DIGITAL CULTURE
ITMO UNIVERSITY

Relational Databases

High School of Digital Culture
ITMO University
dc@itmo.ru

Contents

1 Database Management System

Various information systems surround us in the modern world. These information systems simplify solving a variety of tasks that recently were believed to be very time-consuming. Information systems help us to communicate in social networks, to make purchases or make an appointment with a doctor. At the heart

of any information system there is a database that stores all the necessary information. In our lecture on primary data processing, we used spreadsheets as a tool. But when the amount of data increases and the data has a complex structure, the tools should be very different.

A database is an organized set of data that is generally stored and processed in digital format. The amount of accumulated data can be enormous. But these volumes are easier to deal with when the data has a clear structure, and the typical objects in this data, their elements, or components as well as the relations between them are identified. Such data is called structured data. Often, the data is simply a collection of heterogeneous documents, so there is no well-defined uniform structure in the data. Such data is called unstructured data and is much more difficult to work with. We call the data semi-structured in case at least some elements of the structure are found in it.

As a matter of fact, any information system includes not only data, but also a SOFTWARE aimed at executing specific tasks in the subject area. Still, in any information system you need to be able to describe the structure of the data, to find the data quickly, to modify and to save it, to allow multiple users to work

with it simultaneously and to ensure that the data is not lost or damaged. For these operations, it is logical to use a special SOFTWARE package, that is the database management system.

1.1 What is DBMS?

DBMS can be defined as a set of means and tools designed to manage the database and organize interaction with it. Database management system (DBMS) interacts with final users, applications, and the database itself to collect, store, process and to analyze data. DBMS provides the following features to the user:

1. Persistent storage. Like the file system, DBMS supports the storage of large amounts of data, that exists independently of any processes involving that data. But the advantage of DBMS compared to the file system is data storage in structures that support efficient access to very large amounts of information.

2. Programming interface. DBMS allows a user or an application to access and to modify the data by means of a high-level query language. Again, the advantage of DBMS over the file system is the flexibility to manage stored data in much more complex ways than just reading or writing files.

3. Transaction management. DBMS supports parallel access to data, i.e. simultaneous access to several different processes (called "transactions"). To avoid some undesirable consequences of simultaneous access, the DBMS supports isolation, giving the user the appearance that transactions are executed one at a time; and atomicity, i.e. the requirement that transactions are to be executed completely or not executed at all. DBMS also supports durability, guarantying that data from the completed transactions will survive permanently, as well as providing the database the ability to recover from failures or errors of various types.

Modern DBMS are divided into two classes:

- relational;
- NoSQL.

To store structured data, relational databases are the most common, where stored information is presented in table-oriented view. We should say, that the relational databases are not just flat tables – data from different tables can be linked, there are some integrity constraints on the data, etc.

Relational databases date back to the 70s of the last century, when the first research prototype of a relational database was developed by IBM under the name

System Relational or System R. System R was the first implementation of Structured Query Language (SQL), which is now the standard for relational databases. Since then, many databases based on the relational model have appeared.

Although all database management systems perform the same main tasks, their functions and capabilities may differ significantly.

The most important parameters are performance and reliability. And, needless to say, the price is also important! The cost of commercial versions of some systems is tens or hundreds of thousands of dollars. But almost all commercial systems have free distributed versions, usually with incomplete functionality and with ability to work only on a limited amount of data, but still small commercial systems can be developed on them.

When comparing various popular databases, one should consider whether this particular database is user-friendly and scalable, and make sure that it integrates well with other products or software that are already used in the system to solve the desired tasks. Various DBMS are documented more or less thoroughly. Technical support is also provided in different ways. One should also pay attention to the possibilities of procedural expansion of the SQL and to a supported set of user interfaces provided to develop applications.

The main differences between DBMS

- performance and reliability;
- is it commercial or free software;
- ability to integrate with other systems;
- documentation and technical support;
- built-in features;
- supported set of user interfaces.

1.2 The Most Popular DBMS.

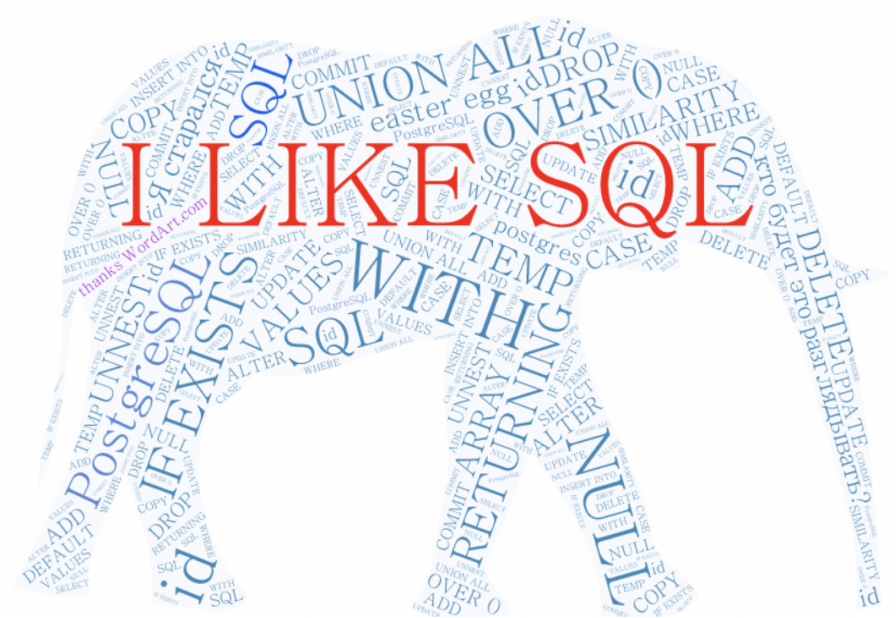
Currently, the most popular relational database is Oracle. Oracle was founded in 1977. The DBMS Oracle developed by this company is considered to be the first commercial DBMS supporting Structured query language (SQL), and one of the first relational DBMS. According to a number of ratings, Oracle is the leading company in databases development for large corporations.

