2.2.3. Queries with joins of relations

If data is collected from more than one relation or from different rows of the same relation within one query, these relations (in the second case they are the same relations) should be joined. Relations can be joined horizontally and vertically. In the case of horizontal connection, joined relations are specified after the FROM clause. In the case of vertical connection the relations are treated as sets of rows and joined using one of the operators on sets i.e. UNION, INTERSECT, MINUS.

Horizontal joining

In the case of a horizontal join, the result relation row is formed as a result of concatenation of the rows of joined relations (listed in the FROM clause after the comma or as part of the JOIN operator) meeting the so-called joining condition. The joining condition must include in its definition a reference to at least one attribute of each of the joined relations. If the JOIN operator is not used in the query (in the FROM clause relations are listed after the comma), then is implemented the cartesian product of the joined relations (with its possible selection of the rows according to the joining condition or according condition mentioned in the WHERE clause). If the JOIN operator is used, the joining condition follows after the ON clause of the operator (theta-join). The most commonly joined relations are relations related referentially (the joining condition is based on keys, i.e. on the primary and foreign key, of the related relations).

Task. *Find female cats who participated in incidents. Display, in addition, the names of the enemies involved in the incidents and descriptions of incidents.*

```
SELECT C.nickname "Female cat",enemy_name "her enemy",
       incident_desc "Incident description"
FROM Cats C,Incidents I
WHERE C.nickname=I.nickname AND gender='W';
```

```
Female cat her enemy       Incident description
---------- --------------- -----------------------------------
EAR        UNRULY DYZIO    HE THREW STONES
FAST       STUPID SOPHIA   SHE USED THE CAT AS A CLOTH
FLUFFY     SLIM            SHE THREW CONES
HEN        DUN             HE CHASED
LADY       KAZIO           HE WANTED TO SKIN OFF
LITTLE     SLYBOOTS        HE RECOMMENDED HIMSEF AS A HUSBAND
MISS       KAZIO           HE CAUGHT THE TAIL AND MADE A WIND
MISS       WILD BILL       HE BITCHED
 8 rows selected
```

The above query implements the join operation using the cartesian product of joined relations. The joined relations `Cats` and `Enemies` have aliases given, respectively as `C` and `I`, to uniquely identify attributes with the same names coming from both relations.

In accordance with the SQL standard, the solve of above task is equivalent to be solve (the SQL dialect proposed by Oracle) using the theta-join JOIN operator, in the ANSI SQL standard saved as INNER JOIN (internal join - the opposite of external joining; Oracle, due to the standard, accepts the INNER clause, although it is optional in his syntax), where the ON clause is followed by a join condition, to which the rows selection condition placed in the WHERE clause may also be transferred:

```
SELECT C.nickname "Female cat",enemy_name "her enemy",
       incident_desc "Incident description"
FROM Cats C JOIN Incidents I ON C.nickname=I.nickname
WHERE gender='W';
```

as well as using the USING clause replacing the ON clause (theta join is performed with = operator, due to one or more common, in terms of name and type, attributes; one of the common columns is omitted in the resulting relation):

```
SELECT nickname "Female cat",enemy_name "her enemy",
       incident_desc "Incident description"
FROM Cats JOIN Incidents USING(nickname)
WHERE gender='W';
```

or possibly using an operator realizing a natural join:

```
SELECT nickname "Female cat",enemy_name "her enemy",
       incident_desc "Incident description"
FROM Cats NATURAL JOIN Incidents
WHERE gender='W';
```

It should be noted here that the columns used in natural joining and in joining using the USING clause (columns due to which joining occurs) cannot be preceded by an alias of the relation from which they come from. In the above two queries in the SELECT clause, the entry `C.nickname` could not appear, where `C` is an alias of `Cats` relation. This does not apply to other attributes of the joined relations. This is due to the fact that, as already mentioned, one occurrence of a common attribute (here `nickname`) is eliminated here.

Task. *Find cats hunting on the site FIELD which have enemies with hostility degree above 5.*

```
SELECT DISTINCT C.nickname "Has enemy in the field"
FROM Cats C,Incidents I,Enemies E,Bands B
WHERE C.band_no=B.band_no AND C.nickname=I.nickname AND
      I.enemy_name=E.enemy_name AND
      site IN ('FIELD','WHOLE AREA') AND hostility_degree>5;

Has enemy in the field
----------------------
TIGER
MISS
TUBE
BOLEK
```

The above task is equivalent to be solved with the use of three operators performing theta-join with the possible transfer of selection conditions to the appropriate ON clauses (to those where the attributes involved in the selection are already available):

```
SELECT DISTINCT C.nickname "Has enemy in the field"
FROM Cats C JOIN Incidents I ON C.nickname=I.nickname
            JOIN Enemies E ON I.enemy_name=E.enemy_name
            JOIN Bands B ON  C.band_no=B.band_no
WHERE site IN ('FIELD','WHOLE AREA') AND hostility_degree>5;
```

as well as using the three times USING clause or e.g. using twice operator NATURAL JOIN and one time the USING clause:

```
SELECT nickname "Has enemy in the field", band_no
FROM Cats NATURAL JOIN Incidents
          NATURAL JOIN Enemies
          JOIN Bands USING(band_no)
WHERE site IN ('FIELD','WHOLE AREA') AND hostility_degree>5;
```

In the above solution, using the natural join operator to join with the `Bands` relation (instead of the USING clause used there) will result in an incorrect result due to the fact that the relations `Bands` and `Cats` have two common attributes (`name` and `band_no`). As part of such a natural join, in the background, an additional join condition would be used i.e. `Cats.name = Bands.name`.

Task. *In each of the bands, apart from his own, Tiger has placed a spy. He can be recognized by the fact that in cat hierarchy he reports directly to the Tiger and not the boss of the band although he is not a member of the Tiger's band. Find all the Tiger spies.*

```
SELECT C1.nickname "Spy",C1.band_no "Band"
FROM Cats C1 JOIN Cats C2 ON C1.chief=C2.nickname AND
                             C1.band_no<>C2.band_no
WHERE C1.chief='TIGER';
```

```
Spy             Band
--------------- ----------------------
ZOMBIES         3
BALD            2
REEF            4
```

The above task was solved with the use of theta-join connecting the `Cats` relation with itself.

