Fundamentals of database design

Creating a database and tables

The quality of the database design can work with it. A well-designed database is easier to work with, easier to write queries. And in this guide, we'll cover the basics of database design.

For high-quality database design, there are different techniques, different sequences of steps or stages, which are similar in many ways. And in general, we can distinguish the following stages:

1. Selection of entities and their attributes, which will be stored in the database, and the formation of tables from them. Atomization of complex attributes is simpler.
2. Definition of unique identifiers (primary keys) of objects stored in table rows
3. Defining relationships between tables using foreign keys
4. Database normalization

In the first stage, entities are selected. An entity is a type of objects that must be stored in a database. Each table in the database must represent one entity. As a rule, entities correspond to objects from the real world.

Each entity defines a set of attributes. An attribute is a property that describes some characteristic of an object.

Each column must store one entity attribute. And each line represents a separate object or entity instance.

Top-down and bottom-up approaches

When designing a database at the stage of selecting entities and their attributes, we can use two approaches: top-down and top-down.

The bottom-up approach involves selecting the necessary attributes that must be stored in the database. Then the selected attributes are grouped by essence, which subsequently creates a table. This approach is most suitable for designing small databases with a small number of attributes.

For example, we are given the following information:

```
Tom attends a math class taught by Professor Smith.
Sam attends a math class taught by Professor Smith.
Tom attends a JavaScript course taught by Assistant Adams.
Bob attends a class on algorithms taught by Assistant Adams.
Sam has the following email address sam@gmail.com and phone +1235768789.
```

What data can we save: student's name, course name, teacher's teaching position, teacher's name, student's email address.

We can then perform groupings on the basis that this data belongs to:

Student	Teacher	Course
Name of the student	The teacher's name	Name of the student
Name of the course	Teacher position	The teacher's name
Date of birth of the student	Name of the course	Name of the course
Student's email address		
Student phone number		

Yes, the available data allow us to distinguish three entities: the student, the teacher and the course. At the same time, we may well add some missing data. It should also be noted that some data may refer to different entities. For example, a course stores information about the student who attends it. And the student saves information about the attended course. Such data redundancy is resolved in subsequent design steps in the database normalization process.

But there can be many such attributes: hundreds and even thousands. And in this case, a top-down approach will be more optimal. This approach involves entity detection. Then the entities are analyzed, the connection between them and the attributes of the entities are revealed.

That is, in this case, we could immediately determine that we need to store data about students, courses and teachers. Then, within each entity, identify attributes

For example, in the "Student" entity, we could highlight such attributes as the student's name, address, phone, height, weight, year of birth. At the same time, we need to take into account not all the properties that, in principle, can be in the "Student" entity, but only those that are important within the framework of the described system. Properties such as student height or weight are unlikely to play a role in this case, so we can cross them off the list of attributes when designing the table.

Sometimes approaches are combined. Different approaches can be used to describe different parts of the system. And then their results are combined.

Atomization of attributes

When defining attributes, complex complex elements are divided into simpler ones. Yes, in the case of a student's name, we can break it down into first and last name. This will allow later to perform operations with these sub-elements separately, for example, to sort students only by last name.

The same applies to the address - we can save the entire address as a whole, or we can break it into parts - house, street, city, etc.

At the same time, it is possible to divide one element into sub-elements, which can always be in demand. In a number of tasks, this may simply not be necessary. It is necessary to highlight only those elements that are really needed.

According to this aspect, we can highlight the following attributes in the "Student" entity: student's name, student's last name, year of birth, city, street, house, phone.

Domain

Each attribute has a domain. A domain represents a set of valid values for one or more attributes. Essentially, the domain defines the content and source of values that attributes can have.

Domains can be different for different attributes, but multiple attributes can have the same domain.

For example, the attributes of the student entity were defined above. Let's define the used domains:

- **Name**. A domain represents all possible names that can be used. Each name is a string with a maximum length of 20 characters (it is unlikely that we will encounter names with more than 20 characters).
- **Name**. A domain represents all possible surnames that can be used. Each last name is a string with a maximum length of 20 characters.
- **Year of birth**. The domain represents all birth years. Each year is a numeric value from 1950 to 2017.

- **City**. The domain represents all the cities of the current country. Each city represents a string of maximum 50 characters.
- **Street**. The domain represents all the streets of the current country. Each street is a string with a maximum length of 50 characters.
- **House**. The domain represents all possible house numbers. Each house number is a number from 1 to, say, 10,000.
- **Phone**. The domain represents all phone numbers. Each number is a string of 11 characters.

