Ian Kenny

October 31, 2016

Databases

Lecture 5

In this lecture

- The Data Manipulation Language.
- The Data Definition Language.
- Introduction to Views.
- Introduction to Triggers.

Introduction

SQL contains what is called a *data manipulation language* (DML) and a *data definition language* (DDL).

The SQL SELECT and INSERT commands are part of the DML. We have already covered SELECT and INSERT but we will briefly cover some other varieties of these commands. There are other commands in the DML that will also now cover.

The DDL contains commands relating to the creating of tables and altering of tables. We will also now cover some of these commands (but not CREATE).

DML

From the DML we will now cover

- INSERT INTO
- SELECT INTO
- INSERT INTO SELECT
- UPDATE
- DELETE

INSERT INTO

The basic form of the SQL INSERT INTO command is

```
INSERT INTO table_name
VALUES (value1,value2,value3,...),
(value1,value2,value3,...),
...);
```

It has the alternative form

```
INSERT INTO table_name (column1,column2,column3,...)
VALUES (value1,value2,value3,...);
```

With this form we specify the columns and their values. In comparison with the first form of this command above, this form simply allows you to specify the columns and values in a different order than they appear in the table.

SELECT INTO

The SELECT INTO command allows us to select data from one or more tables and insert it into a *new* table. The new table is little more than a copy of the existing table(s), or a temporary table containing the result of a query. The basic form of the SQL SELECT INTO command is

```
SELECT column_name(s)
INTO newtable
FROM table_name1[,table_name2] ...
[WHERE condition];
```

As usual, '*' can be used instead of a column list to generate all columns.

Only data is copied. Not constraints and other information.

SELECT INTO

This command creates a new table that is a copy of the Artist table.

```
SELECT *
INTO artistCopy
FROM artist;
```

Only data is copied. Not constraints and other information.

SELECT INTO

This command creates a table containing reviews of albums by The Beatles.

```
SELECT album.title, review.rating
INTO beatlesReviews
FROM artist NATURAL JOIN review NATURAL JOIN album
WHERE artist.name = 'The Beatles';
```

Only data is copied. Not constraints and other information.

This is essentially a standard query command but it creates a new table *in the database* from the query.

INSERT INTO SELECT

This command inserts data from one table into an *existing* table. The tables involved must have the same attributes and domains.

An example also involving the previous command will make this command clearer.

INSERT INTO SELECT

Consider the case where we wish to create a single 'sales' table in the original music database. Remember, that database had separate tables for sales from 2015 and 2016. The following commands will create this table.

```
SELECT *
INTO sales
FROM sale15;

INSERT INTO sales
SELECT *
FROM sale16;
```

Only data is copied. Not constraints and other information.

UPDATE

The basic form of the UPDATE command is

```
UPDATE table_name
SET column1=value1,column2=value2,...
WHERE some_column=some_value;
```

The WHERE clause is **very** important. Without it ALL of the rows will be updated.

UPDATE

This command sets the text of all of reviews where no comment was made to “Hey, why not leave a comment!”

```
UPDATE review
SET text= 'Hey, why not leave a comment!'
WHERE text IS NULL;
```

The WHERE clause is **very** important. Without it ALL of the rows will be updated.

UPDATE

This command updates the rating of all David Bowie albums to 5.

```
UPDATE review
SET rating = 5
WHERE albumid IN
   (SELECT albumid
   FROM album
   WHERE album.artistid IN
      (SELECT artist.artistid
      FROM artist
      WHERE artist.name = 'David Bowie'));
```

DELETE

The basic form of the DELETE command is

```
DELETE FROM table_name
WHERE some_column=some_value;
```

The WHERE clause is **very** important. Without it ALL of the rows will be deleted!

DELETE

This command will delete all reviews left by the customer whose id is 6

```
DELETE FROM review
WHERE custid = 6;
```

The WHERE clause is **very** important. Without it ALL of the rows will be deleted!

DELETE

If you actually *want* to delete all of the rows from a table but not the table itself then you can use DELETE. This command deletes all data from review but not the table itself.

```
DELETE FROM review;
```

DELETE

What if we want to DELETE records that are referenced by other tables? We cannot do that unless we have specified that it is allowed. To do that we need to modify the CREATE TABLE statement so that it allows the referencing row to be deleted when the referenced row is deleted.

Consider the case where we wish to delete the customer whose ID is 6. Maybe they wish to be removed entirely from our database. This means, for example, deleting their reviews. To do this we need to add a clause to the foreign key declaration in the Review table.

DELETE

```
CREATE TABLE review(
albumID int REFERENCES album(albumID) NOT NULL,
custID int REFERENCES customer(custID)
   ON DELETE CASCADE NOT NULL,
rating int NOT NULL,
text varchar (1024),
PRIMARY KEY (albumID, custID));
```

Other values for ON DELETE are RESTRICT which prevents the deletion from the referenced table and NO ACTION which checks if the referential integrity has been violated *after* trying to execute the command.

DDL

From the DDL we will now cover

- DROP
- ALTER

DROP

The DROP command can be used to remove a table and all of its data. This command deletes the table 'label' and all of its data

```
DROP TABLE label;
```

DROP

As with the DML command DELETE, we need to consider the case where dependencies exist between tables. If we attempt to delete a table that has attributes that are referenced from another table (as a foreign key) then the default behaviour is to prevent the delete going ahead. If we wanted to delete, for example, the customer table, there are other tables that reference it. To delete the customer table, and all foreign key references to it, we need to specify CASCADE

```
DROP TABLE customer CASCADE;
```

The foreign key constraints in referencing tables (in this case, for example, review), will be removed.

ALTER

The ALTER command allows us to add or delete a column, or change a column's data type.