Task. *Find the names of cats who joined the herd earlier than their immediate superiors.*

```
SELECT C1.name "In the herd before the boss"
FROM Cats C1,Cats C2
WHERE C1.chief=C2.nickname AND
      C1.in_herd_since<C2.in_herd_since;
```

```
In the herd before the boss
---------------------------
ZUZIA
```

The above task can be equivalently solved using e.g. theta-join:

```
SELECT C1.name "In the herd before the boss"
FROM Cats C1 JOIN Cats C2 ON C1.chief=C2.nickname AND
                            C1.in_herd_since<C2.in_herd_since;
```

In the above solution, you cannot use theta-half-joining because it is not implemented in Oracle (natural half-join is performed with HALF NATURAL JOIN operator).

Task. *Display the names of cats that have not yet participated in the incidents.*

```
SELECT name "No incident"
FROM Cats C LEFT JOIN Incidents I ON C.nickname=I.nickname
WHERE I.nickname IS NULL;

No incident
---------------
MICKA
PUCEK
LUCEK
```

To solve the above task, the LEFT JOIN operator was used. One can also use here the RIGHT JOIN operator, only changing the order of the combined relations. Equivalent operators for LEFT JOIN and RIGHT JOIN are, according to the ANSI standard, operators respectively LEFT OUTER JOIN and RIGHT OUTER JOIN. Oracle allows the word OUTER to be omitted as insignificant. Left and right natural joining works the same as internal joining so it cannot be used here.

The above task can be equivalently solved by using the Oracle operator (+) placed next to one of the attributes of the joining condition. It occurs then, the left-side join of the relation from which the attribute without the (+) sign comes from, with the relation from which the attribute with the sign comes from is performed. Since the Oracle 9i version, it is recommended to use the external join operators (LEFT ..., RIGHT ..., FULL ...) instead of the (+) operator because of their compatibility with the ANSI SQL standard. The solution of the above task, using the (+) operator, is as follows:

```
SELECT name "No incident"
FROM Cats C, Incidents I
WHERE C.nickname=I.nickname(+) AND I.nickname IS NULL;
```

The (+) operator implements here the left-side join of the `Cats` relation with the `Enemies` relation.

Task. *Display a report that returns information about male cats subordinates and superiors. If the cat does not have a subordinate, this should be indicated in the report. Similarly, the report should indicate the absence of a superior.*

```
SELECT NVL(C1.nickname,'No superior') "Superior",
       NVL(C2.nickname,'No subordinate') "Subordinate"
FROM Cats C1 FULL JOIN Cats C2 ON C1.nickname=C2.chief
WHERE DECODE(C1.nickname,NULL,'M',C1.gender)='M' AND
      DECODE(C2.nickname,NULL,'M',C2.gender)='M'
ORDER BY 1;
```

```
Superior        Subordinate
--------------- ---------------
BALD            CAKE
BALD            TUBE
BOLEK           No subordinate
CAKE            No subordinate
MAN             No subordinate
No superior     TIGER
REEF            MAN
REEF            SMALL
SMALL           No subordinate
TIGER           BALD
TIGER           ZOMBIES
TIGER           BOLEK
TIGER           REEF
TUBE            No subordinate
ZERO            No subordinate

 15 rows selected
```

In the solution of the above task, the FULL JOIN (full external join) operator was used (equivalently can be used here, compatible with the ANSI standard, the FULL OUTER JOIN operator). The NVL function returns the value of the first argument if it is non-NULL, or the value of the second argument if the first argument is NULL.

Vertical joining

In the case of vertical joining, the joined relations are treated as sets of rows and the resulting relation is the result of a set operation on these sets. To perform the vertical join, the joining relations must have the same number of attributes and their types must be the same, respectively (relations must have the same scheme). The following set operators are used for vertical joining in Oracle:

UNION - sum without repetitions,
UNION ALL - sum with repetitions,
INTERSECT - product,
MINUS - the difference.

The attribute names of the resulting relation always come from the first joined relation. In query with vertical join, ORDER BY clause may appear only once, as its last. Ordering operation can only be done according to the expression numbers that appear in the SELECT clause of the joined relations.

Task. Specify the functions of cats in bands 1 and 2.

```
SELECT function FROM Cats WHERE band_no=1
UNION
SELECT function FROM Cats WHERE band_no=2;

FUNCTION
----------
BOSS
CATCHER
CATCHING
DIVISIVE
NICE
THUG

 6 rows selected
```

The use of the UNION operator instead of UNION ALL allows you to remove duplicate rows in the resulting relation (in this case, the functions performed in both band 1 and 2).

2.2.4. Queries with subqueries

As mentioned earlier, some SELECT query clauses may contain nested SELECT clauses, i.e. subqueries. In Oracle SQL, this applies to the SELECT, FROM, WHERE and HAVING clauses. For the SELECT clause the subquery must return only one value (the resulting relation of the subquery must consist of one row and one attribute). Depending on whether the subquery refers to an external query (through the attribute from the relation from the external query preceded by an alias of this relation) or not, the subqueries are correlated (bound) and uncorrelated (unbound). A correlated subquery is performed for each row of the external query, an uncorrelated subquery only once at the beginning of the external query action. The subquery cannot contain the ORDER BY clause (except for the subquery in the FROM clause). A correlated subquery cannot appear in the FROM clause.

The following are examples of using subqueries (the first applies to uncorrelated subqueries and later also correlated subqueries).

Task. *Find cats that perform the same function as a cat with nickname LOLA.*

```
SELECT name "LOLA's deputy",band_no "its band"
FROM Cats
WHERE function=(SELECT function
                FROM Cats
                WHERE nickname='LOLA')
      AND nickname!='LOLA';

LOLA's deputy   its band
--------------- ----------------------
SONIA           3
RUDA            1
BELA            2
```

The above task can be solved in an equivalent way by joining the relation `Cats` with the relation `Cats`:

```
SELECT C1.name "LOLA's deputy",C1.band_no "its band"
FROM Cats C1,Cats C2
WHERE C1.function=C2.function AND C2.nickname='LOLA'
      AND C1.nickname!='LOLA';
```

An equivalent solution to the above task can also be a solution using the theta-join operator.