By defining a domain, we immediately see what data and what types the attributes will store. The attribute cannot have any other value that corresponds to the domain. In the example above, each attribute has its own domain. But domains can match. For example, if an entity contained the following two attributes: city of birth and city of residence, then the domain would match and be the same for both attributes.

NULL specifier

When defining attributes and their domain, it is necessary to analyze whether the attribute may have a missing value. The NULL specifier allows you to set the absence of a value. For example, in the example above, the student must have a name, so the situation where the attribute representing the name has no value is unacceptable.

At the same time, a student may not have a phone number or a phone is not required within the system. Therefore, at the design stage of the table, you can specify that this attribute allows NULL values.

As a rule, most modern relational DBMSs support the NULL specifier and allow setting its validity for a table column.

Keys

Keys are a way to identify rows in a table. We can also use keys to link rows between different tables in a relationship.

super key

Superkey is a combination of attributes (columns) that uniquely identify each row of the table. It can be all columns, and several, and one. At the same time, the lines containing the values of these attributes should not be repeated.

For example, we have a Student entity that provides user data and has the following attributes:

- FirstName (name)
- LastName
- Year (year of birth)
- Phone (phone number)

What attributes can make up a super key in this case:

- `{FirstName, LastName, Year, Phone}`
- `{FirstName, Year, Phone}`
- `{LastName, Year, Phone}`
- `{FirstName, Phone}`
- `{LastName, Phone}`
- `{Year, Phone}`
- `{Phone}`

Each student can uniquely identify a phone number, so any sets that contain the Phone attribute represent a superkey.

But, for example, the set {FirstName, LastName, Year} is not a superkey, because theoretically we can have at least two students with the same first name, last name, and year of birth.

A potential key

Candidate key (potential key) is the minimal superkey of a relation (table), that is, a set of attributes that satisfies a number of conditions:

- **Irreducibility**: it cannot be shortened, it contains the minimum possible set of attributes
- **Uniqueness**: it must have unique values regardless of row changes
- **Presence of meaning**: it must be NULL, i.e. it must have a value.

Let's take the previously selected super keys and find the candidate key among them. The first five superkeys do not satisfy the first condition because they can all be reduced to a superkey.

- `{FirstName, LastName, Year, Phone}`
- `{FirstName, Year, Phone}`
- `{LastName, Year, Phone}`
- `{FirstName, Phone}`
- `{LastName, Phone}`
- `{Year, Phone}`

The {Phone} superkey satisfies the first and second conditions because it has a unique value (in this case, all users can only have unique phone numbers). But does it meet the third condition? In general, no, because theoretically a student may not have a phone. In this case, the Phone attribute will be NULL, meaning there will be no value.

At the same time, it may depend on the situation. If in some system the phone number is an integral attribute, for example, used for registration and login, it can be considered a potential key. But in this case we are looking at the general situation. And the understanding of the potential key must be based on the specific system that the database defines.

And in this case, the superkeys of the table do not contain a potential key.

Primary key

The primary key is directly used to identify table rows. It must meet the following restrictions:

- The primary key must be unique at all times
- It must be present in the table and have a value
- It should change its meaning often. Ideally, it should not change values at all.

As a rule, the primary key represents one column of the table, but it can be composite and consist of several columns.

If a potential key can be extracted from the table, it can be used as the primary key. If there are no potential keys, then a special attribute can be added to the entity for the primary key, which is usually called Id or has the form [Entity_name]Id (for

example, StudentId), or can have another name. And usually this attribute acquires an integer value starting from 1.

If we have several potential keys, then the potential keys that are not the primary key are alternative keys.

For example, let's take user submissions on sites with two-factor authentication, where we must have an email address, which often acts as a login, and some phone number. In this case, we can specify the user table using the following attributes:

- Name (user name)
- Email (e-mail address)
- Password
- Phone (phone number)

In this case, the attributes Email and Phone are potential keys, they are mandatory within the framework of the system under consideration and, in principle, are unique. And in theory, we could use one of these attributes as the primary key, then the other would be the alternate key. However, since the values of both attributes can theoretically change, it is still better to define an additional attribute specifically for the primary key.

Foreign keys and relationships

Databases can contain tables connected by various relationships. A relationship is an association between entities of different types.