Microsoft SQL Server, focused mainly on the Windows operating system, is also very popular on the market.

IBM distributes a DBMS called DB2, which is a commercial extension of System R DBMS.

In addition to commercial DBMS there is a number of open source software, such as SQLite, PostgreSQL, and various modifications of MySQL, for example, MariaDB. The leader among them is PostgreSQL due to the support of complex structures as well as a wide range of built-in functions and user-defined data types.

Despite the fact that there are many systems and they have some differences, if you learn how to work with one DBMS, it is not difficult to use a different one.



2 Data Structure

The first step in Database design is to analyze the subject area and to identify the requirements of individual users. Different database users potentially aim at different tasks to be solved on their own data sets. To compile all representations of the database that every user has in mind, a generalized schema is constructed that is called a database schema. Where to start designing the database schema?

First, a generalized informal description is to be created. This description is done by means of natural language, mathematical formulas, and other tools that are clear to all people working on data design. Let's consider the product sales database. The goal is to store information about the products, i.e. product name, price and product item number; information about customers, namely, their name, surname and patronymic (if any) and their home address; and information about orders: who ordered that particular product, when and in what quantity. To make our scheme clear and unambiguous, we assume that each product is ordered separately, i.e. one order contains only one product type.

The next step is to identify the data structure. It is necessary to define the objects to store the information about, to determine their properties, called

attributes, and to describe the relationships that associate the objects with each other.

To describe the data more formally and clearly, an Entity-relationship diagram is often used, or ER diagram, graphically representing the logical structure of the data.

The basic concept in ER-diagram construction is the entity. An entity is an object either real or imagined, a part of the world, or an important event in our subject area. As a rule, each subject area contains a set of entities. Each entity must have a unique name. In our example, products and customers are the entities. In the diagram entities are shown by rectangles.

Thus, an entity describes a class or a set of objects that corresponds to a certain type. An entity instance is a specific object of this type. For example, a Product entity is represented by the instances Pen, Pencil, and Notebook.

Individual characteristics or properties of the entities are called attributes. For each attribute a unique name (within this entity) and a data type must be specified.

For example, the Product entity attributes are product item number, product name, and price. The Customer entity is described by the attributes Last name, First name, Patronymic (if any), Address. In the diagram, we surround the attributes with ovals and associate them with entities by undirected arcs.

We need to distinguish between entity instances of every type. One (or more) attributes that uniquely identify the object are called key attributes. The key attribute is a very important concept for data design. The Product item number can be the key for the entity Product.

Sometimes a natural key cannot be found, for example, there can be namesakes among customers, and the possibility that they live at the same address cannot be excluded either. Then, for the convenience of identification, an artificial or surrogate key is used, that is the Code of the customer. A key attribute is usually underlined with a solid line in the diagram.

Sometimes a composite key is used, that is the combination of several attributes. For example, if we stored passport data of customers, a composite key would be used, namely, the Series and the passport Number Sometimes an entity can have several keys, e.g. in our case a key can be either a Customer Code or a Series + Passport number.

If an entity has more than one key, the most important one should be selected, normally it is the key that is more often searched for. This key is declared a primary key.

Typically, entities interact with each other. To indicate this, the diagrams show relationships. Relationships are connections between the entities. Unlike entities relationships cannot exist on their own without specifying the entities that participate in the relationship. For example, if a customer has ordered a

specific item, an Order relationship appears.

Relationships can have their own attributes. For example, the Order relationship can be associated with the attributes Item Quantity and Order Date. To identify relationships the keys of the related entities are used and, sometimes, the attributes of the relationship. Each relationship in the diagram should have a name. The relationships are shown by diamonds shapes in the diagram.

2.1 Types of Relationships

Multiple entities can be associated in a relationship. We considered a binary relationship between the Product and Customer entities.

If we stored information about the stores where the products have been purchased, then the relationship would be ternary. However, such relationships are difficult to store, and therefore are preferably avoided.

There are also recursive relationships, when objects are associated with other objects of the same type. For example, recursion is usually used to implement hierarchy: Employee -> Manager

Let's consider binary relationships. They are divided into three types depending on the number of participating objects.

One-to-one	1:1
One-to-many	1:N
Many-to-many	M:N

The number of objects determines the degree (or the cardinality) of the relationship. If many objects of one entity type are associated with many objects of another entity type, the relationship is called many-to-many. The relationship degree is indicated graphically at the end of the arc, namely, the multiplicity of the relationship is depicted as a "fork" at the end of the arc. In our example, each customer can order multiple products and each product is to be purchased by multiple customers.

If many objects of one entity type are linked with only one object of another entity type, then we speak about many-to-one relationship. This relationship is shown by a straight line. For example, Products are sold in a Store. Each store can place many products, but each product is sold in exactly one store.

It should be noted the objects can be associated with other objects by several different relationships. Let's imagine that each customer is asked to choose the Favorite product, then for each customer it is exactly the one. But one and the same product can be favorite among several customers forming the connection of many to one. For example, in our scheme the Product and the Customer objects are linked by two relationships, namely the Order and the Favorite product.

The relationship one-to-one is established if an object of one entity type comes into relationship exactly with one object of another entity type. For ex-

ample, let's add a new object to our scheme, that is the store Manager. In each store there is exactly one Manager, and each Manager manages exactly one store.

It remains to mention the modality of relationships. It indicates whether a relationship is mandatory or optional. An optional relationship means that an object of one type can either be associated with one or more object instances of another entity type or remain unrelated to any instance.

The modality of the relationship is also shown graphically. The optional link is marked with a circle at the end of the link. For example, some products have left yet unpurchased by anyone. Then the relationship in the direction from the Products to the Customer is optional.