To add a column, the syntax is as follows

```
ALTER TABLE table_name
ADD new_column_name data_type;
```

For example, to add a column called emailaddress to customer

```
ALTER TABLE customer
ADD emailaddress varchar(128);
```

ALTER

To delete a column, the syntax is

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

For example, to delete the emailaddress column in customer

```
ALTER TABLE customer
DROP COLUMN emailaddress;
```

ALTER

To change the data type for a column, the syntax is

```
ALTER TABLE table_name
ALTER COLUMN column_name new_data_type;
```

For example, to allocate a greater number of characters to the customer email address

```
ALTER TABLE customer
ALTER COLUMN emailaddress varchar(256);
```

Views

We saw above that we can create a new table from a query using SELECT INTO. The result of such a query is a copy of a table (or tables) or a temporary result of a query. A new table is created in the database.

A better approach may be to use *views*. A view is stored as an SQL statement containing a query, but its result looks like a table and can be used as a table. It is not actually a table, however.

Some of the advantages of views are that they can provide additional security control, i.e. producing a view that excludes some information; they are updated when the tables used in the view are updated since they are essentially a query running on those tables. They remain up-to-date, therefore. They can also be used to provide a consistent interface to the data, based upon disparate table structures.

Creating a view

A view is essentially an alias for a query, but one that is stored and can be accessed as if it was a table. The syntax for a view is therefore

```
CREATE VIEW view_name AS
(QUERY)
```

i.e.

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name(s)
WHERE condition
```

More complex statements can also be used.

Creating a view

This query creates a view showing ratings of albums by The Beatles.

```
CREATE VIEW beatlesReviews AS
SELECT album.title, review.rating
FROM review NATURAL JOIN artist NATURAL JOIN album
WHERE artist.name = 'The Beatles';
```

Now we can do

```
SELECT *
FROM beatlesReviews;
```

Introduction to triggers

Triggers enable us to execute procedures when various events occur in the database. Triggers allow us to use a procedural language to define additional processing on the database. Triggers can be applied to events such as INSERT and UPDATE to perform additional processing. They can usually be set to trigger before the event or after it.

Various procedural languages are supported but it is common to use a language native to the DBMS. For example, for *Postgres* the language is PL/pgSQL.

Introduction to triggers

Triggers can be used to maintain the integrity of data in the database (where perhaps the DBMS's own constraints are insufficient); to create meta data, i.e. to store data about the data, changes to the data, etc.; to create new entries in tables when an entry has been added to another table (e.g. to create a new entry in a 'works-in-department' table when a new employee is added), etc.

Introduction to triggers

Triggers are somewhat 'controversial'. This is because, some developers argue, that triggers can cause problems since they can change the state of the data. They argue that this is best done in the application code. If it is done in the application code, they argue, it is transparent. It is easy to forget about a trigger perhaps written years ago and then to wonder why the data is ending up in a state not applied by the application.

Others argue that the database logic is best kept in the database, and this can be enforced by triggers. If the database logic is placed in the applications, it may need maintaining across multiple applications, and different applications may impose different rules due to their local needs or errors.

If data is always be kept in a standard format, for example, this is best enforced in the database since then the standardised format will be accessed by all applications rather than the applications having responsibility for that. Triggers can be used to enforce that.

Creating a trigger in *PostgreSQL*

One of the most common uses of triggers (and, some argue, the only acceptable use of triggers), is to create audit information. This means, when data is changed in a table (e.g. employee salaries ...), updating a separate table with information about the change in the data.

In *PostgreSQL* the process for creating a trigger is

- create the procedure to execute on the trigger.
- create the trigger.

Creating a trigger in *PostgreSQL*

In this example, we see how to monitor price changes to albums. The procedure is defined as follows

```
CREATE OR REPLACE FUNCTION log_price_changes()
RETURNS trigger AS
$BODY$
BEGIN
IF NEW.price <> OLD.price THEN
INSERT INTO pricechanges(albumid,priceBefore, priceAfter)
VALUES(OLD.albumid, OLD.price, NEW.price);
END IF;

RETURN NEW;
END;
$BODY$
LANGUAGE 'plpgsql';
```

In this example, the table pricechanges must already exist.

Creating a trigger in *PostgreSQL*

In this example, we see how to monitor price changes to albums. The trigger is defined as follows

```
CREATE TRIGGER price_changes
AFTER UPDATE
ON album
FOR EACH ROW
EXECUTE PROCEDURE log_price_changes();
```

At the point at which this statement is executed, the referenced table (album in this case) and the procedure must exist.

Creating a trigger in *PostgreSQL*

Now, if an album price changes, that information will be stored in the pricechanges table. For example, if this statement is executed

```
UPDATE album
SET price = 20.99
WHERE albumid = 0;
```

An entry will be made in the pricechanges table with the data

(0, 7.99, 20.99)

Creating a trigger in *PostgreSQL*

Consider a further example where we want to store in a table data about the amount of money spent by customers. Each time they buy an album a single entry is made into the table. The procedure could be defined as follows. This example shows the use of a query in the trigger procedure.

Creating a trigger in *PostgreSQL*

```
CREATE OR REPLACE FUNCTION log_sale()
RETURNS trigger AS
$BODY$
BEGIN
INSERT INTO salessum(custid, total)
  VALUES (NEW.custid, (select price from album where
  album.albumid = NEW.albumid));
RETURN NEW;
END;
$BODY$
LANGUAGE 'plpgsql';
```

Again, table salessum must exist.

Creating a trigger in *PostgreSQL*

The trigger would look like this

```
CREATE TRIGGER new˙sale
AFTER INSERT
ON sale
FOR EACH ROW
EXECUTE PROCEDURE log_sale();
```

The table sale must be defined at this point.