



Data Design

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

Contents

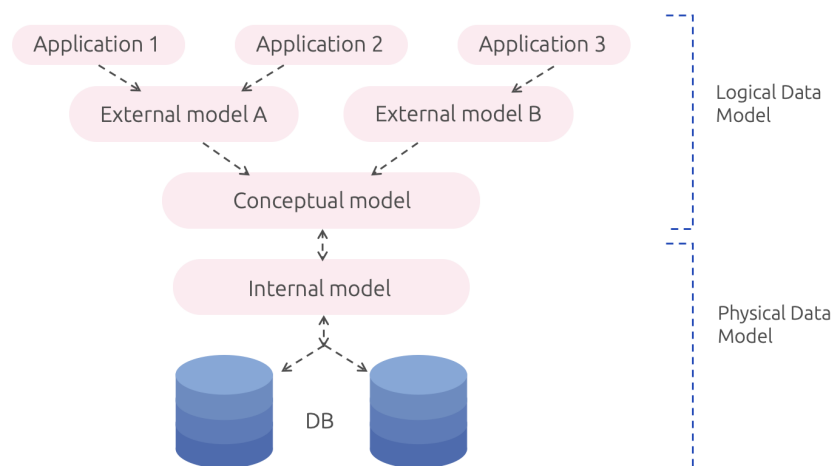
1	Entity-Relationship model diagrams (ER- diagrams). Entities.	2
2	Entity-Relationship diagrams (ER-diagrams). Relationships.	6
3	Converting an ER model into a relational database	13
4	Tables creation	18
5	Integrity constraints	23

1 Entity-Relationship model diagrams (ER- diagrams). Entities.

The process of creating a database always begins with a description of the subject area. The first thing to find out is which user requests the information system will have to answer. It is necessary to understand how the stored descriptions of the structure of information objects or concepts will be related to each other and what are the requirements for valid data values.

A database project should begin with an analysis of the subject area and identifying the requirements for it of individual users (for example, employees of the organization for whom the database is being created).

Types of Data Models in DBMS





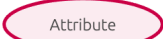
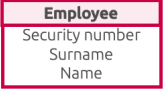
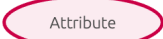
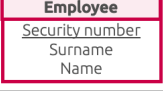

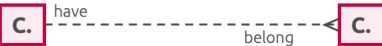
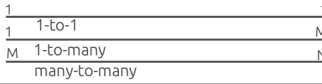
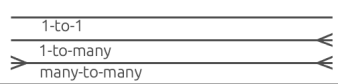
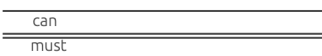
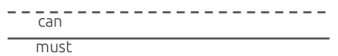
By combining private representations of the contents of the database from the user survey and their views on data that may be required in future applications, the database designer first creates a general informal description of the database. This description, made using natural language, mathematical formulas, tables, graphs and other means, understood by all people working on the design of the database, is called the conceptual level.

The conceptual layer describes the complete logical structure of the stored data. In the last design step, an internal data model is created that describes the representation of the conceptual scheme in the context of the selected DBMS.

To present data at the conceptual level, the corresponding tool is needed, which makes it possible to visualize the data structures of the subject area and the relationships between them. A suitable tool for this is the «Entity-Relationship» model diagrams. The description of such diagrams was first proposed by Peter Chen in 1976.

Many similar models with these sort of designations have been developed. All variants of the entity-relationship diagrams come from one idea – the drawing

Entity-Relationship model

Definition	Cheng's notation	Barker's notation
Entity		
Attribute		
Key attributes		
Relationships		
Relationships types		
Modality		

is always more obvious than the text description. Let's consider the construction of diagrams in Barker's notation. The elements of the domain can be presented as a collection of objects and the relationships between them. The main elements of diagrams are entities, which are stored objects or concepts, attributes that represent properties of entities, and relationships between them.

Let's take for example the database **Exam session**. The database should contain information about students who passed their exams in certain subjects. The student is characterized by a record book, full name, date of birth. May have multiple phones. All students are divided into study groups. Each group has its own praepostor.

The group is assigned exams for specific courses, teachers, exam dates, and audiences. For each passed exam, the student is rated.

Limitations: the record book number is a positive number, there are no digits in the name and the score is from two to five.

1.0.1 Entity

An entity is a real or abstract object of a certain kind that can exist independently of others. An entity is sometimes called a class of similar objects that will be stored in a database.

Each entity must have a unique name, and the same name must always be interpreted in the same way. A list of entity instances forms a set.

The entities in the ER diagrams are represented using rectangles where the name of the entity is indicated in the top. The name of the entity is usually a noun.

We can identify the following entities in the **Экзаменационная сессия**

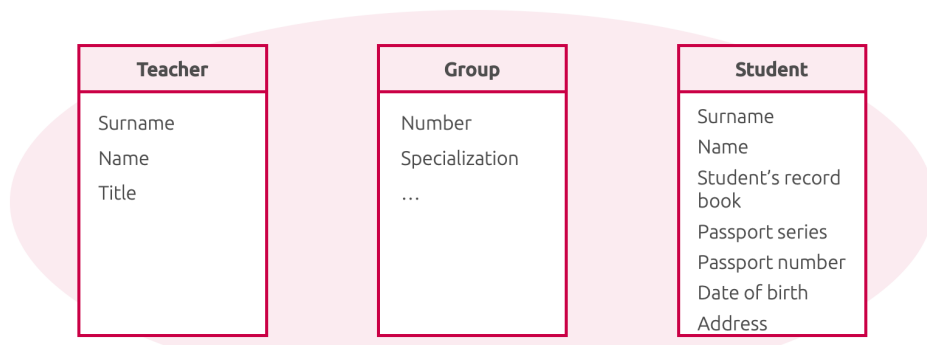
database: Student, Teacher, Exam, Group, Subject.

1.0.2 Attribute

Attribute — a named characteristic of the entity. Its name must be unique to a particular type of entity, but may be the same for different types of entities.

Each attribute has its own data type. For example, a **Student** entity may have a name, surname, passport number, student's record book number, and date of birth. **Group** will be described by number and specialization. **Teacher** — full name and title.

