DATA DEFINITION LANGUAGE (DDL) - Syntax

CREATE TRIGGER Syntax

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    TRIGGER trigger_name
    trigger_time trigger_event
    ON tbl_name FOR EACH ROW
    [trigger_order]
    trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

This statement creates a new trigger. A trigger is a named database object that is associated with a table, and that activates when a particular event occurs for the table. The trigger becomes associated with the table named `tbl_name`, which must refer to a permanent table. You cannot associate a trigger with a `TEMPORARY` table or a view.

Trigger names exist in the schema namespace, meaning that all triggers must have unique names within a schema. Triggers in different schemas can have the same name.

As of MySQL 5.7.2, it is possible to define multiple triggers for a given table that have the same trigger event and action time. For example, you can have two `BEFORE UPDATE` triggers for a table. By default, triggers that have the same trigger event and action time activate in the order they were created. To affect trigger order, specify a `trigger_order` clause that indicates `FOLLOWS` or `PRECEDES` and the name of an existing trigger that also has the same trigger event and action time. With `FOLLOWS`, the new trigger activates after the existing trigger. With `PRECEDES`, the new trigger activates before the existing trigger.

Within the trigger body, you can refer to columns in the subject table (the table associated with the trigger) by using the aliases `OLD` and `NEW`. `OLD.col_name` refers to a column of an existing row before it is updated or deleted. `NEW.`*col_name* refers to the column of a new row to be inserted or an existing row after it is updated.

Triggers cannot use `NEW.`*col_name* or use `OLD.`*col_name* to refer to generated columns

Example 1:

```
mysql> CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));
Query OK, 0 rows affected (0.03 sec)

mysql> CREATE TRIGGER ins_sum BEFORE INSERT ON account
    -> FOR EACH ROW SET @sum = @sum + NEW.amount;
```

```
Query OK, 0 rows affected (0.06 sec)

mysql> SET @sum = 0;
mysql> INSERT INTO account VALUES(137,14.98),(141,1937.50),(97,-100.00);
mysql> SELECT @sum AS 'Total amount inserted';
+-----------------------+
| Total amount inserted |
+-----------------------+
| 1852.48               |
+-----------------------+
```

```
mysql> CREATE TRIGGER ins_transaction BEFORE INSERT ON account
    -> FOR EACH ROW PRECEDES ins_sum
    -> SET
    -> @deposits = @deposits + IF(NEW.amount>0,NEW.amount,0),
    -> @withdrawals = @withdrawals + IF(NEW.amount<0,-NEW.amount,0);
Query OK, 0 rows affected (0.02 sec)

mysql> delimiter //
mysql> CREATE TRIGGER upd_check BEFORE UPDATE ON account
    -> FOR EACH ROW
    -> BEGIN
    ->     IF NEW.amount < 0 THEN
    ->         SET NEW.amount = 0;
    ->     ELSEIF NEW.amount > 100 THEN
    ->         SET NEW.amount = 100;
    ->     END IF;
    -> END;//
mysql> delimiter ;
```

Example 2:

```
CREATE TABLE test1(a1 INT);
CREATE TABLE test2(a2 INT);
CREATE TABLE test3(a3 INT NOT NULL AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE test4(
  a4 INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  b4 INT DEFAULT 0
);

delimiter |

CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  END;
|

delimiter ;

INSERT INTO test3 (a3) VALUES
  (NULL), (NULL), (NULL), (NULL), (NULL),
  (NULL), (NULL), (NULL), (NULL), (NULL);
```

```
INSERT INTO test4 (a4) VALUES
  (0), (0), (0), (0), (0), (0), (0), (0), (0), (0);
```

Suppose that you insert the following values into table `test1` as shown here:

```
mysql> INSERT INTO test1 VALUES
    -> (1), (3), (1), (7), (1), (8), (4), (4);
Query OK, 8 rows affected (0.01 sec)
Records: 8  Duplicates: 0  Warnings: 0
```

As a result, the four tables contain the following data:

```
mysql> SELECT * FROM test1;
+------+
| a1   |
+------+
|    1 |
|    3 |
|    1 |
|    7 |
|    1 |
|    8 |
|    4 |
|    4 |
+------+
8 rows in set (0.00 sec)

mysql> SELECT * FROM test2;
+------+
| a2   |
+------+
|    1 |
|    3 |
|    1 |
|    7 |
|    1 |
|    8 |
|    4 |
|    4 |
+------+
8 rows in set (0.00 sec)

mysql> SELECT * FROM test3;
+----+
| a3 |
+----+
|  2 |
|  5 |
|  6 |
|  9 |
| 10 |
+----+
5 rows in set (0.00 sec)

mysql> SELECT * FROM test4;
+----+------+
| a4 | b4   |
+----+------+
|  1 |     3 |
|  2 |     0 |
|  3 |     1 |
|  4 |     2 |
|  5 |     0 |
|  6 |     0 |
|  7 |     1 |
|  8 |     1 |
|  9 |     0 |
| 10 |     0 |
+----+------+
10 rows in set (0.00 sec)
```

