
elis
MySQL

Il linguaggio SQL

Giovani programmatori e sviluppatori nel settore ICT

Introduction to MySQL

MySQL Websites

- http://www.mysql.com includes:
  - Product information
  - Services (Training, Certification, Consulting, and Support)
  - White papers, webinars, and other resources
  - MySQL Enterprise Edition downloads (trial versions)
- http://dev.mysql.com includes:
  - Developer Zone (forums, articles, Planet MySQL, and more)
  - Documentation
  - Downloads
- https://github.com/mysql
  - Source code for MySQL Server and other MySQL products

Community Resources

- Mailing lists
- Forums
- Developer articles
- MySQL Newsletter (published monthly)
- Planet MySQL blogs
- Social media channels
  - Facebook, Twitter, and Google+
- Physical and virtual events, including:
  - Developer days
  - MySQL Tech Tours
  - Webinars
- Bug tracking
- Github repositories

MySQL-Supported Operating Systems

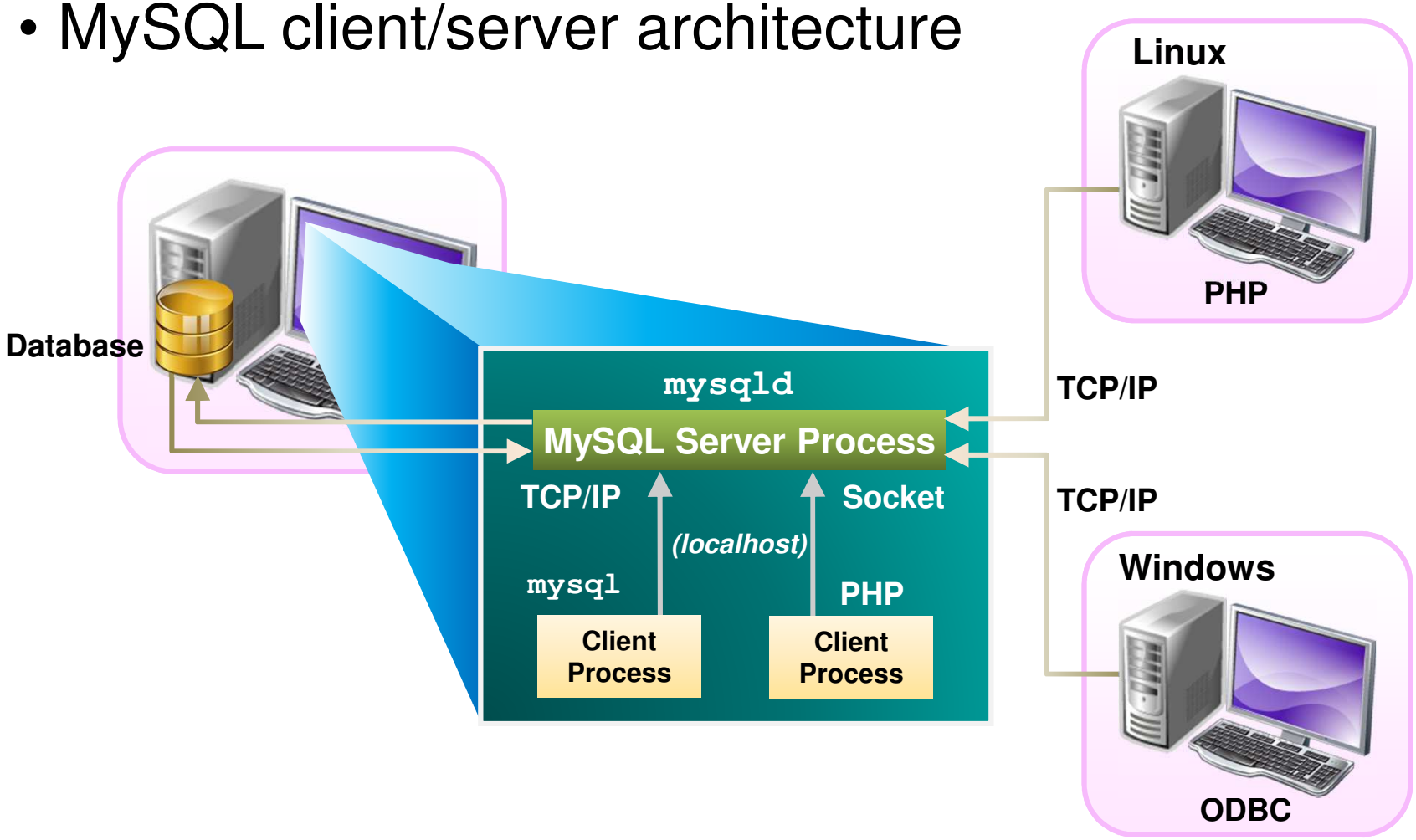
- MySQL:
  - Provides control and flexibility for users
  - Supports multiple commodity platforms, including:
    - Windows (x86, x86_64)
    - Linux (x86, x86_64)
    - Oracle Solaris (SPARC_64, x86_64, x86)
    - Mac OS X (x86, x86_64)
  - Can be compiled to run on other platforms

MySQL Server and Client

- **Connecting to the Database**

MySQL Client/Server Model

- MySQL client/server architecture



MySQL Connectors

- Clients are applications that connect to a MySQL server and issue database statements.
- Connectors (drivers) enable clients to access data on the server.
  - These are available for multiple operating systems.
- The official MySQL-supported connectors include:
  - Connector/ODBC
  - Connector/J
  - Connector/NET
  - Connector/Python
  - Connector/C and Connector C++
- The embedded MySQL server library is `libmysqld`.
- Multiple PHP extensions and APIs are supported.

Installing MySQL

- The download locations for MySQL include:
  - Community Edition (GPL): http://dev.mysql.com/downloads
  - Commercial editions: https://edelivery.oracle.com/
- MySQL is available for multiple platforms, including Windows, Linux, and Mac OS X.
  - This course uses the Linux operating system.
  - Install MySQL on Linux by using Yum with RPM packages.
  - Additional RPM packages are available for MySQL management tools.
- Example databases for testing MySQL features are available at http://dev.mysql.com/doc/index-other.html.
  - This course uses the `world` database for examples and practices.

Starting and Stopping MySQL

- You can start the MySQL server on Linux in the following ways:
  - From the Linux command line
  - Automatically on boot
  - With MySQL Workbench
- In Windows, the MySQL service starts automatically at system startup.
  - You can manage the MySQL service from the Windows Management Console.
  - You can start or stop MySQL server from MySQL Workbench.

Starting and Stopping the MySQL Server on Linux

- On Linux, start the MySQL server and verify the status with the following commands:

```
# service mysqld start
Starting mysqld:                                           [  OK  ]
# service mysqld status
mysqld (pid  22918) is running...
#
```

- On Linux, stop the MySQL server and verify the status with the following command:

```
# service mysqld stop
Stopping mysqld:                                           [  OK  ]
# service mysqld status
mysqld is stopped
#
```

Starting the MySQL Server Automatically

- On Linux, you can set the MySQL server to start automatically on system startup and check the status of automatic startup with the `chkconfig` command:

```
# chkconfig mysqld on
# chkconfig --list mysqld
mysqld              0:off     1:off     2:on 3:on 4:on 5:on
    6:off
#
```

- The numbers followed by "on" or "off" indicate runlevels that control the modes that the server uses when starting. This example, with runlevels 2, 3, 4, and 5 so on, indicates that the MySQL server is using Multi-User Mode with support for MySQL Workbench.

Using the MySQL Command-Line Tool

- In a production environment, applications access the database to query data and update the database.
- The `mysql` client is a useful command-line tool for some of the following tasks:
  - **User administration:** Setting up users and permissions
  - **Database administration:** Importing and exporting data, setting configuration options
  - **Database creation:** Creating databases and checking their structure
  - **Query validation:** Testing and optimizing queries

Starting the MySQL Command-Line Tool

- Start the `mysql` command-line client with the following command:

```
# mysql -u root -p
Enter password: <password>
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.7.12 MySQL Community Server (GPL)
...
Type 'help;' or '\h' for help. Type '\c' to clear the
   current input statement.
mysql>
```

Keyboard Editing

- To edit `mysql` client commands, you can take advantage of the following:
  - Arrow keys
  - Backspace key to delete the character to the left of the cursor
  - The command history preserved during the session
  - Character sequence (**`\c`**) to cancel a command being entered
  - Enforced quote closure

mysql Client Commands

- Some useful `mysql` client commands include the following:
  - `PROMPT`: Changes the prompt
  - `HELP`: Displays a list of MySQL commands
  - `EXIT` or `QUIT`: Exits the `mysql` client

```
mysql> PROMPT (\u@\h) [\d]>\_
PROMPT set to '(\u@\h) [\d]>\_'
(root@localhost) [(none)]> PROMPT
Returning to default PROMPT of mysql>
mysql> HELP
<list of all MySQL commands>
mysql> EXIT
Bye
# service mysqld status
mysqld (pid  22918) is running...
```

tee File Logging

- The `tee` file is a text file that logs all `mysql` client statements and their output.
- Everything you see displayed on the screen is appended into a file that you specify.
- You can start logging by entering the **`tee`** command interactively within the `mysql` client:

```
mysql> tee session_tee_filename.txt
```

- You can stop `tee` file logging with the **`notee`** command.
- Executing `tee` again re-enables logging and appends the new output to the existing file.
- View the `tee` file contents in your favorite text editor.

Setting the Initial Password for the root Account

- The process for setting the initial password for the MySQL `root` login is the following:
  - The first time you start `mysqld` after installation, it generates a random password and stores it in `/var/log/mysqld.log`.
  - Copy the random password from that file and use it to log in to the `mysql` client as `root`.
  - Set the password for the `root` account to one that you select.

Accessing the Initial Random Password for root

- Start the `mysqld` server with the following command:

```
# service mysqld start
Initializing MySQL database:2015-08-05T01:00:36.565655Z0
Starting mysqld:                                           [  OK  ]
#
```

- View the random password with the following command:

```
# head /var/log/mysqld.log
2015-08-05T01:00:38.043563Z 0 [Warning] InnoDB: New log
   files created, LSN=45790
2015-08-05T01:00:38.431937Z 0 [Warning] InnoDB: Creating
   foreign key constraint system tables.
2015-08-05T01:00:40.581515Z 1 [Warning] A temporary
   password is generated for root@localhost: be1(BwQLh;Ia
150805 01:01:01 mysqld_safe Starting mysqld daemon
```

Logging In and Setting the root Password

- Log in to MySQL with the random password copied from the log:

```
# mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
…
Type 'help;' or '\h' for help. Type '\c' to clear the
   current input statement.
```

Enter the following statement to set the `root` password:

```
mysql> ALTER USER USER()
    -> IDENTIFIED BY 'oracle';
Query OK, 0 rows affected (0.00 sec)
mysql>
```

Resetting a Forgotten root Password

- If you forget the password for the `root` account, you can follow these steps to reset the forgotten `root` password:
  - Start the server with the following options:
    - **`--skip-grant-tables`:** Enables anyone to connect without a password
    - **`--skip-networking`:** Prevents remote clients from connecting
  - Log in to the `mysql` client as `root` with no password.
  - Use the `FLUSH PRIVILEGES` statement to load grant tables.
  - Set the password for the `root` account.
  - Stop the server and restart it normally.
  - Log in to the `mysql` client with the new `root` password.

Statements to Reset a Forgotten root Password

- Reset a forgotten `root` password with the following statements:

```
# service mysqld start --skip-grant-tables
--skip-networking
Starting mysqld:                                    [
   OK  ]
# mysql -u root
Welcome to the MySQL monitor. Commands end with ;
   or \g.
…
Type 'help;' or '\h' for help. Type '\c' to clear
   the current input statement.
mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)
mysql> ALTER USER USER()
    -> IDENTIFIED BY 'oracle';
Query OK, 0 rows affected (0.00 sec)
mysql> exit
# service mysqld stop
```

MySQL Workbench

- A unified (GUI) tool that helps database architects, developers, and DBAs manage the MySQL server and data.
  - Use MySQL Workbench for:
    - SQL development
    - Data modeling
    - Server administration
    - Database migration
  - There are three editions of MySQL Workbench:
    - Community Edition (GPL)
    - Standard Edition (Commercial)
    - Enterprise Edition (Commercial)
  - MySQL Workbench Runs on Linux/UNIX, Windows, and Mac OS X.

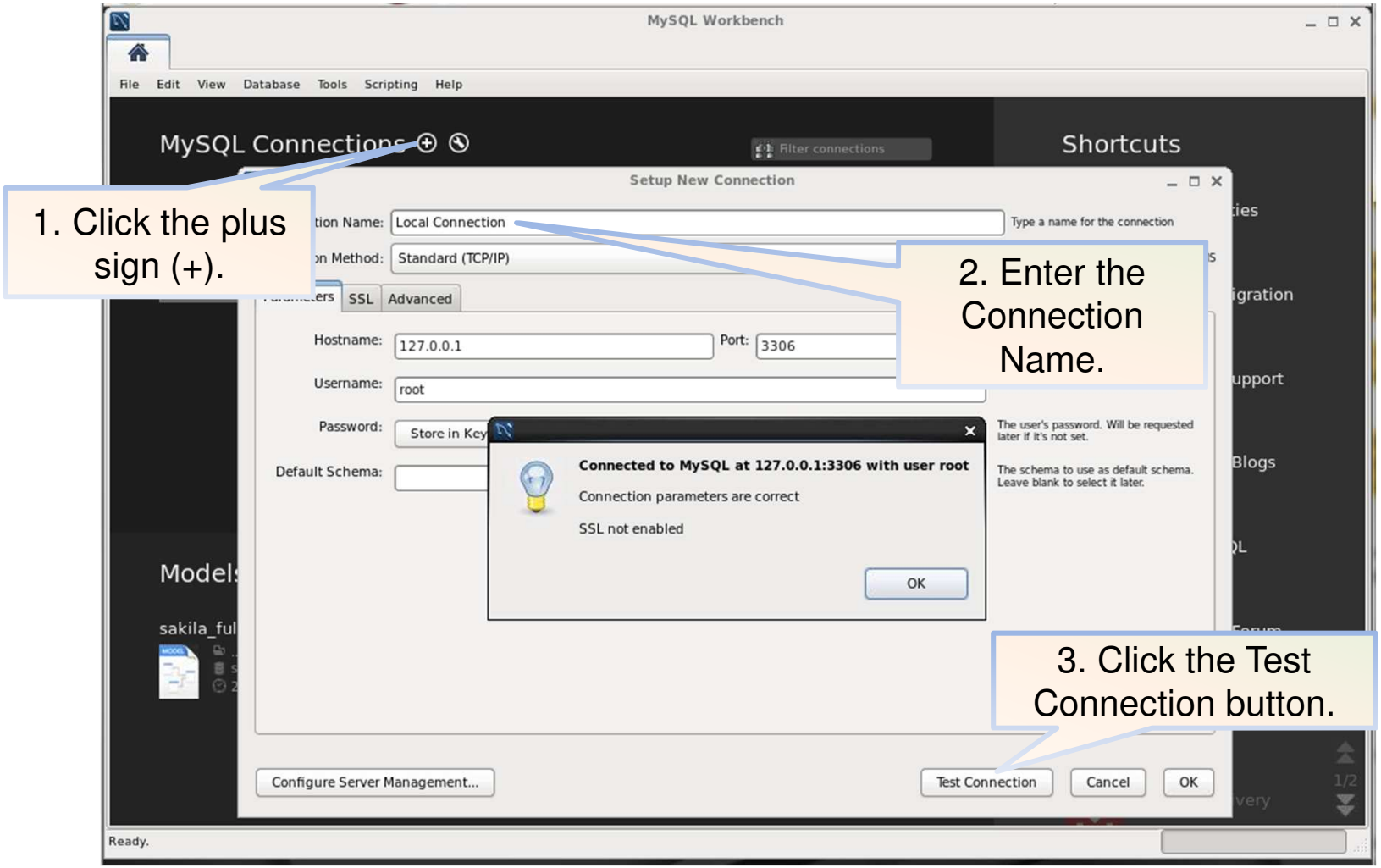
MySQL Workbench Capabilities

- SQL Development
  - Edit and execute SQL queries and scripts.
  - Create or alter database objects.
  - Edit table data.
- Data Modeling
  - Extended entity-relationship (EER) modeling
  - Edit and execute SQL queries and scripts.
  - Design, generate, and manage databases.
- Server Administration
  - Start and stop the server.
  - Edit database server configuration.
  - Manage users.
  - Import and export data.

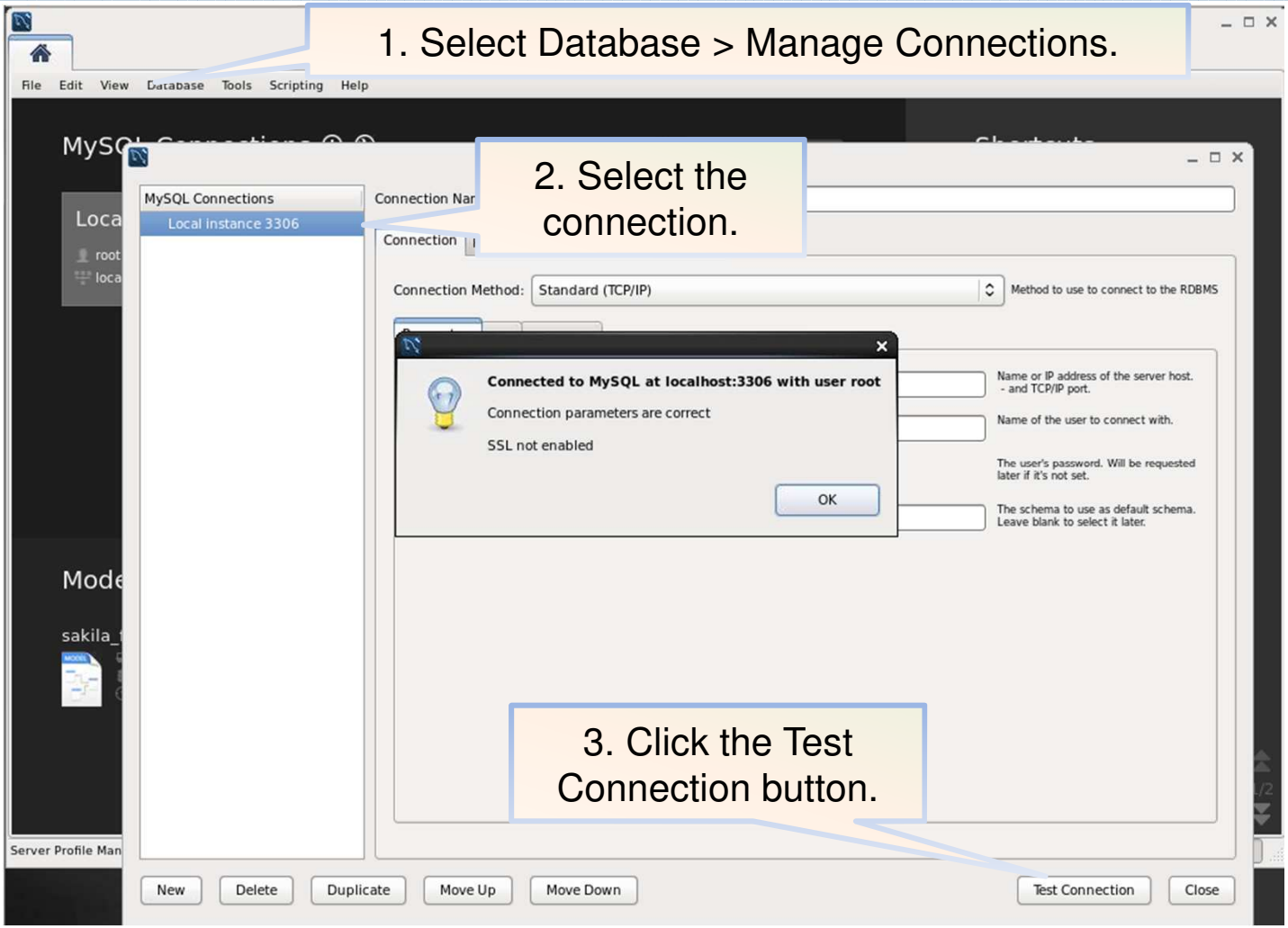
MySQL Workbench Additional Features and Utilities

- Database Migration
  - Migrate tables and objects from other RDBMSs to MySQL.
  - Convert existing applications to run on MySQL on Windows and other platforms.
  - Upgrade to later versions of MySQL.
- Access to MySQL Utilities
  - MySQL Utilities are Python tools for working with MySQL Server.
  - MySQL Workbench provides a GUI for working with these tools.

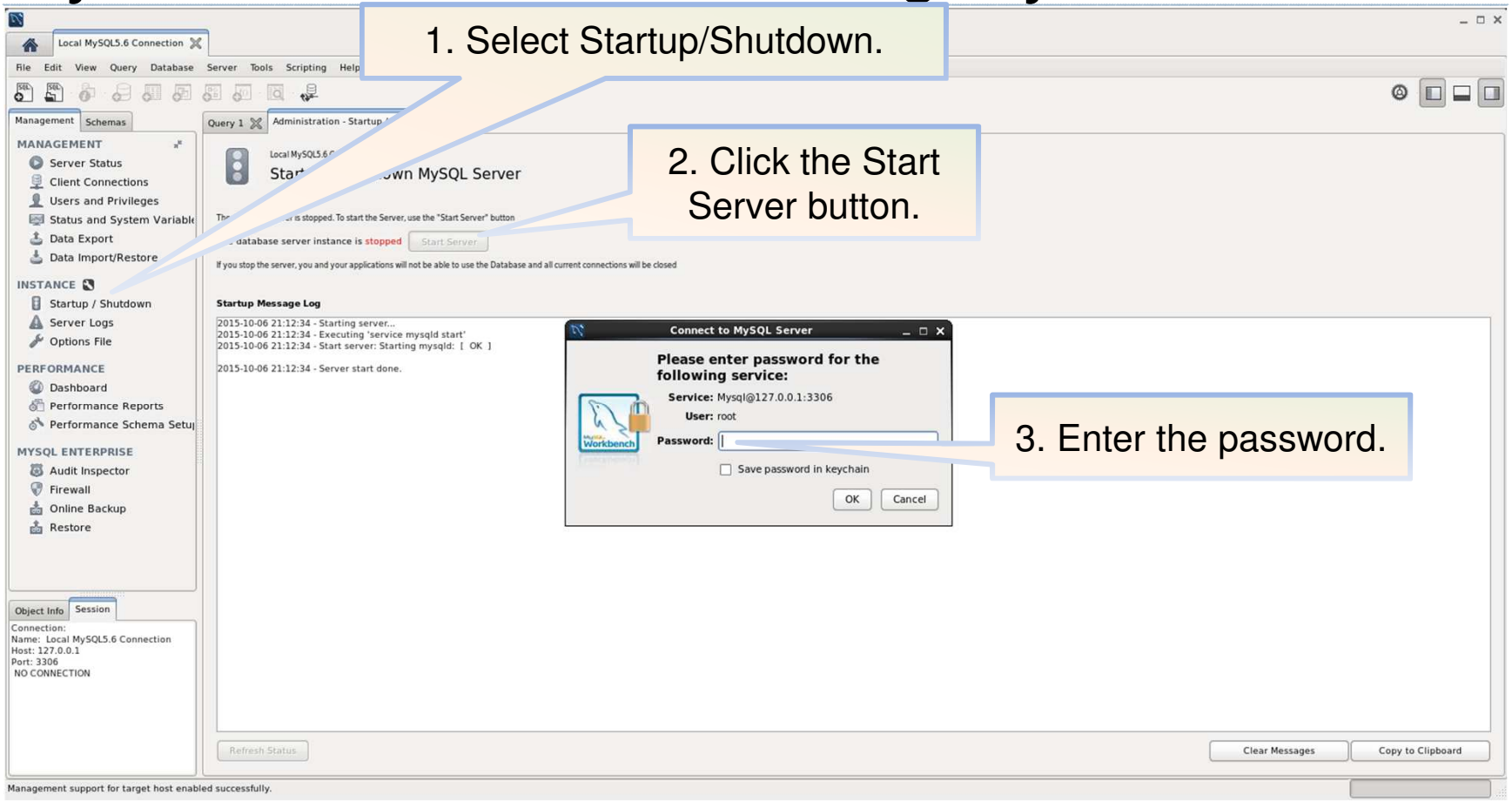
MySQL Workbench: Creating a Connection



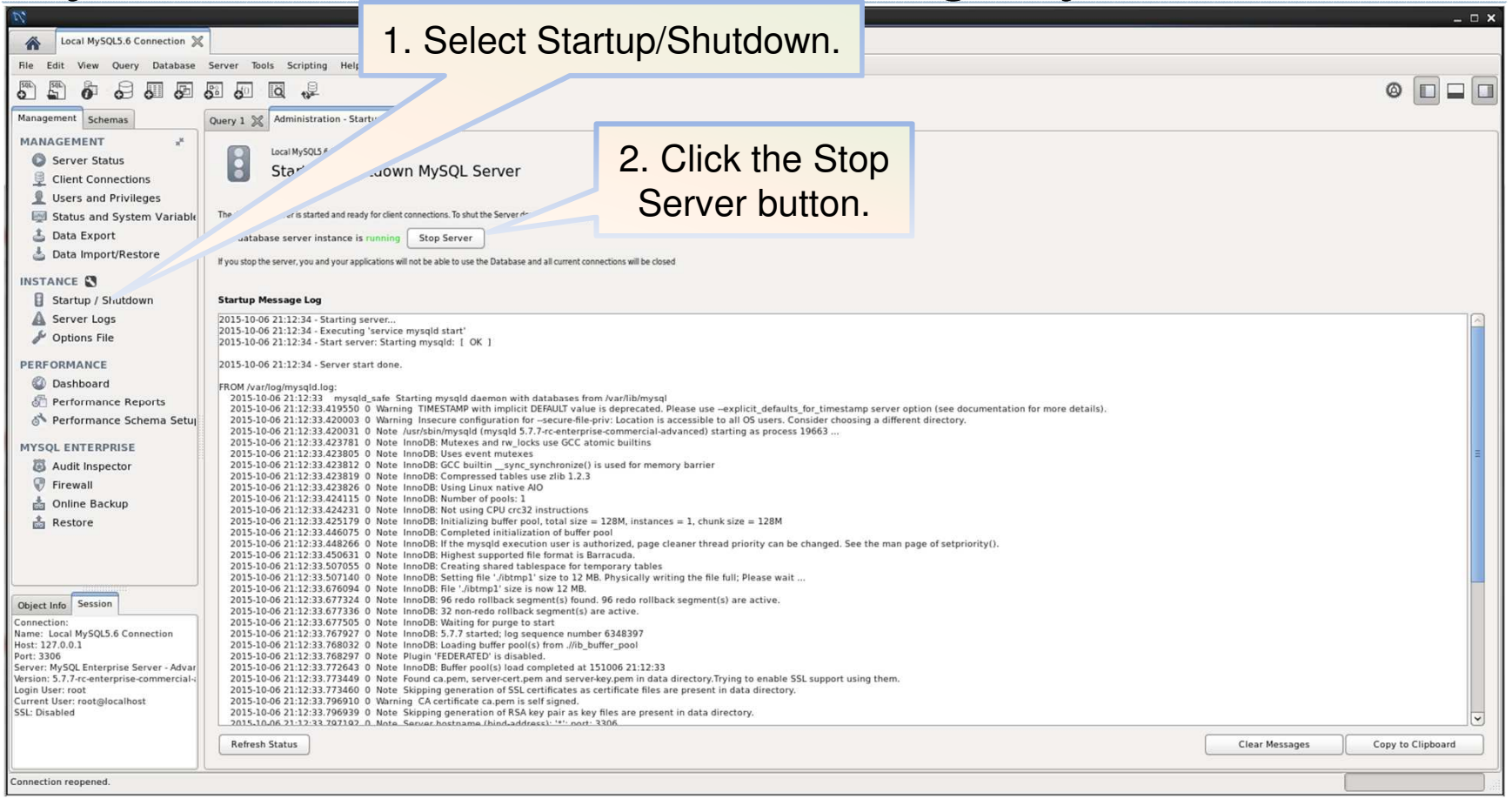
MySQL Workbench: Managing Connections



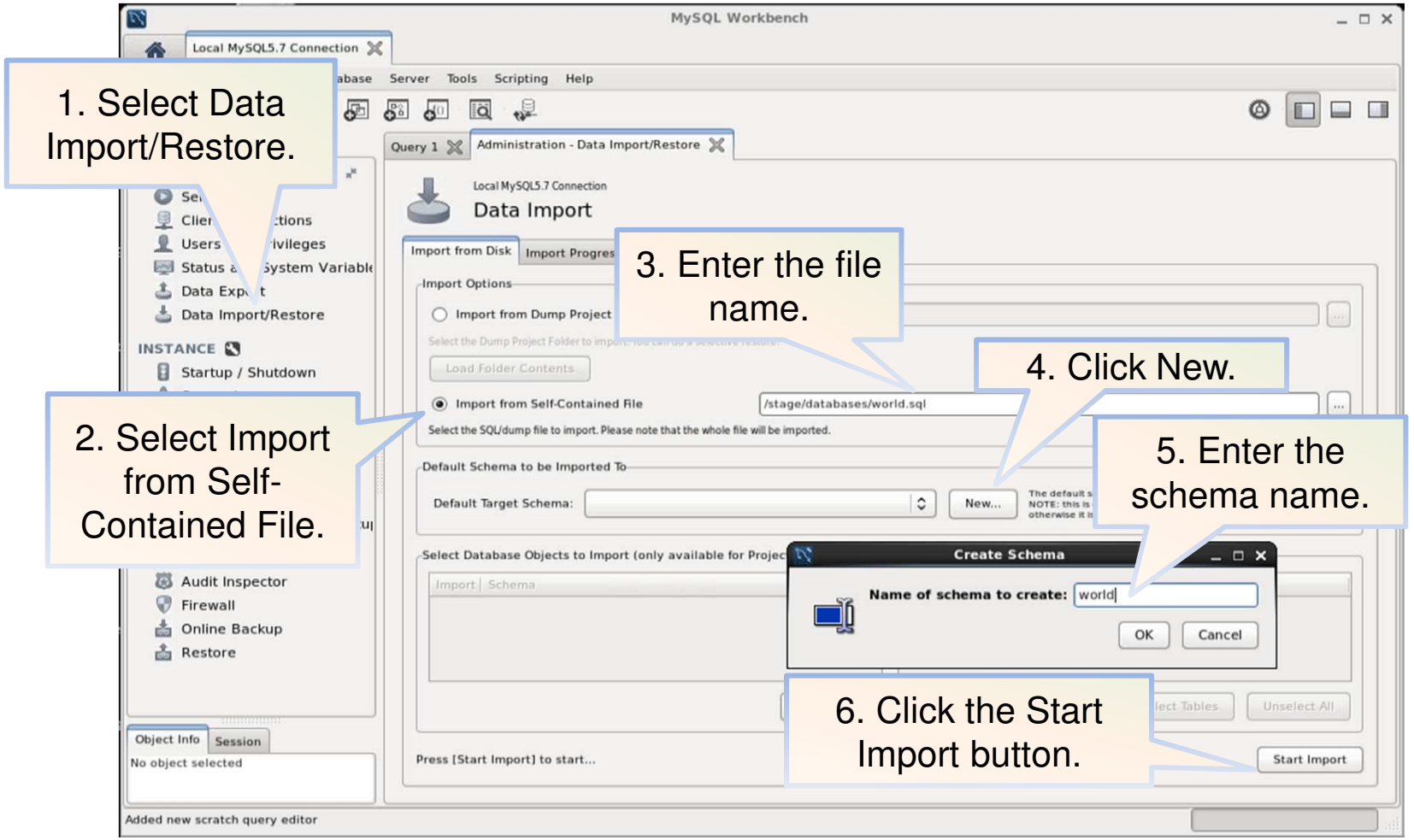
MySQL Workbench: Starting MySQL Server



MySQL Workbench: Stopping MySQL Server



MySQL Workbench: Importing Data



Querying Table Data

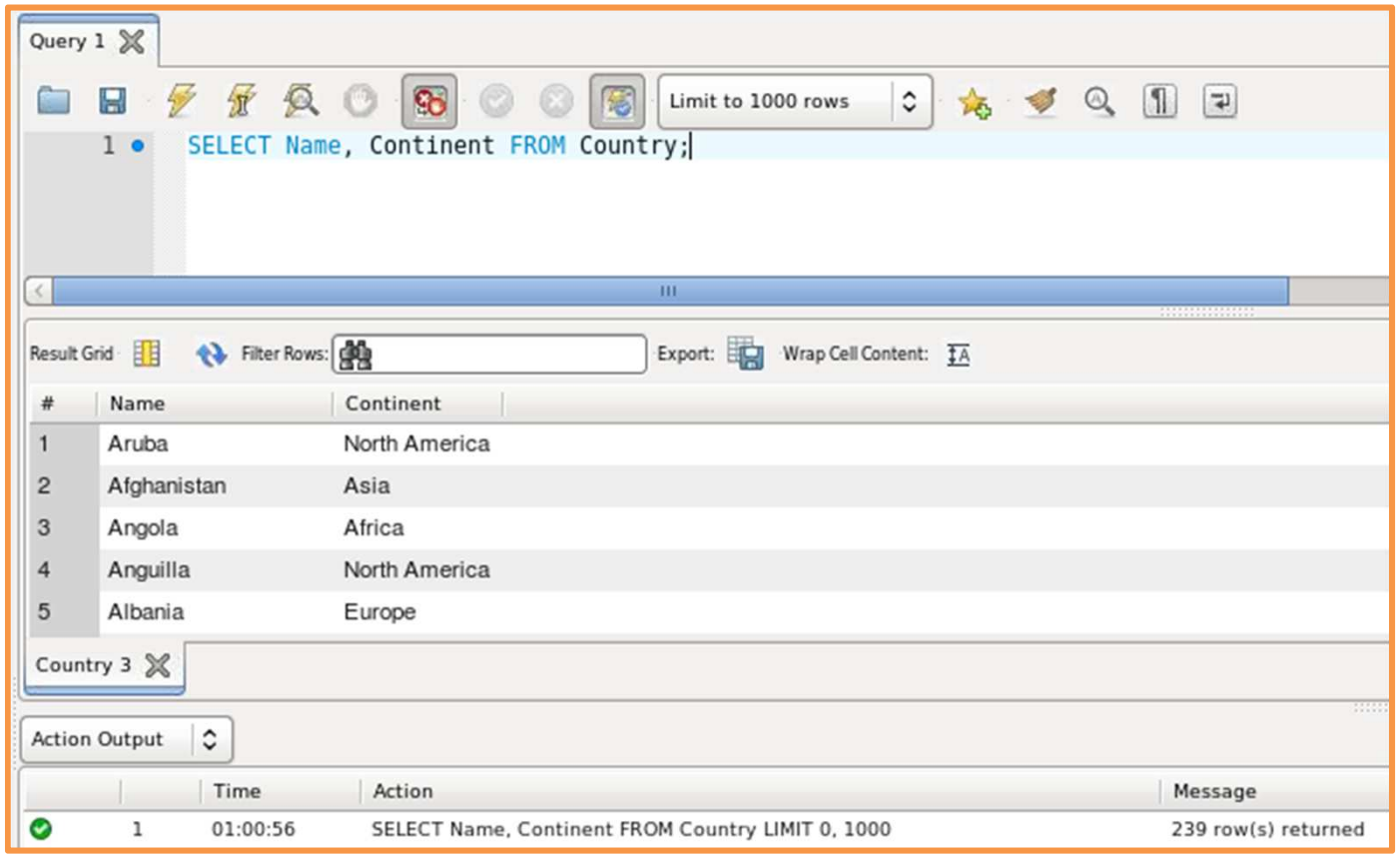
SELECT Statement

- `SELECT` is the most commonly used DML command for queries:
  - Retrieves rows from tables in a database
  - Returns rows as a “result set” in the form of a table
- General syntax:

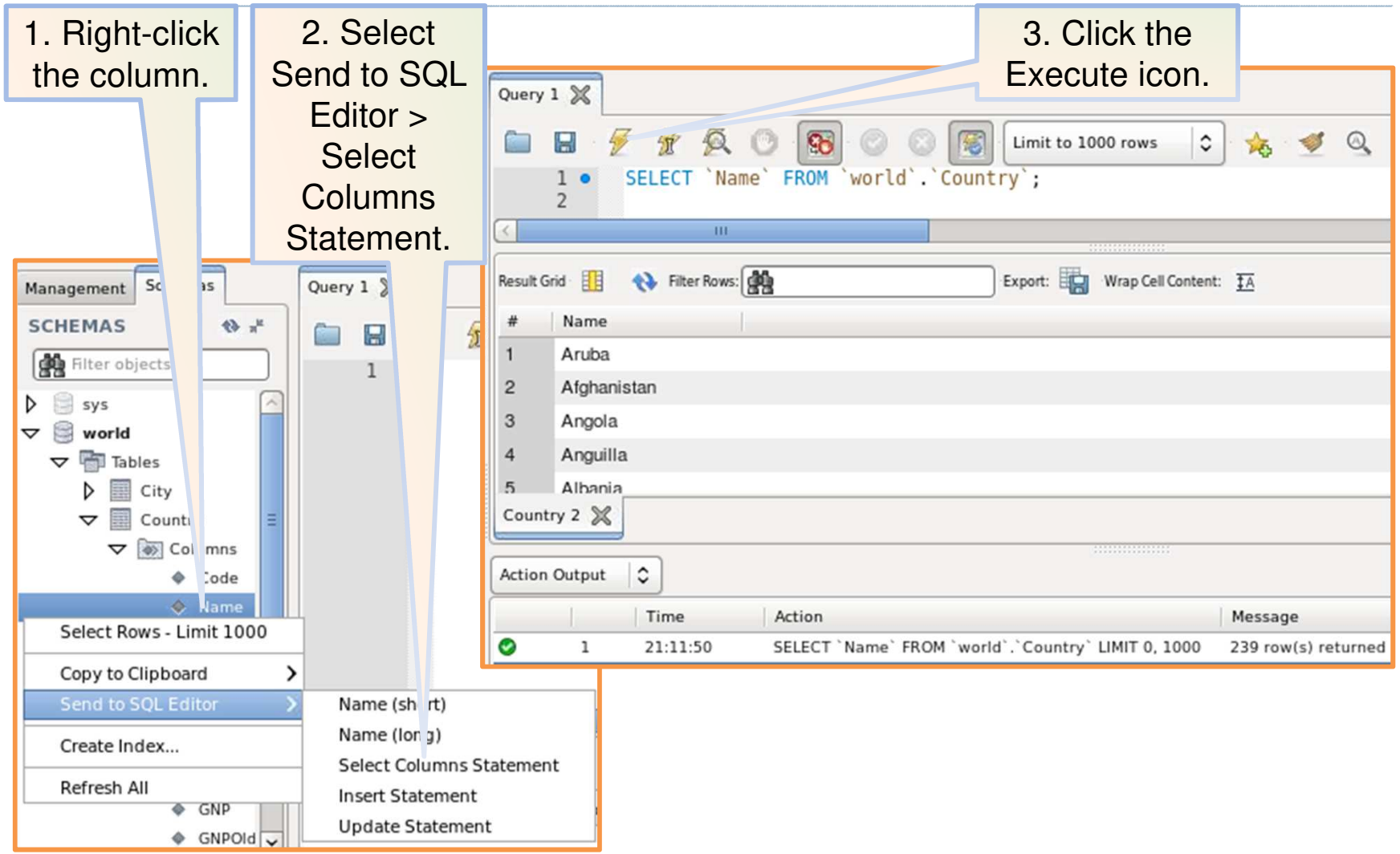
```
SELECT [clause options] column_list
    [FROM table_name] [other_clauses]
```

- ***`column_list`*** is a list of column names that make up the result set.
  - Separate the items in the list with a comma separator (`,`).
  - Use an **`*`** to include all columns rather than list them separately.

