Learning Objectives

Learners will be able to…

- Create links between tables with FOREIGN KEY
- Describe possible cascading actions with data when there are links between tables
- Changing table structures
- Explain database normalization

info

Make Sure You Know

Learners should be familiar with SQL statements for creating tables, defining constraints, and performing basic data manipulation using JOINs,how to define a **FOREIGN KEY**, and comprehend the importance of planning and considerations when altering table structures as these skills are foundational to understanding the concepts mentioned above.

FOREIGN KEY

Since tables in a database usually exist in relation to each other, a mechanism is needed to control their relationship to each other. There is a foreign key constraint for this.

The **FOREIGN KEY** constraint is used to prevent actions that would destroy links between tables. This is a field (or several fields) that refer to the primary key of another table.

Syntax

The syntax may differ from DBMS to DBMS, in the one we are considering there may be the following options.

Assuming this table:

```
CREATE TABLE department (
  id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  name VARCHAR(40) DEFAULT 'unknown'
);
```

There are three different ways to define a foreign key (when dealing with a single PK column) during table creation, and they all result in the same foreign key constraint:

1. In-line without mentioning the target column(primary key will be used):
   ```
   CREATE TABLE employee (       id INTEGER GENERATED BY DEFAULT AS
   IDENTITY PRIMARY KEY,       first_name VARCHAR(50) NOT NULL,
   last_name VARCHAR(50) NOT NULL,       email VARCHAR (100) NOT
   NULL,       salary INT NOT NULL,       department_id INTEGER
   REFERENCES department     );
   ```
2. In-line with mentioning the target column:
   ```
   CREATE TABLE employee (       id INTEGER GENERATED BY DEFAULT AS
   IDENTITY PRIMARY KEY,       first_name VARCHAR(50) NOT NULL,
   last_name VARCHAR(50) NOT NULL,       email VARCHAR (100) NOT
   NULL,       salary INT NOT NULL,       department_id INTEGER
   REFERENCES department (id)     );
   ```
3. Out of line inside the create table:
   ```
   CREATE TABLE employee (       id INTEGER GENERATED BY DEFAULT AS
   IDENTITY PRIMARY KEY,       first_name VARCHAR(50) NOT NULL,
   last_name VARCHAR(50) NOT NULL,       email VARCHAR (100) NOT
   NULL,       salary INT NOT NULL,       department_id INTEGER,
   ```

```
  CONSTRAINT fk_department_id         FOREIGN KEY (department_id)
  REFERENCES department (id)     );
```

info

Composite foreign key

Please note that you can create a composite FOREIGN KEY only in the third way - only there you can list several fields.

After creating such a link, the DBMS will automatically control the integrity of such links when trying to delete or change.

Foreign key in action

Create described structure by own or run a command below.

Try to delete department with id 1.

```
DELETE FROM department WHERE id = 1;
```

You will receive error about violates foreign key constraint.

```
ERROR:  update or delete on table "department" violates foreign
key constraint "fk_department_id" on table "employee"
DETAIL:  Key (id)=(1) is still referenced from table "employee".
```

Referential actions

In the previous step, we got an error when we tried to delete a record. But this is not the only possible action that can be taken when a record is deleted or modified.

The foreign key description can contain two additional actions that are performed when an attempt is made to delete or update.

```
[ON DELETE delete_action]
[ON UPDATE update_action]
```

The delete and update actions determine the behaviors when the primary key in the parent table is deleted and updated. Since the primary key is rarely updated, the **ON UPDATE** action is not often used in practice. Most often used the **ON DELETE** action.

PostgreSQL supports the following actions:

- **NO ACTION** - if any referencing rows still exist when the constraint is checked, an error is raised
- **RESTRICT** - it prevents deletion of a referenced row, difference between no action only in transaction mode - it will fail check immediately.
- **SET NULL** - automatically sets NULL to the foreign key columns in the referencing rows of the child table when the referenced rows in the parent table are deleted.
- **SET DEFAULT** -automatically sets default value to the foreign key columns in the referencing rows of the child table when the referenced rows in the parent table are deleted.
- **CASCADE** - specifies that when a referenced row is deleted, row(s) referencing it should be automatically deleted as well

The default action is **NO ACTION**, which is what we saw when we tried to delete it.

```
CREATE TABLE employee (
  id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  first_name VARCHAR(50) NOT NULL,
  last_name VARCHAR(50) NOT NULL,
  email VARCHAR (100) NOT NULL,
  salary INT NOT NULL,
  department_id INTEGER,
  CONSTRAINT fk_department_id
    FOREIGN KEY (department_id)
    REFERENCES department (id)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);
```

The example above will create a reference in which when you try to delete a department, the associated employees will also be deleted, and when you update, the identifier will be updated.

You could update table definition in `create.sql` drop structure, recreate it and try to delete with new action.

ALTER TABLE

If you created a table and then realized that you made a mistake, or if you received new requirements from the customer, you can delete it and create it again. But if the table is already filled with data, or if it is referenced by other database objects, this can be difficult.