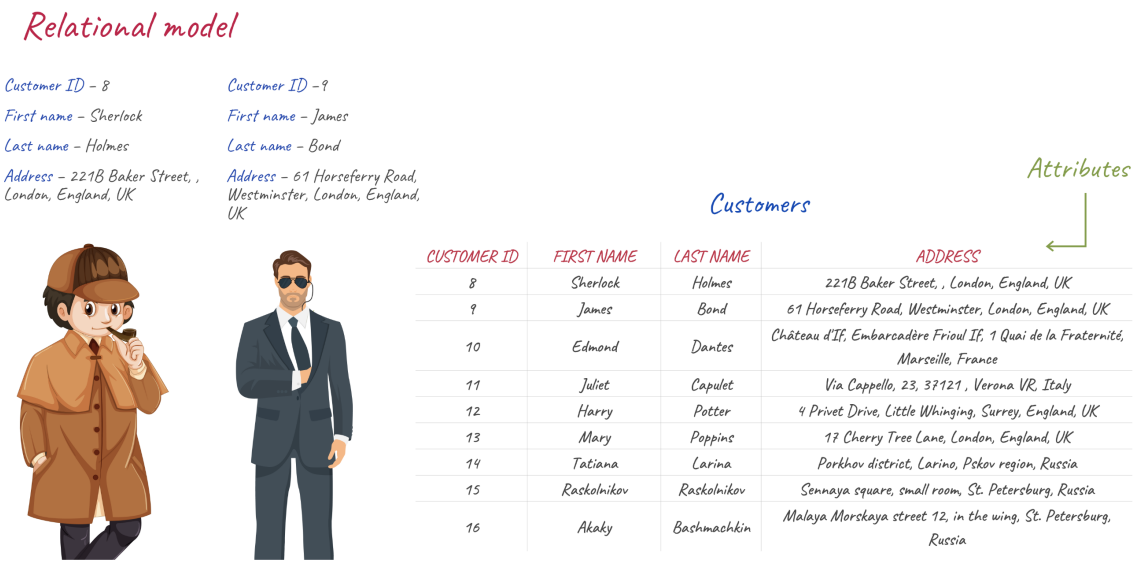
Mandatory relationship means that an object must be matched with at least one instance of another object type. For example, each customer is supposed to order at least one product item.

Mandatory relationship is represented by a straight line. To sum up, we have learned how to describe data in terms of the entity relationship model, which shows us the essence of the database being developed. Now we need to understand how to convert ER-diagrams into storage structures.

3 Tables

After the ER-diagram construction it is necessary to select the DBMS in which the database will be implemented and convert the described scheme into specific storage structures.

The relational model is the most widely used for structured data. In 1970, Edgar Codd wrote an article introducing a relational data model for the first time. The main idea of this article was that all the data visible for users can be represented in table view. Technically speaking, the relational data model is based on the concept of a relationship, defined by a list of its elements and their corresponding values. The relational data model represents information as a set of interrelated tables that are called relationships. Tables consist of columns and rows. Columns correspond to object attributes, and rows correspond to individual objects instances. At the intersection of a row and a column there is a specific attribute value that belongs to some allowable value set.



How do we transfer from the object-relationship model to a relational one? In the ER model, we described information about entities and the relationships between them. It is perfectly clear how to deal with entities, or objects, i.e. each object turns into a separate table. The name of the object becomes the table name, and this name must be unique within one database. The attributes turn into the column names.

The table header consists of attribute names. Each column corresponds to one attribute, and the data type is defined for each column.

Each table row corresponds to a particular object instance. The key of the entity becomes the key of the table. How are associations between the objects represented in a relational model? They are also connected in a relationship.

The schema of this relationship is made up of the key attributes of the objects

involved in the relationship. To make a one-to-one relationship, you can combine the columns that correspond to the attributes of both objects into a single table. A key of one of the objects is selected to be the relationship key.

Store name	Address	Manager name	Phone
BakerStreetBoys	221b Bakerstreet	John Smith	020 7123 1234
Sweet shop	201 Waterloo str.	Tim Horton	204 2914276

To implement the relationship "Manages the store" we make a single table, where the information about the store and its Manager is stored. If many objects of one entity type are related to only one object of another type, then it is a many-to-one relationship.

An example of such a relationship is the "Favorite product" relationship between the customer and the Product. Two tables are required to represent the Customers and the Products.

CUSTOMER ID	FIRST NAME	LAST NAME	ADDRESS	FAVOURITE PRODUCT
8	Sherlock	Holmes	221B Baker Street, , London, England, UK	2
9	James	Bond	61 Horseferry Road, Westminster, London, England, UK	3
10	Edmond	Dantes	Château d'If, Embarcadère Frioul If, 1 Quai de la Fraternité, Marseille, France	1
11	Juliet	Capulet	Via Cappello, 23, 37121 , Verona VR, Italy	4
12	Harry	Potter	4 Privet Drive, Little Whinging, Surrey, England, UK	5
13	Mary	Poppins	17 Cherry Tree Lane, London, England, UK	7
14	Tatiana	Larina	Porkhov district, Larino, Pskov region, Russia	4
15	Raskolnikov	Raskolnikov	Sennaya square, small room, St. Petersburg, Russia	8
16	Akaky	Bashmachkin	Malaya Morskaya street 12, in the wing, St. Petersburg, Russia	6

To implement the relationship, the Products table key is added to the Customers table. And here is an example of the Customers table, with the added column Product Code indicating a favorite product. The Customer Code remains the key of the table.

All we have to do is to implement the most complex, but the most common relationship that is many-to-many. Between the Customers and the Products there is exactly such a relationship named Orders.