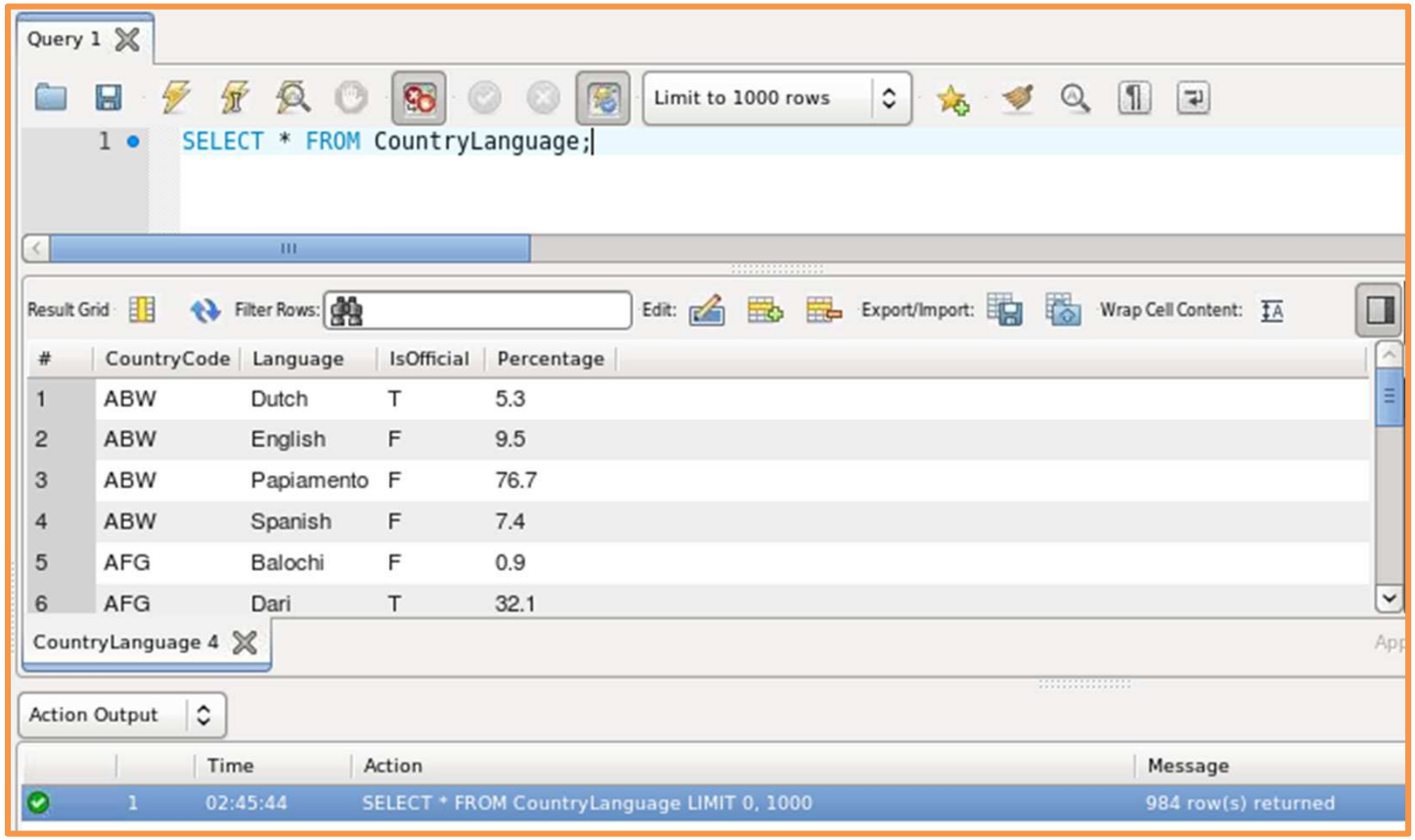
SELECT Statement Example



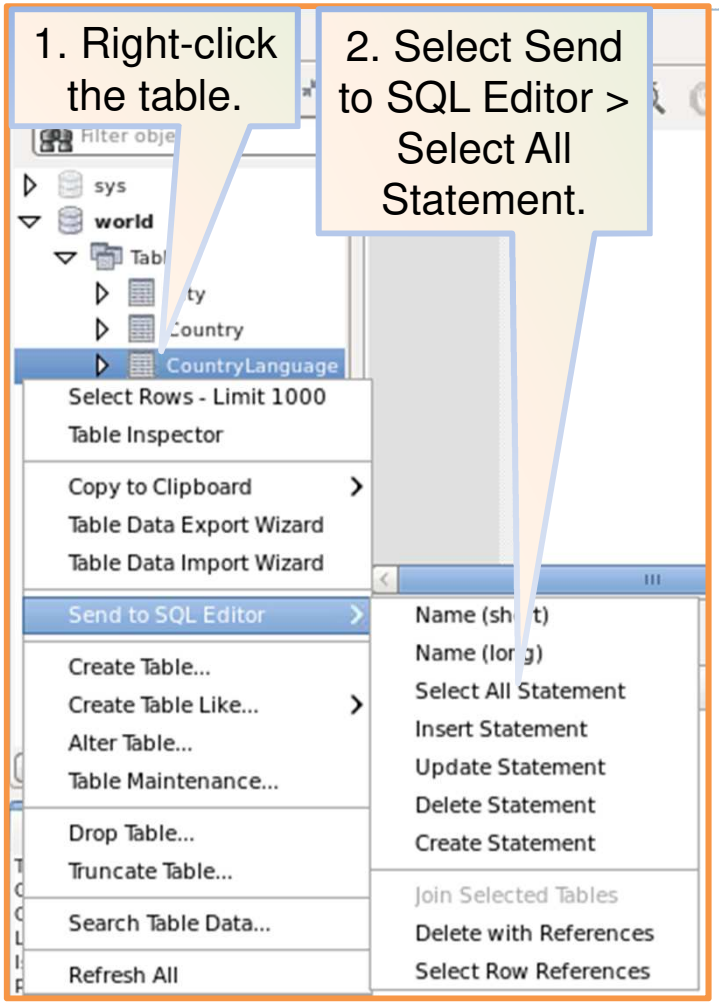
SELECT Statement Example in MySQL Workbench

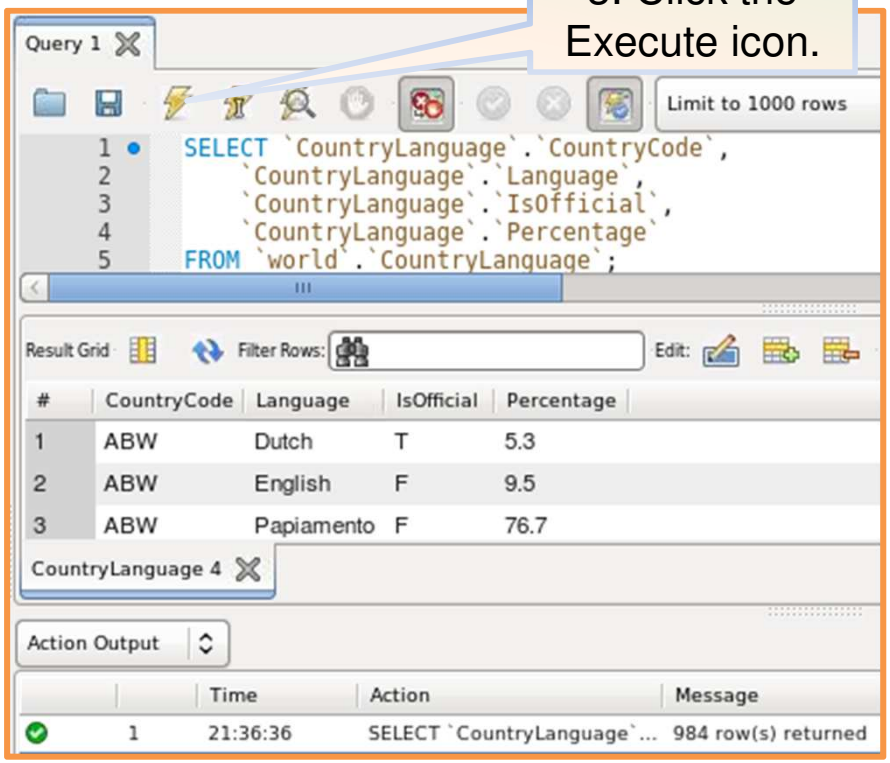


SELECT Statement Example with *



SELECT All Statement in MySQL Workbench





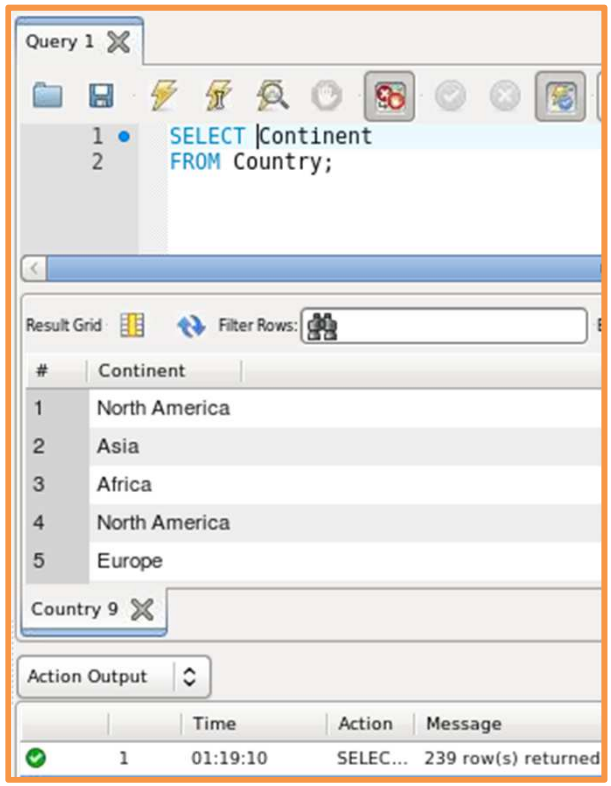
Using SELECT Clauses

- Use optional clauses alone (or in combination) to generate specific query results.
  - Types of clauses:
    - **`DISTINCT:`** Eliminates duplicate rows from the result set
    - **`FROM:`** Specifies which tables to retrieve data from
    - **`WHERE:`** Filters rows according to specified criteria
    - **`ORDER BY:`** Sorts rows in a specified order
    - **`LIMIT:`** Limits the maximum number of rows in the result set
  - The clauses must be used in the correct order:

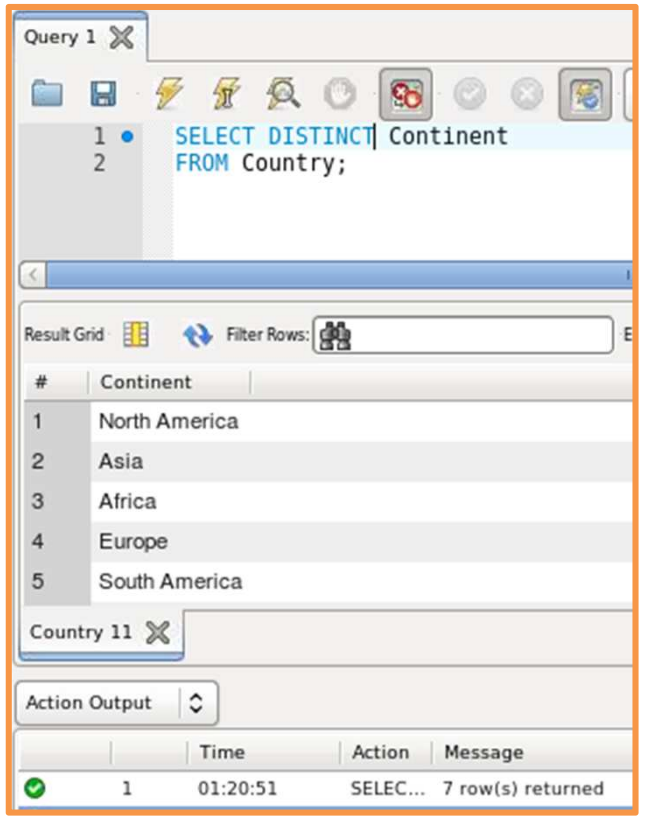
```
SELECT DISTINCT values_to_return
    FROM table_name
    WHERE condition
    ORDER BY how_to_sort
    LIMIT row_count;
```

Using SELECT with DISTINCT

- Removes duplicate rows, so every result set row is unique
- Difference between **`SELECT`** with and without **`DISTINCT`**:

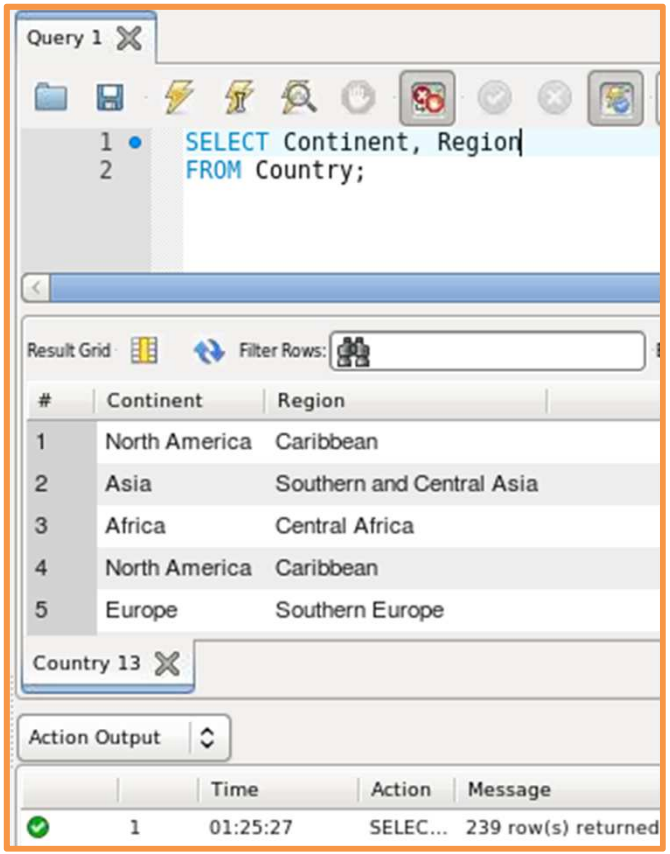


Compared to

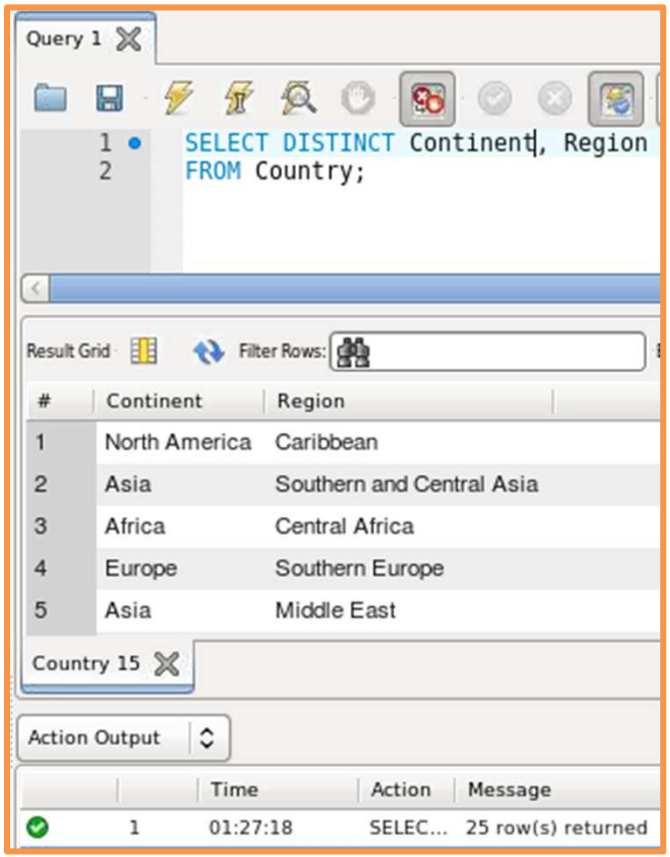


Using SELECT with DISTINCT for Multiple Columns

Example of the difference between `SELECT` without and with `DISTINCT` for multiple columns:

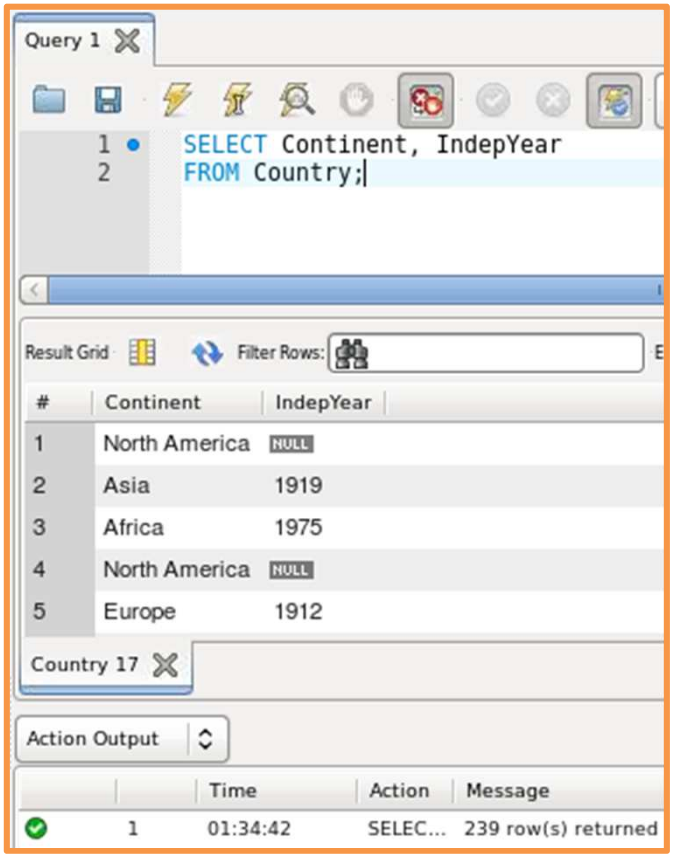


Compared to

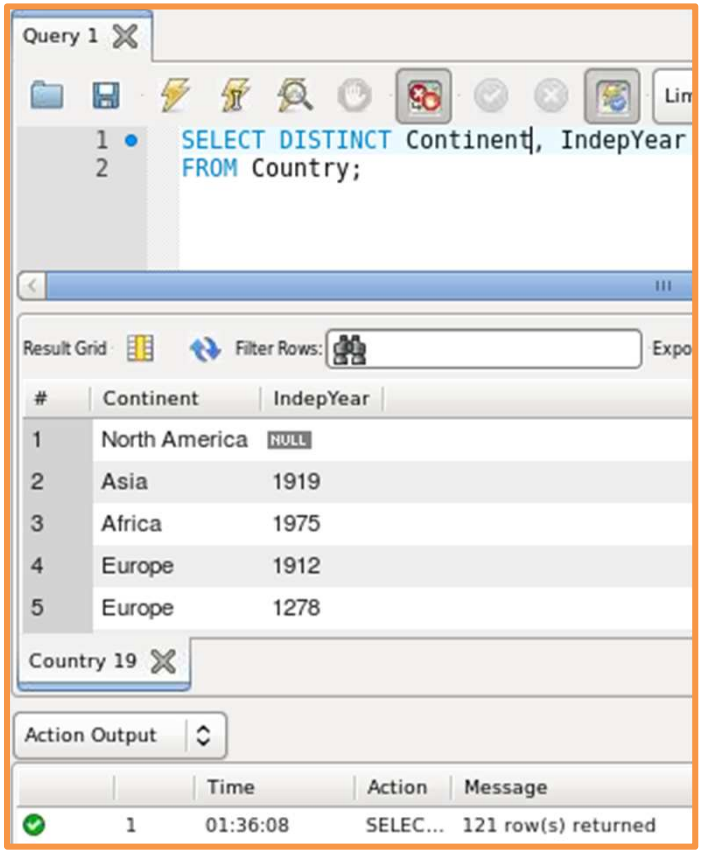


SELECT DISTINCT with NULL Values

- Treats all `NULL`s as the same value
- Retrieval of `NULL` values with and without `DISTINCT`:



Compared to



SELECT with WHERE

- Expressions in a **`WHERE`** clause can use the following types of operators:
  - Arithmetic: `+`, `-`, `*`, `/`, **`DIV`**, `%`, **`MOD`**
  - Comparison: `<`, `<=`, `=`, `<=>`, `<>` or `!=`, `>=`, `>`, **`BETWEEN`**
  - Logical: **`AND`**, **`OR`**, **`XOR`**, **`NOT`**
  - Additional options: **`IN`**, **`IS NULL`**, **`LIKE`**, **`()`**

Examples of Numeric Expressions with WHERE

- Examples of numeric comparisons:

```
•SELECT Name FROM Country WHERE IndepYear = 1919;
•SELECT Name, Population FROM Country
    •WHERE Population > 1000000000;
•SELECT Name, LifeExpectancy FROM Country
    •WHERE LifeExpectancy < 40;
```

Examples of Character Comparisons with WHERE

- Examples of character comparisons:

```
•SELECT Name FROM Country
    •WHERE GovernmentForm = 'Socialistic Republic';
•SELECT Name, Continent FROM Country
    •WHERE Continent != 'North America';
```

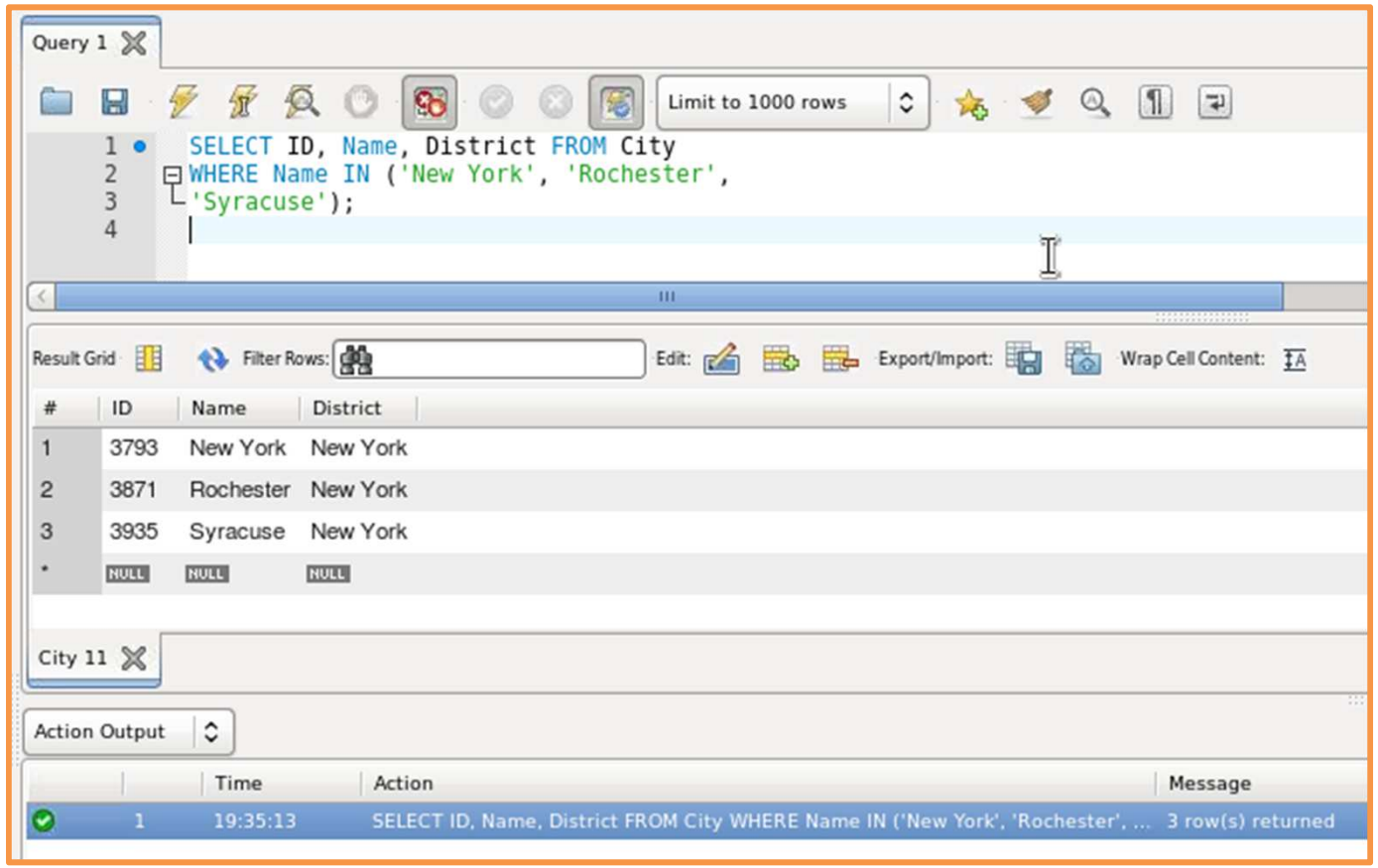
Examples of Character Comparisons by Using LIKE with WHERE

- Examples of character comparisons by using `LIKE`:

```
•SELECT Name, GovernmentForm FROM Country
    •WHERE GovernmentForm LIKE 'Social%';
•SELECT Name FROM Country
    •WHERE Name LIKE '%stan';
•SELECT Code, Name FROM Country
    •WHERE Code LIKE 'CO_';
```

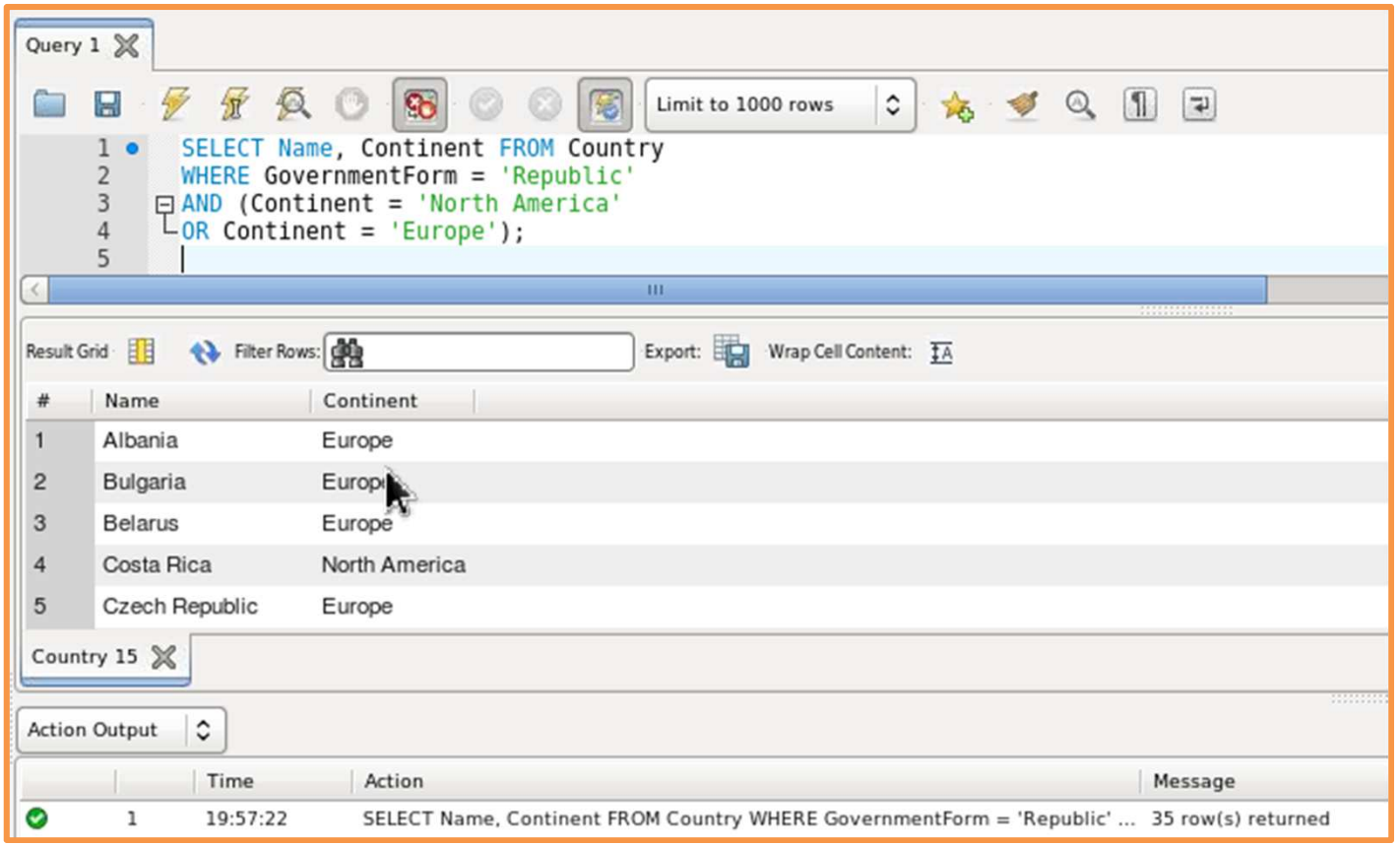
SELECT with WHERE … IN

Example of **`WHERE`** with **`IN`**:



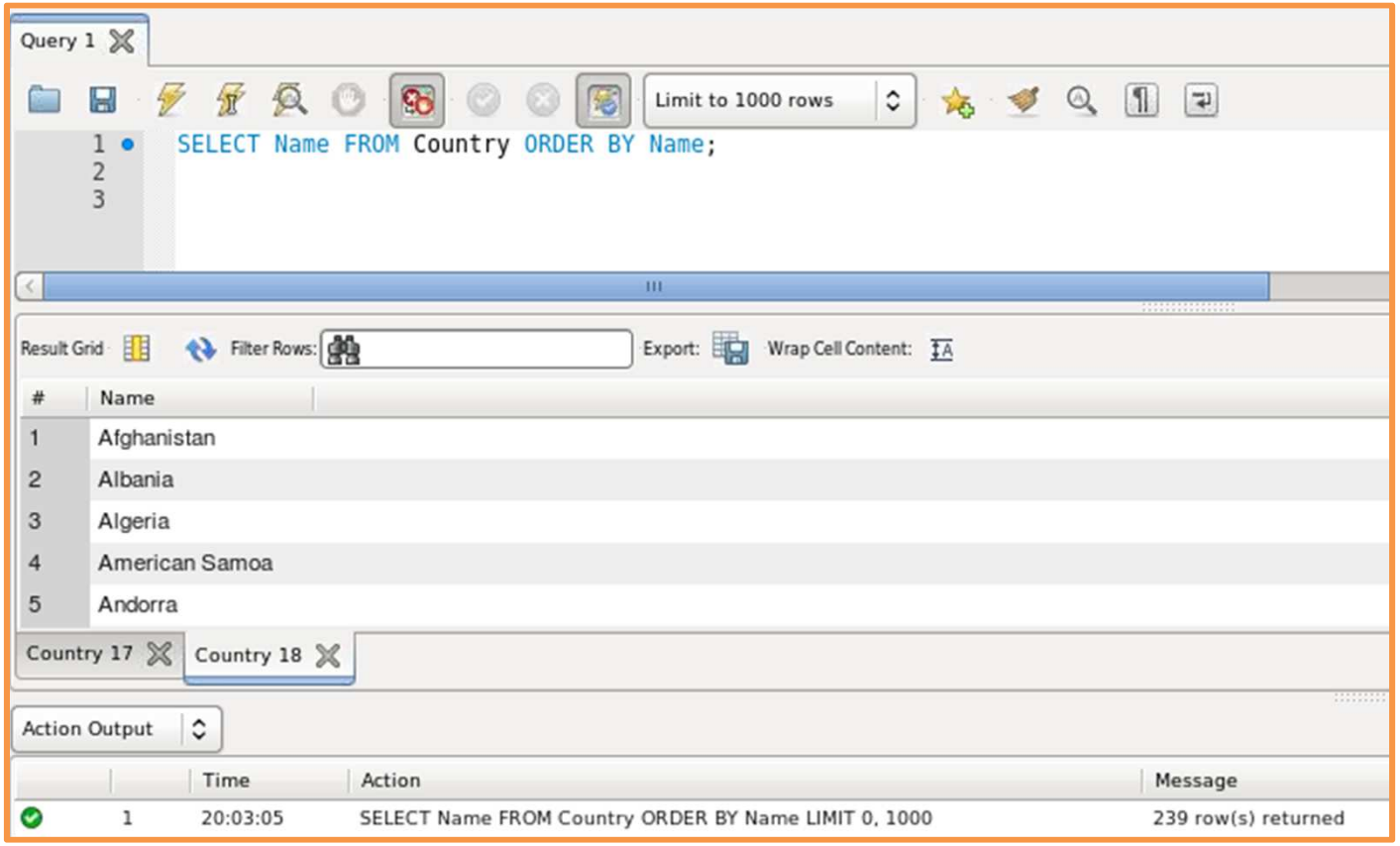
SELECT with WHERE … AND … OR

Example of **`WHERE`** with **`AND`** and **`OR`**:



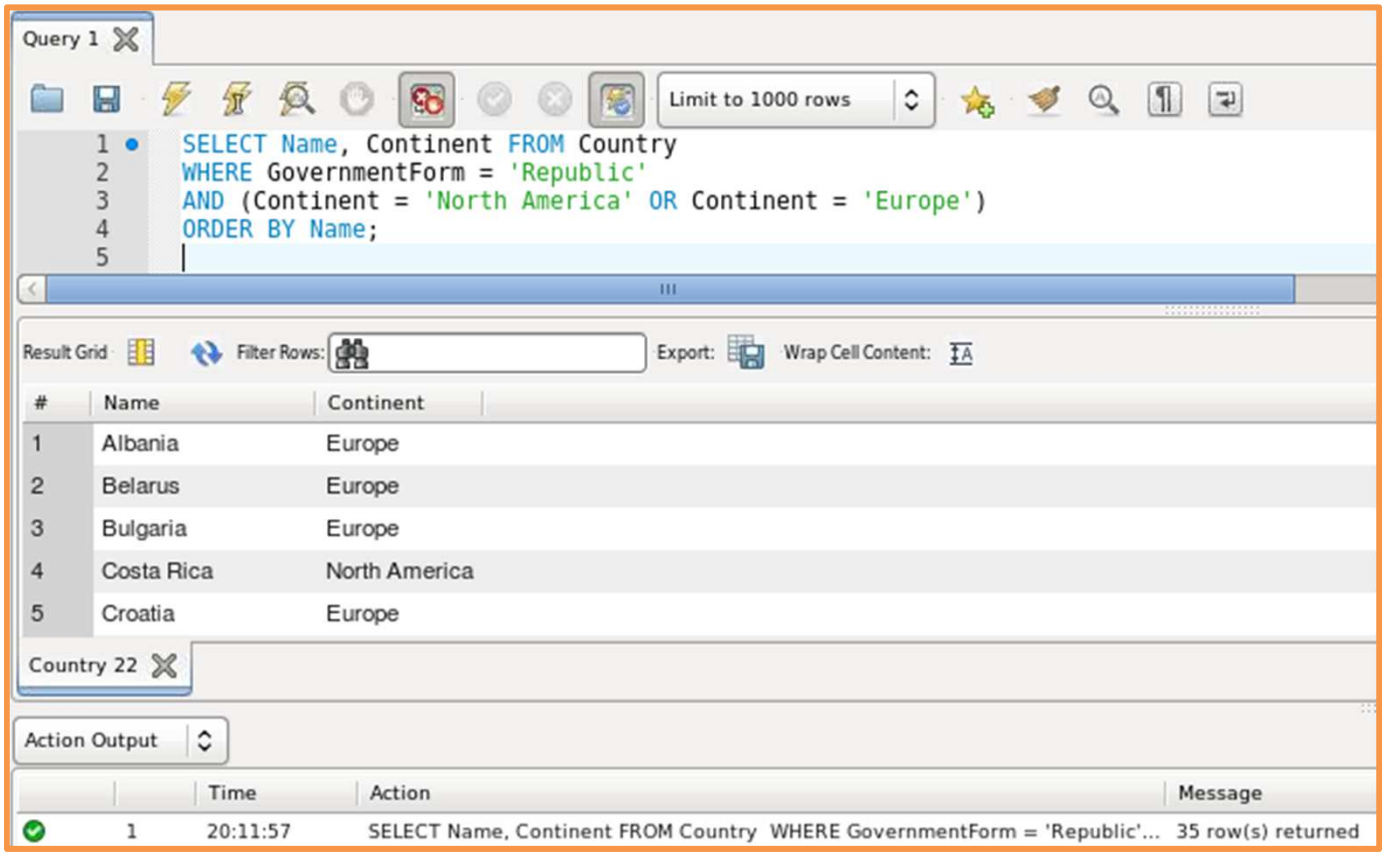
SELECT with ORDER BY

- Returns output rows in a specific order
- Example of **`ORDER BY`**:



