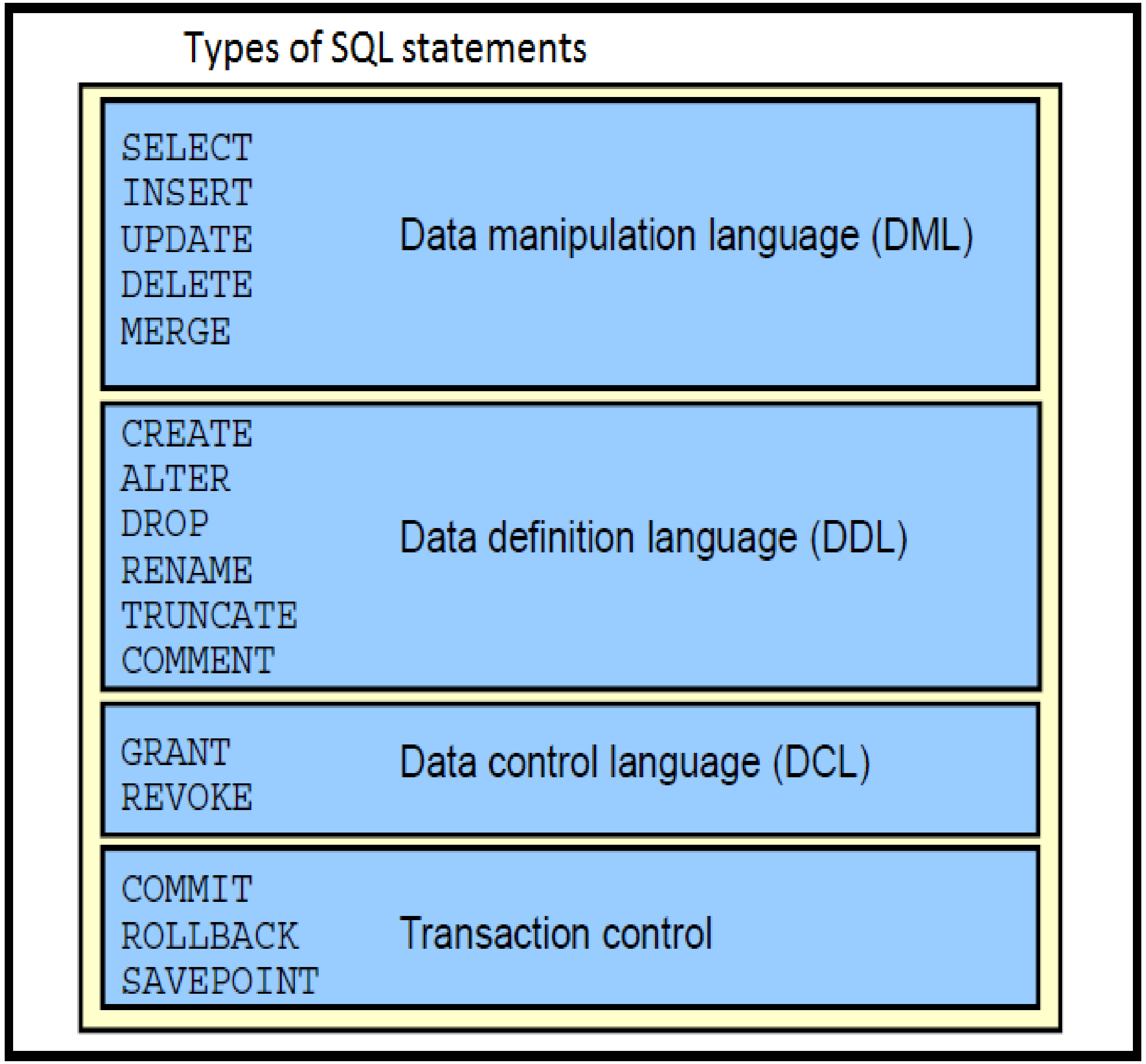
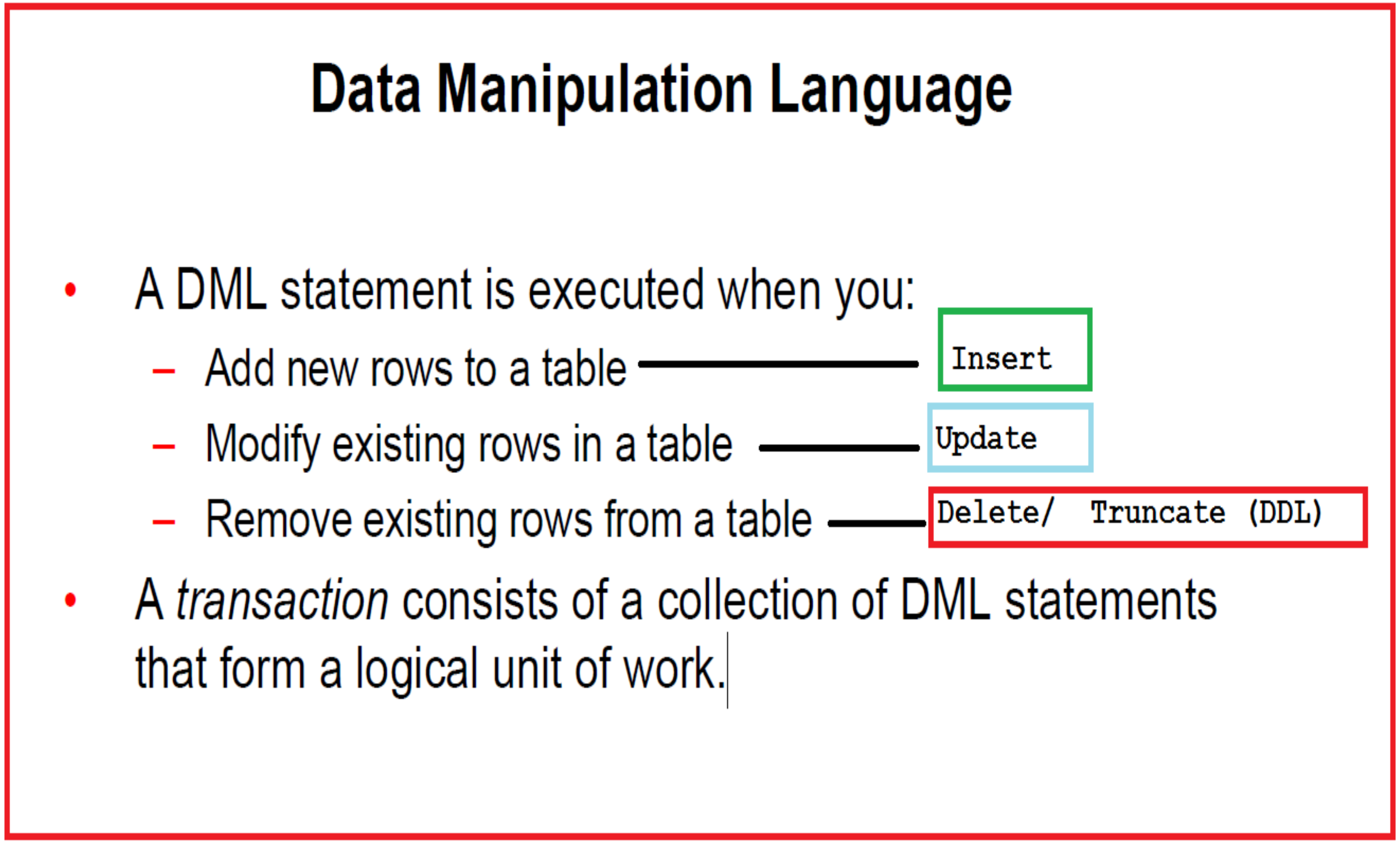
Managing Tables Using DML Statements



Types of SQL statements

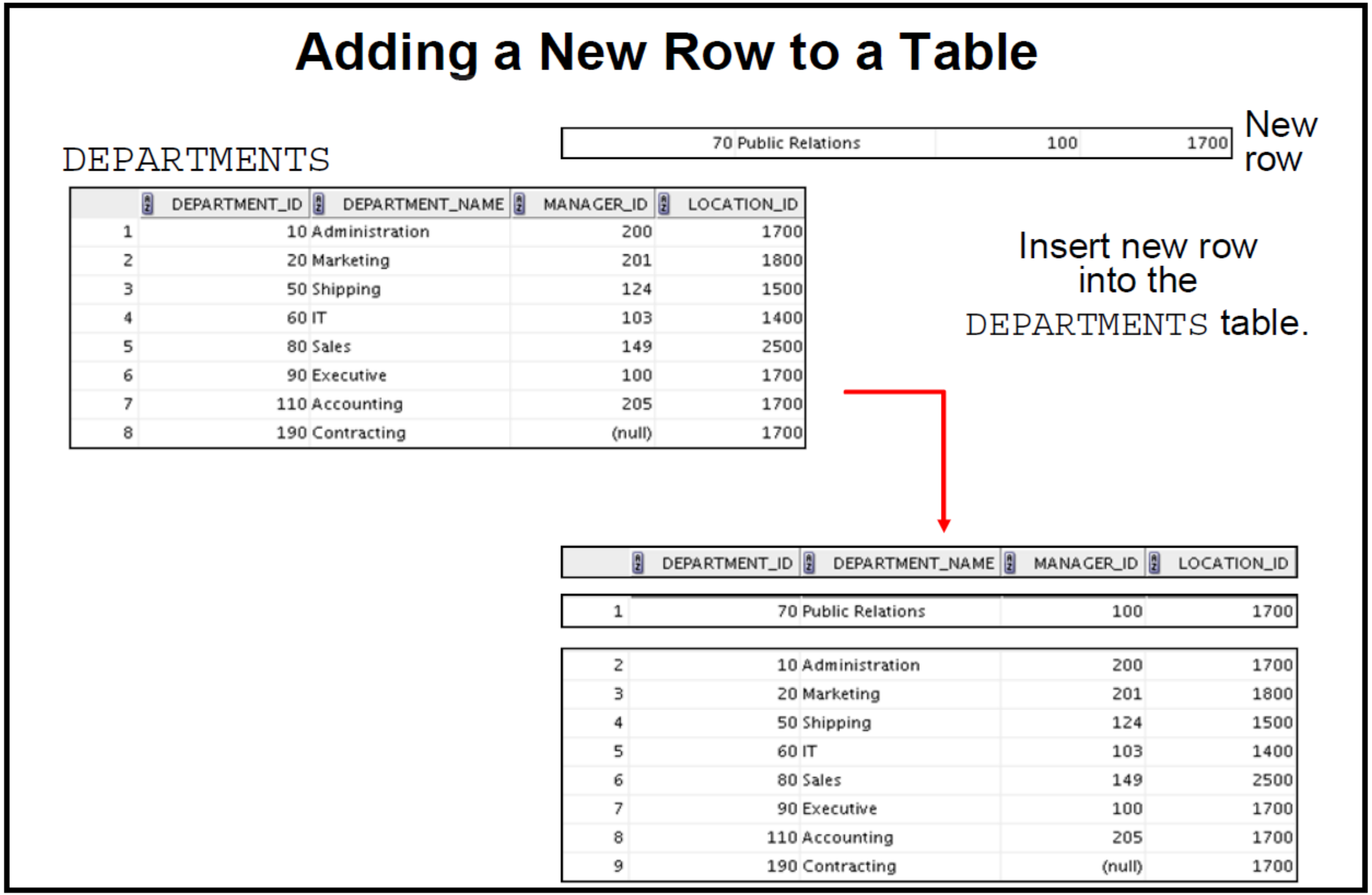
SELECT
INSERT
UPDATE
DELETE
MERGE
	Data manipulation language (DML)

CREATE ALTER DROP RENAME TRUNCATE COMMENT	Data definition language (DDL)
GRANT REVOKE	Data control language (DCL)
COMMIT ROLLBACK SAVEPOINT	Transaction control



Data Manipulation Language

- A DML statement is executed when you:
  - Add new rows to a table — Insert
  - Modify existing rows in a table — Update
  - Remove existing rows from a table — Delete/ Truncate (DDL)
- A *transaction* consists of a collection of DML statements that form a logical unit of work.



Adding a New Row to a Table

New row

70 Public Relations	100	1700

DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

Insert new row into the DEPARTMENTS table.

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	70	Public Relations	100	1700
2	10	Administration	200	1700
3	20	Marketing	201	1800
4	50	Shipping	124	1500
5	60	IT	103	1400
6	80	Sales	149	2500
7	90	Executive	100	1700
8	110	Accounting	205	1700
9	190	Contracting	(null)	1700

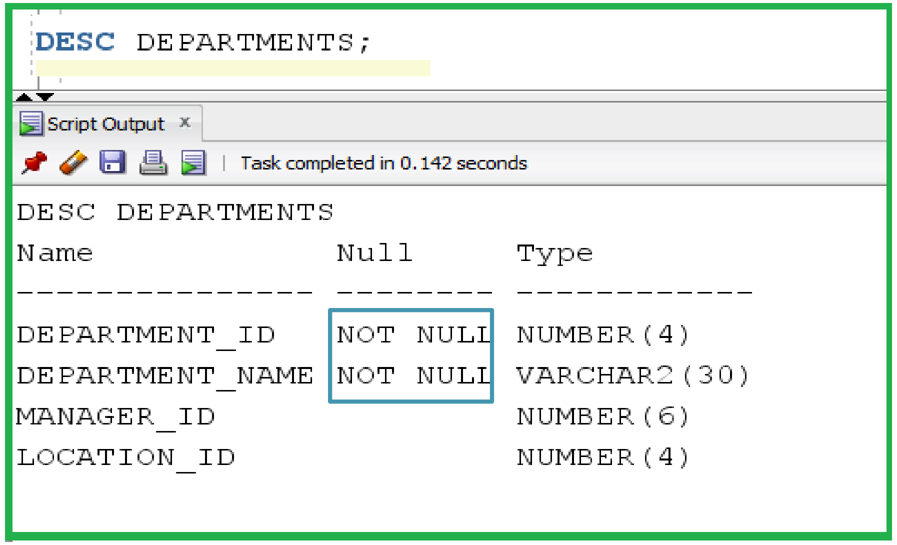
INSERT Statement Syntax

- Add new rows to a table by using the INSERT statement:

```
INSERT INTO   table [(column [, column...])]
VALUES        (value [, value...]);
```

- With this syntax, only one row is inserted at a time.