Task. *Find cats for whose mice ration is greater than the average ration throughout the herd.*

```
SELECT nickname "Nickname",mice_ration "Eats"
FROM Cats
WHERE mice_ration>(SELECT AVG(NVL(mice_ration,0))
                   FROM Cats);

Nickname        Eats
--------------- ----------------------
CAKE            67
TUBE            56
TIGER           103
ZOMBIES         75
BALD            72
FAST            65
REEF            65
HEN             61
 8 rows selected
```

The above task can be solved in an equivalent way by joining the relation `Cats` with the relation, determined by the subquery placed in the FROM clause, that return one number determining the average value of mice consumption in the herd:

```
SELECT nickname "Nickname",mice_ration "Eats"
FROM Cats,(SELECT AVG(NVL(mice_ration,0)) av
           FROM Cats)
WHERE mice_ration>av;
```

or using theta-join operation of the relation `Cats` with the relation returned by the subquery:

```
SELECT nickname "Nickname",mice_ration "Eats"
FROM Cats JOIN (SELECT AVG(NVL(mice_ration,0)) av
                FROM Cats) ON mice_ration>av;
```

Task. *Find cats whose ration of mice belongs to the list of smallest rations from each bands.*

```
SELECT name "Name",NVL(mice_ration,0) "Eats",band_no "Band"
FROM Cats
WHERE NVL(mice_ration,0) IN (SELECT MIN(NVL(mice_ration,0))
                             FROM Cats
                             GROUP BY band_no)
ORDER BY 3,2;

Name            Eats                   Band
--------------- ---------------------- ----------------------
RUDA            22                     1
BELA            24                     2
SONIA           20                     3
LATKA           40                     4
DUDEK           40                     4
```

Task solutions that use joins of the relations are analogous to the solution for a previous task. The operator IN was used in the above solution. When using a subquery with the NOT IN operator in a query, Oracle recommends replacing this operator with an external join operator or NOT EXISTS operator (the latter is defined later in the lecture). In both this cases, unlike the NOT IN operator, the system can use indexes.

Task. *Find the cats with the lowest mice ration in their bands.*

```
SELECT name "Name",mice_ration "Eats",band_no "Band"
FROM Cats
WHERE (NVL(mice_ration,0),band_no) IN
                       (SELECT MIN(NVL(mice_ration,0)),band_no
                        FROM Cats
                        GROUP BY band_no)
ORDER BY band_no;

Name            Eats                   Band
--------------- ---------------------- ----------------------
RUDA            22                     1
BELA            24                     2
SONIA           20                     3
LATKA           40                     4
DUDEK           40                     4
```

An equivalent solution that uses relations joining is as follows:

```
SELECT name "Name",mice_ration "Eats",band_no "Band"
FROM Cats,(SELECT MIN(NVL(mice_ration,0)) mi,band_no nb
           FROM Cats
           GROUP BY band_no)
WHERE band_no=nb AND NVL(mice_ration,0)=mi
ORDER BY band_no;
```

The use of the theta-join of the `Cats` relation with the relation resulting from the subquery for the above task gives the following code:

```
SELECT name "Name",mice_ration "Eats",band_no "Band"
FROM Cats JOIN (SELECT MIN(NVL(mice_ration,0)) mi,band_no nb
                FROM Cats
                GROUP BY band_no) ON band_no=nb AND
                                     NVL(mice_ration,0)=mi
ORDER BY band_no;
```

Another equivalent solution using, this time, correlated subquery is as follows:

```
SELECT name "Name",mice_ration "Eats",band_no "Band"
FROM Cats C
WHERE NVL(mice_ration,0)= (SELECT MIN(NVL(mice_ration,0))
                           FROM Cats
                           WHERE band_no=C.band_no)
ORDER BY band_no;
```

The correlation in the above query occurs in the condition after WHERE clause of the subquery, through the attribute band_no with an alias (the attribute without an alias comes from the relation `Cats` open in the subquery, the attribute with the alias comes from the relation `Cats`, open by an external query). The subquery is performed for each row of external query (for each cat, among the cats belonging to its band, a minimum ration of mice is found).

One can use the ANY operator or the ALL operator for subqueries placed in the WHERE clause that return only one column. These operators always occur with relation operators (=, <,>, <=,> =). For example, expressions:

a) X> ANY subquery,
b) X <ALL subquery,

will be TRUE if X is greater than at least one value returned by the subquery (expression a)) or if X is less than each value returned by the subquery (expression b)). Similarly, one can use the ANY and ALL operators for other relationship operators.

Task. *Find cats whose ration of mice is greater than the lowest mice ration in band 4. Use the ANY operator to solve the task.*

```
SELECT name "Name", NVL(mice_ration,0) "Eats",
       band_no "Band"
FROM Cats
WHERE mice_ration>ANY(SELECT DISTINCT NVL(mice_ration,0)
                      FROM Cats
                      WHERE Band_no=4)
ORDER BY "Eats" DESC;

Name            Eats                   Band
--------------- ---------------------- ----------------------
MRUCZEK         103                    1
KOREK           75                     3
BOLEK           72                     2
JACEK           67                     2
PUCEK           65                     4
ZUZIA           65                     2
PUNIA           61                     3
BARI            56                     2
KSAWERY         51                     4
MELA            51                     4
CHYTRY          50                     1
LUCEK           43                     3

 12 rows selected
```

As mentioned before, subqueries can also be placed in the HAVING clause.

Task. *Find bands in which the average mice ration is higher than the average ration in band 3.*

```
SELECT band_no "Band better than a band 3",
       AVG(NVL(mice_ration,0)) "Average ration in band"
FROM Cats
HAVING AVG(NVL(mice_ration,0))>(SELECT AVG(NVL(mice_ration,0))
                                FROM Cats
                                WHERE band_no=3)
GROUP BY band_no;

Band better than a band 3 Average ration in band
------------------------- ----------------------
1                         50
2                         56,8
```

Subqueries can be nested in and combined by logical operators (the latter in the WHERE and HAVING clauses).

Task. *Find cats whose ration of mice is higher than the highest ration in the PINTO HUNTERS band.*

```
SELECT name "Better than any of PINTO",mice_ration "Eats"
FROM Cats
WHERE mice_ration>(SELECT MAX(NVL(mice_ration,0))
                   FROM Cats
                   WHERE band_no=(SELECT band_no
                                  FROM Bands
                                  WHERE name='PINTO HUNTERS'))
ORDER BY mice_ration DESC;

Better than any of PINTO Eats
------------------------ ----------------------
MRUCZEK                  103
KOREK                    75
BOLEK                    72
JACEK                    67
```

Task. *Find cats from the band WHITE HUNTERS and PINTO HUNTERS, whose ration of mice is higher than the average ration of the whole herd.*

```
SELECT nickname "WHITE and PINTO above!"
FROM Cats
WHERE Band_no IN (SELECT band_no
                  FROM Bands
                  WHERE name IN
                            ('WHITE HUNTERS','PINTO HUNTERS'))
      AND mice_ration>(SELECT AVG(NVL(mice_ration,0))
                       FROM Cats);

WHITE and PINTO above!
----------------------
ZOMBIES
REEF
HEN
```

A subquery can also be placed in a SELECT clause, provided that it returns a relation with one row and one column (one value).