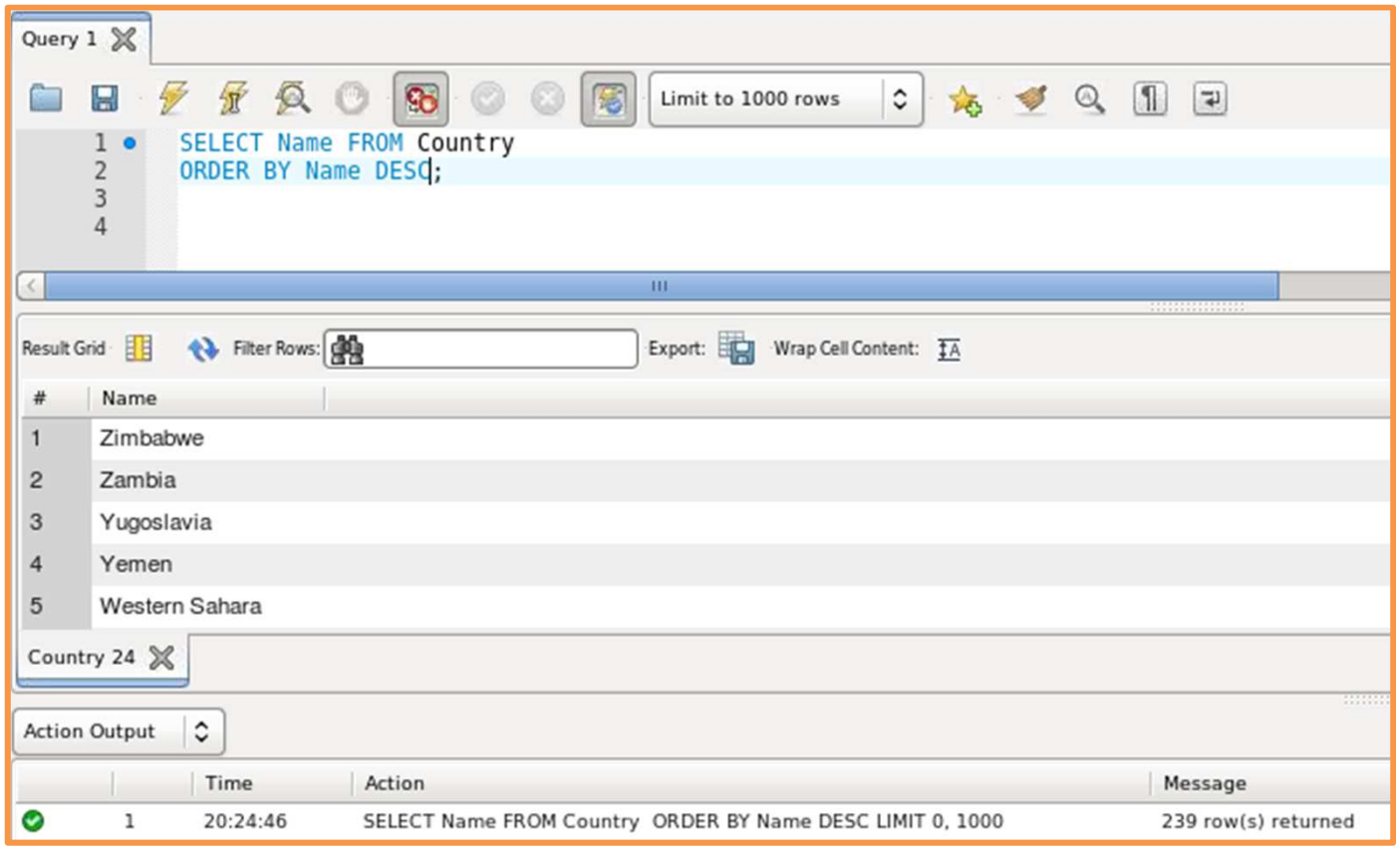
SELECT with ORDER BY: Another Example

The following example uses `ORDER BY` with a more complex `WHERE` clause:



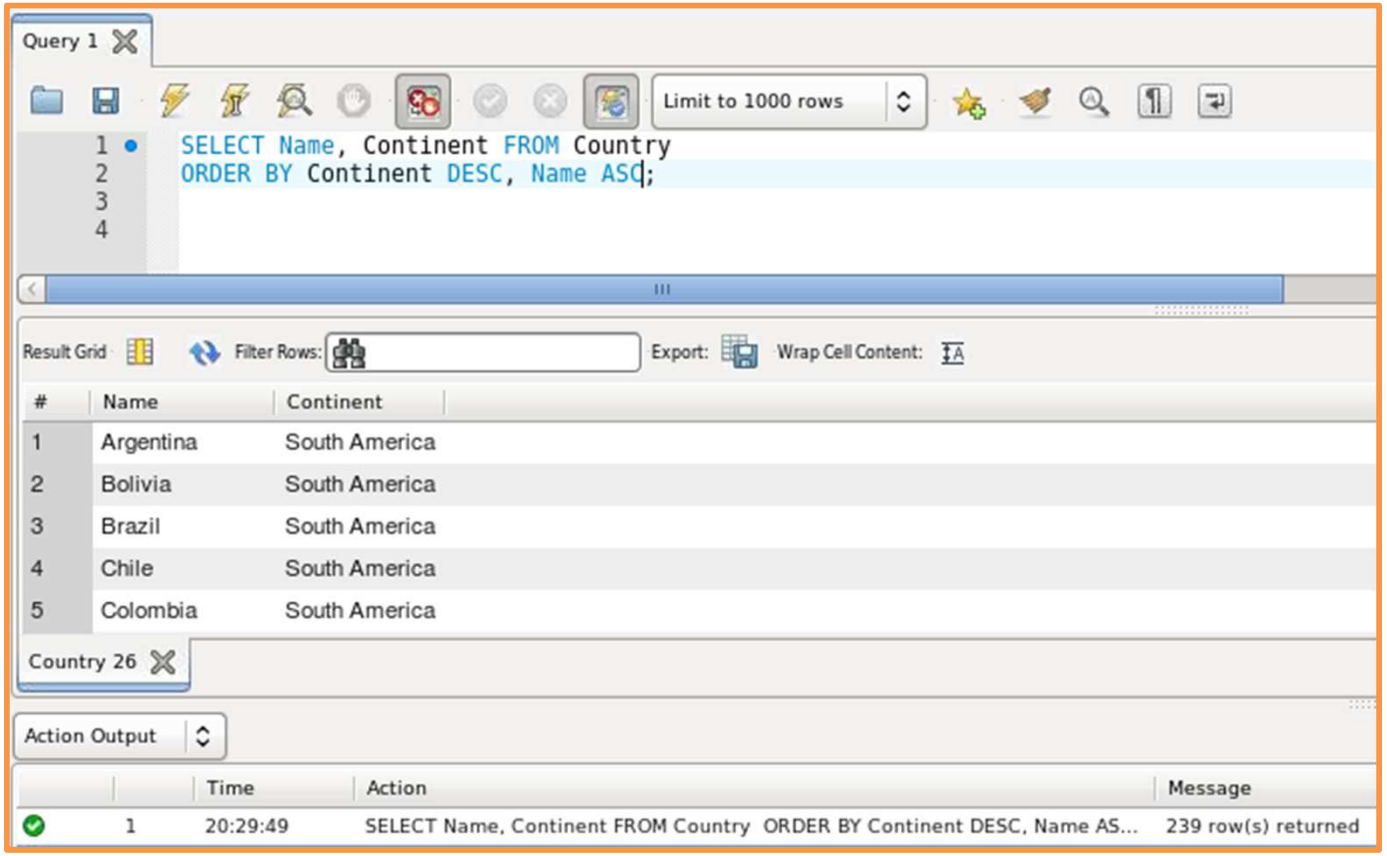
SELECT with ORDER BY with ASC and DESC

- **`ASC:`** Ascending order (default)
- **`DESC:`** Descending order



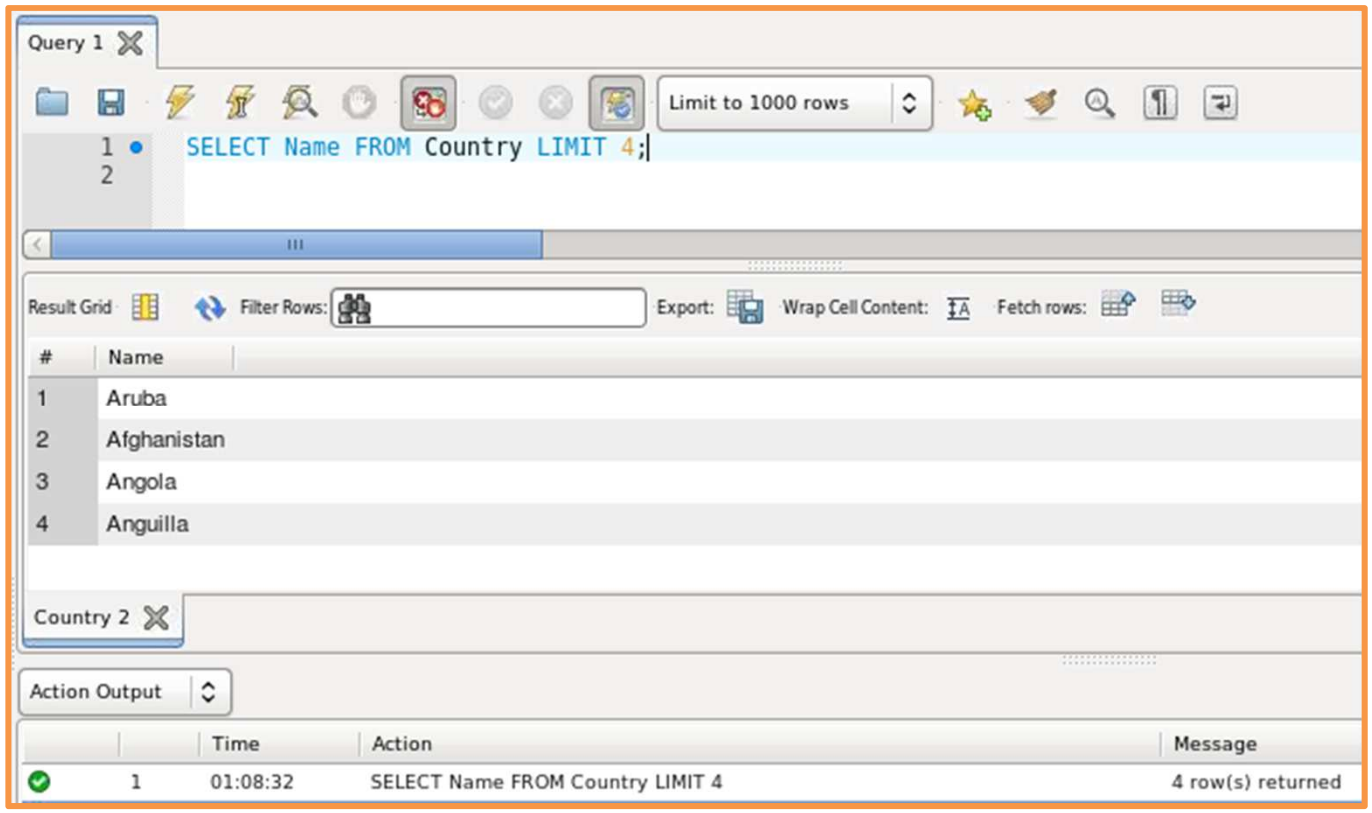
SELECT with ORDER BY with ASC and DESC: Multiple Columns

- You can sort multiple columns simultaneously.
- Example of **`ORDER BY`** with both **`ASC`** and **`DESC`**:



SELECT with LIMIT

- Specifies the number of rows to output in a result set
- Example of **`LIMIT`**:

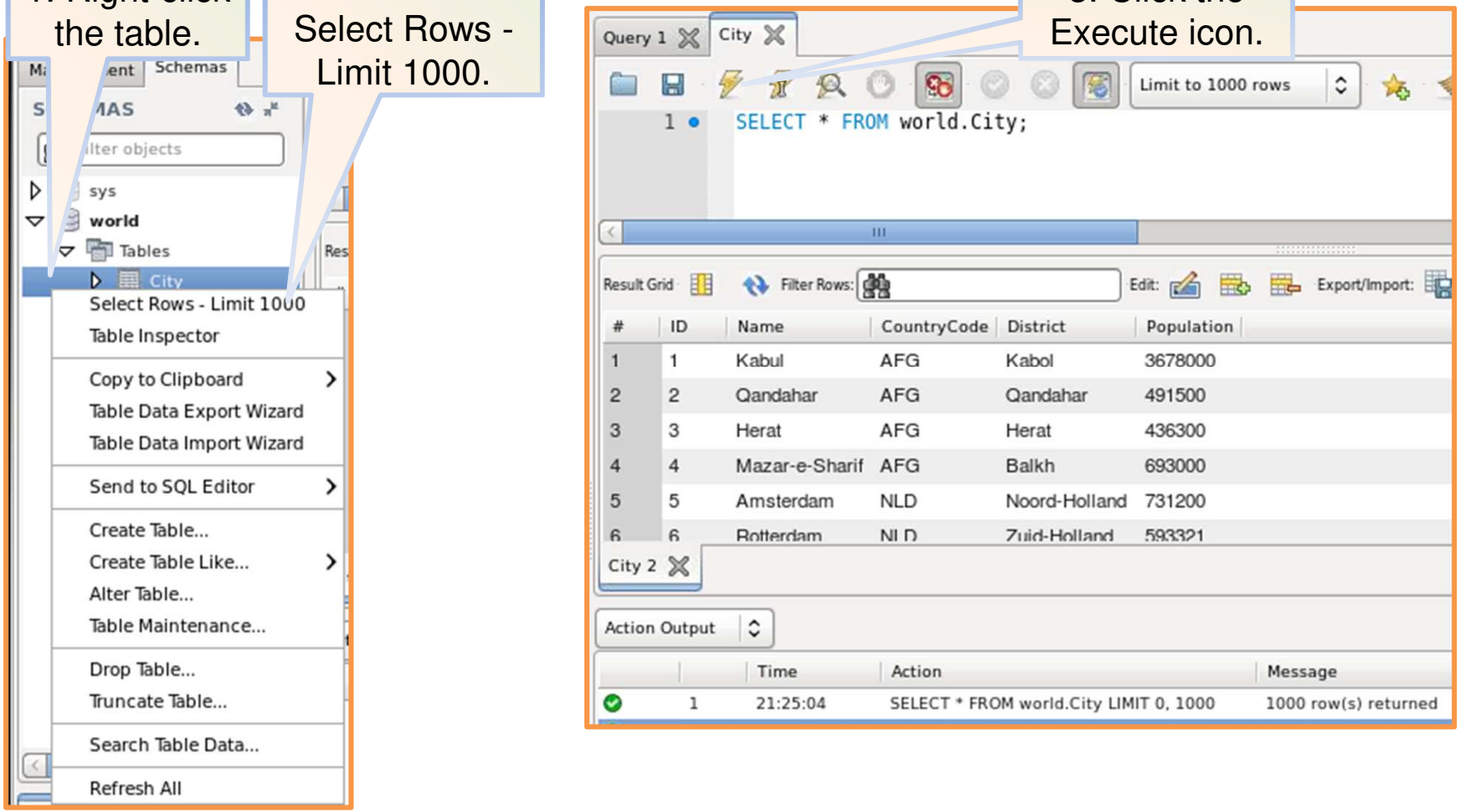


SELECT with LIMIT in MySQL Workbench

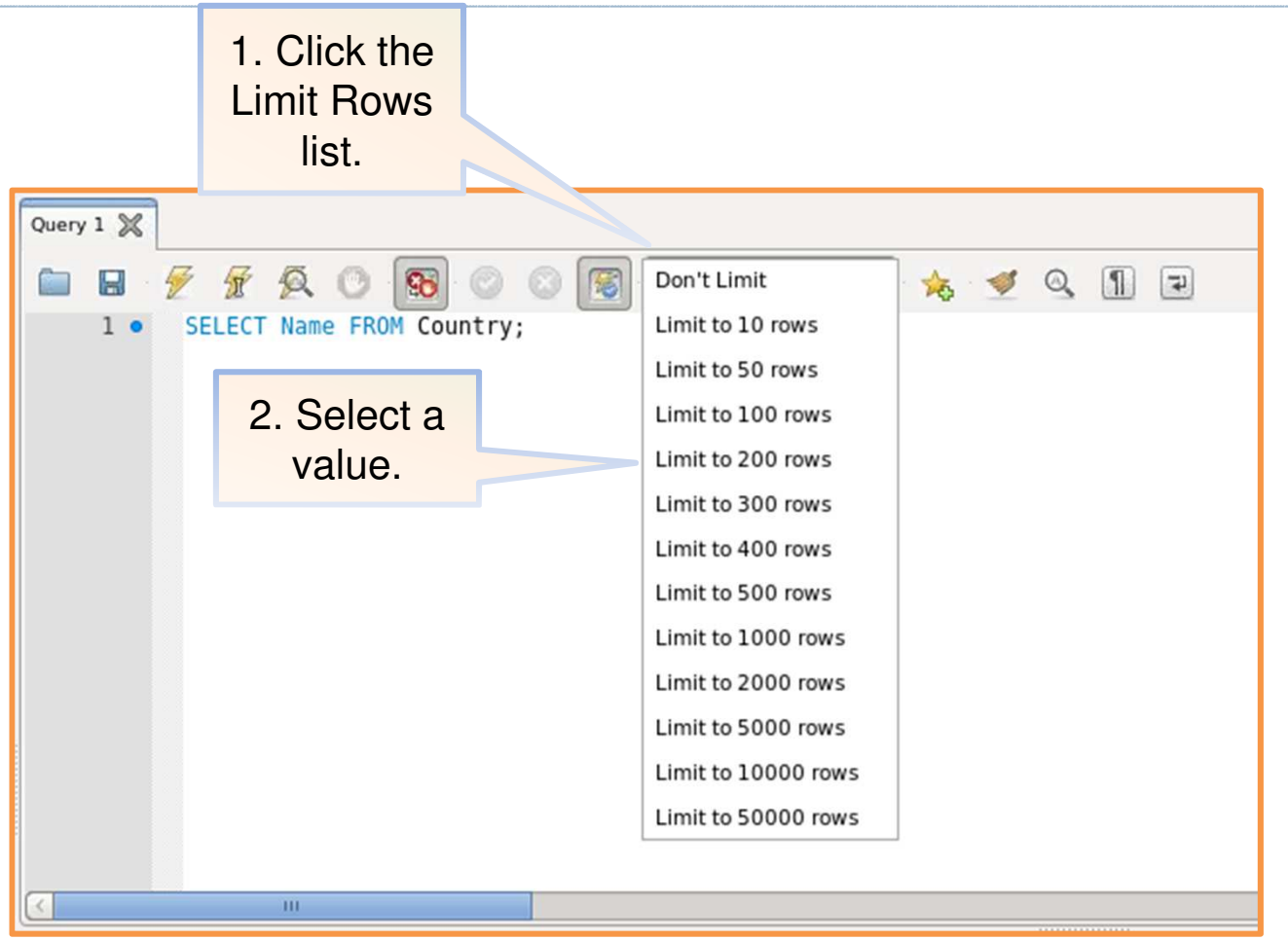
1. Right-click the table.

2. Select Select Rows - Limit 1000.

3. Click the Execute icon.

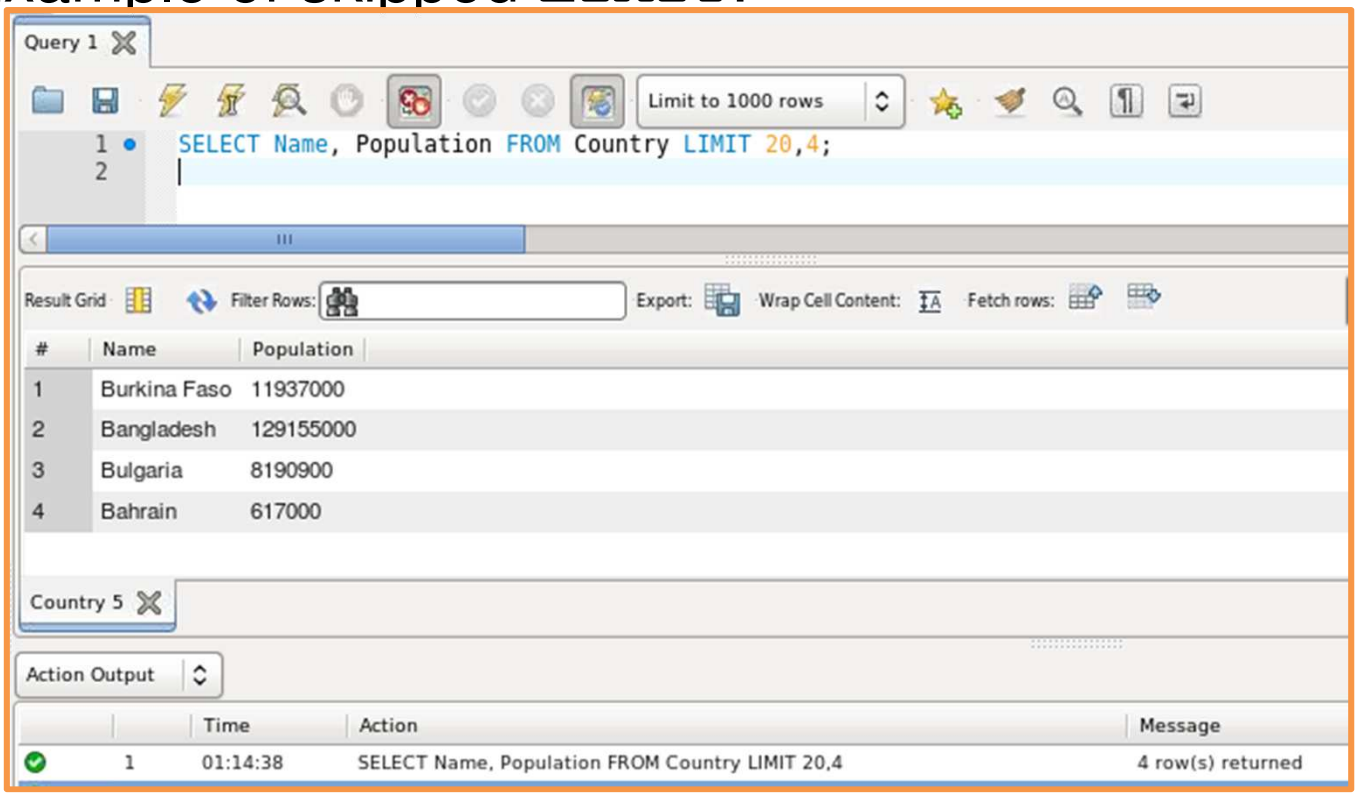


Setting the Limit for SELECT with LIMIT in MySQL Workbench



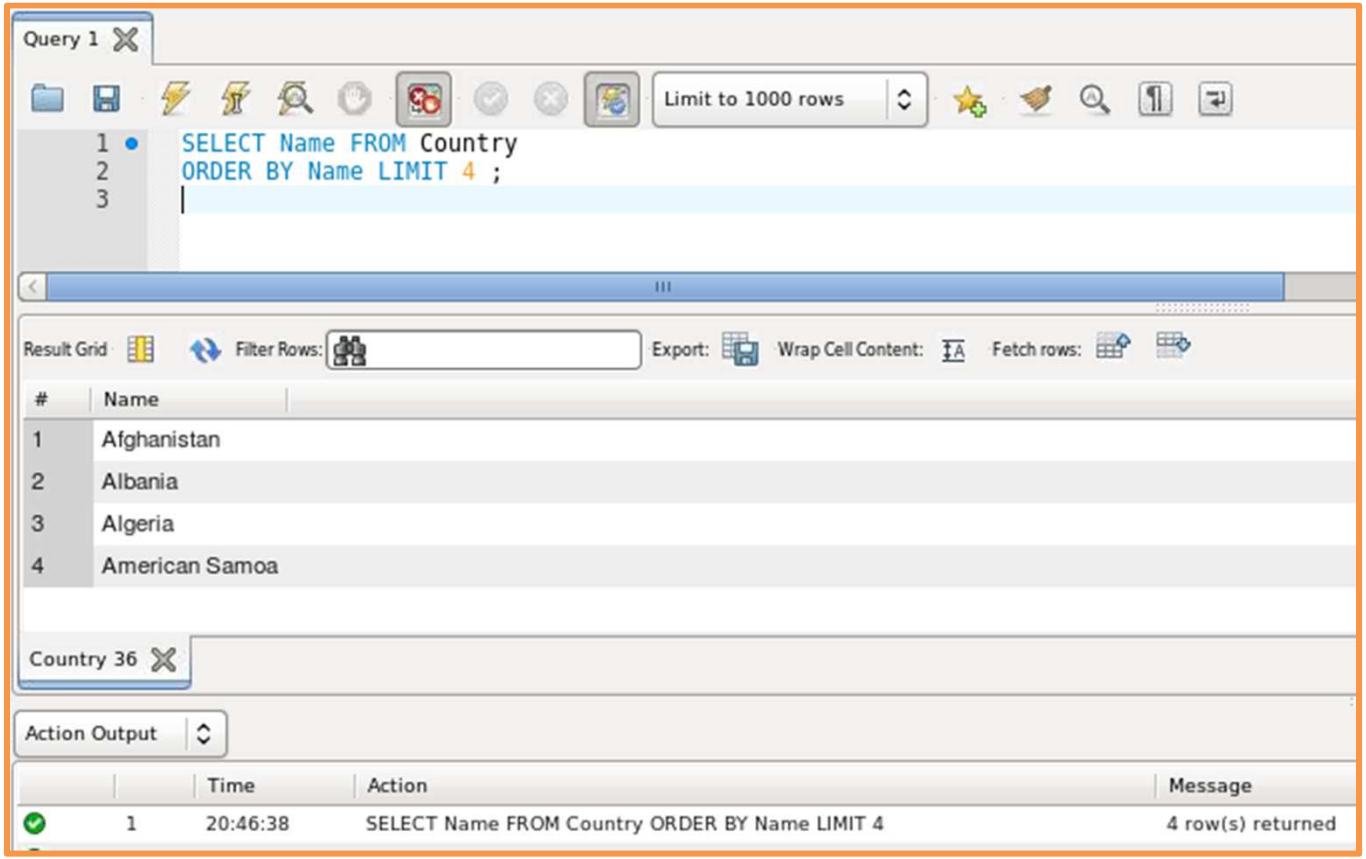
SELECT with LIMIT and Skip Count

- Specifies how many rows to skip before starting the result set
- Example of skipped **`LIMIT`**:



SELECT with LIMIT and ORDER BY

- Returns a specific number of rows in a specific order
- Example:



SELECT Usage Tips

- Treat table and database names as case-sensitive.
- From the `mysql` command-line client:
  - Use **`\c`** to abort a SQL statement
  - Use **`\G`** in place of **`;`** to see results vertically with each column in a row, instead of as a table, for example:

```
mysql> SELECT Name, Continent FROM Country\G
*************************** 1. row ***************************
     Name: Afghanistan
Continent: Asia
*************************** 2. row ***************************
     Name: Angola
Continent: Africa
...
```

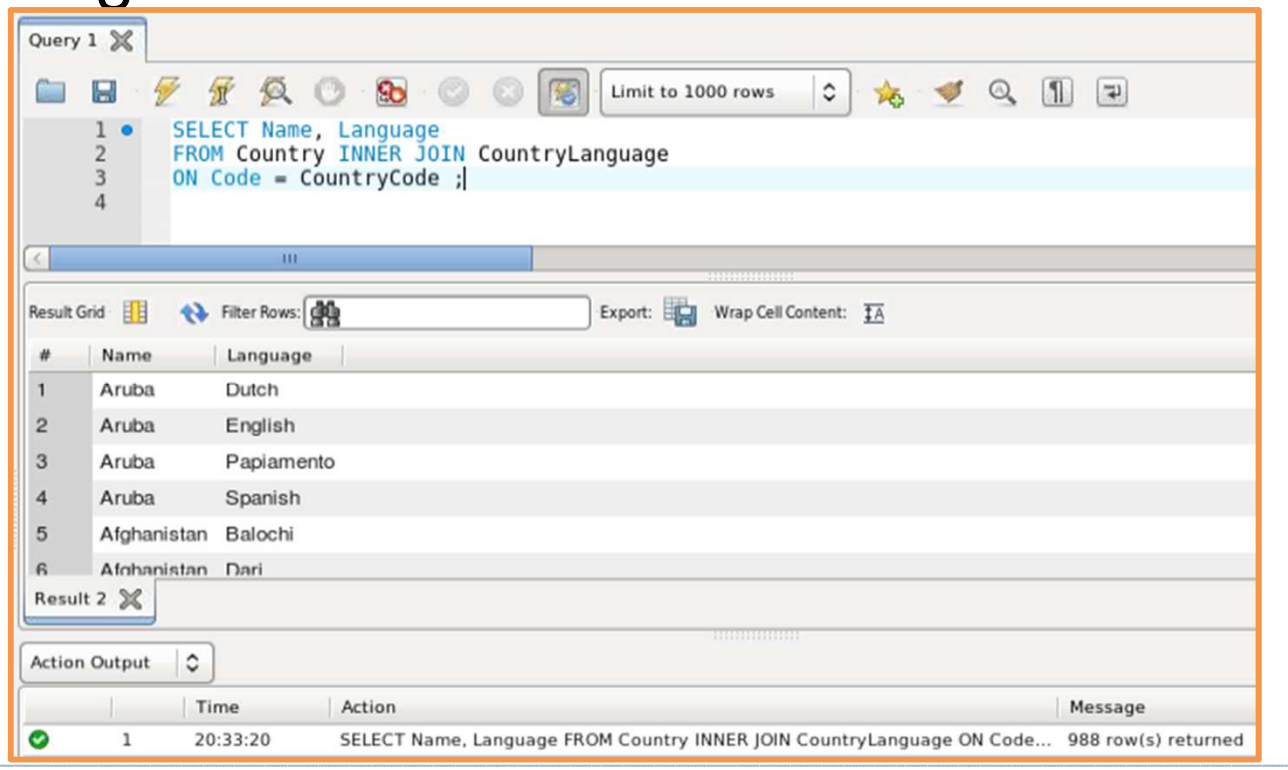
- Use **`*`** (all columns) with caution; it can waste resources and give unpredictable results due to changes in columns.

Querying Multiple Tables

- Some queries need data from multiple tables.
- You use a join operation to combine data from different tables.
- A join creates a temporary result set that contains combined data.
- To join tables, there must be a relationship between certain columns in these tables.

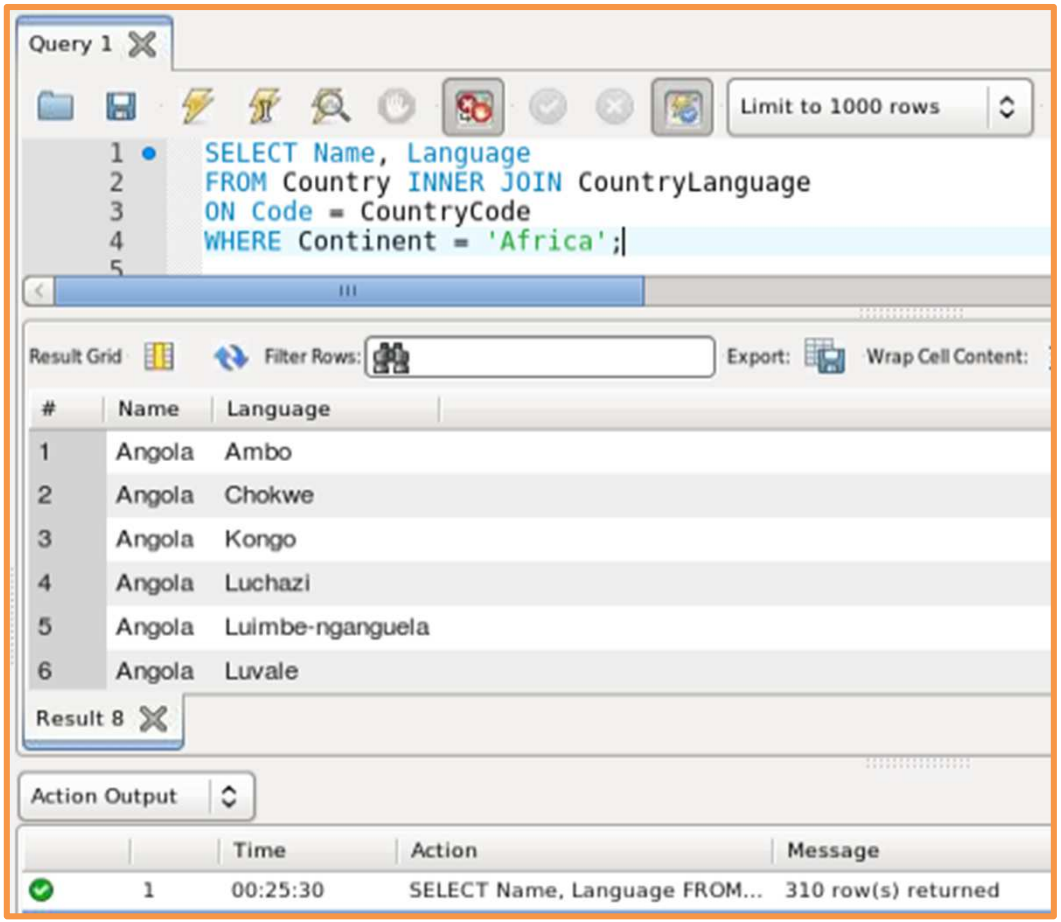
Combining Multiple Tables: INNER JOIN Keyword

- Join tables by including them in the `FROM` clause of the `SELECT` statement, separated by the `INNER JOIN` keyword. Indicate the relationship to use for joining the tables in the `ON` clause.



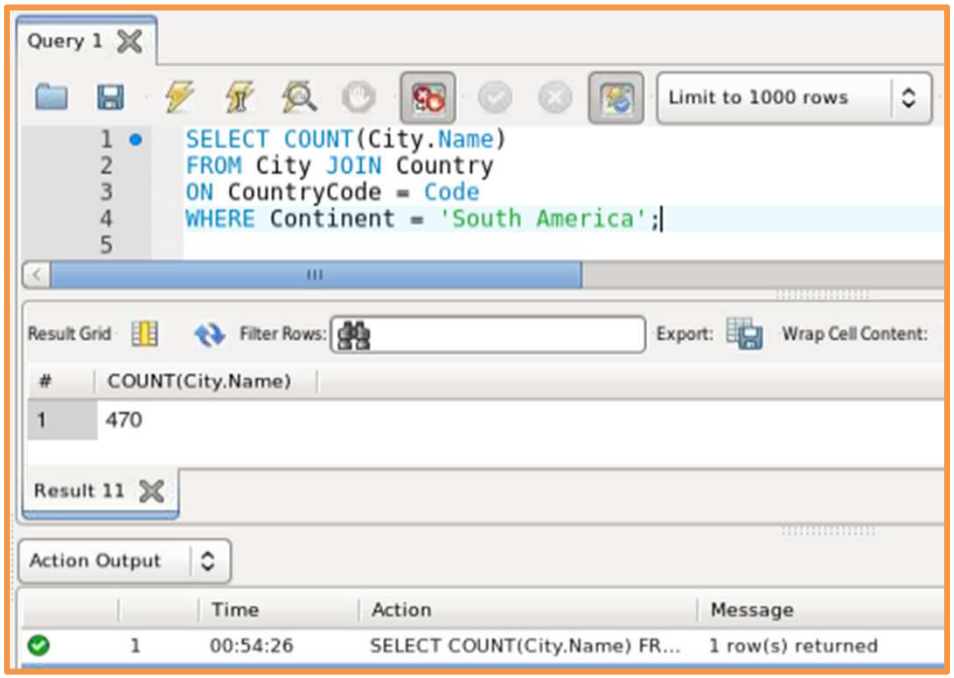
Limiting the Results of a Join by Using a WHERE Clause

- Include a `WHERE` clause to limit the results of an `INNER JOIN`.