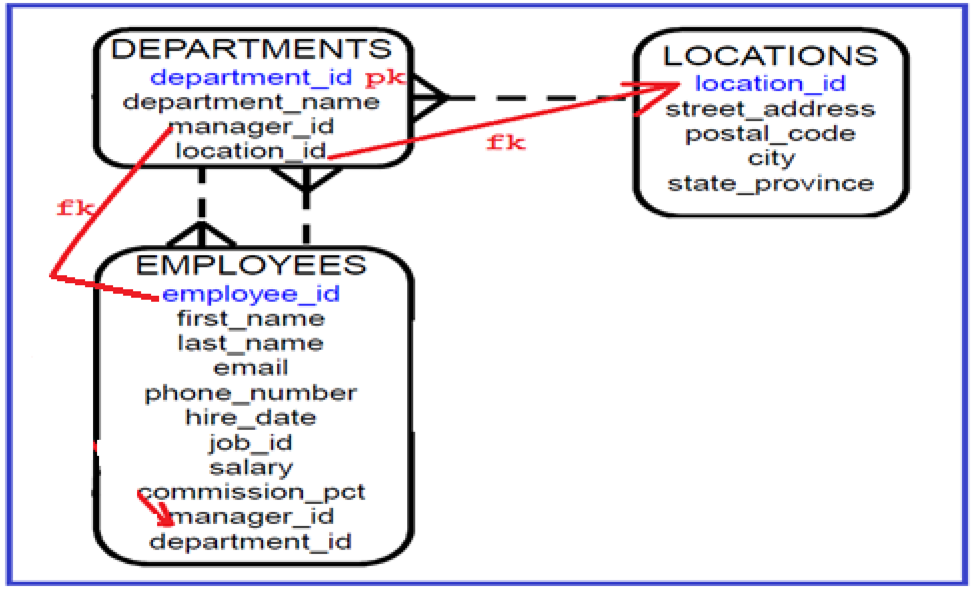
1- You should know the table structure before you make any insert



```
DESC DEPARTMENTS;
```

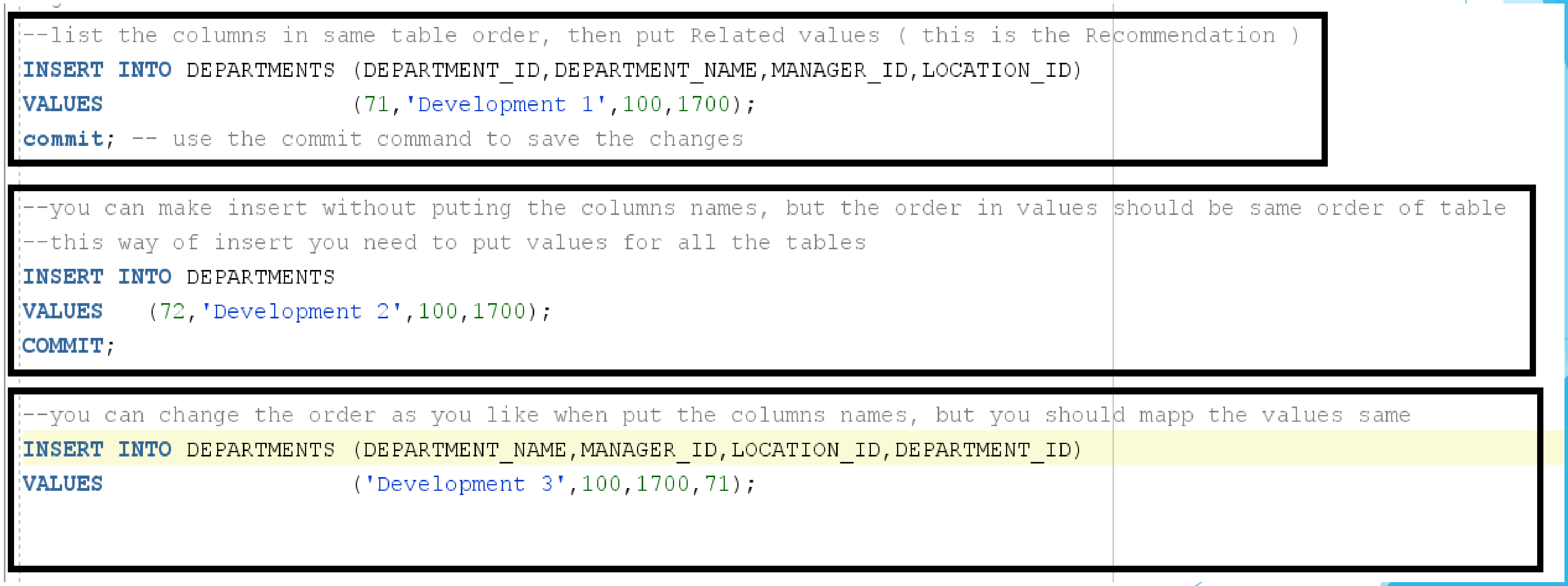
```
DESC DEPARTMENTS
Name            Null     Type
--------------- -------- ------------
DEPARTMENT_ID   NOT NULL NUMBER(4)
DEPARTMENT_NAME NOT NULL VARCHAR2(30)
MANAGER_ID               NUMBER(6)
LOCATION_ID              NUMBER(4)
```

2- you should know also the constraints on the table



Insert rules

- Insert a new row containing values for each column.
- List values in the default order of the columns in the table.
- Optionally, list the columns in the INSERT clause.
- Enclose character and date values within single quotation marks.



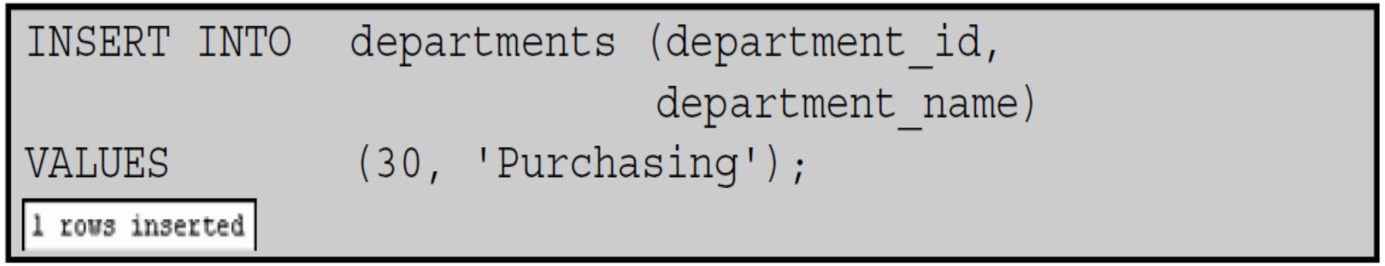
```
--list the columns in same table order, then put Related values ( this is the Recommendation )
INSERT INTO DEPARTMENTS (DEPARTMENT_ID,DEPARTMENT_NAME,MANAGER_ID,LOCATION_ID)
VALUES                  (71,'Development 1',100,1700);
commit; -- use the commit command to save the changes
```

```
--you can make insert without puting the columns names, but the order in values should be same order of table
--this way of insert you need to put values for all the tables
INSERT INTO DEPARTMENTS
VALUES    (72,'Development 2',100,1700);
COMMIT;
```

```
--you can change the order as you like when put the columns names, but you should mapp the values same
INSERT INTO DEPARTMENTS (DEPARTMENT_NAME,MANAGER_ID,LOCATION_ID,DEPARTMENT_ID)
VALUES                  ('Development 3',100,1700,71);
```

Inserting Rows with Null Values

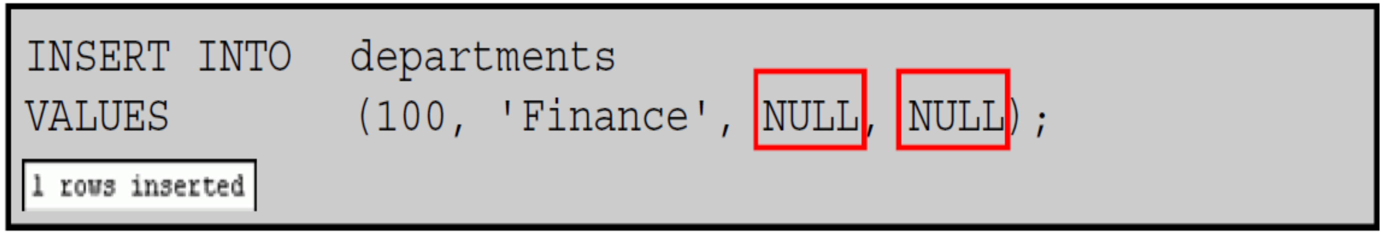
- Implicit method: Omit the column from the column list.



```
INSERT INTO   departments (department_id,
                           department_name)
VALUES        (30, 'Purchasing');
```

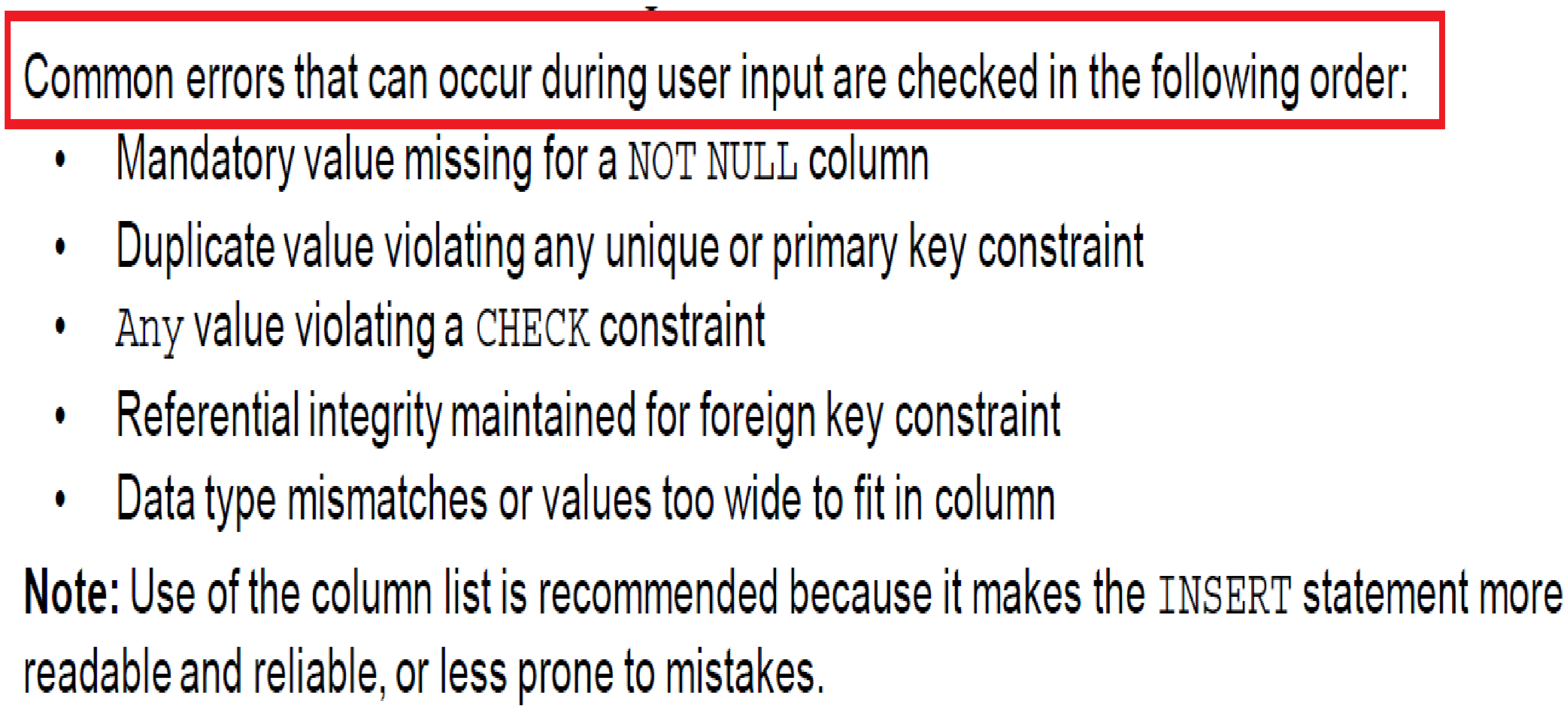
```
1 rows inserted
```

- Explicit method: Specify the `NULL` keyword in the `VALUES` clause.



```
INSERT INTO   departments
VALUES        (100, 'Finance', NULL, NULL);
```

```
1 rows inserted
```

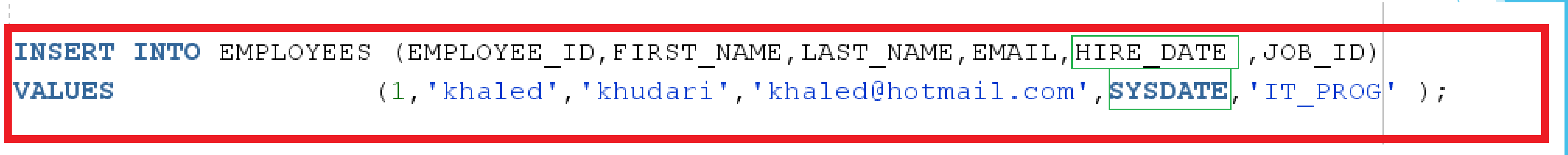


Common errors that can occur during user input are checked in the following order:

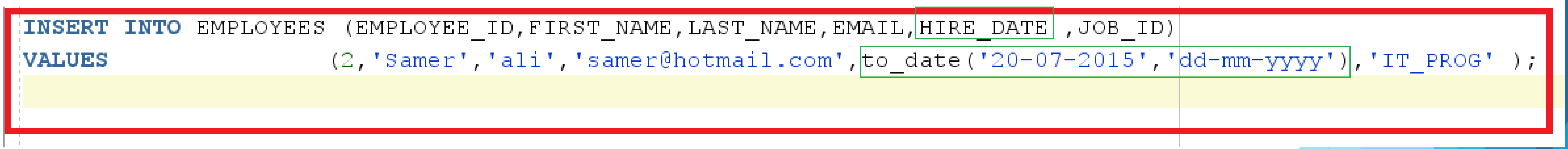
- Mandatory value missing for a NOT NULL column
- Duplicate value violating any unique or primary key constraint
- Any value violating a CHECK constraint
- Referential integrity maintained for foreign key constraint
- Data type mismatches or values too wide to fit in column

Note: Use of the column list is recommended because it makes the INSERT statement more readable and reliable, or less prone to mistakes.

Inserting special values like sysdate, or some other functions



```
INSERT INTO EMPLOYEES (EMPLOYEE_ID,FIRST_NAME,LAST_NAME,EMAIL,HIRE_DATE ,JOB_ID)
VALUES                (1,'khaled','khudari','khaled@hotmail.com',SYSDATE,'IT_PROG' );
```

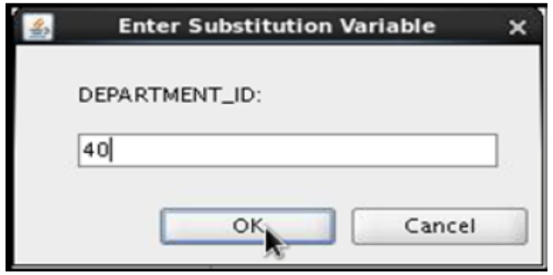


```
INSERT INTO EMPLOYEES (EMPLOYEE_ID,FIRST_NAME,LAST_NAME,EMAIL,HIRE_DATE ,JOB_ID)
VALUES                (2,'Samer','ali','samer@hotmail.com',to_date('20-07-2015','dd-mm-yyyy'),'IT_PROG' );
```

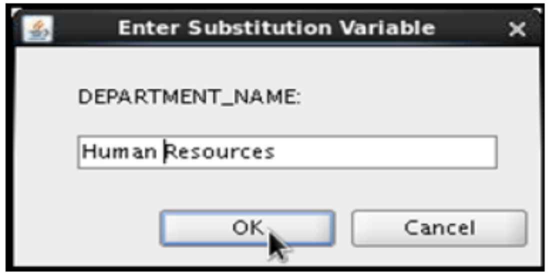
Creating a Script

- Use the & substitution in a SQL statement to prompt for values.
- & is a placeholder for the variable value.

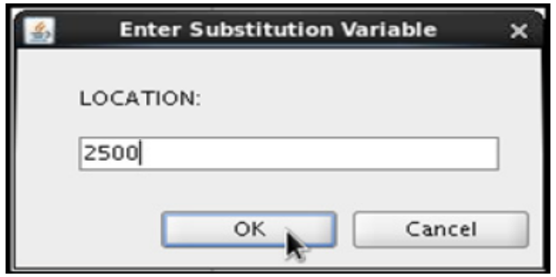
```
INSERT INTO departments
            (department_id, department_name, location_id)
VALUES      (&department_id, '&department_name',&location);
```



DEPARTMENT_ID: 40



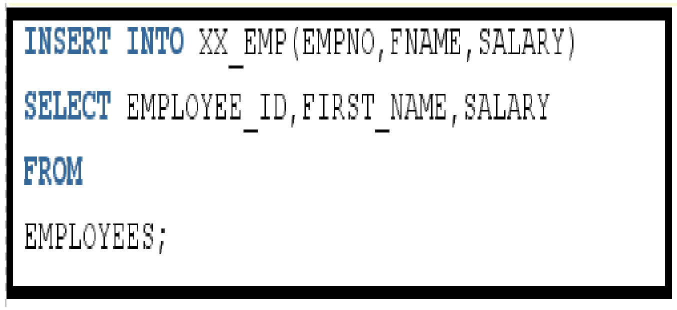
DEPARTMENT_NAME: Human Resources



LOCATION: 2500

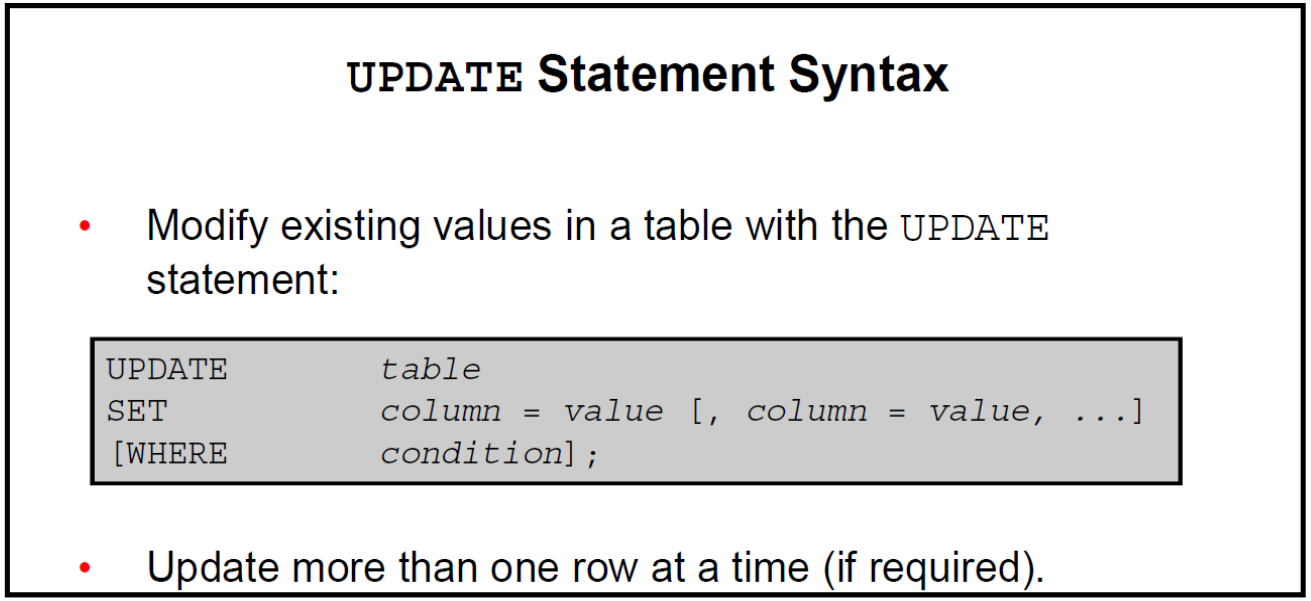
Copying Rows from Another Table

- Write your INSERT statement with a subquery:



```
INSERT INTO XX_EMP(EMPNO,FNAME,SALARY)
SELECT EMPLOYEE_ID,FIRST_NAME,SALARY
FROM
EMPLOYEES;
```