When selecting a relationship, the main or parent table (primary key table/master table) and dependent, child table (foreign key table/child table) are selected. The child table depends on the parent table.

External keys are used to organize communication. A foreign key is one or more columns from one table that is also a potential key from another table. The foreign key does not necessarily have to match the primary key from the main table. Although, typically, a foreign key from a dependent table points to a primary key from the parent table.

Connections between tables are of the following types:

- **To each other**(One to one)
- **One to many**(One to many)
- **Many to many**(Many to many)

One to one communication

This type of connection is not common. And here, only one object of another entity can be compared to an object of one entity. For example, on some sites a user can only have one blog. That is, there is a relation of one user – one blog.

Often, this type of connection involves splitting one large table into several small ones. The main parent table in this case continues to contain frequently used data, while the dependent child table usually stores less frequently used data.

In this regard, the primary key of the dependent table is at the same time a foreign key that refers to the primary key from the main table.

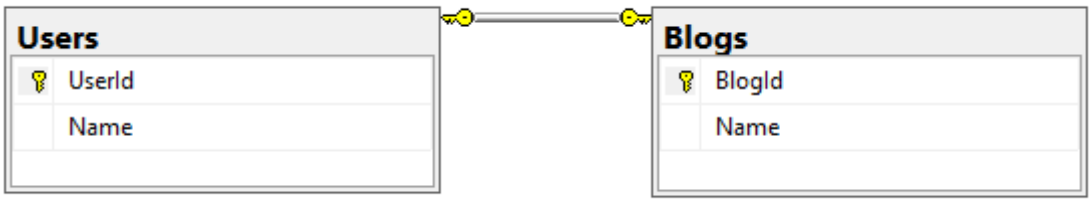
For example, the Users table represents users and has the following columns:

- UserId (identifier, primary key)
- Name (user name)

And the Blogs table represents user blogs and has the following columns:

- BlogId (identifier, primary and foreign key)
- Name (blog name)

In this case, the BlogId column will store the value from the UserId column from the Users table. That is, the BlogId column will act as a primary and foreign key at the same time.



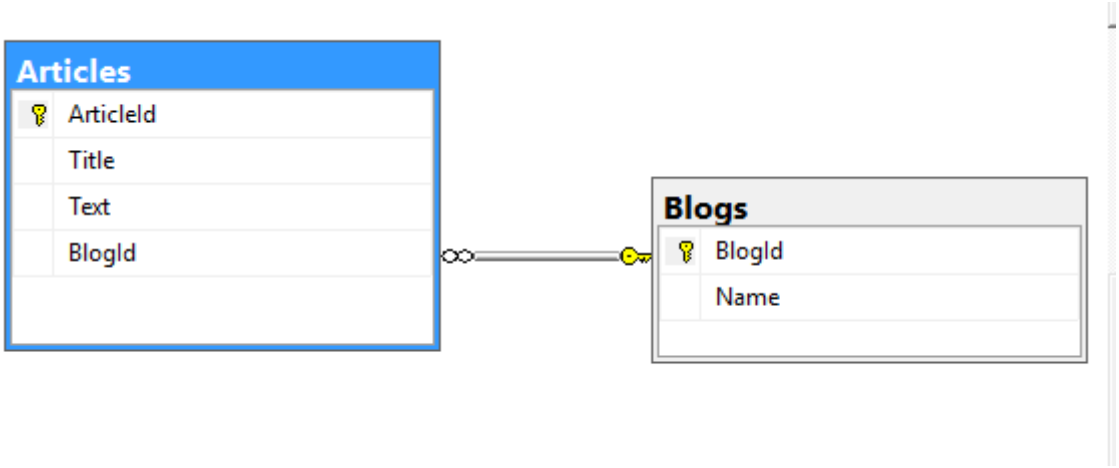
One-to-many connection

This is the most common type of connection. In this type of relationship, multiple rows from the child table depend on a single row from the parent table. For example, one blog can have several articles. In this case, the Blogs table is the parent and the

Articles table is the child. That is, one blog - many articles. Or another example, a football team can have several football players. And at the same time, one football player can play in only one team at a time. That is, one team - many football players. For example, let's have an Articles table that represents blog articles and has the following columns:

- ArticleId (identifier, primary key)
- BlogId (foreign key)
- Title (title of the article)
- Text (article text)

In this case, the BlogId column from the Articles table will store the value from the BlogId column from the Blogs table.