CREATE VIEW Syntax

```
CREATE
    [OR REPLACE]
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    [DEFINER = { user | CURRENT_USER }]
    [SQL SECURITY { DEFINER | INVOKER }]
    VIEW view_name [(column_list)]
    AS select_statement
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

The `CREATE VIEW` statement creates a new view, or replaces an existing view if the `OR REPLACE` clause is given. If the view does not exist, `CREATE OR REPLACE VIEW` is the same as `CREATE VIEW`. If the view does exist, `CREATE OR REPLACE VIEW` is the same as `ALTER VIEW`.

The `select_statement` is a `SELECT` statement that provides the definition of the view. (Selecting from the view selects, in effect, using the `SELECT` statement.) The `select_statement` can select from base tables or other views.

The `WITH CHECK OPTION` clause can be given for an updatable view to prevent inserts or updates to rows except those for which the `WHERE` clause in the *`select_statement`* is true.

In a `WITH CHECK OPTION` clause for an updatable view, the `LOCAL` and `CASCADED` keywords determine the scope of check testing when the view is defined in terms of another view. The `LOCAL` keyword restricts the `CHECK OPTION` only to the view being defined. `CASCADED` causes the checks for underlying views to be evaluated as well. When neither keyword is given, the default is `CASCADED`.

The view definition is “frozen” at creation time and is not affected by subsequent changes to the definitions of the underlying tables. For example, if a view is defined as `SELECT *` on a table, new columns added to the table later do not become part of the view, and columns dropped from the table will result in an error when selecting from the view.

A view belongs to a database. By default, a new view is created in the default database. To create the view explicitly in a given database, use `db_name.view_name` syntax to qualify the view name with the database name:

```
CREATE VIEW test.v AS SELECT * FROM t;
```

A view can be created from many kinds of `SELECT` statements. It can refer to base tables or other views. It can use joins, `UNION`, and subqueries. The `SELECT` need not even refer to any tables:

```
CREATE VIEW v_today (today) AS SELECT CURRENT_DATE;
```

The following example defines a view that selects two columns from another table as well as an expression calculated from those columns:

```
mysql> CREATE TABLE t (qty INT, price INT);
mysql> INSERT INTO t VALUES(3, 50);
mysql> CREATE VIEW v AS SELECT qty, price, qty*price AS value FROM t;
mysql> SELECT * FROM v;
+------+-------+-------+
| qty  | price | value |
+------+-------+-------+
|    3 |    50 |   150 |
+------+-------+-------+
```

A view definition is subject to the following restrictions:

- Before MySQL 5.7.7, the `SELECT` statement cannot contain a subquery in the `FROM` clause.
- The `SELECT` statement cannot refer to system variables or user-defined variables.
- Within a stored program, the `SELECT` statement cannot refer to program parameters or local variables.
- The `SELECT` statement cannot refer to prepared statement parameters.
- Any table or view referred to in the definition must exist. If, after the view has been created, a table or view that the definition refers to is dropped, use of the view results in an error. To check a view definition for problems of this kind, use the `CHECK TABLE` statement.
- The definition cannot refer to a `TEMPORARY` table, and you cannot create a `TEMPORARY` view.
- You cannot associate a trigger with a view.
- Aliases for column names in the `SELECT` statement are checked against the maximum column length of 64 characters (not the maximum alias length of 256 characters).

The results obtained from a view may be affected if you change the query processing environment by changing system variables:

```
mysql> CREATE VIEW v (mycol) AS SELECT 'abc';
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> SET sql_mode = '';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT "mycol" FROM v;
+-------+
| mycol |
+-------+
| mycol |
+-------+
1 row in set (0.01 sec)

mysql> SET sql_mode = 'ANSI_QUOTES';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT "mycol" FROM v;
+-------+
| mycol |
+-------+
| abc   |
+-------+
1 row in set (0.00 sec)
```

Some views are updatable. That is, you can use them in statements such as `UPDATE`, `DELETE`, or `INSERT` to update the contents of the underlying table. For a view to be updatable, there must be a one-to-one relationship between the rows in the view and the rows in the underlying table. There are also certain other constructs that make a view nonupdatable.

A generated column in a view is considered updatable because it is possible to assign to it. However, if such a column is updated explicitly, the only permitted value is `DEFAULT`.

The `WITH CHECK OPTION` clause can be given for an updatable view to prevent inserts or updates to rows except those for which the `WHERE` clause in the `select_statement` is true.

In a `WITH CHECK OPTION` clause for an updatable view, the `LOCAL` and `CASCADED` keywords determine the scope of check testing when the view is defined in terms of another view. The `LOCAL` keyword restricts the `CHECK OPTION` only to the view being defined. `CASCADED` causes the checks for underlying views to be evaluated as well. When neither keyword is given, the default is `CASCADED`.

ALTER DATABASE Syntax