- Do not use the VALUES clause.
- Match the number of columns in the INSERT clause to those in the subquery.
- Inserts all the rows returned by the subquery in the table

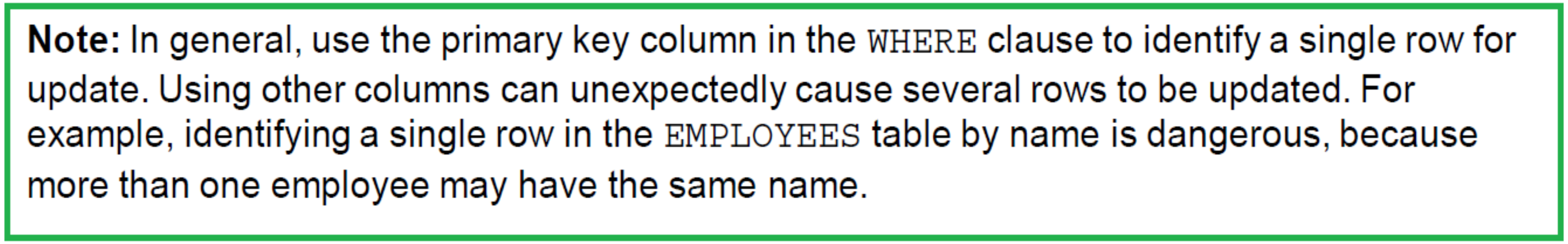


UPDATE Statement Syntax

- Modify existing values in a table with the UPDATE statement:

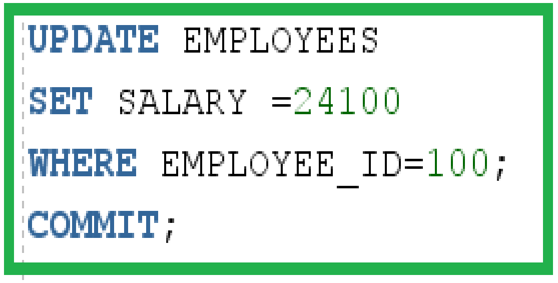
```
UPDATE        table
SET           column = value [, column = value, ...]
[WHERE        condition];
```

- Update more than one row at a time (if required).



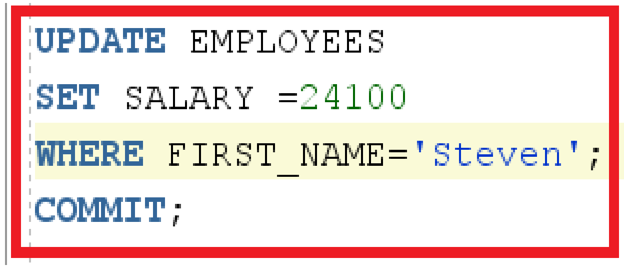
Note: In general, use the primary key column in the WHERE clause to identify a single row for update. Using other columns can unexpectedly cause several rows to be updated. For example, identifying a single row in the EMPLOYEES table by name is dangerous, because more than one employee may have the same name.

Here you guarantee one row update



```
UPDATE EMPLOYEES
SET SALARY =24100
WHERE EMPLOYEE_ID=100;
COMMIT;
```

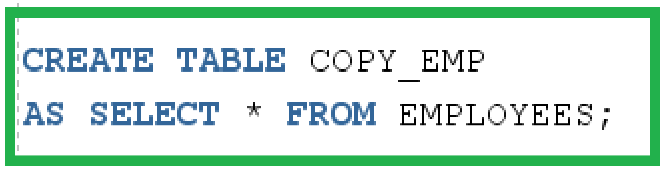
it could be more than one employee with name='Steven'



```
UPDATE EMPLOYEES
SET SALARY =24100
WHERE FIRST_NAME='Steven';
COMMIT;
```

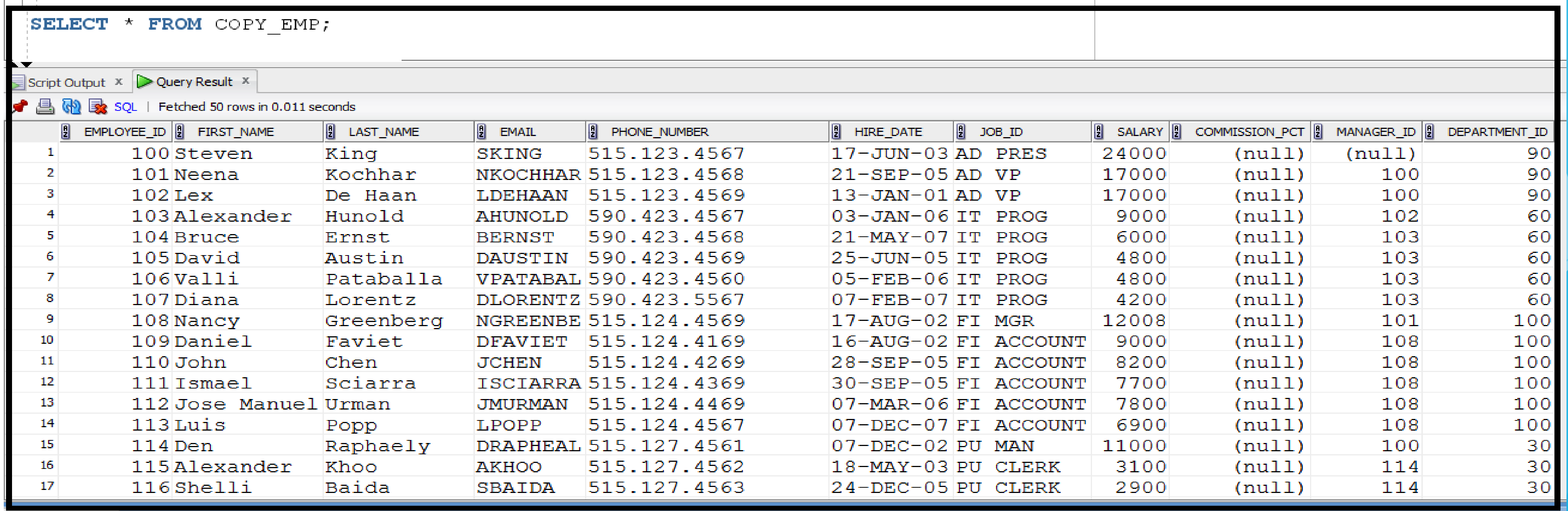
In order to do more practices lets create table called copy_emp
And this table will be a copy from employees table

To do this, we will execute the following SQL:



```
CREATE TABLE COPY_EMP
AS SELECT * FROM EMPLOYEES;
```

This will create table exactly like employees , but without creating the constraints like employee table, expect not null constraints



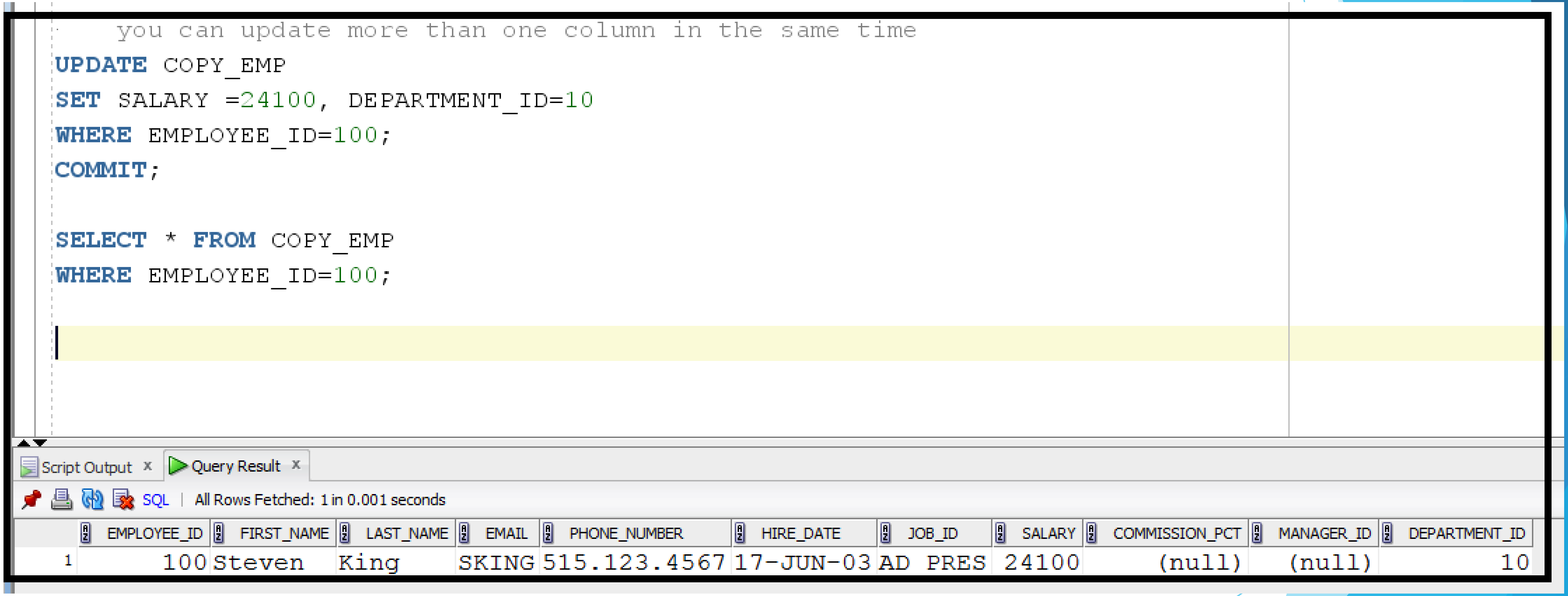
```
SELECT * FROM COPY_EMP;
```

Fetched 50 rows in 0.011 seconds

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	100	Steven	King	SKING	515.123.4567	17-JUN-03	AD PRES	24000	(null)	(null)	90
2	101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-05	AD VP	17000	(null)	100	90
3	102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-01	AD VP	17000	(null)	100	90
4	103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-06	IT PROG	9000	(null)	102	60
5	104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-07	IT PROG	6000	(null)	103	60
6	105	David	Austin	DAUSTIN	590.423.4569	25-JUN-05	IT PROG	4800	(null)	103	60
7	106	Valli	Pataballa	VPATABAL	590.423.4560	05-FEB-06	IT PROG	4800	(null)	103	60
8	107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-07	IT PROG	4200	(null)	103	60
9	108	Nancy	Greenberg	NGREENBE	515.124.4569	17-AUG-02	FI MGR	12008	(null)	101	100
10	109	Daniel	Faviet	DFAVIET	515.124.4169	16-AUG-02	FI ACCOUNT	9000	(null)	108	100
11	110	John	Chen	JCHEN	515.124.4269	28-SEP-05	FI ACCOUNT	8200	(null)	108	100
12	111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-SEP-05	FI ACCOUNT	7700	(null)	108	100
13	112	Jose Manuel	Urman	JMURMAN	515.124.4469	07-MAR-06	FI ACCOUNT	7800	(null)	108	100
14	113	Luis	Popp	LPOPP	515.124.4567	07-DEC-07	FI ACCOUNT	6900	(null)	108	100
15	114	Den	Raphaely	DRAPHEAL	515.127.4561	07-DEC-02	PU MAN	11000	(null)	100	30
16	115	Alexander	Khoo	AKHOO	515.127.4562	18-MAY-03	PU CLERK	3100	(null)	114	30
17	116	Shelli	Baida	SBAIDA	515.127.4563	24-DEC-05	PU CLERK	2900	(null)	114	30

Updating more than one column in the same time

Use comma (,) followed by column name

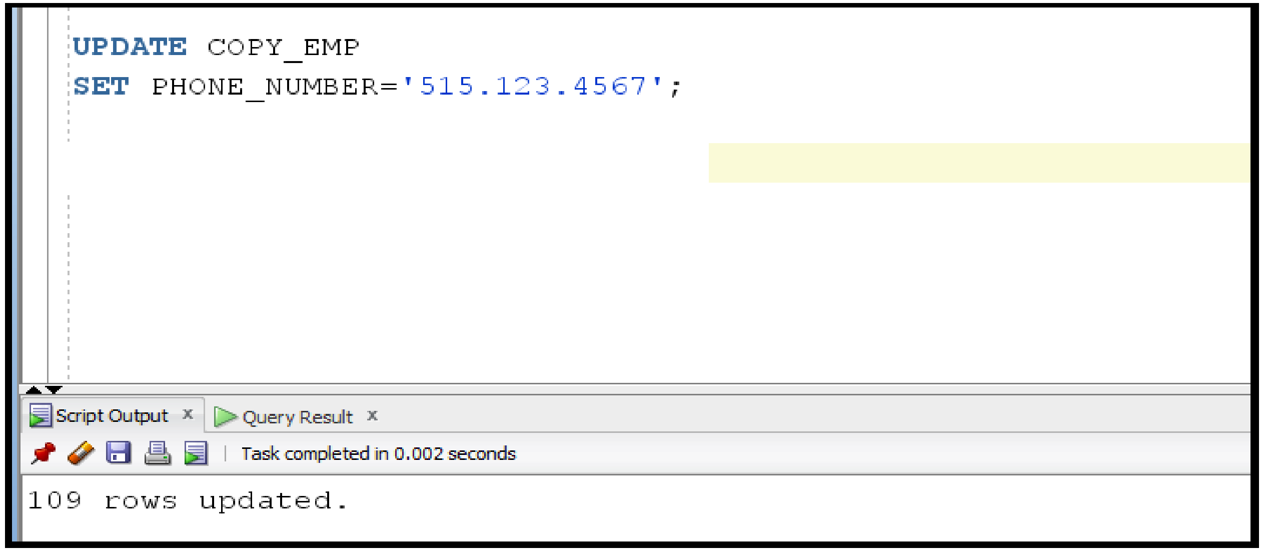


```
    you can update more than one column in the same time
UPDATE COPY_EMP
SET SALARY =24100, DEPARTMENT_ID=10
WHERE EMPLOYEE_ID=100;
COMMIT;

SELECT * FROM COPY_EMP
WHERE EMPLOYEE_ID=100;
```

All Rows Fetched: 1 in 0.001 seconds

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	100	Steven	King	SKING	515.123.4567	17-JUN-03	AD_PRES	24100	(null)	(null)	10

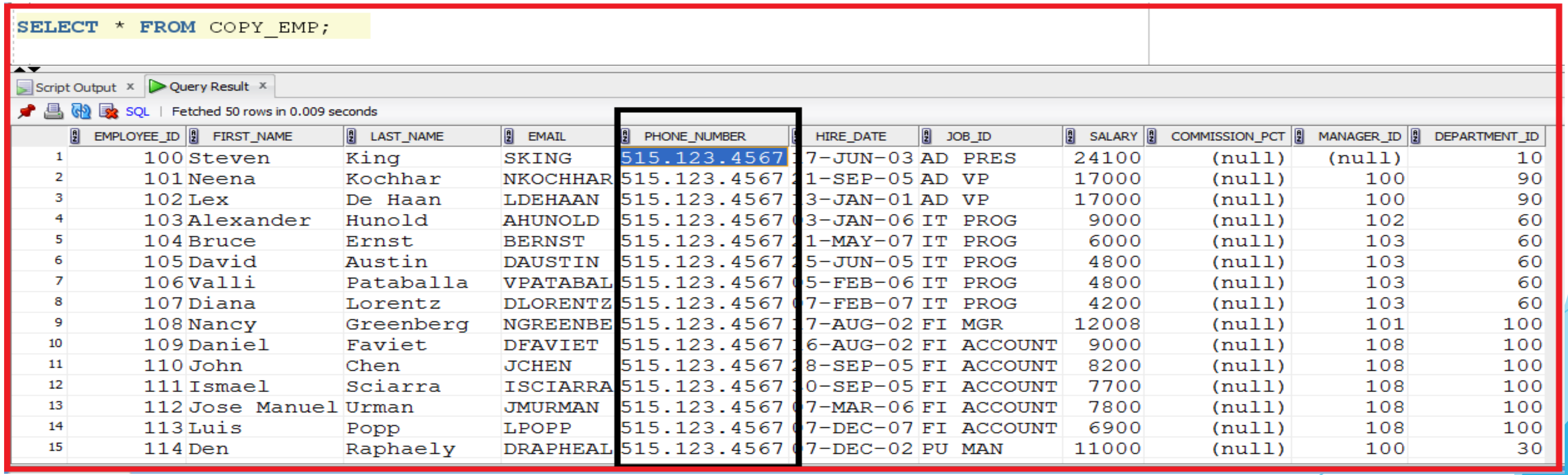
IF there is no where condition then the statement will update all the tables



```
UPDATE COPY_EMP
SET PHONE_NUMBER='515.123.4567';
```

