

Structured Query Language SQL

Contents

1	SELECT Statement	2
2	Selection Conditions	7
3	Aggregate Functions	13
4	Subquery	17
5	Set-theoretic Operations	24
6	Join Tables	29

1 SELECT Statement

So we learned how to design data, create tables with specified column types and integrity constraints and fill them with values. Now let's start writing queries. To write a data query there is only one operator - it is called **SELECT**. Only one operator but you can write very complex queries with it like extract data from multiple tables, satisfying many criteria. Let's take a look at its syntax:

Syntax of SELECT statement

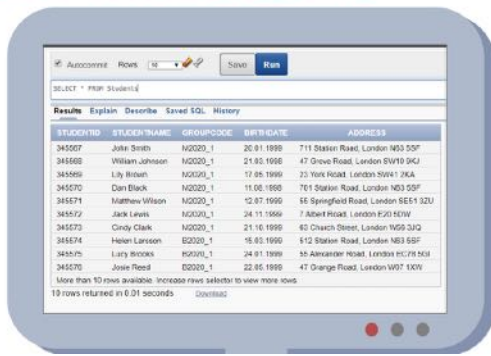
```
SELECT [ALL | DISTINCT]
      {*[name_column [AS new_name]]} [...n]
      FROM name_table [[AS] alias] [...n]
      [WHERE <conditions>]
      [GROUP BY name_column [...n]]
      [HAVING <group selection criteria >]
      [ORDER BY name_column [...n]]
```

- **FROM** – defines the names of the tables;
- **WHERE** – filters the object rows according to the specified conditions;
- **GROUP BY** – groups the rows having the same value in the specified column;
- **HAVING** – filters groups of rows according to the specified condition;
- **SELECT** – selects columns for output data;
- **ORDER BY** – sorts the result-set in ascending or descending order.

Let's start with the simplest query - display the entire contents of the specified table. The table name is specified after the word **FROM**. After **SELECT** the output columns are indicated. If **SELECT** is followed by * (asterisk) it means «all columns». Display the contents of the **Students** table and the **St_Group** table.

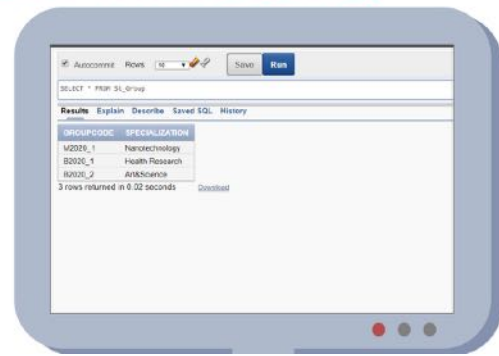
If the number of records are very large the DBMS can display a limited number of rows.

If you want to display not all columns of the table but certain ones, their names must be specified after the word **SELECT**. You can specify table columns in different ways. You can specify * - all table columns, you can list the column names, you can specify the table name before the column name. The table name

SELECT * FROM Students


The screenshot shows a database query interface with a toolbar at the top containing 'Autocommit', 'Rows', a dropdown menu, and 'Save' and 'Run' buttons. Below the toolbar, the query 'SELECT * FROM Students' is entered. The results are displayed in a table with columns: STUDENTID, STUDENTNAME, GROUPCODE, BIRTHDATE, and ADDRESS. The table contains 10 rows of student data. At the bottom, a message states 'More than 10 rows available. Increase rows selected to view more rows. 10 rows returned in 0.01 seconds' with a 'Download' link.

STUDENTID	STUDENTNAME	GROUPCODE	BIRTHDATE	ADDRESS
345007	John Smith	NG020_1	20.01.1998	711 Station Road, London N03 5DP
345008	William Johnson	NG020_1	21.03.1998	47 Grove Road, London SW19 9DA
345009	Lily Brian	NG020_1	17.05.1999	23 York Road, London NW4 1BA
345010	Dan Black	NG020_1	11.06.1999	761 Station Road, London N03 5DP
345011	Matthew Wilson	NG020_1	12.07.1999	56 Springfield Road, London SE14 3ZU
345012	Jack Lewis	NG020_1	24.11.1999	7 Albert Road, London E20 9EW
345013	Cindy Clark	NG020_1	21.10.1999	62 Church Street, London NG9 3JQ
345014	Heaven Larsson	EG020_1	15.03.1999	112 Station Road, London N03 5DP
345015	Larry Brown	NG020_1	24.01.1999	58 Alexander Road, London EC2R 8SA
345016	Joan Reed	EG020_1	22.05.1998	47 Orange Road, London W07 1XU

SELECT * FROM St_Group


The screenshot shows a database query interface with a toolbar at the top containing 'Autocommit', 'Rows', a dropdown menu, and 'Save' and 'Run' buttons. Below the toolbar, the query 'SELECT * FROM St_Group' is entered. The results are displayed in a table with columns: GROUPCODE and SPECIALIZATION. The table contains 3 rows of group data. At the bottom, a message states '3 rows returned in 0.02 seconds' with a 'Download' link.

GROUPCODE	SPECIALIZATION
NG020_1	Nanotechnology
EG020_1	Health Research
EG020_2	Anticancer

Selection of given columns

- *SELECT * FROM table*
- *SELECT table.* FROM table*
- *SELECT column1, column2 FROM table*
- *SELECT table.column1, table.column2 FROM table*

and column name are separated by a dot. Multiple tables can be used in a query and if they contain columns with the same name you must specify the table name before the column name to indicate the column you are referring to. You can specify all columns of a certain table by writing **table name. ***.

In the first query

SELECT StudentID FROM Students

the values of the column **StudentID** of the table **Students** are displayed — these are the numbers of students' record books.

In the second query

SELECT StudentName, GroupCode FROM Students

StudentName, GroupCode records of the table **Students** are displayed. There are many students in each group.

The third query

SELECT GroupCode FROM Students

displays only the groups numbers in which students are studying.

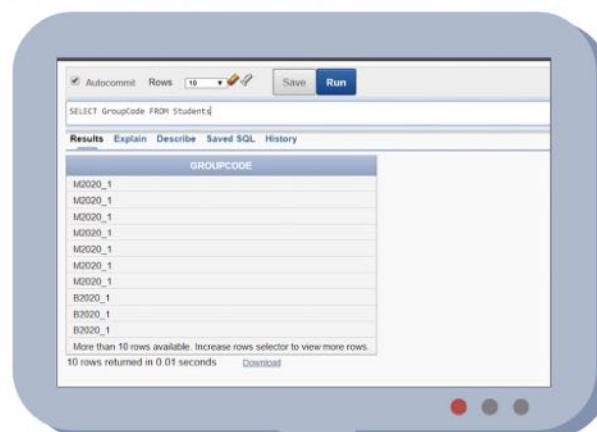
Probably it looks strange that the same group number is displayed several times - the **GroupCode** value is displayed for each row when the query is executed, i.e. for each student despite the recurrence.

Selection of given columns

- *SELECT StudentId FROM Students*
- *SELECT StudentName, GroupCode FROM Students*
- *SELECT GroupCode FROM Students*

StudentId	StudentName	GroupCode	BirthDate	Address
345568	William Johnson	M2020_1	01/20/1999	711 Station Road, London N63 5SF
345569	Lily Brown	M2020_1	05/17/1999	93 Manchester Road, London SE02 6VS
345583	Thomas Wood	B2020_2	09/12/1999	951 Highfield Road, London E16 2RI
345571	Matthew Wilson	M2020_1	07/12/1999	55 Springfield Road, London SE51 3ZU
345579	Michael Young	B2020_1	04/08/1998	29 The Crescent, London W20 9FK

SELECT GroupCode FROM Students



By default all selected values are displayed. To indicate this explicitly you can specify the keyword **ALL** before the field names. To display only unique rows it is necessary to add **DISTINCT** before the field name (or fields). The duplicate rows will be removed then. Each group code is displayed once in a query with group codes from the **Students** table .