Entities attributes in the “Exams” database



1.0.3 Entity identification

Entity instances must be distinct, therefore **entity identifiers** are needed. Entity instances cannot be duplicates, so at least all attribute values form a unique combination.

Quite often not all attributes are needed to identify the entity. There are one or more attributes necessary and sufficient for this. For example, to specify a student, you need to know his student's record book only, but the date of birth and address are not necessary for this.

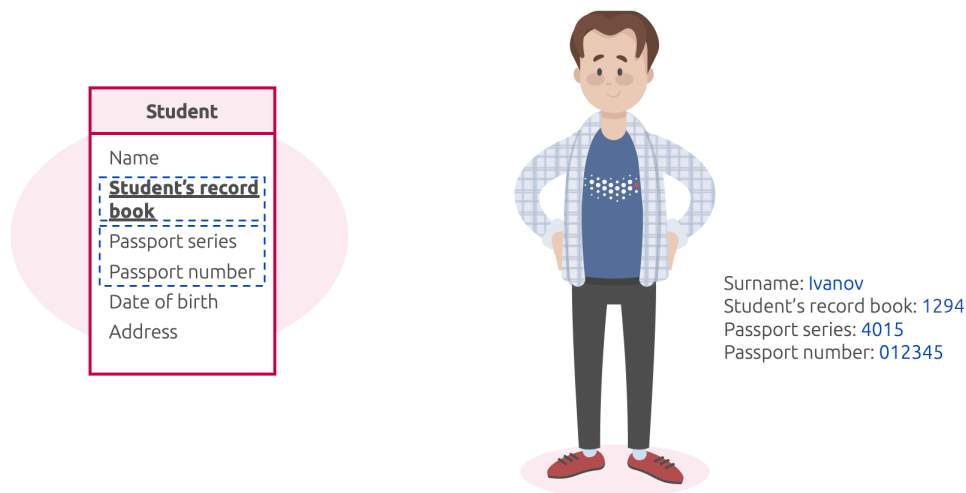
The ER diagrams also show the keys. Recall that the key refers to one or more entity attributes that uniquely identify the entity.

If a key consists of one attribute, then this key is simple. A key consists of several attributes is called a composite one. Sometimes the entity may have several possible keys.

Examples for the student entity can be a simple key — student's record book number, or a compound key — a combination of attributes passport series and number.

If there are several possible keys, one of them, most commonly used in the search, is called the primary key. A good tone is choosing a numeric, short, and

Entities identification



lifelong attribute as the primary key. In the diagram, the attributes that make up the primary key are highlighted by a single line.

There are several possible ways to identify:

- natural keys;
- "on position" (geographical, in order, in time);
- surrogate keys.

Typically, an entity has attributes that describe its properties, which identify each instance of the entity. These keys are called natural.

No natural identifier can be absolutely reliable, therefore surrogate keys are more applicable for the system. But they are useless in the search.

It is not always easy to find such an attribute among the attributes of the entity, and sometimes it simply does not exist. For example, the entity **Subject** — the name as a key cannot be taken, because there can be subjects with the same name, and different teachers can teach the subject.

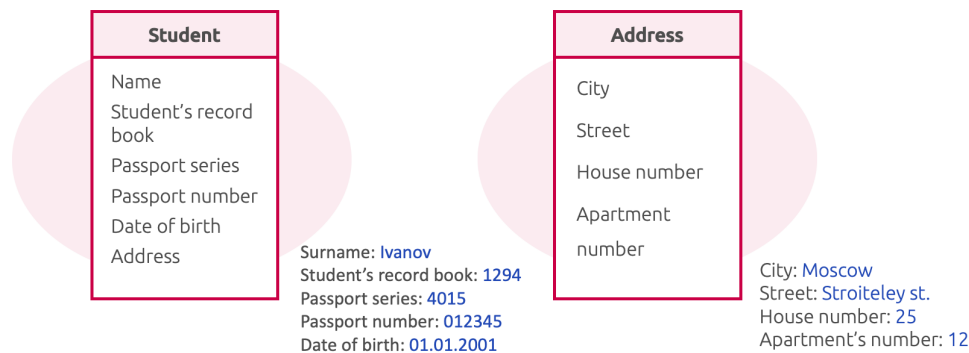
If there is no natural key, an artificial — «surrogate» is invented. For the entity **Subject**, you can define a numerical identifier.

When constructing an ER diagram, it is difficult sometimes to decide where the attribute of the entity is and where the other separate entity is.

There is no absolute difference between entity types and attributes. The attribute is such only in connection with the type of entity. In another context, an attribute can act as an independent entity. For example, an address can be an attribute for a person if we are only interested in the place of residence.

But the address can be an independent object, having its characteristics like City, street and house number. Then we can find out not only where a person

Attribute or other entity



lives, but also to answer more complicated requests: how many people live in such a city or on a particular street. Therefore, the choice between a possible attribute and a new entity must be made based on possible data queries.

2 Entity-Relationship diagrams (ER-diagrams). Relationships.

We've worked out the design of entities, and now let's look at the connections that arise between entities — they're called relationships.

A relationship is the association of two or more entities. If the purpose of the database is only to store separate, unrelated data, its structure can be very simple. And since real databases often contain hundreds or even thousands of entities, theoretically more than a million relationships can be established between them.

This multiplicity of relationships determines the complexity of the infological models. Relationships, as well as entities, have their own name – most often a verb. For example, **Student is in Group**, **Teacher takes Exam**.

The main difference between relationships and entities is that relationships themselves are meaningless and cannot exist without connected entities.

Relationships, like entities, need to be identified. The identifier includes the keys of the linked entities and possibly some selected attributes of the relationships.

Here is an example of a relationship attribute. For example, if a student has passed an exam for a certain subject to a teacher, then the grade will be an attribute of a relationship, not a student, subject or a teacher.