Task. *For each male cat, find the average ration of mice released monthly to the cat in his band.*

```
SELECT nickname "Cat",(SELECT AVG(NVL(mice_ration,0))
                       FROM Cats
                       WHERE band_no=C.band_no)
                                         "Average in the band"
FROM Cats C
WHERE gender='M';

Cat             Average in the band
--------------- ----------------------
CAKE            56,8
TUBE            56,8
ZERO            49,75
SMALL           49,4
TIGER           50
BOLEK           50
ZOMBIES         49,75
BALD            56,8
REEF            49,4
MAN             49,4

 10 rows selected
```

The subquery used in the above solution is correlated.

For correlated subqueries, the operator EXISTS (or NOT EXISTS) is often used. This operator is used to check if the subquery returns any row. If so, the TRUE is returned, if not, FALSE is returned.

Task. *The whole leadership elite of the cats herd came to the conclusion that the potential threat to their power are cats, which do not have subordinates, but are sometimes feisty (have enemies) and in addition their ration of mice is at least equal to the value* `min_mice+(max_mice-min_mice)/3` *(they had to stand out in the past), where* `min_mice` *and* `max_mice` *is determined by their function. Find these cats.*

```
SELECT nickname "For crawling",B.name "Band name"
FROM Cats C JOIN Bands B USING(band_no)
WHERE NOT EXISTS (SELECT nickname
                  FROM Cats
                  WHERE chief=C.nickname)
INTERSECT
SELECT nickname,B.name
FROM Cats JOIN Bands B USING (band_no)
          JOIN Functions USING(function)
WHERE mice_ration>NVL(min_mice,0)+
                 (NVL(max_mice,0)-NVL(min_mice,0))/3
INTERSECT
SELECT nickname,B.name
FROM Cats C JOIN Bands B USING(band_no)
WHERE EXISTS (SELECT nickname
              FROM Incidents
              WHERE nickname=C.nickname)
ORDER BY 2,1;

For crawling    Band name
--------------- --------------------
CAKE            BLACK KNIGHTS
FAST            BLACK KNIGHTS
MISS            BLACK KNIGHTS
TUBE            BLACK KNIGHTS
BOLEK           SUPERIORS
```

The first of the vertically joined relations (returned by query) contains data on cats that do not have subordinates, the second data on cats, which are characterized by the ration of mice much higher than the lower limit of the mice that they can get according to their function and the third data on cats having enemies. The product (common part) of these relations gives resolution of the task.

2.2.5. Pivot tables

The pivot table mechanism allows to reposition columns in the resulting SELECT query in place of rows and vice versa. Since the Oracle 11g version, this mechanism has been implemented using the PIVOT clause that aggregates the result and presents it in rows instead of columns (rotation). The use of aggregation (group function) indicates that PIVOT is a type of group operation. This tool is very well suited for analytical purposes, allowing convenient viewing of e.g. trends. In the SELECT query, the PIVOT clause occurs in the FROM clause after the data source name (relation explicitly specified or in the form of a subquery or resulting from a join), which is the relation to be rotated. This is done in accordance with the syntax:

```
SELECT …
FROM data_source
         PIVOT
         (
          aggregate_function
          FOR column_to_rotation
          IN range_of_data_to_rotation
        )
 WHERE...
```

The FOR clause defines from data of which column in the data source the columns (attributes) will be generated in the relation after rotation are to be derived. This clause therefore defines the rotated column of the data source. Its values become the header of the relation after rotation. The IN clause determines, in the form of a set (elements after the decimal point), which columns of the relation after rotation are to be available in the result relation. According to the attributes selected by the SELECT clause but bypassing these listed in the FOR clause as well as these not being an argument of the aggregate function (this does not apply to the COUNT(*) function, unless, as a function argument, will appear expression instead of *), grouping takes place. An aggregation function is performed for groups created in this way.

Task. *Display functions and total rations of mice for cats from the bands WHITE HUNTER and BLACK KNIGHTS. As a result, do not include the SZEFUNIO function.*

```
SELECT function, B.name "Band name",
       NVL(mice_ration,0)+NVL(mice_extra,0)
           "Total mice ration"
FROM Cats JOIN Bands B USING(band_no)
WHERE B.name IN ('BLACK KNIGHTS','WHITE HUNTERS') AND
      function != 'BOSS';

FUNCTION   Band name            Total mice ration
---------- -------------------- ----------------------
THUG       BLACK KNIGHTS        93
CATCHING   BLACK KNIGHTS        65
CATCHING   BLACK KNIGHTS        67
CATCHER    BLACK KNIGHTS        56
NICE       BLACK KNIGHTS        52
NICE       WHITE HUNTERS        55
CAT        WHITE HUNTERS        43
THUG       WHITE HUNTERS        88
CATCHING   WHITE HUNTERS        61

 9 rows selected
```

The solution to the above task is obvious. The task can be modified for the following problem:

Task. *Display the total rations of mice for bands WHITE HUNTERS and BLACK KNIGHTS, divided into functions performed by cats. Skip in the list the BOSS function.*

```
SELECT function, B.name "Band name",
       SUM(NVL(mice_ration,0)+NVL(mice_extra,0))
               "Total ration for the band"
FROM Cats JOIN Bands B USING(band_no)
WHERE B.name IN ('BLACK KNIGHTS','WHITE HUNTERS')
      AND function != 'BOSS'
GROUP BY function,B.name;
```

```
FUNCTION   Band name            Total ration for the band
---------- -------------------- -------------------------
THUG       WHITE HUNTERS        88
CATCHER    BLACK KNIGHTS        56
CATCHING   WHITE HUNTERS        61
THUG       BLACK KNIGHTS        93
NICE       WHITE HUNTERS        55
CATCHING   BLACK KNIGHTS        132
NICE       BLACK KNIGHTS        52
CAT        WHITE HUNTERS        43
 8 rows selected
```