Many-to-many communication

With this type of relationship, one row from table A can be associated with many rows from table B. In turn, one row from table A can be associated with many rows from table A. A typical example is students and courses: one student can attend several courses and, accordingly, several students can enroll in one course.

Another example is articles and tags: multiple tags can be defined for one article, and one tag can be defined for multiple articles.

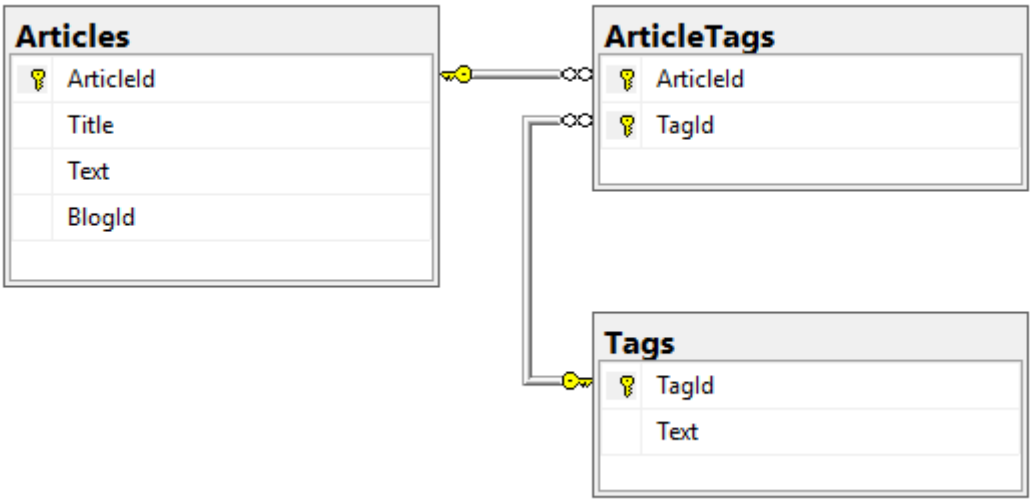
But in SQL Server, at the database level, we cannot establish a one-to-many relationship between two tables. This is done in the form of an auxiliary intermediate table. Sometimes the data in this intermediate table represents a separate entity.

For example, in the case of articles and tags, let there be a Tags table that has two columns:

- TagId (identifier, primary key)
- Text (tag text)

Also, let there be an intermediate table ArticleTags with the following fields:

- TagId (identifier, primary and foreign key)
- ArticleIdId (identifier, primary and foreign key)



Technically, we will get two one-to-many relationships. The TagId column from the ArticleTags table will reference the TagId column from the Tags table. And the ArticleId column from the ArticleTags table will reference the ArticleId column from the Articles table. That is, the TagId and ArticleId columns in the ArticleTags table represent a composite primary key and at the same time are external keys for communication with the Articles and Tags tables.

Referential data integrity

When changing primary and foreign keys, you should observe such an aspect as referential integrity of data (referential integrity). Its main idea is that two tables in the database that store the same data maintain their consistency. Data integrity represents correctly constructed relationships between tables with the correct setting of links between them. In which cases data integrity may be violated:

- **Withdrawal anomaly**(deletion anomaly). Occurs when a row is deleted from the main table. In this case, the foreign key from the dependent table continues to refer to the remote row from the parent table
- **Insertion anomaly**(insertion anomaly). Occurs when inserting a row into a dependent table. In this case, the foreign key from the dependent table does not match the primary key of any of the rows from the main table.
- **Update anomalies**(update anomaly). With such an anomaly, several rows of the same table may contain data belonging to one and the same object. When data is changed in one row, it may conflict with data from another row.

Deletion anomaly

To resolve the foreign key deletion anomaly, one of two constraints should be set:

- If a row from a dependent table necessarily requires a row from the main table, then cascading deletion is set for the foreign key. That is, when a row is deleted from the main table, the associated row(s) is deleted from the dependent table.
- If a row from a dependent table allows no connection with a row from the main table (that is, such a connection is optional), then the foreign key is set to NULL when the related row is deleted from the main table. In this, the foreign key column must be NULL.

Insertion anomaly

To resolve an insert anomaly when adding data to a dependent table, the column that represents the foreign key must be NULL. And thus, if the added object has no connection with the main table, then the foreign key column will have a NULL value.

Update anomalies

To solve the problem of the update anomaly, normalization is used, which will be discussed later.