2.0.1 Relationships properties

Relationships may have different characteristics. The most important are:

- dimension;
- cardinality;
- modality

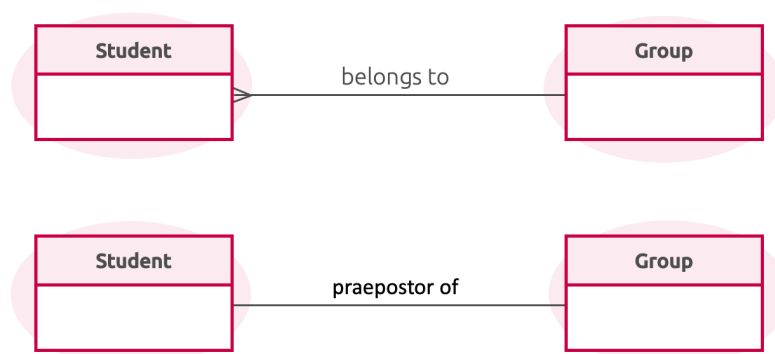
Relationships properties

- Relationships can have their own attributes
- **The difference between entities and relationships:**
the latter can not exist without the corresponding entities
- **Relationship identification:**
the key includes the corresponding entities keys and possibly allocated attributes of the relations

The degree of relationship is the number of entities that are connected. The most typical one is – binary when the relationship is defined between two entities.

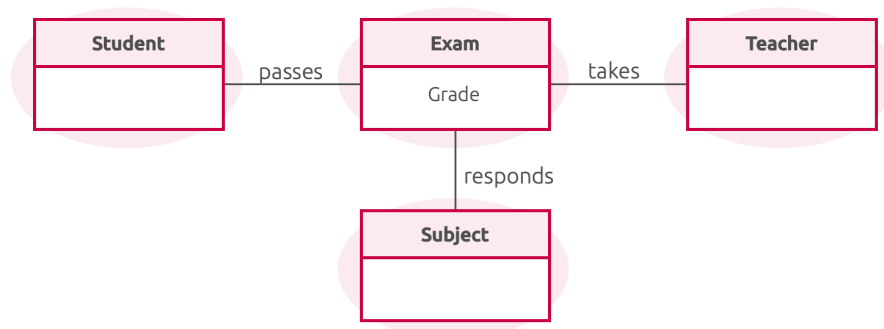
If the three entities are connected, as in the example with the exam: student, teacher and subject, then the relationship is called ternary. There are unary, or recursive, relationships that occur within one entity — for example, one employee directs the others. In general, the relationship between n entities is called n-ary.

Example: Binary relationship



Let's consider a **binary relationship** between a student and a group. Several sets of relationships can be defined between two entities. For example, a student can *be in a* group, or can *be a praepostor* of the group. These are different relationships between the same entities.

Example: Ternary relationship



An example of a **ternary relationship** may be an exam: three entities participate in this relationship – **Student**, **Teacher** and **Subject**. The grade is a relationship's own attribute. Often, for convenience, instead of n-ary relationships, only binary ones are used. To do this, we will have to enter the entity Exam instead of the relationship *exam*.

Here is an example of a **recursive relationship**. For example, consider the **Employee** entity. Some employees can manage others while remaining employees at the same time. It appears that the entity has a relationship with other instances of the same entity.

There are «correct» (independent) and «weak» entities. Weak entities cannot exist without relationship to other (strong) entities, or are dependent on other entities.

Let's pretend that we need to store information about a student's phones. At first glance, it may seem that a phone number is just an attribute. But a student may have several phone numbers – home, mobile, office, several mobile numbers. Then it is more correct to separate the phone into a separate entity. The **Phone** entity, in addition to the number, may have other attributes: phone type, city code, etc. Some phone numbers may be the same, for example, home and work. Technically, to distinguish them you need to come up with some sort of identifier, or a surrogate key. But it is unlikely that we will need a phone number by itself without information about who its owner is.

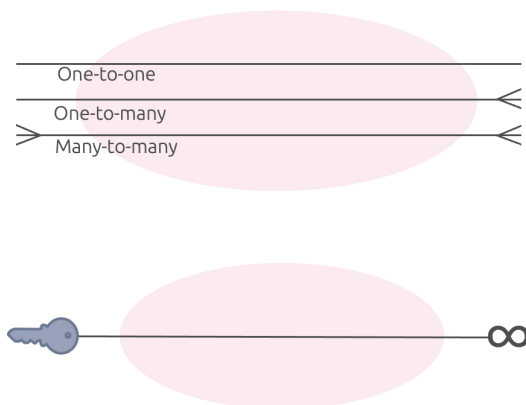
In this case, information about a weak entity is stored together with a relationship that indicates an independent entity, and a surrogate key is not needed. Such entities and relationships in diagrams are drawn in a double line sometimes.

Cardinality indicates the maximum number of instances of one entity associated with instances of another entity.

They are divided into three types depending on the number of entities involved.

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

Binary relationships



Cardinality indicates the maximum number of instances of one entity associated with instances of another entity. The cardinality of relationships represents the end of a line connecting entities. The relationship «one» denotes a straight line at the end of the line, many — an inverted arrow, which is often called the «Crow's Foot».

The diagrams sometimes display the relationship «one» as a key, and the relationship «many» as an infinity sign.

2.0.2 Relationships one-to-one

The most easily described, but least often found in nature, is the relationship like one to one. It occurs when exactly one instance of one entity communicates with one and only one instance of the other entity.

As examples, we can mention the relationship between the capital city and the country where the city is the capital, the rector — the university (the rector heads the university). Objects are connected by a straight line.

- **“One-to-one”**: one instance of one entity communicates with one and only one instance of the other entity



2.0.3 Relationships one-to-many

One-to-many: Each instance of the first entity may correspond to several instances of the other entity, but each instance of the second entity corresponds to no more than one instance of the first entity.

- **“One-to-many”**: Each instance of the first entity may correspond to several instances of the other entity, but each instance of the second entity corresponds to no more than one instance of the first entity



For example: each department may have many employees, but each employee works in only one department.

2.0.4 Relationships many-to-many

There are also relationships of many-to-many: each instance of the first entity may correspond to several instances of another entity, and vice versa.

- **“Many-to-many”**: each instance of the first entity may correspond to several instances of another entity, and vice versa.



2.0.5 Modality of relationships

The last property of the relationships we consider is called **modality**. The optional relationships (conditional) -modality «can» means that an instance of one entity may be associated with one or more instances of another entity, or may not be associated with any instance.

- The passenger can have a ticket.
- A person can have a car.
- The student can pass the exam.

