Complex Systems Theory and Practice

Lecture 2

Piotr Syga

2021-10-15

Base of a query

Schema of a simple query

Select *columns*
From *table*
Where *condition*

Base of a query

Schema of a simple query

Select *columns*
From *table*
Where *condition*

Example

Select Speed
From A
Where weight>150;

Base of a query

Schema of a simple query

Select *columns*
From *table*
Where *condition*

Example

```
Select Distinct  speed
From             A
Where            weight>150;
```

More than a set.. – Ordering and result manipulation

As a general rule we want to interpret a table as a set, yet it may not be true in some specific cases, in particular after a query involving projection. Hence the result of the query is not a set:

sorting — `ORDER BY` *column*(, *column2*, . . .) (`DESC`)

grouping — `GROUP BY` *column*(, *column2*, . . .) – we use aggregation functions so that columns not being grouped may be replaced by a single entry

filtering — `HAVING` *condition* – allows filtering **after** the query, hence filtering by aggregated column is possible

row cap — `LIMIT` (`TOP`, `ROWNUM`< number) – **Important:** the place of a cap in the query differs for each capping keyword

Aggregation

- `MIN(`*column*`)`, `MAX(`*column*`)`, `SUM(`*column*`)`
- `AVG(`*column*`)`, `VARIANCE(`*column*`)`, `STD(`*column*`)`
- `COUNT(`*column*`)`, `COUNT(`*column, column2*`)`, `COUNT(*)`
- `GROUP_CONCAT(`*column*`)`
- bit operations (logical and shifts), file operations, ...

Applying to the base schema

Schema of a longer query

Select	*columns*
From	*table alias* Join *table alias2*
Where	*conditions*
Group by	*columns*
Order by	*columns*
Having	*condition*

Applying to the base schema

The order

1. We establish syntactical correctness of the query.
2. We check whether all entities exists and do we have access to them.
3. `FROM` clause is considered (also creating aliases and derived tables)
4. `WHERE` clause is verified.
5. `GROUP`ing takes place.
6. Aggregate functions are calculated at the `SELECT` clause.
7. `ORDER`ing of the result.
8. Filtering with `HAVING` clause.
9. Projection (selecting only the stated columns) at the `SELECT` is made.

It is important to note that independent parts of queries may be considered in a different order due to optimization, most often when subqueries may be modified.

CREATING a table

Basics

`CREATE TABLE` Name (column type);

Types

Numeric	String	Date
bit(n)	(national) char(n) ($\star$)	date
(unsigned) #int(n)	(national) varchar(n) ($\star$)	datetime
bool	binary(n)	timestamp
(unsigned) decimal(n,d)	text(n)	time
(unsigned) float(n,d)	blob(n)	year
(unsigned) double(n,d)	enum (*vals*)($\star$) | set(*vals*)($\star$)	

– int, tinyint, smallint, mediumint, bigint

$\star$: `CHARACTER SET _ COLLATE _`

`SHOW CHARACTER SET;`

`SHOW COLLATION WHERE` Charset ='*selected_charset*';

Enhancing the creation

```
CREATE (TEMPORARY) TABLE (IF NOT EXISTS) Name(
column0 int NOT NULL AUTO_INCREMENT,
column1 type1 (COMMENT comment text),
column2 type2 (NULL),
column3 type3 (DEFAULT val),
refToTab2 varchar(30),
FOREIGN KEY (refToTab2)
 REFERENCES Tab2(name)
 ON DELETE CASCADE,
CHECK (column2>=threshval),
CONSTRAINT constName CHECK (column1<tval AND column3 LIKE patt)
);
```

Example

```
CREATE TABLE Metals (
Id  int NOT NULL AUTO_INCREMENT,
name varchar(50) NOT NULL,
alloy Boolean,
action enum ('push', 'pull),
actType smallint,
class smallint,
FOREIGN KEY (actType) REFERENCES Externatilities(id)
 ON DELETE CASCADE,
FOREIGN KEY (class) REFERENCES Classes(id) ON DELETE CASCADE
);
```

Note: We assume the existence of the referenced tables.

Learning the schema

SHOW keyword

In order to list all the tables in the current database we may use `SHOW TABLES`. The `SHOW` keyword has more uses: listing events, errors, columns, `CREATE` statement of a given entity, access levels, character coding, plugins, replication modes or databases located on the server.

Table schema

To learn the names of the columns in a table, their type, constraints and default values we may use `SHOW` as well as `DESCRIBE` keywords:
`SHOW COLUMNS FROM` *Table*;
`DESCRIBE` *Table*;

Filling out the tables

known rows

```
INSERT INTO Metals
VALUES ('pewter',1,'push',2,1);
```

limited values

```
INSERT INTO Metals (name,alloy)
VALUES ('gold',0);
```

Note: The schema has to allow NULLs in the skipped attributes or have default values, if not an error is returned.

selected data

```
INSERT INTO EnhAlloy (name, actType)
SELECT name, actType
FROM Metals WHERE alloy='true' AND class=(SELECT id
from Classes WHERE name='Enhancement');
```

Modifying the record

Update

We cannot 'add' a single value to a record, however we can modify exsisting record or its part

```
UPDATE Metals
SET action = 'pull'
WHERE name = 'gold';
```

Delete

Similarly, we can delete a single record: `DELETE FROM Metals WHERE name LIKE '%ol%';`

Note: Deleting records based on condition other than involving PK will rise a warning and may require modifying `SQL_SAFE_UPDATES` flag.

