Database Management Systems

MySQL - Integrity Control

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
and
Centre for Artificial Intelligence and Machine Learning
Indian Statistical Institute, Kolkata

May, 2022

Basics

The term integrity in databases refers to the accuracy and consistency of data. The integrity control can set the following types of constraints on the data items:

- Basic integrity constraints
- Advanced integrity constraints

Basic integrity constraints

The basic integrity constraints are of following four types:

- Defining the `primary key` constraint
  – specified as `primary key` $(A_1, \ldots, A_k)$
- Defining the `foreign key` constraint
  – specified as `foreign key` $(A_p, \ldots, A_q)$ `references` $R(B_p, \ldots, B_q)$
- Defining the `uniqueness` constraint
  – specified as `unique`
- Defining the `nullity` constraint
  – specified as `not null`
- Defining the `check` constraint
  – specified as `check <predicate>`

Consider a relational schema

- BRANCH $= \langle \underline{branch\_id} : integer, branch\_name : string, branch\_city : string, assets : real\rangle$
- CUSTOMER $= \langle \underline{customer\_id} : integer, customer\_name : string, customer\_street : string, customer\_city : string, account\_number : integer\rangle$
- LOAN $= \langle loan\_number : integer, branch\_name : string, amount : real\rangle$
- BORROWER $= \langle customer\_name : string, loan\_number : integer\rangle$
- ACCOUNT $= \langle \underline{account\_number} : integer, branch\_name : string, balance : real\rangle$
- DEPOSITOR $= \langle customer\_name : string, account\_number : integer\rangle$

Basic integrity constraints – `primary key`

The table `BRANCH` can be created with the following SQL query:

```
 create table BRANCH(
  branch_id int(10) not null,
  branch_name varchar(30),
  branch_city varchar(30),
  assets float(20,2),
  primary key (branch_id)
);
```

Note: Multiple attributes can be defined (by putting them together as arguments separated by comma) as the primary key.

Basic integrity constraints – foreign key

The table CUSTOMER can be created with the following SQL query:

```
 create table CUSTOMER(
  customer_id int(20) not null,
  customer_name varchar(30),
  customer_street varchar(30),
  customer_city varchar(30),
  account_number int(20),
  primary key (customer_id),
  foreign key (account_number) references
Account(account_number)
);
```

Basic integrity constraints – foreign key

The table CUSTOMER can be created with the following SQL query:

```
 create table CUSTOMER(
  customer_id int(20) not null,
  customer_name varchar(30),
  customer_street varchar(30),
  customer_city varchar(30),
  account_number int(20),
  primary key (customer_id),
  foreign key (account_number) references
Account(account_number)
);
```

Note: The attribute serving as the foreign key in one table might have a different name in the referenced table.

Basic integrity constraints – unique

The table CUSTOMER can be created with the following SQL query:

```
 create table CUSTOMER(
  customer_id int(20) not null,
  customer_name varchar(30),
  customer_street varchar(30),
  customer_city varchar(30),
  account_number int(20),
  unique(customer_id)
);
```

Basic integrity constraints – null

The table LOAN can be created with the following SQL query:

```
 create table LOAN(
  loan_number int(10) not null,
  branch_name varchar(30),
  amount float(15,2)
);
```

Basic integrity constraints – check

It can be ensured that a predicate on the attributes (say the amount is non-negative) must be satisfied by every tuple in the table `LOAN` by writing an SQL query as follows:

```
 create table LOAN(
  loan_number int(10) not null,
  branch_name varchar(30),
  amount float(15,2),
  check (amount >= 0)
);
```

Advanced integrity constraints

The advanced integrity constraints can be specified with triggers. It protects the integrity of data in databases and is also useful to automate some database operations such as logging, auditing, etc.

A trigger (or database trigger) is a stored program that executes automatically in response to a specific event. E.g., insert, update or delete occurred in a table.

Note: An SQL trigger is a set of SQL statements that are stored in the database catalog.