Task completed in 0.002 seconds

```
109 rows updated.
```

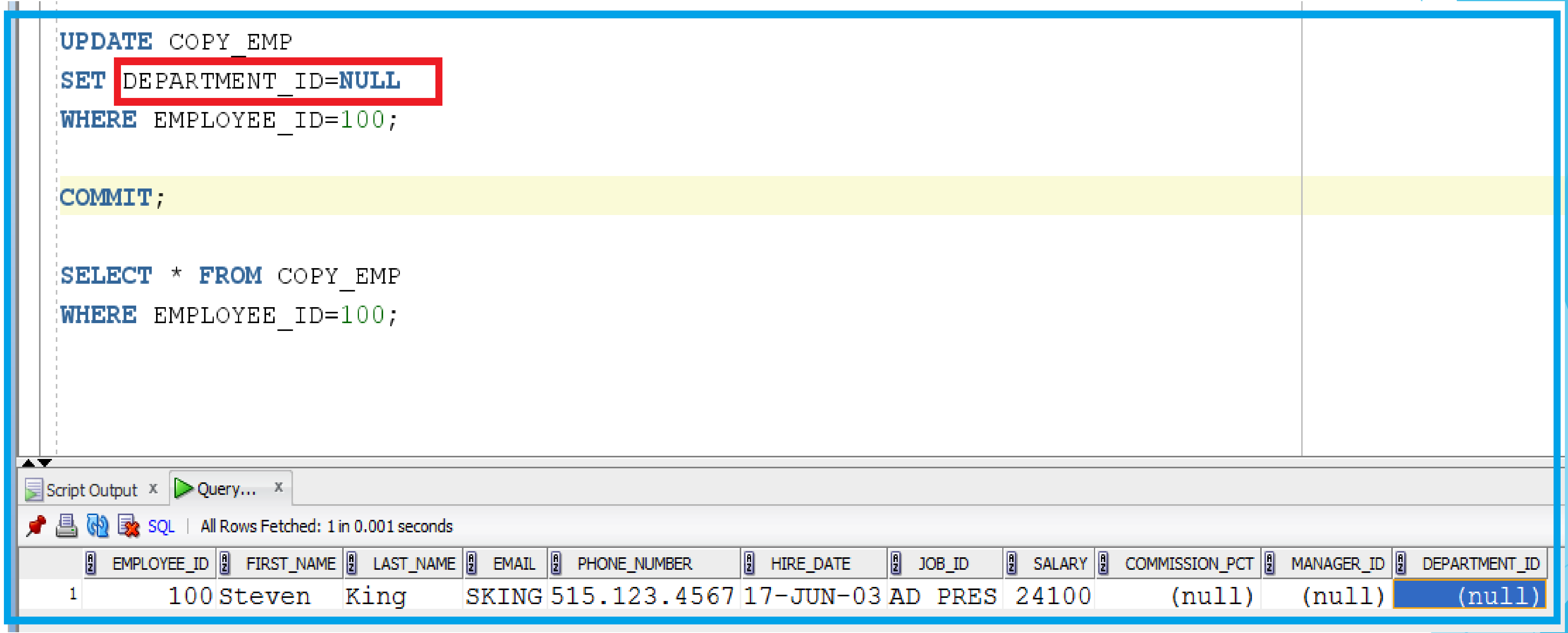


```
SELECT * FROM COPY_EMP;
```

Fetched 50 rows in 0.009 seconds

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	100	Steven	King	SKING	515.123.4567	17-JUN-03	AD_PRES	24100	(null)	(null)	10
2	101	Neena	Kochhar	NKOCHHAR	515.123.4567	21-SEP-05	AD_VP	17000	(null)	100	90
3	102	Lex	De Haan	LDEHAAN	515.123.4567	13-JAN-01	AD_VP	17000	(null)	100	90
4	103	Alexander	Hunold	AHUNOLD	515.123.4567	03-JAN-06	IT_PROG	9000	(null)	102	60
5	104	Bruce	Ernst	BERNST	515.123.4567	21-MAY-07	IT_PROG	6000	(null)	103	60
6	105	David	Austin	DAUSTIN	515.123.4567	25-JUN-05	IT_PROG	4800	(null)	103	60
7	106	Valli	Pataballa	VPATABAL	515.123.4567	05-FEB-06	IT_PROG	4800	(null)	103	60
8	107	Diana	Lorentz	DLORENTZ	515.123.4567	07-FEB-07	IT_PROG	4200	(null)	103	60
9	108	Nancy	Greenberg	NGREENBE	515.123.4567	17-AUG-02	FI_MGR	12008	(null)	101	100
10	109	Daniel	Faviet	DFAVIET	515.123.4567	16-AUG-02	FI_ACCOUNT	9000	(null)	108	100
11	110	John	Chen	JCHEN	515.123.4567	28-SEP-05	FI_ACCOUNT	8200	(null)	108	100
12	111	Ismael	Sciarra	ISCIARRA	515.123.4567	30-SEP-05	FI_ACCOUNT	7700	(null)	108	100
13	112	Jose Manuel	Urman	JMURMAN	515.123.4567	07-MAR-06	FI_ACCOUNT	7800	(null)	108	100
14	113	Luis	Popp	LPOPP	515.123.4567	07-DEC-07	FI_ACCOUNT	6900	(null)	108	100
15	114	Den	Raphaely	DRAPHEAL	515.123.4567	07-DEC-02	PU_MAN	11000	(null)	100	30

You can set the column to null in the update statement.



```
UPDATE COPY_EMP
SET DEPARTMENT_ID=NULL
WHERE EMPLOYEE_ID=100;

COMMIT;

SELECT * FROM COPY_EMP
WHERE EMPLOYEE_ID=100;
```

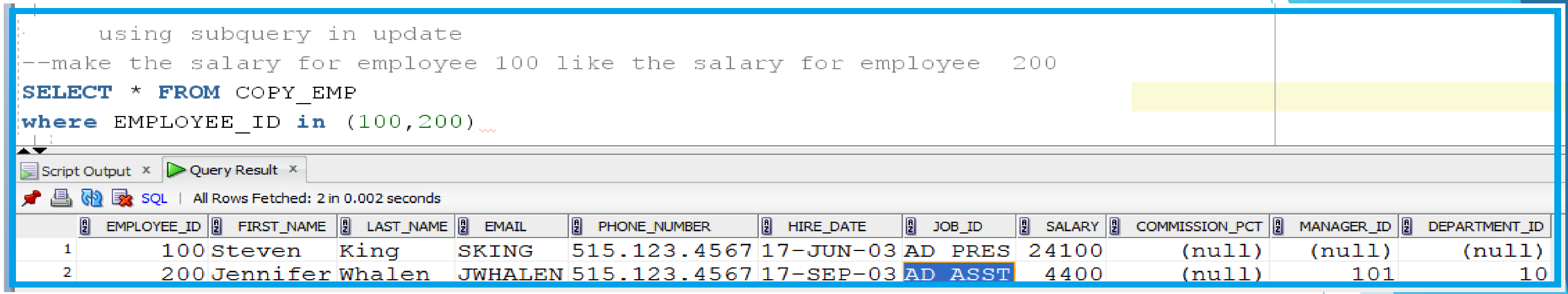
All Rows Fetched: 1 in 0.001 seconds

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	100	Steven	King	SKING	515.123.4567	17-JUN-03	AD_PRES	24100	(null)	(null)	(null)

But there is nothing in select ……..column =null

In select we use column is null/ column is not null

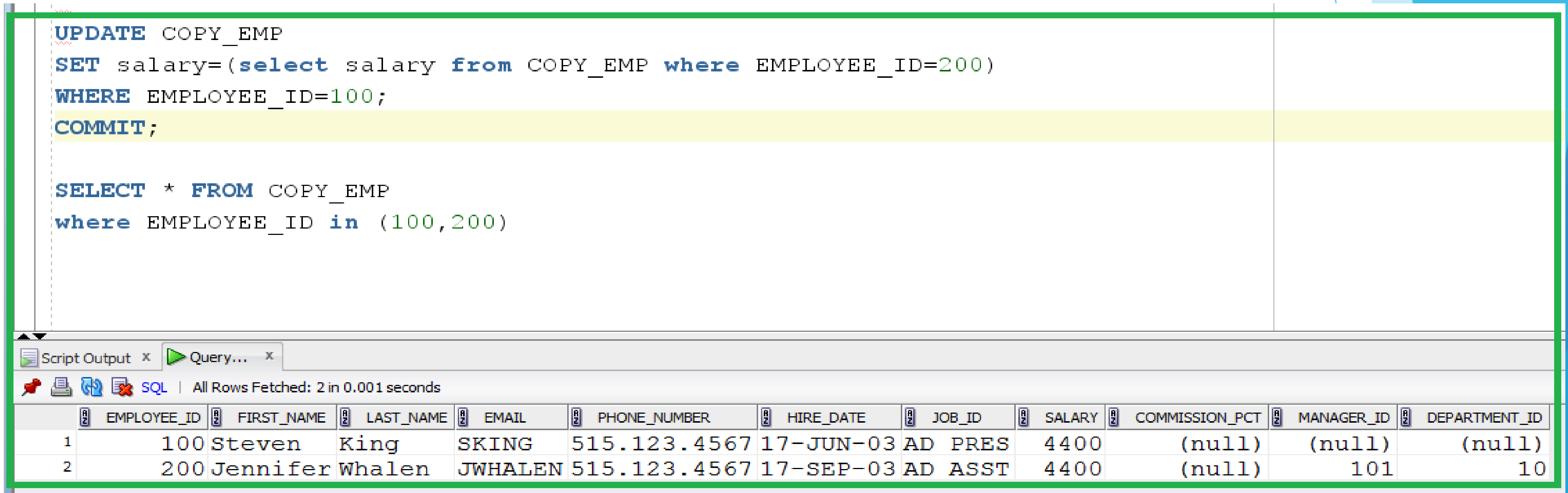
Using Subquery in Update



```
    using subquery in update
--make the salary for employee 100 like the salary for employee  200
SELECT * FROM COPY_EMP
where EMPLOYEE_ID in (100,200)
```

All Rows Fetched: 2 in 0.002 seconds

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	100	Steven	King	SKING	515.123.4567	17-JUN-03	AD_PRES	24100	(null)	(null)	(null)
2	200	Jennifer	Whalen	JWHALEN	515.123.4567	17-SEP-03	AD_ASST	4400	(null)	101	10



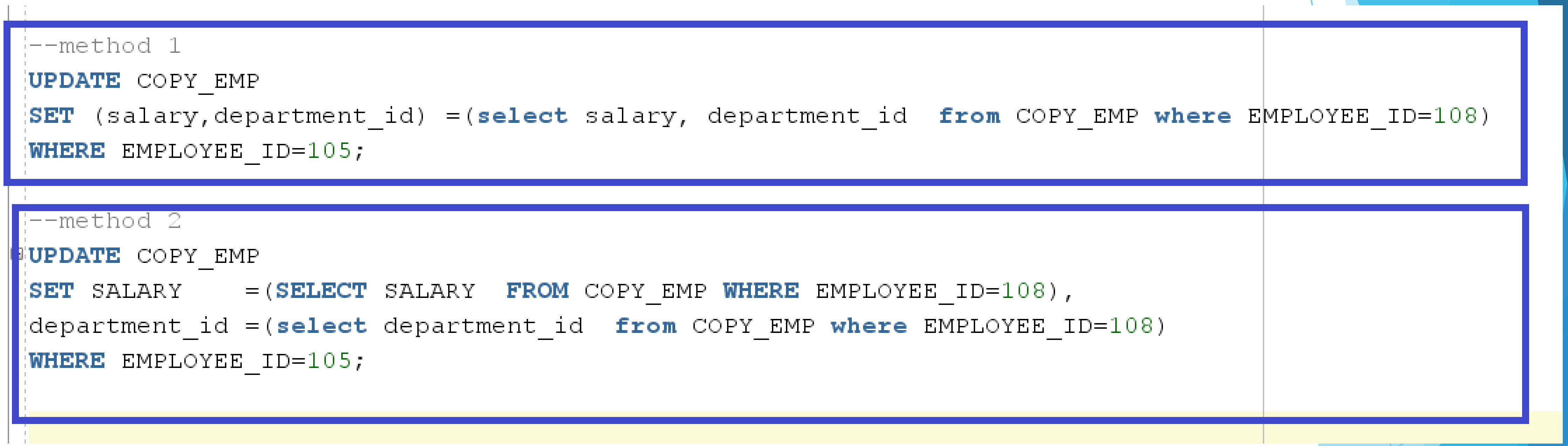
```
UPDATE COPY_EMP
SET salary=(select salary from COPY_EMP where EMPLOYEE_ID=200)
WHERE EMPLOYEE_ID=100;
COMMIT;

SELECT * FROM COPY_EMP
where EMPLOYEE_ID in (100,200)
```

All Rows Fetched: 2 in 0.001 seconds

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	100	Steven	King	SKING	515.123.4567	17-JUN-03	AD_PRES	4400	(null)	(null)	(null)
2	200	Jennifer	Whalen	JWHALEN	515.123.4567	17-SEP-03	AD_ASST	4400	(null)	101	10

Using Subquery in Update

make the salary and department id for employee 105 like the salary and department id for employee 108   ????????