Mandatory -modality «must» means that an instance of one entity must be associated with at least one instance of another entity.

- Each course of lectures must have a teacher.
- Each department must have a supervisor.
- Each ticket is issued to a certain passenger.

Modality of relationships



Whether entity records are required in the relationship?

- **Modality type** ————— mandatory
- **Non-modality type** - - - - - non-mandatory

If you consider the relationship between the student and the exam, then there should be a modality «can» because the student may not pass any exam and may be an exam that no one has passed yet.

For example: each staff member may be involved in several projects and each project involves several staff members. Student-course: each student studies multiple courses, and each course is studied by many students.

So we looked at all the elements of the ER diagrams.

To build a diagram, you need to do the following steps:

1. Define entities.
2. Define entity attributes.
3. Define primary keys.
4. Define the relationships between entities.
5. Define cardinality.
6. Draw ER-diagram.
7. Check ER-diagram.

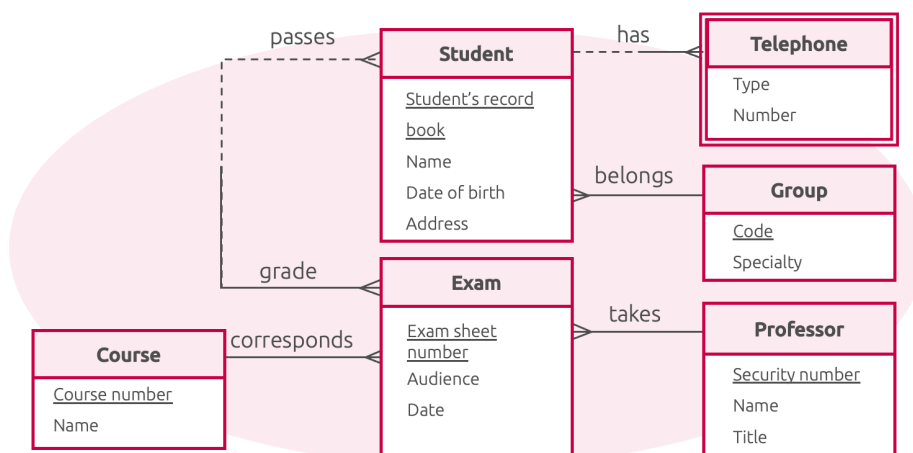
Let's remind the subject area we were considering: **Examination Session**. Our database should contain information on students who have passed exams in certain subjects. The student is characterized by a student's record book, full name, date of birth. May have multiple phones. All students are divided into study groups. For a group, an examination is scheduled for a specific subject, a teacher is assigned. The date of the exam and the audience is specified. For each passed exam, the student is rated. Limitations: The student's record book number is a positive number, there are no digits in the name, the score is from two to five. The student is in one study group and can only pass one exam in one day.

3 Converting an ER model into a relational database

We have learned to define data structures in terms of an entity-relationship model using ER diagrams. The next step after this is to select a specific DBMS that supports some data model and display the diagrams on the structures from the selected DBMS. If the choice is relational DBMS then it is necessary to transform the data structures of the ER-model into relations.

Let's remember that relations are database objects that store all the information available to the user.

ER-diagram example



The relation has a header, or a scheme describing a table structure consisting of attributes, or column names, and a relation body composed of rows or tuples – they contain the data itself. The values of each attribute are contained in a valid set of possible values called a domain.

Relation scheme – an ordered set of attributes. Each attribute has a name unique to the relation, and a data type is specified for each attribute.

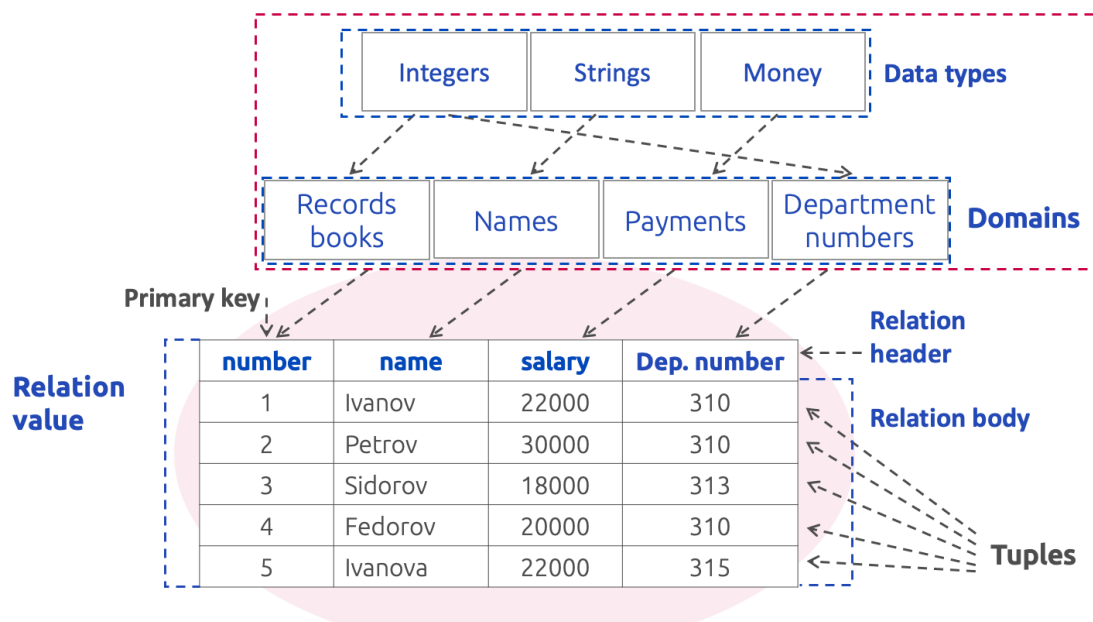
3.0.1 Data types

There may be some differences between supported data types and their names in different DBMS, but in any database you can store numbers, strings, date and time information.

Numerical data types can be integer and fractional, given with different accuracy, i.e. it is possible to explicitly specify the number of characters in the integer and fractional parts of the number.