```
ALTER {DATABASE | SCHEMA} [db_name]
    alter_specification ...
ALTER {DATABASE | SCHEMA} db_name
    UPGRADE DATA DIRECTORY NAME

alter_specification:
    [DEFAULT] CHARACTER SET [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name
```

`ALTER DATABASE` enables you to change the overall characteristics of a database. These characteristics are stored in the `db.opt` file in the database directory. To use `ALTER DATABASE`, you need the `ALTER` privilege on the database. `ALTER SCHEMA` is a synonym for `ALTER DATABASE`.

The database name can be omitted from the first syntax, in which case the statement applies to the default database.

You can see what character sets and collations are available using, respectively, the `SHOW CHARACTER SET` and `SHOW COLLATION` statements.

ALTER TABLE Syntax

```
ALTER [IGNORE] TABLE tbl_name
    [alter_specification [, alter_specification] ...]
    [partition_options]

alter_specification:
    table_options
  | ADD [COLUMN] col_name column_definition
        [FIRST | AFTER col_name ]
  | ADD [COLUMN] (col_name column_definition,...)
  | ADD {INDEX|KEY} [index_name]
        [index_type] (index_col_name,...) [index_option] ...
  | ADD [CONSTRAINT [symbol]] PRIMARY KEY
        [index_type] (index_col_name,...) [index_option] ...
  | ADD [CONSTRAINT [symbol]]
        UNIQUE [INDEX|KEY] [index_name]
        [index_type] (index_col_name,...) [index_option] ...
  | ADD FULLTEXT [INDEX|KEY] [index_name]
        (index_col_name,...) [index_option] ...
  | ADD SPATIAL [INDEX|KEY] [index_name]
        (index_col_name,...) [index_option] ...
  | ADD [CONSTRAINT [symbol]]
        FOREIGN KEY [index_name] (index_col_name,...)
        reference_definition
  | ALGORITHM [=] {DEFAULT|INPLACE|COPY}
  | ALTER [COLUMN] col_name {SET DEFAULT literal | DROP DEFAULT}
  | CHANGE [COLUMN] old_col_name new_col_name column_definition
        [FIRST|AFTER col_name]
  | LOCK [=] {DEFAULT|NONE|SHARED|EXCLUSIVE}
  | MODIFY [COLUMN] col_name column_definition
        [FIRST | AFTER col_name]
  | DROP [COLUMN] col_name
  | DROP PRIMARY KEY
  | DROP {INDEX|KEY} index_name
  | DROP FOREIGN KEY fk_symbol
  | DISABLE KEYS
  | ENABLE KEYS
  | RENAME [TO|AS] new_tbl_name
  | RENAME {INDEX|KEY} old_index_name TO new_index_name
  | ORDER BY col_name [, col_name] ...
  | CONVERT TO CHARACTER SET charset_name [COLLATE collation_name]
  | [DEFAULT] CHARACTER SET [=] charset_name [COLLATE [=] collation_name]
  | DISCARD TABLESPACE
  | IMPORT TABLESPACE
  | FORCE
  | {WITHOUT|WITH} VALIDATION
  | ADD PARTITION (partition_definition)
  | DROP PARTITION partition_names
  | DISCARD PARTITION {partition_names | ALL} TABLESPACE
  | IMPORT PARTITION {partition_names | ALL} TABLESPACE
  | TRUNCATE PARTITION {partition_names | ALL}
  | COALESCE PARTITION number
  | REORGANIZE PARTITION partition_names INTO (partition_definitions)
  | EXCHANGE PARTITION partition_name WITH TABLE tbl_name [{WITH|WITHOUT}
VALIDATION]
  | ANALYZE PARTITION {partition_names | ALL}
  | CHECK PARTITION {partition_names | ALL}
  | OPTIMIZE PARTITION {partition_names | ALL}
  | REBUILD PARTITION {partition_names | ALL}
  | REPAIR PARTITION {partition_names | ALL}
  | REMOVE PARTITIONING
  | UPGRADE PARTITIONING

index_col_name:
    col_name [(length)] [ASC | DESC]

index_type:
    USING {BTREE | HASH}

index_option:
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'

table_options:
    table_option [[,] table_option] ...  (see CREATE TABLE options)

partition_options:
    (see CREATE TABLE options)
```

`ALTER TABLE` changes the structure of a table. For example, you can add or delete columns, create or destroy indexes, change the type of existing columns, or rename columns or the table itself. You can also change characteristics such as the storage engine used for the table or the table comment.

Following the table name, specify the alterations to be made. If none are given, `ALTER TABLE` does nothing.

The syntax for many of the permissible alterations is similar to clauses of the `CREATE TABLE` statement.

Use of table options with `ALTER TABLE` provides a convenient way of altering single table characteristics. For example, if `t1` is currently not an `InnoDB` table, this statement changes its storage engine to `InnoDB`:

```
ALTER TABLE t1 ENGINE = InnoDB;
```

To reset the current auto-increment value:

```
ALTER TABLE t1 AUTO_INCREMENT = 13;
```

To change the default table character set:

```
ALTER TABLE t1 CHARACTER SET = utf8;
```

To add (or change) a table comment:

```
ALTER TABLE t1 COMMENT = 'New table comment';
```

You can issue multiple `ADD`, `ALTER`, `DROP`, and `CHANGE` clauses in a single `ALTER TABLE` statement, separated by commas. This is a MySQL extension to standard SQL, which permits only one of each clause per `ALTER TABLE` statement. For example, to drop multiple columns in a single statement, do this:

```
ALTER TABLE t2 DROP COLUMN c, DROP COLUMN d;
```

You can rename a column using a `CHANGE` *`old_col_name new_col_name column_definition`* clause. To do so, specify the old and new column names and the definition that the column currently has. For example, to rename an `INTEGER` column from `a` to `b`, you can do this:

```
ALTER TABLE t1 CHANGE a b INTEGER;
```

To change a column's type but not the name, `CHANGE` syntax still requires an old and new column name, even if they are the same. For example:

```
ALTER TABLE t1 CHANGE b b BIGINT NOT NULL;
```

You can also use `MODIFY` to change a column's type without renaming it:

```
ALTER TABLE t1 MODIFY b BIGINT NOT NULL;
```

`MODIFY` is an extension to `ALTER TABLE` for Oracle compatibility.

When you use `CHANGE` or `MODIFY`, *`column_definition`* must include the data type and all attributes that should apply to the new column, other than index attributes such as `PRIMARY KEY` or `UNIQUE`. Attributes present in the original definition but not specified for the new definition are not carried forward. Suppose that a column `col1` is defined as `INT UNSIGNED DEFAULT 1 COMMENT 'my column'` and you modify the column as follows:

```
ALTER TABLE t1 MODIFY col1 BIGINT;
```

The resulting column will be defined as `BIGINT`, but will not include the attributes `UNSIGNED DEFAULT 1 COMMENT 'my column'`. To retain them, the statement should be:

```
ALTER TABLE t1 MODIFY col1 BIGINT UNSIGNED DEFAULT 1 COMMENT 'my column';
```

`DROP PRIMARY KEY` drops the primary key.

`RENAME INDEX` *`old_index_name`* `TO` *`new_index_name`* renames an index.

The `FOREIGN KEY` and `REFERENCES` clauses are supported by the `InnoDB` and `NDB` storage engines, which implement `ADD [CONSTRAINT [`*`symbol`*`]] FOREIGN KEY [`*`index_name`*`] (...) REFERENCES ... (...)`.

For `ALTER TABLE`, unlike `CREATE TABLE`, `ADD FOREIGN KEY` ignores *`index_name`* if given and uses an automatically generated foreign key name. As a workaround, include the `CONSTRAINT` clause to specify the foreign key name:

```
ADD CONSTRAINT name FOREIGN KEY (....) ...
```

`ALTER TABLE` to drop foreign keys:

```
ALTER TABLE tbl_name DROP FOREIGN KEY fk_symbol;
```

To change the table default character set and all character columns (`CHAR`, `VARCHAR`, `TEXT`) to a new character set, use a statement like this:

Use `MODIFY` to change individual columns. For example:

```
ALTER TABLE t MODIFY latin1_text_col TEXT CHARACTER SET utf8;
ALTER TABLE t MODIFY latin1_varchar_col VARCHAR(M) CHARACTER SET utf8;
```

To change only the *default* character set for a table, use this statement:

```
ALTER TABLE tbl_name DEFAULT CHARACTER SET charset_name;
```

ALTER TABLE and Generated Columns

`ALTER TABLE` operations permitted for generated columns are `ADD`, `MODIFY`, and `CHANGE`.

ALTER TABLE Examples

Begin with a table `t1` that is created as shown here:

```
CREATE TABLE t1 (a INTEGER,b CHAR(10));
```

To rename the table from `t1` to `t2`:

```
ALTER TABLE t1 RENAME t2;
```

To change column `a` from `INTEGER` to `TINYINT NOT NULL` (leaving the name the same), and to change column `b` from `CHAR(10)` to `CHAR(20)` as well as renaming it from `b` to `c`:

```
ALTER TABLE t2 MODIFY a TINYINT NOT NULL, CHANGE b c CHAR(20);
```

To add a new `TIMESTAMP` column named `d`:

```
ALTER TABLE t2 ADD d TIMESTAMP;
```

To add an index on column `d` and a `UNIQUE` index on column `a`:

```
ALTER TABLE t2 ADD INDEX (d), ADD UNIQUE (a);
```

To remove column `c`:

```
ALTER TABLE t2 DROP COLUMN c;
```

To add a new `AUTO_INCREMENT` integer column named `c`:

```
ALTER TABLE t2 ADD c INT UNSIGNED NOT NULL AUTO_INCREMENT,
  ADD PRIMARY KEY (c);
```

We indexed `c` (as a `PRIMARY KEY`) because `AUTO_INCREMENT` columns must be indexed, and we declare `c` as `NOT NULL` because primary key columns cannot be `NULL`.

ALTER VIEW Syntax

```
ALTER
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    [DEFINER = { user | CURRENT_USER }]
    [SQL SECURITY { DEFINER | INVOKER }]
    VIEW view_name [(column_list)]
    AS select_statement
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

This statement changes the definition of a view, which must exist.

