Databases

Seminar 3

Useful functions

- **IN** - select values which match any value in the list of values:
  X IN (a1, a2, ..., an) ≡ X = a1 or X = a2 or ... or X = an
- **BETWEEN** - select values within a given range:
  X BETWEEN A AND B ≡ (X >= A and X <= B) or (X <= A and X >= B)
- **LIKE** - satisfies a text pattern:
  X LIKE '0%abc_0', where _ denotes 1 symbol, % - any sequence of symbols (can be of length zero).
- **DISTINCT ON** - keeps only the first row of each set of rows where the given expressions evaluate to equal:
  outputs the number of unique departments:
  SELECT count(DISTINCT ON department_nm)
  FROM salary;

- IF ... THEN ... [ELSIF ... THEN ... ELSE ...] END IF - branching conditions:

```
SELECT
    IF number = 0 THEN
        'zero'
    ELSIF number > 0 THEN
        'positive'
    ELSIF number < 0 THEN
        'negative'
    ELSE
        'NULL'
    END IF AS number_class
FROM
    numbers.
```

Keyword WITH

WITH provides a way to write additional statements for use in large queries.

These operators, which are also called Common Table Expressions (CTE),

can be represented as definitions of temporary tables that exist only for one query.

More details will simply be available at the following seminars.

Complex query WITH:

```
WITH
    regional_sales AS (
        SELECT
            region,
            SUM(amount)
AS

total_sales
        FROM  orders
        GROUP BY
region
    ),
    top_regions AS (
        SELECT region
        FROM

regional_sales
        WHERE
            total_sales
>

(SELECT
SUM(total_sales)/10
FROM
regional_sales)
    )
SELECT
    region,    product,
    SUM(quantity) AS


product_units,
    SUM(amount) AS

product_sales
FROM orders
WHERE
    region IN
        (SELECT region
FROM

top_regions)
GROUP BY
    region,    product;
```

Subqueries

A subquery is a query contained in another SQL query. A query containing another subquery is called a containing expression.

- A subquery is always enclosed in parentheses and is usually executed before the containing expression.
- Subqueries can be nested into each other.
- In SELECT, subqueries can be used in all sections except GROUP BY.

Classification of subqueries

1. By interacting with the containing expression:
- Related (i.e. referring to the columns of the main query):
  - To write such queries, it is useful to use aliases. (SELECT ... AS T)
  - For cases when the same table is used in the main query and in the subquery, the use of aliases is mandatory!
  - Are executed for each line of the containing expression.
1. Unrelated (i.e. completely self—contained and independent of the main query) - executed before executing the containing expression.
- By the result of the execution:
  - Scalar (1 column and 1 row)
  - Non - scalar

Using subqueries

Section	Related	Unrelated	Non - scalar
Select	+	+	-
From	-	+	+
Where	+	+	+
Having	+	+	+
Order by	+	+	-

Predicates (for subqueries of the form 1 column and several rows)

EXISTS — The EXISTS condition is TRUE if and only if the cardinality of the subquery result table is greater than zero, otherwise the condition is FALSE:

```
SELECT     SupplierName
FROM     Suppliers
WHERE
    EXISTS(
        SELECT  ProductName
        FROM  Products
        WHERE
            SupplierId = Suppliers.supplierId
            AND Price < 20
    );
```

In

IN - The IN predicate for subqueries works the same as for normal queries (checking if a value is in the list):c

```
SELECT
    emp_id,
    fname,
    lname,
    title
FROM
    employee
WHERE
    emp_id IN(
        SELECT
            superior_emp_id
        FROM
            employee
    );
```

ALL

ALL - TRUE if the result of the subquery is empty or the value of the predicate is TRUE for each row of the subquery; if at least something is FALSE, then it will return FALSE, in all other cases it will return UNKNOWN:

```
SELECT
    EMP_NO
FROM
    EMP
WHERE
    DEPT_NO = 65
    AND EMP_SAL >= ALL(
        SELECT
            EMP1.EMP_SAL
        FROM
            EMP AS EMP1
        WHERE
            EMP.DEPT_NO = EMP1.DEPT_NO
    );
```