1.0.1 Rows Sorting

Without the sort criteria the selected rows are shown in random order. To show it in a certain order the keyword **ORDER BY** is needed followed by the column or columns that are the sort criteria. The sorting order can be by increasing **ASC** or by descending **DESC**. If the criteria are specified but not the order ascending sorting is used by default then. For example, you can display all the contents of the **Students** table by ordering rows by **StudentName** or by Group Code or by

Unique records

- `SELECT GroupCode FROM Students`
- `SELECT ALL GroupCode FROM Students`
- `SELECT DISTINCT GroupCode FROM Students`

Sorting records

- `SELECT * FROM Students ORDER BY StudentName`

StudentId	StudentName	GroupCode	BirthDate	Address
345569	Lily Brown	M2020_1	05/17/1999	93 Manchester Road, London SE02 6VS
345571	Matthew Wilson	M2020_1	07/12/1999	55 Springfield Road, London SE51 3ZU
345579	Michael Young	B2020_1	04/08/1998	29 The Crescent, London W20 9FK
345583	Thomas Wood	B2020_2	09/12/1999	951 Highfield Road, London E16 2RI
345568	William Johnson	M2020_1	01/20/1999	711 Station Road, London N63 5SF

Group Descending Code:

```
SELECT * FROM Students ORDER BY StudentName
```

```
SELECT * FROM Students ORDER BY GroupCode
```

```
SELECT * FROM Students ORDER BY GroupCode DESC
```

To sort within the group sorting by name is added.

```
SELECT * FROM Students ORDER BY GroupCode DESC, StudentName
```

It is possible not to display all fields but only the student name and group code:

```
SELECT StudentName, GroupCode FROM Students  
ORDER BY GroupCode DESC, StudentName
```

Even fields that are not included in the final select can be sorted. For example, if you specify the `StudentId` field as a sort criteria the rows are ordered by increasing of these values but only the specified columns are displayed. `StudentName`, `GroupCode`.

```
SELECT StudentName, GroupCode FROM Students ORDER BY StudentId
```

Instead of field names it is possible to sort their sequence numbers in the source table. But this method of specifying columns is not popular.

```
SELECT StudentName, GroupCode FROM Students ORDER by 2, 1
```

1.0.2 Attributes Renaming

Selected columns names come from the original table. To rename them it is necessary to specify a new name after the original column name and the keyword **as**:

```
SELECT StudentName as Name, GroupCode as Group_Name FROM Students
```

Renaming attributes

- *SELECT StudentName as **Name**, GroupCode as **Group** FROM Students*



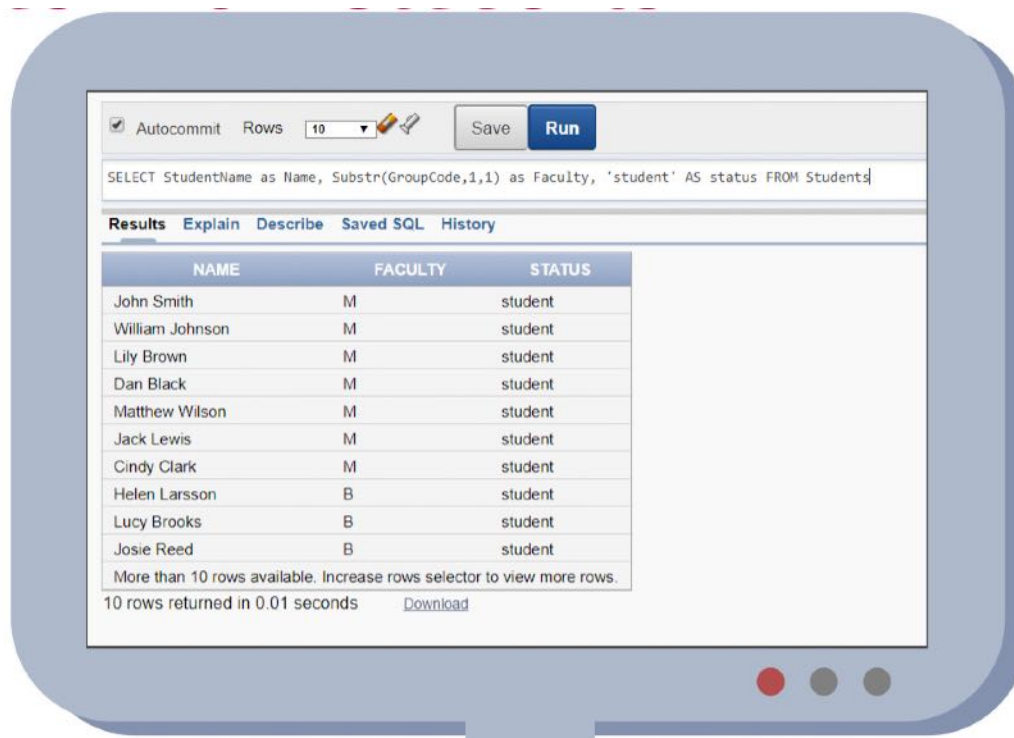
StudentId	Name	Group	BirthDate	Address
345568	William Johnson	M2020_1	01/20/1999	711 Station Road, London N63 5SF
345569	Lily Brown	M2020_1	05/17/1999	93 Manchester Road, London SE02 6VS
345583	Thomas Wood	B2020_2	09/12/1999	951 Highfield Road, London E16 2RI
345571	Matthew Wilson	M2020_1	07/12/1999	55 Springfield Road, London SE51 3ZU
345579	Michael Young	B2020_1	04/08/1998	29 The Crescent, London W20 9FK

In fact, it is possible not only display the values of the columns in the table but to make up expressions with them. Arithmetic operations, string conversion functions, etc. can be used. For example, take the first letter of the group code and call it a faculty field.

```
SELECT StudentName as Name, Substr(GroupCode,1,1) as Faculty  
FROM Students
```

Or add a status with the Student value as an additional column for the selected data.

```
SELECT StudentName as Name, Substr(GroupCode,1,1) as Faculty,  
'student' AS status FROM Students
```



2 Selection Conditions

So far we have selected columns but we have displayed values for all records of the table except removing duplicates. Often it is not necessary to display all of them but only those that return only a few records that meet certain criteria/condition.

The condition is specified after the **WHERE**. What could be the condition? This is some logical expression that can be a sort of combination of column names, constants, logical operators, arithmetic operators, functions, etc. The simplest example is to compare a column value with a constant. For example, select records from **Students** where **StudentId** = 345569.

```
SELECT * FROM Students WHERE StudentId = 345569
```

In this case the result of the selection is the only row, because the specified condition is the exact value of the primary key. If you specify a group code as a condition, for example, **M2020_1**, the corresponding rows of all students in the group will be displayed.

```
SELECT * FROM Students WHERE GroupCode = 'M2020_1'
```

Note how constants are specified: strings and dates are typed in quotation marks usually in single ones but numeric constants are without quotation marks.

So one way to define a condition is a comparison operation. Under this type of condition the value of one expression is compared to the value of the other.

The result of the selection includes rows for which the value of the comparison operator is true. Of course, not only equality but also $<>$, $>$, $>=$, $<$, $<=$. Each

Selection condition – comparison

Comparison: the results of the calculation of one expression are compared with the results of the another calculation.

< 345569

StudentId	StudentName	GroupCode	BirthDate	Address
345568	William Johnson	M2020_1	01/20/1999	711 Station Road, London N63 5SF
345569	Lily Brown	M2020_1	05/17/1999	93 Manchester Road, London SE02 6VS
345583	Thomas Wood	B2020_2	09/12/1999	951 Highfield Road, London E16 2RI
345571	Matthew Wilson	M2020_1	07/12/1999	55 Springfield Road, London SE51 3ZU
345579	Michael Young	B2020_1	04/08/1998	29 The Crescent, London W20 9FK

of the mentioned comparison operators is binary. It means that two operands are needed. The result of the execution is a logical value (true - false).

For example, select all students with **StudentId** < 345569 value or all students whose code is not M2020_1

```
SELECT * FROM Students WHERE StudentId < 345569
```

```
SELECT * FROM Students WHERE GroupCode <> 'M2020_1'
```

Selection condition – set membership

Set membership: to check whether the result of expression calculations belongs to a given set of values.

\in
 $\{M2020_1,$
 $B2020_1\}$

StudentId	StudentName	GroupCode	BirthDate	Address
345568	William Johnson	M2020_1 ✓	01/20/1999	711 Station Road, London N63 5SF
345569	Lily Brown	M2020_1 ✓	05/17/1999	93 Manchester Road, London SE02 6VS
345583	Thomas Wood	B2020_2 ✗	09/12/1999	951 Highfield Road, London E16 2RI
345571	Matthew Wilson	M2020_1 ✓	07/12/1999	55 Springfield Road, London SE51 3ZU
345579	Michael Young	B2020_1 ✓	04/08/1998	29 The Crescent, London W20 9FK

Another way to specify a selection condition is to check whether the result of the calculation of an expression matches any value in a list of values. The

keyword **IN** is used to check it. The elements of a range can be constants of the corresponding type listed by comma or the results of a subquery.

Let's make a selection from the table **Students** with a condition that the result is within a specified range. For example, set the selection criteria as follows: the group code belongs to the range 'M2020_1', 'B2020_1'

```
SELECT * FROM Students WHERE GroupCode IN ('M2020_1', 'B2020_1')
```

When designing the table we mentioned that some columns might not be set. For example, you can save information about a student if you do not know his date of birth. Or schedule an exam for a group on a certain subject specifying a date, but not the classroom. How can we find records whose attribute values are not specified?

Selection condition – non-specified attribute value

NULL: to check if the column contains a null (non-specified value).