Triggers in MySQL

One can define at most six triggers for each table. These are activated in coordination with the following events.

- **BEFORE INSERT** — before data is inserted into the table.
- **AFTER INSERT** — after data is inserted into the table.
- **BEFORE UPDATE** -– before data in the table is updated.
- **AFTER UPDATE** — after data in the table is updated.
- **BEFORE DELETE** -– before data is removed from the table.
- **AFTER DELETE** — after data is removed from the table.

Note: For MySQL version 5.7.2+, one can define multiple triggers for the same trigger event and action time.

Triggers in MySQL

To display all the triggers in the database, the following SQL query is used:

```
show triggers;
```

To delete a particular trigger from the database, the following SQL query is used:

```
drop trigger <trigger_name>;
```

Note: One must have MySQL `SUPERUSER` privileges for running trigger.

Creating triggers in MySQL

A trigger must be associated with a specific table and is written as:

```
create trigger <trigger_name>
 <trigger_time> <trigger_event> on <table_name>
 for each row
 <Triggered SQL statement>;
```

- The `trigger_time` can be `before/after`.
- The `trigger_event` can be `insert/update/delete`.
- The clause `for each row` says that the trigger activation will occur for the rows of the table, not for the table as a whole.
- The logic for the trigger is placed as a block of SQL statements.

Note: Delimiters are required in some interfaces (e.g. MySQL client).

Creating triggers in MySQL – Compound statements

A trigger can accommodate compound SQL statements as follows:

```
create trigger <trigger_name>
 <trigger_time> <trigger_event> on <table_name>
 for each row
 begin
 <Triggered SQL statement(s)>
 end;
```

Note: The `begin-end` block can take multiple SQL statements.

Creating triggers in MySQL – OLD and NEW keywords

Due to the changes in a table due to triggering, there might be some new attributes as well as some old ones. Therefore, within a triggered SQL statement, attributes are explicitly referred to as `NEW.<attribute_name>` or `OLD.<attribute_name>`.

- With `insert`, only `NEW` is legal.
- With `delete`, only `OLD` is legal.
- With `update`, both `NEW` and `OLD` are legal.

Note: The objects `NEW.<attribute_name>` and `OLD.<attribute_name>` are referred to as transition variables.

Creating triggers in MySQL – OLD and NEW keywords

Table: FACULTY

ID	NAME	EMAIL	AGE
1	Ansuman Banerjee	ansuman@isical.ac.in	45
2	Sourav Chakraborty	sourav@isical.ac.in	41
3	Malay Bhattacharyya	malaybhattacharyya@isical.ac.in	39

Creating triggers in MySQL – OLD and NEW keywords

Table: FACULTY

ID	NAME	EMAIL	AGE
1	Ansuman Banerjee	ansuman@isical.ac.in	45
2	Sourav Chakraborty	sourav@isical.ac.in	41
3	Malay Bhattacharyya	malaybhattacharyya@isical.ac.in	39

Consider the following table to keep the update details on the FACULTY table.

```
 create table FACULTY_LOG(
  ID int auto_increment primary key,
  NAME varchar(50) not null,
  LOGTIME datetime default null,
  ACTION varchar(50) default null
);
```

Creating triggers in MySQL – OLD and NEW keywords

Triggers can be set to act on the FACULTY_LOG table before making updates on the FACULTY table as follows.

```
 create trigger before_FACULTY_update
before update on FACULTY
for each row
  insert into FACULTY_LOG values (OLD.ID, OLD.NAME,
NOW(), ‘Update’);
```

Note: We use the OLD keyword to access ID and NAME attributes of the tuples affected by the trigger.