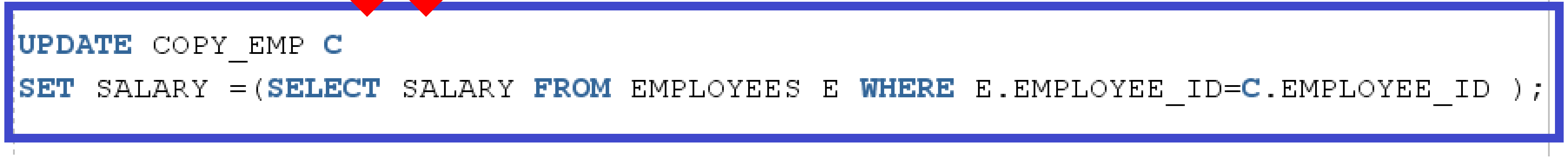


```
--method 1
UPDATE COPY_EMP
SET (salary,department_id) =(select salary, department_id  from COPY_EMP where EMPLOYEE_ID=108)
WHERE EMPLOYEE_ID=105;

--method 2
UPDATE COPY_EMP
SET SALARY     =(SELECT SALARY  FROM COPY_EMP WHERE EMPLOYEE_ID=108),
department_id =(select department_id  from COPY_EMP where EMPLOYEE_ID=108)
WHERE EMPLOYEE_ID=105;
```

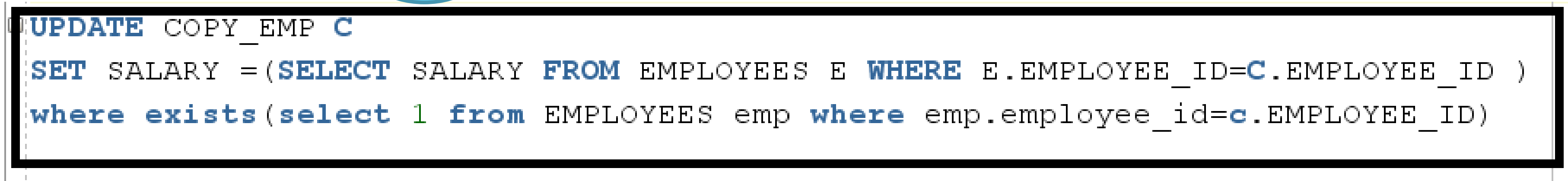
Update based on another table

Make all the salaries in table copy_emp like the salaries in table employees

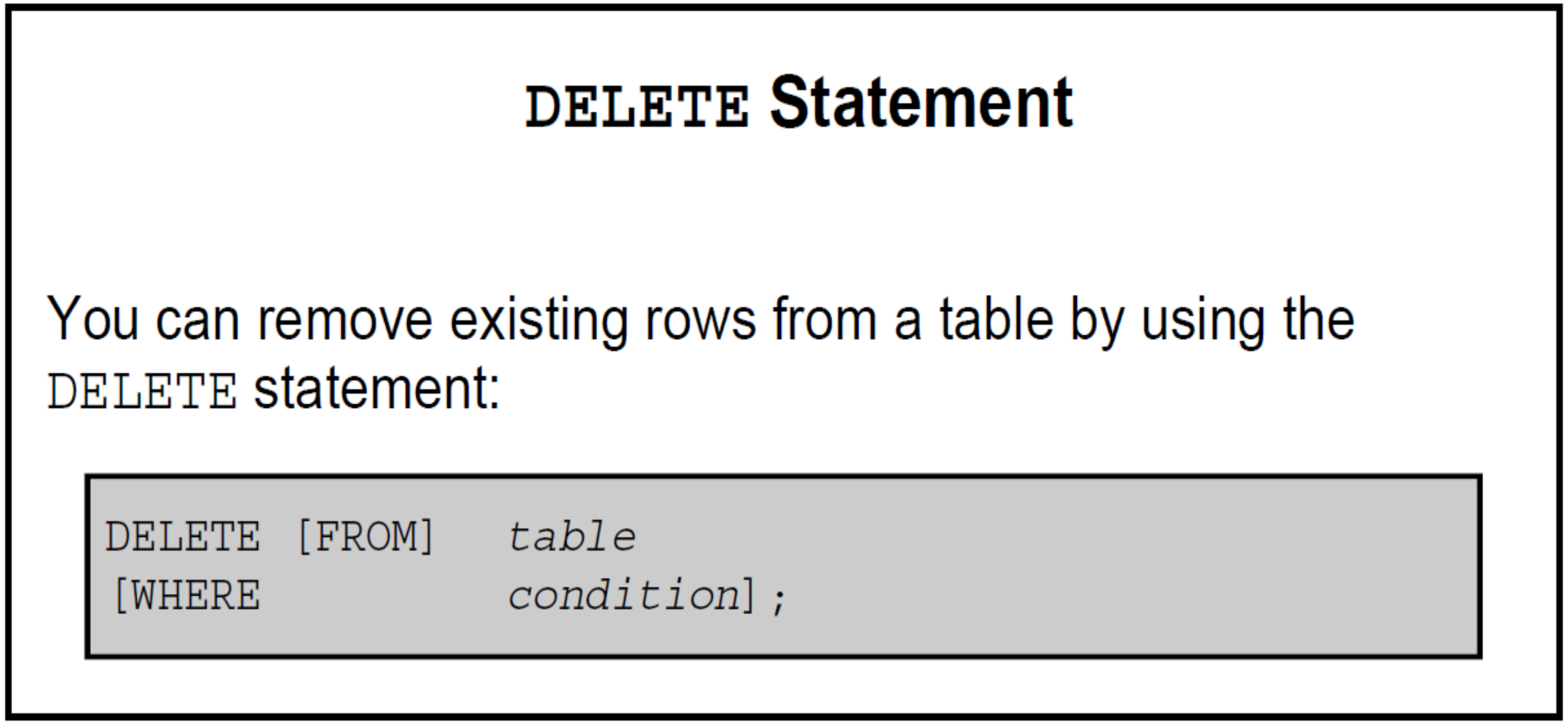
In this case the additional rows in table copy_emp that doesn't meet the conditions will be updated by null values.
When there is no where in the SQL statement this mean all rows will be updated



```sql
UPDATE COPY_EMP C
SET SALARY =(SELECT SALARY FROM EMPLOYEES E WHERE E.EMPLOYEE_ID=C.EMPLOYEE_ID );
```



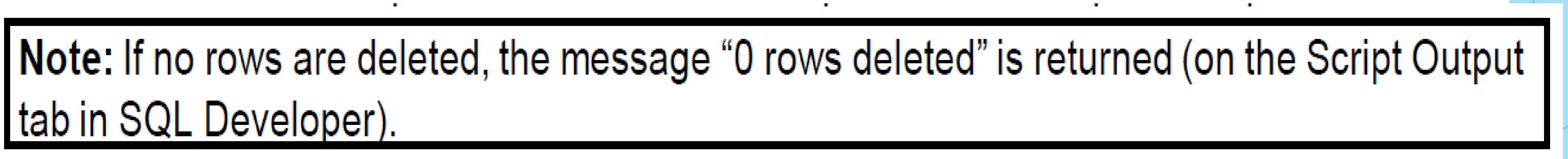
```sql
UPDATE COPY_EMP C
SET SALARY =(SELECT SALARY FROM EMPLOYEES E WHERE E.EMPLOYEE_ID=C.EMPLOYEE_ID )
where exists(select 1 from EMPLOYEES emp where emp.employee_id=c.EMPLOYEE_ID)
```



DELETE Statement

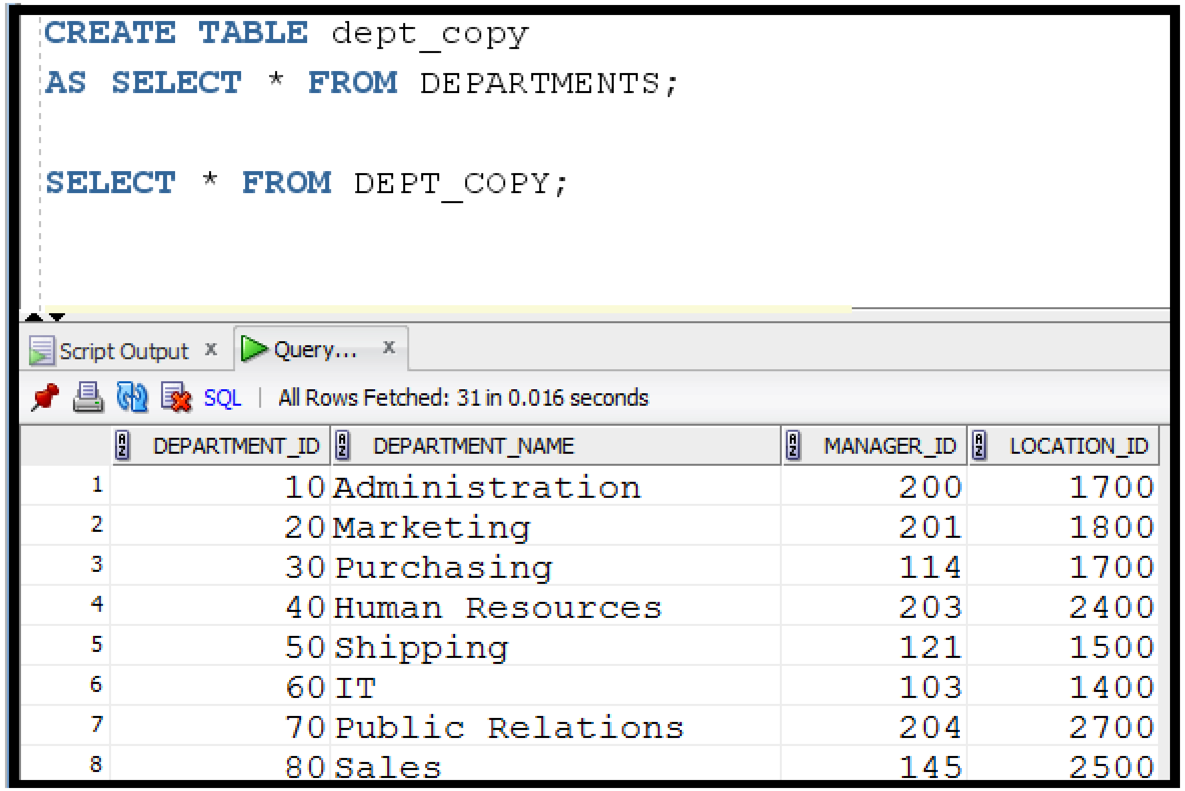
You can remove existing rows from a table by using the DELETE statement:

```
DELETE  [FROM]    table
[WHERE            condition];
```



Note: If no rows are deleted, the message "0 rows deleted" is returned (on the Script Output tab in SQL Developer).

We will create table dept_copy in order to do some practices

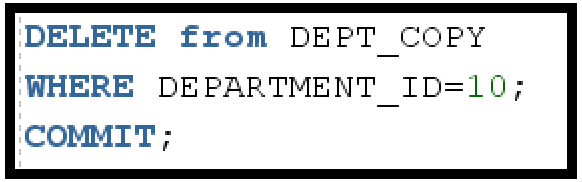


```
CREATE TABLE dept_copy
AS SELECT * FROM DEPARTMENTS;

SELECT * FROM DEPT_COPY;
```

All Rows Fetched: 31 in 0.016 seconds

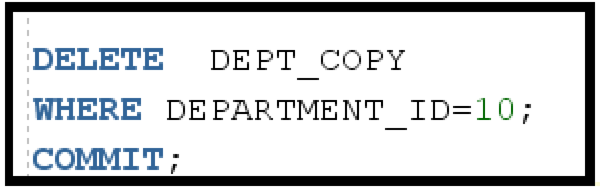
	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	30	Purchasing	114	1700
4	40	Human Resources	203	2400
5	50	Shipping	121	1500
6	60	IT	103	1400
7	70	Public Relations	204	2700
8	80	Sales	145	2500

Note: dept_copy table will have no constraints (PK,FK,..)
Because create table based on select doesn’t copy any constraint like PK,FK
But it make copy for not null constraints



```
DELETE from DEPT_COPY
WHERE DEPARTMENT_ID=10;
COMMIT;
```

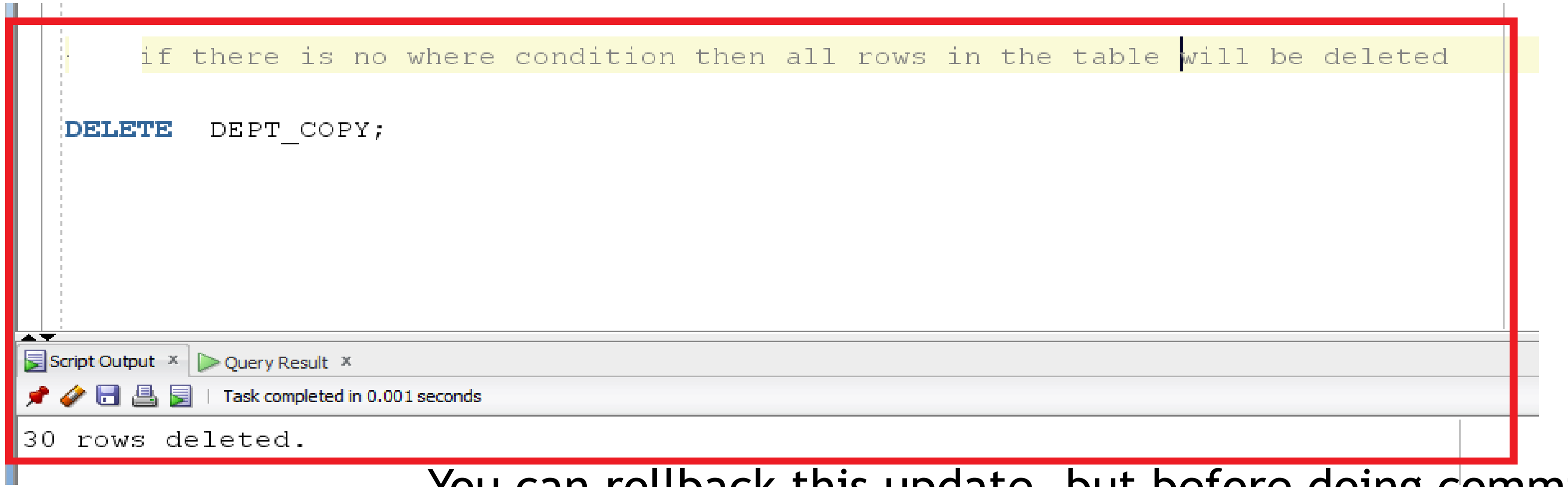
The will return 1 row deleted



```
DELETE  DEPT_COPY
WHERE DEPARTMENT_ID=10;
COMMIT;
```

this will return 0 rows deleted, you already deleted in the previous SQL

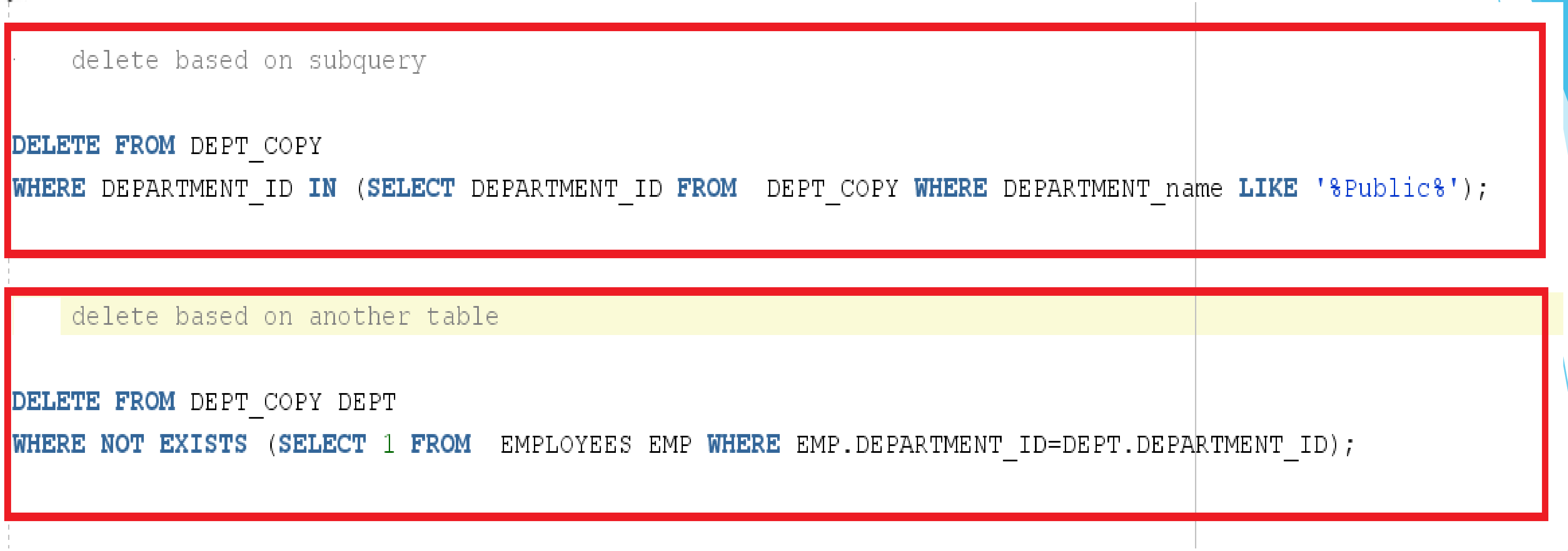
note: the keyword **FROM** is optional



```
    if there is no where condition then all rows in the table will be deleted

DELETE  DEPT_COPY;
```

