SQL DDL. Database Integrity Constraints.

Database Integrity Introduction.

Integrity - data validity at any given time. But this goal can be achieved only within certain limits: the DBMS cannot control the correctness of each individual value entered into the database, although each value can be checked for plausibility.

For example, it cannot be found that the value entered 5 (representing the day of the week) should actually be equal to 3. On the other hand, the value 9 will obviously be erroneous and the DBMS should reject it. However, for this, she should be informed that the numbers of the days of the week must belong to the set (1, 2, 3, 4, 5, 6, 7).

Maintaining integrity is protection against getting into the database **invalid data** (for example, to prevent user errors when entering data) and protection against **incorrect changes** or destruction (not to be confused with illegal changes and destruction, which are a security problem).

A database is in a consistent state if all integrity constraints are met for this state. Integrity constraint is a statement that can be true or false.

Examples of integrity constraints include the following business rules:

1. The age of an employee cannot be less than 18 and more than 65 years.
2. Each employee has a unique personnel number.
3. An employee is required to be listed in only one department.
4. The invoice amount must equal the sum of the prices of goods multiplied by the number of goods.
   This means that a change in the quantity of goods in the invoice cannot be performed in one operation, you need to:
      Step 1. Insert the goods;
      Step 2. Recalculate the value of the invoice amount.

If, after performing the first operation and before performing the second, the system crashes, then only the first operation will actually be performed and the database will remain in an unintegrity state. Using a transaction will help to avoid this problem - the partial changes will be canceled (ROLLBACK).

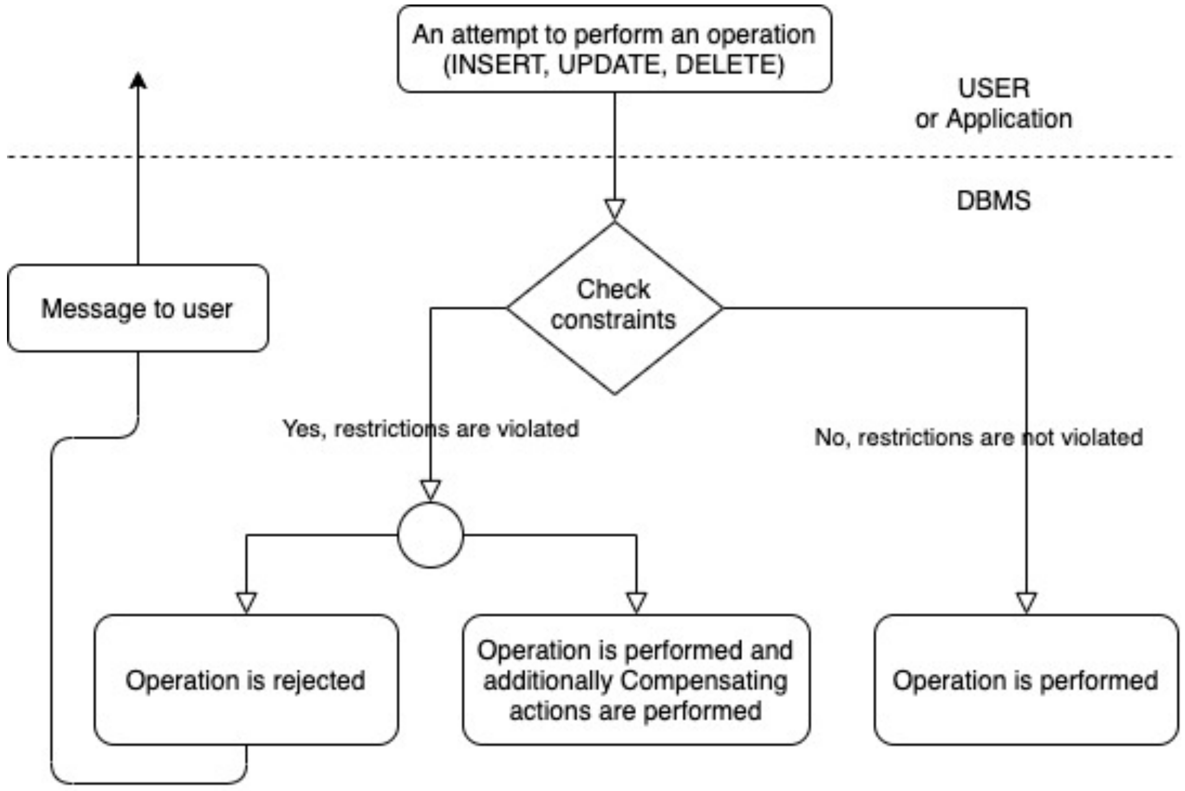
A transaction is a sequence of data manipulation operators that runs as a whole (**all or nothing!**) and transfers the database from one integrity state to another integrity state.