ANY

ANY - FALSE if the result of the subquery is empty or the condition value is FALSE for each row of the subquery; if at least something is TRUE, then it will return TRUE, otherwise it will return UNKNOWN:

```
SELECT
    EMP_NO
FROM
    EMP
WHERE
    DEPT_NO = 65
    AND EMP_SAL > ANY(
        SELECT
            EMP1.EMP_SAL
        FROM
            EMP AS EMP1
        WHERE
            EMP.DEPT_NO = EMP1.DEPT_NO
    );
```

CREATE TABLE AS

CREATE TABLE AS - creates a table and fills it with the data obtained as a result of the SELECT execution. The columns of this table are given names and data types according to the columns of the SELECT result (although the column names can be overridden by explicitly adding a list of new colum

```
CREATE TABLE NEW_TABLE AS
    SELECT
        *
    FROM
        OLD_TABLE;
```

Referential integrity

Referential integrity is a necessary quality of a relational database where all values of all foreign keys are valid. A database has the property of referential integrity when the condition of referential integrity is met for any pair of relations linked by a foreign key in it.

In real databases, referential integrity is not always maintained. Violations of referential integrity may occur due to

- a developer skipping the link creation step.
- incorrect operation of the application software:
  - Incomplete recording of objects (object data is placed in several tables, and one of them is not updated).
  - Incorrect edit of the link (the value of the foreign key is changed to one that does not match any record in the linked table)
  - Editing the primary key without cascading updates (the primary key is edited in the referenced table, but the foreign keys in the tables associated with it remain unchanged)
  - Deleting a record without cascading updates (a record that is referenced by foreign keys of other tables is deleted from the table, while the foreign keys in the related records do not change. As a result, all records of other tables referring to it become incorrect)

Referential integrity

- System software and hardware failures:
  - If it is necessary to add data about an object to several pages, then during the transaction the referential integrity will be violated — information has already been entered into some tables, but not in others. So, if the operation is interrupted before completion for some technical reasons, then some of the added records will remain with incorrect links.
- Sometimes, in practice, in the absence of any information about the object, the key remains empty (NULL value). And although this is unacceptable in theory, in practice it can sometimes be convenient.

Constraints

- NOT NULL - all values are not NULL
- UNIQUE - all values are unique
- PRIMARY KEY - primary key of the database. In some DBMS, an additional constraint NOT NULL has to be added (more often it is set to NOT NULL and UNIQUE by default)
- FOREIGN KEY - foreign key, requires a link to another table
- CHECK - checks that the value satisfies condition.
- DEFAULT - sets a value by default. Used if the user did not specify the column value.

- NOT NULL

```
CREATE TABLE PERSON (
    ID         INTEGER      NOT NULL,
    LAST_NAME  VARCHAR(255) NOT NULL,
    FIRST_NAME VARCHAR(255) NOT NULL,
    AGE        INTEGER
);
```

- UNIQUE

```
CREATE TABLE PERSON (
    ID         INTEGER      NOT NULL UNIQUE,
    LAST_NAME  VARCHAR(255) NOT NULL,
    FIRST_NAME VARCHAR(255) NOT NULL,
    AGE        INTEGER
);

ALTER TABLE PERSON ADD UNIQUE (ID);
```

```
ALTER TABLE PERSON
ADD CONSTRAINT
UC_Person UNIQUE (ID,
LAST_NAME);

ALTER TABLE PERSON
DROP CONSTRAINT
UC_Person;
```

- PRIMARY KEY

```
CREATE TABLE PERSON (
    ID          INTEGER      PRIMARY KEY,
    LAST_NAME   VARCHAR(255) NOT NULL,
    FIRST_NAME VARCHAR(255) NOT NULL,
    AGE         INTEGER
);

ALTER TABLE PERSON ADD PRIMARY KEY
(ID);
```

```
CREATE TABLE PERSON (
    ID          INTEGER,
    LAST_NAME   VARCHAR(255),
    FIRST_NAME VARCHAR(255) NOT
NULL,
    AGE         INTEGER,
    CONSTRAINT PK_Person PRIMARY
KEY (ID, LAST_NAME)
);

ALTER TABLE PERSON
ADD CONSTRAINT PK_Person
PRIMARY KEY (ID, LAST_NAME);

ALTER TABLE PERSON
DROP CONSTRAINT PK_Person;
```