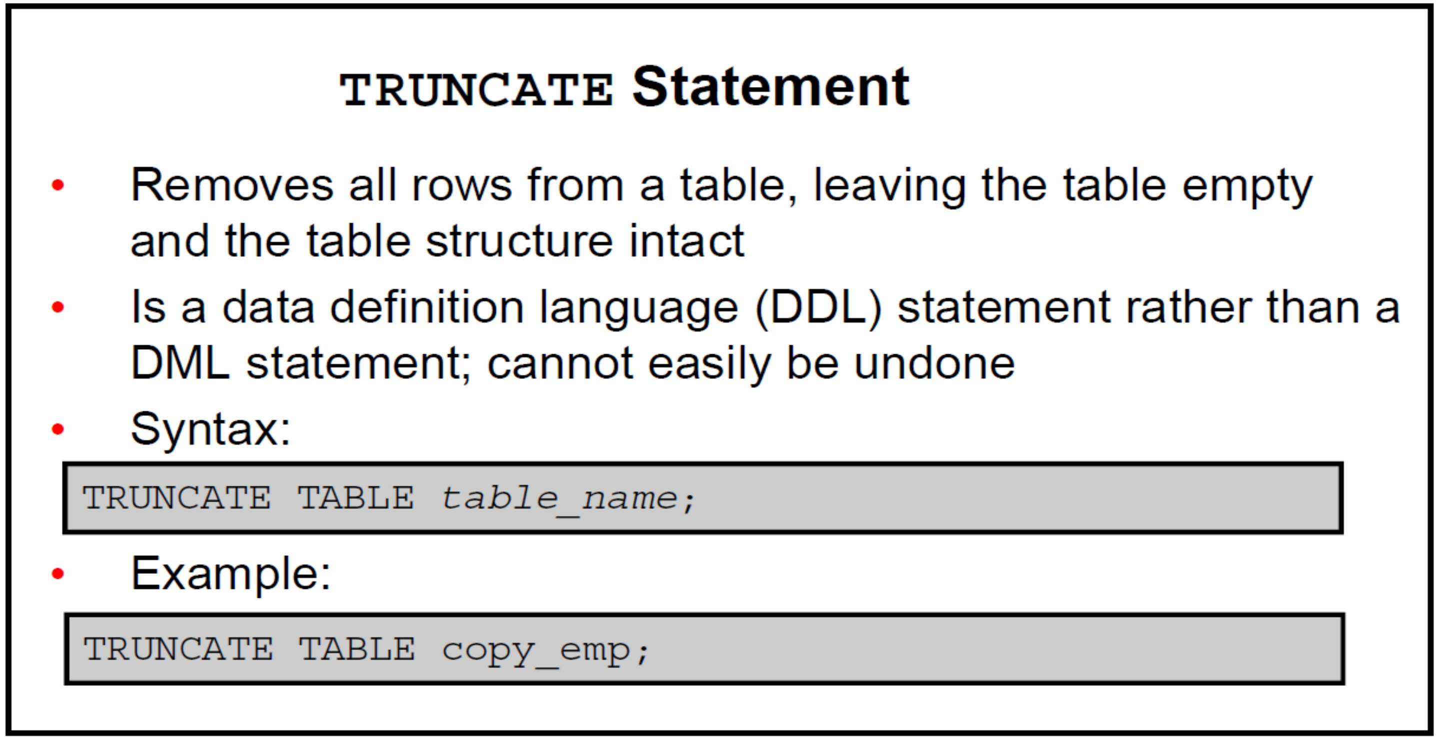
```
30 rows deleted.
```

You can rollback this update, but before doing commit,
If you do commit, then you can not do rollback



```
    delete based on subquery

DELETE FROM DEPT_COPY
WHERE DEPARTMENT_ID IN (SELECT DEPARTMENT_ID FROM  DEPT_COPY WHERE DEPARTMENT_name LIKE '%Public%');
```

```
    delete based on another table

DELETE FROM DEPT_COPY DEPT
WHERE NOT EXISTS (SELECT 1 FROM  EMPLOYEES EMP WHERE EMP.DEPARTMENT_ID=DEPT.DEPARTMENT_ID);
```



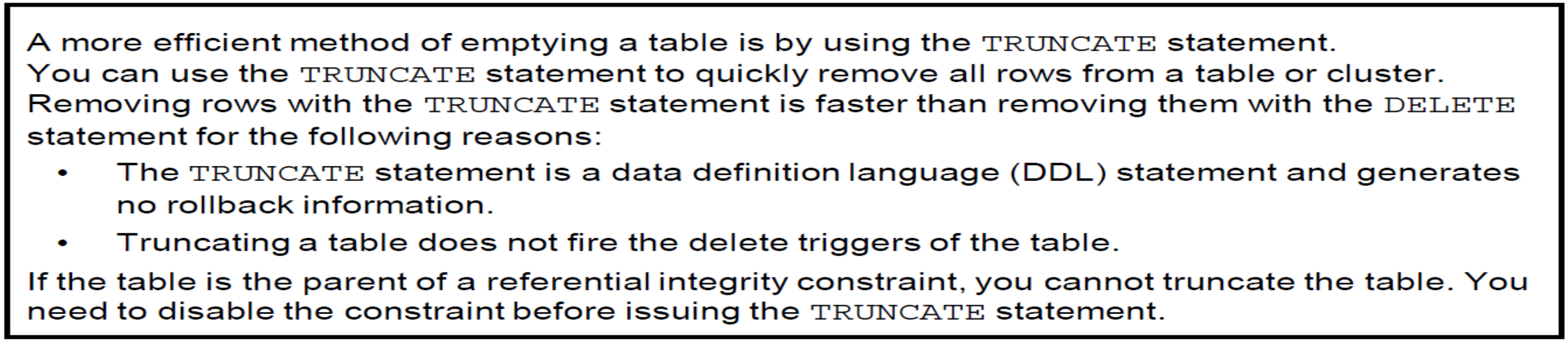
TRUNCATE Statement

- Removes all rows from a table, leaving the table empty and the table structure intact
- Is a data definition language (DDL) statement rather than a DML statement; cannot easily be undone
- Syntax:

```
TRUNCATE TABLE table_name;
```

- Example:

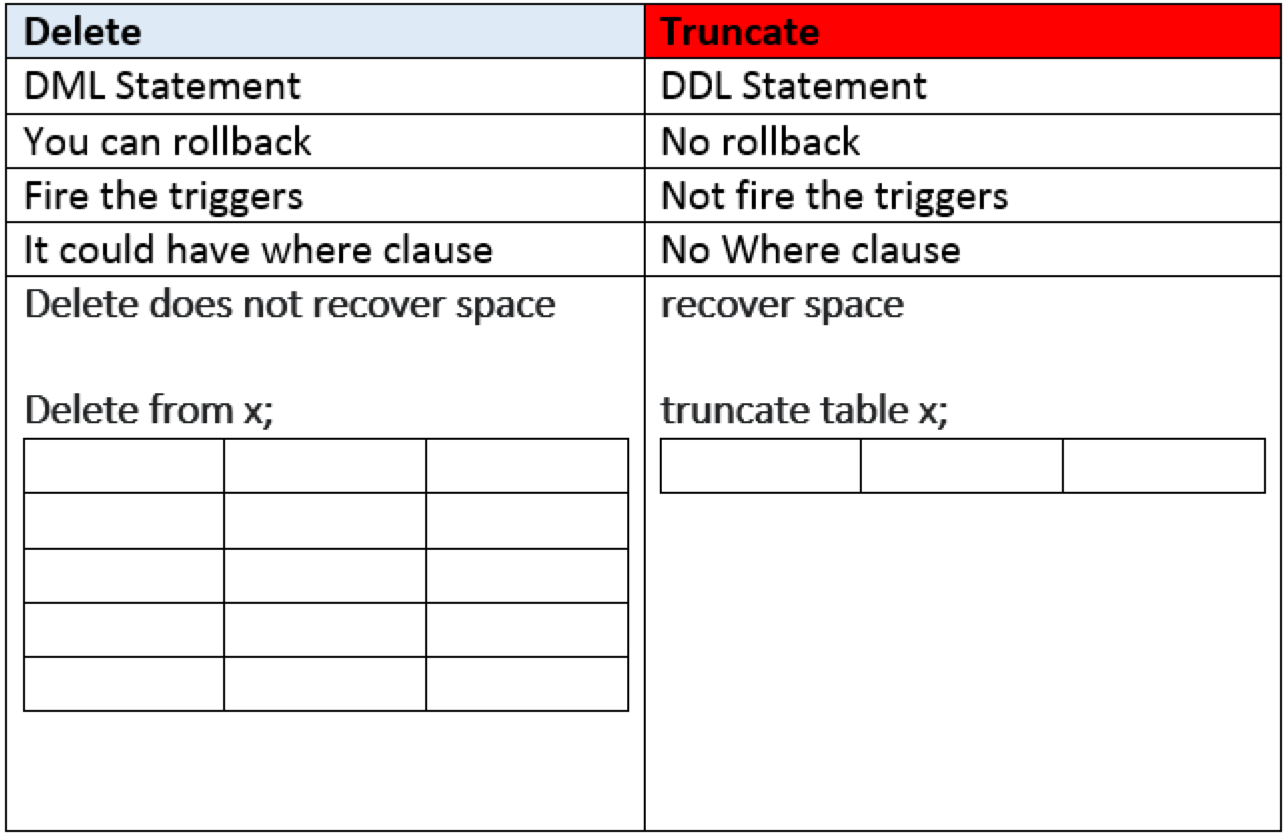
```
TRUNCATE TABLE copy_emp;
```



A more efficient method of emptying a table is by using the TRUNCATE statement. You can use the TRUNCATE statement to quickly remove all rows from a table or cluster. Removing rows with the TRUNCATE statement is faster than removing them with the DELETE statement for the following reasons:

- The TRUNCATE statement is a data definition language (DDL) statement and generates no rollback information.
- Truncating a table does not fire the delete triggers of the table.

If the table is the parent of a referential integrity constraint, you cannot truncate the table. You need to disable the constraint before issuing the TRUNCATE statement.



<table>
<tr><th>Delete</th><th>Truncate</th></tr>
<tr><td>DML Statement</td><td>DDL Statement</td></tr>
<tr><td>You can rollback</td><td>No rollback</td></tr>
<tr><td>Fire the triggers</td><td>Not fire the triggers</td></tr>
<tr><td>It could have where clause</td><td>No Where clause</td></tr>
<tr><td>Delete does not recover space

Delete from x;<table><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr></table></td><td>recover space

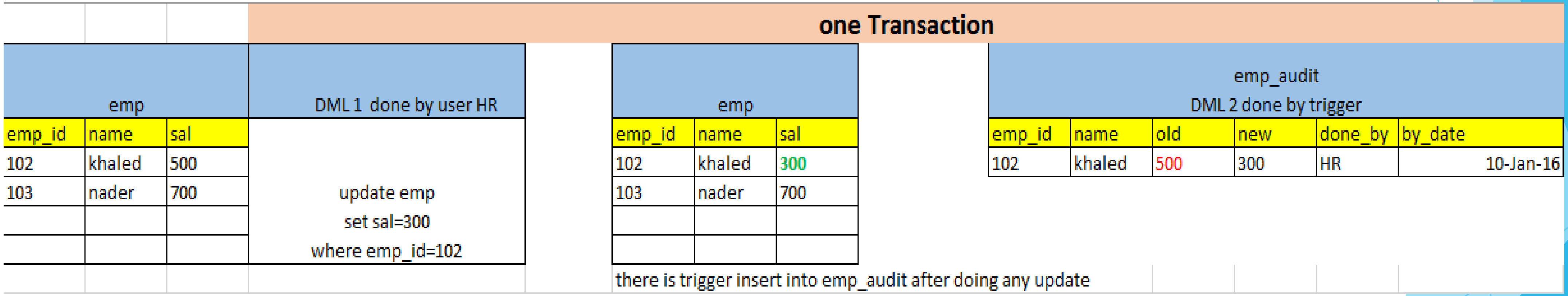
truncate table x;<table><tr><td></td><td></td><td></td></tr></table></td></tr>
</table>

Database Transactions

A database transaction consists of one of the following:

- DML statements that constitute one consistent change to the data
- One DDL statement
- One data control language (DCL) statement

Example1: DML statements from one transaction



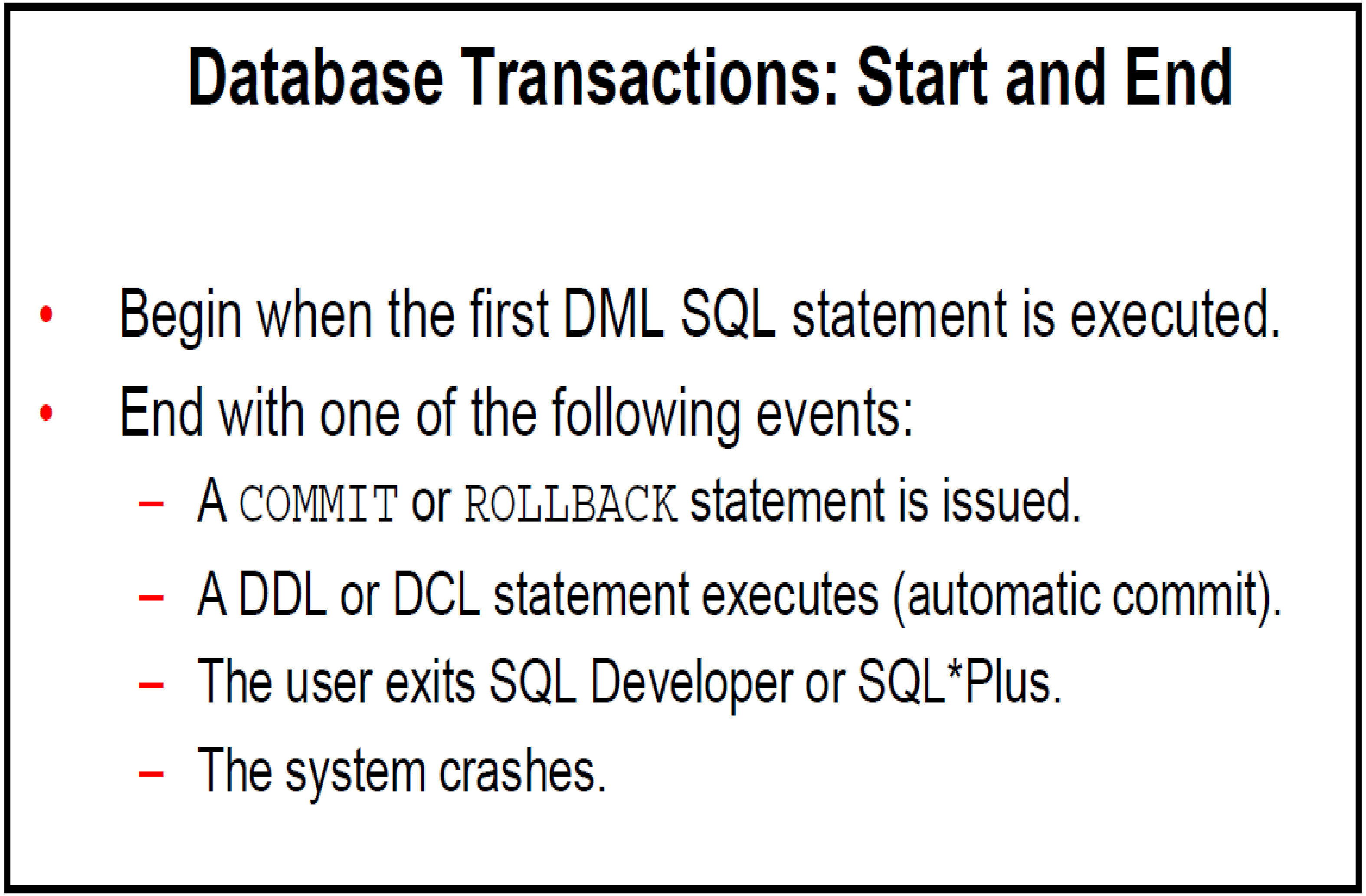
			one Transaction											
emp			DML 1 done by user HR		emp				emp_audit DML 2 done by trigger					
emp_id	name	sal			emp_id	name	sal		emp_id	name	old	new	done_by	by_date
102	khaled	500			102	khaled	300		102	khaled	500	300	HR	10-Jan-16
103	nader	700	update emp		103	nader	700							
			set sal=300											
			where emp_id=102											
					there is trigger insert into emp_audit after doing any update									

Example2: DDL statement (one transaction)

Create table emp_copy as select * from employees

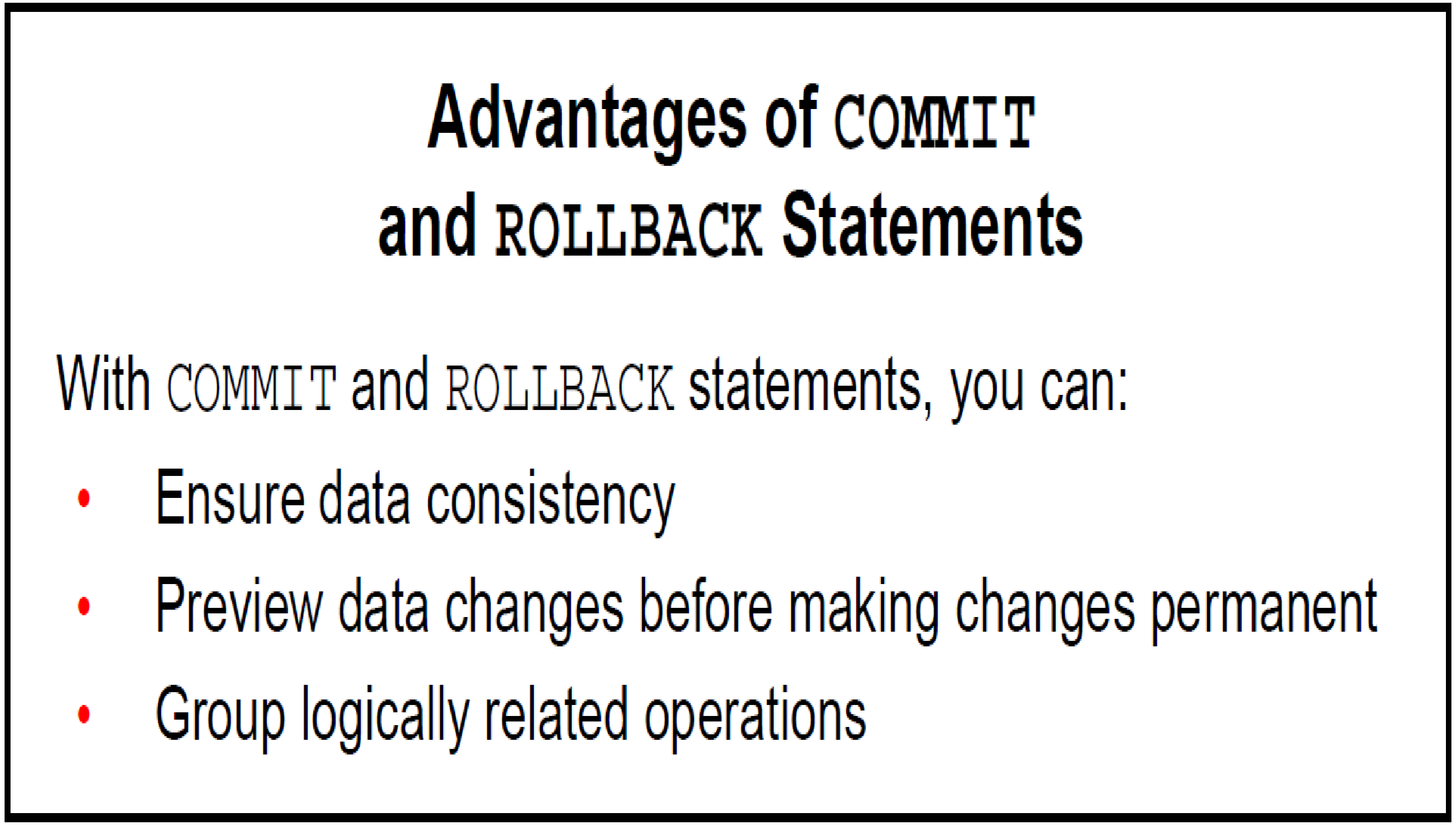
Example3: DCL statement (one transaction)

Grant select on hr.employees to scott



Database Transactions: Start and End

- Begin when the first DML SQL statement is executed.
- End with one of the following events:
  - A COMMIT or ROLLBACK statement is issued.
  - A DDL or DCL statement executes (automatic commit).
  - The user exits SQL Developer or SQL*Plus.
  - The system crashes.