Example 1 are restrictions on the data **column** (type or domain). Example 2 presents a constraint that implements the integrity of an **entity**. Example 3 presents a constraint implementing **referential** integrity. Example 4 describes a restriction which is a rather arbitrary statement.

System response to attempted integrity violation:

1. **Failure** to perform an "illegal" operation (error message).

2. **Execute** operation and **compensating actions** (implemented by trigger mechanisms, built-in procedures and transactions).


An attempt to perform an operation (INSERT, UPDATE, DELETE)
USER or Application
DBMS
Message to user
Check constraints
Yes, restrictions are violated
No, restrictions are not violated
Operation is rejected
Operation is performed and additionally Compensating actions are performed
Operation is performed

For example, if the system knows that there must be integers in the range "Employee Age" in the range from 18 to 65, then the system rejects an attempt to enter the age value of 66. In this case, some message can be generated for the user.

In contrast, in Example 4, the system allows the insertion of a record for a new product (which leads to a violation of the integrity of the database), but **automatically** performs compensating actions by changing the value of the sum field.

Classification of Integrity Constraints in a relation database.

- By implementation methods.
  - **Declarative** support for integrity constraints - using the data definition language (DDL).
  - **Procedural** support for integrity constraints - through triggers and procedures, they are more functional, but less resource-saving.
- By check time.
  - **Immediately** check constraints.
  - **Deferred** check constraints.
- By scope area.
  - **Domain** Integrity (attribute-data type).

    Responsible for having valid values in the corresponding database field. This is ensured by the data type (domain), conditions on the value, the prohibition of empty values, triggers and stored procedures, keys.

    For example, a surname, as a rule, should consist of letters, and a zip code LV - of 4 digits.
  - **Entity** Integrity (table-primary key).

    This is ensured by limiting the uniqueness of the PK and the record. This is done by the DBMS using two restrictions:

    - when adding records to the table, the uniqueness of their primary keys is checked;
    - It is not allowed to change attribute values included in the primary key.
  - **Referential** Integrity (relationship-foreign key).

    It is provided by the primary and external key system, assigned to the DBMS, it will not allow adding an entry containing a foreign key with a nonexistent (undefined) value.

    For example, you can guarantee that we will not have orders placed for buyers who are not in the database.
  - **Transaction** Integrity (database).

    Transactions are used to ensure integrity in the event of database restrictions, rather than any separate operations.
  - **Application** Integrity (application).

    These are limitations in the client application, developers should take care that errors occurring during integrity violations are caught by the client application.

MySQL Integrity Constraints Implementation.

Modern DBMSs have a number of constraint tools to ensure integrity. For example MySQL support:

- **Domain** Integrity Constraints:
    - DATA TYPES
    - DEFAULT
    - UNIQUE
    - CHECK
    - NOT NULL
    - AUTO_INCREMENT
    - TRIGGER (This will be reviewed later.)
- **Entity** Integrity Constraints:
    - PRIMARY KEY
- **Referential** Integrity Constraints:
    - FOREIGN KEY Uniquely identifies a row/record in another table
- **Transaction** Integrity Constraints (This will be reviewed later.):
    - BEGIN – begin a Transaction.
    - COMMIT– commits a Transaction
    - ROLLBACK– rollbacks a transaction in case of any error occurs.