Products

Product ID	Product name	Price ($)
1	Notebook	0,20
2	Ruler	0,50
3	Textbook	0,30
4	Apple	0,80
5	Orange	0,40
6	Pencil	0,20
7	Pen	0,10
8	Peach	0,30

Customers

CUSTOMER ID	FIRST NAME	LAST NAME	ADDRESS
8	Sherlock	Holmes	221B Baker Street, , London, England, UK
9	James	Bond	61 Horseferry Road, Westminster, London, England, UK
10	Edmond	Dantes	Château d'If, Embarcadère Frioul If, 1 Quai de la Fraternité, Marseille, France
11	Juliet	Capulet	Via Cappello, 23, 37121 , Verona VR, Italy
12	Harry	Potter	4 Privet Drive, Little Whinging, Surrey, England, UK
13	Mary	Poppins	17 Cherry Tree Lane, London, England, UK
14	Tatiana	Larina	Porkhov district, Larino, Pskov region, Russia
15	Raskolnikov	Raskolnikov	Sennaya square, small room, St. Petersburg, Russia
16	Akaky	Bashmachkin	Malaya Morskaya street 12, in the wing, St. Petersburg, Russia

Orders

Date	Customer ID	Product ID	Product Name	Price	Amount
23.02.19	7	5	Orange	0,40	2
26.02.19	2	2	Ruler	0,50	22
15.02.19	5	5	Orange	0,40	37
30.10.19	3	3	Textbook	0,30	33
03.05.19	6	5	Orange	0,40	88
22.10.19	8	2	Ruler	0,50	55
17.07.19	1	1	Notebook	0,20	67
09.02.19	7	3	Textbook	0,30	77
03.07.19	4	1	Notebook	0,20	36
18.09.19	10	4	Apple	0,80	26
01.01.19	1	1	Notebook	0,20	66

To implement a many-to-many relationship, a separate table is created, where columns store the key attributes of the related objects. Let's copy the columns Customer Code, Product item number. Let's name the new table Orders. Do not forget to add the relationship's own attributes, that is the product quantity and the order date.

Thus, any objects and relationships of the real world can be presented in the form of a table.

4 SQL DDL

To create a table, special commands are to be used. In relational DBMS for this purpose a structured query language (SQL) is used that is widespread and standardized programming language.

All SQL commands are divided into several groups depending on their purpose:

1. Data Definition Language, DDL: these type of commands are used to create the database itself, as well as tables and other necessary objects.

2. Data Manipulation Language, DML: by these commands, tables are filled with data, modified or deleted; besides the data could be searched for given particular criteria.

3. Data Control Language, DCL: these operators are needed to provide users with access to the database.

4. Transaction Control Language, TCL: the transaction (and concurrency) control mechanism provides the possibility of simultaneous work of several users with the database ensuring that they do not interfere with each other. Transactions also allow you to restore the correct state of the database after possible failures.

Data Definition Language, DDL.

The first thing to create is the database itself, that is, to allocate some memory area to store all database objects. The database is created by the `CREATE` command.

In some DBMS another important object is used, namely, a schema containing all database objects. In this case, the database is considered to be a set of schemas.

When creating a database, a name must be specified. In addition you can specify many parameters, such as storage location and initial storage size, growth rate, etc.

After the database itself is created, you can start creating tables. Tables in the database are also created using the `CREATE` command. A table name and a description of the columns must be specified.

To define each column of the table, its name and data type is to be specified. The column name should preferably correspond to the logic of data organization. For example, the name of the column containing the Product Code could be `Id` or `Product_id`.

The main types to store the data are text, numbers, and time. To store data in XML format, monetary, bit and spatial data, for organizing a hierarchical structure, as well as different time formats special data types are used. Data types description may differ slightly in different DBMS, our examples are given based on PostgreSQL.

Integers or real numbers are to be stored. For example, in the Products table there is an attribute Product Id that is a whole number. So, the integer data type is suitable for it. But the number in the column Price must be real with two decimal places, that is `Numeric(10.2)`. Field parameters are the total number of digits both before and after the decimal point and the number of digits after the decimal point. The field product name is a string. The length of the string could be fixed, i.e. by specifying the length we limit the maximum string size, for example, `char(50)`. In case the string is shorter than 50 characters, it will be padded with spaces, and if it is longer, only the first 50 characters are inserted as value of the column, the rest will be truncated. In case the length of the attribute varies depending on the content, a variable length data type is to be used, namely,

`varchar(n)`. In that case the field will not be padded with spaces, and the string length will be stored along with the string value.

```
CREATE TABLE Products (
Product_ID integer,
Product_NAME varchar(25),
Price numeric(10.2)
)
```

Thus, we have defined the columns of The Products table.

Optionally, you can specify the constraints that the data must satisfy. The purpose of constraints is to prevent incorrect data from entering the database.

By default (unless otherwise stated), table fields can contain an unset i.e. null, value. However, only some fields are possible to leave unset, otherwise it makes no sense to include this data item in the database. First, the Product must have a name and an article. And the Price field is allowed to have null values.

```
CREATE TABLE Products (
Product_ID integer NOT NULL,
Product_NAME text NOT NULL,
Price numeric(10.2) NULL
)
```

The primary key of the table is a very important constraint. Its values must be set and it must be unique for each row. Only one primary key is to be specified in the table.

```
CREATE TABLE Products (
Product_ID integer PRIMARY KEY,
Product_NAME text NOT NULL,
Price numeric(10.2) NULL
)
```

If other fields of the table must be unique, `UNIQUE` is to be added in the command. In case all products must have unique names, the description of the created table would look the following way:

```
CREATE TABLE Products (
Product_ID integer PRIMARY KEY,
Product_NAME text UNIQUE NOT NULL,
Price numeric(10.2) NULL
)
```

A combination of several fields can be declared unique. For example, a constraint could be made that the table should contain no products with the same name and the same price.

```
CREATE TABLE Products (
Product_ID integer PRIMARY KEY,
Product_NAME text NOT NULL,
Price numeric(10.2) NULL,
UNIQUE(Product_NAME, Price)
)
```

You can add a field validation rule to specify some conditions that all fields must meet. For example, we indicate that the product prices are always positive.

```
CREATE TABLE Products (
Product_ID integer PRIMARY KEY,
Product_NAME text NOT NULL,
Price numeric(10.2) NULL CHECK(Price>0)
)
```

Now we can create another table named Customers.

```
CREATE TABLE Customers (
Customer_ID integer PRIMARY KEY,
Cust_Last_NAME text NOT NULL,
Cust_First_NAME text NOT NULL,
Cust_Address text NULL
)
```