The above result would be much more readable if presented in the form of an array in which the band names would be the names of subsequent columns, within which the sum of mice allocations for selected functions would be specified. This is a task in which the PIVOT clause can be used. The source of data would be defined by the relation defined by the subquery returning the function of the cat, its total mice ration and the name of the band to which it belongs. The aggregate function in PIVOT command would be SUM on total rations of mice. The column for rotation would be the name of the band and the range of rotated data would be two values of the names of the band i.e. WHITE HUNTERS and BLACK KNIGHTS. The such described command is presented below.

```
SELECT *
FROM (SELECT function, B.name band_name,
      NVL(mice_ration,0)+NVL(mice_extra,0) mice_total
      FROM Cats JOIN Bands B USING(band_no))
      PIVOT
      (
       SUM(mice_total)
       FOR band_name
       IN ('BLACK KNIGHTS' "Band BLACK KNIGHTS",
           'WHITE HUNTERS' "Band WHITE HUNTERS")
            )
       WHERE function != 'BOSS'
       ORDER BY "Band BLACK KNIGHTS" DESC;
```

```
FUNCTION   Band BLACK KNIGHTS     Band WHITE HUNTERS
---------- ---------------------- ----------------------
CAT                               43
DIVISIVE
CATCHING   132                    61
THUG       93                     88
CATCHER    56
NICE       52                     55
 6 rows selected
```

The band names listed within the scope of the data of the PIVOT clause (IN operator) have their aliases defined. Except band names (whose values are to be column headers) and the total mice ration (on which the aggregation function is applied), the data source returns only the `function` column. Grouping occurs by this column. If `COUNT(*)` were used instead of `SUM(mouse_total)`, the grouping would also be done by `mice_total`. If the data source, except the function, the name of the band, and the total mice ration, would return other attributes, the grouping would be done by function and these attributes. For example, let gender will be additionally selected. Grouping takes place, as the result below shows, due to the function and the gender.

```
SELECT *
FROM (SELECT function, B.name band_name,gender,
      NVL(mice_ration,0)+NVL(mice_extra,0) mice_total
      FROM Cats JOIN Bands B USING(band_no))
      PIVOT
      (
       SUM(mice_total)
       FOR band_name
       IN ('BLACK KNIGHTS' "Band BLACK KNIGHTS",
           'WHITE HUNTERS' "Band WHITE HUNTERS")
            )
       WHERE function != 'BOSS'
       ORDER BY "Band BLACK KNIGHTS" DESC;

FUNCTION   GENDER  Band BLACK KNIGHTS     Band WHITE HUNTERS
---------- ------ ---------------------- ---------------------
CAT        W
DIVISIVE   M
CAT        M                              43
CATCHER    W
THUG       M      93                      88
CATCHING   M      67
CATCHING   W      65                      61
CATCHER    M      56
NICE       W      52                      55

 9 rows selected
```

2.2.6. WITH clause

Placed before the SELECT query, the WITH clause lets to name relations effecting from subqueries in the FROM clause of that query. By optimizing the query, Oracle implements such a subquery in the form of a materialized view or temporary table. The names specified in the WITH clause can be used in a query, which makes it more readable. A fragment of the SELECT query syntax with the WITH clause has the form:

WITH {SubqueryName AS (subquery) [, …]}
SELECT [**DISTINCT** | **ALL**] { expression [alias] [, ...]} | *
FROM {RelationViewNameSuqueryName [alias]
    [join_operator
    RelationViewNameSuqueryName [alias]
    [**ON** join_condition]] [, ...]}
Rest_of_SELECT_command

The WITH clause can also be applied in a subquery and be used in its FROM clause. The names defined in it are visible in the query related to it and in all its subqueries. These names can also be used as part of the FROM clause to define other names placed in the WITH clause.

Task. *Find cats whose ration of mice is greater than the average ration of the all herd.*

```
WITH Av AS
    (SELECT AVG(NVL(mice_ration,0)) avgmr
     FROM Cats)
SELECT nickname "Nickname",NVL(mice_ration,0) "Eats"
FROM Cats JOIN Av ON mice_ration>avgmr;

Nickname        Eats
--------------- ----------------------
CAKE            67
TUBE            56
TIGER           103
ZOMBIES         75
BALD            72
FAST            65
REEF            65
HEN             61
 8 rows selected
```

Task. *Find female cats that have participated in incidents with enemies with hostility degree above 5.*

```
WITH Ladies AS
     (SELECT nickname
      FROM Cats
      WHERE gender='W'),
     Incidents5 AS
     (SELECT nickname
      FROM Incidents NATURAL JOIN Enemies
      WHERE hostility_degree>5)
SELECT DISTINCT nickname "Feisty female cats"
FROM Ladies NATURAL JOIN Incidents5;

Feisty female cats
------------------
EAR
MISS
LADY
```

or alternatively

```
WITH Ladies AS
     (SELECT nickname
      FROM Cats
      WHERE gender='W'),
     FeistyFemaleCats AS
     (SELECT DISTINCT nickname
      FROM Incidents NATURAL JOIN Enemies
                     NATURAL JOIN Ladies
      WHERE hostility_degree>5)
SELECT nickname "Feisty Female Cats"
FROM FeistyFemaleCats;
```

2.3. Data modification in relation

The DML (Data Manipulation Language) component of SQL is used to specify commands that modify data in a relation. Within this component one can primarily distinguished INSERT command that adds new rows to the relation, the UPDATE command that modifies existing data in the relation and the DELETE command which removes the rows from the relation.

2.3.1. INSERT command

The INSERT command is used to insert one or more rows directly or indirectly into an existing relation. The latter case occurs when insertion takes place through a simple view, also known as a modifiable perspective (both concepts will be presented later in the lecture). The syntax for the INSERT command is as follows:

INSERT INTO RelationViewName [({attribute [, …]})]
{**VALUES** ({value [, …]})} | subquery

The list of attributes specified after comma specifies the names of the attributes whose values will be filled in. All attributes not listed must be optional (NULL) or have a default value defined (specified in the CREATE TABLE command - DDL component of SQL language). The lack of a list of attributes in the command indicates that all attributes of the relation will be filled in the order of their definition in the CREATE TABLE command. Data can be specified explicitly in the VALUES clause through a list of values specified after comma or implicitly through the subquery. In the first case into the relation one row is inserted, in the second case, as many rows as the rows return the subquery. The number of explicitly entered values as well as the number of values returned by the subquery must be equal to the number of attributes specified in the attribute list (if any) and the types of these values must match the types of the respective attributes.

There is also a version of the INSERT command that allows you to insert multiple rows as part of one such command. The short version of this command has the following syntax:

INSERT ALL {INTO RelationViewName [({attribute [, …]})]
VALUES ({value [, …]}) [...]}
{SELECT * FROM Dual }| subquery

The above version of the INSERT command reduces the time to load data into the database (only one connection to the database) and can be used to batch rewrite data from one database to another, when it is certain that the source data is correct. In this version, it is also possible to enter rows to many different relations with one command. The subquery returning rows to insert can also be data source. In this case, the values in the VALUES clause will be the names of the expressions (their aliases) or the names of the attributes returned by the subquery.