DATA TYPES - It has been reviewed before.

DEFAULT

The DEFAULT constraint is used to provide a default value for a column. The default value will be added to all new records IF no other value is specified. The DEFAULT constraint can also be used to insert system values, by using functions like GETDATE():

```
CREATE TABLE Orders (
  ID int NOT NULL,
  FirstName varchar(255),
  City varchar(255) DEFAULT 'Riga',
  OrderDate date DEFAULT GETDATE()
);
```

UNIQUE

Unique - ensure the uniqueness of values in a column or a set of columns. Sometimes, you want to make sure that the values in a column or a set of columns are **not duplicate**.

For example, duplicate emails in the employees table are not acceptable. Since the email column is not the part of the primary key, the only way to prevent duplicate values in the email column is to use a UNIQUE constraint.

You can have **multiple** UNIQUE constraints in a table, all UNIQUE constraints must have a different set of columns. UNIQUE constraint **allows** NULL values in MySQL DBMS.

Creating UNIQUE constraints using the column constraint syntax

```
CREATE TABLE users (
  user_id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(255) NOT NULL UNIQUE,
  password VARCHAR(255) NOT NULL
);
```

If you insert or update the value that is the same as the one which already exists, the RDBMS will reject the change and return an error.

Equivalent Creating UNIQUE constraints using the table constraint syntax

```
CREATE TABLE users (
  user_id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(255) NOT NULL,
  password VARCHAR(255) NOT NULL,
  CONSTRAINT uc_username UNIQUE (username)
);
```

Adding and removing UNIQUE constraints syntax

```
ALTER TABLE users ADD new_column data_type UNIQUE;
ALTER TABLE table_name DROP CONSTRAINT unique_constraint_name;
```

CHECK

A CHECK constraint validate data before it is stored in one or more columns based on a Boolean expression with AND, OR, >, <, >=, <=, =, != operators null, numeric and string functions using.

```
CHECK(Boolean_expression)
```

You can define a CHECK constraint on a single column or the whole table. If you define the CHECK constraint on a single column, the CHECK constraint checks value for this column only. However, if you define a CHECK constraint on a table, it limits value in a column based on values in other columns of the same row.

It is important to note that the CHECK constraint is satisfied when the Boolean expression returns true or the NULL value. Most Boolean expressions evaluate to NULL if one of the operands is NULL, they will not prevent null values in the constrained column. To make sure that the column does not contain the NULL values, you use the NOT NULL constraint.

MySQL CHECK constraint – column constraint example

```
CREATE TABLE products (
  product_id INT PRIMARY KEY,
  product_name VARCHAR(255) NOT NULL,
  selling_price NUMERIC(10,2) CHECK (selling_price > 0),
  cost NUMERIC (10, 2) CHECK (cost > 0)
);
```

MySQL CHECK constraint – table constraint example

```
CREATE TABLE products (
  product_id INT PRIMARY KEY,
  product_name VARCHAR (255) NOT NULL,
  selling_price NUMERIC (10, 2),
  cost NUMERIC (10, 2) CHECK (cost > 0),
  CHECK (selling_price > cost)
);
```

NOT NULL

The NOT NULL constraint is a column constraint that defines the rule which constrains a column to have non-NULL values only. It means that when we use the INSERT statement to insert a new row into the table, we have to specify the values for the NOT NULL columns.

The following statement illustrates the NOT NULL constraint syntax.

```
CREATE TABLE table_name(
  ...
  column_name data_type NOT NULL,
  ...
);
```

Logically, an NOT NULL constraint is equivalent to a CHECK constraint, therefore, the above statement is equivalent to the following statement.

```
CREATE TABLE table_name (
  ...
  column_name data_type,
  ...
  CHECK (column_name IS NOT NULL)
);
```

For change the constraint of a column that accepts a NULL value to not accept a NULL value. To carry the change, we use these two steps:

First, update all current NULL values to non-NULL values using the UPDATE statement.

```
UPDATE table_name
SET column_name = 0
WHERE
  column_name IS NULL;
```