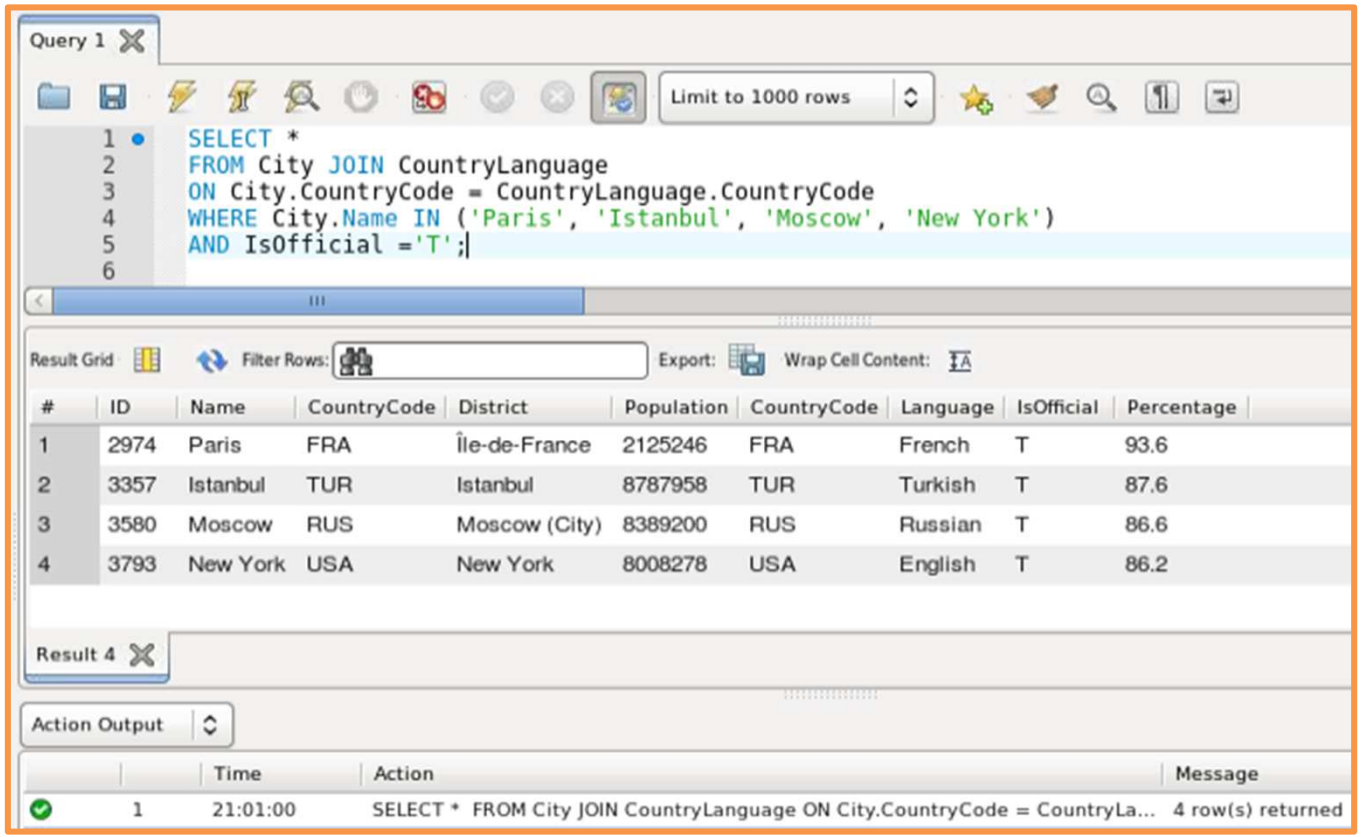
JOIN Keyword

- The `JOIN` keyword is equivalent to `INNER JOIN`. Use an `ON` clause to indicate the relationship for the join and a `WHERE` clause to limit results.



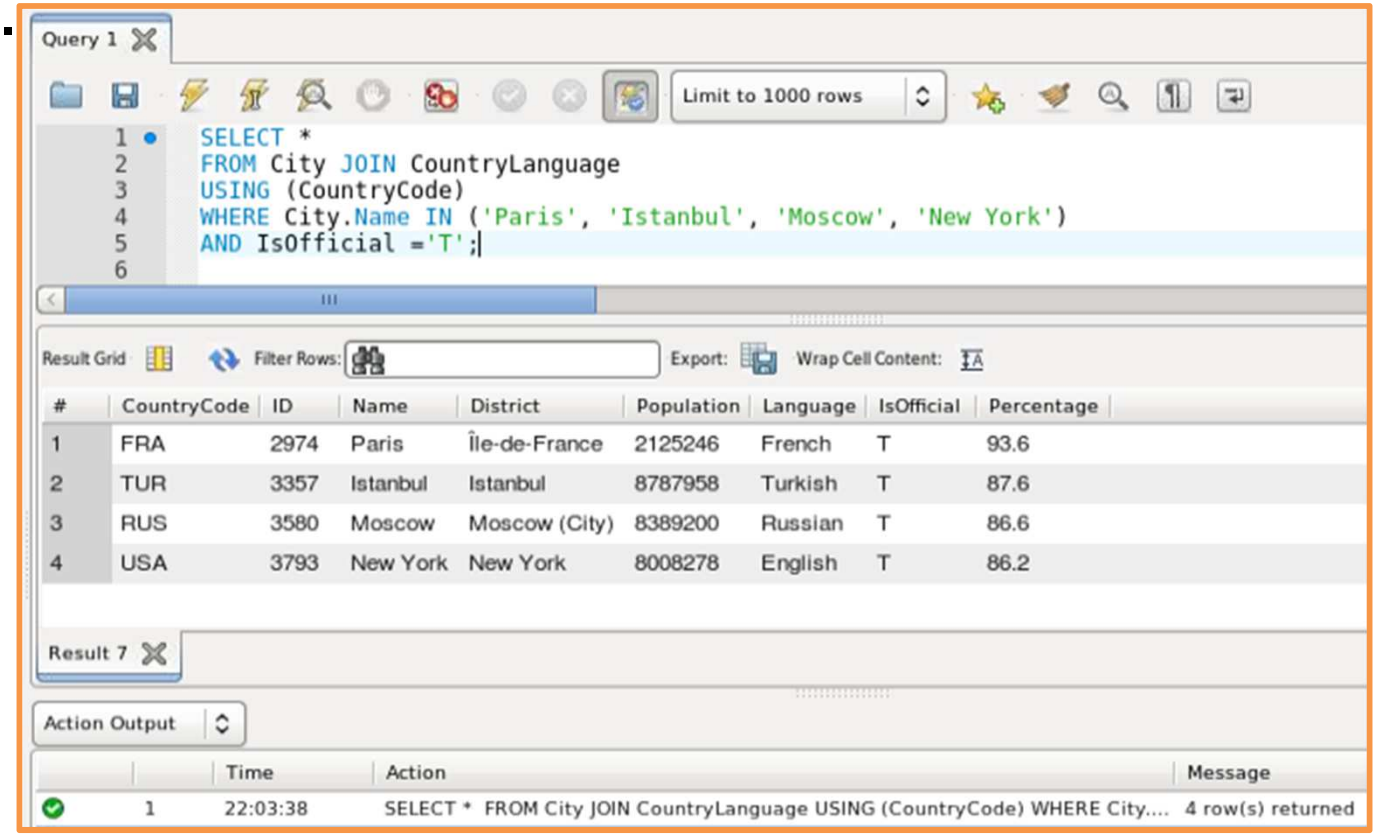
Joins: Columns That Are Not Related

- Columns from different tables can be matched for a join even if the columns are not related.



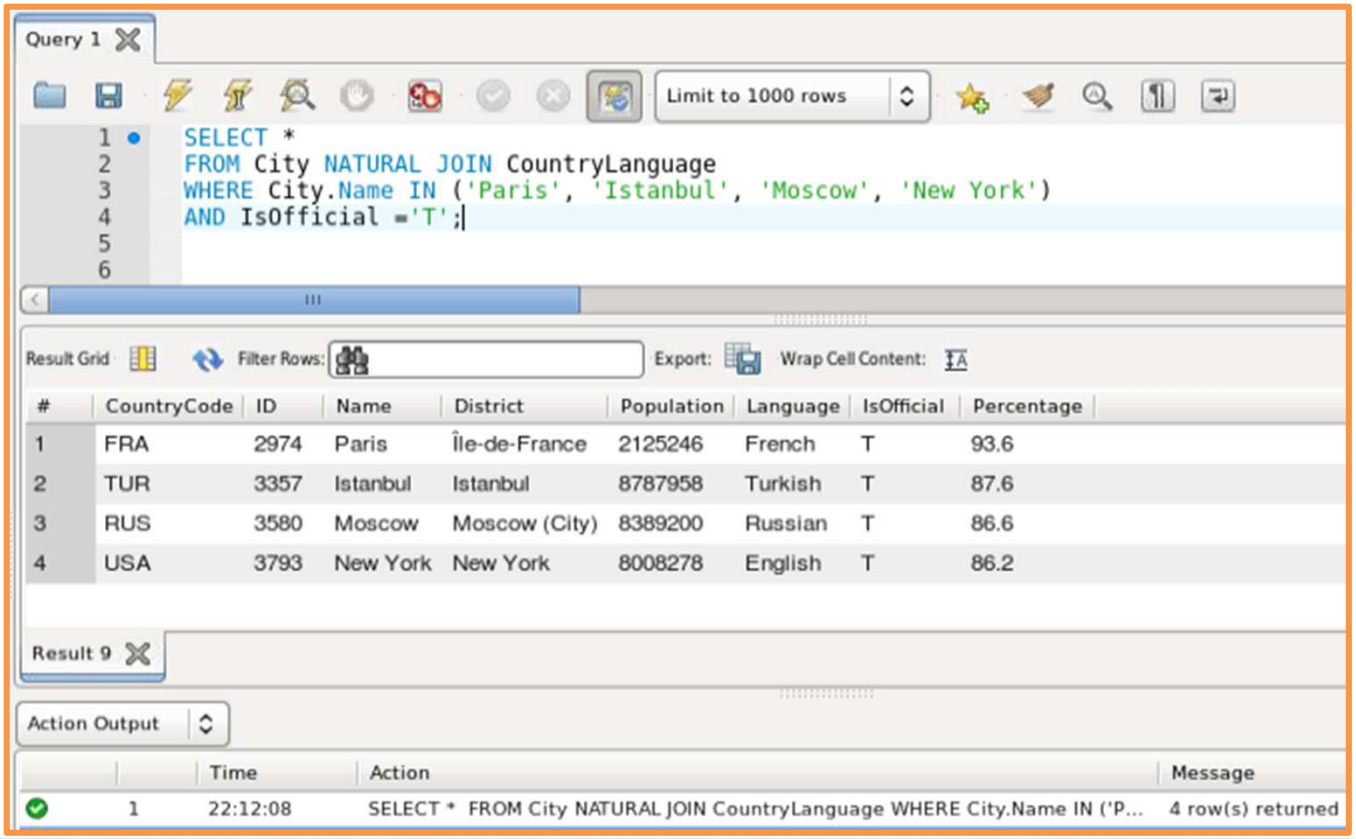
USING Keyword

- If the columns in the `ON` clause have the same names in both tables, you can use the `USING` keyword.



NATURAL JOIN Keyword

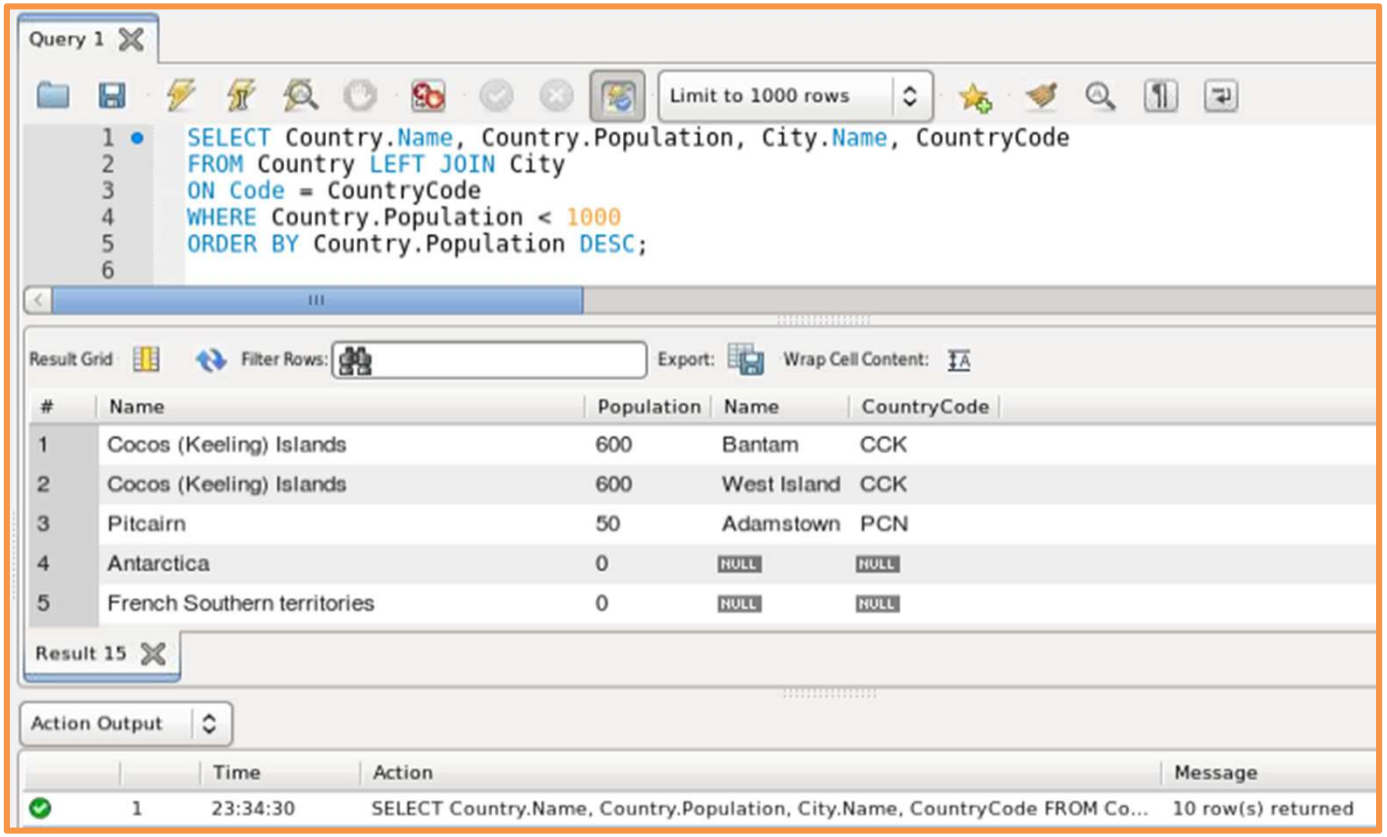
- The `NATURAL JOIN` keyword can replace a `USING` clause that lists all the column names in common between the tables.



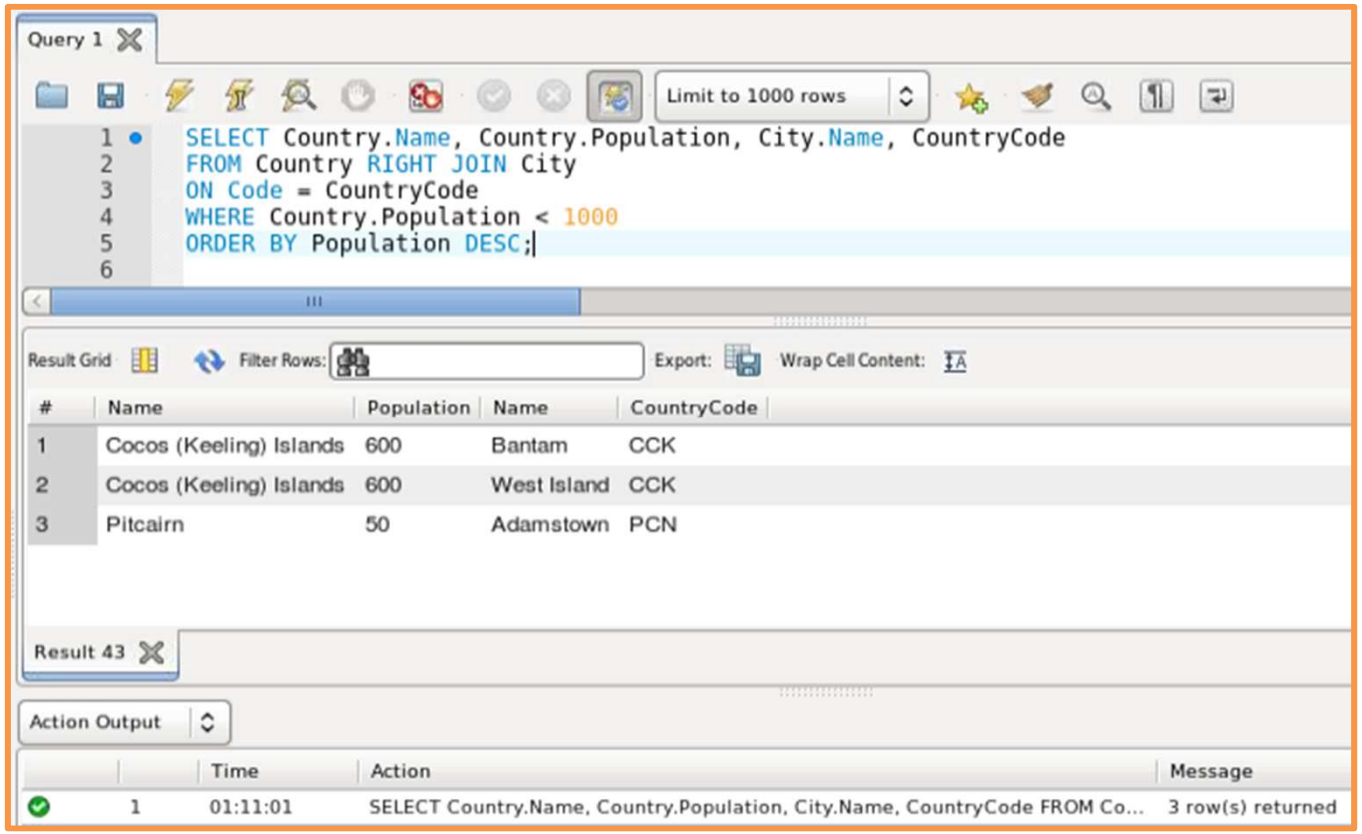
Outer Joins

- Outer joins include `LEFT OUTER JOIN` and `RIGHT OUTER JOIN`, or `LEFT JOIN` and `RIGHT JOIN`. The results of an outer join include all the rows from one table that meet the selection criteria in the `WHERE` clause, but only those rows from the other table that match with an `ON` or `USING` clause.
  - `LEFT JOIN` returns all rows from the left table but only matching rows from the right table.
  - `RIGHT JOIN` returns all rows from the right table but only matching rows from the left table.
  - The left table is listed first or to the left of the `LEFT JOIN` or `RIGHT JOIN` keyword. The right table is listed after or to the right of the keyword.

LEFT JOIN Example



RIGHT JOIN Example



Using OUTER JOIN with IS NULL

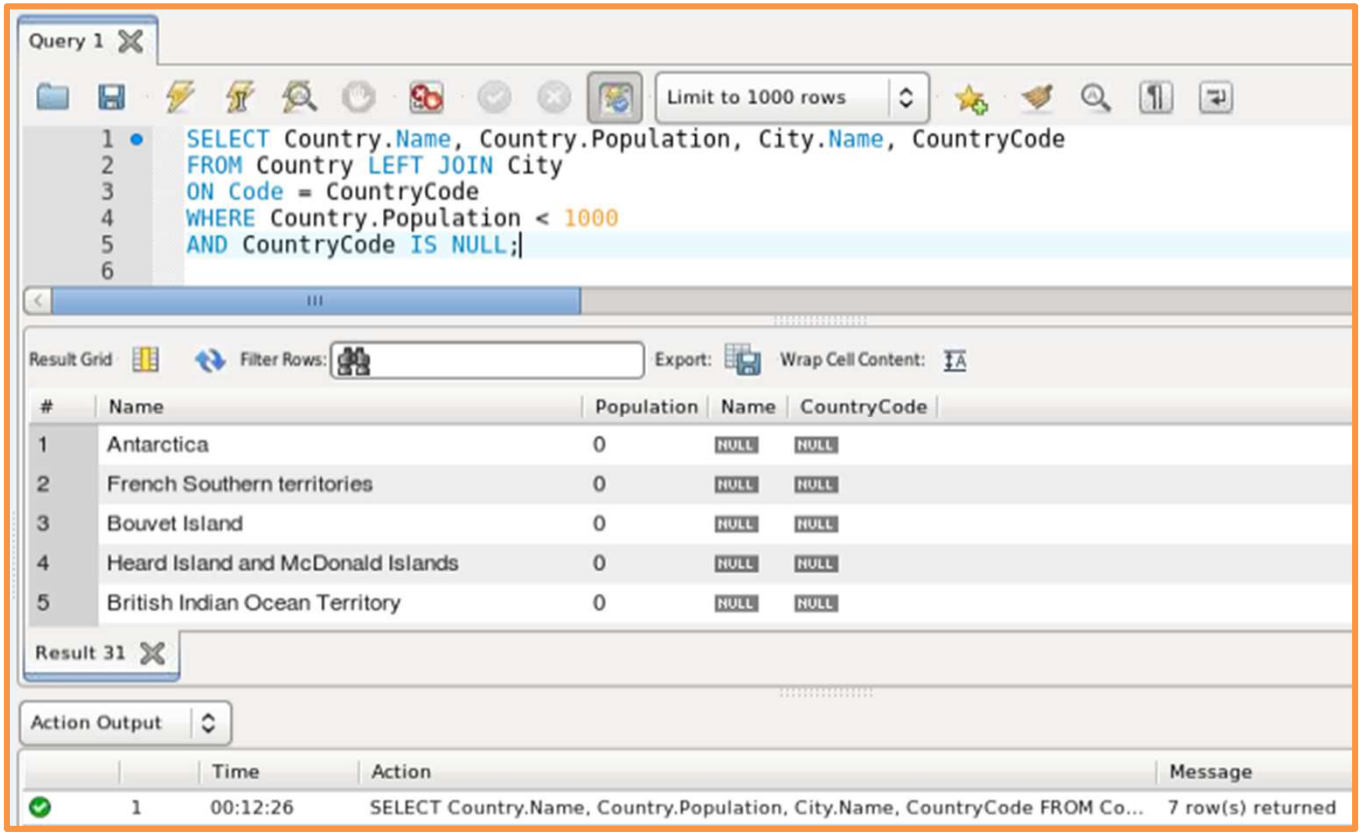
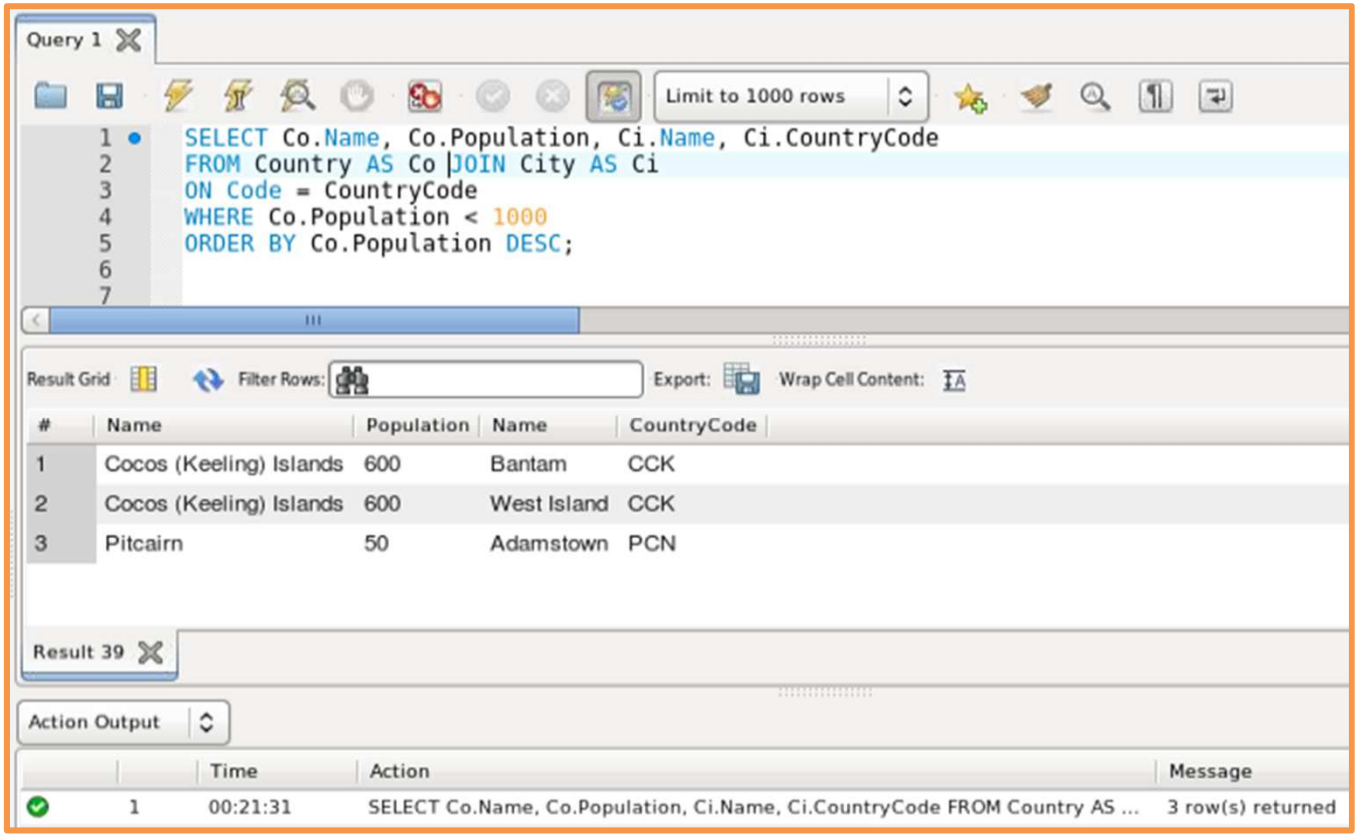


Table Name Aliases

- Table names can be aliased by using the `AS` keyword.



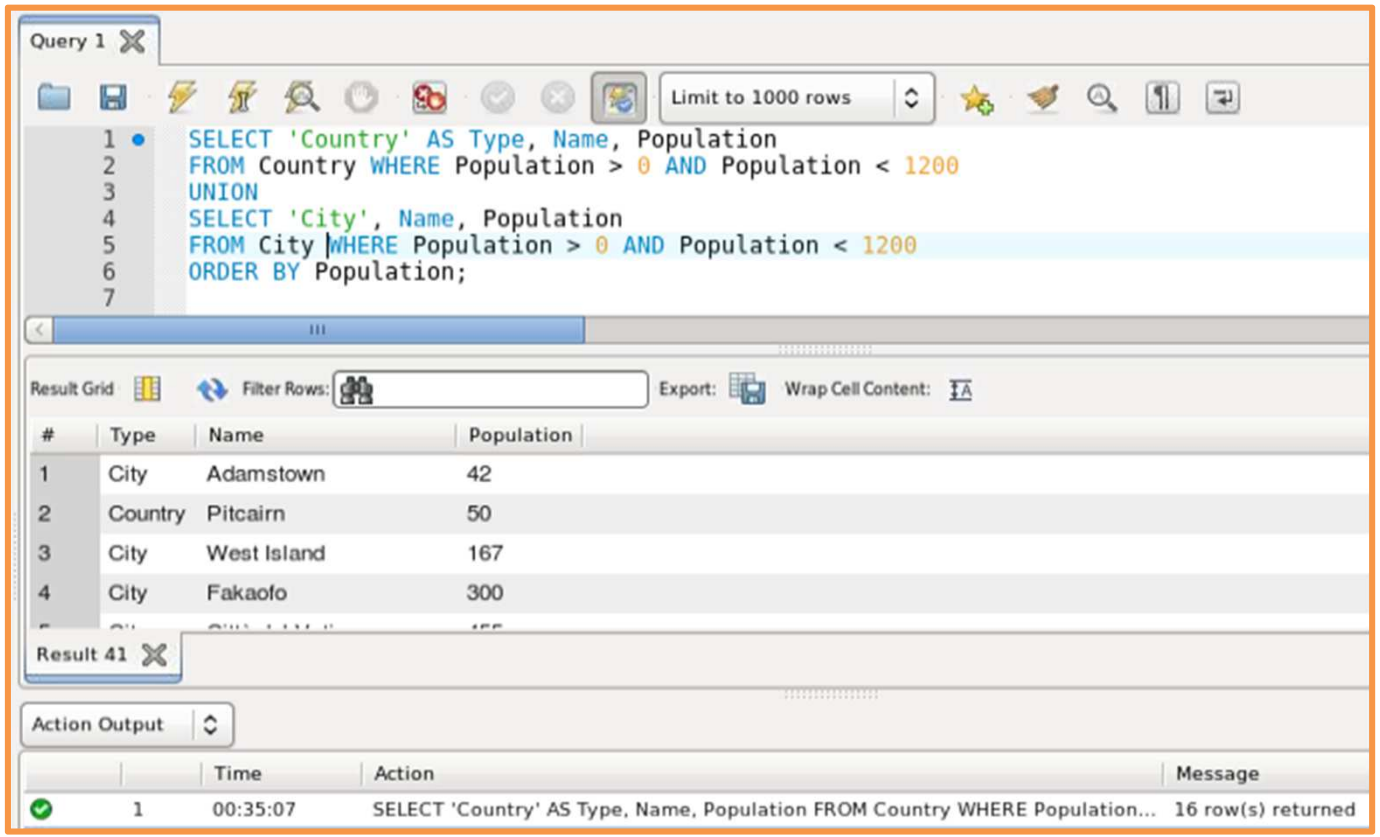
Unions

- Use the `UNION` keyword to combine results from multiple `SELECT` statements in a single result set.
- Syntax:

```
SELECT_statement
UNION [ALL | DISTINCT]
SELECT_statement
```

  - The number of columns in each *`SELECT_statement`* must be the same.
  - The columns in the same position for each *`SELECT_statement`* must be compatible data types.
  - Default is `UNION DISTINCT`, which removes duplicate rows.
  - Use `UNION ALL` to retain duplicate rows.

Union Example



Subqueries

Subqueries

- A *subquery* is a query that appears between parentheses as part of another SQL statement.
  - From the perspective of the query that contains the subquery, the subquery is sometimes referred to as a *nested query*.
  - From the perspective of the subquery itself, a query that (directly or indirectly) contains the subquery can be referred to as an *outer query*, a *containing query,* or an *enclosing query*.
  - An expression containing a subquery can be referred to as the *outer expression*, *containing expression,* or *enclosing expression*.

Subquery: Example

The following query uses a subquery to list the languages spoken in Finland.


```
SELECT Language FROM CountryLanguage WHERE CountryCode = 'FIN';
```

Subquery Categories

- Subqueries are categorized according to the type of results they return to the containing expression.
  - **Scalar subqueries:** The subquery is treated as a single *value*.
  - **Row subqueries:** The subquery is treated as a single *row*.
  - **Table subqueries:** The subquery is treated as a read-only *table*.

Scalar Subqueries

- A *scalar subquery* evaluates to a single-value expression (a single row containing a single column).
  - This applies even if the subquery fails to retrieve a row.
    - If the row is retrieved, its column value is treated as a scalar value.
    - If no row is retrieved, the subquery evaluates to `NULL`.
  - You can use the value returned from a scalar subquery in the outer expression just like any other value.
  - If a scalar subquery returns more than one row or

```
mysql> SELECT 'Finland' = (SELECT Name FROM Country);
ERROR 1242 (21000): Subquery returns more than 1 row

mysql> SELECT 'Fin' = (SELECT * FROM Country);
ERROR 1241 (21000): Operand should contain 1 column(s)
```

Example 1: Scalar Subquery

Scalar queries are useful for aggregating data. For example:

- List each country and its population as a

```
mysql> SELECT Country.Name,
    -> 100 * Country.Population /
    -> (SELECT SUM(Population) FROM Country)
    -> AS pct_of_world_pop
    -> FROM Country;
+-------------------------------------------+------------------+
| Name                                      | pct_of_world_pop |
+-------------------------------------------+------------------+
| Aruba                                     |           0.0017 |
| Afghanistan                               |           0.3738 |
| Angola                                    |           0.2119 |
| Anguilla                                  |           0.0001 |
| Albania                                   |           0.0560 |
| Andorra                                   |           0.0013 |
...
239 rows in set (#.## sec)
```

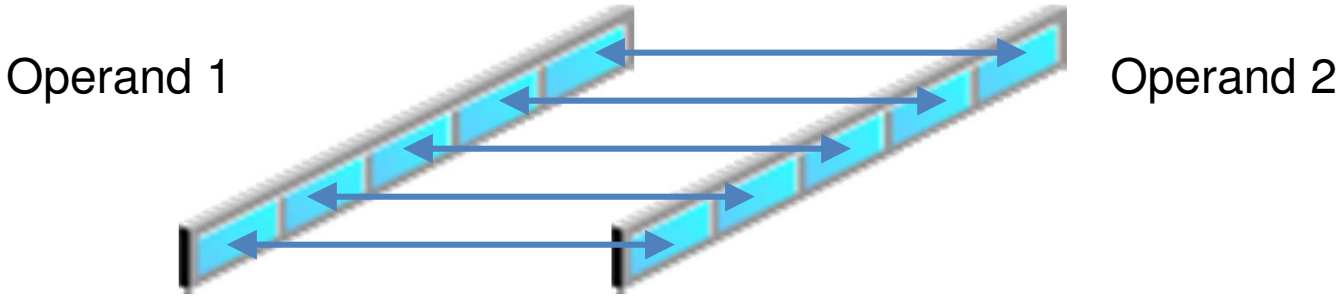
Example 2: Scalar Subquery

- List the number of cities and the number of languages for each country:

```
mysql> SELECT Name,
    -> (SELECT COUNT(*) FROM City
    ->   WHERE CountryCode = Code) AS Cities,
    -> (SELECT COUNT(*) FROM CountryLanguage
    ->   WHERE CountryCode = Code) AS Languages
    -> FROM Country;
+----------------------+--------+-----------+
| Name                 | Cities | Languages |
+----------------------+--------+-----------+
| Aruba                |      1 |         4 |
| Afghanistan          |      4 |         5 |
| Angola               |      5 |         9 |
| Anguilla             |      2 |         1 |
| Albania              |      1 |         3 |
| Andorra              |      1 |         4 |
| Netherlands Antilles |      1 |         3 |
| United Arab Emirates |      5 |         2 |
...
239 rows in set (#.## sec)
```

Row Subqueries

- *Row subqueries* return a single row containing at least two columns.
  - If the subquery does not retrieve any rows, the expression in which you use it is `FALSE`.
- You can use row subqueries (and other row constructors) as operands for the comparison operators only if both operands are row constructors with the same number of columns.
  - MySQL evaluates these operators by performing a pairwise comparison of the individual (scalar) column values.



Row Subquery Comparison

- For the *equality* operators (= and <=>):
  - *All* pairwise comparisons need to evaluate to true in order for the expression to be true.
    - Two rows are equal only if all corresponding column values are equal.
- For the *inequality* operators (!= and <>):
  - *Only one* pairwise comparison must be true in order for the expression to be true.
    - If even one column pair is not equal, the rows are not equal.
- For the *comparison* operators (>, <, >= and <=):
  - The column *values* and the *order* of the columns is significant.
    - For example, `(SELECT 2, 1)` is larger than `(SELECT 1, 100)`

Row Subqueries: Examples

- In the following example, the subquery is equal to the row constructor literal `('London', 'GBR')`:

```
mysql> SELECT ('London', 'GBR') =
    -> (SELECT Name, CountryCode FROM City
    ->   WHERE ID = 456) AS IsLondon;
+----------+
| IsLondon |
+----------+
| 1        |
+----------+
```

- The following example compares two subqueries for

```
mysql> SELECT (SELECT ID, Name, CountryCode FROM City WHERE ID = 456) =
    -> (SELECT ID, Name, CountryCode FROM City WHERE CountryCode = 'GBR'
    ->    AND Name = 'London') AS IsEqual;
+----------+
| IsEqual  |
+----------+
| 1        |
+----------+
```

Invalid Row Subqueries

- You get a runtime error if:
  - A row subquery retrieves more than one row:

```
mysql> SELECT (null, null) <=>
    ->  (SELECT GNP, Capital FROM Country LIMIT 2);
ERROR 1242 (21000): Subquery returns more than 1 row
```

    - The subquery returns two rows.
  - You attempt to compare two row constructors with a different number of columns:

```
mysql> SELECT (null, null, null) <=>
    ->  (SELECT GNP, Capital FROM Country LIMIT 0);
ERROR 1241 (21000): Operand should contain 3 column(s)
```

    - The row constructor `(null, null, null)` contains two columns.
    - The row subquery returns a row with three columns.

Table Subqueries

- Return a result set containing zero or more rows with one or more columns
  - The result set behaves like a read-only table.
  - The number of rows or columns in the result set is irrelevant. It is still treated as a table.
- Can appear in two different contexts:
  - In the `FROM` clause of an enclosing query
  - As right operands to the following operators to return a Boolean result:
    - The logical operators `IN` and `EXISTS`
    - The regular comparison operators (`=, !=, <>, <, >, <= or >=`) quantified with `ALL`, `ANY`, and `SOME`

Subqueries in the FROM Clause

- The result set of a subquery in the `FROM` clause is treated in the same way as a base table or a view.

```
SELECT *
FROM (SELECT Code, Name
  FROM Country
  WHERE IndepYear IS NOT NULL) AS IndependentCountry;
```

  - The subquery executes first and produces a temporary result set
  - You must specify a table alias for the result set or you

```
mysql> SELECT * FROM (SELECT Code, Name FROM Country WHERE IndepYear IS NOT NULL);
ERROR 1248 (42000): Every derived table must have its own alias
```

  - The subquery result is discarded after the full statement is executed

Calculating Aggregates of Aggregates

- Subqueries in the `FROM` clause are especially useful for calculating *aggregates of aggregates*. For example, to display the average population of each continent:

```
mysql> SELECT AVG(cont_sum)
    -> FROM (
    ->   SELECT Continent, SUM(Population) AS cont_sum
    ->   FROM Country
    ->   GROUP BY Continent
    -> ) AS t;
+----------------+
| AVG(cont_sum)  |
+----------------+
| 868392778.5714 |
+----------------+
1 row in set (#.## sec)
```

  - This statement evaluates the table subquery first, calculating the total of all country populations for each continent.
  - The outer query then uses the `AVG` function to aggregate the resulting rows (one for each continent).