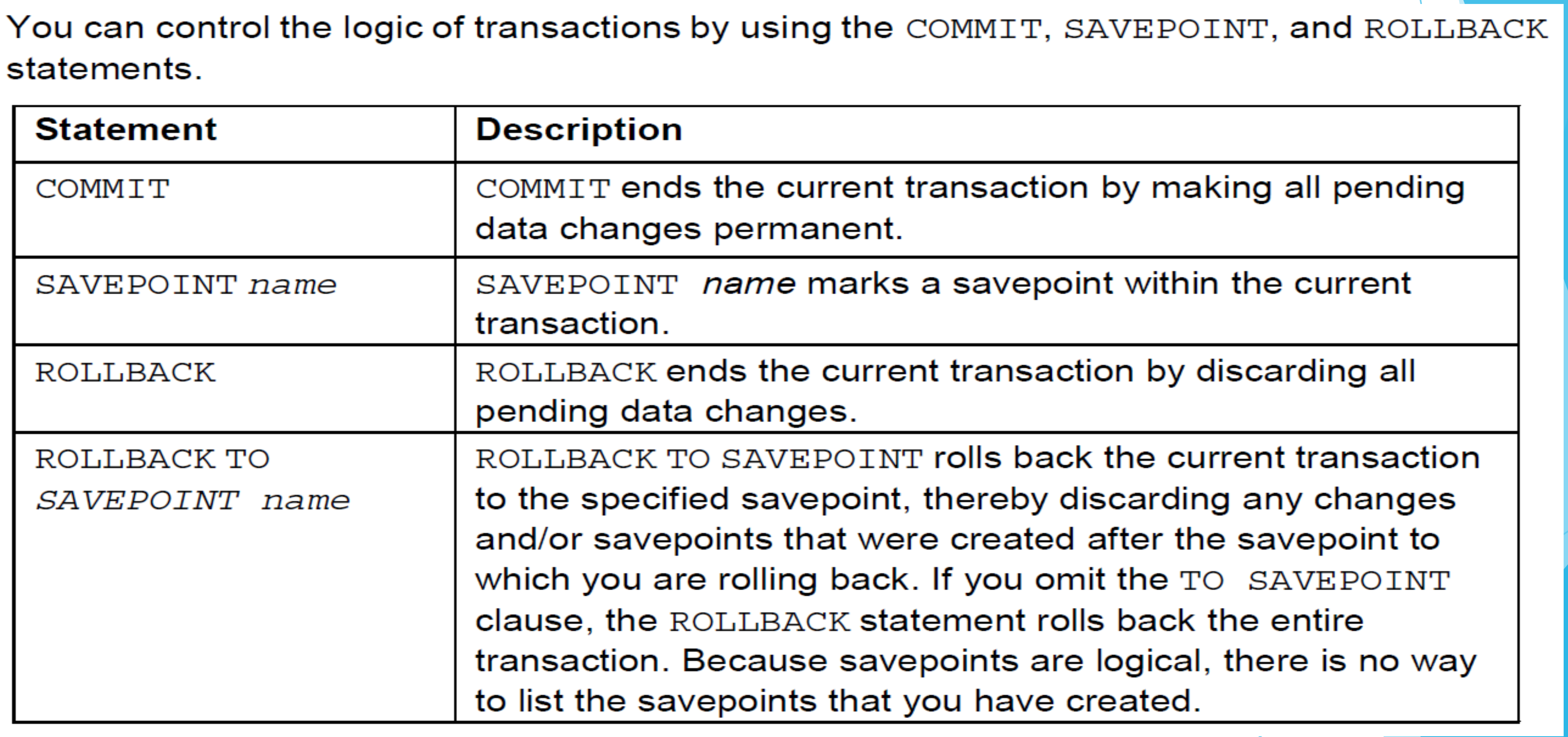


Advantages of COMMIT and ROLLBACK Statements

With COMMIT and ROLLBACK statements, you can:

- Ensure data consistency
- Preview data changes before making changes permanent
- Group logically related operations

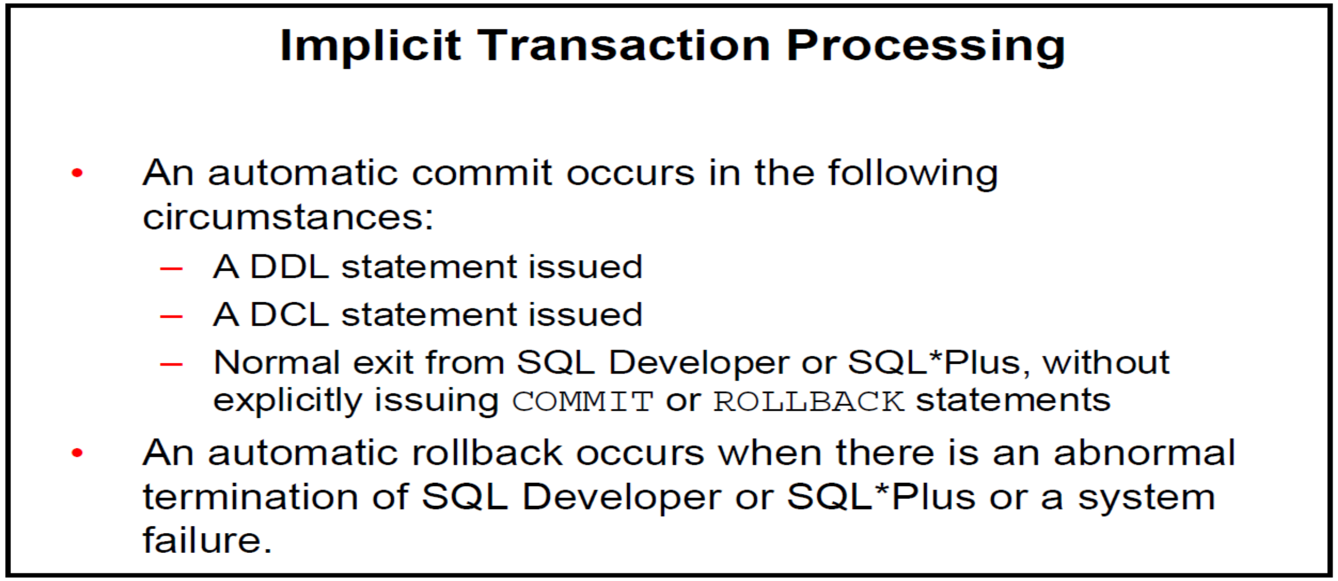
Explicit Transaction Control Statements



You can control the logic of transactions by using the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements.

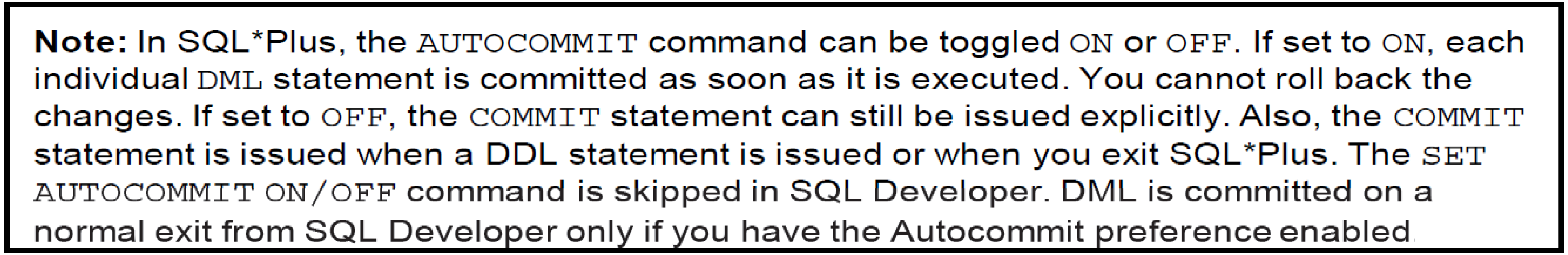
Statement	Description
`COMMIT`	`COMMIT` ends the current transaction by making all pending data changes permanent.
`SAVEPOINT` *name*	`SAVEPOINT` *name* marks a savepoint within the current transaction.
`ROLLBACK`	`ROLLBACK` ends the current transaction by discarding all pending data changes.

`ROLLBACK TO`
`SAVEPOINT name`
	`ROLLBACK TO SAVEPOINT` rolls back the current transaction to the specified savepoint, thereby discarding any changes and/or savepoints that were created after the savepoint to which you are rolling back. If you omit the `TO SAVEPOINT` clause, the `ROLLBACK` statement rolls back the entire transaction. Because savepoints are logical, there is no way to list the savepoints that you have created.

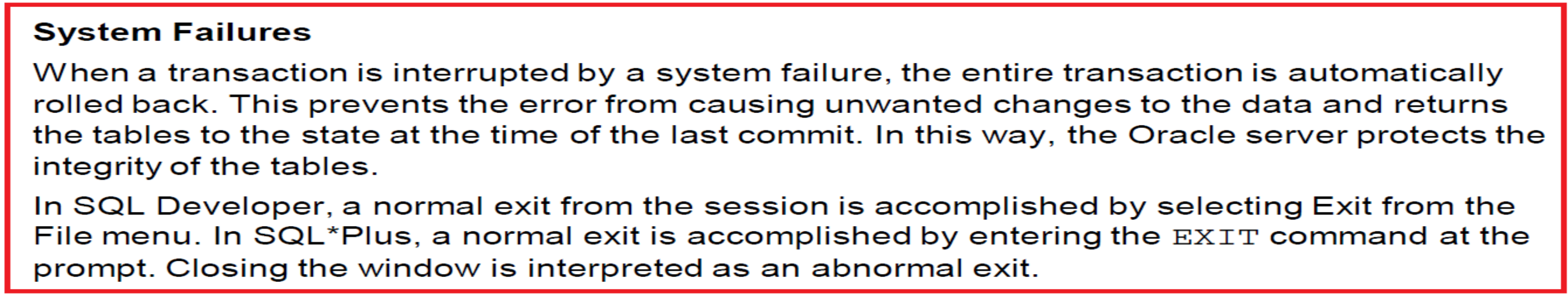


Implicit Transaction Processing

- An automatic commit occurs in the following circumstances:
  - A DDL statement issued
  - A DCL statement issued
  - Normal exit from SQL Developer or SQL*Plus, without explicitly issuing `COMMIT` or `ROLLBACK` statements
- An automatic rollback occurs when there is an abnormal termination of SQL Developer or SQL*Plus or a system failure.



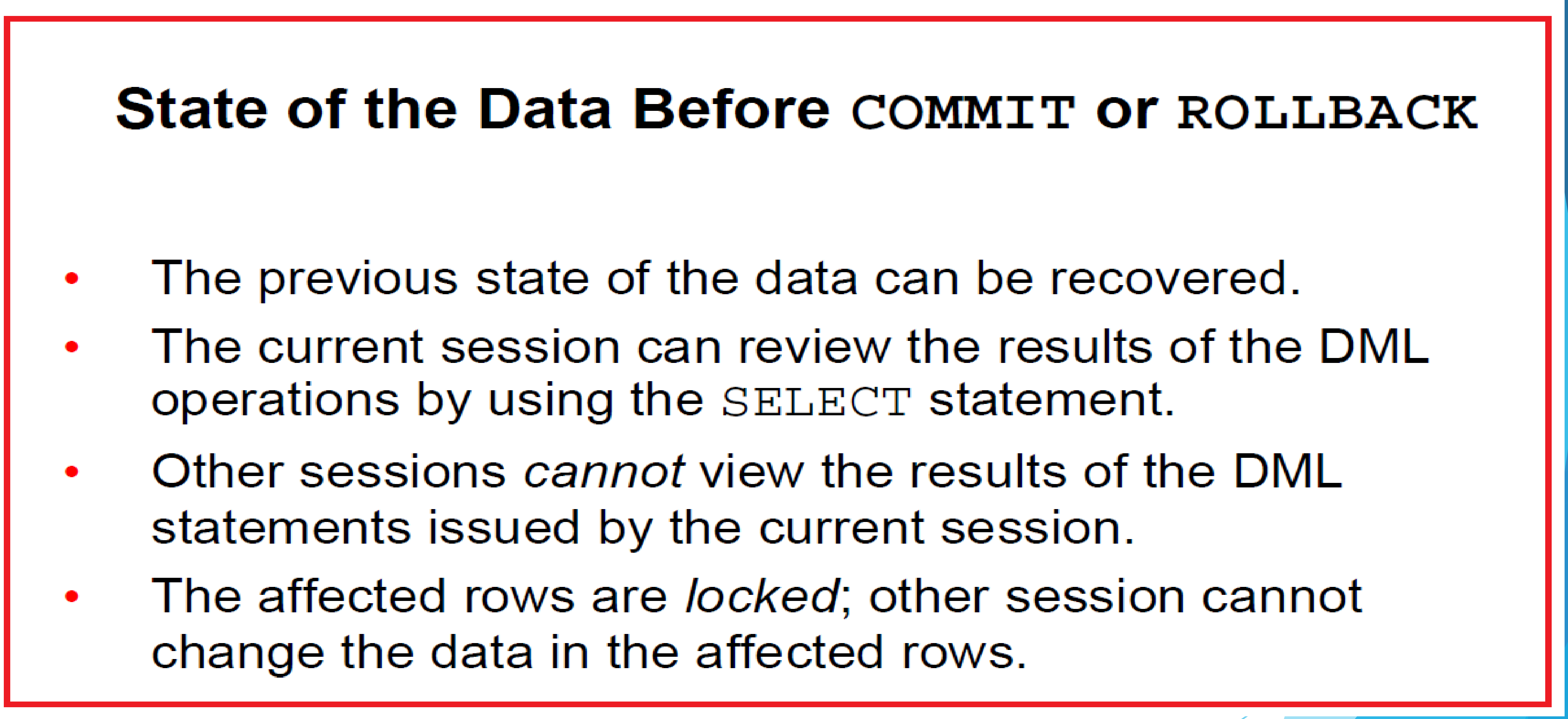
Note: In SQL*Plus, the `AUTOCOMMIT` command can be toggled `ON` or `OFF`. If set to `ON`, each individual `DML` statement is committed as soon as it is executed. You cannot roll back the changes. If set to `OFF`, the `COMMIT` statement can still be issued explicitly. Also, the `COMMIT` statement is issued when a DDL statement is issued or when you exit SQL*Plus. The `SET AUTOCOMMIT ON/OFF` command is skipped in SQL Developer. DML is committed on a normal exit from SQL Developer only if you have the Autocommit preference enabled.