- FOREIGN KEY

```
CREATE TABLE ORDER (
   ORDER_ID     INTEGER,
   ORDER_NUMBER INTEGER NOT NULL,
   PERSON_ID    INTEGER,
   PRIMARY KEY (ORDER_ID),
   CONSTRAINT FK_PersonOrder FOREIGN
KEY (PERSON_ID) REFERENCES
PERSON(PERSON_ID)
);

ALTER TABLE ORDER ADD
CONSTRAINT FK_PersonOrder
FOREIGN KEY (PERSON_ID)
REFERENCES PERSON(PERSON_ID);
ALTER TABLE ORDER DROP
CONSTRAINT FK_PersonOrder;
```

```
CREATE TABLE ORDER (
   ORDER_ID     INTEGER PRIMARY
KEY,
   ORDER_NUMBER INTEGER NOT
NULL,
   PERSON_ID    INTEGER REFERENCES
PERSON(PERSON_ID)
);

ALTER TABLE ORDER
ADD FOREIGN KEY (PERSON_ID)
REFERENCES PERSON(PERSON_ID);
```

Constraints for referential integrity

When choosing a FK, the number of matched attributes in the linked tables must be the same. Also, in a table for which the attributes are PKs, these attributes must satisfy the PK constraints. Otherwise the system throws the following error: there is no unique constraint matching given keys for referenced table.
Constraints for referential integrity:

- CASCADE - deletes\updates the referencing rows in the child table when the referenced row is deleted\updated in the parent table which has a primary key.
- RESTRICT - a row can not be deleted\modified while referencing rows it exist.
- NO ACTION
  - Similar to RESTRICT, but the check happens at the end of the transaction
  - To get a different result from RESTRICT, you need to explicitly specify the expression (SET CONSTRAINTS) in the transaction
    SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }

Constraints for referential integrity

- SET NULL - when a row is deleted from the main table, the corresponding value in the child table becomes NULL
- SET DEFAULT - similar to SET NULL, but instead of NULL a different value is set by default.

```
CREATE TABLE ORDER (
    ORDER_ID     INTEGER,
    ORDER_NUMBER INTEGER NOT NULL,
    PERSON_ID    INTEGER,

    PRIMARY KEY (ORDER_ID),
    CONSTRAINT FK_PersonOrder FOREIGN KEY (PERSON_ID)
        REFERENCES PERSON(PERSON_ID)
            ON DELETE RESTRICT
            ON UPDATE RESTRICT
);
```

Constraints for referential integrity

- DEFAULT

```
CREATE TABLE ORDER (
    ORDER_ID     INTEGER PRIMARY KEY,
    ORDER_NUMBER INTEGER NOT NULL,
    ORDER_DATE   DATE    DEFAULT now()::date
);

ALTER TABLE ORDER;
ALTER COLUMN ORDER_DATE DROP DEFAULT;
```

Constraints for referential integrity

- CHECK

```
CREATE TABLE PERSON (
    ID         INTEGER      NOT NULL,
    LAST_NAME  VARCHAR(255) NOT
NULL,
    FIRST_NAME VARCHAR(255) NOT
NULL,
    AGE        INTEGER      CHECK (AGE
>= 18)
);

ALTER TABLE PERSON ADD CHECK
(AGE >= 18);
```

```
CREATE TABLE PERSON (
        ID         INTEGER      NOT NULL,
        LAST_NAME   VARCHAR(255) NOT
    NULL,
        FIRST_NAME  VARCHAR(255) NOT
    NULL,
        AGE        INTEGER,
        CITY       VARCHAR(255),
        CONSTRAINT CHK_Person CHECK
    (AGE >= 18 AND CITY = 'Moscow')
);
ALTER TABLE PERSON ADD CONSTRAINT
CHK_Person
CHECK (AGE >= 18 AND CITY =
'Moscow');

ALTER TABLE PERSON DROP CONSTRAINT
CHK_Person;
```

Get all constraints

- Identifies all columns in the current database that are used by some constraint:
  SELECT * FROM information_schema.constraint_column_usage;
- Contains all constraints belonging to tables that the current user owns or has some privilege other than SELECT on:
  SELECT * FROM information_schema.table_constraints;
- Identifies all columns in the current database that are restricted by some unique, PK, or FK constraint:
  SELECT * FROM information_schema.key_column_usage;
- Contains all `CHECK` constraints:
  SELECT * FROM information_schema.check_constraints;
- Contains all `DEFAULT` and `NOT NULL` constraints:
  SELECT * FROM information_schema.columns;