StudentId	StudentName	GroupCode	BirthDate	Address
345568	William Johnson	M2020_1	01/20/1999 ✗	711 Station Road, London N63 5SF
345569	Lily Brown	M2020_1	05/17/1999 ✗	93 Manchester Road, London SE02 6VS
345583	Thomas Wood	B2020_2	- ✓	951 Highfield Road, London E16 2RI
345571	Matthew Wilson	M2020_1	07/12/1999 ✗	55 Springfield Road, London SE51 3ZU
345579	Michael Young	B2020_1	04/08/1998 ✗	29 The Crescent, London W20 9FK

At first glance, seems that an unspecified string is equal to an empty string and an unspecified number is a zero. But this is not the case. For unspecified data, null values, there is a special designation - **NULL**. With these values you have to be very accurate. If ordinary given values we can always compare whether they are equal or not, then with the non-data values there is uncertainty. If different exams are scheduled but classrooms are not specified can we assert the classroom is the same then? Of course not. But we can not assert that they are different either. In the same way, when we compare a given value to non-data, we can not get "true" or "false" but an uncertain value instead. To verify that a column value is not valid it is necessary to use the phrase **IS NULL**, which means an empty value, and **IS NOT NULL**, which means a specified, defined value.

Let's create some queries. For example, select students who have no date of birth:

```
SELECT * FROM Students WHERE BirthDate IS NULL
```

Or select exams for which no classroom is defined:

```
SELECT * from Exam_Sheet WHERE Classroom IS NULL
```

And now the classroom is defined:

```
SELECT * from Exam_Sheet WHERE Classroom IS NOT NULL
```

Selection condition – strings patterns

Pattern matching: to check whether some string value matches the specified pattern.



StudentId	StudentName		GroupCode	BirthDate	Address
345568	William Johnson	✓	M2020_1	01/20/1999	711 Station Road, London N63 5SF
345569	Lily Brown	✓	M2020_1	05/17/1999	93 Manchester Road, London SE02 6VS
345583	Thomas Wood	✗	B2020_2	09/12/1999	951 Highfield Road, London E16 2RI
345571	Matthew Wilson	✓	M2020_1	07/12/1999	55 Springfield Road, London SE51 3ZU
345579	Michael Young	✗	B2020_1	04/08/1998	29 The Crescent, London W20 9FK

For string values it is possible to search by a given pattern. For example, to select all string column values that begin with a particular letter, or contain a specific sequence of characters.

To do this, use the keyword **LIKE** followed the column name. After that it is necessary to determine the pattern itself. Special characters can be used: **%** means any sequence of characters, including an empty one, **_** (underlining) means any character.

Examples:

- **'x%'** – any strings that start with the letter x;
- **'ab_'** – the length of strings equals to 3 and the first characters of the string are ab;
- **'%t'** – any sequence of characters ends with the character t;
- **'%car%'** – any sequence of characters contains the word car at any position of the string.

Apply the pattern search to the **GroupCode** row column of the **Students** table. For example, find all students whose group code begins with the letter M - **%** after the letter in the pattern indicates an arbitrary sequence of characters:

**SELECT * FROM Students
WHERE GroupCode LIKE 'M%'**

STUDENTID	STUDENTNAME	GROUPCODE	BIRTHDATE	ADDRESS
345567	John Smith	M2020_1	20-01-1999	711 Station Road, London N63 5SF
345568	William Johnson	M2020_1	21-03-1999	47 Grove Road, London SW10 9KJ
345569	Lily Brown	M2020_1	17-05-1999	23 York Road, London SW14 2NA
345570	Dan Black	M2020_1	11-08-1998	701 Station Road, London N63 5SF
345571	Matthew Wilson	M2020_1	12-07-1999	55 Springfield Road, London SE51 3ZU
345572	Jack Lewis	M2020_1	24-11-1999	7 Albert Road, London E20 6DN
345573	Cindy Clark	M2020_1	21-10-1999	63 Church Street, London W9E 3JG

7 rows returned in 0.01 seconds

**SELECT * FROM Students
WHERE GroupCode LIKE '_2020%'**

STUDENTID	STUDENTNAME	GROUPCODE	BIRTHDATE	ADDRESS
345567	John Smith	M2020_1	20-01-1999	711 Station Road, London N63 5SF
345568	William Johnson	M2020_1	21-03-1999	47 Grove Road, London SW10 9KJ
345569	Lily Brown	M2020_1	17-05-1999	23 York Road, London SW14 2NA
345570	Dan Black	M2020_1	11-08-1998	701 Station Road, London N63 5SF
345571	Matthew Wilson	M2020_1	12-07-1999	55 Springfield Road, London SE51 3ZU
345572	Jack Lewis	M2020_1	24-11-1999	7 Albert Road, London E20 6DN
345573	Cindy Clark	M2020_1	21-10-1999	63 Church Street, London W9E 3JG
345574	Helen Larsen	B2020_1	05-03-1999	512 Station Road, London N63 5SF
345575	Lucy Brooks	B2020_1	24-01-1999	59 Alexander Road, London EC7B 5JZ
345576	Anna Reed	B2020_1	22-06-1999	47 George Road, London W9F 9KJ

More than 10 rows available. Increase rows selector to view more rows.
10 rows returned in 0.01 seconds

SELECT * FROM Students WHERE GroupCode LIKE 'M%'

The next query selects all the groups of the 2020 year of admission - in the pattern the first symbol can be any followed by the 2020 characters then an arbitrary sequence of characters:

SELECT * FROM Students WHERE GroupCode LIKE '_2020%'

One more query with the pattern is to find all students whose last names start with the letter B. The **StudentName** field contains both first and last names of students. In order to search by the last name pattern, a space sign is used

Selection condition – strings patterns

Pattern matching: to check whether some string value matches the specified pattern.



StudentId	StudentName	GroupCode	BirthDate	Address
345568	William Johnson ✗	M2020_1	01/20/1999	711 Station Road, London N63 5SF
345569	Lily Brown ✓	M2020_1	05/17/1999	93 Manchester Road, London SE02 6VS
345583	Thomas Wood ✗	B2020_2	09/12/1999	951 Highfield Road, London E16 2RI
345571	Matthew Wilson ✗	M2020_1	07/12/1999	55 Springfield Road, London SE51 3ZU
345579	Michael Young ✗	B2020_1	04/08/1998	29 The Crescent, London W20 9FK

before the letter B and a percentage sign is added before and after the space and the letter B - any sequence of characters:

SELECT * FROM Students WHERE StudentName LIKE '% B%'

2.0.1 Logical Expressions

So far we have considered only simple selection conditions, when column values or expressions have been compared with constants or values of other expressions. But often it is necessary to match several criteria at once. To match

Logical expressions

- Operations of the same order are calculated from left to right.
- Arithmetic operations are calculated first.
- The NOT operation is executed before the AND and OR operations are executed.
- AND operations are executed before OR operations.



several conditions they can be joined by logical operations: negation, multiplication and addition: NOT, AND, OR. Parentheses are used as well. Logical expressions are written after the word **WHERE**.

Let's start with the simplest logical expression – with the negation NOT. We selected records that fit the given pattern using the LIKE sentence. So we got all the students who live on **Station Road**:

```
SELECT * FROM Students WHERE Address LIKE '% Station Road%'
```

If we specify a negation operation before LIKE then the query selects everyone who does not live on **Station Road**:

```
SELECT * FROM Students WHERE Address NOT LIKE '% Station Road%'
```

Now let's create a logical expression by combining two conditions: the address has a substring **Station Road** and the date of birth is greater or equal to 01/01/1999 (January 1, 1999). We used the logical AND operation to fulfill both of these conditions:

```
SELECT * FROM Students  
WHERE Address LIKE '% Station Road%'  
AND  
BirthDate >= '01/01/1999'
```

Another example: Select all students whose `StudentId` field value is less than 345574 or more 345586. The logical `OR` operation requires at least one of the following conditions to be fulfilled:

```
SELECT * FROM Students  
WHERE StudentId < 345574 OR StudentId > 345586
```

3 Aggregate Functions

Sometimes it is necessary not only select records from the table but to execute some aggregating processing of these rows – to get the number of records, the sum or average value of the numeric column, the earliest/later date, etc. Aggregate functions help us to solve such problems.

An aggregate function is a function that executes an operation on a set of column values (or expressions) and returns a single value as a result.

All DBMS support at least five standard aggregate functions:

- `COUNT (Выражение)` – returns the number of records;
- `MIN (Выражение)` – returns the smallest value of the selected column;
- `MAX (Выражение)` – returns the largest value of the selected column;
- `AVG (Выражение)` – returns the average value of a numeric column;
- `SUM (Выражение)` – returns the total sum of a numeric column.

The arguments of the sum function and the average value can be numbers only, i.e. you can use either columns with numeric data types or numeric expressions. The remaining functions can also have other types of values as arguments. The `COUNT` function can have the `*` character as its argument meaning «all records».