System Failures

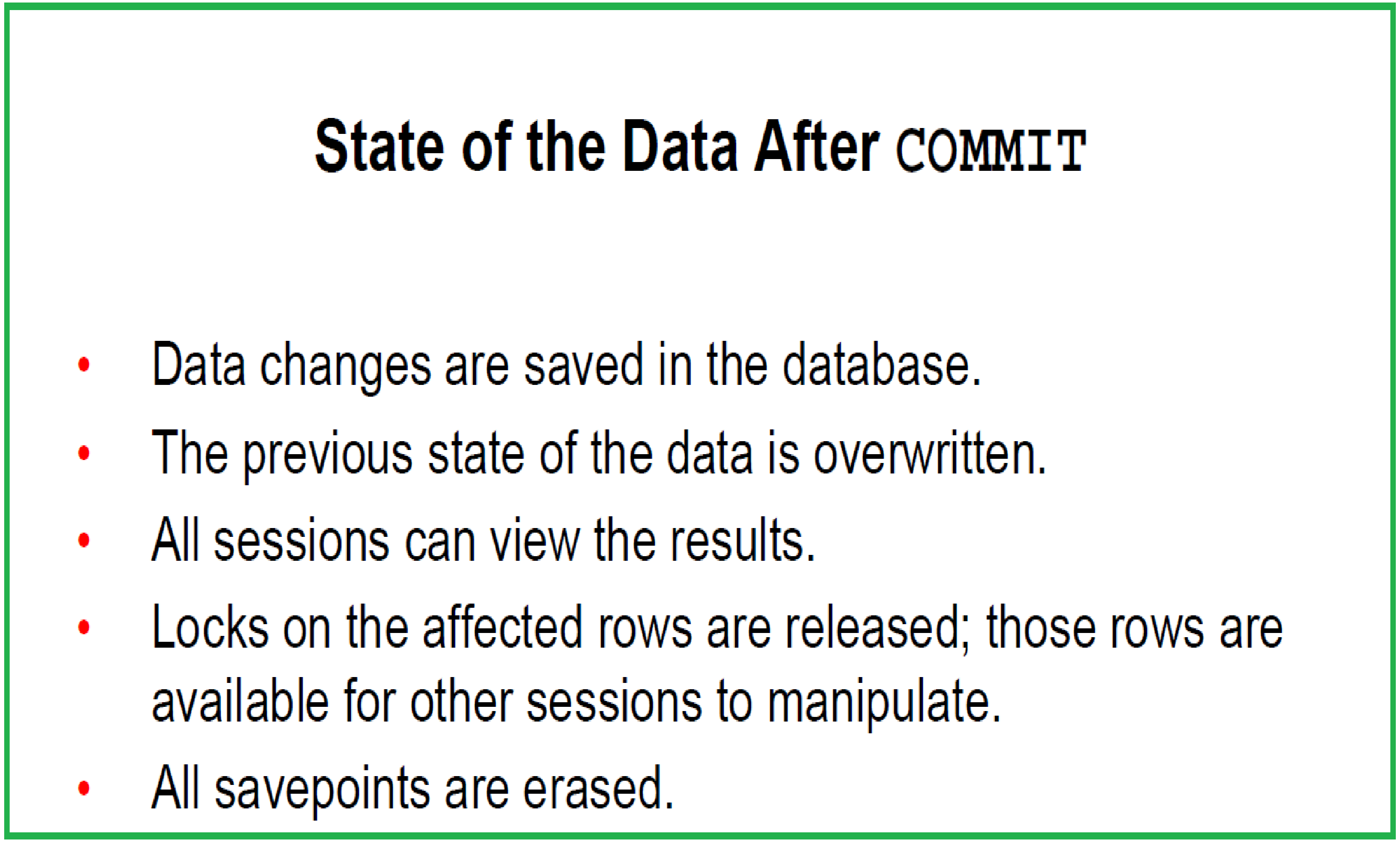
When a transaction is interrupted by a system failure, the entire transaction is automatically rolled back. This prevents the error from causing unwanted changes to the data and returns the tables to the state at the time of the last commit. In this way, the Oracle server protects the integrity of the tables.

In SQL Developer, a normal exit from the session is accomplished by selecting Exit from the File menu. In SQL*Plus, a normal exit is accomplished by entering the `EXIT` command at the prompt. Closing the window is interpreted as an abnormal exit.



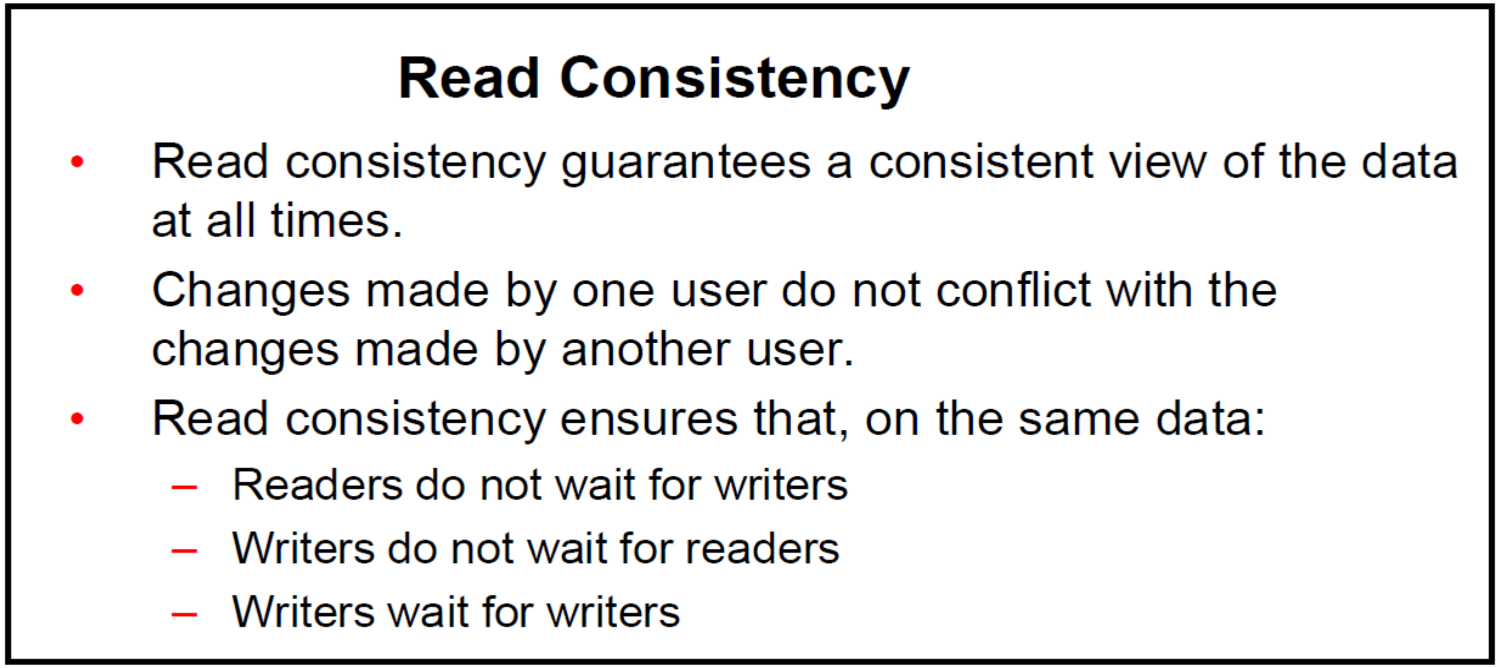
State of the Data Before `COMMIT` or `ROLLBACK`

- The previous state of the data can be recovered.
- The current session can review the results of the DML operations by using the `SELECT` statement.
- Other sessions *cannot* view the results of the DML statements issued by the current session.
- The affected rows are *locked*; other session cannot change the data in the affected rows.



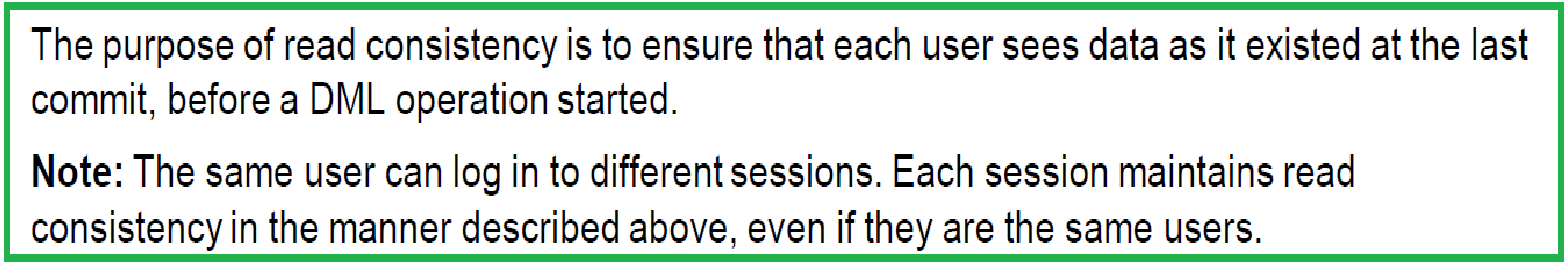
State of the Data After COMMIT

- Data changes are saved in the database.
- The previous state of the data is overwritten.
- All sessions can view the results.
- Locks on the affected rows are released; those rows are available for other sessions to manipulate.
- All savepoints are erased.



Read Consistency

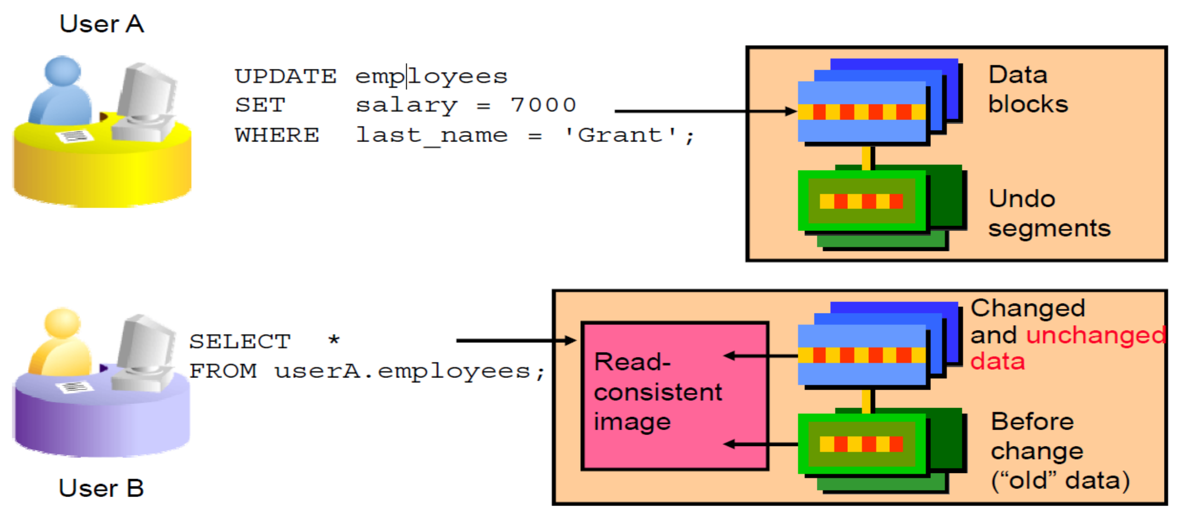
- Read consistency guarantees a consistent view of the data at all times.
- Changes made by one user do not conflict with the changes made by another user.
- Read consistency ensures that, on the same data:
  - Readers do not wait for writers
  - Writers do not wait for readers
  - Writers wait for writers



The purpose of read consistency is to ensure that each user sees data as it existed at the last commit, before a DML operation started.

Note: The same user can log in to different sessions. Each session maintains read consistency in the manner described above, even if they are the same users.

Implementing Read Consistency



```
UPDATE  employees
SET     salary = 7000
WHERE   last_name = 'Grant';
```

```
SELECT   *
FROM userA.employees;
```

Read consistency is an automatic implementation. It keeps a partial copy of the database in the undo segments. The read-consistent image is constructed from the committed data in the table and the old data that is being changed and is not yet committed from the undo segment.

When an insert, update, or delete operation is made on the database, the Oracle server takes a copy of the data before it is changed and writes it to an *undo segment*.

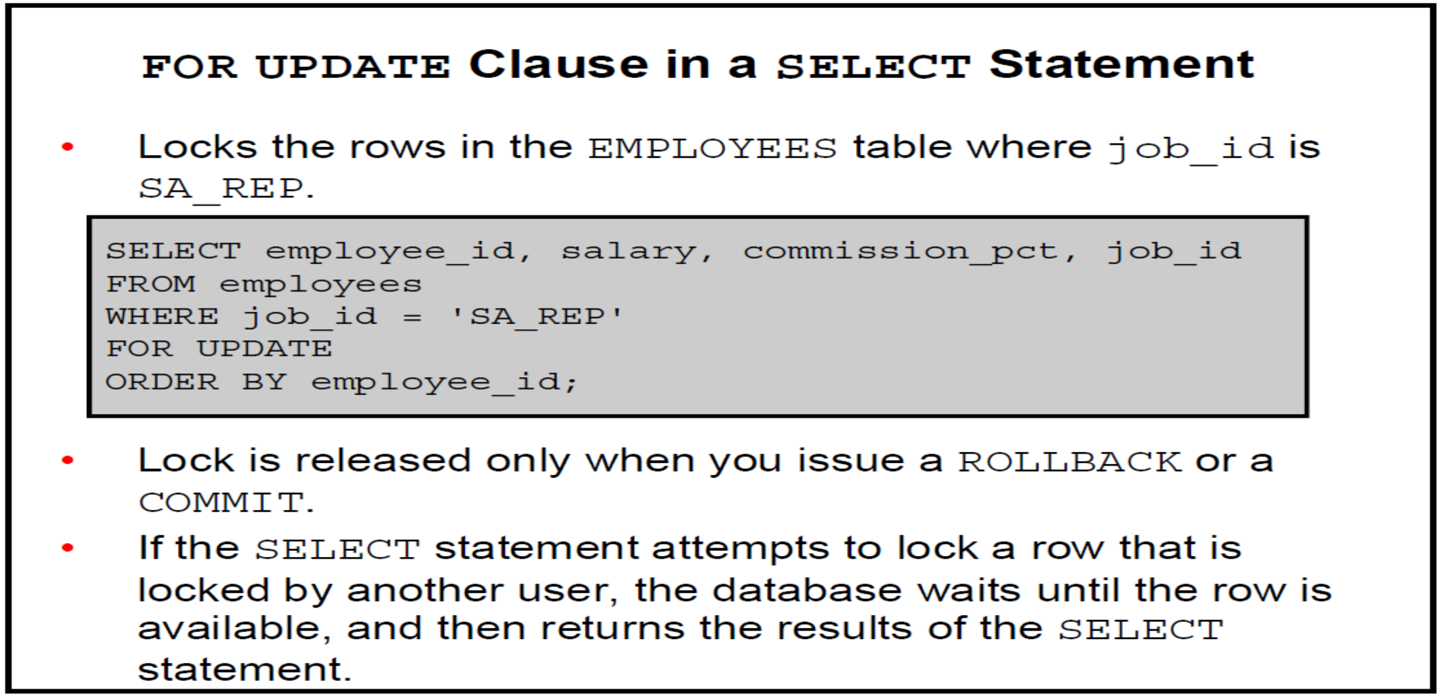
All readers, except the one who issued the change, see the database as it existed before the changes started; they view the undo segment’s “snapshot” of the data.

Before the changes are committed to the database, only the user who is modifying the data sees the database with the alterations. Everyone else sees the snapshot in the undo segment. This guarantees that readers of the data read consistent data that is not currently undergoing change.

When a DML statement is committed, the change made to the database becomes visible to anyone issuing a `SELECT` statement *after* the commit is done. The space occupied by the *old* data in the undo segment file is freed for reuse.

If the transaction is rolled back, the changes are undone:

- The original, older version of the data in the undo segment is written back to the table.
- All users see the database as it existed before the transaction began.



FOR UPDATE Clause in a SELECT Statement

- Locks the rows in the EMPLOYEES table where job_id is SA_REP.

```
SELECT employee_id, salary, commission_pct, job_id
FROM employees
WHERE job_id = 'SA_REP'
FOR UPDATE
ORDER BY employee_id;
```

- Lock is released only when you issue a ROLLBACK or a COMMIT.
- If the SELECT statement attempts to lock a row that is locked by another user, the database waits until the row is available, and then returns the results of the SELECT statement.

When you issue a SELECT...FOR UPDATE statement, the relational database management system (RDBMS) automatically obtains exclusive row-level locks on all the rows identified by the SELECT statement, thereby holding the records “for your changes only.” No one else will be able to change any of these records until you perform a ROLLBACK or a COMMIT.

Thank You