Note that `LAST_INSERT_ID()` is tied to the session, so even if multiple connections are inserting into the same table, each with get its own id.

Your client API probably has an alternative way of getting the `LAST_INSERT_ID()` without actually performing a `SELECT` and handing the value back to the client instead of leaving it in an `@variable` inside MySQL. Such is usually preferable.

Longer, more detailed, example

The "normal" usage of IODKU is to trigger "duplicate key" based on some `UNIQUE` key, not the `AUTO_INCREMENT PRIMARY KEY`. The following demonstrates such. Note that it does *not* supply the `id` in the INSERT.

Setup for examples to follow:

```
CREATE TABLE iodku (
    id INT AUTO_INCREMENT NOT NULL,
    name VARCHAR(99) NOT NULL,
    misc INT NOT NULL,
    PRIMARY KEY(id),
    UNIQUE(name)
) ENGINE=InnoDB;

INSERT INTO iodku (name, misc)
    VALUES
    ('Leslie', 123),
    ('Sally', 456);
```

```
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0
+----+--------+------+
| id | name   | misc |
+----+--------+------+
|  1 | Leslie |  123 |
|  2 | Sally  |  456 |
+----+--------+------+
```

The case of IODKU performing an "update" and `LAST_INSERT_ID()` retrieving the relevant `id`:

```
INSERT INTO iodku (name, misc)
    VALUES
    ('Sally', 3333)            -- should update
    ON DUPLICATE KEY UPDATE    -- `name` will trigger "duplicate key"
    id = LAST_INSERT_ID(id),
    misc = VALUES(misc);
SELECT LAST_INSERT_ID();       -- picking up existing value
```

```
+------------------+
| LAST_INSERT_ID() |
+------------------+
|                2 |
+------------------+
```

The case where IODKU performs an "insert" and `LAST_INSERT_ID()` retrieves the new `id`:

```
INSERT INTO iodku (name, misc)
    VALUES
    ('Dana', 789)          -- Should insert
```

```
    ON DUPLICATE KEY UPDATE
    id = LAST_INSERT_ID(id),
    misc = VALUES(misc);
SELECT LAST_INSERT_ID();   -- picking up new value
```

```
+------------------+
| LAST_INSERT_ID() |
+------------------+
|                3 |
+------------------+
```

Resulting table contents:

```
SELECT * FROM iodku;
```

```
+----+--------+------+
| id | name   | misc |
+----+--------+------+
|  1 | Leslie |  123 |
|  2 | Sally  | 3333 |  -- IODKU changed this
|  3 | Dana   |  789 |  -- IODKU added this
+----+--------+------+
```

Section 10.5: INSERT SELECT (Inserting data from another Table)

This is the basic way to insert data from another table with the SELECT statement.

```
INSERT INTO `tableA` (`field_one`, `field_two`)
   SELECT `tableB`.`field_one`, `tableB`.`field_two`
   FROM `tableB`
   WHERE `tableB`.clmn <> 'someValue'
   ORDER BY `tableB`.`sorting_clmn`;
```

You can `SELECT * FROM`, but then `tableA` and `tableB` *must* have matching column count and corresponding datatypes.

Columns with `AUTO_INCREMENT` are treated as in the `INSERT` with `VALUES` clause.

This syntax makes it easy to fill (temporary) tables with data from other tables, even more so when the data is to be filtered on the insert.

Section 10.6: Lost AUTO_INCREMENT ids

Several 'insert' functions can "burn" ids. Here is an example, using InnoDB (other Engines may work differently):

```
CREATE TABLE Burn (
    id SMALLINT UNSIGNED AUTO_INCREMENT NOT NULL,
    name VARCHAR(99) NOT NULL,
    PRIMARY KEY(id),
    UNIQUE(name)
        ) ENGINE=InnoDB;

INSERT IGNORE INTO Burn (name) VALUES ('first'), ('second');
SELECT LAST_INSERT_ID();          -- 1
SELECT * FROM Burn ORDER BY id;
```