Character data can be represented by fixed-length or variable-length strings. When describing the type the size is indicated in parentheses. For variable-length strings this size sets the maximum possible string length and for fixed-length

Scheme of relations



strings, the exact string length. If you do not specify the size then in some DBMS the length of the field will be equal to one character, in some — a string of variable length of the maximum possible size.

The date and time data type is also used in any DBMS but this is often the type that differs. There are many different formats for date and time, for example, storing in one type of date and time or separately.

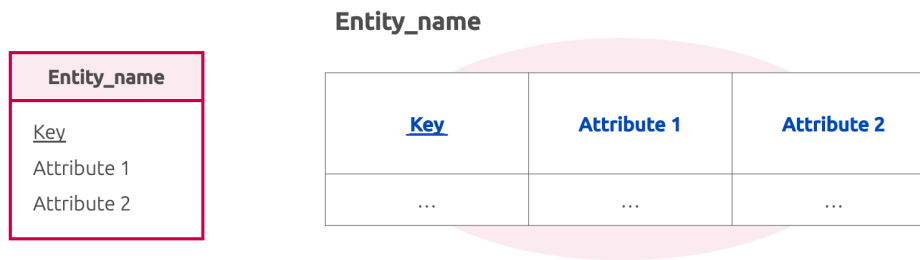
- Numbers
- Strings
- **Date/time**

Type	Example value
time	'13:52:03'
date	'12-10-25'
datetime	'11:10:17.1234567'
smalldatetime	'15-02-19 12:22'

A number of DBMS provides special data types for storing data in other formats: hierarchical data, XML, JSON and others.

For each entity described in the ER model, a separate table must be created in the database. The entity name becomes the table name. Each attribute becomes a column. Latin letters are used for table and column names. The appropriate data type is defined for each column. The entity ID is converted into a table key.

Let's remind that a possible key is a minimal set of attributes by which you can define all the others. The primary key is chosen as one of the possible keys usually used most often in the search.



For example, in the **Exam Session** database, you need to create a table for the entity **Student** with the attributes of the student's record book, full name, passport series and number, date of birth and address. Let's create a table **STUDENT** with fields:

- Student's record book number – **StudentId** – integer;
- Full name – **StudentName** – string of variable length up to 100 characters;
- Passport series – **Pass_s** – string up to 4 characters;
- Passport number – **Pass_num** – string up to 6 characters;
- D date of birth – **BirthDate** – Date format;
- Address – **Address** – string of variable length up to 100 characters.

Possible keys – student's record book number, series and passport number. Choose **StudentId** as the key.

Create a table for the **Group** entity by calling **ST_GROUP**. The fields are:

- Group code – **GroupCode** – string up to 32 characters;
- Specialization – **Specialization** – string of variable length up to 100 characters.

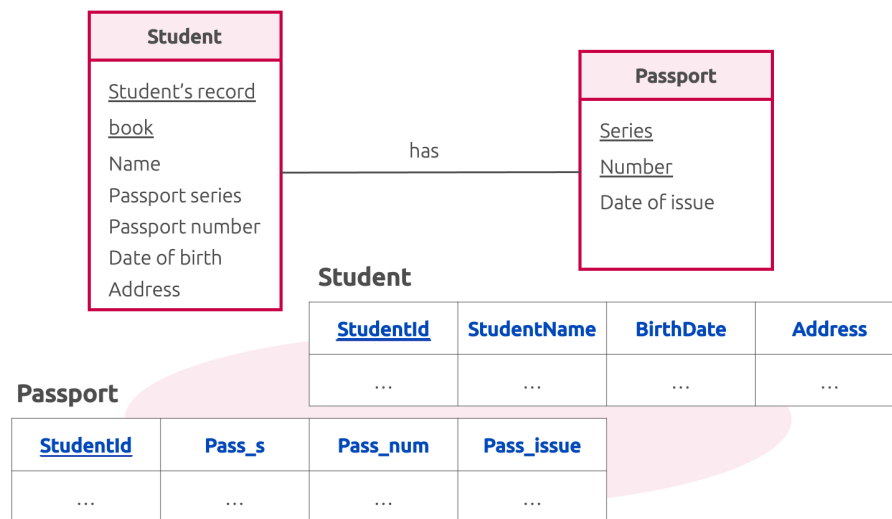
Key – **GroupCode**.

Relationships are also stored in a relation. To display a relationship, key attributes of the objects involved in the relationship and the corresponding attributes, if any, are required. One-to-one relationships are stored in a single table, whose columns correspond to the attributes of both entities.

Consider the simplest example of a 1:1. Suppose you want to store student passport information. Then we would enter a separate entity – passport number with attributes series, number and date of issue.

Each student has exactly one passport, and each passport corresponds to exactly one person. In this case, it was possible to present this information in one table **student** with attributes Student ID, Full Name, Passport Series, Passport

1:1 relationships

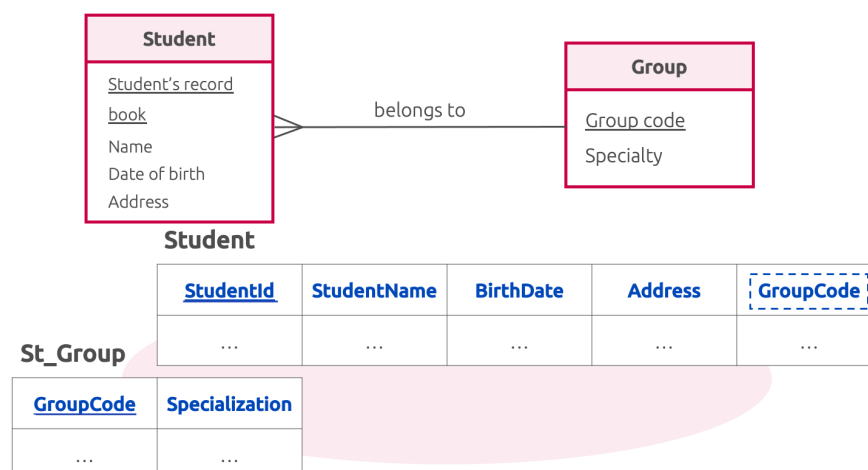


Number, Passport Issuance Date, Date of Birth, Address. The relationship key can be student ID or series and passport number.