DROP DATABASE Syntax

```
DROP {DATABASE | SCHEMA} [IF EXISTS] db_name
```

`DROP DATABASE` drops all tables in the database and deletes the database.

DROP TABLE Syntax

```
DROP [TEMPORARY] TABLE [IF EXISTS]
    tbl_name [, tbl_name] ...
    [RESTRICT | CASCADE]
```

`DROP TABLE` removes one or more tables.

DROP INDEX Syntax

```
DROP INDEX index_name ON tbl_name
    [algorithm_option | lock_option] ...

algorithm_option:
    ALGORITHM [=] {DEFAULT|INPLACE|COPY}

lock_option:
    LOCK [=] {DEFAULT|NONE|SHARED|EXCLUSIVE}
```

`DROP INDEX` drops the index named *`index_name`* from the table *`tbl_name`*.

To drop a primary key, the index name is always `PRIMARY`, which must be specified as a quoted identifier because `PRIMARY` is a reserved word:

```
DROP INDEX `PRIMARY` ON t;
```

DROP TRIGGER Syntax

```
DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name
```

This statement drops a trigger.

DROP VIEW Syntax

```
DROP VIEW [IF EXISTS]
    view_name [, view_name] ...
    [RESTRICT | CASCADE]
```

`DROP VIEW` removes one or more views.

RENAME TABLE Syntax

```
RENAME TABLE tbl_name TO new_tbl_name
    [, tbl_name2 TO new_tbl_name2] ...
```

This statement renames one or more tables. The rename operation is done atomically, which means that no other session can access any of the tables while the rename is running.

For example, a table named `old_table` can be renamed to `new_table` as shown here:

```
RENAME TABLE old_table TO new_table;
```

This statement is equivalent to the following `ALTER TABLE` statement:

```
ALTER TABLE old_table RENAME new_table;
```

If the statement renames more than one table, renaming operations are done from left to right. If you want to swap two table names, you can do so like this (assuming that `tmp_table` does not already exist):

```
RENAME TABLE old_table TO tmp_table,
             new_table TO old_table,
             tmp_table TO new_table;
```

As long as two databases are on the same file system, you can use `RENAME TABLE` to move a table from one database to another:

```
RENAME TABLE current_db.tbl_name TO other_db.tbl_name;
```

If there are any triggers associated with a table which is moved to a different database using `RENAME TABLE`, then the statement fails with the error Trigger in wrong schema.

Foreign keys that point to the renamed table are not automatically updated. In such cases, you must drop and re-create the foreign keys in order for them to function properly.

`RENAME TABLE` also works for views, as long as you do not try to rename a view into a different database.

TRUNCATE TABLE Syntax

```
TRUNCATE [TABLE] tbl_name
```

`TRUNCATE TABLE` empties a table completely.

ZADACI

Primer trigera nad više atributa u relaciji:

1.Zadatak
Data je baza podataka Test sa sledećim relacijama:
student_mast(STUDENT_ID, NAME, ST_CLASS)
student_marks(STUDENT_ID, NAME,SUB1, SUB2, SUB3, SUB4, SUB5, TOTAL, PER_MARKS, GRADE)
student_log(USER_ID, DESCRIPTION)

Pretpostavimo da tabela student_marks sadrži podatke o studentima koji su se prijavili za polaganje testa. Kako test još nije završen, pretpostavimo da se u tabeli za svakog studenta nalaze samo vrednosti pripadajućih atributa STUDENT_ID i NAME dok su vrednosti svih ostalih atributa postavljene na podrazumevane vrednosti (0 ili NULL). Završen je test i studenti su dobili ocene za 5 različitih predmeta. Potrebno je izmeniti podatke o studentima u tabeli student_marks, pri čemu je za svakog studenta potrebno uneti ocene koje je dobio na testu za svaki od 5 predmeta. U skladu sa dobijenim ocenama potrebno je automatski izmeniti ukupnu ocenu za sve predmete (TOTAL), prosečnu ocenu (PER_MARKS) i ukupan
uspeh na testu (GRADE). Vrednosti ovih atributa se izračunavaju na sledeći način:
Ukupna ocena: TOTAL = SUB1 + SUB2 + SUB3 + SUB4 + SUB5
Prosečna ocena: PER_MARKS= (TOTAL)/5
Uspeh: GRADE=
- If PER_MARKS>=90 -> ‘EXCELLENT’
- If PER_MARKS>=75 AND PER_MARKS<90 -> ‘VERY GOOD’
- If PER_MARKS>=60 AND PER_MARKS<75 -> ‘GOOD’
- If PER_MARKS>=40 AND PER_MARKS<60 -> ‘AVERAGE’
- If PER_MARKS<40-> ‘NOT PROMOTED’