Task. *Tiger decided to punish the insubordination of his subordinates by sending them temporarily to cellars belonging to the inhabitants of the village of Wólka Mała. Define in* `Bands` *relation a new band called Exiles, whose hunting place will be cellars.*

```
INSERT INTO Bands (band_no,name,site,band_chief)
VALUES(6,'EXILES','CELLARS',NULL);

1 rows inserted.

ROLLBACK;

rollback complete.
```

Attribute values after the VALUES clause are listed in the order in which they are defined in the CREATE TABLE command, so one can omit the attribute names and write down the above command as follows:

```
INSERT INTO Bands
VALUES(6,'EXILES','CELLARS',NULL);

1 rows inserted.

ROLLBACK;

rollback complete.
```

Task. *Several cats, previously non-attached, decided to join the herd. To facilitate their admission, the relation called `New` with the attributes `nickname`, `name`, `sex`, where data of new cats were aggregate, was prepared. Using the `New` relation, add to the herd new members.*

```
INSERT INTO Cats
(nickname,name,gender,in_herd_since,chief)
 SELECT nickname,name,sex,SYSDATE,'TIGER'
 FROM New;
```

```
5 rows inserted.
```

```
ROLLBACK;
```

```
rollback complete.
```

Attributes of the `Cats` relation that do not get value under the above command are optional.

2.3.2. UPDATE command

The UPDATE command is used to change the value of selected attributes in selected rows directly in the indicated relation or indirectly through a simple view otherwise called as a modifiable view (the last concept will be presented later in the lecture). The command syntax is as follows:

UPDATE RelationViewName [alias]
SET {attribute_name = expression [, ...]}
[**WHERE** condition]

After the UPDATE clause, the name of the modified relation (or view, which is modifiable) is determined. Alias means an alternative (replacement) name. The values of the attributes listed after the SET clause are modified. These attributes get the new values specified by the expression (constant is its special case). As part of the expression defining the new attribute value, a subquery may appear. The condition after the WHERE clause determines which rows are modified. This condition can also be specified by a subquery. The absence of the WHERE clause in the UPDATE command causes modification of the values of selected attributes in all rows of the relation.

Task. *Promote a female cat named LATKA to the CATCHER function, giving her a 30% increase of ration of mice.*

```
UPDATE Cats
SET funkcja='CATCHER',
    mice_ration=ROUND(NVL(mice_ration,0)*1.3,0)
WHERE name='LATKA';

1 rows modified.

ROLLBACK;

rollback complete.
```

Task. *Tiger, as an enlightened ruler, decided that he would not manage additional mice rations under the influence of current emotions. Instead, he decided to make monthly payments based on the "merits" remembered during a month in the* `Extra_additions` *relation. Modify the value of additional rations of mice on the base of "merits" of cats remembered in the* `Extra_additions` *relation..*

Sample content of the `Extra_additions` relation:

```
SELECT nickname "Nickname", extra_add "Extra addition"
FROM Extra_additions;

Nickname  Extra addition
--------- --------------
TIGER                 10
LOLA                   5
BOLEK                 10
TIIGER                 5
LOLA                  -2


UPDATE Cats C
SET mice_extra=(SELECT NVL(C.mice_extra,0)+SUM(extra_add)
                FROM Extra_additions E
                WHERE E.nickname =C.nickname)
WHERE nickname IN (SELECT nickname
                   FROM Extra_additions);


3 rows modified.

ROLLBACK;

rollback complete.
```

The above command uses a correlated subquery to modify the value of additional rations and an uncorrelated subquery to identify the cats whose data are to be modified.

During modification a relation, one can set the attribute to NULL value. This is the only case when the = operator is used instead of the IS NULL operator in handling NULL value.

Task. *As part of the fight against the crisis, Tiger ordered a temporary suspension of issuing additional rations of mice. Accomplish this task by appropriately modifying the* `mice_extra` *attribute in the* `Cats` *relation.*

```
UPDATE Cats
SET mice_extra=NULL
WHERE nickname<>'TIGER';

17 rows modified.

ROLLBACK;

rollback complete.
```

2.3.3. DELETE command

The DELETE command is used to remove selected rows directly from the indicated relation or indirectly through a simple view otherwise known as a modifiable view (the last concept will be presented later in the lecture). The command syntax is as follows:

DELETE FROM RelationViewName
[**WHERE** condition]

After DELETE FROM clause the name of the relation to modification is specified (or modifiable view). The condition after the WHERE clause determines which rows will be removed. If this clause does not occur, the DELETE command deletes all rows of the relation.

Task. *As it turns out, however, being an enlightened ruler also has its limits. Present the highly understandable decision of the Tiger on removing his data from the* `Extra_additions` *relation when news arrived about of the visit of the Cat's Control Chamber.*

```
DELETE FROM Extra_additions
WHERE nickname='TIGER';

2 rows was deleted.
```

2.4. Views

Views are another database objects which are created and deleted using component DDL of SQL language.

View - a selection of data and expressions values constructed on the basis of data taken from one or several relations or other views

Views are visible by the user as tables and in the database they are remembered in the form of their definitions (this does not apply to so-called materialized views). Every reference to view creates its structure.

Views are defined for the following reasons:

- to limit access to some data in a relation,
- to help the user retrieve the results of complex queries using simple queries based on views,
- to free the user from analyzing the data structure,
- to provide information that is seen differently by different users.

The choice and definition of view is part of the physical design of the database.

Views can be divided into **simple** and **complex**. Simple views are those in which the SELECT command has the following characteristics:

- do not contains joins,
- do not contains group or analytical functions, ordering, DISTINCT qualifier, GROUP BY clause,
- do not contains correlated subqueries or subqueries in the SELECT clause,
- do not contains CONNECT BY and START WITH clauses.

All other views are called complex. The essence of a simple view is that through it (as opposed to a complex view) one can perform DML operations on the relation on which the view is built. There are, however, exceptions to the prohibition of modification through complex views built using joins, hence the concept of a **modifiable** view (different from a complex view), i.e. one through which can be performed DML operations on relations, from which view is built. Modifiable view is therefore view, which:

- do not contains joins, with the exception of joins for which joined relations keep key *(key-preserved tables)* in view. If the view contains a join for which the linked relations keep the key, then the DML operation on the view must apply to only one relation. Additionally:
    - to perform the INSERT operation, the view must select primary key attributes and all mandatory attributes of the key-preserved relation,
    - to perform the UPDATE operation, all modified attributes must come from a key-preserved relation,
    - for the DELETE operation, the join operation can only apply to one key-preserved relation.
- do not contains group or analytical functions, ordering, DISTINCT qualifier, GROUP BY clause,
- do not contains correlated subqueries and subqueries in the SELECT clause
- do not contains CONNECT BY and START WITH clauses,
- do not contains expressions or pseudo-columns (INSERT and UPDATE commands based on the view cannot apply to them).