Sometimes, on the contrary, attributes of the same entity are stored in two different tables. For example, for security reasons, we can store student passport numbers in a separate table.

Then one table would have the fields Student's record book number – **StudentId**, **StudentName** – student name, date of birth and address. Key – **StudentID**. Another table would contain the **Pass_s** passport series and **Pass_num** passport number and **StudentId** – key.

1:M relationships



If there is one-to-many relationship between entities, such as between entities **Group** and **Student** – one or more (many) students study in each group, and each student studies in exactly one group, then this relationship can be implemented

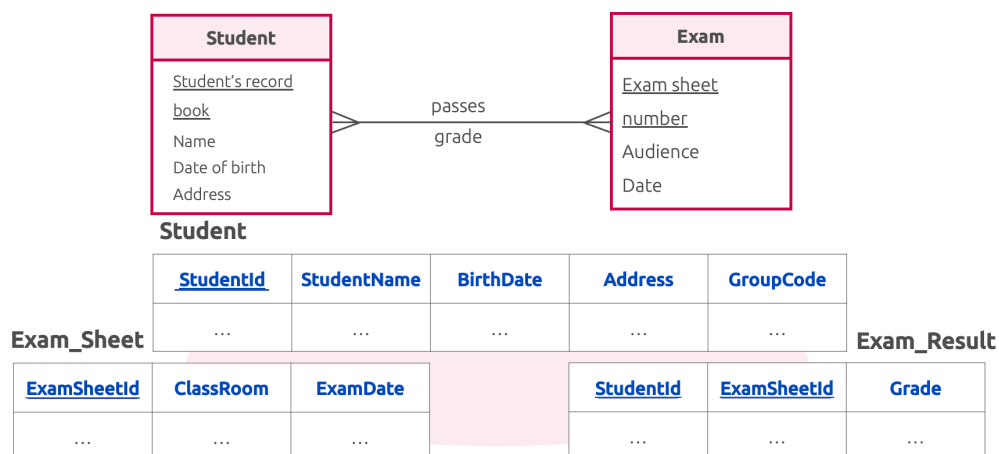
as follows: in the table of entities with the end of the relationship «many», we add the key of the entity with the end of the relationship «one».

It also stores information about relationships to weak entities. A student may have multiple phone numbers, and some numbers may match, such as home numbers.

We add to the table of phone numbers the key of the entity **Student** – **StudentID**. The relationship key is a composite: **Phone** and **StudentID**.

To store many-to-many relationships, you have to create a separate table that serves to display the relationship. The scheme of this relationship is composed of the key attributes of the objects involved in the relationships, and the attributes of the relationship, if any. For example, the relationship between the entity **Student** and **Exam** is many-to-many. The student gets a grade for the exam – the grade attribute is a property of the relationship. Creating a table **Exam Result** with the fields Student ID, Exam ID and grade – integers.

M:N relationships



4 Tables creation

We looked at how you can map a conceptual database model in the form of an ER diagram to a relational, table model. The next step is to describe the tables and integrity constraints in the context of a specific DBMS. To do this, there is a special language in any DBMS that makes it possible to create objects and work with them. The most common is structured query language **SQL**.

SQL – The most common and structured query language most often used for relational databases and supported by most DBMS



It is most often used for work with relational databases and is supported by most DBMS. Therefore, we will consider it.

All language operators can be divided into three groups:

- data definition operators;
- data manipulation operators;
- data management operators.

To describe the structure of tables and other objects, DDL operators are needed- the data definition language. There are only three operators: **CREATE**, **ALTER** and **DROP** — for creating, modifying, and deleting database objects.

SQL DDL (Data Definition Language)

- **CREATE** <OBJECT> [OPTIONS] - creation
- **ALTER** <OBJECT> [OPTIONS] - modification
- **DROP** <OBJECT> [OPTIONS] - deletion

The operator options are highly dependent on the specific object being created and the context of the DBMS used.

The data manipulation language contains operators that allow you to display and modify the contents of the tables: add and edit rows/records, delete and search them according to the specified criteria.

DML (Data Manipulation Language)

- *SELECT*
- *DELETE*
- *INSERT*
- *TRUNCATE*
- *UPDATE*

The data control language consists of operators that manage user rights in the database. For example, whether to allow the user to read the data from the table, execute the stored procedure, etc.

DCL (Data Control Language)

- *GRANT* - to assign privileges
- *REVOKE* - to withdraw privileges

DBMS provide a database environment in which you can write and execute commands. Database creation starts with a special command, which reserves

Database creation

- *CREATE DATABASE db_name [options];*
- *CREATE USER [options];*
- *CREATE SCHEMA [options];*



some storage area where all database objects will be saved. Of course, when creating a database, you can specify not only the name of the database, but the location of its storage, specify the default formats and specify many more useful parameters.

Now we can start creating tables. We need the **CREATE TABLE** operator, and then in parentheses describe the table that you are creating. The table description consists of column descriptions and integrity constraints. The table columns represent the attributes of stored entities. Integrity constraints are rules

that data must satisfy. For example, in a surname there are no numbers, the score for the exam can be from 2 to 5. It is very important to specify when describing the table what values can take stored data to prevent the entry of incorrect values. To create a simple table, you simply need to describe the columns in parentheses,

Tables creation

- **CREATE TABLE** (
 {
 {Column name}
 {Data type}
 [Default value]
 [List of integrity rules]
 }+
)



specifying the column name and type.

For example, describe the columns of the table **STUDENTS**. The **CREATE TABLE** operator creates an empty table with five columns.

```
CREATE TABLE STUDENTS (  

    StudentId INTEGER,  

    StudentName VARCHAR(100),  

    GroupCode VARCHAR(32),  

    BirthDate DATE,  

    Address VARCHAR(100)  

);
```

StudentId	StudentName	GroupCode	BirthDate	Address
-----------	-------------	-----------	-----------	---------

After a table is created, its structure can be changed. This is done by using the **ALTER TABLE** operator. After **ALTER TABLE** you specify the name of the table you want to change, and then you describe the action you need to change the table. To add a column, use the phrase **ADD COLUMN** where the column name is specified.