Rešenje:

Zadatak ćemo rešiti korišćenjem trigera nad tabelom student_marks:

```
USE Test;
DELIMITER $$
CREATE TRIGGER student_marks_BUPD BEFORE UPDATE ON student_marks
FOR EACH ROW
BEGIN
SET NEW.TOTAL = NEW.SUB1 + NEW.SUB2 + NEW.SUB3 + NEW.SUB4 + NEW.SUB5;
SET NEW.PER_MARKS = NEW.TOTAL/5;
IF NEW.PER_MARKS >=90 THEN SET NEW.GRADE = 'EXCELLENT';
ELSEIF NEW.PER_MARKS>=75 AND NEW.PER_MARKS<90 THEN SET NEW.GRADE = 'VERY
GOOD';
ELSEIF NEW.PER_MARKS>=60 AND NEW.PER_MARKS<75 THEN SET NEW.GRADE =
'GOOD';
ELSEIF NEW.PER_MARKS>=40 AND NEW.PER_MARKS<60 THEN SET NEW.GRADE =
'AVERAGE';
ELSE SET NEW.GRADE = 'NOT PROMOTED';
END IF;
END;
$$
```

Primer trigera nad više relacija:

2.Zadatak
Data je baza Test iz prethodnog zadatka. Potrebno je za sve studente iz tabele student_mast promeniti razred u sledeći. Kreirati triger koji proverava da uneti razred nije manji od starog razreda. Kreirati triger koji će obezbediti da se posle svake izmene pojedinačne n-torke u tabeli student_mast doda nova n-torka u tabelu student_log sa informacijom o korisniku (user_id) koji je izvršio izmenu i opisom u vezi izvršene izmene. Kreirati i triger koji nakon svakog brisanja n-torke iz tabele student_mast dodaje novu vrstu u tabelu student_log sa informacijom o korisniku koji vr ši brisanje i kratkim opisom izvršenog brisanja.

Rešenje:

```
Use Test;
DELIMITER $$
CREATE TRIGGER student_mast_BUPD BEFORE UPDATE ON student_mast
FOR EACH ROW
BEGIN
DECLARE msg VARCHAR(255);
IF (NEW.ST_CLASS < OLD.ST_CLASS)
THEN
SET msg='Greska: novi razred je manji od starog!';
SIGNAL sqlstate '45000' SET message_text= msg;
END IF ;
END $$
CREATE TRIGGER student_mast_AUPD AFTER UPDATE ON student_mast
FOR EACH ROW
BEGIN
INSERT into student_log VALUES (user(), CONCAT('Izmena podataka o student ' , OLD.NAME, '
Prethodni razred : ',OLD.ST_CLASS,' Novi razred : ', NEW.st_class));
END$$
CREATE TRIGGER student_mast_ADEL AFTER DELETE ON student_mast
FOR EACH ROW
BEGIN INSERT into student_log VALUES (user(), CONCAT('Izbrisani podaci o studentu:
',OLD.NAME,' Razred :',OLD.ST_CLASS, '-> Datum brisanja : ', NOW()));
END;
$$
DELIMITER ;
```

3.Zadatak
Data je relacija emp_details kao deo baze ljudskih resursa:
emp_details(EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, JOB_ID, SALARY, COMMISSION_PCT)
Kreirati triger koji omogućava da se nakon svakog unošenja podataka o zaposlenima u tabelu emp_details izvrši provera njihove zarade i ako je zarada veća od 20000 da se za tog zaposlenog postavi provizija od urađenog posla (COMMISSION_PCT) na 0.1 a u suprotnom na 0.5.

Rešenje:

```
DELIMITER $$
CREATE TRIGGER emp_details_AINS AFTER INSERT ON emp_details
FOR EACH ROW
BEGIN
IF NEW.SALARY>
=20000 THEN
UPDATE emp_details SET NEW.COMMISSION_PCT=0.1;
ELSEIF NEW.SALARY<20000 THEN
UPDATE emp_details SET NEW.COMMISSION_PCT=0.5;
END IF;
END$$
DELIMITER;
```

(prema knjizi prof. Gordane Pavlović-Lažetić http://poincare.matf.bg.ac.rs/~gordana//FINALE.pdf)
Data je aproksimacija izdavačke relacione baze sa sledećom shemom:

P (P_SIF, IME, BR_NASLOVA, DRZAVA)
I (I_SIF, NAZIV, STATUS, DRZAVA)
K (K_SIF, NASLOV, OBLAST)
KP (K_SIF, P_SIF, R_BROJ)
KI (K_SIF, I_SIF, IZDANJE, GODINA, TIRAZ)

Prve tri relacije predstavljaju tipove entiteta PISAC, IZDAVAC i KNJIGA redom. Relacija KP predstavlja apstraktni tip entiteta AUTOR, tj. odnos između tipova entiteta KNJIGA i PISAC, dok relacija KI predstavlja apstraktni tip entiteta IZDAVASTVO, tj. odnos između tipova entiteta KNJIGA i IZDAVAC.