The relation keep key *(the key-preserved table)* in view if each unique attribute of the relation is also unique within the result of the view.

The basic syntax for the command defining the view is as follows:

CREATE VIEW View_name [({view_attribute [, ...]})]
AS SELECT_command
[**WITH CHECK OPTION** [**CONSTRAINT** constraint_name]] |
[**WITH READ ONLY**]

where the view attributes are the new names of the expressions (attributes) specified in the SELECT clause of the SELECT command (this list does not have to occur if all expressions are attributes of the relation/view; then the view will have attributes with names that are the same as the corresponding them attributes of the relation/perspective from which data are collected), the optional WITH CHECK OPTION clause (only for simple views) allows updating through the view only then, if changed or new records will still appear in the view (they will meet the condition after the WHERE clause of the SELECT command). After the WITH CHECK OPTION clause, one can additionally place the CONSTRAINT constraint_name clause to name the constraint one are creating. The optional WITH READ ONLY clause (for simple views only) prevents data updates through the view.

Task. *Define a simple view providing part of the data (nickname, date of entry to the herd, ration of mice) of cats that belong to band No. 3.*

```
CREATE VIEW Band3
AS SELECT nickname,in_herd_since,mice_ration
   FROM Cats
   WHERE band_no=3;

view BAND3 created.
```

The defined view is a simple view so it is possible through it modify data in the `Cats` relation.

```
UPDATE Band3
SET mice_ration=55
WHERE nickname='ZERO';

1 rows updated.
```

The fact that the data has actually been modified is shown in the following query.

```
SELECT nickname,mice_ration
FROM Cats
WHERE nickname='ZERO';

NICKNAME        MICE_RATION
--------------- ----------------------
ZERO            55

ROLLBACK;

rollback complete.
```

The ROLLBACK is an element of DCL component of the SQL language and is used to roll back of performing transaction (in the above case, the transaction consisted of one UPDATE command).

Task. *Define a complex view which provide the names of the bands and the minimum, maximum and average mice ration in each band.*

```
CREATE VIEW Mice_in_bands (name,minm,maxm,average)
AS SELECT B.name,MIN(mice_ration),MAX(mice_ration),
          AVG(mice_ration)
   FROM Cats JOIN Bands B USING(band_no)
   GROUP BY B.name;

view MICE_IN_BANDS created.

SELECT * FROM Mice_in_bands;

NAME                 MINM           MAXM           AVERAGE
-----------------    -------------- -------------- --------------
BLACK KNIGHTS        24             72             56,8
WHITE HUNTERS        20             75             49,75
SUPERIORS            22             1              50
PINTO HUNTERS        40             65             49,4
```

Simple views with the WITH CHECK OPTION clause can be a tool for introducing additional constraints on data.

Task. *Define a view that provides part of the data of cats from band No. 4, that enables DML operations through the view only for this band.*

```
CREATE VIEW Band4
AS SELECT nickname,name,function,mice_ration,band_no
   FROM Cats WHERE band_no=4
   WITH CHECK OPTION;

view BAND4 created.
```

An example of an impossible operation:

```
INSERT INTO Band4 VALUES ('LALA','KOBOL','CAT',30,3);

Error starting at line 1 in command:
INSERT INTO Band4 VALUES ('LALA','KOBOL','CAT',30,3)
Error report:
SQL Error: ORA-01402: naruszenie klauzuli WHERE dla
perspektywy z WITH CHECK OPTION
01402. 00000 - "view WITH CHECK OPTION where-clause
violation"
*Cause:
*Action:
```

Task. *After a long reflection, the Tiger came to the conclusion that the situation in the herd is as perfect as he is perfect. Therefore, he ordered a IT specialist to define a "guarding" view so that the existing status quo could not be violated. The view ought to ensure that:*

- *no new band could be created,*
- *no cat outside the chiefs' elite has usurped the right to be a chief,*
- *cats could only perform existing functions,*
- *the ration of the mice was at least equal to the value of cats social minimum level (6 mice) and that could not exceed the ration of the mice of Tiger.*

Define a view that meets these requirements.

```
CREATE OR REPLACE VIEW Status_quo_of_cats
AS SELECT nickname,name,chief,function,mice_ration,band_no
   FROM Cats
   WHERE (band_no IN (SELECT band_no FROM Cats)
                    OR band_no=5 OR band_no IS NULL)
          AND (chief IN (SELECT chief FROM Cats)
                OR chief IS NULL)
          AND (function IN (SELECT function FROM Cats)
              OR function='HONORARY' OR function IS NULL)
          AND (mice_ration BETWEEN 6
                           AND(SELECT mice_ration
                               FROM Cats
                               WHERE nickname='TIGER')
              OR mice_ration IS NULL)
          AND nickname<>'TIGER'
   WITH CHECK OPTION CONSTRAINT nothing_more;

view STATUS_QUO_OF_CATS created.
```

As mentioned earlier, operations DML via view are only allowed for simple and modifiable views. One can only perform the DELETE operation fully through these views. An UPDATE operation requires an additional condition: the view attribute cannot be defined by expression. The INSERT operation is even more demanding: in addition to meeting all previous conditions, mandatory attributes are required to be chosen by the view.

The view is removed by the DROP VIEW command with the following syntax:

DROP VIEW View_name

2.5. Indexes

Indexes do not belong to the ANSI SQL standard although they are implemented by most DBMS, including Oracle. Their creation and removal is another elements of the DDL component of SQL language.

Index - a data structure for:
- accelerating the search for data from relation,
- enforcing unique attribute values.

The basic type of indexes is usually created under DBMS using the so-called balanced B-trees (B*-tree indexes). This guarantees, approximately, the same access times to all rows of the relation, regardless of their location in the relation. On some systems, including Oracle, for UNIQUE and PRIMARY KEY bonds of consistency, indexes are created implicitly (automatically). To explicit creation of index, one use the CREATE INDEX command with the following syntax:

CREATE [**UNIQUE**] **INDEX** Index_name
ON Table_name({attribute_name [**DESC** | <u>**ASC**</u>] [, ...]})

where the UNIQUE forces the uniqueness of the set of attribute values listed after the relation name. DESC specifies the descending direction of building of the index column (by default, the index column is built in ascending order - ASC) via the index attribute. Index attribute may be also argument of function. Such an index is called a functional index.