Aggregate values for all rows

`SELECT COUNT(*) FROM Students`



StudentId	StudentName	GroupCode	BirthDate	Address
345568	William Johnson	M2020_1	01/20/1999	711 Station Road, London N63 5SF
345569	Lily Brown	M2020_1	05/17/1999	93 Manchester Road, London SE02 6VS
345583	Thomas Wood	B2020_2	09/12/1999	951 Highfield Road, London E16 2RI
345571	Matthew Wilson	M2020_1	07/12/1999	55 Springfield Road, London SE51 3ZU
345579	Michael Young	B2020_1	04/08/1998	29 The Crescent, London W20 9FK

3.0.1 Aggregate Values for All Rows

Aggregate functions can be applied to the entire table or to individual groups of records. If the aggregate function applies to the entire table its name is specified after the **SELECT** command.

For example, let's find the total number of students:

```
SELECT COUNT(*) FROM Students
```

Let's call the result `Number_Of_Students`:

```
SELECT COUNT(*) AS Number_Of_Students FROM Students
```

Execute the query

```
SELECT MAX(BirthDate) FROM Students
```

and that way we can find the date of birth of the youngest student. In one query you can use several aggregation functions at once: for example, you can find both the number of students and the date of birth of the youngest student.

But it is impossible to use column values together with aggregation functions without aggregation in one query. For example, we have just found the date of birth of the youngest student - we would like to know his name. But the query

```
SELECT StudentsName, MAX(BirthDate) FROM Students
```

is incorrect. To find out the name of the youngest student we have to create a slightly more complex query.

3.0.2 Aggregate Values for All Rows With a Condition

If the selection condition of the rows in the table is specified together with the aggregation function it applies only to rows that meet the specified condition.

For example, let's count not all students in the table, but only those who study in the group M2020_1:

```
SELECT COUNT(*) FROM Students WHERE GroupCode='M2020_1'
```

Or count students from two groups at once: 'B2020_1' и 'B2020_2':

```
SELECT COUNT(*) FROM Students  
WHERE GroupCode='B2020_1' OR GroupCode='B2020_2'
```


When counting the rows arguments of the COUNT function, not only the * sign can be used, but the number of values of individual columns can be counted. For example, let's count the number of groups:

```
SELECT COUNT(GroupCode) FROM Students
```

There are as many of them as there are rows in the table **Students**:

```
SELECT GroupCode FROM Students
```

Aggregate unique records



How many different groups are there?

```
SELECT COUNT(DISTINCT GroupCode) FROM Students
```

StudentId	StudentName	GroupCode	BirthDate	Address
345568	William Johnson	M2020_1	01/20/1999	711 Station Road, London N63 5SF
345569	Lily Brown	M2020_1	05/17/1999	93 Manchester Road, London SE02 6VS
345583	Thomas Wood	B2020_2	09/12/1999	951 Highfield Road, London E16 2RI
345571	Matthew Wilson	M2020_1	07/12/1999	55 Springfield Road, London SE51 3ZU
345579	Michael Young	B2020_1	04/08/1998	29 The Crescent, London W20 9FK

The aggregation function does not count the number of unique values of a given column but the total number of values. Can we count the number of different groups? Of course, the word **DISTINCT** can help us. We have already used it to select only unique group values from the **Students** table:

```
SELECT DISTINCT GroupCode FROM Students
```


With this keyword we can count the number of unique groups also:

```
SELECT COUNT(DISTINCT GroupCode) FROM Students
```


3.0.3 Aggregation With Grouping

We can select the number of students from one group, the second, the third. But if there are many groups do we really have to count for each group separately setting its code as a condition? There is an easier way it is called grouping. As a

Aggregate with grouping



How many students are in each of the groups?

```
SELECT COUNT(*) FROM Students GROUP BY GroupCode
```

StudentId	StudentName	GroupCode	BirthDate	Address
345568	William Johnson	M2020_1	01/20/1999	711 Station Road, London N63 5SF
345569	Lily Brown	M2020_1	05/17/1999	93 Manchester Road, London SE02 6VS
345583	Thomas Wood	B2020_2	09/12/1999	951 Highfield Road, London E16 2RI
345571	Matthew Wilson	M2020_1	07/12/1999	55 Springfield Road, London SE51 3ZU
345579	Michael Young	B2020_1	04/08/1998	29 The Crescent, London W20 9FK

condition, we can specify a column (or several columns) and those rows in which the values of the specified condition match are in the same group.

For example, group the table **Student** by the value of the field **GroupCode**. Now the aggregation functions specified after the word **SELECT** are executed for each group separately.

```
SELECT COUNT(*) FROM Students GROUP BY GroupCode
```

Now we can find out the number of students in each group separately. As a result of the query we can get a column of numbers each of which corresponds to the value of the aggregation function for each group. But now how to find out which digit corresponds to which group?

To do this it is necessary to specify the name of the grouping column together with the aggregation function:

```
SELECT GroupCode, COUNT(*) FROM Students GROUP BY GroupCode
```

Grouping columns names can and should be placed after **SELECT** together with aggregation functions. But the names of other columns that are not grouped cannot be used in the query.

Executing a grouping it is possible to use the **HAVING** statement - it allows you to filter groups. After the **HAVING** construction we can write aggregate functions and ranges of the values of the columns to be grouped.

Suppose we do not need to know the result of the aggregating function for each group but only those groups in which the result of the aggregate function

satisfies a given condition. For example, we found the number of students in each group and display that number together with the group numbers. Now we can select only group numbers with fewer than 8 students:

```
SELECT GroupCode FROM Students
GROUP BY GroupCode
HAVING COUNT(*) < 8
```

The following example selects the group numbers and the number of students in them if the group has more than 3 students and the group name starts with B.

```
SELECT GroupCode, COUNT(*) FROM Students
GROUP BY GroupCode
HAVING COUNT(*) > 3 AND GroupCode LIKE 'B%'
```

After **HAVING** can we restrict the values of individual columns rather than the result of the aggregation function? For example, write a condition for the number of students and a condition for the value of the field `StudentID < 345577`?

```
SELECT GroupCode FROM Students
GROUP BY GroupCode
HAVING COUNT(*) < 8 AND StudentID < 345577
```

It is incorrect.

To select rows that match a certain condition we should use the condition after the **WHERE** keyword. Then first the rows for which the condition is satisfied will be selected from the table. Then these rows will be grouped and the aggregation function specified after **HAVING** will be performed for each group and its result will be checked:

```
SELECT GroupCode FROM Students
WHERE StudentID < 345577
GROUP BY GroupCode
HAVING COUNT(*) < 8
```

It is possible to apply sorting to grouped aggregation. For example, sort the list of groups by descending, starting with the largest group.

4 Subquery

Let's go back to the **SELECT** statement description to understand the execution order. First queries the table specified after **FROM**. Then the records from the table that match the condition specified after **WHERE** are selected. The rows

are grouped according to the criteria after **GROUP BY**. Then the groups are filtered according to the **HAVING** condition. The resulting aggregate functions are counted or the columns of the table are selected, and finally the final result is sorted in the specified order.

SELECT execution order

- *FROM*
- *WHERE*
- *GROUP BY*
- *HAVING*
- *SELECT*
- ***ORDER BY***



Queries may contain subqueries. Nested queries can be used after the word **SELECT** instead of the table name in the query and after the words **WHERE** and **HAVING**. Subqueries are needed to combine or map data from multiple tables. The subquery is always enclosed in parentheses.

Subqueries can return one value or the result can have a whole table. There are subqueries that are called correlated if they refer to columns in the table of the external query.

Let's return to the query we used to find the youngest student:

```
SELECT MAX(BirthDate) FROM Students
```


Now we want to know the student's name. Hopefully we remember not to use aggregation functions and table columns without grouping.

The subquery can help. We find the date of birth of the youngest student and then select from the table the student whose date of birth is the found maximum. We compared the value of the **BirthDate** column to the result of the subquery using equality because we are sure that the result of the subquery is a single scalar value.

```
SELECT StudentName FROM Students  
WHERE BirthDate = (SELECT MAX(BirthDate) FROM Students)
```

If the result of the subquery is a list of values the resulting query cannot be executed.