The last table to create a table `Orders`, storing the information about the customer, the product and the ordered product quantity.

```
CREATE TABLE Orders (
Order_ID integer PRIMARY KEY,
Customer_ID integer,
Product_ID integer,
Quantity integer,
Order_date date
)
```

This table is needed to establish the relationship between the Customers and the Products. But what happens if we make a mistake in the value of the `Product_ID` field? For example, let's try to input information about the purchase of an item with a code not existing in the Products table? In the Products table

we will find no such items. To ensure data integrity, only those `Product` item codes are allowed to input in the table Orders that are already included in the Product table. In this case, a foreign key constraint will help us. We will link the fields of the `Orders` and `Products` tables together to prevent incorrect values from being placed in the table.

```
CREATE TABLE Orders (
Order_ID integer PRIMARY KEY,
Customer_ID integer,
Product_ID integer REFERENCES Product(Product_ID),
Quantity integer,
Order_date date
)
```

Similarly, we will link the `Orders` table with the `Customers` table

```
CREATE TABLE Orders (
Order_ID integer PRIMARY KEY,
Customer_ID integer REFERENCES Customers (Customer_ID),
Product_ID integer REFERENCES Product(Product_ID),
Quantity integer,
Order_date date
)
```

The foreign key constraint is used to link fields from different tables. The linked fields must be of the same type, and the field which is referred to must have unique values or must be the primary key.

We have learned how to create tables. If the table is not needed any more, it can be deleted by the `DROP TABLE` command. In that case both the data if there was any in the table and the table structure are deleted.

With `ALTER TABLE`, you can change the table structure, that is to add or remove columns or integrity constraints. For example, to change the column `Customer_ID` of the `Orders` table we add a constraint that all values should not be null.

```
ALTER TABLE Orders MODIFY Customer_ID integer not null;
```

5 SELECT

We learned how to create tables, now we need to fill them with data. The `INSERT` statement help us to do this. Let's insert a row in the Products table:

```
INSERT INTO Products VALUES(8, 'Pencil', 0.3)
```

If you want to modify the record the `UPDATE` command is used:

```
UPDATE Products SET Product_Name='Color pencil'
WHERE Product_ID=8
```

All the constraints specified in the table description are checked when the rows are inserted and modified. For example, if you try to add the following record to the Products table, an error will occur, because the product with Product Item number equal to 1 already exists, and the Product item number field is declared to be a primary key, and therefore must contain only unique values.

```
INSERT INTO Products VALUES(8, 'Felt Pen'', 1.35)
```

To delete a record, the DELETE command is used:

```
DELETE FROM Products WHERE Product_ID=8
```

Be careful when deleting rows. If you run DELETE without clauses, all rows in the table will be removed.

```
DELETE FROM Products
```

To search for data in tables, there is only one command or statement `SELECT`, that returns a set of records from database tables meeting the specified criteria. `SELECT` results in one or more rows or columns from one or more database tables.

The simplest query is to show all the rows in the table.

```
SELECT * FROM Products
```

Product_ID	Product_Name	Price
1	Notebook	0.20
2	Ruler	0.50
3	Textbook	
4	Apple	0.80
5	Orange	0.40
6	Peach	0.20
7	Pen	

Processing of the `SELECT` statement starts with the `FROM` clause, which specifies the tables to select the data from. Columns are specified after the statement `SELECT`. `*` means that all columns of the table should be displayed

If you want to display only some columns, or change their order, you must specify their names explicitly.

```
SELECT Product_ID, Product_Name FROM Products
```

Product_ID	Product_Name
1	Notebook
2	Ruler
3	Textbook
4	Apple
5	Orange
6	Peach
7	Pen

To rename the displayed columns, specify the new names after the column name.

```
SELECT Product_ID ID, Product_Name Name FROM Products
```

ID	Name
1	Notebook
2	Ruler
3	Textbook
4	Apple
5	Orange
6	Peach
7	Pen

The following query displays the column with price values from the Products table

```
SELECT Price FROM Products
```

Price
0.20
0.50
-
0.80
0.40
0.20
-

We see that the same price 0.2 is displayed twice, because two products have the same prices, i.e. Notebook and Peach.

If you want to see only unique rows, the `DISTINCT` keyword should be added to the statement.

```
SELECT DISTINCT Price FROM Products
```

Price
0.20
0.50
-
0.80
0.40
-

The expressions are built by combining column names and constants. For example, let's calculate the total price of 100 pieces of every product.

```
SELECT Product_ID, Product_Name, Price*100 as Price_Per_100
FROM Products
```

Product_ID	Product_Name	Price_Per_100
1	Notebook	20
2	Ruler	50
3	Textbook	
4	Apple	80
5	Orange	40
6	Peach	20
7	Pen	

Use `ORDER BY` clause to sort the selected set, specifying the sorting criteria after `ORDER BY`. By default, the result set is sorted in ascending order.

```
SELECT * FROM Products ORDER BY Price
```

Product_ID	Product_Name	Price
1	Notebook	0.20
6	Peach	0.20
5	Orange	0.40
2	Ruler	0.50
4	Apple	0.80
3	Textbook	
7	Pen	

Instead of the column name, you can specify its ordinal number in the result set. The `Price` column is the third in the table, so you can construct the expression the following way:

```
SELECT * FROM Products ORDER BY 3
```

Product_ID	Product_Name	Price
1	Notebook	0.20
6	Peach	0.20
5	Orange	0.40
2	Ruler	0.50
4	Apple	0.80
3	Textbook	
7	Pen	

If you want to sort the result set in descending order, the keyword `DESC` is used after the sorting criterion:

```
SELECT * FROM Products ORDER BY 3 DESC
```

Product_ID	Product_Name	Price
4	Apple	0.80
2	Ruler	0.50
5	Orange	0.40
1	Notebook	0.20
6	Peach	0.20
3	Textbook	
7	Pen	

You can add more sorting criteria, for example, sort the rows with the same price by Product item number.

```
SELECT * FROM Products ORDER BY 3 DESC, Product_ID
```