4.Zadatak
Prilikom registracije novog autorskog dela, odnosno prilikom unošenja n-torke u reklaciju KP, uvećati vrednost atributa BR_NASLOVA za jedan u okviru relacije P za pisca koji je autor unetog autorskog dela.

Rešenje:

```
DELIMITER $$
CREATE TRIGGER KP_UNOS AFTER INSERT ON KP
FOR EACH ROW
BEGIN
DECLARE x INTEGER;
SET x = NEW.P_SIF;
UPDATE P
SET BR_NASLOVA=BR_NASLOVA+1 WHERE P_SIF=x;
END
$$
DELIMITER ;
```

Na sličan način mogu se definisati trigeri pri brisanju, odnosno izmeni podataka o autorskom delu:

```
DELIMITER $$
CREATE TRIGGER KP_BRIS AFTER DELETE ON KP
FOR EACH ROW
BEGIN
DECLARE x INTEGER;
SET x = NEW.P_SIF;
UPDATE P
SET BR_NASLOVA=BR_NASLOVA-1 WHERE P_SIF=x;
END
$$

CREATE TRIGGER KP_AZUR AFTER UPDATE ON KP
FOR EACH ROW
BEGIN
DECLARE x,y INTEGER;
SET x = OLD.P_SIF;
SET y = NEW.P_SIF;
UPDATE P
SET BR_NASLOVA=BR_NASLOVA-1 WHERE P_SIF=x;
UPDATE P
SET BR_NASLOVA=BR_NASLOVA+1 WHERE P_SIF=y;
END
$$
DELIMITER ;
```

Pogledi

Primer 1: Pogled koji ograničava pristup tabeli Naslov na redove sa šifrom ‘PJ’

```
CREATE VIEW NaslovPJ
AS SELECT *
FROM Naslov
WHERE SifO=‘PJ’;
```

Primer 2: Pogled koji ograničava pristup tabeli Naslov na kolone SifN i Naziv

```
CREATE VIEW Naslov1
AS SELECT SifN, Naziv
FROM Naslov;
```

Pogledi u već spominjanom primer sa fudbalerima
Data je šema relacione baze podataka fubalskogsaveza za potrebe evidencije utakmica jedne sezone (pretpostavka je da fudbaleri ne mogu da menjaju tim u kome igraju, u toku sezone):
FUDBALER (SifF, Ime, SifT)
TIM (SifT, Naziv, Mesto)
UTAKMICA (SifU, SifTDomaci, SifTGost, Kolo, Ishod, Godina)
IGRAO (SifF, SifU, PozicijaIgraca)
GOL (SifG, SifU, SifF, RedniBrGola, Minut)
KARTON (SifK, SifU, SifF, Tip, Minut)
Napomena: Ishod(1-pobeda domaćih, 2-pobeda gostiju, X-nerešeno)

5. Zadatak
Sastaviti SQL skript koji kao rezultat daje šifre i imena fudbalera kao i prosečan broj golova po utakmici koji su ti fudbaleri dali, ali samo za one fudbalere koji su dali bar jedan gol nakon što su dobili žuti karton, i to u utakmici gde njihov tim bio gostujući.

Rešenje:

```
CREATE VIEW BrUtakmica(SifF, BrUtakmica)
AS SELECT SifF, COUNT(*) FROM IGRAO GROUP BY SifF;
CREATE VIEW BrGolova(SifF, BrGolova)
AS SELECT SifF, COUNT(*) FROM GOL GROUP BY SifF;
CREATE VIEW DaliGolNakonKartona(SifF)
AS SELECT SifF
FROM FUDBALER F, UTAKMICA U, GOL G, KARTON K
WHERE F.SifF=G.SifF
AND F.SifF=K.SifF
AND G.Minut> K.Minut
AND G.SifU=K.SifU
AND G.SifU=U.SifU
AND F.SifT=U.SifTGost;
--Nije gledan tip kartona, jer se samo nakon žutog može nastaviti i dati gol
SELECT F.SifF, F.Ime, BG.BrGolova/ BU.BrUtakmica
FROM FUDBALER F, BrGolova BG, BrUtakmica BU
WHERE F.SifF=BG.SifF AND F.SifF=BU.SifF
AND F.SifF IN (SELECT SifF FROM DaliGolNakonKartona);
--Poslednji uslov je tu da bi dobili samo one igrače koji su od interesa
```

6. Zadatak
Sastaviti SQL skript koji kao rezultat daje šifre i imena fudbalera, ali samo za one fudbalere koji su dali bar dva gola i to na nekoj od utakmica na kojoj je njihov tim pobedio, i pri tom su tačno dva gola tog fudbalera bili ujedno i poslednji golovi na utakmici.