The table structure modification

- **ALTER TABLE** < Name >{
 {ADD| DROP | ALTER} COLUMN
 Column name
 Type
 [Default value]
 [List of integrity rules]
 }+



For example, you can add the phone number – field `Phone_number`, data type `CHAR(11)`, then make the field length 10 characters, and remove the column.

```
ALTER TABLE STUDENTS
ADD COLUMN Phone_number CHAR(11)
```

```
ALTER TABLE STUDENTS
ALTER COLUMN Phone_number char(10);
```

```
ALTER TABLE STUDENTS
DROP COLUMN Phone_number
```

StudentId	StudentName	GroupCode	BirthDate	Address
-----------	-------------	-----------	-----------	---------

You can delete a table using the `DROP TABLE` operator.

- `DROP TABLE <Name>`

```
DROP TABLE Students
```

StudentId	StudentName	GroupCode	BirthDate	Address	Phone_number
-----------	-------------	-----------	-----------	---------	--------------

The `INSERT INTO` operator is used to add records.

Adding records to a table

- `INSERT INTO <table-name> [(column1, ..., columnn)]
values (value1, ..., valuen)`

The following example demonstrates adding records to the `STUDENTS` table. For the first time, the column names were not specified, because the data being added corresponded to the column order specified when creating the table. The second time, the values of the fields `Student Id` and `studentName` were specified in the wrong order, so after the table name, the order of the fields of the added values was specified.

The `BirthDate` column has a data type – date. When entering values of this type, you need to know the default format – dates can be presented in different formats. Sometimes it is difficult to determine which one. In this situation, the function of converting a string to a date can come to the rescue. In some DBMS, this is the `to_date`. The function parameters are the date itself in the form of a string and its format.

INSERT INTO STUDENTS

VALUES(1345568, 'William Johnson', 'M2020_1', '01/20/1999', '711 Station Road, London N63 5SF');

INSERT INTO STUDENTS (StudentName, StudentId, GroupCode, BirthDate, Address)

VALUES('Lily Brown', 345569, to_date('17.05.1999','dd.mm.yyyy'), 'M2020_1', '93 Manchester Road, London SE02 6VS');

StudentId	StudentName	GroupCode	BirthDate	Address
1345568	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

There are two operators that can be used to delete the records: **DELETE** and **TRUNCATE**. The first operator removes rows that meet the specified criteria from the table and the second removes all records from the table.

DELETE FROM STUDENTS WHERE StudentId = 1294;

TRUNCATE TABLE STUDENTS;

StudentId	StudentName	GroupCode	BirthDate	Address
...
1294	Thomas Wood	B2020_2	12/09/1999	951 Highfield Road, London E16 2RI
...

As an example, let us give the command to delete from the table the **STUDENTS** row with the parameter **StudentId = 1294**. If you execute the second operator specified in the figure, all records will be deleted from the **STUDENTS** table.

The **UPDATE** operator is used to edit records.

Updating records from a table

UPDATE <table name>

SET column1 = expression1,

...

columnm = expressionm

[WHERE <condition>]

For example, change the address of a student with **StudentId = 345569**. Note that with the **UPDATE** operator you can change the value of not only one

field, but several fields at once, such as the address and name, as shown in the second command.

```
UPDATE STUDENTS SET Address = '93 Manchester Road, London SE02 6VS'
WHERE StudentId = 345569;
```

```
UPDATE STUDENTS SET Address = '93 Manchester Road, London SE02 6VS',
StudentName='Lily Smith'
WHERE StudentId = 345569;
```

StudentId	StudentName	GroupCode	BirthDate	Address
345569	Lily Brown	M2020_1	05/17/1999	29 The Crescent, London W20 9FK

5 Integrity constraints

We have learned how to create tables and change their structure. The column name and data type are specified for each column in the table.

We have also learned how to fill tables with values using the **INSERT** statement. Adding records to the table automatically checks the matches of the records so we can be sure that the integer field will not get a letter, and in the field with the date type cannot enter the 32nd date of the 15th month. But it may be necessary to set more severe limits on the possible values of the data than just matching the data type.

```
CREATE TABLE STUDENTS (
StudentId INTEGER,
StudentName VARCHAR(100),
GroupCode VARCHAR(32),
BirthDate DATE,
Address VARCHAR(100)
);
```

StudentId	StudentName	GroupCode	BirthDate	Address
A1345568	William Johnson	M2020_1	01/20/1999	711 Station Road, London N63 5SF
345569	Lily Brown	M2020_1	32/17/1999	93 Manchester Road, London SE02 6VS
-550	Joy9n Sirl	B2020_2	02/02/1900	951 Highfield Road, London E16 2RI

For example, it may be necessary to specify that the identifier is a positive number, and there are no digits in the surname, some fields may take values from a limited set — for example, gender is male and female, days of the week are only 7, etc.

Integrity constraints that define possible values for the data will help. There are several categories of integrity rules:

Integrity constraints

```
CREATE TABLE T1 (  
    {column name _1} {data type _1} [Limitations for column list _1],  
    {column name _2} {data type _2} [Limitations for column list _2],  
    ...  
    {column name _n} {data type _n} [Limitations for column list _n],  
    [The list of limitations]  
)
```

- limit of undefined values;
- limit of the valid value range;
- limitation of uniqueness;
- primary key limit;
- referenced integrity.

Some attributes of the entity may have an undefined value, while others must always be specified. These are required attributes without which the table row is meaningless. For example, it is difficult to imagine a student who does not have a name or a student's record book number. You can specify the NOT NULL constraint for field values to be defined always.

Limit of undefined values

```
CREATE TABLE STUDENTS (  
    StudentId INTEGER,  
    StudentName VARCHAR(100) NOT NULL,  
    GroupCode VARCHAR(32),  
    BirthDate DATE,  
    Address VARCHAR(100) NULL  
);
```

If the NOT NULL constraint is not specified, the default field values may not be set. This can be written explicitly, for example for the – field address – the word NULL next to the field means that its values may not be defined.

According to the business logic of some data, the values of some attributes in the table must be unique. For example, the **StudentId** field value that corresponds to the student's record book number must be unique. To make it impossible to repeat values in a column, you need to set the UNIQUE limit.