Product_ID	Product_Name	Price
4	Apple	0.80
2	Ruler	0.50
5	Orange	0.40
1	Notebook	0.20
6	Peach	0.20
3	Textbook	
7	Pen	

So far, the result set displayed all the rows of the table, only the rows order was changed or some additional attributes were added. Now let's learn how to limit the displayed rows by specifying the selection conditions. A simple search condition is composed of column names, constants, and comparison operations.

```
SELECT * FROM Products WHERE Price<0.5
```

Product_ID	Product_Name	Price
1	Notebook	0.20
5	Orange	0.40
6	Peach	0.20

There can be several selection conditions. Simple clauses can be linked by logical operators `AND, OR, NOT`:

```
SELECT * FROM Products WHERE Price<0.5 AND Price>0.3
```

Product_ID	Product_Name	Price
5	Orange	0.40

```
SELECT * FROM Products WHERE Price>=0.5 OR Price<0.3
```

Product_ID	Product_Name	Price
1	Notebook	0.20
2	Ruler	0.50
4	Apple	0.80
6	Peach	0.20

```
SELECT * FROM Products WHERE Price>0.4 AND NOT Product_ID=2
```

Product_ID	Product_Name	Price
4	Apple	0.80

A comparison with unset values denoted by the `NULL` keyword could be used. In that case a special operator `IS` is needed

```
SELECT * FROM Products WHERE Price IS NULL
```

Product_ID	Product_Name	Price
3	Textbook	
7	Pen	

or `IS NOT`:

```
SELECT * FROM Products WHERE Price IS NOT NULL
```

Product_ID	Product_Name	Price
1	Notebook	0.20
2	Ruler	0.50
4	Apple	0.80
5	Orange	0.40
6	Peach	0.20

In case a column value equality should be checked against a list of values, the following query is constructed:

```
SELECT * FROM Products WHERE Product_id IN (1, 5, 6)
```

Product_ID	Product_Name	Price
1	Notebook	0.20
5	Orange	0.40
6	Peach	0.20

This way we will find all the product rows where the Product item number takes one of the specified values. The templates help to search for text fields by using the `LIKE` operation. In addition to the words searched, the template can include special characters `%` percent and `_` underscore. The underscore (`_`) replaces any character in the template; and the percent sign (`%`) replaces any (sequence of characters (including empty).

```
SELECT * FROM Products WHERE Product_name LIKE 'P%'
```

Product_ID	Product_Name	Price
4	Apple	0.80
5	Orange	0.40

Let's remember the schema of our database. We have tables `Customers`, `Products` and `Orders`. The `Orders` table links the `Customers` and Products tables. How do we trace this connection?

For example, we are going to find out what orders are related to the product Pencil. Of course, one can solve the problem in a few steps.

First, find the product code of the product Pencil:

```
SELECT Product_ID FROM Products WHERE Product_Name='Pencil'
```

Then find the orders from the `Orders` table that include the found product Code.

```
SELECT * FROM Orders WHERE Product_ID=5
```

The question is whether it is possible to find the necessary information in one step? Certainly.

```
SELECT * FROM Orders WHERE
Product_ID=(SELECT Product_ID FROM Products WHERE
Product_Name='Pencil')
```

We have used a subquery, when some queries are nested within another one. A subquery can be used instead of a table name or instead of a set after the `FROM` or `WHERE` clauses, or as an expression after the `SELECT` clause. For example, subqueries can be used to find out the product name in every order.

```
SELECT Order_ID,
(SELECT Product_Name FROM Products WHERE
Products.Product_id=Orders.Product_id) FROM Orders
```

The subquery returns the product name that corresponds to the product item code. It should be noted that the table name appears before the `Product_ID` name. In fact, you could always specify the table name before the column name. However, it must be specified if there are two tables in the query and they have columns with the same names. Otherwise, if the table name is not specified, it is not clear what column is in question.

Let's try to write this request in the other way. We want to join the rows of the `Orders` table and the `Products` table. The product code is the joining criterion.

To do this, a special request type with a `JOIN` is used.

```
SELECT * FROM Orders JOIN Products ON
Products.Product_id=Orders.Product_id
```

The rows of the two tables are joined by the matching values of the `Product_ID` field. Have you noticed that the `Product_ID` column is repeated twice in the sample? Let's display it just once.

```
SELECT Products.*, Order_ID, Customer_ID, Quantity
FROM Orders JOIN Products ON
Products.Product_id=Orders.Product_id
```

And what happens to the products that no one has ever ordered? Such rows of the table `Products` are not joined, as in the `Orders` table no rows have matching values of `Products`. So, such products are not included in the final set. Therefore, the `JOIN` clause is also called `INNER JOIN`. If we need to get all the products in the final sample when joining the tables, then a full joint should be used (clause `FULL JOIN`).

```
SELECT * FROM Orders FULL JOIN Products ON
Products.Product_id=Orders.Product_id
```

Then rows containing the products that have not been ordered yet are supplemented with the unset values.

Inner joint can be rewritten in the way of so-called product of tables. What will be the result of the query?

```
SELECT * FROM Orders, Products
```

In fact, we get all the rows from the tables listed in the query. Now we can select only the rows where the `Product_ID` from the table `Products` matches with the `Product_ID` from the table orders.

```
SELECT * FROM Orders, Products WHERE
Products.Product_id=Orders.Product_id
```

Often we need not only to select the table rows according to the specified criteria, but also to carry out some analysis, that is, to count the number of rows in the sample, to calculate the sum or the average value of the numerical column, or to determine the minimum/maximum value.

To do this, we use the built-in aggregate functions. The aggregate function can be called immediately after the `SELECT` statement. The parameters of aggregate functions can be columns, or some expressions made up of column names and constants).

The most common set of built-in aggregate functions in any DBMS is sum, number of rows, average, minimum and maximum.

