**MySQL**

**Assignment -4**

**Trainer – Nishi Verma**

1) The Maximum Value for a Column

"What is the highest item number?"

```
SELECT MAX(article) AS article FROM shop;
+---------+
| article |
+---------+
|       4 |
```

2) The Row Holding the Maximum of a Certain Column

*Task: Find the number, dealer, and price of the most expensive article.*

```
SELECT article, dealer, price
FROM   shop
WHERE  price=(SELECT MAX(price) FROM shop);
+---------+--------+-------+
| article | dealer | price |
+---------+--------+-------+
|    0004 | D      | 19.95 |
```

Other solutions are to use a LEFT JOIN or to sort all rows descending by price and get only the first row using the MySQL-specific LIMIT clause:

```
SELECT s1.article, s1.dealer, s1.price
FROM shop s1
LEFT JOIN shop s2 ON s1.price < s2.price
WHERE s2.article IS NULL;
SELECT article, dealer, price
FROM shop
ORDER BY price DESC
LIMIT 1;
```

3) Maximum of Column per Group

*Task: Find the highest price per article.*

```
SELECT article, MAX(price) AS price
FROM   shop
GROUP BY article
ORDER BY article;
+---------+-------+
| article | price |
+---------+-------+
|    0001 |  3.99 |
|    0002 | 10.99 |
|    0003 |  1.69 |
|    0004 | 19.95 |
+---------+-------+
```

4) The Rows Holding the Group-wise Maximum of a Certain Column

*Task: For each article, find the dealer or dealers with the most expensive price.*
This problem can be solved with a subquery like this one:

```
SELECT article, dealer, price
FROM   shop s1
WHERE  price=(SELECT MAX(s2.price)
              FROM shop s2
              WHERE s1.article = s2.article)
ORDER BY article;
+---------+--------+-------+
| article | dealer | price |
+---------+--------+-------+
|    0001 | B      |  3.99 |
|    0002 | A      | 10.99 |
|    0003 | C      |  1.69 |
|    0004 | D      | 19.95 |
+---------+--------+-------+
```

Other possibilities for solving the problem are to use an uncorrelated subquery in the FROM clause, a LEFT JOIN, or a common table expression with a window function.
Uncorrelated subquery:

```
SELECT s1.article, dealer, s1.price
FROM shop s1
JOIN (
  SELECT article, MAX(price) AS price
  FROM shop
  GROUP BY article) AS s2
  ON s1.article = s2.article AND s1.price = s2.price
ORDER BY article;
```

Left Join :

```
SELECT s1.article, dealer, s1.price
FROM shop s1
JOIN (
  SELECT article, MAX(price) AS price
  FROM shop
  GROUP BY article) AS s2
  ON s1.article = s2.article AND s1.price = s2.price
ORDER BY article;
```

5) Using Foreign Keys

In MySQL, InnoDB tables support checking of foreign key constraints. See The InnoDB Storage Engine, and FOREIGN KEY Constraint Differences.
A foreign key constraint is not required merely to join two tables. For storage engines other than InnoDB, it is possible when defining a column to use
a REFERENCES ***tbl_name***(***col_name***) clause, which has no actual effect, and *serves only as a memo or comment to you that the column which you are currently defining is intended to refer to a column in another table*. It is extremely important to realize when using this syntax that:

- MySQL does not perform any sort of check to make sure that ***col_name*** actually exists in ***tbl_name*** (or even that ***tbl_name*** itself exists).
- MySQL does not perform any sort of action on ***tbl_name*** such as deleting rows in response to actions taken on rows in the table which you are defining; in other words, this syntax induces no ON DELETE or ON UPDATE behavior whatsoever. (Although you can write an ON DELETE or ON UPDATE clause as part of the REFERENCES clause, it is also ignored.)
- This syntax creates a *column*; it does ***not*** create any sort of index or key.

```
CREATE TABLE person (
    id SMALLINT NOT NULL AUTO_INCREMENT,
    name CHAR(60) NOT NULL,
    PRIMARY KEY (id)
);
CREATE TABLE shirt (
    id SMALLINT  NOT NULL AUTO_INCREMENT,
    style ENUM('t-shirt', 'polo', 'dress') NOT NULL,
    color ENUM('red', 'blue', 'orange', 'white', 'black') NOT NULL,
    owner SMALLINT UNSIGNED NOT NULL REFERENCES person(id),
    PRIMARY KEY (id)
);
INSERT INTO person VALUES (NULL, 'Antonio Paz');
SELECT @last := LAST_INSERT_ID();
INSERT INTO shirt VALUES
(NULL, 'polo', 'blue', @last),
(NULL, 'dress', 'white', @last),
(NULL, 't-shirt', 'blue', @last);
INSERT INTO person VALUES (NULL, 'Lilliana Angelovska');
SELECT @last := LAST_INSERT_ID();
INSERT INTO shirt VALUES
(NULL, 'dress', 'orange', @last),
(NULL, 'polo', 'red', @last),
(NULL, 'dress', 'blue', @last),
(NULL, 't-shirt', 'white', @last);
SELECT * FROM person;
+----+--------------------+
| id | name               |
+----+--------------------+
|  1 | Antonio Paz        |
|  2 | Lilliana Angelovska |
+----+--------------------+
SELECT * FROM shirt;
+----+---------+--------+-------+
| id | style   | color  | owner |
+----+---------+--------+-------+
|  1 | polo    | blue   |     1 |
|  2 | dress   | white  |     1 |
|  3 | t-shirt | blue   |     1 |
|  4 | dress   | orange |     2 |
|  5 | polo    | red    |     2 |
|  6 | dress   | blue   |     2 |
|  7 | t-shirt | white  |     2 |
+----+---------+--------+-------+
SELECT s.* FROM person p INNER JOIN shirt s
  ON s.owner = p.id
```

```
 WHERE p.name LIKE 'Lilliana%'
   AND s.color <> 'white';
+----+-------+--------+-------+
| id | style | color  | owner |
+----+-------+--------+-------+
|  4 | dress | orange |     2 |
|  5 | polo  | red    |     2 |
|  6 | dress | blue   |     2 |
+----+-------+--------+-------+
```

6) Calculating Visits Per Day

The following example shows how you can use the bit group functions to calculate the number of days per month a user has visited a Web page.

```
CREATE TABLE t1 (year YEAR, month INT UNSIGNED,
            day INT UNSIGNED);
INSERT INTO t1 VALUES(2000,1,1),(2000,1,20),(2000,1,30),(2000,2,2),
           (2000,2,23),(2000,2,23);

SELECT year,month,BIT_COUNT(BIT_OR(1<<day)) AS days FROM t1
       GROUP BY year,month;
```