Subqueries



What is the name of the youngest student?

```
SELECT StudentsName, MAX(BirthDate) FROM Students
```

StudentId	StudentName	GroupCode	BirthDate	Address
345568	William Johnson	M2020_1	01/20/1999	711 Station Road, London N63 5SF
345569	Lily Brown	M2020_1	05/17/1999	93 Manchester Road, London SE02 6VS
345583	Thomas Wood	B2020_2	09/12/1999	951 Highfield Road, London E16 2RI
345571	Matthew Wilson	M2020_1	07/12/1999	55 Springfield Road, London SE51 3ZU
345579	Michael Young	B2020_1	04/08/1998	29 The Crescent, London W20 9FK

Consider another example. Suppose we need to find students who specialize in Nanotechnology (Nanotechnology). To do this from the table **St_Group** find a value **GroupCode**:

```
SELECT GroupCode FROM St_Group
WHERE Specialization = 'Nanotechnology'
```

Now find students by the specified group code from the table **Students**:

```
SELECT StudentName FROM Students
WHERE GroupCode='M2020_1'
```

There are two simple queries above but in fact they can be combined in one. Substitute the first query instead of the value of the **GroupCode** field in the second query.

```
SELECT StudentName FROM Students
WHERE GroupCode = (SELECT GroupCode FROM St_Group
WHERE Specialization = 'Nanotechnology')
```

So far we have used the fact that the subquery returns a single result. Let's try multiple values as the first query result. We can select students who specialize in Nanotechnology or Healthcare. Again execute a query to the table **St_Group** and now this query returns two values of the field **GroupCode**:

```
SELECT GroupCode FROM St_Group
WHERE Specialization = 'Nanotechnology'
OR Specialization = 'Health Research'
```

This query can be rewritten by a set membership where the specialization names are the elements of the set.

```
SELECT GroupCode FROM St_Group
WHERE Specialization IN ('Nanotechnology', 'Health Research')
```

We can specify the elements of a set by enumerating constants or we can form them from the results of the subquery. Write a request to find the names of students specializing in Nanotechnology and Health Research using a subquery:

```
SELECT StudentName FROM Students
WHERE GroupCode IN (SELECT GroupCode FROM St_Group
WHERE Specialization = 'Nanotechnology'
OR Specialization = 'Health Research')
```

The subquery can be used after **SELECT**, together with the column names but if it returns exactly one value only.

For example, select a list of students' names and the corresponding group codes:

```
SELECT StudentsName, GroupCode FROM Students
```

Now add a column with group specialization. This column does not exist in the table **Students** but it is in the table **St_Group**. We know that for each group code there is exactly one specialization.

```
SELECT GroupCode, Specialization FROM St_Group
```

Subqueries

```
SELECT StudentName, GroupCode,
(SELECT Specialization FROM St_Group WHERE
St_Group.GroupCode=Students.GroupCode) FROM Students;
```

StudentName	GroupCode	GroupCode	Specialization
William Johnson	M2020_1	M2020_1	Nanotechnology
Lily Brown	M2020_1	B2020_1	Health Research
Thomas Wood	B2020_2	B2020_2	Art&Science
Matthew Wilson	M2020_1		
Michael Young	B2020_1		



Then we can add a subquery to the student's specialization result to return the specialization field from the table **St_Group** comparing the **GroupCode** field from the table **St_Group** with the student **GroupCode** field value:

```
SELECT StudentName, GroupCode,
(SELECT Specialization FROM St_Group
WHERE St_Group.GroupCode=Students.GroupCode) FROM Students
```

You have noticed that there are two tables in the query and both of them have the **GroupCode** field that we use in the query. We have to specify what field we're talking about. To do this, the fields are preceded by the table name. Now our subquery is correlated, so it selects the values from the **St_Group** table but uses inside the field value from the **Students** table taken from the external query.

Using subqueries we can use **EXISTS**, **ANY** and **ALL** operators which correspondingly mean «Exists», «At least One» and «All» respectively.

Forms with subqueries

- *EXISTS*
- *ANY*
- *ALL*




They can be used as follows. The **EXISTS** statement is specified after **WHERE** followed by a subquery, usually correlated. The **EXISTS** statement returns Truth if there is at least one record as a result of subquery execution.

```
SELECT * FROM T
WHERE EXISTS (вложенный запрос)
```

For example, select students who have at least one phone in the table **PHONE_LIST**. Remind that each student may have several phones, or may not have one – between the entities **Student** and **Phone** the relationship is many-to-many, modality «can». In the table **PHONE_LIST** along with each phone number



the student's record book is stored which shows who owns the phone. In the external request, We can look for the student's record book numbers and students'



StudentId	PhoneType	Phone
345568	cell	07107534674
345571	cell	07300678543
345579	cell	07911365676
345583	cell	07931676657
345587	cell	07931676776

surnames in the external query. And in the subquery we have to check that for the current number of the student's record book there is at least one record in the phone table.

```
SELECT StudentID, StudentName FROM Students
WHERE EXISTS (SELECT * FROM PHONE_LIST WHERE
Students.StudentID=PHONE_LIST.StudentID)
```

As a result of the query we select the student's record book numbers and the names of students who have at least one phone.

If we add the NOT operator before EXISTS the query returns the list of students without the specified phone numbers. The external query returns all rows of the table **Students** for which no record exists in the table **PHONE_LIST**:

```
SELECT StudentID, StudentName FROM Students
WHERE NOT EXISTS (SELECT * FROM PHONE_LIST WHERE
Students.StudentID=PHONE_LIST.StudentID)
```

The ALL, ANY operators can be used if the subquery returns a single column whose values are compared to a given scalar value.

For example, let's find students who have got at least one min exam grade (2). Student grades are stored in the table **EXAM_RESULT** and the subquery returns all grades for each student from the table **EXAM_RESULT** then the external query selects all records from the **Students** table for which there is a 2.

```
SELECT StudentID, StudentName FROM Students
WHERE 2 = ANY (SELECT Grade FROM EXAM_RESULT WHERE
Students.StudentID=EXAM_RESULT.StudentID)
```

With the ALL operator we can find students who have got only 4 and 5. The subquery still returns all scores for each student from the **EXAM_RESULT** table but the external query selects those rows for which all >3.

ALL/ANY operators

*SELECT * FROM T WHERE <expression>
<comparison operation> ANY(subquery)*

*SELECT * FROM T WHERE < expression >
<comparison operation> ALL(subquery)*



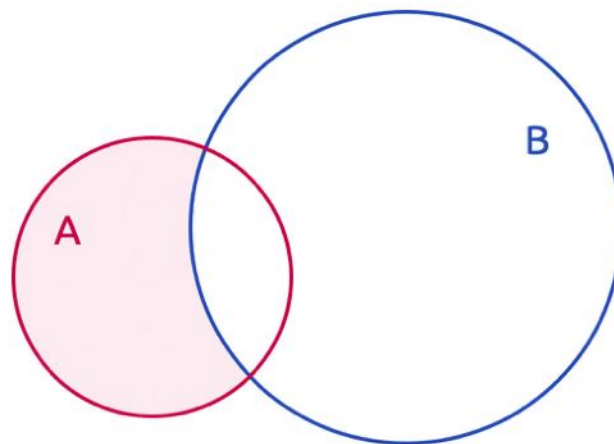
```
SELECT StudentID, StudentName FROM Students  
WHERE 3 < ALL (SELECT Grade FROM EXAM_RESULT WHERE  
Students.StudentID=EXAM_RESULT.StudentID)
```


5 Set-theoretic Operations

Strings in relational tables can be considered as sets and sets can be treated as multiple theoretic operations: union, intersection, difference, Cartesian product. With them we can combine the results of two queries, find the common part - the rows that are in both selections, find the difference - the rows that are in one select but not in the second and compare rows of several tables.

Set-theoretic operations

- *UNION*
- *INTERSECT*
- *EXCEPT*



In order to be able to perform these operations with queries they must be compatible by type: they must have the same number of columns and columns with the same order number must have compatible data types. What types of data

Set-theoretic operations

- *SELECT ... UNION SELECT...*
- *SELECT ... INTERSECT SELECT...*
- *SELECT ... EXCEPT/MINUS SELECT...*



are considered as compatible? When it is easy to convert one type to another. For example, any integer can be made fractional, short string long, etc. For example, a string cannot be converted to a number. So the column types are very important for these operations. But the field names don't have to match.