Normalization

Normalization is the process of dividing data into separate tables. Normalization eliminates data redundancy (data redundancy) and thus avoids the violation of data integrity due to its change, that is, avoids change anomalies (update anomaly).

As a rule, normalization is mainly used in the bottom-up approach to database design, that is, when we group all the attributes that need to be stored in the database, for which tables are then created. However, with a bottom-up approach, when entities are first discovered, and then their attributes and relationships between them, normalization can also be used, for example, to check the correctness of designed tables.

In non-normalized form, the table can store information about two or more entities. It can also contain repeating columns. Columns can also store repeating values. In normalized form, each table stores information about only one entity.

Normalization involves the use of normal forms of the data structure. There are 7 normal forms. Each normal form (except the first) implies that a previous normal form has already been applied to the data. For example, before applying the third normal form of the data, the second normal form must be applied. And, strictly speaking, a database is considered normalized if the third normal form or higher is applied to it.

First Normal Form (1NF) dictates that the data stored at the intersection of rows and columns must represent a scalar value, and tables must not contain repeating rows.

Second normal form (2NF) requires that each column that is a key must depend on the primary key.

Third normal form (3NF) states that each column that is a key must depend only on the primary key.

The Boyce-Codd Normal Form (BCNF) is a slightly stricter version of the third normal form.

The fourth normal form (4NF) is used to eliminate multivalued dependencies - such dependencies where a column with a primary key has a one-to-many relationship

with a column that is not a key. This normal form eliminates incorrect relations by many, many.

Fifth normal form (5NF) divides tables into smaller tables to eliminate data redundancy. Partitioning continues until it is possible to reproduce the original table by combining small tables.

Sixth normal form (domain key normal form/6NF). Each relationship constraint between tables must depend only on key constraints and domain constraints, where the domain represents the set of valid values for a column. This form prevents the addition of invalid data by setting the constraint only at the table relationship level, but from the table or column level. This form is usually not applicable only at the DBMS level, particularly in SQL Server.

Functional dependence

The key concept of normalization is functional dependence. Functional dependence defines the connection between the attributes of the relationship. For example, if attribute B is functionally dependent on attribute A (A → B), each value of attribute A is associated with only one value of attribute B. Moreover, attributes A can consist of one or more attributes. That is, if two lines have the same value of attribute A, then they necessarily have the same value of attribute B. At the same time, for one attribute value, there may be several different values of attribute A. Attribute A in this dependence is also called a determinant .

For example, consider the following table representing university courses:

Course	Teacher	Position
Mathematics	Smith	Professor
Algorithms	Adams	Assistant
JavaScript	Adams	Assistant

Here, the Teacher attribute is functionally dependent on the Course attribute (Course → Teacher). That is, knowing the name of the course, we can determine its teacher. And in this case, we can say that there is a 1:1 relationship between the Course and Teacher attributes, and a 1:N relationship between Teacher and Course, since there

are several courses that can be taught by one teacher. At the same time, the Course attribute does not depend on the Teacher attribute.

In addition, two more functional dependencies can be traced here. In particular, the Position attribute depends on the Teacher attribute (Teacher → Position). Knowing the teacher's name, we can determine his position.

Also, the Position attribute is functionally dependent on the Course attribute - knowing the name of the course, we can tell the teacher's position.

In a table in a normalized database, the only determinant must be an attribute that is the primary key. And the rest of the attributes must be functionally dependent on the primary key.

For example, in this case, we can take the name of the course as the primary key, taking into account that courses can only have unique names. However, the position of a teacher in this case will depend on two attributes at once - Course and Teacher. And similar dependencies may indicate that the database and the course table in particular have design flaws and may be the source of update anomalies.

The first normal form

First normal form dictates that the table must not contain repeating columns or columns that contain sets of values. A non-normalized table in this case may contain one or more groups of repeating data. A repeating group is a group of one or more table attributes that can have multiple values for a key table attribute.

The result of applying the first form should be the presence for one entity attribute of only one column in the table, which should contain a scalar value.

There are two steps to go to the non-normalized table to the first normal form. The first method is called flattening. It involves decomposing a string with repeating groups of data, in which a separate row is created for each repeating group. The resulting table will contain the atomic values of each of the attributes. Although at the same time this approach will increase data redundancy.

The second approach involves assigning a single attribute or group of attributes to the key of the non-normalized table, and then removing the duplicate groups from

the table and placing them in a separate table with copies of the key from the original table.