There are two types of indexes:
- simple index: based on one attribute,
- complex index: based on more than one attribute.

This distinction may have consequences in algorithms that optimize command performance. For example, in Oracle only simple indexes are merged, i.e. used together.

The DBMS may decide, on the base of analysis of the operation of the query performed by the tool optimizing, whether or not to use the index (if the system is equipped with such an optimizer - Oracle has it).

The following conditions must be met for the index to be used:
- the indexed attribute must appear in the WHERE clause,
- the attribute in the WHERE clause cannot be a function argument or part of an expression.

The selection and definition of indexes is part of the physical database design. When choosing them, the following principles should apply:
- setting up an index is beneficial for relations with more rows (over 200 in Oracle). With small relations, the index reading time may be greater than the gain of the command execution time,
- indexation is recommended due to attributes whose values are rather not repeat,
- indexation is advisable for the attributes often used in the WHERE clause (also in connecting conditions),
- if two or more attributes often appear together in a WHERE clause, one should create complex indexes,
- avoid more than three indexes for one relation (overloading DML operations). This rule does not apply if SELECT is the most frequently used command,
- with batch modification of the relation, it is advisable to temporarily delete the indexes superimposed on it, because each batch command causes an independent refresh of the index, which takes time. After batch modification, one ought to restore the indexes.

Indexes are deleted using the DROP INDEX command with the following syntax:

DROP INDEX Index_name

2.6. Sequences

Another useful database object which is created, modified and deleted using the commands of DDL component of Oracle dialect of SQL language, is sequence.

Sequence - a database object for generating unique values, usually for primary keys.

To create, modify and delete sequences, one use the DDL commands, i.e. CREATE, ALTER and DROP, respectively, with the syntax:

CREATE SEQUENCE sequence_name
[**START WITH** begining_value]
[**INCREMENT BY** step]
[**MAXVALUE** maximum_value | **NOMAXVALUE**]
[**MINVALUE** minimum_value | **NOMINVALUE**]
[**CYCLE** | **NOCYCLE**];

ALTER SEQUENCE sequence_name
[**INCREMENT BY** step]
[**MAXVALUE** maximum_value | **NOMAXVALUE**]
[**MINVALUE** minimum_value | **NOMINVALUE**]
[**CYCLE** | **NOCYCLE**];

DROP SEQUENCE sequence_name;

where START WITH specifies the first number to be returned by the sequence, INCREMENT BY determines about how much are to be increased following numbers(1 by default), MAXVALUE and MINVALUE specify the upper and lower limits of the sequence value (default values: NOMAXVALUE and NOMINVALUE respectively), CYCLE determines whether the sequence should be created cyclically (NOCYCLE by default) from MINVALUE to MAXVALUE (reverse for a descending sequence).

The CURRVAL pseudo attribute is used to read the current sequence value.

```
SELECT sequence_name.CURRVAL FROM Dual;
```

Increasing the sequence value is obtained by using the NEXTVAL pseudo attribute.

```
SELECT sequence_name.NEXTVAL FROM Dual;
```

Warning:
Multiple references to NEXTVAL in the same command will return the same number.

Task. *Create a sequence that provides the opportunity to enter new bands into `Bands` relation with consecutive their numbers, starting from 6.*

```
CREATE SEQUENCE Band_numbers
START WITH 6;

sequence BAND_NUMBERS created.

INSERT INTO Bands
VALUES (Band_numbers.NEXTVAL,'NEW','FOREST',NULL);

1 rows inserted.

SELECT * FROM Bands;

BAND_NO           NAME             SITE            BAND_CHIEF
----------------- ---------------- --------------- -----------
1                 SUPERIORS        WHOLE AREA      TIGER
2                 BLACK KNIGHTS    FIELD           BALD
3                 WHITE HUNTERS    ORCHARD         ZOMBIES
4                 PINTO HUNTERS    HILLOCK         REEF
5                 ROCKERSI         FARM
9                 NEW              FOREST

 6 rows selected

ROLLBACK;

rollback complete.

DROP SEQUENCE Band_numbers;

sequence BAND_NUMBERS dropped.
```

The CURRVAL and NEXTVAL pseudo attributes can be used:
- in the SELECT clause of the SELECT statement,
- in the list of values in INSERT command,
- in the SET clause of the UPDATE command
- only in the main (most external) query.

The CURRVAL and NEXTVAL pseudo attributes cannot be used:
- in the SELECT clause defining the view,
- with the DISTINCT qualifier,
- if in the command appears ORDER BY, GROUP BY, CONNECT BY, HAVING clauses,
- with the operators UNION, INTERSECT, MINUS
- in subqueries.

2.7. Transaction processing

Transaction management commands belong to the DCL component of the SQL language.

Transaction - a successful or unsuccessful operation consisting of a series of changes in one or more relations of a database.

The two basic commands for the DCL component of SQL language are the COMMIT command applied to explicitly commit the transaction and the ROLLBACK command to explicitly roll back the transaction.

There are two types of transactions:
• DDL transactions - equivalent to single DDL operation,
• DML transactions - consisting of any number of DML operations.

Every transaction has its beginning and its end.

Transaction start - the first executable DML or DDL instruction.

End of transaction - occurrence of one of the following cases:

- COMMIT (explicit transaction confirmation) or ROLLBACK (explicit transaction rolling back) command,
- DDL command,
- some type of error (e.g. dead-lock),
- end of the program session,
- computer failure.

In addition to the user explicitly committing or rolling back a transaction, the system may approve or rollback the transaction implicitly.

The transaction is committed implicitly:

- before DDL command,
- after the DDL command,
- after normal disconnection from the base.

The transaction is rolled back implicitly:

- after a system error.

The command with an error in the DML transaction is automatically removed without losing any previous changes made during the transaction ("STATEMENT LEVEL ROLLBACK" mechanism based on creating system save points before each DML command). Save points allow one to separation part of the transaction for the rolling back only that part. The save point is explicitly created according to the syntax:

SAVEPOINT savepoint_name;

Rolling back part of the transaction to the save point (without closing) executes the command:

ROLLBACK TO SAVEPOINT savepoint_name;

DML commands can be implicitly committed during the transaction. To allow this, one ought to use the SET AUTOCOMMIT command with the syntax:

SET AUTOCOMMIT {**ON** | <u>**OFF**</u> | number_of_command}

After executing the above command, the DML transaction commands will be always implicitly committed when their number will already equals to the number of specified in the syntax above. If an ON element occurs, an implicit commit will follow each DML command (number_commands = 1).