Creating triggers in MySQL – OLD and NEW keywords

Consider the following SQL query that makes an update operation on the `FACULTY` table at 00:00:01 AM on February 02, 2022.

```
 update FACULTY set AGE = 40 where NAME = "Malay
Bhattacharyya";
```

Creating triggers in MySQL – OLD and NEW keywords

Consider the following SQL query that makes an update operation on the `FACULTY` table at 00:00:01 AM on February 02, 2022.

```
update FACULTY set AGE = 40 where NAME = "Malay
Bhattacharyya";
```

This will make the following entry to the `FACULTY_LOG` table:

Table: FACULTY_LOG

ID	NAME	LOGTIME	ACTION
3	Malay Bhattacharyya	2023-02-02 00:00:01	Update

Creating triggers in MySQL – Variable declaration

Variables can be declared as follows.

```
 declare Age, Experience int default 0;
declare Name varchar(50);
declare Today date default current_date;
declare Var1, Var2, Var3 double(10,2);
```

Creating triggers in MySQL – Variable assignment

Values can be assigned to variables as follows.

```
 set var1 = 101;
set str1 = ‘Hello world’;
set v1 = 19.99;
```

Creating triggers in MySQL – Handler declarations

The handler statements deal with one or more conditions. If one of these conditions occurs, the specified statement executes. To declare handlers, the following SQL query is used:

```
declare <handler_action> handler for <condition_value>
<statement>;
```

- The `<handler_action>` can be `continue/exit/undo`.
- The `<condition_value>` can be `mysql_error_code/sqlstate/sqlwarning/sqlexception/not found`.

Creating triggers in MySQL – Conditional flow

The `if-else` construct works as follows.

```
 if <condition> then
  <statement>;
else
  <statement>;
end if;
```

Creating triggers in MySQL – Conditional flow

Consider the following table that stores names, ages and course names of some students.

```
 create table STUDENT(
  AGE int,
  NAME varchar(50),
  COURSE varchar(50)
);
```

Creating triggers in MySQL – Conditional flow

Triggers can be set to act on the Age attribute for non-negativity check before making insertions in the STUDENT table as follows.

```
 create trigger agecheck
before insert on STUDENT
for each row
begin
  if NEW.AGE < 0 then
    set NEW.AGE = abs(NEW.AGE);
  end if
end;
```

Note: We use the NEW keyword to access the new values inserted in the table.

Creating triggers in MySQL – Conditional flow

Consider the following SQL query:

```
 insert into STUDENT values (23, ‘Sujan’, ‘MTech’),
(22, ‘Vikas’, ‘MTech’), (-24, ‘Ravindra’, ‘MTech’),
(23, ‘Uddalok’, ‘MTech’);
```

This will turn the `STUDENT` table as follows.

Table: STUDENT

AGE	NAME	COURSE
23	Sujan	MTech
22	Vikas	MTech
24	Ravindra	MTech
23	Uddalok	MTech

Creating triggers in MySQL – Repetitive flow

The `loop` construct works as follows.

```
 <loop_name>:  loop
  <statement>;
  if <condition> then
    <statement>;
    leave <loop_name>;
  end if;
  <statement>
end loop <loop_name>;
```

Creating triggers in MySQL – Repetitive flow

The `while` construct works as follows.

```
 <loop_name>:  while <condition> do
  <statement>
  if <condition> then
    leave <loop_name>;
  end if;
  <statement>
end while;
```

Creating triggers in MySQL – Repetitive flow

The `repeat-until` construct works as follows.

```
 <loop_name>:  repeat
  <statement>;
  if <condition> then
    <statement>;
    leave <loop_name>;
  end if;
  <statement>;
until <condition> end repeat;
```

Note: The repetitive block executes once irrespective of the `<condition>`.