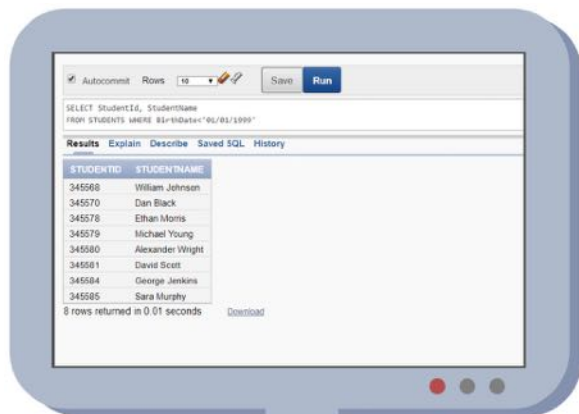
Let's create two selections. For example, the first query returns the **StudentId** and **StudentName** fields from the **Students** table with the **BirthDate** < '01/01/1999':

```
SELECT StudentId, StudentName FROM STUDENTS
WHERE BirthDate < '01/01/1999'
```

Another query returns us records where dates of birth **BirthDate** > '06/01/1998':

```
SELECT StudentId, StudentName FROM STUDENTS
WHERE BirthDate > '06/01/1998'
```

There are two columns in both queries - the first is numeric and the second is a string. Let's demonstrate the operations of unification, intersection and difference



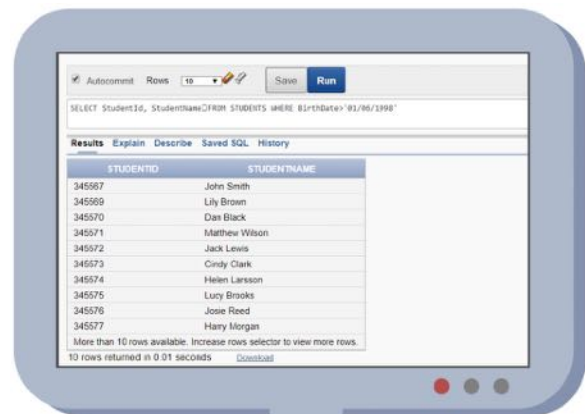
Autocommit Rows: 10 Save Run

SELECT StudentId, StudentName FROM STUDENTS WHERE BirthDate < '01/01/1999'

Results Explain Describe Saved SQL History

STUDENTID	STUDENTNAME
345568	William Johnson
345570	Dan Black
345578	Ethan Morris
345579	Michael Young
345580	Alexander Wright
345581	David Scott
345584	George Jenkins
345585	Sara Murphy

8 rows returned in 0.01 seconds Download



Autocommit Rows: 10 Save Run

SELECT StudentId, StudentName FROM STUDENTS WHERE BirthDate > '06/01/1998'

Results Explain Describe Saved SQL History

STUDENTID	STUDENTNAME
345567	John Smith
345569	Lily Brown
345570	Dan Black
345571	Matthew Wilson
345572	Jack Lewis
345573	Cindy Clark
345574	Helen Larsson
345575	Lucy Brooks
345576	Josee Reed
345577	Harry Morgan

More than 10 rows available. Increase rows selector to view more rows.
10 rows returned in 0.01 seconds Download

on this data.

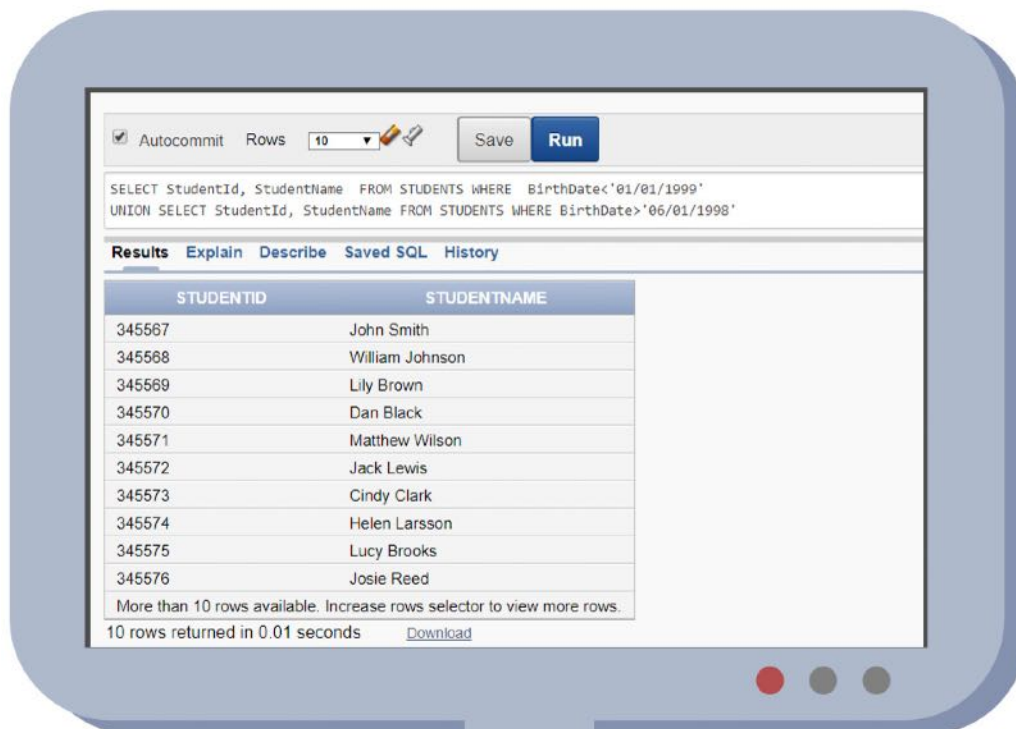
Start with the union operation. To do this we need to join two queries with the **UNION** operator:

```
SELECT StudentId, StudentName FROM STUDENTS
WHERE BirthDate < '01/01/1999'
UNION
SELECT StudentId, StudentName FROM STUDENTS
WHERE BirthDate > '06/01/1998'
```

Note that there are duplicated records in the initial queries but there are none of them as a result.

If you want to include all rows in the result regardless of duplicates **UNION ALL** is required instead of **UNION** :

```
SELECT StudentId, StudentName FROM STUDENTS
WHERE BirthDate < '01/01/1999'
UNION ALL
SELECT StudentId, StudentName FROM STUDENTS
WHERE BirthDate > '06/01/1998'
```



Now let's find the intersection of the two queries using the **INTERSECT** operator:

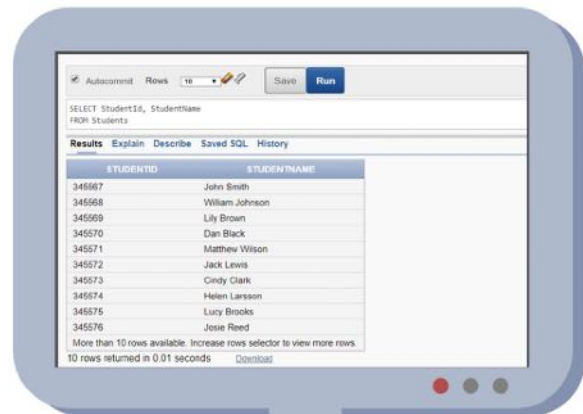
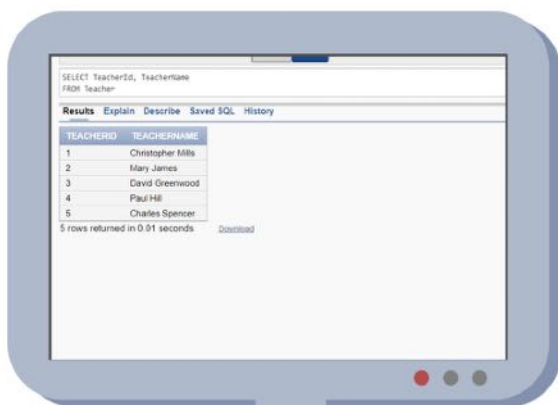
```
SELECT StudentId, StudentName FROM STUDENTS
WHERE BirthDate < '01/01/1999'
INTERSECT
SELECT StudentId, StudentName FROM STUDENTS
WHERE BirthDate > '06/01/1998'
```

The result of the intersection is four records - these are the ones where the date of birth from June 1, 1998 to January 1, 1999.

Records that are in the first query but not in the second query can be found by using the difference statement which is indicated in some systems by the word **EXCEPT**, in some – the word **MINUS**:

```
SELECT StudentId, StudentName FROM STUDENTS
WHERE BirthDate < '01/01/1999'
MINUS
SELECT StudentId, StudentName FROM STUDENTS
WHERE BirthDate > '06/01/1998'
```

Here is another example of union. Let's make a selection from the table **Teachers**: select the fields **TeacherId**, **TeacherName**. Can this query be joined with a query from the **Students** table? Each query has two columns, the first columns are integer, the second are string. There are 100 characters in the stu-