Subqueries Using the IN Operator

- You can use a table subquery as the right operand for the `IN` operator.
- The `IN` operator examines a table subquery for a value equal to its left operand, and returns:
  - `TRUE` if at least one such value exists
  - `FALSE` if there is no such value
- If the table subquery returns only a single column, the left operand must be a scalar value expression.

Subquery Using the IN Operator: Example

```
mysql> SELECT * FROM City WHERE CountryCode IN(
    ->   SELECT Code FROM Country WHERE Continent = 'Asia');
+------+----------------+-------------+----------------------+------------+
| ID   | Name           | CountryCode | District             | Population |
+------+----------------+-------------+----------------------+------------+
|    1 | Kabul          | AFG         | Kabol                |    1780000 |
|    2 | Qandahar       | AFG         | Qandahar             |     237500 |
|    3 | Herat          | AFG         | Herat                |     186800 |
|    4 | Mazar-e-Sharif | AFG         | Balkh                |     127800 |
|   64 | Dubai          | ARE         | Dubai                |     669181 |
|   65 | Abu Dhabi      | ARE         | Abu Dhabi            |     398695 |
|   66 | Sharja         | ARE         | Sharja               |     320095 |
|   67 | al-Ayn         | ARE         | Abu Dhabi            |     225970 |
|   68 | Ajman          | ARE         | Ajman                |     114395 |
|  126 | Yerevan        | ARM         | Yerevan              |    1248700 |
|  127 | Gjumri         | ARM         | Širak                |     211700 |
|  128 | Vanadzor       | ARM         | Lori                 |     172700 |
|  144 | Baku           | AZE         | Baki                 |    1787800 |
|  145 | Gäncä          | AZE         | Gäncä                |     299300 |
|  146 | Sumqayit       | AZE         | Sumqayit             |     283000 |
|  147 | Mingäçevir     | AZE         | Mingäçevir           |      93900 |
|  150 | Dhaka          | BGD         | Dhaka                |    3612850 |
|  151 | Chittagong     | BGD         | Chittagong           |    1392860 |
|  152 | Khulna         | BGD         | Khulna               |     663340 |
...
1766 rows in set (#.## sec)
```

Pairwise IN Operation

- The following example illustrates *pairwise column comparison* to list Asian cities with the same name as their respective countries:

```
mysql> SELECT * FROM City WHERE (CountryCode, Name) IN (
    ->   SELECT Code, Name FROM Country WHERE Continent = 'Asia');
+------+-----------+-------------+----------+------------+
| ID   | Name      | CountryCode | District | Population |
+------+-----------+-------------+----------+------------+
| 2429 | Kuwait    | KWT         | al-Asima |      28859 |
| 2454 | Macao     | MAC         | Macau    |     437500 |
| 3208 | Singapore | SGP         | -        |    4017733 |
+------+-----------+-------------+----------+------------+
3 rows in set (#.## sec)
```

  - The subquery retrieves the codes and names of all Asian countries.
  - The outer query retrieves cities and compares the city's name and country code to each of the Asian country code and name rows.

Using NOT IN

- Negate the `IN` operator by prefixing the `IN` keyword with the `NOT` keyword.
  - `NOT IN` evaluates to `TRUE` if the result set does not contain an entry that is equal to the left operand.
  - If the result set does contain an entry equal to the left operand, `NOT IN` evaluates to `FALSE`.

The IN Operator and NULL

- If it is impossible to determine whether the right operand contains a value equal to the left operand, `IN` evaluates to `NULL`.
- There are two different scenarios in which `IN` can evaluate to `NULL`:
  - If the left operand is `NULL` and the result set formed by the right operand is not empty
  - If the result set contains at least one row for which at least one of the columns is `NULL`, and no row matches the left operand

EXISTS Operator

- The `EXISTS` operator accepts a single right argument, which must be a table subquery.
  - If the subquery result set contains at least one row, then the `EXISTS` expression evaluates to `TRUE`.
  - Otherwise, it returns `FALSE`.
- For example, the following query retrieves all cities that are the capital of some country:

```
mysql> SELECT * FROM City WHERE EXISTS (
    ->   SELECT NULL FROM Country WHERE Capital = ID);
+------+-------------+-------------+---------------+------------+
| ID   | Name        | CountryCode | District      | Population |
+------+-------------+-------------+---------------+------------+
|    1 | Kabul       | AFG         | Kabol         |    1780000 |
|    5 | Amsterdam   | NLD         | Noord-Holland |     731200 |
|   33 | Willemstad  | ANT         | Curaçao       |       2345 |
...
232 rows in set (#.## sec)
```

Using NOT EXISTS

- Negate `EXISTS` by placing the `NOT` keyword before the `EXISTS` keyword.
  - `NOT EXISTS` returns `TRUE` only if its table subquery argument is empty, and `FALSE` otherwise.
- The following query illustrates how to use `NOT EXISTS` to find all the countries where English is not spoken:

```
mysql> SELECT Code, Name FROM Country WHERE NOT EXISTS (
    ->   SELECT NULL FROM CountryLanguage WHERE CountryCode = Code
    ->   AND Language = 'English');
+------+--------------------------------------------+
| Code | Name                                       |
+------+--------------------------------------------+
| AFG  | Afghanistan                                |
| AGO  | Angola                                     |
| ALB  | Albania                                    |
| AND  | Andorra                                    |
| ARE  | United Arab Emirates                       |
...
179 rows in set (#.## sec)
```

Table Subquery Quantifiers

- `ALL`, `ANY`, and `SOME` are quantifiers. They differ from operators in that they:
  - Are used in *conjunction* with the regular comparison operators (`=`, `!=`, `<>`, `<`, `>`, `<=`, and `>=`).
  - Specify how to apply an operator to repeatedly compare a single operand on the left side of an operator to the multiple values returned by a subquery on the right side of the operator.
- The following is an example using the `ANY` quantifier:

```
mysql> SELECT 'Finland' = ANY (
    ->   SELECT Name FROM Country) AS IsCountry;
+-----------+
| IsCountry |
+-----------+
|         1 |
+-----------+
1 row in set (#.## sec)
```

ALL, ANY, and SOME

- `ALL`: The comparison is `TRUE` if the operator is applied to all items in the subquery result set and returns `TRUE` in all cases.
  - For example, to list countries when their population exceeds that of all cities:

```
SELECT * FROM Country WHERE Population > ALL (
  SELECT Population FROM City);
```

- `ANY`: The operator is applied to the items in the subquery result set until at least one comparison returns `TRUE`. It returns `FALSE` if none of the comparisons returned `TRUE`.
- `SOME`: This is an alias of `ANY`, and has exactly the same effect. However, you might find the statement easier to read when using the `SOME` keyword instead of the `ANY` keyword.

Alternatives to ANY and ALL

- `ANY` and `ALL` can sometimes be confusing to read. The following list shows more readable alternatives for several quantified operator expressions.

Quantified operator expression	Alternative
`scalar > ANY(SELECT column...)`	`scalar > (SELECT MIN(column)...)`
`scalar < ANY(SELECT column...)`	`scalar < (SELECT MAX(column)...)`
`scalar > ALL(SELECT column...)`	`scalar > (SELECT MAX(column)...)`
`scalar < ALL(SELECT column...)`	`scalar < (SELECT MIN(column)...)`
`scalar = ANY(SELECT column...)`	`scalar IN (SELECT column...)`
`scalar <> ALL(SELECT column...)`	`scalar NOT IN (SELECT column...)`

Correlated Subqueries

A *correlated subquery:*

- Contains one or more expressions that are derived from the outer query and is, therefore, dependent on the outer query. For example:

```
SELECT *
FROM Country
WHERE NOT EXISTS (
   SELECT NULL
   FROM City
   WHERE CountryCode = Code
);
```

  - The Code column referenced by the subquery depends on the `Country` table in the outer query.
- Can appear in any context where a subquery is valid, except in the `FROM` clause

Correlated Subquery Scope

- Column references in correlated subqueries resolve using the nearest possible scope. Consider this

```
SELECT *
FROM City
WHERE CountryCode IN (
    SELECT Code
    FROM Country
    WHERE Name = 'Belgium'
);
```

Both tables contain `Name` columns

  - The subquery is considered first and the nearest possible source for the `Name` column is the `Country` table in its `FROM` clause
  - The subquery uses the `Name` column in the `Country` table
  - The outer query uses the `Name` column in the `City` table

Modifying Data with Subqueries

- You can use subqueries with data modification statements.

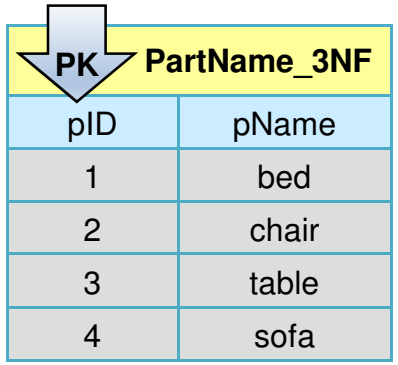
```
DELETE FROM City
WHERE CountryCode IN (
    SELECT Code FROM Country
    WHERE LifeExpectancy < 70.0
);
```

```
UPDATE Country
SET Population = (
    SELECT SUM(Population) FROM City
    WHERE CountryCode = Code
);
```

Data Types

Data Types as Part of Database Design

- A well-designed database uses the most suitable data type for each data item.
  - Think about the data each column needs to store.
  - Assign column data types accordingly:
    - Example: `PartName_3NF` in the `Inventory` database. The `pName` column contains character data and the `pID` column contains numeric data.
  - Follow the ABCs of data types:
    - **A**ppropriate
    - **B**rief
    - **C**omplete



Data Types: Overview

- Numeric data types
  - Integer, decimal, and floating point data
- Temporal data types
  - Date and time data
- String data types
  - Character strings and binary strings

Numeric Data Types

- Numeric data types:
  - **Integer:** Whole numbers
  - **Fixed-point:** Exact-value fractional numbers
  - **Floating-point:** Approximate-value fractional numbers
  - **Bit:** Bit-field values
- Factors to consider with numeric data types:
  - Range of values to be stored
  - Amount of storage space required
  - Column precision and scale (floating-point and fixed-point)

Numeric: Integer Data Types

- Data types:
  - **`TINYINT`:** Very small integer data type
  - **`SMALLINT`:** Small integer data type
  - **`MEDIUMINT`:** Medium-sized integer data type
  - **`INT`**, **`INTEGER`:** Normal (average) size integer data type
  - **`BIGINT`:** Large integer data type
- Example:
  - `world` database, `City` table, `Population` column: **`Population INT(11)`**
  - Largest value stored: 10500000 (uses 8; 11 allowed)
- Attributes:
  - **`UNSIGNED`**: Has no negative numbers
  - **`ZEROFILL`:** Replaces padding spaces with zeroes

Numeric: Integer Data Types Comparison

Type	Storage Required	Signed Range	Unsigned Range
`TINYINT`	1 byte	–128 to 127	0 to 255
`SMALLINT`	2 bytes	–32,768 to 32,767	0 to 65,535
`MEDIUMINT`	3 bytes	–8,388,608 to 8,388,607	0 to 16,777,215
`INT`, `INTEGER`	4 bytes	–2,147,683,648 to 2,147,483,647	0 to 4,294,967,295
`BIGINT`	8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615

Numeric: Fixed-Point Data Types

- Used for exact-value numbers: Integer, fractional, or both
- Data types:
  - **`DECIMAL`** (or **`NUMERIC`**):
    - Fixed-decimal, binary storage format
    - Preserve exact precision
- Example:
  - To represent currency values, such as dollars and cents:

    ```
    cost DECIMAL(10,2)
    ```

  - Example value output:

    650.88

Numeric: Floating-Point Data Types

- Used for approximate-value numbers: Integer, fractional, or both
- Data types:
  - **`FLOAT:`** Stores single-precision values using four bytes
  - **`DOUBLE:`** Stores double-precision values using eight bytes
- Can declare with precision and scale
- Example:
  - `world` database, `Country` table, `GNP` column:
    `GNP FLOAT(10,2)`
  - Largest value stored in the `GNP` column:
    8510700.00 (up to 10 digits available, 9 displayed, 2 decimal digits)

Numeric: Floating-Point Data Types Comparison

Type	Storage Required	Signed Range	Unsigned Range
FLOAT	4 bytes	-3.402823466E+38 to -1.175494351E-38 to 1.175494351E-38 to 3.402823466E+38	0 and 1.175494351E-38 to 3.402823466E+38
DOUBLE	8 bytes	-1.7976931348623157E+308 to -2.2250738585072014E-308 and 2.2250738585072014E-308 to 1.7976931348623157E+308	0 and 2.2250738585072014E-308 to 1.7976931348623157E+308

Numeric: Bit Data Types

- Data type:
  - **`BIT(M)`**
- Bit-field values:
  - Column Width (**`M`**): Number of bits per value
  - 1 to 64 bits
- Example:
  - Storing 4 bits and 20 bits:

```
bit_col1 BIT(4)
bit_col2 BIT(20)
```

- Can assign values by using bit-field notation

`b'1111'` or **`0b1111`**

`b'10000000'` or **`0b10000000`**

Temporal Data Types

- **`DATE`**:
  - `YYYY-MM-DD` as in 2016-01-04
- **`TIME`**:
  - `HH:MM:SS` as in 12:59:02
- **`DATETIME`**:
  - `YYYY-MM-DD HH:MM:SS` as in 2016-01-04 12:59:02
- **`TIMESTAMP`**:
  - Current date and time as in 2016-01-04 12:59:02
- **`YEAR`**:
  - Four digits `(YYYY)` as in 1969
- `TIME`, `DATETIME`, and `TIMESTAMP` values can accept fractional seconds with up to microseconds (six digits of precision).

Temporal Data Types Comparison

Type	Storage Required	Range
`DATE`	3 bytes	1000-01-01 to 9999-12-31
`TIME`	3 bytes	-838:59:59 to 838:59:59 Or -838:59:59.000000 to 838:59:59.000000
`DATETIME`	8 bytes	1000-01-01 00:00:00 to 9999-12-31 23:59:59 Or 1000-01-01 00:00:00.000000 to 9999-12-31 23:59:59.999999
`TIMESTAMP`	4 bytes	1970-01-01 00:00:01 to 2038-01-19 03:14:07 Or 1970-01-01 00:00:01.000000 to 2038-01-19 03:14:07.999999
`YEAR`	1 byte	1901 to 2155

String Types

- `CHAR` and `VARCHAR` types
- `BINARY` and `VARBINARY` types
- `BLOB` and `TEXT` types
- `ENUM` type
- `SET` type

String: CHAR and VARCHAR Types

- Data types:
  - **`CHAR:`** Fixed-length (static) character strings
  - **`VARCHAR:`** Variable-length (dynamic) character strings
- `CHAR` example:
  - `world` database, `CountryLanguage` table, `Language` column defined as:
    `Language CHAR(30)`
  - Largest value (needs 25):

    ```
    Southern Slavic Languages
    ```
- `VARCHAR` example:

  `mfrName VARCHAR(45)`

String: BINARY and VARBINARY Types

- Data types:
  - **`BINARY:`** Fixed-length (static) binary byte strings
  - **`VARBINARY:`** Variable-length (dynamic) binary byte strings
- `BINARY` example:

```
bin_val BINARY(6)
```

  - Stores 6 bytes of binary data, right-padded with zeros
- `VARBINARY` example:

```
varbin_val VARBINARY(6)
```

  - Stores up to 6 bytes of binary data with no padding

String: BLOB and TEXT Types

- **`BLOB`**:
  - **B**inary **L**arge **Ob**ject
  - Variable-length (dynamic) binary byte strings
  - Includes `TINYBLOB`, `BLOB`, `MEDIUMBLOB`, and `LONGBLOB`
- **`TEXT`**:
  - Variable-length (dynamic) character strings
  - Includes `TINYTEXT`, `TEXT`, `MEDIUMTEXT`, and `LONGTEXT`
- `BLOB` and `TEXT` data types can store extremely long values, such as media files.

ENUM and SET Data Types

- Data types:
  - **`ENUM`**: Single string value chosen from a list of enumerated string values
  - **`SET`**: Zero or more string values chosen from a list of string values
- **`ENUM`** example:
  - `Country` table, `Continent` column:

```
Continent ENUM('Asia', 'Europe', 'North
      America',     'Africa', 'Oceana', 'Antarctica',
'South       America')
```

- **`SET`** example:

```
Symptom SET('sneezing', 'runny nose', 'stuffy head',
      'red eyes')
```

String Data Types Comparison

Type	Storage Required	Maximum Length
CHAR (M)	*M* bytes	255 characters
BINARY (M)	*M* bytes	255 bytes
VARCHAR (M), **VARBINARY (M)**	*L* + 1 or 2 bytes	65,535 bytes
TINYBLOB, **TINYTEXT**	*L* + 1 bytes	255 bytes
BLOB, **TEXT**	*L* + 2 bytes	65,535 bytes
MEDIUMBLOB, **MEDIUMTEXT**	*L* + 3 bytes	16,777,215 bytes
LONGBLOB, **LONGTEXT**	*L* + 4 bytes	4,294,967,295 bytes
ENUM	1 or 2 bytes	65,535 values
SET	1, 2, 3, 4, or 8 bytes	64 members

Character Set and Collation Support

- **Character set:** A named, encoded group of characters
  - All character strings belong to a specific character set.
- **Collation:** A named collating sequence for a character set
  - Defines character sort order
  - Affects comparison of characters and strings
- Column definitions for string data types can specify a character set or collation for the column.
- Character set and collation example:

```
CREATE TABLE t
( c1 VARCHAR(20) CHARACTER SET utf8,
  c2 TEXT CHARACTER SET latin1
   COLLATE latin1_general_cs );
```

Choosing Data Types

- When selecting a data type, consider:
  - What sort of data it will store. For example, use a numeric data type for a number (instead of a character string).
  - The size of the data it will store:
    - Max size of integers, for example:
      - Age: **`TINYINT UNSIGNED`** (0 to 255)
      - A country's population: **`INT UNSIGNED`** (0 to 4,294,967,295)
    - Max length of character strings, for example:
      - Student grade: **`CHAR(2)`** (A+, B-)
      - Last name: **`VARCHAR(64)`**

Setting Data Types to NULL

- **`NULL`** is a SQL keyword used to define data types that allow missing values. It means one of two things:
  - Unknown: There is a value, but the precise value is not known at this time.
  - Not applicable: If a value is specified, it would not be accurately representative.
- Use **`NULL`**:
  - In place of a real value in several situations
    - For example: “no value,” “unknown value,” “missing value,” “out of range,” “not applicable,” and so on
  - To represent an empty query result
- Plan for null values during database design.

When to Use NULL

- During database design, in cases where column information is not available, determine whether null values are allowed.
- For example, the `world` database's `Country` table contains countries with no life expectancy data and should allow nulls.
- Nulls are allowed by default.
- Use **`NOT NULL`** if a column must not allow nulls, to ensure data integrity.
- You can apply **`NULL`** and **`NOT NULL`** to existing column definitions.

Working with Strings

SQL Expressions

- A SQL expression is a combination of one or more values and SQL functions (“terms”) that, combined with operators and rules of precedence and association, evaluates to a single value. An expression in MySQL:
  - Can appear in many places as part of SQL statements, such as the `SELECT` list, the `WHERE` clause, the `ORDER BY` clause, and the `GROUP BY` clause to identify which rows to affect
  - Generally assumes the data type of its components
  - Can include many different terms, including:
    - Literals (numbers, strings, dates, and times)
    - Built-in constants such as `NULL`, `TRUE`, and `FALSE`
    - Column references
    - Function calls

String Expressions

- When using string expressions you
  - Must quote literal string values
    - MySQL allows either single (`'`) or double (`''`) quotation marks.
    - The `ANSI_QUOTES` SQL mode interprets double-quoted characters as identifiers.
    - Single quotation marks are preferred for portability
  - Can use the following string data types: `CHAR`, `VARCHAR`, `TEXT`, and `BLOB`
  - Can use string expressions in comparison operations
    - Comparison operators (such as =, <>, <, and `BETWEEN...AND`) can be applied to string values.
  - Perform most string operations by using functions

String Functions

- Perform operations on strings, such as:
  - Calculating string lengths
  - Extracting pieces of strings
  - Searching for substrings or replacing them
  - Performing case conversion
- String functions are divided into two categories:
  - **Numeric:** Return numbers
  - **String:** Return strings

Concatenating Strings Using MySQL Functions

- Use the `CONCAT()` function

```
mysql> SELECT CONCAT('abc', 'def', REPEAT('X',3));
+--------------------------------------+
| CONCAT('abc', 'def', REPEAT('X',3)) |
+--------------------------------------+
| abcdefXXX                            |
+--------------------------------------+
1 row in set (0.00 sec)
```

- Use the `CONCAT_WS()` function

```
mysql> SELECT CONCAT_WS(' ', 'abc', 'def', REPEAT('X',3));
+---------------------------------------------+
| CONCAT_WS(' ', 'abc', 'def', REPEAT('X',3)) |
+---------------------------------------------+
| abc def XXX                                 |
+---------------------------------------------+
1 row in set (0.00 sec)
```

Concatenating NULL Values

- Any `NULL` argument passed to `CONCAT()` causes it to return `NULL`.
- `CONCAT_WS()` ignores `NULL` arguments.
- Comparing `CONCAT()` and `CONCAT_WS()`:

```
mysql> SELECT CONCAT('a','b'),
    -> CONCAT('a',NULL,'b');
+-----------------+----------------------+
| CONCAT('a','b') | CONCAT('a',NULL,'b') |
+-----------------+----------------------+
| ab              | NULL                 |
+-----------------+----------------------+
mysql> SELECT CONCAT_WS('/','a','b'),
    -> CONCAT_WS('/','a', NULL,'b');
+------------------------+-----------------------------+
| CONCAT_WS('/','a','b') | CONCAT_WS('/','a',NULL,'b') |
+------------------------+-----------------------------+
| a/b                    | a/b                         |
+------------------------+-----------------------------+
```

Concatenating Strings Using the OR (||) Operator

1. Set the `PIPES_AS_CONCAT` SQL mode

```
mysql> SET sql_mode='PIPES_AS_CONCAT';
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

2. Use the `OR` (`||`) operator to concatenate strings

```
mysql> SELECT 'abc' || 'def';
+----------------+
| 'abc' || 'def' |
+----------------+
| abcdef         |
+----------------+
1 row in set (0.0 sec)
```

Retrieving the Left and Right Portions of a String

- Use *LEFT(str,len)* to retrieve the *len* left-most characters of the string *str*

```
mysql> SELECT LEFT('Alice and Bob', 5);
+--------------------------+
| LEFT('Alice and Bob', 5) |
+--------------------------+
| Alice                    |
+--------------------------+
1 row in set (0.00 sec)
```

- Use *RIGHT(str,len)* to retrieve the *len* right-most characters of the string *str*

```
mysql> SELECT RIGHT('Alice and Bob', 3);
+---------------------------+
| RIGHT('Alice and Bob', 3) |
+---------------------------+
| Bob                       |
+---------------------------+
1 row in set (0.00 sec)
```

Retrieving Part of a String

- Use `SUBSTRING(str, pos, len)` to return `len` characters of the string, starting at `pos`

```
mysql> SELECT SUBSTRING('Alice and Bob', 1, 5);
+----------------------------------+
| SUBSTRING('Alice and Bob', 1, 5) |
+----------------------------------+
| Alice                            |
+----------------------------------+
1 row in set (0.00 sec)
```

Return a part of the first string starting from the offset:

```
mysql> SELECT SUBSTRING_INDEX(
    -> 'Alice and Bob', 'and', 1)
    -> AS Result;
+--------+
| Result |
+--------+
| Alice  |
+--------+
1 row in set (0.00 sec)
```

Return a part of the second string starting from the offset:

```
mysql> SELECT SUBSTRING_INDEX(
    -> 'Alice and Bob', 'and', -1)
    -> AS Result;
+--------+
| Result |
+--------+
|  Bob   |
+--------+
1 row in set (0.00 sec)
```

Trimming a String

- Use `LTRIM()`, `RTRIM()`, or `TRIM()` to remove space characters appearing at the left, right, or both extremities of string, respectively

```
mysql> SELECT CONCAT('<', LTRIM(' Alice '), '>'),
    -> CONCAT('<', RTRIM(' Alice '), '>'),
    -> CONCAT('<', TRIM(' Alice '), '>') \G
*********************** 1. row ***********************
CONCAT('<', LTRIM(' Alice '), '>'): <Alice >
CONCAT('<', RTRIM(' Alice '), '>'): < Alice>
CONCAT('<', TRIM(' Alice '), '>'): <Alice>
```

- Remove spaces or specific characters from the beginning (`LEADING`), end (`TRAILING`) or both (`BOTH`) sides of a string. For example:

```
mysql> SELECT TRIM(LEADING 'Cha' FROM 'ChaChaChalice');
+-----------------------------------------+
| TRIM(LEADING 'Cha' FROM 'ChaChaChalice') |
+-----------------------------------------+
| lice                                    |
+-----------------------------------------+
```

Inserting Into and Replacing Portions of a String

- Use `REPLACE(`*`str`*`,`*`substr`*`,`*`newstring`*`)` to replace *`substr`* in *`str`* with *`newstring`*

```
mysql> SELECT REPLACE('Alice & Bob', '&', 'and');
+-----------------------------------+
| REPLACE('Alice & Bob', '&', 'and') |
+-----------------------------------+
| Alice and Bob                     |
+-----------------------------------+
```

- Use `INSERT(`*`str`*`,`*`pos`*`,`*`len`*`,`*`newstring`*`)` to place *`newstring`* at *`pos`* in *`str`*, overwriting *`len`* characters

```
mysql> SELECT INSERT('Alice and Bob', 6, 5, ', Carol & ');
+-------------------------------------------+
| INSERT('Alice and Bob', 6, 5, ', Carol & ') |
+-------------------------------------------+
| Alice, Carol & Bob                        |
+-------------------------------------------+
```

Determining String Length

- Use the `LENGTH()` and `CHAR_LENGTH()` to return the string lengths in byte and character

```
mysql> SELECT LENGTH('MySQL'),
    -> CHAR_LENGTH('MySQL');
+----------------+---------------------+
| LENGTH('MySQL') | CHAR_LENGTH('MySQL') |
+----------------+---------------------+
| 5              | 5                   |
+----------------+---------------------+

mysql> SELECT LENGTH(CONVERT('MySQL' USING ucs2))
    -> AS length,
    -> CHAR_LENGTH(CONVERT('MySQL' USING ucs2))
    -> AS c_length;
+------------+------------+
| length     | c_length   |
+------------+------------+
| 10         | 5          |
+------------+------------+
```

Comparing Strings

- Use `STRCMP(str1,str2)` to compare *str1* and *str2*.
  - Check return value: 0 = *str1* and *str2* are the same, –1 = *str1* is smaller than *str2*, 1 = *str1* is larger than *str2*

```
mysql> SELECT STRCMP('abc','def'),
    -> STRCMP('def','def'),
    -> STRCMP('def','abc');
+--------------------+--------------------+--------------------+
|STRCMP('abc','def') |STRCMP('def','def') | STRCMP('def','abc') |
+--------------------+--------------------+--------------------+
| -1                 | 0                  | 1                  |
+--------------------+--------------------+--------------------+
```

- Use the equals operator (=) to test for string equality:

```
mysql> SELECT 'abc' = 'def', 'def' = 'def', 'def' = 'abc';
+---------------+---------------+---------------+
| 'abc' = 'def' | 'def' = 'def' | 'def' = 'abc' |
+---------------+---------------+---------------+
| 0             | 1             | 0             |
+---------------+---------------+---------------+
```

Finding Strings Within Strings

- Use `INSTR()`, `LOCATE()` or `POSITION()` to return the position of the specified string inside another string:

```
mysql> SELECT INSTR('Alice and Bob', 'and'),
    -> LOCATE('and', 'Alice and Bob'),
    -> POSITION('and' IN 'Alice and Bob')
    -> \G
*********************** 1. row ***********************
INSTR('Alice and Bob', 'and'): 7
LOCATE('and', 'Alice and Bob'): 7
POSITION('and' IN 'Alice and Bob'): 7
```

- Use `LOCATE(`*`searchstr,str,pos`*`)`, where *`pos`* specifies the start position for the search :

```
mysql> SELECT LOCATE(' ', 'Alice and Bob', 7);
+---------------------------------+
| LOCATE(' ', 'Alice and Bob', 7) |
+---------------------------------+
| 10                              |
+---------------------------------+
```

String Comparison and Sorting Behavior

- String comparison and sort behavior depends on the *collation*
- Collations can be case sensitive, case insensitive, and language-specific
- The default collation in MySQL is `latin1_swedish_ci`. The `_ci` suffix denotes a case insensitive collation. Example behavior:

```
mysql> CREATE TABLE t1 (col1 CHAR(3) CHARACTER SET latin1);
mysql> INSERT INTO t1 (col1) VALUES('AAAA'),('bbbb'),('aaaa'),('BBBB');
mysql> SELECT col1 FROM t1 ORDER BY col1;
+-------+
| c     |
+-------+
| AAAA  |
| aaaa  |
| bbbb  |
| BBBB  |
+-------+
```

Character Sets and Collations

- A *character set* is a set of symbols and encodings. A *collation* is a set of rules for comparing characters in a character set.
- MySQL allows you to:
  - Store strings using a variety of character sets.
  - Compare strings using a variety of collations.
  - Mix strings with different character sets or collations in the same server, the same database, or even the same table.
  - Enable specification of character set and collation at any level.
  - Display the available character sets and collations using the following syntax

```
SHOW COLLATION [LIKE 'pattern' | WHERE expr]

SHOW CHARACTER SET [LIKE 'pattern' | WHERE expr]
```

Character Set and Collation Functions

- Use `CHARSET(str)` and `COLLATION(str)` to retrieve the character set and collation of *str*, respectively

```
mysql> SELECT USER(), CHARSET(USER()), COLLATION(USER());
+----------------+-----------------+-------------------+
| USER()         | CHARSET(USER()) | COLLATION(USER()) |
+----------------+-----------------+-------------------+
| root@localhost | utf8            | utf8_general_ci   |
+----------------+-----------------+-------------------+
```

- Use `CONVERT()` or `CAST()` to change a string's character set

```
mysql> SELECT
CONVERT(_latin1'Müller'
USING utf8) AS Result;
+-----------+
| Result    |
+-----------+
| MÃ¼ller   |
+-----------+
1 row in set (0.00 sec)
```

```
mysql> SELECT CAST(_latin1'test'
AS CHAR CHARACTER SET utf8) AS
Result;
+--------+
| Result |
+--------+
| test   |
+--------+
1 row in set (0.00 sec)
```

Changing the Default Collation

- Use the `COLLATE` operator on the column you want to affect

```
mysql> SELECT col1 FROM t1 ORDER BY col1 COLLATE latin1_general_cs;
+-------+
| col1  |
+-------+
| AAAA  |
| aaaa  |
| BBBB  |
| bbbb  |
+-------+
```

- Specify a character set and collation for the entire database

```
CREATE DATABASE mydb
  DEFAULT CHARACTER SET utf8
  DEFAULT COLLATE utf8_general_ci;
```

- Set the `--character-set-server` and `--collation-server` options when you start the MySQL server

String Pattern Matching

- Use the `LIKE` pattern-matching operator to perform comparisons based on similarity.
  - General syntax:

```
SELECT...<expression> LIKE '<pattern>'...
```

- Use metacharacters with `LIKE`:
  - The percent character '%' matches any sequence of zero or more characters.
  - The underscore character '_' matches any single character.
- Use `NOT LIKE` for opposite comparisons.

String Pattern Matching: LIKE Versus NOT LIKE

- Example of `LIKE`:

```
mysql> SELECT Name FROM Country
    -> WHERE Name LIKE 'United%';
+---------------------------------+
| Name                            |
+---------------------------------+
| United Arab Emirates            |
| United Kingdom                  |
| United States Minor Outlying Isl. |
| United States                   |
+---------------------------------+
```

- Example of logical `NOT LIKE`:

```
mysql> SELECT Name FROM Country
    -> WHERE Name NOT LIKE 'United%';
+--------------------------------+
| Name                           |
+--------------------------------+
| Aruba                          |
...
| Zimbabwe                       |
+--------------------------------+
```

String Pattern Matching: Regular Expressions

A regular expression is a powerful way of specifying a pattern for a complex search.

- Use the `RLIKE` or `REGEXP` operators.
  - General syntax:

```
SELECT... <string-expr>
        RLIKE|REGEXP '<regexp-string>'...
```

- Use for complex patterns that go beyond general wildcard characters such as `'%'` and `'_'` in `LIKE` patterns.
- Use to test the following:
  - URLs
  - IP addresses
  - Email addresses
  - Postal codes

String Pattern Matching: RLIKE Syntax

- The `RLIKE` expression is a character string with its own syntax to allow a declarative denotation of complex patterns.
  - Literal text: Match character *exactly.*
  - Period: (`.`) Match any single character.
  - Occurrence: (`*`, `?`, `+`, `{m,n}`) Specify the number of occurrences.
  - Choice: (`|`) Match either of two alternatives.
  - Escape: (`\`) Escape special characters
  - Anchors: Match special positions. (`^` and `$` match start and end of input; `[[:>:]]` and `[[:<:]]` match word boundaries.)
  - Character classes: `[`*`spec`*`]` Match against a collection.
  - Parentheses: `(` *`and`* `)` Group a sequence of elements.

String Pattern Matching: RLIKE Character Class

- Notate character class specifications in brackets (`[` and `]`).
  The syntax for this notation is as follows:
  - Literal text: `'[abc]'` matches either `'a'` or `'b'` or `'c'`.
  - Negation: `'[^abc]'` matches everything *except* `'a'`, `'b'`, or `'c'`.
  - Ranges: `'[a-z]'` matches `'a'`, `'b'`, `'c'`,. . . `'z'`.
  - Named range: `'[[:space:]]'` matches all whitespace characters.

String Pattern Matching: Regular Expressions

- Simple (all matches possible):

```
mysql> SELECT Name FROM City WHERE Name RLIKE 'nat';
+-----------------+
| Name            |
+-----------------+
| Natal           |
| Cabanatuan      |
| Maunath Bhanjan |
| Minatitlβn      |
| Cincinnati      |
+-----------------+
```

- Anchors (only complete match):

```
mysql> SELECT Name FROM City WHERE Name RLIKE '^new.*rk$';
+----------+
| Name     |
+----------+
| New York |
| Newark   |
+----------+
```

String Pattern Matching: Regular Expressions

- Alternation (choice):

```
mysql> SELECT Name FROM City
    -> WHERE Name RLIKE ' Los | Las ';
+------------------------------+
| Name                         |
+------------------------------+
| San Nicolßs de los Arroyos   |
| Santiago de los Caballeros   |
| Santo Domingo de los Colorados |
| Victoria de las Tunas        |
| San Nicolßs de los Garza     |
| Chilpancingo de los Bravo    |
| San Crist≤bal de las Casas   |
| East Los Angeles             |
| North Las Vegas              |
+------------------------------+
```

- Equivalent character class notation:

```
mysql> SELECT Name FROM City
    -> WHERE Name RLIKE ' L[ao]s ';
```

String Pattern Matching: Ranges of Characters

- Character classes offer convenient features to match an entire range of characters. The following example uses a character range to match Argentian postal codes:

```
SELECT CityName, StreetName FROM Addresses
    WHERE PostalCode
    RLIKE '^[A-HJ-NP-Z][0-9]{4}([A-Z]{3})?$';
```

  - Letter (A..H; J..N; P..Z) for the province: `[A-HJ-NP-Z]`
  - Four digits for the municipality: `[0-9]{4}`
  - (Optional) Three letters for the side of the street block: `([A-Z]{3})?`

String Pattern Matching: Escaping Characters

To match characters that are part of the regular expression syntax themselves, you must “escape” them.

- Requires ‘double’ escaping:

```
mysql> SELECT '\\' RLIKE '\\';
ERROR 1139 (42000): Got error 'trailing backslash (\)' from regexp

mysql> SELECT '\\' RLIKE '\\\\';
+-------------------+
| '\\' RLIKE '\\\\' |
+-------------------+
| 1                 |
+-------------------+
```

- No escape possible inside character class:

```
mysql> SELECT '\\' RLIKE '[\\]';
+-------------------+
| '\\' RLIKE '[\\]' |
+-------------------+
| 1                 |
+-------------------+
```

Full-Text Search: Overview

- Characteristics of `LIKE` and regular expression search:
  - **Poor performance:** MySQL has to scan the entire table.
  - **Little flexibility:** It is hard to implement certain searches.
  - **No relevance factor:** You cannot specify which results are most relevant.
- Characteristics of full-text search:
  - Requires the **`FULLTEXT`** index on searchable text columns
  - Better performance for complex queries
  - An automatically rebuilt index when data changes
  - Native SQL-like interface to search
  - Used on `CHAR`, `VARCHAR`, or `TEXT` columns
  - Works for `InnoDB` and `MyISAM` tables
  - Supports Natural Language and Boolean searches

Full-Text Search: Natural Language Mode

- Default is `IN NATURAL LANGUAGE MODE`.
- Example using the `film` table in `sakila` database:
  - Add `FULLTEXT` index to `description` column:

```
ALTER TABLE film
ADD FULLTEXT (description);
```

  - Use `MATCH()...AGAINST()` with delimited list of search keywords:

```
mysql> SELECT title, description FROM film
    -> WHERE MATCH(description) AGAINST ('composer, monkey');
...
164 rows in set (0.00 sec)
```

  - Results are returned in order of relevance
- Use delimiters between keywords:
  - Example: “ ” (space), “,” (comma), “.” (period)
- Use a stopword list to ignore certain words.

Full-Text Search: Boolean Mode

- Specify `IN BOOLEAN MODE` in the `AGAINST()` function to search for *words* instead of *concepts*
- Enables very sophisticated queries using operators

- Example using the – (omit) operator to find all occurrences of the word “composer” in `description` where there is no instance of the word “monkey”:

```
mysql> SELECT title, description FROM film
    -> WHERE MATCH(description)
    -> AGAINST ('composer -monkey' IN BOOLEAN MODE );
...
83 rows in set (0.00 sec)
```

  - Results are not returned in order of relevance.

Full-Text Search: Boolean Mode Operators

Operator	Usage
no operator	The word is optional, but rows containing it are ranked higher, as `IN NATURAL LANGUAGE MODE`.
`@distance`	Tests whether two words start in a specified number of words of each other (InnoDB only)
+	This word must be present in every row returned by the query.
–	This word must not be present in every row returned by the query.
> and <	Increases (>) or decrease(<) a word's relevance
( )	Parentheses group words into subexpressions.
~	Is used to mark "noise" words, causing their contribution to the row's relevance to be negative. Rows with negated words have a lower overall relevance but are not omitted entirely.
*	Is appended to a word as a wildcard
" "	Rows must match the literal contents of the string.

Working with Numeric and Temporal Data

Numeric Expressions

- Numbers can be exact-value or approximate-value literals.
  - **Exact-value literals:**
    - Used just as specified in SQL statements
    - Written as integer or decimal values, with no exponent
    - Not subject to the inexactness produced by rounding error
  - **Approximate-value literals:**
    - Not always used as specified in SQL statements
    - Written as floating-point numbers in scientific notation, with an exponent
    - Subject to rounding errors
- Nearly all numeric expressions that contain `NULL` return `NULL`.

Numeric Expressions: Examples

- Comparing the use of exact-values literal compared to approximate-value literals in expressions

```
mysql> SELECT 1.1 + 2.2 = 3.3, 1.1E0 + 2.2E0 = 3.3E0;
+----------------+----------------------+
| 1.1 + 2.2 = 3.3 | 1.1E0 + 2.2E0 = 3.3E0 |
+----------------+----------------------+
| 1               | 0                    |
+----------------+----------------------+
```

- Mixing numbers with strings

```
mysql> SELECT 1 + '1', 1 = '1';
+---------+---------+
| 1 + '1' | 1 = '1' |
+---------+---------+
| 2       | 1       |
+---------+---------+
```

Temporal Data Types

Data Type	Default Format
DATE	YYYY-MM-DD
TIME	HH:MI:SS
DATETIME	YYYY-MM-DD HH:MI:SS
TIMESTAMP	YYYY-MM-DD HH:MI:SS
DAY	DD
MONTH	MM
QUARTER	Q
YEAR	YYYY

Interval Arithmetic

- To perform interval arithmetic, use the `INTERVAL` keyword and specify the unit value.
  - The order of operands in interval addition is not

```
mysql> SELECT '2012-01-01' + INTERVAL 10 DAY,
    -> INTERVAL 10 DAY + '2012-01-01';
+-----------------------------+-----------------------------+
| '2012-01-01' + INTERVAL 10 DAY | INTERVAL 10 DAY + '2012-01-01' |
+-----------------------------+-----------------------------+
| 2012-01-11                  | 2012-01-11                  |
+-----------------------------+-----------------------------+
```

  - In interval subtraction, the interval value must be the

```
mysql> SELECT '2012-01-01' - INTERVAL 10 DAY;
+-----------------------------+
| '2012-01-01' - INTERVAL 10 DAY |
+-----------------------------+
| 2011-12-22                  |
+-----------------------------+
```

Numeric Functions

- Numeric functions perform different of mathematical operations.

Function Syntax	Definition
`ABS(<number>)`	Returns the absolute value of number
`SIGN(<number>)`	Returns –1, 0, or 1 depending on whether the number is negative, zero, or positive
`TRUNCATE(<number>, <decimals>)`	Returns number truncated to decimals
`FLOOR(<number>)`	Rounds number down to the closest lower integer
`CEILING(<number>)`	Rounds number up to the closest higher integer
`ROUND(<number>)`	Rounds number to the closest integer
`SIN(), COS(), TAN(), PI(), DEGREES(), RADIANS()`	Perform trigonometric calculations, including conversions between degrees and radians

Using ABS() and SIGN()

- `ABS()` returns the absolute value of negative and positive values

```
mysql> SELECT ABS(-42), ABS(42);
+----------+---------+
| ABS(-42) | ABS(42) |
+----------+---------+
|       42 |      42 |
+----------+---------+
```

- `SIGN()` returns the results of sign determination (–1 = negative, 0 = zero, 1 = positive)

```
mysql> SELECT SIGN(-42), SIGN(-1), SIGN(0), SIGN(1), SIGN(42);
+-----------+----------+---------+---------+----------+
| SIGN(-42) | SIGN(-1) | SIGN(0) | SIGN(1) | SIGN(42) |
+-----------+----------+---------+---------+----------+
| -1        | -1       | 0       | 1       | 1        |
+-----------+----------+---------+---------+----------+
```

Approximating Numbers With FLOOR() and CEILING()

- `FLOOR()` returns the largest integer not greater than its argument

```
mysql> SELECT FLOOR(-14.7), FLOOR(14.7);
+--------------+-------------+
| FLOOR(-14.7) | FLOOR(14.7) |
+--------------+-------------+
| -15          | 14          |
+--------------+-------------+
```

- `CEILING()` returns the smallest integer not less than its argument

```
mysql> SELECT CEILING(-14.7), CEILING(14.7);
+----------------+---------------+
| CEILING(-14.7) | CEILING(14.7) |
+----------------+---------------+
| -14            | 15            |
+----------------+---------------+
```

Approximating Numbers With ROUND()

- Rounding up to the next integer with exact values

```
mysql> SELECT ROUND(28.5), ROUND(-28.5);
+-------------+--------------+
| ROUND(28.5) | ROUND(-28.5) |
+-------------+--------------+
| 29          | -29          |
+-------------+--------------+
```

- Rounding up to the next integer with approximate values:

```
mysql> SELECT ROUND(2.85E1), ROUND(-2.85E1);
+---------------+----------------+
| ROUND(2.85E1) | ROUND(-2.85E1) |
+---------------+----------------+
| 28            | -28            |
+---------------+----------------+
```

Trigonometric Functions

- Perform trigonometric calculations, including conversions between degrees and radians.

```
mysql> SELECT SIN(0), COS(0), TAN(0);
+--------+--------+--------+
| SIN(0) | COS(0) | TAN(0) |
+--------+--------+--------+
| 0      | 1      | 0      |
+--------+--------+--------+
...

mysql> SELECT PI(), DEGREES(PI()), RADIANS(180);
+----------+----------------+------------------+
| PI()     | DEGREES(PI()) | RADIANS(180)     |
+----------+----------------+------------------+
| 3.141593 | 180           | 3.1415926535898 |
+----------+----------------+------------------+
```

Temporal Functions

- Perform operations such as:
  - Extracting parts of dates and times
  - Reformatting values
  - Converting values to seconds or days
- Use more than one format for the same information:
  - 2012-02-10 22:50:15
  - Friday, February 10, 2012
- Generate temporal data in many ways:
  - Copy existing data.
  - Execute the built-in function.
  - Build a string representation to be evaluated by the server.

Temporal Functions: Function Types

Function Syntax	Definition
`NOW()`	Current time as set on the server host (`TIME` format)
`CURDATE()`	Current date as set on the server host (`DATE` format)
`CURTIME()`	Current time as set on the server host (`TIME` format)
`DATEDIFF()`	Subtracts two dates
`DATE_ADD()`	Adds time values (intervals) to a date value
`YEAR(<date_expression>)`	Year in four-digit `YEAR` format
`MONTH(<date_expression>)`	Month of the year in integer format, per expression
`DAYOFMONTH(<date_expression>)` or `DAY(<date_expression>)`	Day of the month in integer format, per expression
`DAYNAME(<date_expression>)`	Day of the week in string format, per expression (English)
`HOUR(<date_expression>)`	Hour of the day in integer format (in 0–23 range), per expression
`MINUTE(<date_expression>)`	Minute of the day in integer format, per expression
`SECOND(<date_expression>)`	Second of the minute in integer format, per expression
`GET_FORMAT(<date_expression>)`	Returns a date format string, per values indicated for date-type and international format

Displaying the Current Date and Time

- Display current date and time:

```
mysql> SELECT NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2015-11-11 10:40:43 |
+---------------------+
```

- Extract current temporal data (date, time, and day of week):

```
mysql> SELECT CURDATE(), CURTIME(), DAYNAME(NOW());
+------------+-----------+----------------+
| CURDATE()  | CURTIME() | DAYNAME(NOW()) |
+------------+-----------+----------------+
| 2015-11-11 | 10:41:47  | Wednesday      |
+------------+-----------+----------------+
```

Determining Date and Time Format

- Use the `GET_FORMAT()` function to retrieve the format for the specified data type and standard
  - Syntax:

```
GET_FORMAT({DATE|TIME|DATETIME},
{'EUR'|'USA'|'JIS'|'ISO'|'INTERNAL'})
```

- Example, using `DATE` type and `EUR/USA` formats

```
mysql> SELECT GET_FORMAT(DATE, 'EUR'), GET_FORMAT(DATE, 'USA');
+--------------------------+--------------------------+
| GET_FORMAT(DATE, 'EUR') | GET_FORMAT(DATE, 'USA') |
+--------------------------+--------------------------+
| %d.%m.%Y                 | %m.%d.%Y                 |
+--------------------------+--------------------------+
```

Customizing the Output Format

- Use the `DATE_FORMAT()` function with format

```
mysql> SELECT NOW(), DATE_FORMAT(NOW(), '%W the %D of %M');
+---------------------+-------------------------------------+
| NOW()               | DATE_FORMAT(NOW(), '%W the %D of %M') |
+---------------------+-------------------------------------+
| 2015-11-11 11:11:04 | Wednesday the 11th of November      |
+---------------------+-------------------------------------+
```

- Other specifiers include
  - `%a`: Abbreviated weekdays name (`Sun...Sat`)
  - `%b`: Abbreviated month name (`Jan...Dec`)
  - `%d`: Numeric day of month (`0,1,2...`)
  - `%m`: Numeric month (`1...12`)
  - `%r`: 12 hr time (`hh:mm:ss`, followed by `AM` or `PM`)
  - `%T`: 24 hr time (`hh:mm:ss`)
  - `%Y`: Four-digit year (`YYYY`)

Retrieving Components of Dates and Times

```
mysql> SELECT YEAR('2015-11-11'),
    -> MONTH ('2015-11-11'),
    -> DAYOFMONTH('2015-11-11');
+--------------------+---------------------+--------------------------+
| YEAR('2015-11-11') | MONTH ('2015-11-11') | DAYOFMONTH('2015-11-11') |
+--------------------+---------------------+--------------------------+
|               2015 |                  11 |                       11 |
+--------------------+---------------------+--------------------------+
1 row in set (0.00 sec)

mysql> SELECT DAYOFYEAR('2015-11-11');
+-------------------------+
| DAYOFYEAR('2015-11-11') |
+-------------------------+
|                     315 |
+-------------------------+
1 row in set (0.00 sec)

mysql> SELECT HOUR('11:28:45'),
    -> MINUTE('11:28:45'),
    -> SECOND('11:28:45');
+------------------+--------------------+--------------------+
| HOUR('11:28:45') | MINUTE('11:28:45') | SECOND('11:28:45') |
+------------------+--------------------+--------------------+
|               11 |                 28 |                 45 |
+------------------+--------------------+--------------------+
1 row in set (0.00 sec)
```

Creating Date and Time Values

- Use `MAKEDATE(year,dayofyear)` to create dates

```
mysql> SELECT MAKEDATE(2015, 230);
+--------------------+
| MAKEDATE(2015, 230) |
+--------------------+
| 2015-08-18         |
+--------------------+
```

- Use `MAKETIME(hour,min,sec)` to create times

```
mysql> SELECT MAKETIME(11,20,35);
+-------------------+
| MAKETIME(11,20,35) |
+-------------------+
| 11:20:35          |
+-------------------+
```

Tables

Using the SHOW CREATE TABLE Statement

- Issue `SHOW CREATE TABLE` to understand the structure of a table. It displays the `CREATE TABLE` statement that creates the specified table. For example:

```
mysql> SHOW CREATE TABLE City\G
    ******************** 1. row ********************
           Table: City
    Create Table: CREATE TABLE `City` (
      `ID` int(11) NOT NULL auto_increment,
      `Name` char(35) NOT NULL default '',
      `CountryCode` char(3) NOT NULL default '',
      `District` char(20) NOT NULL default '',
      `Population` int(11) NOT NULL default '0',
      PRIMARY KEY  (`ID`)
    ) ENGINE=InnoDB DEFAULT CHARSET=latin1
    1 row in set (0.00 sec)
```

Creating New Tables from Existing Tables

- MySQL provides two ways to create a new table based on an existing table:
  - Use `CREATE TABLE` with `SELECT` to create a new table that stores the result set returned by the `SELECT` query.
    - You can use almost any type of `SELECT` statement and reference multiple tables with joins and unions.
  - `Use CREATE TABLE` with `LIKE` to create a table with the same structure as the one specified.
    - Does not keep foreign key assignments
    - Does not copy any data

Creating a Table from Data in Existing Tables

- Use `CREATE TABLE` with `SELECT`. With this approach:
  - You do not have to create the table structure first and then populate it with data.
    - The statement creates the table based on the `SELECT` query results.
  - You can use multiple tables with joins, subqueries, or unions in the `SELECT` statement, or even a `SELECT` statement without any table.
  - Your new table might not retain all the features of the original:
    - No table options are copied, including indexes and constraints
    - The data types might be different, particularly if the column is based on an expression

Examples: CREATE TABLE with SELECT

- The following examples create new tables based on the `City` table. The new tables contain:
  - All rows from the original `City` table:

```
CREATE TABLE CitySelect1 SELECT * FROM City;
```

  - All cities with a population of over 2,000,000:

```
CREATE TABLE CitySelect2 SELECT * FROM City
                         WHERE Population > 2000000;
```

  - Only the `Name` and `CountryCode` columns:

```
CREATE TABLE CitySelect3 SELECT Name, CountryCode
                         FROM City;
```

  - An empty copy of the table:

```
CREATE TABLE CitySelect4 SELECT * FROM City LIMIT 0;
```

Copying an Existing Table Structure

- Use `CREATE TABLE` with `LIKE`. With this approach:
  - Indexes and column options are preserved in the new table
  - No row data is copied

```
mysql> CREATE TABLE CityLike LIKE City;
Query OK, 0 rows affected (0.05 sec)
```

  - The new table is empty

Comparing SELECT with LIKE Results

```
mysql> CREATE TABLE CitySelect
         SELECT * FROM City;

Query OK, 4079 rows affected (1.44 sec)
Records: 4079  Duplicates: 0  Warnings:
0

mysql> SHOW CREATE TABLE CitySelect\G
*************** 1. row ***************
       Table: CitySelect
Create Table: CREATE TABLE `CitySelect`
(
  `ID` int(11) NOT NULL DEFAULT '0',
  `Name` char(35) NOT NULL DEFAULT '',
  `CountryCode` char(3) NOT NULL
DEFAULT '',
  `District` char(20) NOT NULL DEFAULT
'',
  `Population` int(11) NOT NULL DEFAULT
'0'
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

```
mysql> CREATE TABLE CityLike
         LIKE City;

Query OK, 0 rows affected (0.49 sec)

mysql> SHOW CREATE TABLE CityLike\G
*************** 1. row ***************
       Table: CityLike
Create Table: CREATE TABLE `CityLike` (
  `ID` int(11) NOT NULL AUTO_INCREMENT,
  `Name` char(35) NOT NULL DEFAULT '',
  `CountryCode` char(3) NOT NULL
DEFAULT '',
  `District` char(20) NOT NULL DEFAULT
'',
  `Population` int(11) NOT NULL DEFAULT
'0',
  PRIMARY KEY (`ID`),
  KEY `CountryCode` (`CountryCode`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

Using Temporary Tables

- Use temporary tables to improve the efficiency of queries that repeatedly access large amounts of data in multiple tables. A temporary table:
  - Is created by the `CREATE TEMPORARY TABLE` statement
  - Is visible only to the client that created and kept it
  - Is kept only until the client disconnects
  - Does not conflict with other clients using the same table name
  - Overrides an existing table until the temporary table is dropped
  - Does not appear in `SHOW TABLES` or `INFORMATION_SCHEMA.TABLES`.

Example: Creating a Temporary Table

The following SQL creates a temporary table from the existing `City` table. The temporary table contains the names of all the cities in the state of Texas (USA).

```
mysql> CREATE TEMPORARY TABLE Texas
    -> SELECT Name FROM City WHERE District='Texas';
Query OK, 26 rows affected (0.01 sec)
Records: 26  Duplicates: 0  Warnings: 0
```

Modifying Table Data

Inserting Data in a Table

- Populate a table with row data by using the

```
INSERT INTO table_name (<column_list>)
    VALUES (<value_list>)
```

  - **table_name:** Table to which you add row data
  - **column_list:** Column names from the specified table
  - **value_list:** Value expressions for columns
- Example:

```
INSERT INTO City (ID, Name, CountryCode)
    VALUES (NULL, 'Essaouira', 'MAR'),
    (NULL, 'Sankt-Augustin', 'DEU');
```

Inserting Data in a Table: INSERT with SET

- Use `INSERT` with the `SET` option to indicate column names and values. For example:

```
INSERT INTO City SET ID=NULL, Name='Essaouira',
    CountryCode='MAR';
INSERT INTO City SET ID=NULL,
    Name='Sankt-Augustin', CountryCode='DEU';
```

- Content of the new rows:

```
+------+---------------+-------------+----------+------------+
| ID   | Name          | CountryCode | District | Population |
+------+---------------+-------------+----------+------------+
| 4080 | Essaouira     | MAR         |          |          0 |
| 4081 | Sankt-Augustin | DEU         |          |          0 |
+------+---------------+-------------+----------+------------+
```

Inserting Data in a Table: INSERT with SELECT

- Use `INSERT` with the `SELECT` option to copy rows from an existing table, or (temporarily) to store a

```
INSERT INTO table_name (<column_list>)
    (<query_expression>)
```

  - **`table_name` and `column_list`:** Work the same way as they do for `INSERT...VALUES` statements
  - **`query_expression`:** Must produce exactly as many columns as specified by `column_list`
- Example:

```
INSERT INTO Top10Cities (ID, Name, CountryCode)
    SELECT ID, Name, CountryCode FROM City
            ORDER BY Population DESC LIMIT 10;
```

Retrieving the AUTO_INCREMENT Value of the Last Insertion

- The MySQL-specific `SELECT LAST_INSERT_ID()` statement retrieves the last `AUTO_INCREMENT` value.

```
mysql> INSERT INTO City (Name, CountryCode)
    -> VALUES ('Sarah City', 'USA');
Query OK, 1 row affected (0.0 sec)

mysql> SELECT LAST_INSERT_ID();
+------------------+
| LAST_INSERT_ID() |
+------------------+
| 4080             |
+------------------+
```

Retrieving the ID of the Last Insert by Using PHP/PDO

- Use the PDO database connection object's `lastInsertId()` method to retrieve the most recent `AUTO_INCREMENT` value.

```
$dbh->exec ("INSERT INTO City (Name, CountryCode)
                    VALUES ('Waterford', 'IRL')");
$last_id = $dbh->lastInsertId();
```

Retrieving the ID of the Last Insert by Using Connector/J

- Using the `Statement` or `PreparedStatement` object:
  1. Include the `Statement.RETURN_GENERATED_KEYS` argument in the call to `executeUpdate()`
- Example (using `Statement`):

```
Statement s = conn.createStatement ();

s.executeUpdate ("INSERT INTO City (Name, CountryCode)"

                 + VALUES ('Waterford', 'IRL'),

                 Statement.RETURN_GENERATED_KEYS);
```

Retrieving the ID of the Last Insert by Using Connector/J

2. Call `getGeneratedKeys()` to retrieve the sequence value

- Example (using `Statement`):

```
long last_id;
ResultSet rs = s.getGeneratedKeys ();
if (rs.next ())
{
  last_id = rs.getLong (1);
}
else
{
  throw new SQLException
    ("getGeneratedKeys() produced no value");
}
rs.close ();
s.close ();
```

Retrieving the Last Insert ID by Using Connector/Python

- Use the `lastrowid` cursor object attribute.
- Example:

```
cursor = conn.cursor()

cursor.execute('''
               INSERT INTO City (Name, CountryCode)
               VALUES ('Waterford', 'IRL')
               ''')

last_id = cursor.lastrowid
```

DELETE Statement

- Use the `DELETE` statement to remove entire rows:

```
DELETE FROM table_name
[WHERE condition]
[ORDER BY ...]
[LIMIT row_count]
```

- Filter the rows to be deleted by using the `WHERE` clause:

```
DELETE FROM CountryLanguage WHERE IsOfficial='F';
```

- Omit the `WHERE` clause to delete all rows from the table.
- Use `DELETE` with extreme caution because it does not have the “undo” feature.
  - Consider setting the `--safe-updates` option.

DELETE with ORDER BY and LIMIT: Descending

- Control the order and count of `DELETE` by using `ORDER BY` and `LIMIT`.
- Example using rows sorted in descending order:

```
DELETE FROM CountryLanguage
WHERE Language = 'MySQL'
ORDER BY CountryCode DESC
LIMIT 1;
```

- Effect on the `CountryLanguage` table:

```
+-------------+----------+------------+------------+
| CountryCode | Language | IsOfficial | Percentage |
+-------------+----------+------------+------------+
| BEL         | MySQL    | F          |        0.0 |
| FJI         | MySQL    | F          |        0.0 |
| GRL         | MySQL    | F          |        0.0 |
+-------------+----------+------------+------------+
```

Removed

DELETE with ORDER BY and LIMIT: Ascending

- Example using rows sorted in ascending order:

```
DELETE FROM CountryLanguage
WHERE Language = 'MySQL'
ORDER BY CountryCode ASC
LIMIT 1;
```

- Effect on the `CountryLanguage` table:

```
+-------------+----------+------------+------------+
| CountryCode | Language | IsOfficial | Percentage |
+-------------+----------+------------+------------+
| BEL         | MySQL    | F          |        0.0 |
| FJI         | MySQL    | F          |        0.0 |
| GRL         | MySQL    | F          |        0.0 |
+-------------+----------+------------+------------+
```

Removed

UPDATE Statement

- Modifies the contents of existing rows
- Is used with the `SET` clause for column assignments

- General syntax:

```
UPDATE table_name SET column=expression
[,column=expression,...]
[WHERE condition] [other_clauses]
```

- Example:

```
mysql> UPDATE Country
    -> SET Population = Population * 2,
    ->     Region = 'Dolphin Country'
    -> WHERE Code = 'SWE';
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

UPDATE Statement: Column Assignment Effects

- Effects are subject to column constraints:
  - When you try to update a column to a value that does not match the column definition, the value is converted or truncated by the server.
- `UPDATE` has no effect under the following conditions:
  - When it matches no rows for updating:
    - Due to an empty table
    - If no rows match the `WHERE` clause
  - When it does not actually *change* any column values:
    - When the given value is the same as the existing value
    - If there is no row in the table that contains the specified key values

UPDATE Statement: Order of Affected Rows

- There is no guarantee of the order in which rows are updated. This can result in errors such as the following:

```
mysql> UPDATE City SET ID = ID+1;
ERROR 1062 (23000): Duplicate entry '2' for key 'PRIMARY'
```

- Use `ORDER BY` to control the order. For example:

```
mysql> UPDATE City SET ID = ID+1
    -> ORDER BY ID DESC;
Query OK, 4079 rows affected (0.13 sec)
Rows matched: 4079 Changed: 4079 Warnings: 0
```

  - Table contents after update (ordered by `ID`):

```
+----+---------------+-------------+--------------+------------+
| ID | Name          | CountryCode | District     | Population |
+----+---------------+-------------+--------------+------------+
|  2 | Kabul         | AFG         | Kabol        |    1780000 |
|  3 | Qandahar      | AFG         | Qandahar     |     237500 |
|  4 | Herat         | AFG         | Herat        |     186800 |
...
```

UPDATE with LIMIT

- Use `LIMIT` to control the number of rows updated. For example:

```
mysql> UPDATE City SET ID = ID-1 LIMIT 1;
```

  - Contents of the affected row:

```
+----+---------------+-------------+--------------+------------+
| ID | Name          | CountryCode | District     | Population |
+----+---------------+-------------+--------------+------------+
|  1 | Kabul         | AFG         | Kabol        |    1780000 |
|  3 | Qandahar      | AFG         | Qandahar     |     237500 |
|  4 | Herat         | AFG         | Herat        |     186800 |
...
```

Note that only the first `ID` was affected by the update. The other IDs kept their existing values.

- Combine `ORDER BY` and `LIMIT` for more control over updates.

REPLACE Statement

- Performs the same as `INSERT` *except* that when an old row in the table has the same value as a new row for a `PRIMARY KEY` or `UNIQUE` constraint, the old row is deleted before the new row is inserted.
- Is a MySQL extension to the SQL standard
- General syntax:

```
REPLACE INTO table_name (<column_list>)
    VALUES (<value_list>)
```

- Example:

```
REPLACE INTO CountryLanguage (CountryCode, Language,
                              Percentage)
VALUES ('ALB', 'Albaniana', 78.1);
```

REPLACE Results

- Example:

```
Query OK, 2 rows affected (0.06 sec)
```

- Returns a count to indicate the number of rows affected
  - The count is the sum of the rows deleted and inserted.
  - Count of `1`: One row was inserted and no rows were deleted.
  - Count greater than `1`: One or more old rows were deleted before the new row was inserted.
- Row count makes it easy to determine if `REPLACE` only added a row or if it also replaced rows.

REPLACE Algorithm

- MySQL uses the following algorithm for `REPLACE`:
  1. Try to insert the new row in the table.
  2. When insertion fails because of a duplicate key error (for a primary key or unique constraint), delete the conflicting row that has the duplicate key value from the table.
  3. Try again to insert the new row in the table.

Insertion of Duplicate Rows

- Use `ON DUPLICATE KEY UPDATE` with `INSERT` to prevent a duplicate row error.
  - The behavior of `ON DUPLICATE KEY UPDATE` is similar to `REPLACE` except for the following:
    - `REPLACE`
      - Discards the *existing* row
      - Adds the *new* row to the table
    - `ON DUPLICATE KEY UPDATE`
      - Discards the *new* row
      - Preserves the *existing* row
  - This option is specific to MySQL.

Removing Table Rows by Truncating

- Use the `TRUNCATE TABLE` (or just `TRUNCATE`) statement to remove **all** rows from a table.
  - General syntax:

```
TRUNCATE [TABLE] <table_name>;
```

  - Comparing `DELETE` to `TRUNCATE TABLE`:

DELETE	TRUNCATE TABLE
Can delete specific rows with `WHERE`	Deletes *all* rows; no filtering is possible
Is usually slower than `TRUNCATE TABLE`	Is usually faster than `DELETE`
Returns a true row count	Can return a row count of zero
Can be rolled back as part of a transaction	Cannot be rolled back in a transaction
Does not reset the `AUTO_INCREMENT` sequence	Can reset the sequence of `AUTO_INCREMENT` values

Transactions

What Is a Transaction?

- A set of data manipulation tasks treated as a single unit of work
- Takes the database from one consistent state to another consistent state

Nontransactional Process

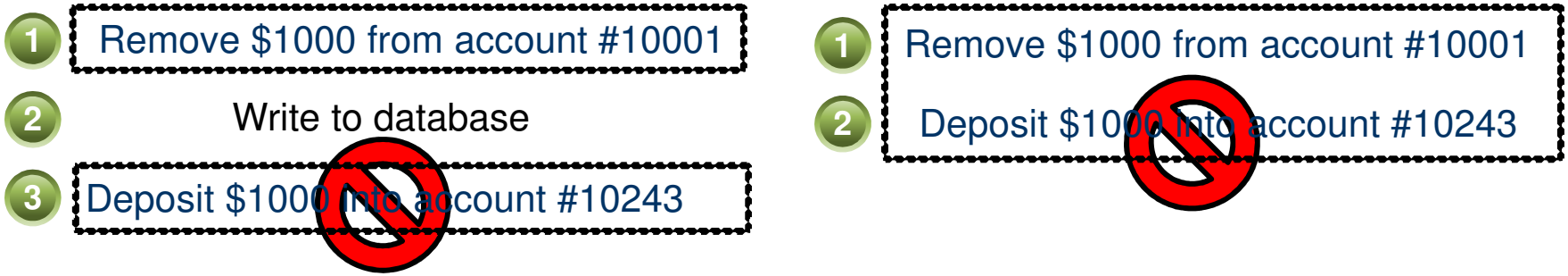
1. Remove $1000 from account #10001
2. Write to database
3. Deposit $1000 into account #10243
4. Write to database

Transactional Process

1. Remove $1000 from account #10001
2. Deposit $1000 into account #10243
3. Write to database

Handling Partial Operations

- All of the data manipulation steps must be carried out successfully. If any step fails, the system must:
  - Permanently retain those operations that succeeded completely
  - Disregard those operations containing steps that did not succeed



The ACID Properties of Transactions

- Transactional systems are **ACID** compliant, where "ACID" stands for:
  - **Atomic:** All the statements execute successfully or are canceled as a unit. A transaction is "all-or-nothing."
  - **Consistent:** A transaction changes the database from one consistent data state to another.
  - **Isolated:** One transaction does not affect another. Each transaction locks the results of its individual steps until the whole transaction ends.
  - **Durable:** All the changes made by a transaction that ends successfully are recorded correctly in the database. Changes are not lost.

Transaction Control Statements

- Use the following statements to control transactions explicitly:
  - `START TRANSACTION` (or `BEGIN`): Commences a new transaction
  - `COMMIT`: Commits the current transaction, making its changes permanent
  - `ROLLBACK`: Rolls back the current transaction, canceling its changes
  - `SET AUTOCOMMIT`: Disables or enables the default `autocommit` mode for the current connection

autocommit Mode

- `autocommit` mode determines how and when new transactions are started.
  - Enabled (default):
    - A single SQL statement implicitly starts a new transaction.
    - All changes to a table take effect immediately.
    - You can execute multi-statement transactions using the `START TRANSACTION` statement, which temporarily disables `autocommit` until you execute `COMMIT` or `ROLLBACK`.
  - Disabled:
    - Transactions span multiple statements by default.
    - You must explicitly `COMMIT` a transaction to persist its effect or `ROLLBACK` to cancel it.

Controlling autocommit Mode

- The server variable `autocommit` specifies whether `autocommit` mode is enabled.
  - By default, `autocommit` is enabled for new sessions.
  - Disable `autocommit` in the current session with:

```
mysql> SET autocommit = OFF;
```

  - Disable `autocommit` globally in an options file:

```
[mysqld]
autocommit=0
```

  - Disable `autocommit` globally with a `mysqld` startup

```
# mysqld --autocommit=0
```

Statements Causing an Implicit COMMIT

- The `COMMIT` statement explicitly persists the effects of the current transaction to the database.
- Many statements commit transactions implicitly, including:
  - Transaction control statements such as `START TRANSACTION` and `SET autocommit = ON;`
  - Data Definition Language statements (DDL), such as `ALTER`, `CREATE`, `DROP`, and `TRUNCATE TABLE`.
  - Data access and user management statements, such as `GRANT`, `REVOKE`, `SET PASSWORD`
  - Locking statements, such as `LOCK TABLES`, `UNLOCK TABLES`
- Statements that cause an implicit `COMMIT` cannot be rolled back.

General Approach to Coding Transactional Statements

- All MySQL APIs support transactions.
  - Many APIs provide an abstracted interface for working with transactions.
    - You can see the resulting SQL statements by enabling and then examining the general query log.
  - With others, you must construct the transactional SQL statements yourself.
- Within the code that executes your transaction:
  - Use your programming language's ability to handle exceptions, if it exists
    - Execute the transaction statements in a block.
    - Raise an exception and roll back if statements in the block fail.
  - Keep the amount of tests required to determine if a rollback is needed to a minimum.

Coding Transactional Statements in PHP/PDO

```
01  try
02  {
03     $dbh->beginTransaction ();
04     $dbh->exec ("UPDATE Country SET GNP = (GNP * 1.1) WHERE Name =

                   'Armenia'");
05     $dbh->exec ("UPDATE Country SET GNP = (GNP * 1.1) WHERE Name =

                   'Bermuda'");
06     $dbh->commit ();
07  }
08  catch (Exception $e)
09  {
10     print ("Transaction failed, rolling back. Error was:\n");
11     print ($e->getMessage () . "\n");
12     try
13     {
14        $dbh->rollback ();
15    }
16    catch (Exception $er) {
17              print ("Unable to roll back. Error was:\n");
18       print ($er->getMessage () . "\n");
19    }
20  }
```

Coding Transactional Statements in Connector/J

```
01  try {
02     conn.setAutoCommit (false);
03     Statement s = conn.createStatement ();
04     s.executeUpdate ("UPDATE Country SET GNP = (GNP * 1.1) WHERE Name

            = 'Armenia'");
05     s.executeUpdate ("UPDATE Country SET GNP = (GNP * 1.1) WHERE Name

            = 'Bermuda'");
06     s.close ();
07     conn.commit ();
08     conn.setAutoCommit (true);
09  }
10  catch (SQLException e) {
11     System.err.println ("Transaction failed, rolling back. Error: ");
12     printSQLErrorMessage (e);  // custom function
13     try  {
14       conn.rollback ();
15       conn.setAutoCommit (true);
16     }
17     catch (SQLException er) {
18        System.err.println ("Roll back failed. Error: ");
19       printSQLErrorMessage (er); // custom function
20     }
21  }
```

Coding Transactional Statements in Connector/Python

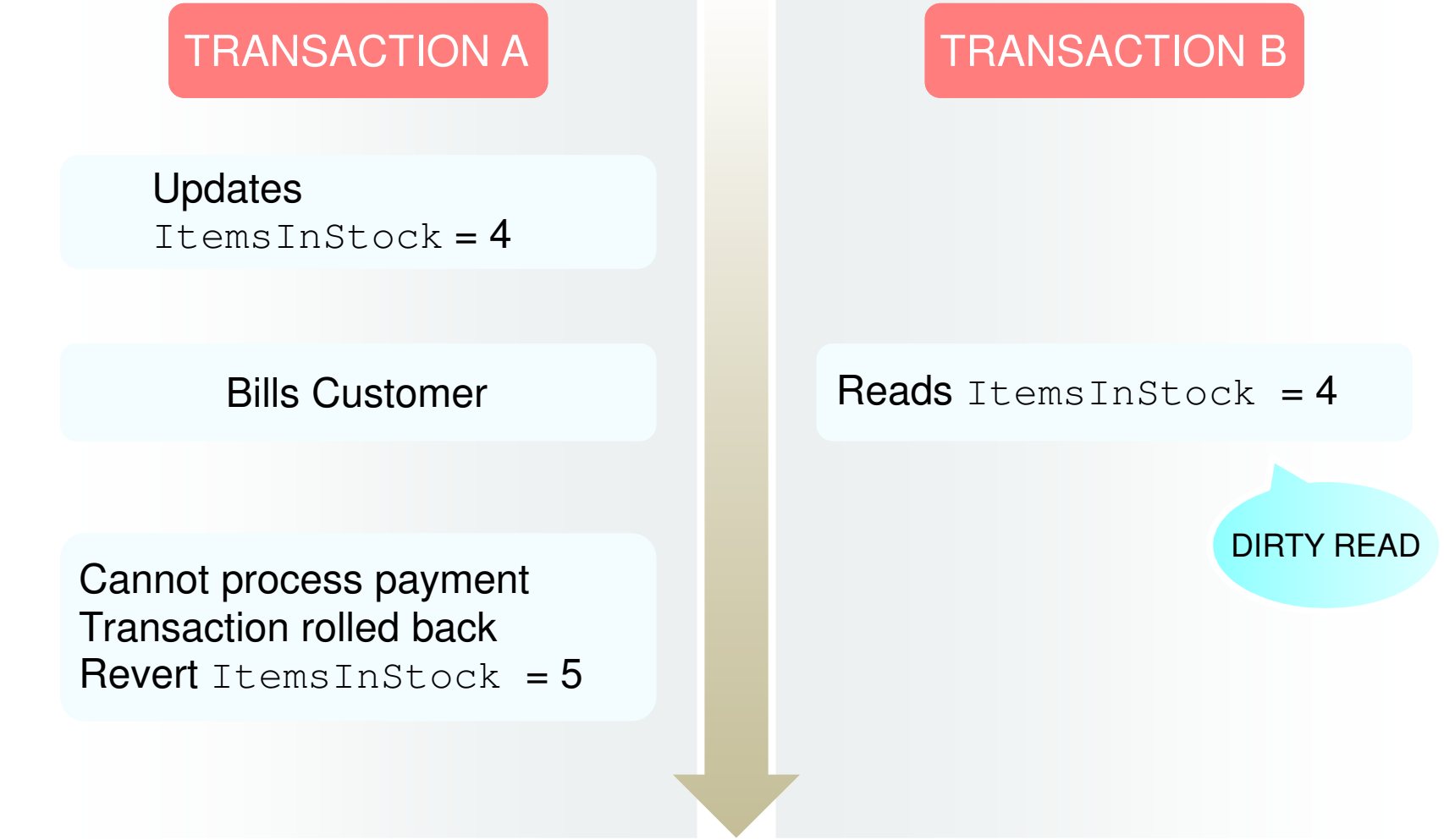
```
01    try:
02       cursor = conn.cursor()
03       cursor.execute("UPDATE Country SET GNP = (GNP * 1.1)
                                                    WHERE
      Name = 'Armenia'")
04       cursor.execute("UPDATE Country SET GNP = (GNP * 1.1)

      WHERE       Name = 'Bermuda'")
05       cursor.close()
06       conn.commit()
07    except mysql.connector.Error as e:
08       print("Transaction failed, rolling back. Error was:")
09       print(e)
10       try:
11         conn.rollback()
12       except mysql.connector.Error as er:
13         print("Unable to roll back. Error was:")
14         print(er)
```

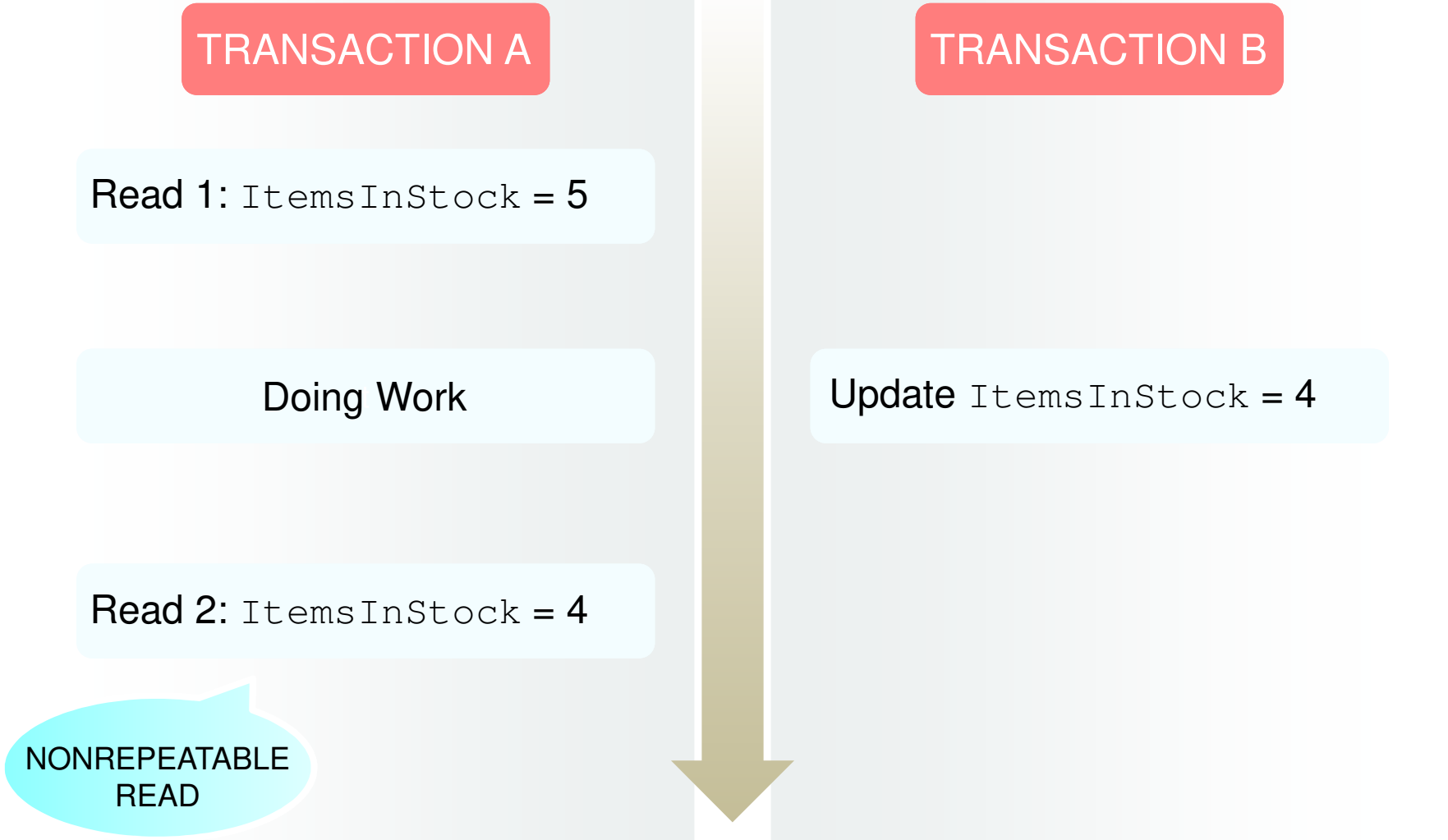
Consistency Issues

- When multiple clients concurrently access data from the same table, the following consistency issues can occur:
  - **Dirty reads:** A dirty read occurs when a transaction reads the changes made by another uncommitted transaction.
  - **Nonrepeatable reads:** A nonrepeatable read occurs when the same read operation yields different results when it is repeated at a later time within the same transaction.
  - **Phantom reads:** A phantom is a row that appears that was not previously visible within the same transaction.

