the table EMP_LOG.

- automatic propagation of modifications. For example, if a manager is transfered to another department, a trigger can be defined that automatically transfers the manager's employees to the new department.

5.2.5 More about Triggers

If a trigger is specified within the SQL*Plus shell, the definition must end with a point "." in the last line. Issuing the command **run** causes SQL*Plus to compile this trigger definition. A trigger definition can be loaded from a file using the command @. Note that the last line in the file must consist of a slash "/".

A trigger definition cannot be changed, it can only be re-created using the **or replace** clause. The command **drop** <trigger name> deletes a trigger.

After a trigger definition has been successfully compiled, the trigger automatically is enabled. The command **alter trigger** <trigger name> **disable** is used to deactivate a trigger. All triggers defined on a table can be (de)activated using the command

alter table <Tabelle> **enable** | **disable all trigger**;

The data dictionary stores information about triggers in the table USER_TRIGGERS. The information includes the trigger name, type, table, and the code for the PL/SQL block.