Second, add the NOT NULL constraint to the column using the ALTER TABLE statement

```
ALTER TABLE table_name
MODIFY column_name data_type NOT NULL;
```

AUTO_INCREMENT

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

```
CREATE TABLE animals (
id MEDIUMINT NOT NULL AUTO_INCREMENT,
name CHAR(30) NOT NULL,
PRIMARY KEY (id)
);
INSERT INTO animals (name) VALUES ("dog"),("cat"),("penguin"),("mouse")
SELECT * FROM animals;
```

Result:

```
+----+---------+
| id | name    |
+----+---------+
|  1 | dog     |
|  2 | cat     |
|  3 | penguin |
|  4 | mouse   |
+----+---------+
```

By default, the starting value for AUTO_INCREMENT is 1, and it will increment by 1 for each new record. To let the AUTO_INCREMENT sequence start with another value, use the following SQL statement:

```
ALTER TABLE Persons AUTO_INCREMENT=100;
```

Remark. The unique values of a 4-byte counter at a generation rate of one value per second will last for more than 120 years!

Calculate. For 1 year, consumed (365d*24h*60m*60s*1 value per second) = 2ˆ25 counter values. The possible number of values is 2ˆ(4*8)= 2ˆ32 ➔ 2ˆ32/2ˆ25 = 2ˆ7 = 128 years.

PRIMARY KEY

A table consists of columns and rows. Typically, a table has a column or set of columns whose value uniquely identify each row in the table. This column or the set of columns is called the **primary key**.

The primary key that consists of two or more columns is also known as the **composite primary key**.

Each table has **one and only one** primary key. The primary key a combination of a NOT NULL and UNIQUE.

In case the primary key consists of two or more columns, the values may be duplicate in one column, but the combination of values from all columns in the primary key must be unique.

Generally, you define the primary key when creating the table. If the primary key consists of one column, you can use the PRIMARY KEY constraint as column or table constraint.

```
CREATE TABLE projects (
  project_id INT PRIMARY KEY,
  project_name VARCHAR(255),
  start_date DATE NOT NULL
);
```

In case the primary key consists of two or more columns, you must use the PRIMARY KEY constraint as the table constraint.

```
CREATE TABLE project_assignments (
  project_id INT,
  employee_id INT,
  join_date DATE NOT NULL,
  CONSTRAINT pk_assgn PRIMARY KEY (project_id , employee_id)
);
```

Adding and Removing the primary key

```
ALTER TABLE project_milestones ADD PRIMARY KEY (milestone_id);
ALTER TABLE table_name DROP PRIMARY KEY
```

FOREIGN KEY

A foreign key is a column or a group of columns that enforces a link between the data in two tables. In a foreign key reference, the primary key column (or columns) of the first table is referenced by the column (or columns) of the second table. The column (or columns) of the second table becomes the foreign key.

You use the FOREIGN KEY constraint to create a foreign key when you create or alter table.

Example

```
CREATE TABLE projects (
  project_id INT AUTO_INCREMENT PRIMARY KEY,
  project_name VARCHAR(255),
  start_date DATE NOT NULL
);
CREATE TABLE project_stag(
  stag_id INT AUTO_INCREMENT PRIMARY KEY,
  project_id INT,
  stag_name VARCHAR(100)
);
```

Each project may have 0 or more stags while one stag must belong to 1 and only 1 project. In other words, a stag cannot exist without a project.

Unfortunately, a row might be added to the project_stag table that does not correspond to any row in the projects table. Or user may delete a row in the projects table, leaving orphaned rows in the project_stag table. This causes the application **not to work properly**.

The solution is to add an SQL FOREIGN KEY constraint to the project_stag table to enforce the relationship between two tables.

```
CREATE TABLE project_milestones (
  stag_id INT AUTO_INCREMENT PRIMARY KEY,
  project_id INT,
  stag_name VARCHAR(100),
  FOREIGN KEY (project_id)
    REFERENCES projects (project_id)
);
```