For example, the simplest aggregate function counts the number of rows.

```
SELECT COUNT(*) FROM Orders
```

This query will count the total number of rows in the `Orders` table. How to find the price of the most expensive product? Let's use the `MAX` function.

```
SELECT MAX(Price) FROM Products
```

Similarly, you can find the price of the cheapest product, and the average product price.

```
SELECT MIN(Price) FROM Products
```

```
SELECT AVG(Price) FROM Products
```

Here a condition is added (the clause `WHERE`) to select the rows to apply the aggregate functions to.

```
SELECT AVG(Price) FROM Products WHERE Price>0.4
```

If we use aggregate functions for the entire table, then you cannot use column names next to the aggregation functions after the `SELECT` statement as we did it before. For example, we found the price of the most expensive item.

```
SELECT MAX(Price) FROM Products
```

But how to find the product itself? We would like to write a query like this:

```
SELECT MAX(Price), Product_ID, Product_Name FROM Products
```

But you can't write like that. Why is this query wrong? Let's imagine that we have several products with the maximum price. What product you should to select in this case? Let's write this query correctly:

```
SELECT Product_ID, Product_Name FROM Products
WHERE Price= (SELECT MAX(Price) FROM Products)
```

You can also group rows by a specific criterion and calculate aggregate functions for each group. For example, let's group the rows of the `Orders` table by product to find out the number of times each product was purchased.

```
SELECT COUNT(*) as total, Product_ID FROM Orders
GROUP BY Product_ID
```

If we use grouping, the grouped fields can be selected together with the aggregate functions.

Let's add the product name to the sample. There are several ways to do it. For example, using a `JOIN` statement:

```
SELECT COUNT(*) as total, Orders.Product_id, Product_Name
FROM Orders JOIN Products on Products.Product_id=Orders.Product_id
GROUP BY Orders.Product_id, Product_Name
```

Or by Cartesian product of the sets of records from two or more joined tables

```
SELECT COUNT(*) as total, Orders.Product_id, Product_Name
FROM Orders, Products
WHERE Products.Product_id=Orders.Product_id
GROUP BY Orders.Product_id, Product_Name
```

Or we can include a subquery

```
SELECT COUNT(*) as total, Orders.Product_id,
(SELECT Product_Name FROM Products WHERE
Products.Product_id=Orders.Product_id ) AS Name
FROM Orders
GROUP BY Orders.Product_id
```

Some constraints can be specified for groups. For example, let's calculate the average product item number in each order, while considering only the products purchased two or more times.

```
SELECT Product_id, AVG(Quantity) as Avg_Quantity
FROM Orders
GROUP BY Orders.Product_id
HAVING COUNT(*)>1
```

And finally, let's deal with another complex query. How to find out the cost of each order? We need to know the price of each product, it is in the `Products` table, and the number of products in each order, this information is stored in the `Orders` table.

```
SELECT Order_ID, Customer_ID, Products.Product_Name,
Quantity*Price as Order_Price
FROM Orders JOIN Products ON
Products.Product_id=Orders.Product_id
WHERE Price IS NOT NULL
```

6 Database Objects

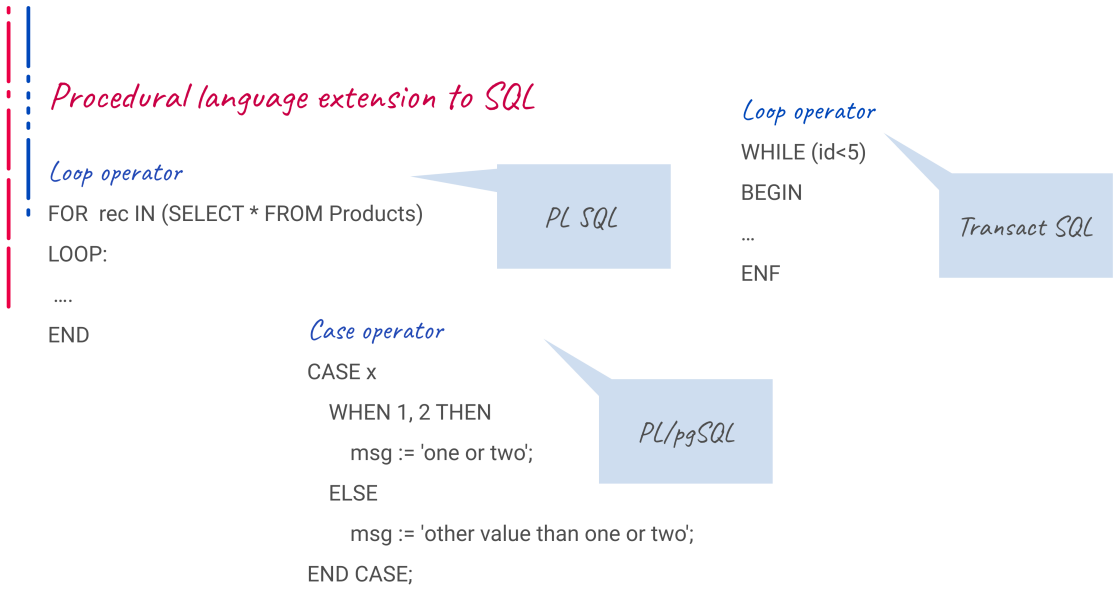
In addition to tables, which directly store data, DBMS stores a large number of different additional objects. They are needed to provide users with many useful and convenient features in addition to data storage.

Let's start with views. View can store a large and complex query that database users can address to simply by its name.

```
CREATE VIEW Products_ID_and_Name as
SELECT Product_ID, Product_Name FROM Products

SELECT * FROM Products_ID_and_Name
```

SQL is a very powerful data manipulation language, but it lacks control structures such as conditional operators, loop operators, and others to solve complex data processing tasks, that can be found in other programming languages. Therefore, many DBMS have procedural extensions to the SQL language, that is a full-fledged programming language that provides also SQL commands. Using this language you can write procedures and functions that are permanently stored in the database and are executed in DBMS environment.