To change the structure of an already existing table, use the ALTER TABLE statement.

With it, you can add and remove columns and constraints, change default values and types of columns, and rename columns and tables.

Add new column

```
ALTER TABLE table_name
ADD column_name datatype;
```

Example below will add to `employee` table a new column `hire_date` with datatype `TIMESTAMP WITHOUT TIME ZONE` and default value of current date and time.

```
ALTER TABLE employee
ADD COLUMN hire_date TIMESTAMP WITHOUT TIME ZONE
DEFAULT now() NOT NULL
```

In this case, you can immediately define additional column restrictions, similar to adding restrictions when creating a table(`NOT NULL` was added).

Rename existing column

To rename a column in a table, use the following syntax.

```
ALTER TABLE table_name
RENAME COLUMN old_name to new_name;
```

```
ALTER TABLE employee
ALTER COLUMN hire_date TO employee_hire_date;
```

Remove existing column

You can delete a column like this.

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

This will remove all data from the deleted column.

```
ALTER TABLE employee
DROP COLUMN hire_date;
```

Add new constraint

If you need to add a new constraint to the table, then this can be done in the following way.

```
-- age should be positive
ALTER TABLE employee ADD CHECK (age >= 0);

-- email should be unique
ALTER TABLE employee ADD CONSTRAINT uq_email UNIQUE (email);

-- add foreign key for manager_id field
ALTER TABLE employee ADD CONSTRAINT fk_manager_id FOREIGN KEY
        (some_id) REFERENCES manager_table;

-- name of department should not be null
ALTER TABLE department ALTER COLUMN name SET NOT NULL;
```

To clarify the syntax, it is best to refer to the documentation of the DBMS used.

Remove constraint

The constraint is removed by its name.

```
ALTER TABLE employee DROP CONSTRAINT uq_email;
```

This way you can remove constraints of any type, except NOT NULL (they have no names).

```
ALTER TABLE department ALTER COLUMN name DROP NOT NULL;
```

Database Normalization

Relational databases are based on a mathematical basis that uses sets. With it, you can transform data so that it takes up less space, as well as searching by elements is fast and efficient. This process is called normalization.

info

Normalization

Normalization is the process to eliminate data redundancy and enhance data integrity in the table.

Suppose that we keep a record of which parts are in cars and how many have been repaired by workers. As a result, we may end up with a table with repeated names and car parts in it, which is not very efficient in terms of data storage.

Worker	Part name	Part count
Henry Adamson	Brake Disc	1
Henry Adamson	Spark Plug	2
Henry Adamson	Brake Disc	3
Henry Adamson	Oil Pump	1
Connor Wilson	Brake Disc	2
Connor Wilson	Spark Plug	1

Now imagine that our table will have a size of a million records and only two of these workers will repeat in it. They have a name length of 13 characters, so just to store this information we need 13 million bytes of free disk space.

If we transfer the last names to a separate table and store the primary key instead as an integer value, instead of 12 bytes we will use from 1 to 4 bytes of data (depending on the type). Thus, we can get a benefit of 13 to 3 times with a little more than just optimizing the storage of names. Of course, 2 workers is not so much, but even if we have 10,000 of them, it is still more profitable than storing the same values one million times.

And we can carry out the same operation with the names of parts of the car. As a result, we get a table that is divided into parts, but with the help of join, we can get the original data back.

Id	Worker
1	Henry Adamson
2	Connor Wilson

Id	Part name
1	Brake Disc
2	Spark Plug
3	Oil Pump

Id	Worker	Part name	Part count
1	1	1	1
2	1	2	2
3	1	1	3
4	1	2	1
5	2	1	2
6	2	2	1

Database normalization rules

It is desirable to carry out the process of data normalization at the stage of database design.

Normalization assumes the use of normal forms in relation to the structure of the available data. There are several normalization rules. Each of them is called “normal form” (NF). Each such form, except the first, assumes that the previous normal form has already been applied to the data. There are 7 normal forms in total.

- 1 NF,
- 2 NF,
- 3 NF,
- BCNF(Boyce-Codd normal form),
- 4 NF,
- 5 NF,
- 6 NF.

In practice, normalization is usually performed only up to the 3rd form.

First normal form

- Eliminate repeating groups in individual tables.
- Create a separate table for each set of related data.
- Identify each set of related data with a primary key.

After reduction to this form, no duplicates and compound data should remain in the database.

Second normal form

- Create separate tables for sets of values that apply to multiple records.
- Relate these tables with a foreign key.

Records should only depend on the table’s primary key (composite key, if necessary).

Third normal form

- Eliminate fields that do not depend on the key

If the contents of a field group can relate to more than one record in a table, consider putting those fields in a separate table.

Normalization benefits

With normalization, we can get the following benefits:
- Reduces the size of the database and saves space.
- The probability of errors in the data and their inconsistency is reduced.
- Search speeds up and work with the database becomes more convenient.