Creating multiple triggers in MySQL

Multiple triggers are written as follows:

```
 delimiter $$ -- The end delimiter is now ‘$$’
create trigger <trigger_name>
  <trigger_time> <trigger_event> on <table_name>
  for each row
  <Triggered SQL statement> -- Works well with ‘;’
create trigger <trigger_name>
  <trigger_time> <trigger_event> on <table_name>
  for each row
  begin
  <Triggered SQL statement(s)> -- Works well with ‘;’
  end;
$$
delimiter ; -- The delimiter is changed back to ‘;’
```

Limitations of triggers in MySQL

A MySQL trigger cannot perform the following things:

- Using `SHOW`, `LOAD DATA`, `LOAD TABLE`, `BACKUP DATABASE`, `RESTORE`, `FLUSH`, `RETURN` statements.
- Using statements that commit or rollback implicitly or explicitly such as `COMMIT`, `ROLLBACK`, `START TRANSACTION`, `LOCK/UNLOCK TABLES`, `ALTER`, `CREATE`, `DROP`, `RENAME`.
- Using prepared statements such as `PREPARE`, `EXECUTE`.
- Using dynamic SQL statements.

Basics of cursors

A cursor allows to iterate a set of rows returned by a query and process each row accordingly. It is used to handle a result set inside a stored procedure.

MySQL cursors have the following properties:

- **Read-only:** Cursors cannot be used to update data in the underlying table.
- **Non-scrollable:** Rows can only be fetched in the order determined by the select statement. One cannot skip rows or jump to a specific row in the result set.
- **Asensitive:** MySQL cursors are asensitive. It is often faster because it does not need to make a temporary copy of data.

Basics of cursors

A cursor allows to iterate a set of rows returned by a query and process each row accordingly. It is used to handle a result set inside a stored procedure.

MySQL cursors have the following properties:

- **Read-only:** Cursors cannot be used to update data in the underlying table.
- **Non-scrollable:** Rows can only be fetched in the order determined by the select statement. One cannot skip rows or jump to a specific row in the result set.
- **Asensitive:** MySQL cursors are asensitive. It is often faster because it does not need to make a temporary copy of data.

Note: Cursors are of two types – asensitive and insensitive. An asensitive cursor points to the actual data, whereas an insensitive cursor uses a temporary copy of the data. It is safer not to update the data used by an asensitive cursor.

Creating cursors

To define a cursor, the following SQL query is used:

```
declare <cursor_name> cursor for <select_statement>;
```

To initialize the result set for the cursor, before fetching rows from the result set, the following SQL query is used:

```
open <cursor_name>;
```

To retrieve the next row pointed by the cursor and move the cursor to the next row in the result set, the following SQL query is used:

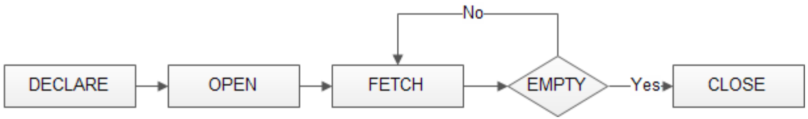
```
fetch <cursor_name> into <list_variables>;
```

To close a cursor, the following SQL query is used:

```
close <cursor_name>;
```

Data handling with cursors

The entire life cycle of an MySQL cursor is illustrated in the following diagram.



Data handling with cursors – An example

Suppose we want to build an email list of all employees from the `FACULTY` table.

Let us first declare some variables, a cursor for looping over the emails of employees, and a NOT FOUND handler.

```
 declare Var_Done int default 0;
declare email varchar(500) default "";
declare email_cursor cursor for select email from
FACULTY;
declare continue handler for not found set Var_Done =
1;
```

Data handling with cursors – An example

Now, open the email_cursor as follows:

```
open email_cursor;
```

Then, iterate the email list, and concatenate all the emails where each email is separated by a semicolon as follows:

```
 get_email:  loop
  fetch email_cursor into email;
  if Var_Done = 1 then
    leave get_email;
  end if;
  set email_list = concat(email,";",email_list);
end loop get_email;
```

Problems

1. Consider the following schema of an online code repository system like GitHub:
   - Contributor = $\langle contributor\_name : string, contributor\_id : integer \rangle$
   - Code-Group = $\langle \underline{contributor\_id} : integer, code\_group : string, count\_submissions : integer \rangle$

   i) Set the basic integrity constraints on this schema.
   ii) Write an SQL trigger to restrict all the possible events that can make violations to the above schema.