Rešenje:

```
CREATE VIEW Uslov(SifF, SifU)
AS SELECT F.SifF,U.SifU
FROM FUDBALER F, UTAKMICA U, GOL G
WHERE F.SifF=G.SifF
AND G.SifU=U.SifU
AND ( (F.SifT=U.SifTDomaci AND Ishod=‘1’)
OR (F.SifT=U.SifTGost AND Ishod=‘2’)
)
AND NOT EXISTS (SELECT SifF
FROM GOL L
WHERE L.SifF<> F.SifF
AND L.SifU=U.SifU
AND L.Minut> G.Minut
)
GROU BY F.SifF, U.SifU
HAVING COUNT(*)=2;
SELECT DISTINCT F.SifF, F.Ime
FROM FUDBALER F, Uslov U
WHERE F.SifF=U.SifF;
```

7. Zadatak
Data je šema relacione baze podataka, za potrebe izračunavanja pomoću matrica:
MATRICA(SifM, Naziv, BrVrsta, BrKolona);
PODACI(SifP, I, J, Vrednost, SifM);
Sastaviti SQL skript koji vraća vrstu, kolonu i vrednost svakog od elemenata matrice nastale množenjem matrica sa nazivima A i B.

Rešenje:

```
SELECT T2.I, T4.J, SUM(T2.Vrednost* T4. Vrednost) #T2.I su vrste iz A, T4.J su kolone iz B
FROM Matrica T1, Podaci T2, Matrica T3, Podaci T4

WHERE T1.SifM= T2.SifM AND T1.Naziv = ’A’ # T1 je matrica A, T2.Vrednost je njen podatak
AND T3.SifM= T4.SifM AND T3.Naziv = ’B’ # T3 je matrica B, T4.Vrednost je njen podatak
AND T2.J = T4.I # T2.J = T4.I = k, jer se množi a [i][k]*b[k][j]
GROUP BY T2.I, T4.J;
```

Dakle, izlazni rezultat je:

```
1 1 12
2 1 7
```

```
1 2 23
2 2 14
1 3 8
2 3 5
```

8. Zadatak
Data je šema relacione deo baze podataka fakulteta:
STUDENT (SifS, Ime, BrIndeksa)
PROFESOR (SifP, Ime, SifO)
ODSEK (SifO, Naziv)
KURS (SifK, Naziv, BrKredita, SifO)
UČIONICA (SifU, BrMesta)
PREDUSLOV (SifK, SifKP)
POHAĐA(SifS, SifR) # sifra studenta, sifra u rasporedu
RASPORED (SifR, SifP, SifK, SifU, Termin, Dan, Br.Prijavljenih)

Sastaviti SQL skript koji proverava kvalitet rasporeda studenata i pored broja indeksa ispisuje i odgovarajuću poruku. Raspored studenta je loš ukoliko u svom rasporedu bar jednog dana ima prekid u terminima u kojima prati predavanje, u suprotnom raspored se smatra dobrim. Za sve studente sa lošim rasporedom treba ispisati broj indeksa i poruku LOS, a pored broja indeksa studenata sa dobrim rasporedom poruku DOBAR.

Rešenje:

```
CREATE VIEW LosRaspored(SifS, Dan)
AS SELECT P.SifS, R.DanFROM POHADJA P, RASPORED R
WHERE P.SifR=R.SifR
GROUP BY P.SifS, R.Dan
HAVING COUNT(R.Termin) < (MAX(R.Termin)-MIN(R.Termin)+1);
SELECT S.BrIndeksa, 'LOS'FROM STUDENT S
WHERE S.SifS IN (SELECT SifS FROM LosRaspored)
UNION
SELECT S.BrIndeksa, 'DOBAR'
FROM STUDENT S
WHERE S.SifS NOT IN (SELECT SifS FROM LosRaspored)
AND S.SifS IN (SELECT SifS FROM POHADJA);
```

9. Zadatak
Data je šema relacione baze podataka veleprodajnog lanca prodavnica:
PRODAVNICA(SifP, Adresa, SifM);
MESTO(SifM, Naziv);
KLIJENT(SifK, Naziv, SifM);
RACUN(SifR, SifK, SifP, SifRa, Datum);
PROIZVOD(SifPr, Naziv, Cena);
STAVKA_RACUNA(SifS, SifR, SifPr, RedniBr, Kolicina, Iznos);
RADNIK(SifRa, Ime, SifP);
Sastaviti SQL skript koji za svaki datum, za koji je izdat bar jedan račun, daje ukupan iznos računa izdatih do tog datuma (uključujući i posmatrani datum).

Rešenje:

```
CREATE VIEW IznosPoDatumu(Datum, Total)
AS SELECT R.Datum, SUM(S.Iznos)
FROM RACUN R, STAVKA_RACUNA S
WHERE R.SifR=S.SifR
GROUP BY R.Datum;
SELECT A.Datum, SUM(B.Total)
FROM IznosPoDatumu A, IznosPoDatumu B
WHERE A.Datum>= B.Datum
GROUP BY A.Datum;
```