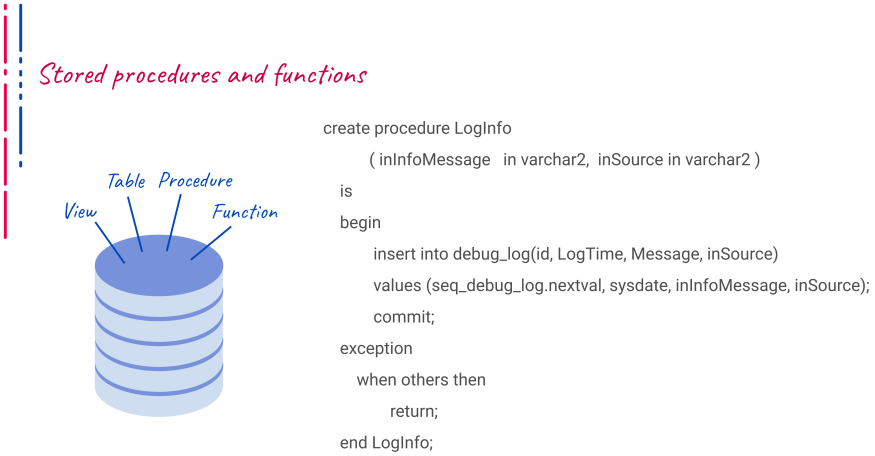


Procedures and functions allow you to store a commonly used set of data operations that can be built into various database applications. In addition, DBMS contains many built-in functions, such as mathematical, statistical, row functions, data conversion functions.

However, the DBMS provides its own procedural language extensions to SQL, thus making the task of software portability more complicated.

How does it work? Let's imagine that we are developing an application for the Bank that gives out a lot of loans every day. It is clear that loan processing consists of several simple operations. It is necessary to enter information about the client, check that he has a clear credit history, identify his average salary and other important parameters, and then depending on all the above the maximum loan amount is estimated. All these operations are to be described sequentially and to be stored in the database. Then the application can simply call the function and set its parameters.

Stored procedures or functions can be called from an application that solves specific tasks using a database. And there is a special kind of procedures that are not called explicitly, but are executed automatically in response to certain changes in the database. These procedures are called triggers.



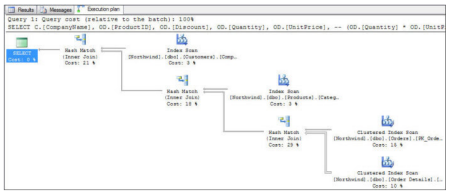
Triggers are special procedures that allow you to determine the response to events that occur in a database and to make some changes in data in response. For example, the University enrolled a new student – a new entry is added to the Student table. Then a student card with a unique number should be issued. It is necessary to check whether he is other town, and in case he is, provide a place in the student residence. This chain of events can be created using triggers.

A very important part of DBMS is query optimizer. This component determines the most effective execution way or execution plan. We have learned how to write complex queries that can collect and aggregate data from multiple tables. These queries can be executed differently. For example, you can get from St. Petersburg to Moscow by train, by plane or by car. And the difference in performance, or the execution speed of the same query may differ by orders of magnitude. Why does it happen? By changing the order of elementary operations which the query can be broken into, and by using additional objects that allow to quickly find the desired values upon assigned criterion in a large table of a million rows. These objects are called indexes. Indexes for the `PRIMARY KEY` and `UNIQUE` fields are created automatically when the table is designed, and in

Query optimization

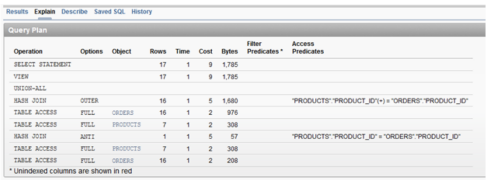
SELECT * FROM CUSTOMERS WHERE

CUST_LAST_NAME='Bond'



SELECT * FROM ORDERS FULL JOIN PRODUCTS ON

PRODUCTS.Product_id= ORDERS.Product_id



addition they can be created by the database administrator for other columns that are frequently searched for. The optimizer determines whether to perform a direct table scan or use indexes based on data distribution in the tables that the command addresses to. Another important internal component of the database to

Database indexes

Index 1

CREATE TABLE Products (

Product_ID integer PRIMARY KEY,

Product_Name varchar(32) UNIQUE NOT NULL,

Price Numeric(10,2) NULL

)

Index 2

CREATE INDEX Ind1 on Products(Product_Name, Price)

Index 3

be mentioned is the transaction Manager. You probably know the term "transaction" from the financial sector: when you pay for the goods you receive a message from a bank that payment from your card is arranged that means that the transaction or money transfer took place. The essence of a transaction is to bind multiple commands into a single operation on all-or-nothing basis, which means the transaction is either entirely completed or not executed at all. Transaction

Manager keeps track of all changes that occur in the database, recording them in a special log. While working with data, various malfunctions can occur. For example, errors in the program, power outage, equipment damage. In case of any failures and errors, the transaction log allows you to restore the correct state of the database. The transaction Manager also coordinates how several users can work with the data simultaneously. Some actions with the data are not allowed be performed at the same time – for example, you cannot allow multiple users to book the same seat on the same flight. At the same time, you should not prohibit viewing data at the same time. The transaction Manager uses locks and other mechanisms to provide access to data in different modes, so that the actions of users do not contradict each other.

For security and data protection purposes, not all users may have access to all information stored in the database. For example, it is improper if personal data will be stored in the public domain. Let alone, you cannot give everyone the rights to adjust these data. Each user can see only a part of the stored data, and only some operations can be available for him. Database and dataset access rights are granted to specific users and define user access rights settings. If there are many users with the same set of rights, they are combined into roles. Such data access rights are usually given by the database administrator.

Thus, we have shown that relational databases are a powerful and reliable mechanism for storing and manipulating data. It would seem that by means of relational DBMS all problems of big data are solved. However, it's not that easy. Relational databases are well suited for storing structured data, and modern data streams sometimes do not have a clear structure – these are documents, photos, video and audio files. For their storage and processing, special storage facilities are developed, usually suitable for data of a certain type. Such systems are called NoSQL.