Let's consider the application of normalization from an example. Let us have a system described by the following information:

```
1

2   Tom attends a math class taught by Smith. Record date 11/06/2017.
    Sam attends a class on algorithms taught by Adams. Date of recording
    12/06/2017.
3   Bob attends a math class taught by Smith. Record date 06/13/2017.
    Tom attends a JavaScript class taught by Adams. Record date 06/14/2017.
4   Sam has two email addresses: sam@gmail.com and sam@hotmail.com.
    A university can only have one course with a specific name. One teacher
5   can teach several courses.
6
```

First, let's define a non-normalized StudentCourses table that contains all of this information:

StudentId	Name	Emails	Course1	Date1	TeacherId1	Teacher1	Course2	Date2	TeacherId2	Teacher2
1	Tom		Mathematics	11/06/2017	1	Smith	JavaScript	14/06/2017	2	Adams
2	Sam	sam@gmail.com sam@hotmail.com	Algorithms	12/06/2017	2	Adams				
3	Bob		Mathematics	13/06/2017	1	Smith				

A unique StudentId identifier is defined for each student, as well as the attribute Name (name), Emails (all email addresses), Course1/Course2 (course), Date1/Date2 (date of entry), Teacher1/Teacher2 (teacher). Also, to distinguish between teachers (since theoretically there can be teachers with the same last name), the attribute TeacherId1/TeacherId2 is added. For courses, such an identifier is not needed, because in our case the name of the course is unique.

Since Tom is enrolled in two courses at once, several attributes had to be duplicated. But what will happen when Tom, in an effort to get certificates that no one needs, enrolls in a dozen more courses?

This table is an excellent example of a deviation from the first normal form. First of all, we see a group of repeating attributes that represent data for one course: Course, Date, TeacherId, Teacher. These attributes represent a repeating group that can be tentatively named StudentCourse.

```
StudentCourse = (Course, Date, TeacherId, Teacher)
```

The second problem is that the Emails attribute contains a set of email addresses. In fact, this attribute also forms a repeating group.

To get rid of the first repeating group of attributes, we will use the first approach: we will create a separate line for each repeating group.

StudentId	Name	Emails	CourseId	Course	Date	TeacherId	Teacher
1	volume		1	Mathematics	11/06/2017	1	Smith
1	volume		2	JavaScript	14/06/2017	2	Adams
2	Sam	sam@gmail.com sam @hotmail.com	3	Algorithms	12/06/2017	2	Adams
3	Bob		1	Mathematics	13/06/2017	1	Smith

In this case, data redundancy increased, but we got rid of the duplicate group. It should also be noted that the StudentId attribute cannot now be used as a primary key. And in this case, only one potential key is visible, which will be used as the primary key - it is two columns StudentId and Course. But the name of the course is not the best clue, considering that this name can be edited and changed. Therefore, one more attribute is added for each course - CourseId - a unique course number, which together with StudentId forms the primary key. Although, in principle, the name of the course could be left as part of the primary key, taking into account that it is unique.

To get rid of the second repeating group - the Emails attribute, we will use the second approach: moving this group with a copy of the key to a separate table. To do this, define the Emails table:

Email	StudentId
sam@gmail.com	2
sam@hotmail.com	2

Since the e-mail address is in principle unique, it can be made the primary key.

Thus, the Emails table will have a one-to-many relationship with the StudentCourses table (one student - many email addresses). And in this case, the StudentCourses table will be shortened like this:

StudentId	Name	CourseId	Course	Date	TeacherId	Teacher

1	volume	1	Mathematics	11/06/2017	1	Smith
1	volume	2	JavaScript	14/06/2017	2	Adams
2	Sam	3	Algorithms	12/06/2017	2	Adams
3	Bob	1	Mathematics	13/06/2017	1	Smith

Now we don't have any repeating columns, but the data redundancy has increased, since student Tom already has two rows defined in the table, and the Id is repeating accordingly. However, the 1st normal form is applied.

In principle, it can be noted that if repeating groups contain unique values for each row of the table (as in the case of e-mail addresses), then we are dealing with a potential one-to-many relationship. If repeating groups contain non-unique values that can have different rows of the table (as in the case of course attributes), then this hides a potential many-to-many relationship.

The second normal form

In second normal form, every column in the table that is a key must depend on the key .