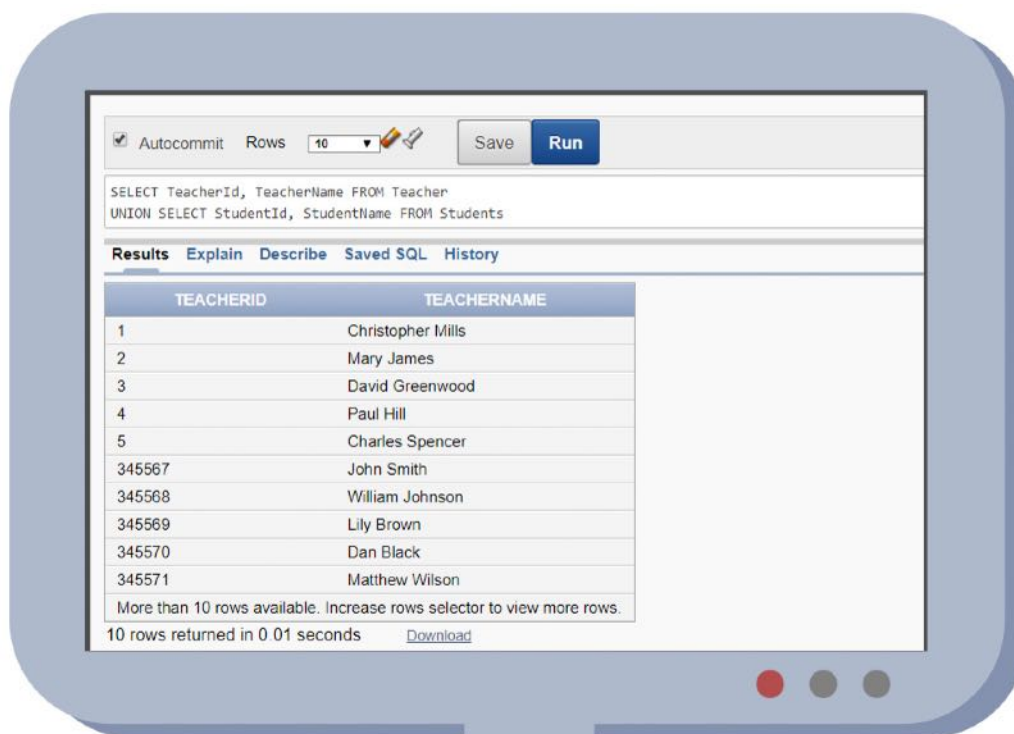


dent's name and only 50 in the teacher's name but nevertheless these data types are compatible.

```
SELECT TeacherId, TeacherName FROM Teacher
```

```
SELECT StudentId, StudentName FROM Students
```

As a result of this union we can get a table with columns named as in the



first of the joined tables and the length of the string field is the maximum of the lengths of the corresponding fields.

```
SELECT TeacherId, TeacherName FROM Teacher
```

```
UNION
```

```
SELECT StudentId, StudentName FROM Students
```

We see that the first column of the query is named `TeacherId`, а второй – `TeacherName`, it is not correctly. Rename the union query columns to `Id` and `Name`:

```
SELECT TeacherId AS Id, TeacherName AS Name
FROM Teacher
UNION
SELECT StudentId AS Id, StudentName AS Name
FROM Students
```

Now the first selection column is named `Id`, and the second - `Name`.

In order not to confuse who is who we can add another column - `Status` to the final query. We can use the `Teacher` status for teachers and for students - `Student`:

```
SELECT
TeacherId AS Id, TeacherName AS Name,
'Teacher' AS Status FROM Teacher
UNION
SELECT StudentId AS Id, StudentName AS Name,
'Student' AS Status FROM Students
```

The last set-theoretic operation we consider is the Cartesian product. If you execute a Cartesian product of two tables you get a set consisting of all possible pairs of combinations in which the first element of the pair is a row of the first table and the second element of the pair is a row of the second table. It is

Cartesian product

- `SELECT *`

FROM Table1, Table2



very capacious operation the number of records of the final query is equal to the product of the number of rows of the first and second tables. To perform this operation we need to list the tables with which we want to execute the Cartesian product after the keyword `FROM`.

Here is an example of a Cartesian product by multiplying the table with study groups and the table with courses. As a result, we get a table in which all courses are mapped to each group and, conversely, all groups are mapped to each course. Select three columns from the result: group code, group specialization and course name:

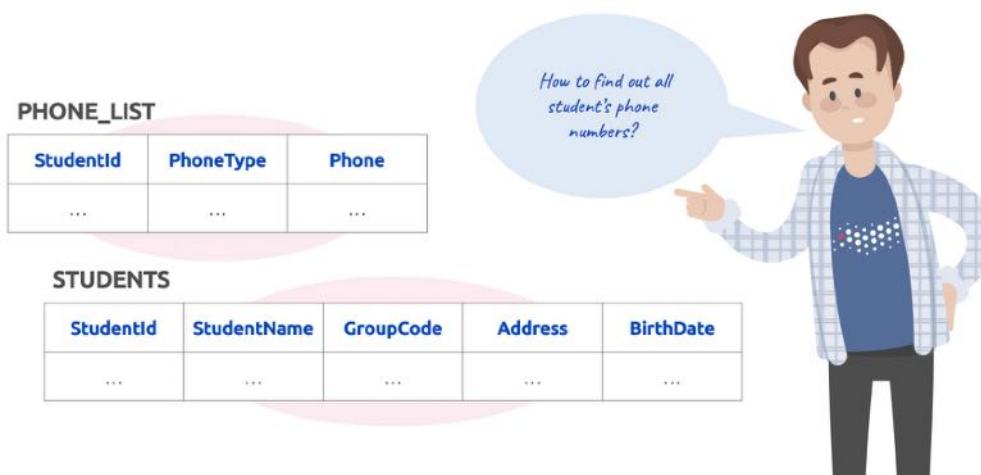
```
SELECT GroupCode, Specialization, CourseTitle
FROM ST_GROUP, COURSE
```

So, we have considered the basic set-theoretic operations that are often used when working with databases.

6 Join Tables

Let's remember that tables are not only to store information about objects but also to implement relationships. For example, we store student's phone numbers in a separate table. To keep the relationship we store together with the

Inner join



phone number the key to the Student object – the student's record book number – the **Student Id** field. How do we find out all the phone numbers belonging to a student? It is necessary to take student's record book number and find all the records from the table **PHONE_LIST** that correspond to this number. There is another way – join the tables **Student** and **PHONE_LIST** by the **Student Id** field.

For this purpose there is a special operation – **JOIN**. In general, it looks like this: after **FROM** in the **SELECT** statement the following data is specified: the name of the first table, the keyword **JOIN**, then the name of the second table, then the keyword **ON** after which the connection criteria are specified. There is another

STUDENTS + PHONE_LIST

StudentId	StudentName	GroupCode	Address	BirthDate	PhoneType	Phone
...

JOIN

- *SELECT **
FROM Table1 JOIN Table2
ON Table1.field1=Table2.field2
- *SELECT **
FROM Table1, Table2 WHERE
Table1.field1=Table2.field2



form for joining tables - via the Cartesian product with the following condition.

Let's join the table **Students** and the table **PHONE_LIST** by the matching values of the field **StudentId**. Note that this field occurs in both tables, so in the connection criteria we must specify the table name before the field name:

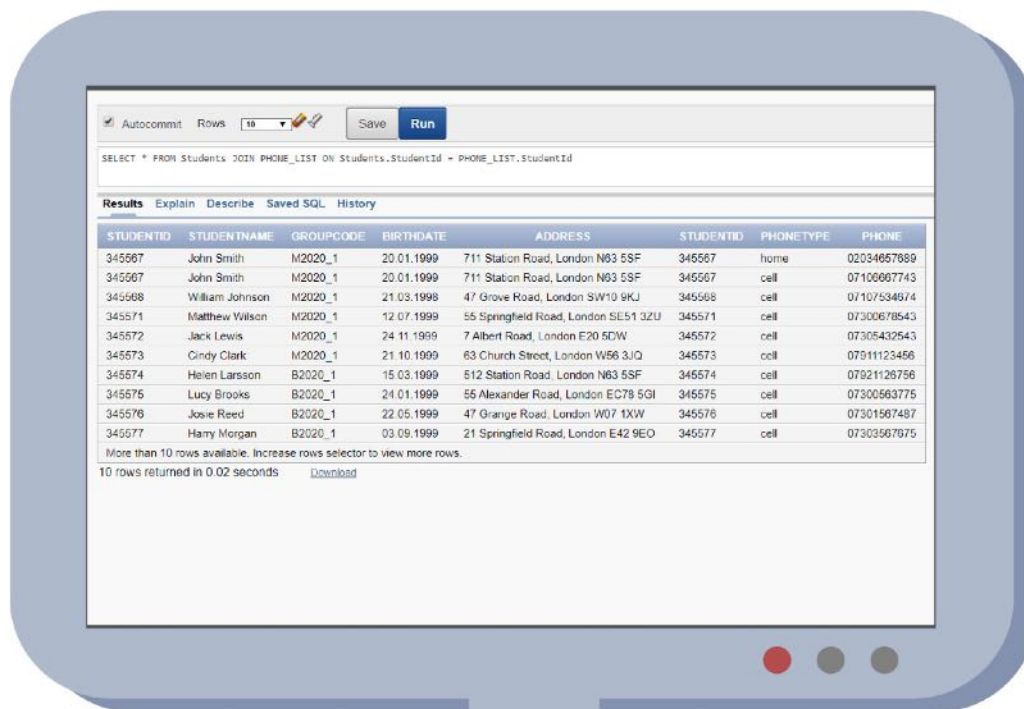
```
SELECT * FROM Students JOIN PHONE_LIST
ON Students.StudentId = PHONE_LIST.StudentId
```

Consider the same query in another form of record – via the Cartesian product with the following condition:

```
SELECT * FROM Students, PHONE_LIST
WHERE Students.StudentId = PHONE_LIST.StudentId
```

Now we see full information about students and their phones. Note that the selection did not include data about students who do not have a phone. For example, we do not see in the final selection Lily Brown with the student's record book number 345569.