Limitation of uniqueness

```
CREATE TABLE STUDENTS (  
    StudentId INTEGER UNIQUE NOT NULL,  
    StudentName VARCHAR(100) NOT NULL,  
    GroupCode VARCHAR(32),  
    BirthDate DATE,  
    Address VARCHAR(100) NULL  
    UNIQUE (GroupCode, StudentName, BirthDate)  
);
```

Sometimes it is necessary to make not a separate field, but a combination of fields. For example, we require that there are no two students with the same name and the same date of birth in one group.

When restrictions affect more than one field, the constraints are described after all fields in the table. You need to specify the **UNIQUE** limit, then in parentheses list the fields whose values should be a unique combination. The field uniqueness constraint can be enhanced by adding the **NOT NULL** constraint to the **StudentId** field.

Primary key limit

```
CREATE TABLE STUDENTS (  
    StudentId INTEGER PRIMARY KEY,  
    StudentName VARCHAR(100) NOT NULL,  
    GroupCode VARCHAR(32),  
    BirthDate DATE,  
    Address VARCHAR(100) NULL  
    UNIQUE (GroupCode, StudentName, BirthDate)  
);
```

Stronger than uniqueness is the primary key constraint. This restriction requires the uniqueness of the field values, the absence of undefined values. In addition, there can be only one primary key in the table. But it should be noted that the primary key can be a combination of several columns of the table.

A very important constraint that allows you to store relationships in tables is referential integrity. To demonstrate it, consider another table **ST_GROUP**. Each student belongs to a certain group.

Primary key limit

```
CREATE TABLE EXAM_RESULT
  (StudentId INTEGER NOT NULL,
   ExamSheetId INTEGER NOT NULL,
   Mark INTEGER,
   PRIMARY KEY(StudentId, ExamSheetId)
);
```

Look at the contents of the tables STUDENTS and ST_GROUP. William and

Referenced integrity

Students

StudentId	StudentName	GroupCode	BirthDate	Address
1345568	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
130568	John Sirl	M2020_2	02/02/1999	951 Highfield Road, London E16 2RI

St_group

M2020_1	Nanotechnology
M2020_1	Health Research
B2020_2	Art&Science

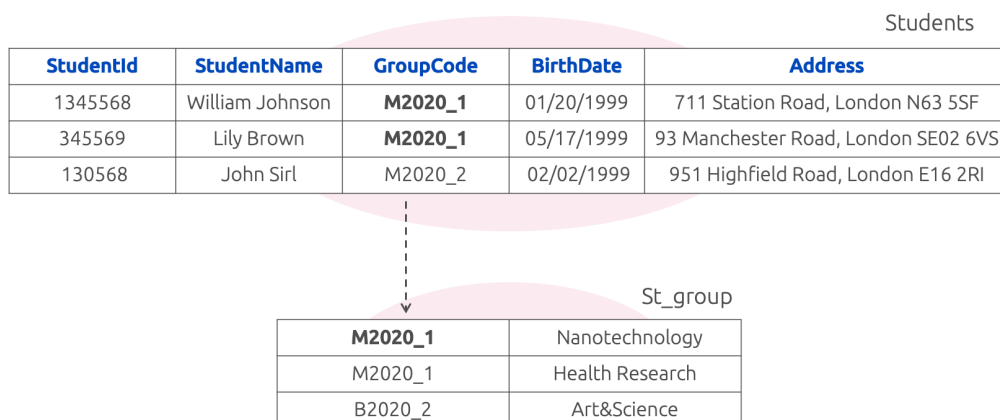
Lily study in the group M2020_1. From the table ST_GROUP we can find their specialization. But how to find John's specialization? He studies in the group M2020_2, but there is no such group in the table ST_GROUP. We see that the logic of the data is broken. You cannot specify a group for a student that is not in the list of groups. To prevent this from happening, it is necessary to link the columns of the two tables with the appropriate integrity rule.

Referenced integrity

```
CREATE TABLE ST_GROUP (
    GroupCode VARCHAR(32) PRIMARY KEY,
    Specialization VARCHAR(100) NOT NULL
);

CREATE TABLE STUDENTS (
    StudentId INTEGER PRIMARY KEY,
    StudentName VARCHAR(100) NOT NULL,
    GroupCode VARCHAR(32) REFERENCES ST_GROUP(GroupCode),
    BirthDate DATE,
    Address VARCHAR(100) NULL;
    UNIQUE (GroupCode, StudentName, BirthDate)
);
```

Let's set a foreign key for the column with the group code in the **STUDENTS** table – the column **GroupCode** from the **ST_GROUP** table. Such restrictions will



not allow adding to the **STUDENTS** table values in the **GroupCode** field that are not present in the corresponding column of the linked table. The table **ST_GROUP** is also called the parent.

Note that the external key must be of the same type as the linked column – in our case they both have **VARCHAR(32)**.

In addition, the foreign key field must be declared either a unique value or a primary key.

If the rows of the parent table **ST_GROUP** are changed or deleted the corresponding rows in the **STUDENTS** table are automatically searched. Suppose we want to delete from the table **ST_GROUP** the group **M2020_1** specialty **Nanotechnology**.

Then, depending on the referential integrity settings, several options are possible: for example, you can prohibit the deletion of this group, because there are students associated with it; or you can delete all related records of dependent

tables – when deleting a group, delete students who study in it. The last type of restrictions that we will consider is related to specifying an acceptable range of values.

Limit of the valid value range

```
CREATE TABLE STUDENTS (  
    StudentId INTEGER PRIMARY KEY CHECK (StudentId>0),  
    StudentName VARCHAR(100) NOT NULL,  
    GroupCode VARCHAR(32) REFERENCES ST_GROUP(GroupCode) ,  
    BirthDate DATE,  
    Address VARCHAR(100) NULL;  
    UNIQUE (GroupCode, StudentName, BirthDate)  
);
```

This restriction is indicated by the keyword **CHECK()** after the attribute description. If you want to verify that several simple conditions are met, you can combine them with logical operators, for example: **CHECK(StudentID>5 AND StudentID<500)**

When you add, delete, and edit records in tables, all specified integrity limits are automatically checked.

In this way, integrity constraints will prevent incorrect values from entering the database.