The key point of the second normal form is complete functional dependence. It assumes that attribute B is fully functionally dependent on attribute A if attribute B is functionally dependent on the full value of attribute A and not on any subset of values from attribute A. That is, if attribute A consists of several values, say A1 and A2, then the attribute is completely functionally dependent on A, if it depends on both A1 and A2 (A1, A2 → B).

If the attribute depends only on any subset of attribute A, for example, only on A1, then there is a partial functional dependence.

This form applies to tables that have a composite primary key, that is, where the primary key consists of several attributes. If the table does not have a primary key, then in this case it is considered that the other attributes are automatically fully functionally dependent on the primary key.

Second normal form applies only to tables that are in first normal form. After applying the second form, all columns of the table depend on the primary key.

Let's take the StudentCourses table formed in the previous topic after applying the first normal form:

StudentId	Name	CourseId	Course	Date	TeacherId	Teacher
1	volume	1	Mathematics	11/06/2017	1	Smith
1	volume	2	JavaScript	14/06/2017	2	Adams
2	Sam	3	Algorithms	12/06/2017	2	Adams
3	Bob	1	Mathematics	13/06/2017	1	Smith

This table currently has a composite primary key of StudentId+CourseId. What functional dependencies on key attributes can be highlighted here:

```
StudentId, CourseId → Date
StudentId → Name
CourseId → Course, TeacherId, Teacher
```

Only the Date attribute depends on both parts of the composite key StudentId+CourseId - the date on which the student with the identifier StudentId enrolled in the course with the identifier CourseId.

The Name attribute depends only on a part of the composite key - on the StudentId attribute, because knowing the student's ID, you can say what the name is. In this case, there is a fact of partial dependence.

The Course, TeacherId, Teacher, Position attributes depend on another part of the key – the CourseId attribute. Knowing the value of CourseId, you can tell what the course is called, what the teacher is in the course, what position he holds. Again, there is a partial dependency here.

The presence of partial dependencies indicates that the table is not in the second normal form. And to move to this form, you need to move the attributes that are not part of the primary key to a new table, along with a copy of the part of the primary key on which they functionally depend.

In our case, three will come out of one table. Students table:

StudentId	Name
1	volume
2	Sam
3	Bob

Courses table:

CourseId	Course	TeacherId	Teacher
1	Mathematics	1	Smith
2	JavaScript	2	Adams
3	Algorithms	2	Adams

And the StudentCourses table:

StudentId	CourseId	Date
1	1	11/06/2017
1	2	14/06/2017
2	3	12/06/2017
3	1	13/06/2017

The result was the formation of a many-to-many relationship (many students - many courses) between the Students and Courses tables through the StudentCourses table. Thus, the database moved to the second normal form.

The third normal form

The third normal form assumes that each column that is not a key must depend only on a column that is a key, that is, there must be no transitive functional dependency. Transitive functional dependence is expressed as follows: A → B and B → C. That is, attribute C transitively depends on attribute A, if the attribute depends on attribute B, and the attribute depends on attribute A (provided that attribute A does not functionally depend on either attribute, not from attribute C).

If a column depends on more than just the primary key, then that column is not in the table it should be in, or is derived from other columns.

For normalization, the attributes transitively dependent on the key are transferred from the source table to a separate table with a copy of the attribute on which they directly depend.

When using the third normal form, the table must be in the second normal form. 3NF significantly reduces data redundancy.

For example, let's take the Courses table created in the previous topic, which contains information about courses and is in the second normal form:

CourseId	Course	TeacherId	Teacher

1	Mathematics	1	Smith
2	JavaScript	2	Adams
3	Algorithms	2	Adams

What functional dependencies can be distinguished here:

CourseId → Course, TeacherId, Teacher

Course → CourseId, TeacherId, Teacher

TeacherId → Teacher

The second dependency is actually similar to the first one and indicates that the Course attribute is a potential key.

The third dependency says that, knowing the teacher's ID, we can learn about his last name and title. That is, through the TeacherId attribute, the Teacher attribute depends on the CourseId (CourseId → TeacherId and TeacherId → Teacher). And in this case, we can talk about the transitive dependence of Teacher, Position on CourseId.

For normalization, it is necessary to make a separate table for the TeacherId and Teacher attributes. For this, let there be a separate table Teachers:

TeacherId	Teacher
1	Smith
2	Adams

And the Courses table will be shortened like this:

CourseId	Course	TeacherId
1	Mathematics	1
2	JavaScript	2
3	Algorithms	2