A regular join is also called «inner» because only rows containing the values used in the connection condition that are «inside» of both the first and second tables are included in the selection.



If we need to include all students in the selection despite the absence of a phone number we can use the left join – **LEFT JOIN**. All rows of the first table are included in the left join. If they are related to the rows of the second table then they are displayed with the corresponding values and if not they are displayed with the undefined values.

```
SELECT * FROM Students LEFT JOIN PHONE_LIST
ON Students.StudentId = PHONE_LIST.StudentId
ORDER BY Students.StudentId
```

Now we see a list of all students and those who do not have a phone number, in place of its value is **NULL** – an unspecified value.

In addition to the inner and left, there are also right and full joins. **RIGHT JOIN** - returns rows from the right table, even if there is no match in the left; **FULL JOIN** - returns rows from both tables joining where the connection criteria have been met.

So far we have joined different tables. But sometimes it is necessary to join the table to itself. This join is often called as **SELF JOIN**. For example, select all possible pairs of students. How to do it? If we just join the table to itself we can't figure out where the query indicates which table. It is not possible to join this way. **SELF JOIN** is used to join a table to itself as if it were two different tables, temporarily renaming at least one of them.

For symmetry, we rename both inclusions of the **Students** table - **a** and **b**. We can join by the fields **StudentId** so that the pair does not include the same person twice.

STUDENTID	STUDENTNAME	GROUPCODE	BIRTHDATE	ADDRESS	STUDENTID	PHONETYPE	PHONE
345567	John Smith	M2020_1	20.01.1999	711 Station Road, London N63 5SF	345567	cell	07106667743
345567	John Smith	M2020_1	20.01.1999	711 Station Road, London N63 5SF	345567	home	02034657689
345568	William Johnson	M2020_1	21.03.1998	47 Grove Road, London SW10 9KJ	345568	cell	07107534874
345569	Lily Brown	M2020_1	17.05.1999	23 York Road, London SW41 2KA	-	-	-
345570	Dan Black	M2020_1	11.08.1998	701 Station Road, London N83 5SF	-	-	-
345571	Matthew Wilson	M2020_1	12.07.1999	55 Springfield Road, London SE51 3ZU	345571	cell	07300678543
345572	Jack Lewis	M2020_1	24.11.1999	7 Albert Road, London E20 5DW	345572	cell	07305432543
345573	Cindy Clark	M2020_1	21.10.1999	63 Church Street, London V66 3JQ	345573	cell	07911123456
345574	Helen Larsson	B2020_1	15.03.1999	512 Station Road, London N63 5SF	345574	cell	07921126756
345575	Lucy Brooks	B2020_1	24.01.1999	55 Alexander Road, London EC78 5GI	345575	cell	07300563775

```

SELECT a.StudentName, b.StudentName FROM STUDENTS
a JOIN STUDENTS b
ON
a.StudentId <> b.StudentId
ORDER BY a.StudentName, b.StudentName

```

In the resulting selection we see names for all possible pairs of students.

It is possible to join not only two tables, but three or more - as much as is necessary for the data selection. To join several tables do it sequentially: specify the name of the first table, then the keyword **JOIN**, the name of the second table and the connection criteria, then write **JOIN** again, the name of the third table and the connection criteria.

Types of joins

- *JOIN*: returns records when there is at least one match in both tables
- *LEFT JOIN*: returns records from the left table even if there are no matches in the right
- *RIGHT JOIN*: returns records from the right table even if there are no matches in the left
- *FULL JOIN*: returns records from both tables



SELF JOIN

- ~~*SELECT **~~
~~*FROM S JOIN S ON S.field1 = S.field1*~~
- *SELECT a.*, b.**
FROM S as a
JOIN S as b
ON a.field1 = b.field2

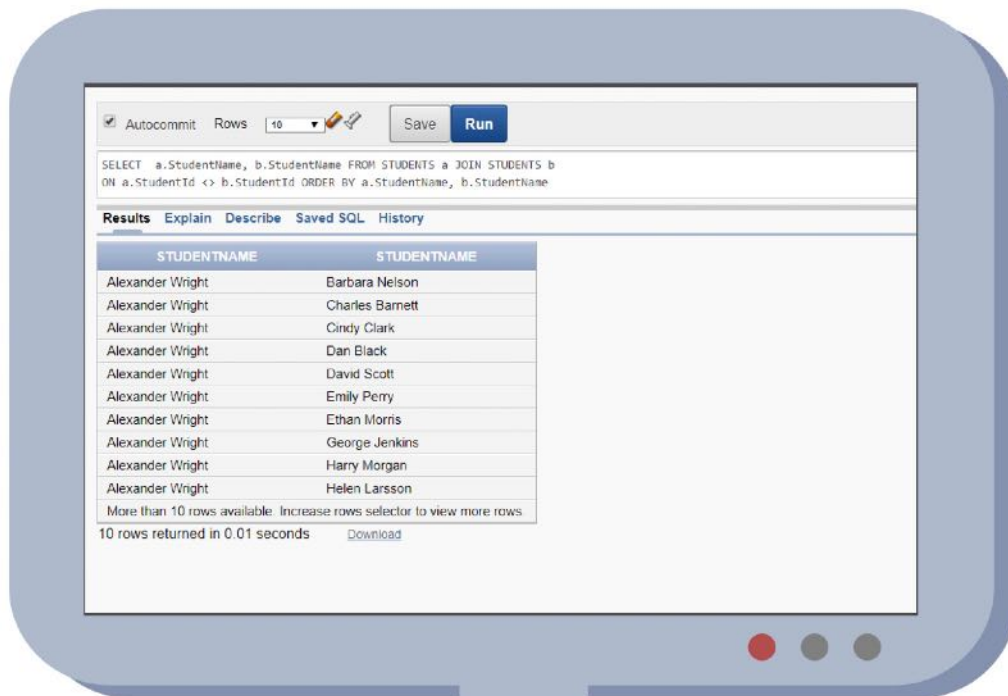


Consider the table describing the exam – it contains the exam number, the group code, **CourseId** - the index for the course to be taken, **TeacherId** - the index for the teacher who will take the exam, the classroom and the date of the exam:

```
SELECT * FROM EXAM_SHEET
```

To select the exam number, the name of the subject and the name of the teacher as a result of the query it is necessary to join the table **EXAM_SHEET** with the tables **COURSE** and **TEACHER**. In this query the **COURSE** table is first joined by the **CourseID** field, then **TEACHER** by the **TeacherID** field. The order of specifying these tables for the join operation is not important in this case.

```
SELECT ExamSheetId, CourseTitle, TeacherName, ClassRoom, ExamDate  
FROM EXAM_SHEET JOIN COURSE  
ON EXAM_SHEET.CourseId = COURSE.CourseId  
JOIN TEACHER ON EXAM_SHEET.TeacherId = TEACHER.TeacherId
```



The result of the double join is shown in the figure.

In this way we have learned to create complex queries specifying many criteria.

Union of three tables

- *SELECT **
FROM Table1
JOIN Table2
ON Table1.field1 = Table2.field2
JOIN Table3
ON Table2.field3 = Table3.field4
- *SELECT **
FROM Table1, Table2, Table3
WHERE Table1.field1 = Table2.field2
AND Table2.field3 = Table3.field4



EXAM_SHEET

ExamSheetId	GroupCode	CourseId	TeacherId	ClassRoom	ExamDate
...

Autocommit Rows: 10 Save Run

```
SELECT ExamSheetId, CourseTitle, TeacherName, ClassRoom, ExamDate FROM EXAM_SHEET JOIN COURSE
ON EXAM_SHEET.CourseId = COURSE.CourseId JOIN TEACHER ON EXAM_SHEET.TeacherId = TEACHER.TeacherId
```

Results Explain Describe Saved SQL History

EXAMSHEETID	COURSETITLE	TEACHERNAME	CLASSROOM	EXAMDATE
8	Data storage	David Greenwood	2410	27.01.2020
4	Data storage	Paul Hill	2411	26.01.2020
1	Data storage	Christopher Mills	2408	25.01.2020
9	Machine Learning	David Greenwood	2411	28.01.2020
5	Machine Learning	Charles Spencer	2412	24.01.2020
2	Machine Learning	Mary James	2408	27.01.2020
10	Data processing and analysis	Paul Hill	-	28.01.2020
6	Data processing and analysis	Christopher Mills	2410	25.01.2020
3	Data processing and analysis	David Greenwood	2410	28.01.2020
7	Artificial intelligence	Mary James	2410	27.01.2020

10 rows returned in 0.02 seconds [Download](#)