Extended deletes and updates

Updating with multiple tables

```
UPDATE Tab T1 INNER JOIN Tab T2 ON T1.ID=T2.ID
AND T2.to_update = 1
SET T1.update_count = T1.update_count+1;
```

Deleting with multiple tables

```
DELETE FROM Tab1
WHERE ID IN (SELECT T1.ID FROM a T1 JOIN b T2 ON
T1.ID = T2.ID);
```

Modifying the structure of the table

When modifying the table, not its content, we always use `ALTER TABLE` keywords.

Adding a column

```
ALTER TABLE Metals ADD (COLUMN) Description int FIRST;
ALTER TABLE Metals ADD (COLUMN) Compound1 varchar(15) AFTER alloy;
ALTER TABLE Metals ADD (COLUMN) Perc1 int AFTER Compound1;
ALTER TABLE Metals ADD (COLUMN) Compound2 varchar(15) AFTER Perc1;
ALTER TABLE Metals ADD (COLUMN) Perc2 int AFTER Compound2;
```

Modifying a column

`ALTER TABLE` *name* `MODIFY` *column* `type DEFAULT` *default value null options* `COMMENT` *comments*;

Removing columns

A single or multiple columns

```
ALTER TABLE Metals DROP (COLUMN) Description;

ALTER TABLE Metals DROP COLUMN Perc1, DROP COLUMN Perc2;
```

Dependencies

Note, that dropping a column that is used outside of a table (e.g., trigger, index, FK) will rise error and stop the script.

Changing constraints

We can add or drop a constraints just as the columns

Changing starting point of artificial keys

```
ALTER TABLE Metals AUTO_INCREMENT=17;
```

Adding constraints

```
ALTER TABLE Metals ADD CHECK (class<5);
ALTER TABLE Metals
ADD CONSTRAINT cstName CHECK (actType<=2 AND class<5);
```

Dropping constaints

```
ALTER TABLE Metals DROP CHECK cstName;
```

Views – the general idea

- simplifying data
- sharing data with users that do not have access to given tables
- merging information from multiple tables, filtering out columns and rows
- 'saving' queries
- backwards compability

Creating Views

Making a view

`CREATE (OR REPLACE)` *algorithm* `VIEW` name(columns) `AS`
`SELECT` columns `FROM` Table;

Algoritms

`TEMPTABLE` – required for aggregate functions, grouping, filtering, set operations, subqueries ...

`MERGE` – simple, time efficient solution, allows using indexes

`UNDEFINED` – checks if `MERGE` method may be used in given case or forbidden keywords are used if such keywords are present in the select `TEMPTABLE` is used

Changing views

The views may be modified just as tables (`ALTER`, `DROP`) if given user has appropriate access level. Note that using view's columns 'outside' does not result in error.

Creating and altering the database

Basics

`CREATE DATABASE (IF NOT EXISTS)` database `(CHARACTER SET` charset`)`;
`DROP DATABASE (IF EXISTS)` database;

Altering

`ALTER DATABASE` database `CHARACTER SET` charset (other properties);

Users in the database

Checking existing users

`SELECT * FROM mysql.user;`
Most important columns: User, Host, Password/authentication_string.

Adding a user

```
CREATE USER 'user'@'localhost'; (CREATE USER 'user'@'%';)
SET PASSWORD FOR 'user'@'localhost' = PASSWORD('pass');
ALTER USER 'user'@'localhost' WITH MAX_QUERIES_PER_HOUR
(MAX_UPDATES_PER_HOUR | MAX_CONNECTIONS_PER_HOUR) 10;
```

Access levels

Most columns in mysql.user describes the access level of a user to the the database.
List of all permissions of a user may be verified by `SHOW GRANTS FOR user@host`.

Adding permissions

```
GRANT ALL PRIVILEGES ON *.* TO 'user'@'localhost';
GRANT SELECT ON *.* TO 'usr'@'localhost';
FLUSH PRIVILEGES;
```

Examples of Privileges	Objects
Alter	*.*
Create	db.*
Drop	table
Select	

List of all permissions may be learned by `SHOW PRIVILEGES`.
To remove a permission we `REVOKE` a grant, e.g., REVOKE ALL ON Metals FROM 'user'@'localhost';

Creating a procedure

Simple example

```
DELIMITER $$
CREATE PROCEDURE counter (OUT result INT)
BEGIN
 SELECT COUNT(*) INTO result FROM Metals;
END $$
DELIMITER ;
```

Using a procedure

```
CALL counter(@res);
SELECT @res;
```

Removing a procedure

```
DROP IF EXISTS counter;
```

Creating a function

Simple example

```
CREATE FUNCTION strMerger (str varchar(10), s varchar(10))
 RETURNS VARCHAR(21) DETERMINISTIC
 RETURN CONCAT(str,' ',s);
```

Using a function

```
SELECT strMerger('complex','systems');
```

Important: Note the difference between variables(`var`) and user defined variables (`@var`).

Prepared statements

```
SET @col = "name";
SET @str = CONCAT("SELECT ", @col, " FROM Metals");
PREPARE stmt FROM @str;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

How to make a procedure that allows using @col as a input parameter?

Prepared statements

```
Delimiter $$
CREATE PROCEDURE uniCol (IN col VARCHAR(50))
BEGIN
SET @str = CONCAT("SELECT ", col, " FROM Metals");
PREPARE stmt FROM @str;
EXECUTE stmt;
END$$
DELIMITER ;
```

We can call the procedure by `CALL uniCol('name')`, `CALL uniCol('name, id')` or `SET @c = "name";CALL uniCol(@c)'`