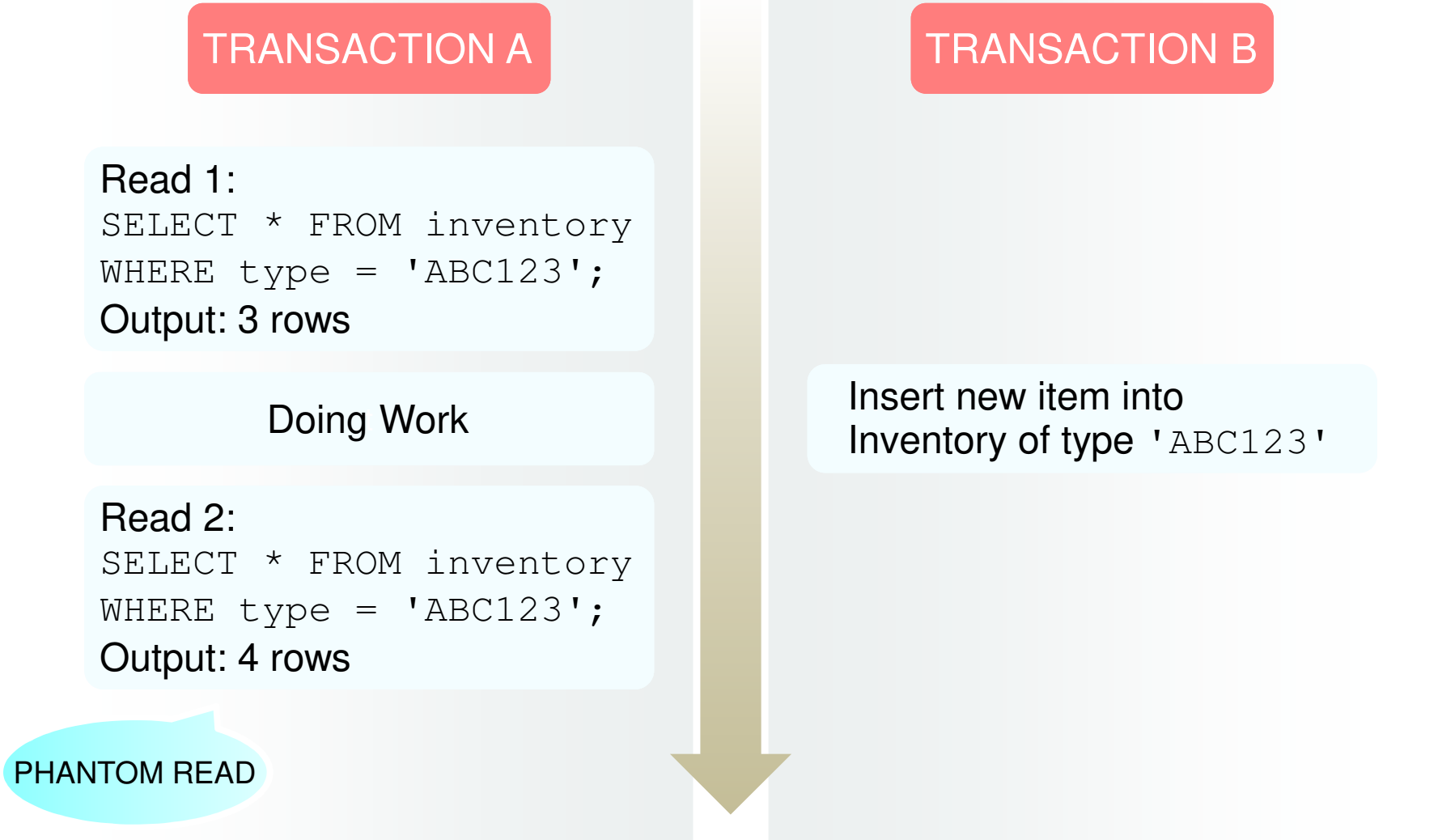
Dirty Read: Example



Nonrepeatable Read: Example



Phantom Read: Example



Isolation Levels

- **Isolation levels** control how changes made by transactions affect other transactions happening at the same time. This is made possible by *multiversioning*.
  - `READ UNCOMMITTED` allows a transaction to see uncommitted changes made by other transactions.
  - `READ COMMITTED` allows a transaction to see changes made by other transactions only if they have been committed.
  - `REPEATABLE READ` (the default) ensures that if a transaction issues the same `SELECT` query twice, it gets the same result both times, regardless of committed or uncommitted changes made by other transactions.
  - `SERIALIZABLE` completely isolates the effects of one transaction from another.

Resolving Consistency Issues with Isolation Levels

- The following table displays which consistency issues are possible with each isolation level.

Isolation Level	Dirty reads	Nonrepeatable reads	Phantom reads
`READ UNCOMMITTED`	Possible	Possible	Possible
`READ COMMITTED`	Not possible	Possible	Possible
`REPEATABLE READ*`	Not possible	Not possible	Not possible
`SERIALIZABLE`	Not possible	Not possible	Not possible

- * The default isolation level is `REPEATABLE READ`.

Setting Isolation Levels

- The transaction isolation level is a server option you can set at startup, or as a session or global option.

```
[mysqld]
transaction-isolation = READ-COMMITTED
```

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

startup:

Displaying the Current Isolation Level

- Examine the value of the `tx_isolation` server variable
  - `@@session.tx_isolation` or `@@session.tx_isolation`: The isolation level for the current connection
  - `@@global.tx_isolation`: The isolation level for all connections to the server

```
mysql> SELECT @@tx_isolation;
+-----------------+
| @@tx_isolation  |
+-----------------+
| REPEATABLE-READ |
+-----------------+
1 row in set (#.## sec)
```

Locking

- Locking prevents problems that can occur when two or more clients access the same data at the same time.
  - The lock allows access to data by the client that holds the lock, but limits what other clients can do with that data.
  - MySQL supports two types of lock:
    - **Shared:** If a client wants to read data, other clients that want to read the same data do not cause a conflict, and they all can read at the same time. However, another client that wants to write (modify) data must wait until the read has finished.
    - **Exclusive:** If a client wants to write data, all other clients must wait until the write has finished before reading or writing.

Locking Reads

- MySQL has two locking modifiers that you can add to the end of `SELECT` statements:
  - `LOCK IN SHARE MODE`: This clause locks each selected row with a shared lock. This means that no other transactions can take exclusive locks, but other transactions can use shared locks. Because read operations do not lock rows, they are not affected by the locks.
  - `FOR UPDATE`: This clause locks each selected row with an exclusive lock. It prevents others from acquiring any lock on the rows, but allows reading the rows.

SELECT...LOCK IN SHARE MODE: Example

- **Problem:** Adding a new Australian city in the `world` database:
  1. You issue a `SELECT` query to verify that the country code for Australia (`AUS`) exists in the `Country` table.
  2. You insert a new row in the `City` table with a country code of `AUS`.
  3. Another user deletes AUS from the country table before your new `City` row is inserted.
  4. Your `City` row is now orphaned, because its parent country no longer exists.
- **Solution:** Execute the initial `SELECT` of the `Country` table using `LOCK IN SHARE MODE`:

```
mysql> SELECT Code FROM Country WHERE Name='Australia'
    -> LOCK IN SHARE MODE;
```

SELECT...FOR UPDATE: Example

- **Problem:** You are manually maintaining an integer counter column in a table called `CityCodes`, which you use to assign a unique identifier to each city added to the `City` table.
  - To calculate the next available code, you read the current code and add 1 to its value.
  - In a concurrent environment, two clients could retrieve the same value for the counter simultaneously and use that value to construct the next code in the sequence, resulting in a duplicate key error.
- **Solution:** Execute the initial `SELECT` of the `CityCodes` table using `FOR UPDATE`:

```
mysql> SELECT counter INTO @@counter FROM CityCodes
    -> FOR UPDATE;
```

View

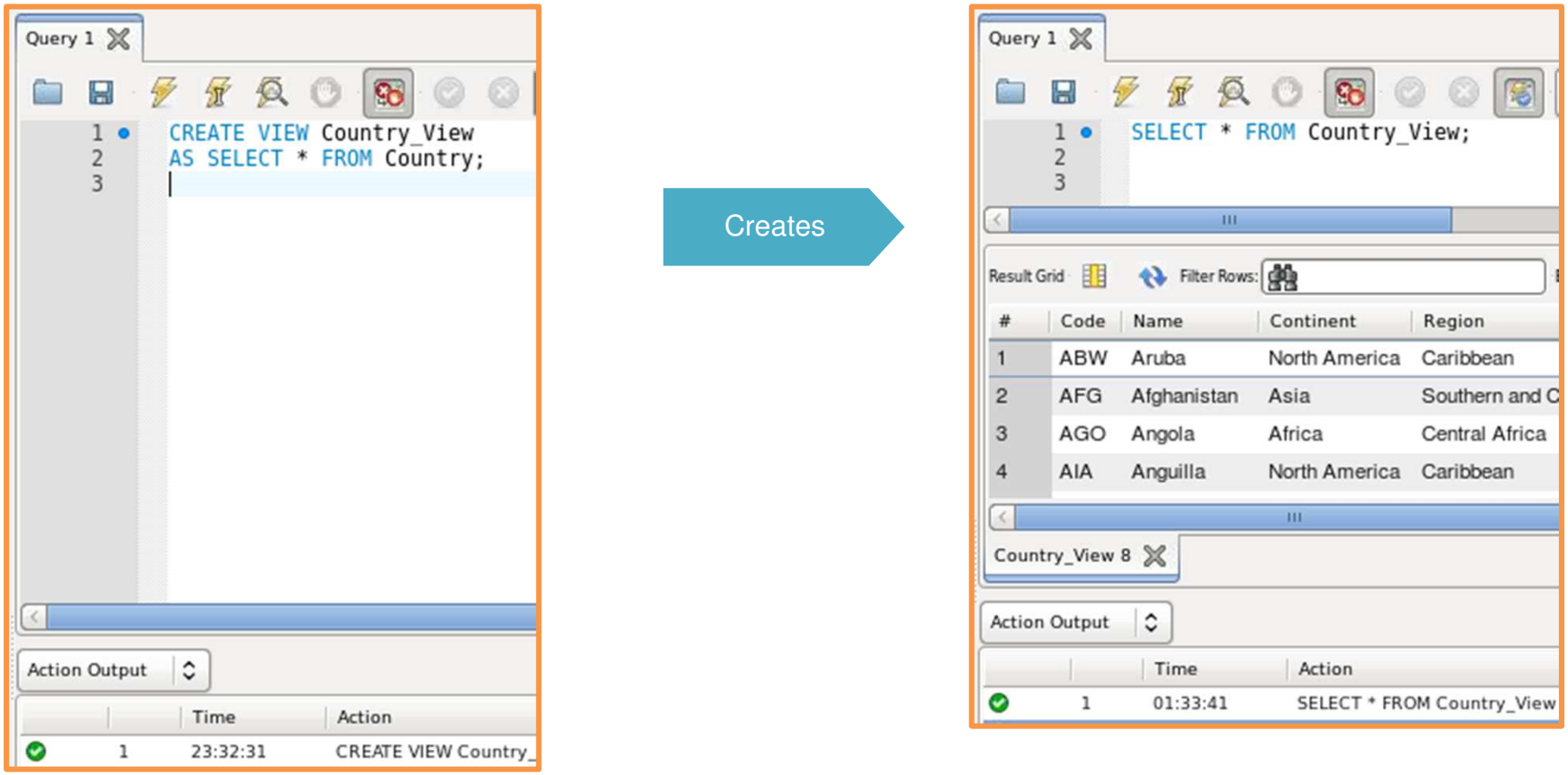
Creating Views to Reuse Queries

- A view is a virtual table defined by the result of a `SELECT` query.
  - A custom view of your tables makes some operations simpler.
  - Views provide additional benefits compared to selecting data directly from base tables.
  - You can include regular tables or other views in your defining `SELECT` statement.
  - Creation syntax:

```
CREATE VIEW view_name [column_list]
    AS SELECT_expression
```

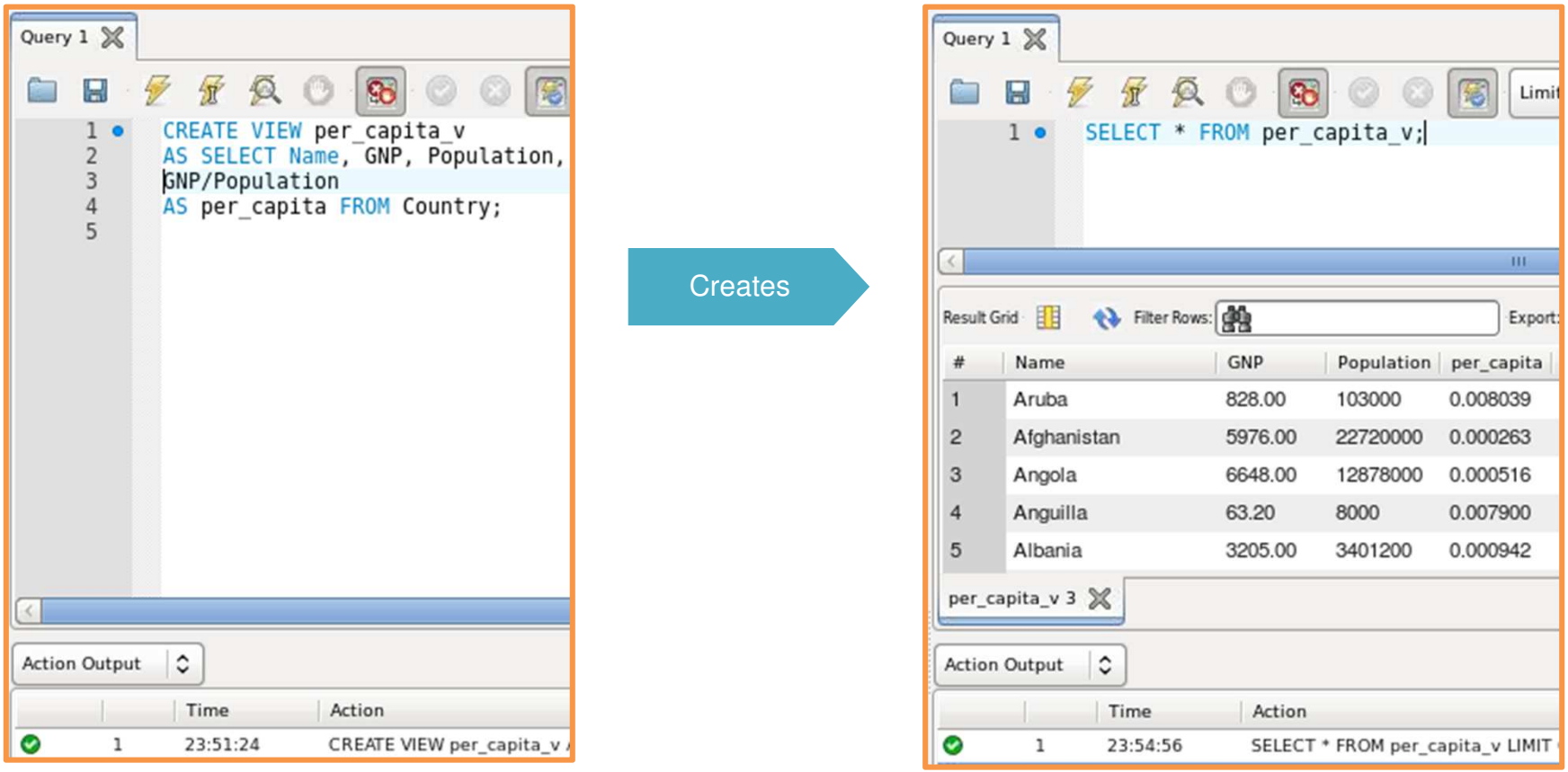
CREATE VIEW: Basic Example

This example creates a view containing the entire contents of the Country table:



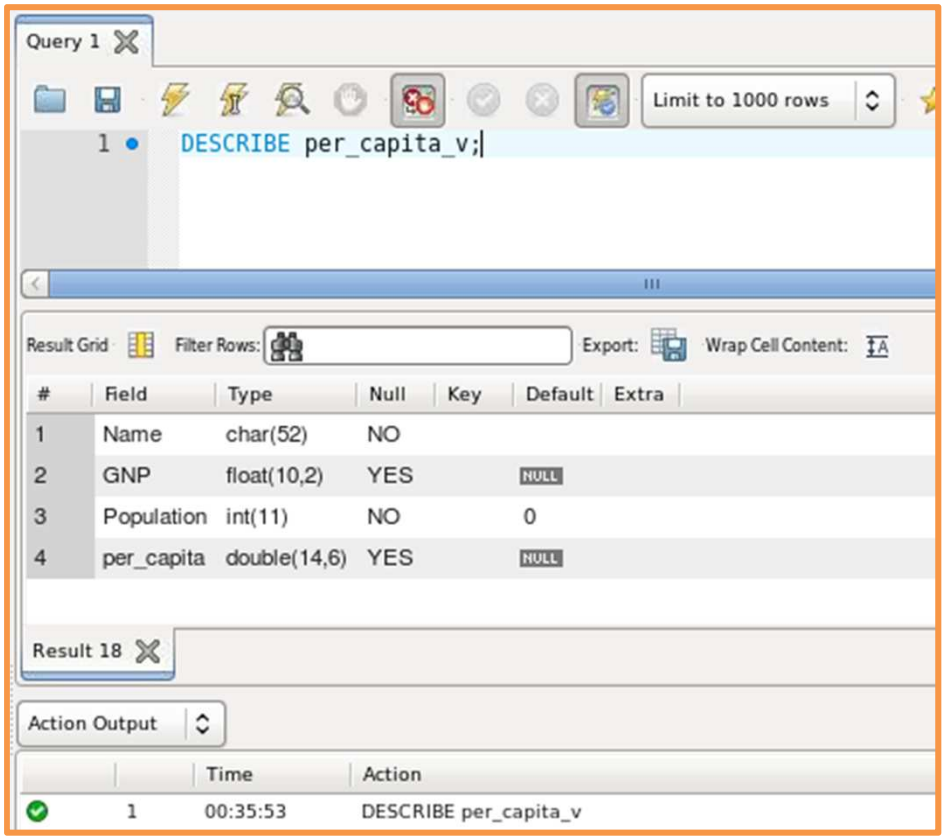
CREATE VIEW: More Complex Example

More complex example, using computed values:



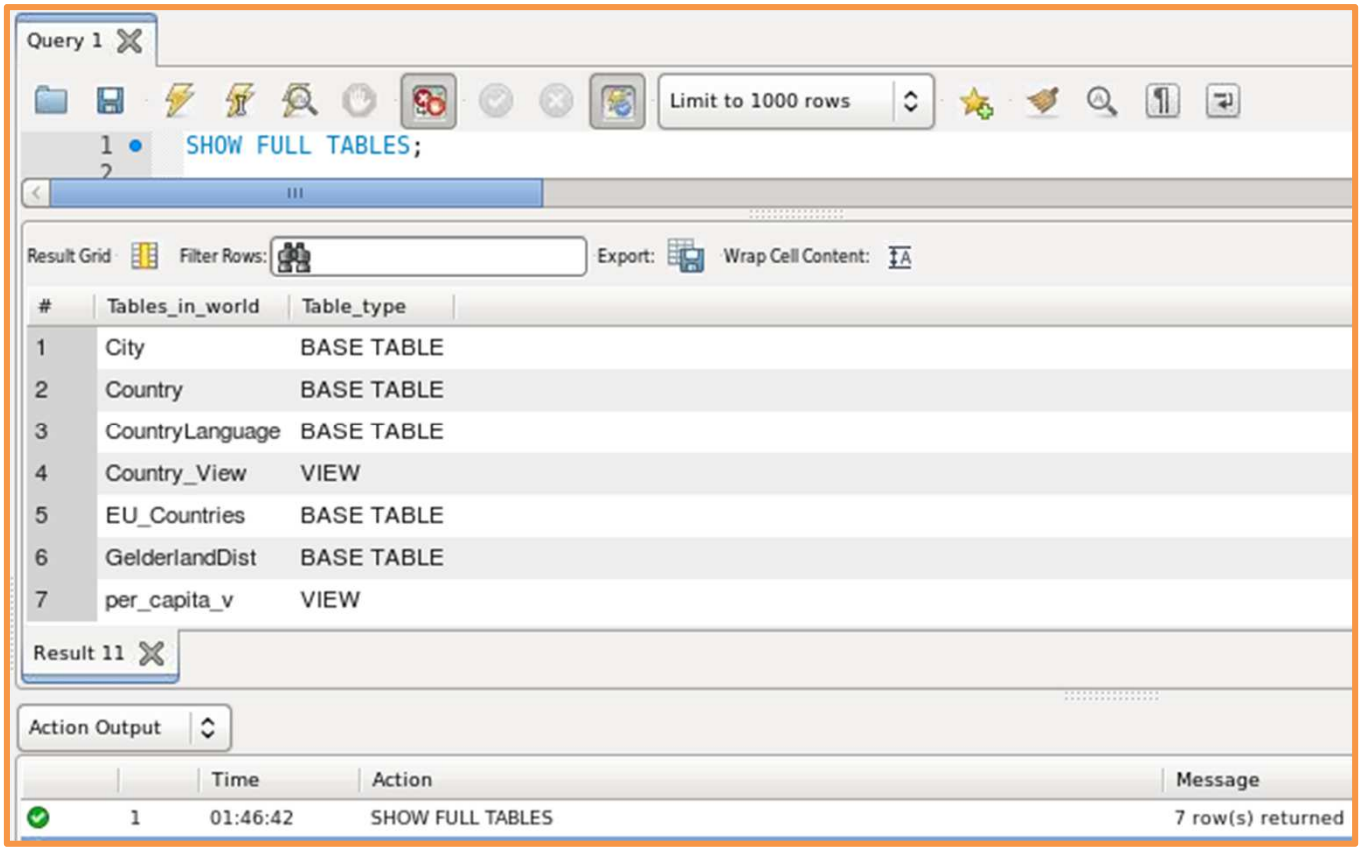
Displaying View Information

- Base table statements, like `DESCRIBE`, also work for views.
- Example using **`DESCRIBE view_name`**:



Views: Showing Table Types

- You can use **`SHOW FULL TABLES`** to differentiate base tables and views.



Querying Data from an Application

- Example of an application querying and displaying data:

```
# php samplephp.php
City              District       Population
New York          New York       8008278
Buffalo           New York       292648
Rochester         New York       219773
Yonkers           New York       196086
Syracuse          New York       147306
Albany            New York       93994
```

Query being executed by the application:

```
SELECT Name, District, Population FROM City
WHERE District = 'New York`
```

Connecting to MySQL from an Application

- PHP code to make a connection to a database:

```
$mysqli = new mysqli("localhost", "root", "MySQL5.7",
    "world");
if ($mysqli->connect_errno) {
    echo "Error connecting: ". $mysqli->connect_error;
}
```

Executing Queries in an Application

- PHP code to execute a query:

```
$res = $mysqli->query("SELECT Name, District,
    Population FROM City WHERE District = 'New York'");
if (! $res) {
    echo "Query failed: " . $mysqli->connect_error;
}
```

Displaying Result Sets in an Application

- PHP code to display the result set:

```
while ($row = $res->fetch_assoc()){
    echo $row['Name'] . "      \t"
           , $row['District'], "\t"
           . $row['Population'] . "\n";
}
```

Exporting Data as a Text File

- You can export data from a table to create a text file. Some reasons for exporting data:
  - Loading the data into another application, such as a spreadsheet, report writer, word processor, or other application that can process the data loaded from a text file
  - Copying databases from one server to another:
    - From within the same host
    - To another host
  - Testing a new MySQL release with real data
  - Transferring data from one RDBMS to another
  - Keeping a backup copy of a version of your data

Using SELECT … INTO OUTFILE to Create a Delimited File

- Write query results directly into a file by using **SELECT** with the **INTO OUTFILE** clause. **INTO OUTFILE** writes the results of the **SELECT** operation to the output file instead of returning them to the client.
  - Syntax:

```
SELECT column_list
INTO OUTFILE '/path/filename'
[format_options]
FROM table [select_clauses];
```

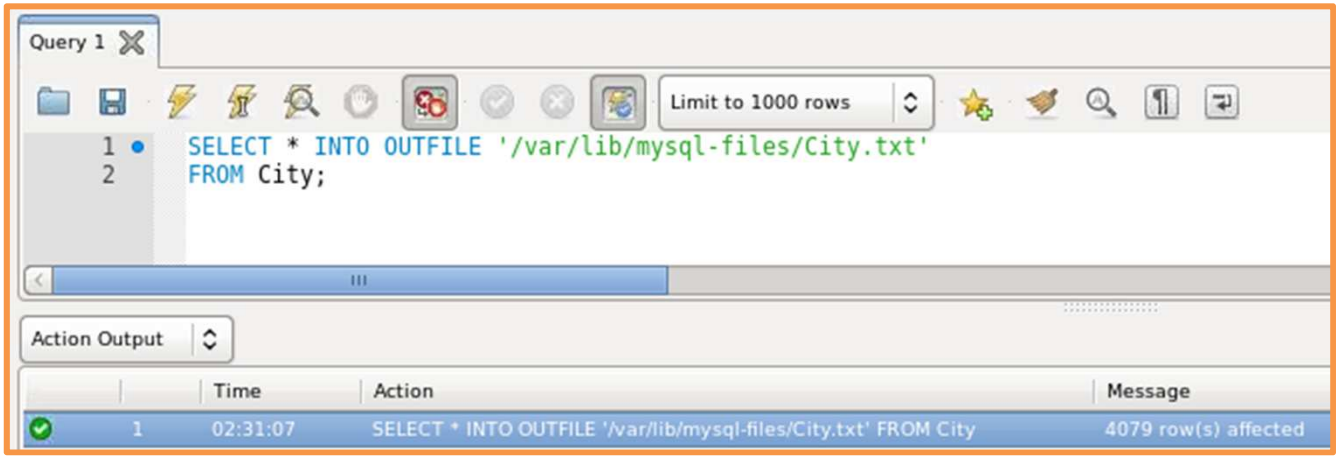
    - List the columns or specify * for all columns.
    - Specify the output file location on the host.
    - Provide the name of the table to export.

Exporting with a Query: INTO OUTFILE

- The process creates a new output file. It cannot already exist.
- Each line in the output file represents one row of data.
- Default file format:
  - Columns are delimited by tabs.
  - Lines are terminated by the newline character.
  - Null values are exported as `NULL`.
- Change the file format with the **`INTO OUTFILE`** format options.
  - Column delimiters (for example, comma or semicolon)
  - Line terminators (for example, carriage return)
  - Character strings enclosed in quotation marks (either " or ')

Example of SELECT … INTO OUTFILE

- Example of exporting with a query:



- Contents of the first few lines of the output file:

```
1           Kabul                AFG                 Kabol               1780000
2           Qandahar             AFG                 Qandahar            237500
3           Herat                AFG                 Herat               186800
4           Mazar-e-Sharif               AFG                 Balkh
    127800
5           Amsterdam                    NLD                 Noord-Holland
            731200
```

Format Options for SELECT … INTO OUTFILE

- Syntax for the format options:

```
SELECT column_list
INTO OUTFILE '/path/filename'
FIELDS TERMINATED BY 'delimiter'
[OPTIONALLY] ENCLOSED BY 'character'
LINES TERMINATED BY 'terminator'
FROM table;
```

  - Replace tab with a delimiter like comma (,) or semicolon (;).
  - Enclose data from each column in a character such as ' or ".
  - Optionally, enclose only fields with string data types in a character.
  - Change the line terminator to a different value.

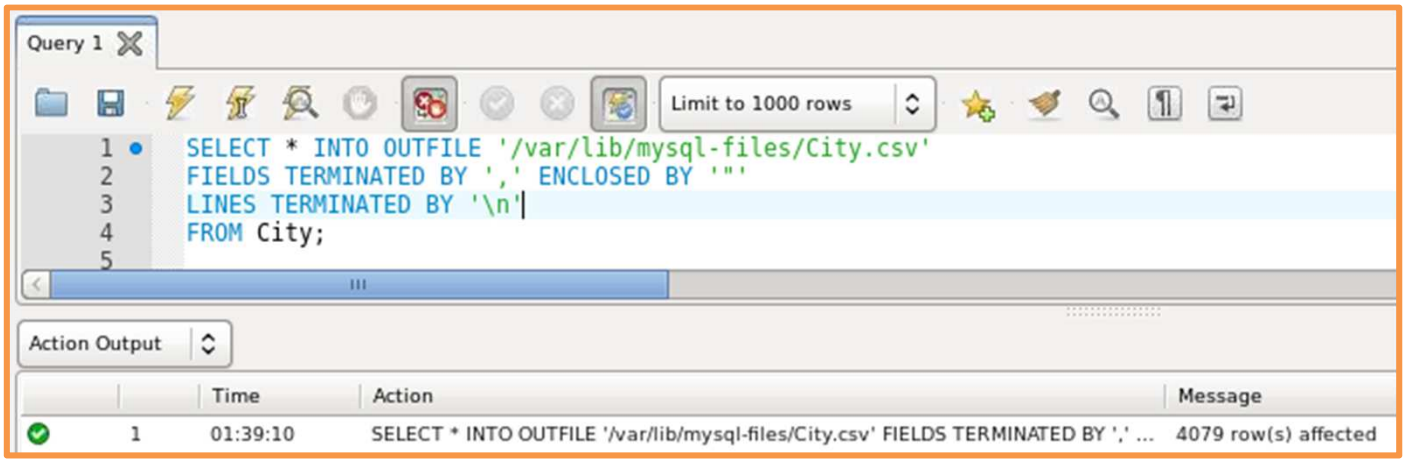
Exporting with a Query: CSV Format

- The comma-separated values (CSV) format:
  - Separates columns with commas
  - Encloses values in double quotation marks
  - Terminates lines with newline
- Syntax with CSV format options:

```
SELECT column_list
INTO OUTFILE '/path/filename'
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
FROM table;
```

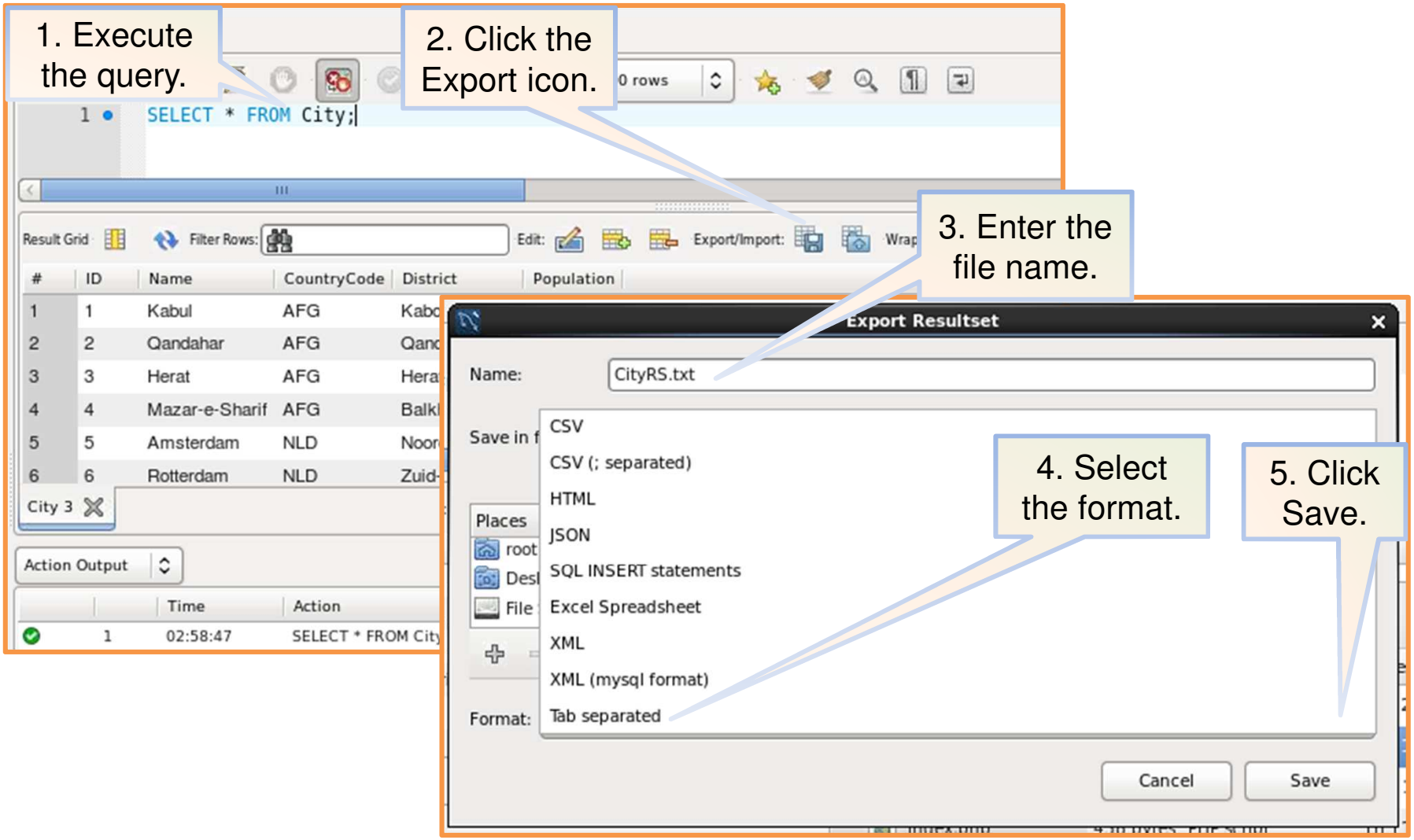
Example of SELECT INTO OUTFILE: CSV Format

- Example of exporting in CSV Format:



- Contents of the first few lines of the output file:
  - `"1","Kabul","AFG","Kabol","1780000"`
  - `"2","Qandahar","AFG","Qandahar","237500"`
  - `"3","Herat","AFG","Herat","186800"`
  - `"4","Mazar-e-Sharif","AFG","Balkh","127800"`

Exporting Data from MySQL Workbench Result Set in Tab-Separated Format

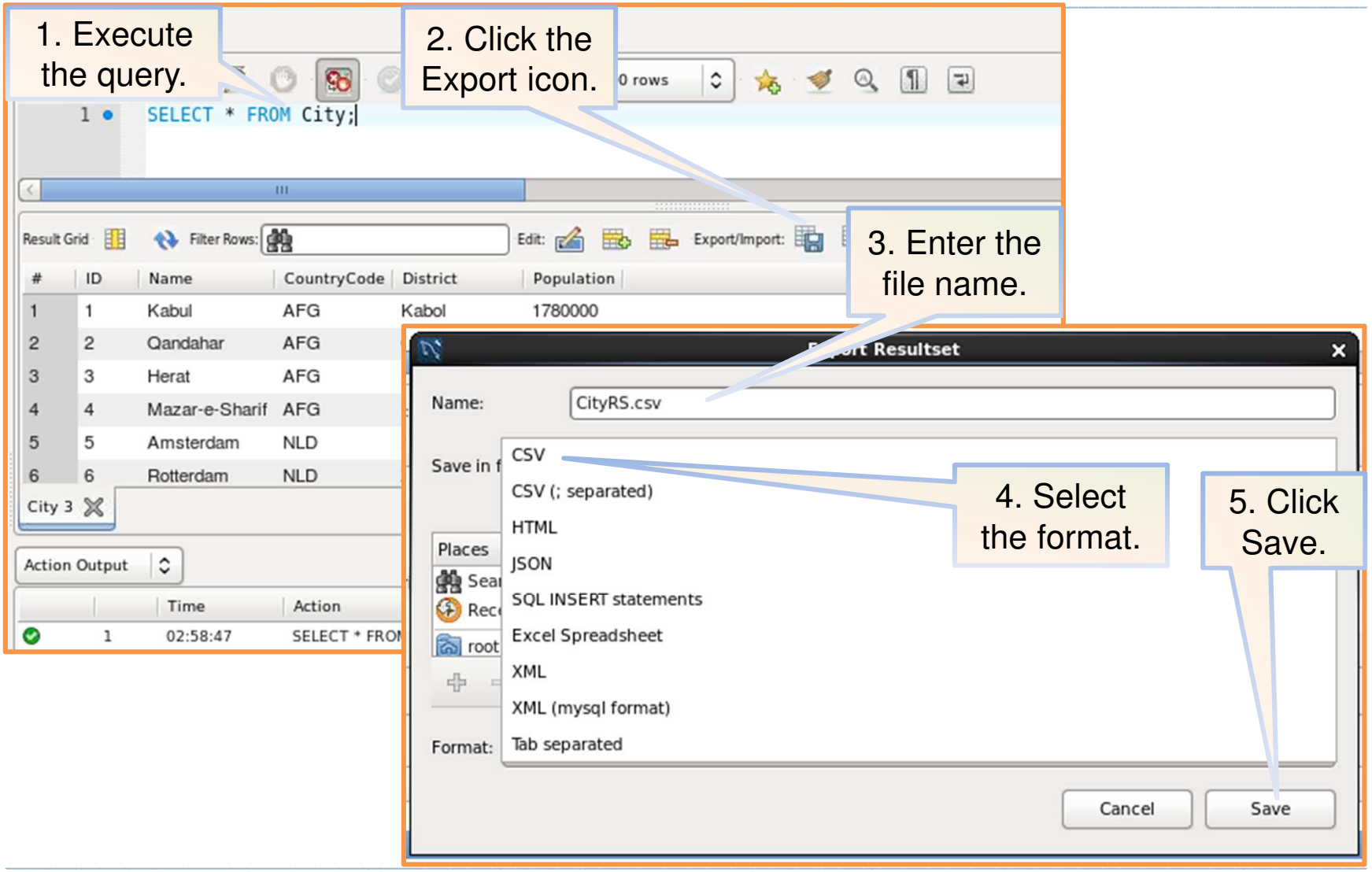


Contents of the Tab-Separated File After Exporting the Result Set of SELECT * FROM City;

- Contents of the output file:

```
ID  Name   CountryCode   District      Population
1   Kabul  AFG    Kabol  1780000
2   Qandahar      AFG    Qandahar      237500
3   Herat   AFG   Herat  186800
4   Mazar-e-Sharif      AFG    Balkh  127800
…
995       Probolinggo   IDN    "East Java"   120770
996       Cilegon       IDN    "West Java"   117000
997       Cianjur       IDN    "West Java"   114300
998       Ciparay       IDN    "West Java"   111500
999       Lhokseumawe   IDN    Aceh   109600
1000      Taman  IDN    "East Java"   107000
```

Exporting Data in CSV Format from the MySQL Workbench Result Set

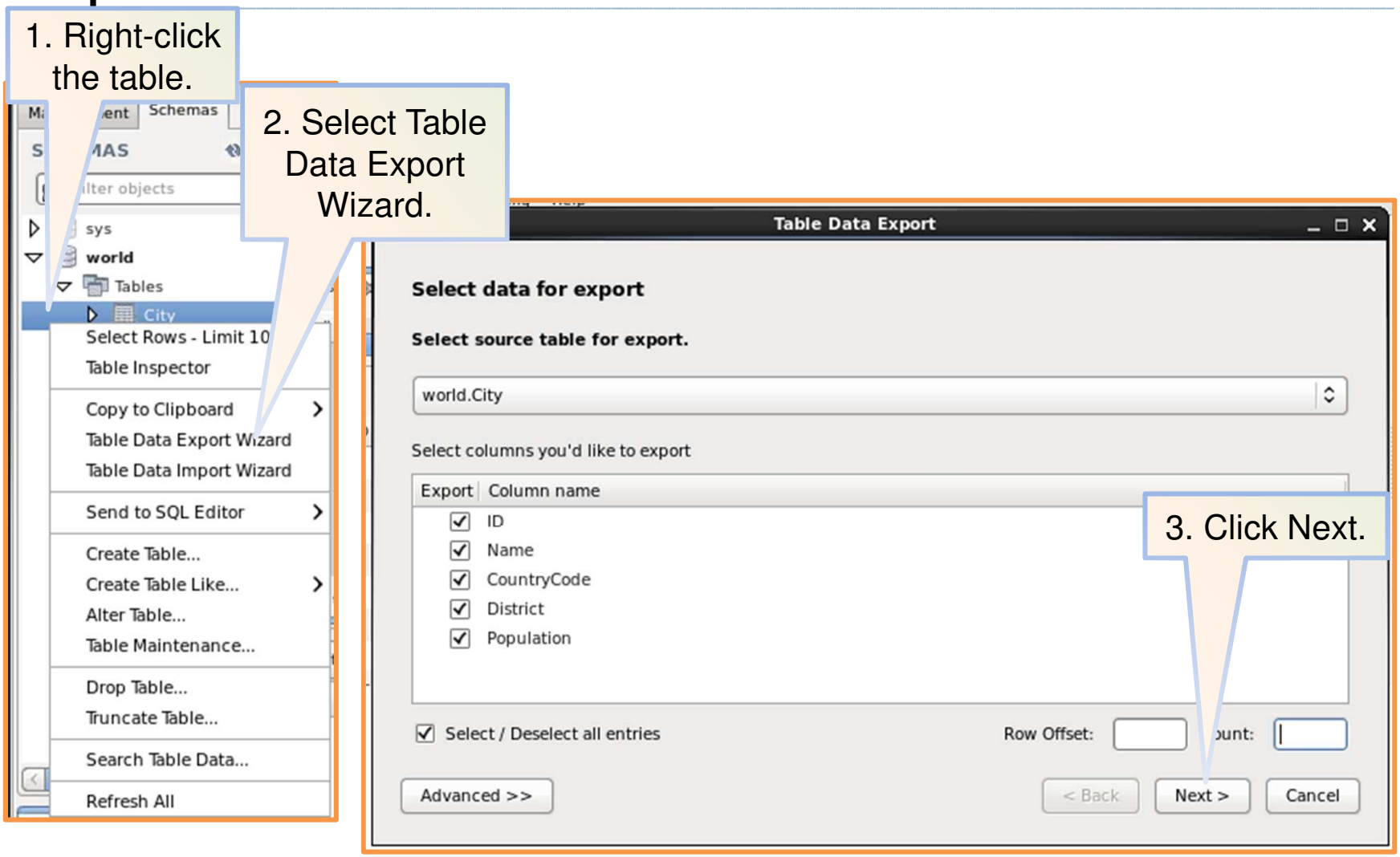


Contents of the CSV File after Exporting the Result Set of SELECT * FROM City;

- Contents of the output file:

```
•ID,Name,CountryCode,District,Population
• 1,Kabul,AFG,Kabol,1780000
• 2,Qandahar,AFG,Qandahar,237500
• 3,Herat,AFG,Herat,186800
• 4,Mazar-e-Sharif,AFG,Balkh,127800
• …
•995,Probolinggo,IDN,"East Java",120770
•996,Cilegon,IDN,"West Java",117000
•997,Cianjur,IDN,"West Java",114300
•998,Ciparay,IDN,"West Java",111500
•999,Lhokseumawe,IDN,Aceh,109600
•1000,Taman,IDN,"East Java",107000
```

Starting the MySQL Workbench Table Data Export Wizard



Selecting Options for the MySQL Workbench Table Data Export Wizard

- The Wizard exports data in CSV or JSON. Select options for the CSV format export.

Contents of the CSV File from the MySQL Workbench Table Data Export Wizard

- Contents of the output file:

```
"ID","Name","CountryCode","District","Population"
1,"Kabul","AFG","Kabol",1780000
2,"Qandahar","AFG","Qandahar",237500
3,"Herat","AFG","Herat",186800
4,"Mazar-e-Sharif","AFG","Balkh",127800
…
4074,"Gaza","PSE","Gaza",353632
4075,"Khan Yunis","PSE","Khan Yunis",123175
4076,"Hebron","PSE","Hebron",119401
4077,"Jabaliya","PSE","North Gaza",113901
4078,"Nablus","PSE","Nablus",100231
4079,"Rafah","PSE","Rafah",92020
```

Importing Data from a Text File

- You can import a data file generated by MySQL or some other application into a MySQL table. You should know the following characteristics of the data file:
  - Column value separator
  - Order of the columns
  - Row separator
  - File system where the file resides
  - What characters enclose the values (example: double quotation marks)
  - Whether the file has a header row with the column names
  - If you need certain privileges to access the file
  - How you want to process any rows that have a unique key column that matches an existing value in the table

Scenarios for Importing Data from a Text File

- A few of the possible scenarios for importing data from a text file include:
  - Populating a newly created, empty table
  - Adding data to an existing table
  - Updating data in an existing table
- When importing data into an existing table that has a primary key or unique key, rows that have the same value as the primary key or unique key cannot be added to the table. You can specify:
  - The data being imported replaces the row in the table with the same primary key or unique key.
  - The data being imported is ignored and the row in the table with the matching key is left unchanged.

Importing with the LOAD DATA INFILE Statement

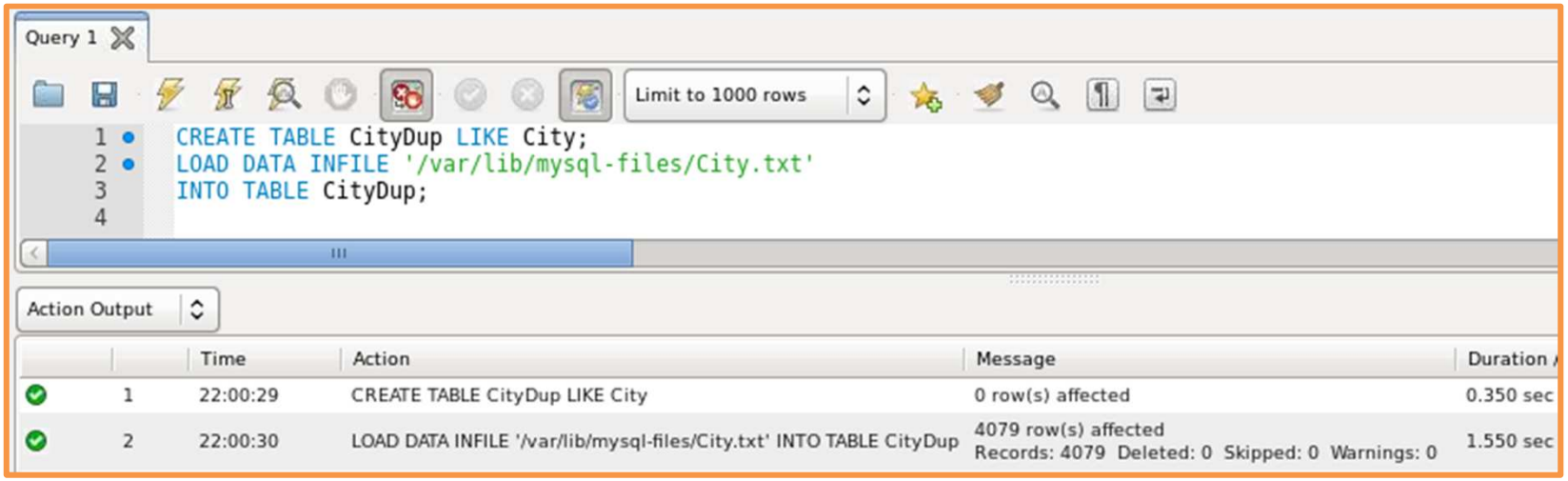
- **LOAD DATA INFILE** is the opposite of **SELECT…INTO OUTFILE**.
  - However, it uses similar clauses and format specifiers.
- Syntax:

```
LOAD DATA INFILE '/path/filename'
[REPLACE | IGNORE]
INTO TABLE table
[FIELDS TERMINATED BY 'delimiter'
[OPTIONALLY] ENCLOSED BY 'character'
LINES TERMINATED BY 'terminator']
[IGNORE number LINES]
[column_list];
```

Importing with LOAD DATA INFILE: File Control

- MySQL file assumptions and defaults:
  - File resides on the server host.
  - Fields are terminated by '\t'.
  - Each input line contains a value for each column in the table.
- Optional clauses give you more control:
  - Control of duplicate records/rows
  - Data file format specification (field and line terminators)
  - Skipping data file rows
  - Loading specific table columns

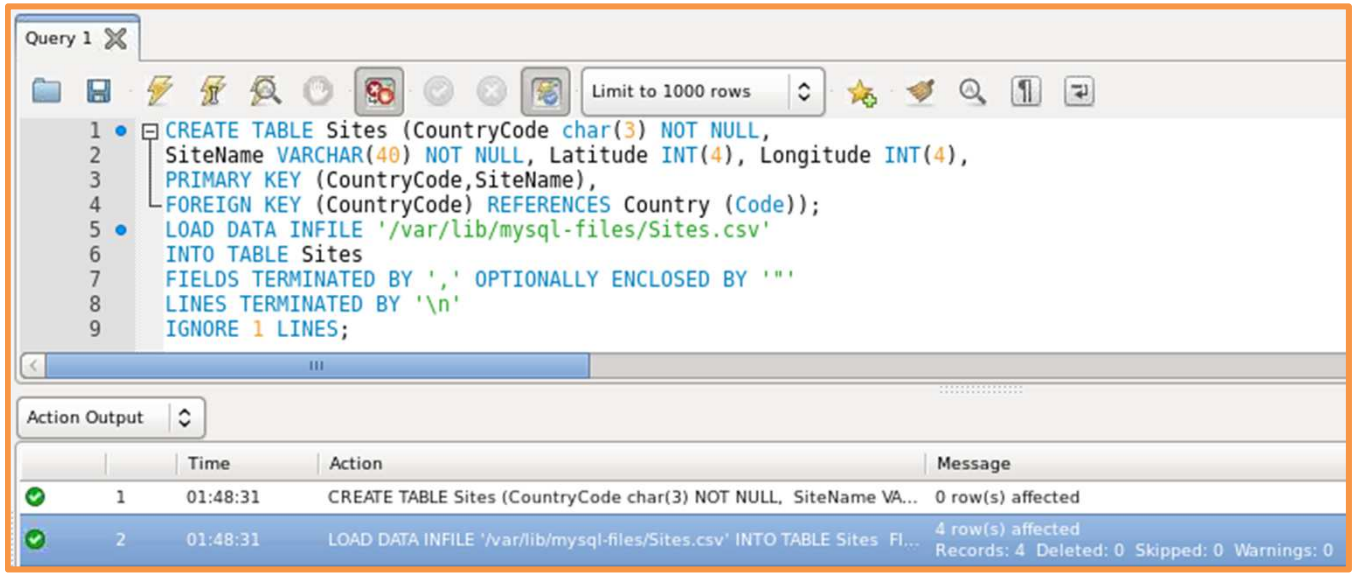
Example of LOAD DATA INFILE into an Empty Table



- Contents of the first few lines of the input file:

```
1               Kabul                   AFG                     Kabol                   1780000
2               Qandahar                AFG                     Qandahar                237500
3               Herat                   AFG                     Herat                   186800
4               Mazar-e-Sharif                  AFG                     Balkh
    127800
5               Amsterdam                       NLD                     Noord-Holland
                731200
```

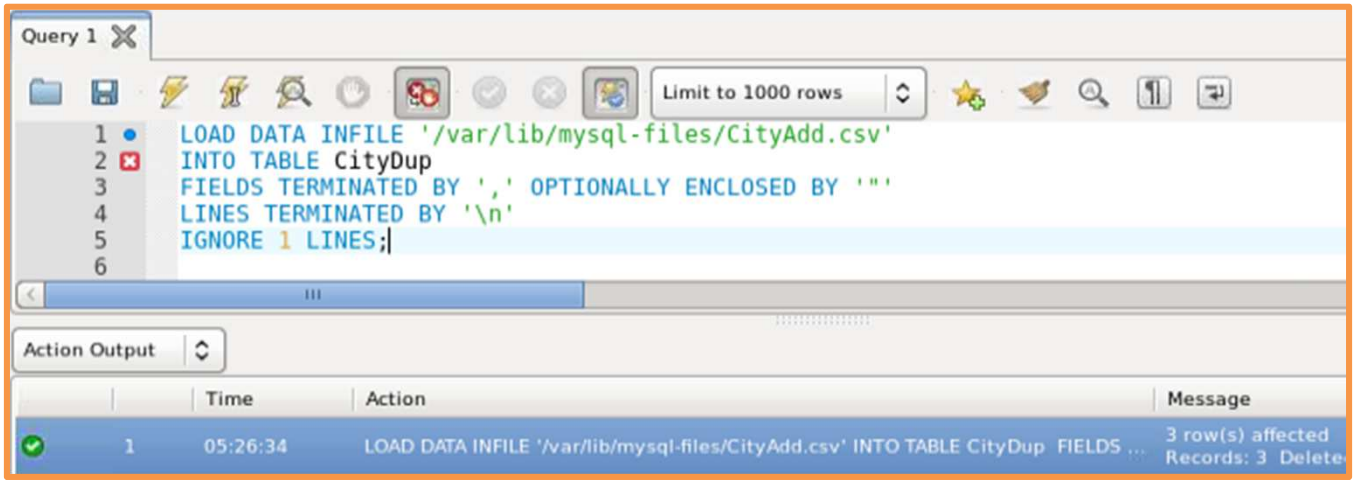
Example of LOAD DATA INFILE into an Empty Table from a CSV Formatted File



- Contents of the input file:

```
"CountryCode","SiteName","Latitude","Longitude"
"FRA","Eiffel Tower",49,2
"USA","Grand Canyon",36,-114
"IND","Taj Mahal",27,78
"AUS","Sydney Opera House",-29,167
```

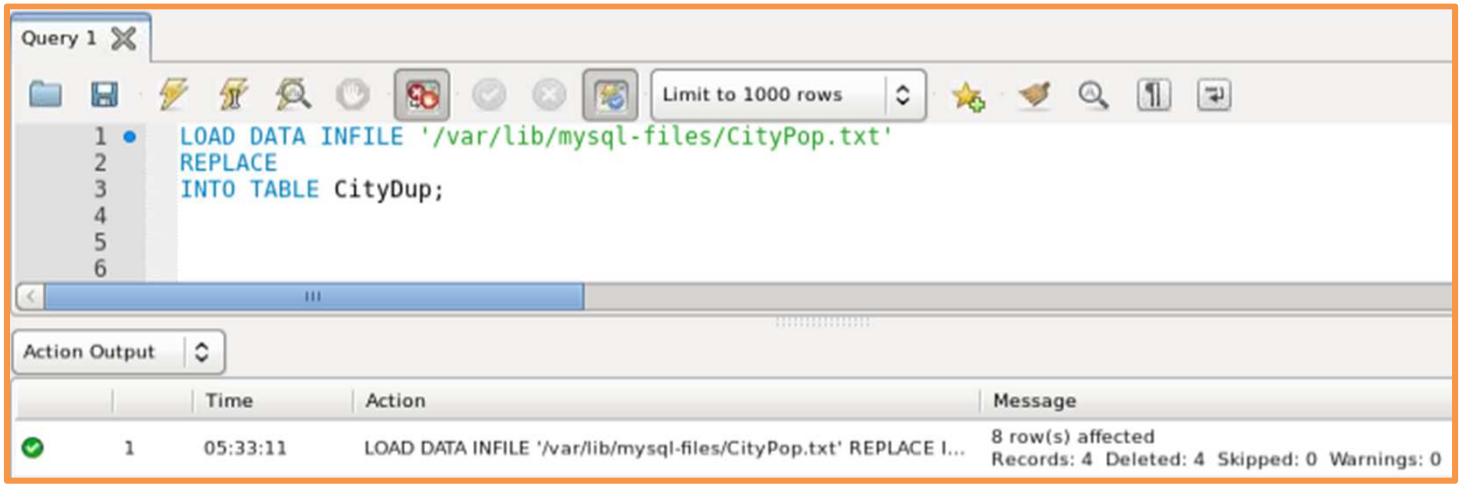
Example of LOAD DATA INFILE to Add Data to an Existing Table



- Contents of the input file:

```
•"ID","Name","CountryCode","District","Population"
•4080,"Juneau","USA","Alaska",32660
•4081,"Dover","USA","Delaware",37366
•4082,"Helena","USA","Montana",29596
```

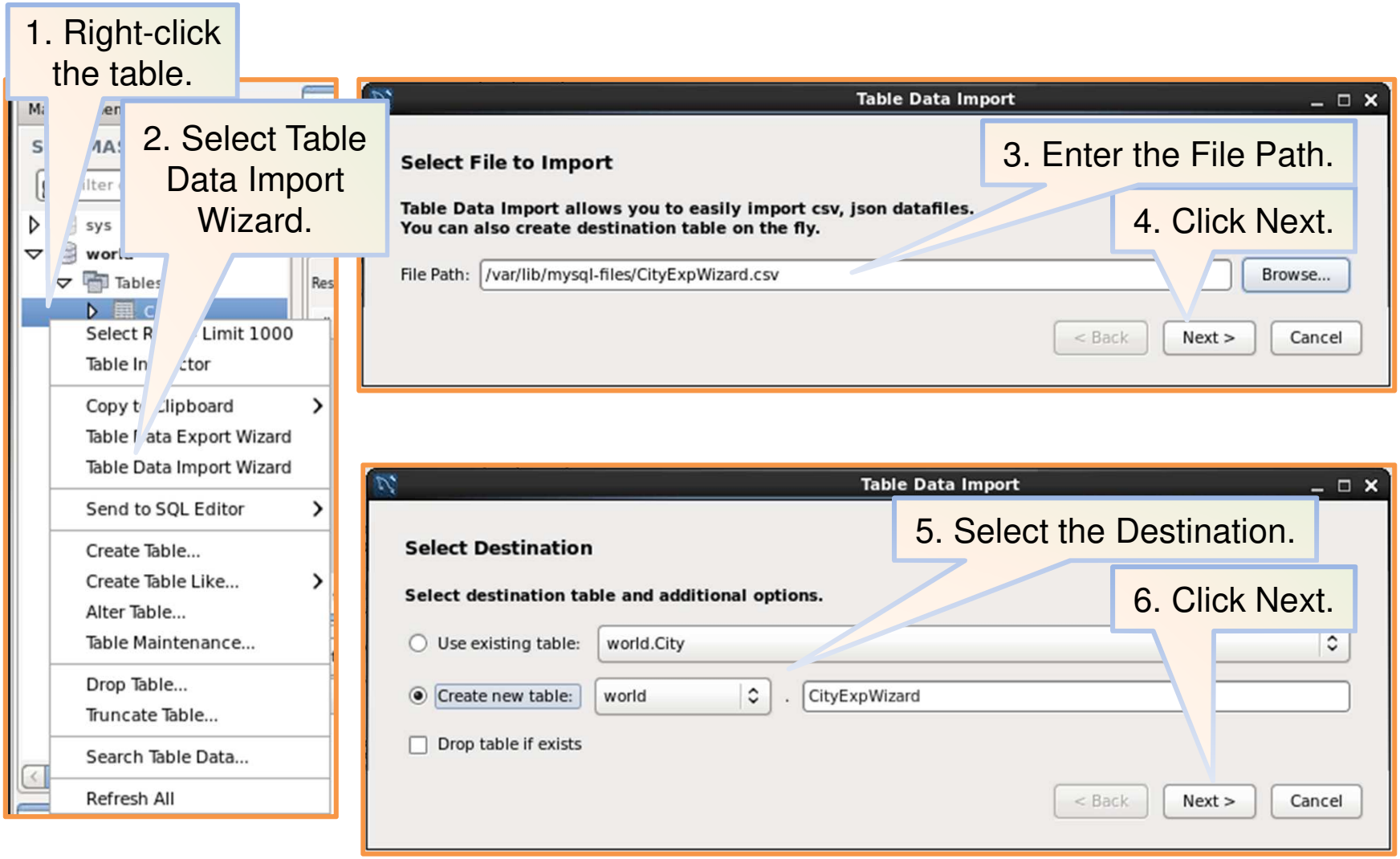
Example of LOAD DATA INFILE to Update Data in an Existing Table



- Contents of the input file:

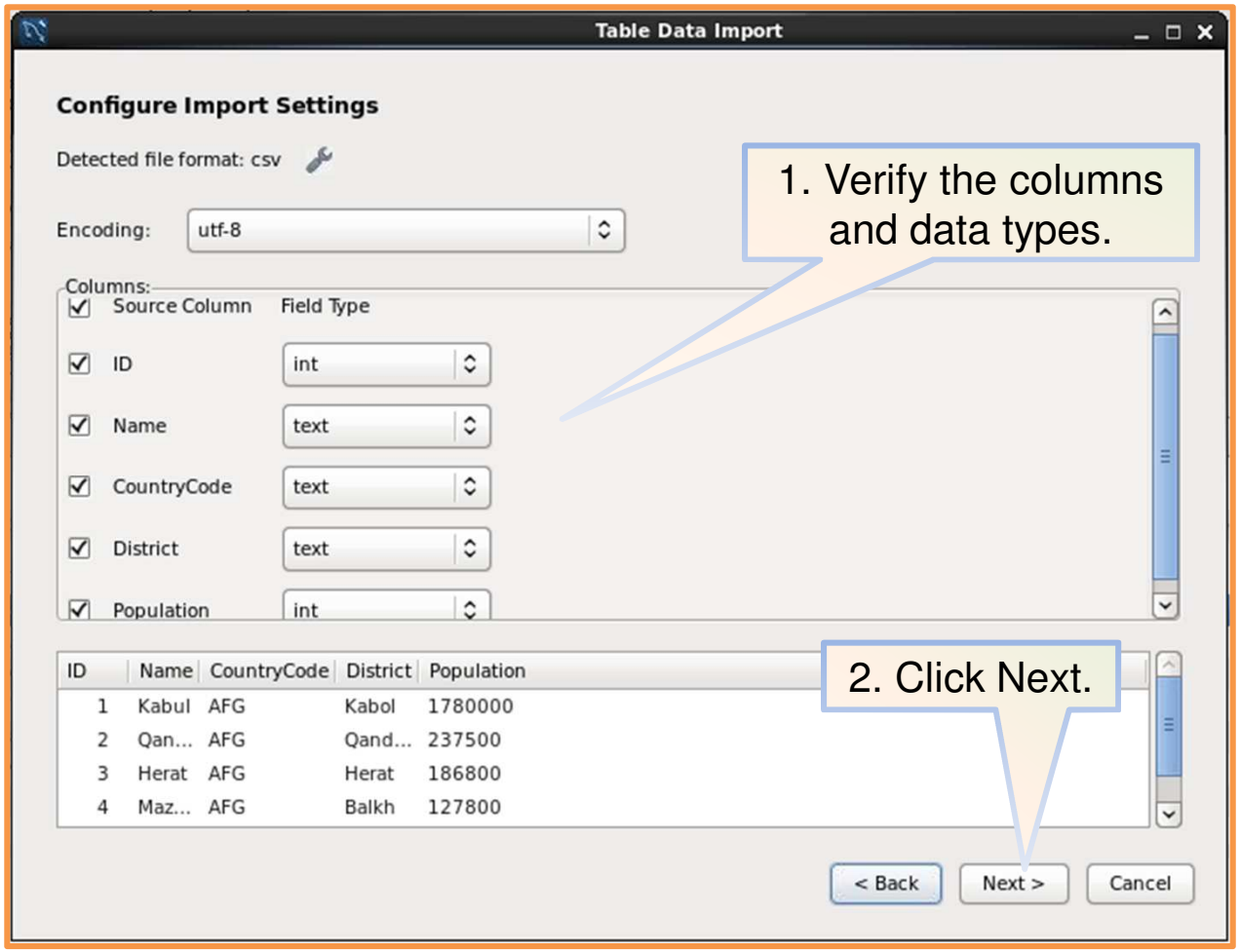
```
1  Kabul  AFG    Kabol  3678000
2  Qandahar       AFG    Qandahar       491500
3  Herat   AFG   Herat  436300
4  Mazar-e-Sharif       AFG    Balkh  693000
```

Starting the MySQL Workbench Table Data Import Wizard



Selecting Options for the MySQL Workbench Table Data Import Wizard

- The Wizard analyzes the file.



Troubleshooting Messages

- MySQL provides error and warning messages.
- Error message example:

```
SELECT Name FROM City LIMIT;
ERROR 1064 (42000): You have an error in your SQL
syntax; check the manual that corresponds to your MySQL
server version for the right syntax to use near '' at
line 1
```

SQLSTATE

MySQL error code

error message

- This error is due to the missing **`row_count`** value for the **`LIMIT`** clause.

Troubleshooting Message Options

- The following statements control display of errors and warnings:
  - **`SHOW WARNINGS:`** Displays any error, warning, and informational messages generated by the last statement
  - **`SHOW ERRORS:`** Displays error messages only
- Refer to the error code definitions in the *MySQL Reference Manual*.
- After you understand the cause of the error or warning, make any needed changes and run the SQL statement again.

Index

Index Types

- **Nonunique:** Values can appear multiple times in the index.
  - This is the default index type.
- **Unique:** Values must be unique (or `NULL`).
  - Primary keys are unique indexes and may not contain `NULL`.
- **Fulltext:** Values are character string data, and the index supports full text searching.
- **Spatial:** Values are spatial data types such as `GEOMETRY`, `POINT`, or `POLYGON`.

MySQL Enterprise Monitor Query Analyzer

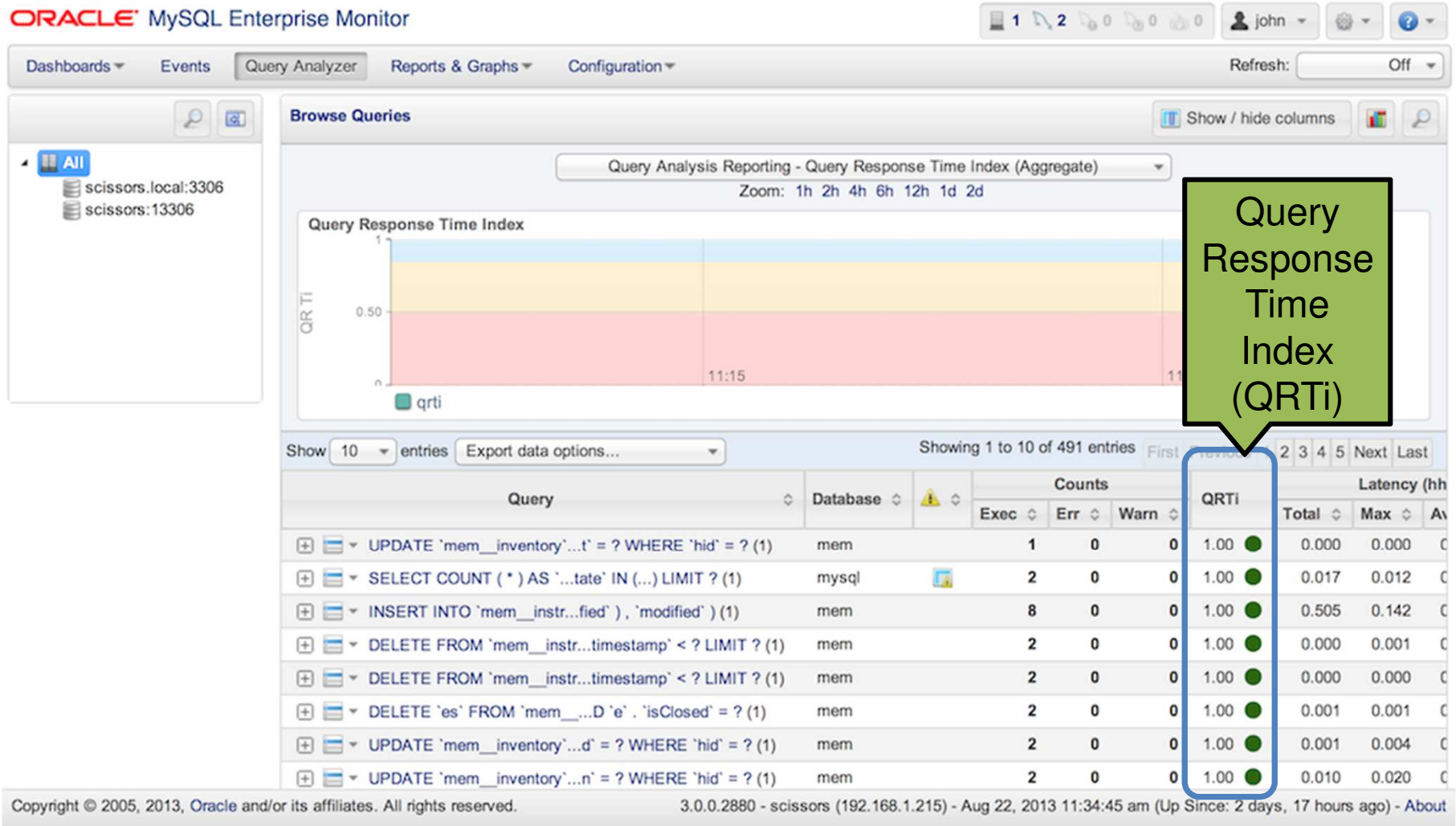
- Monitors SQL statements executed on the MySQL server:
  - Nature of query
  - Number of executions
  - Execution times
- Extracts this information from the Performance Schema using the MySQL Enterprise Monitor Agent by default
  - Can configure client applications to route database requests via MySQL Proxy and MySQL Aggregator
  - Can install a Connector plugin to send this information directly to the MySQL Enterprise Monitor Service Manager
- Normalizes queries for easier reporting
  - Combines similar queries with different literal values
- Enables you to drill into each query for detailed information

Query Response Time Index

- MySQL Enterprise Monitor Query Analyzer generates a Query Response Time Index (QRTi) for each query.
  - Enables an “at-a-glance” indicator of query performance

Status	Default time value	Assigned Value	Meaning
Optimum	100ms	1.0 (100%)	All instances of the query were within the acceptable response time.
Acceptable	4 * Optimum: 100ms to 400ms	0.5 (50%)	Query execution time was less than 400 ms (the default acceptable response time).
Unacceptable	> Acceptable	0 (0%)	All instances of the query exceeded the acceptable response time.

Query Analyzer User Interface



Topics

- Identifying Slow Queries
- `EXPLAIN` Statement
- Working with Indexes
- Index Statistics

Creating Indexes to Improve Query Performance

- Enables efficient access to specific rows
  - A query that searches for a value in an indexed field can immediately find rows containing that value.
    - A *seek*, which is efficient
  - A query that searches for a value in an unindexed field must read all rows to find rows containing that value.
    - A *scan*, which is inefficient
- Supports the following operations:
  - Direct value matching
    - Example: Find words such as `xenolith`.
  - Existence checking
    - Example: Determine that the word `xzzq` does not exist.
  - Range scans
    - Example: Find all words that begin with `xy`.

Creating and Dropping Indexes on Existing Tables

- To change a table's primary key:

```
ALTER TABLE table DROP PRIMARY KEY,
   ADD PRIMARY KEY(col1, col2);
```

- To add a unique key:

```
ALTER TABLE table ADD UNIQUE(col3);
CREATE UNIQUE INDEX index2 ON table(col4);
```

- To add an ordered (nonunique) index with a name:

```
CREATE INDEX index3 ON table(col5);
```

- To drop a nonprimary key by name:

```
ALTER TABLE table DROP INDEX indexname;
DROP INDEX indexname ON table;
```

Displaying Indexes with SHOW INDEXES FROM

```
mysql> SHOW INDEXES FROM departments\G
*************************** 1. row ***************************
        Table: departments
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: dept_no
    Collation: A
  Cardinality: 9
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
*************************** 2. row ***************************
        Table: departments
   Non_unique: 0
     Key_name: dept_name
 Seq_in_index: 1
  Column_name: dept_name
    Collation: A
  Cardinality: 9
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
2 rows in set (0.00 sec)
```

Grazie